PyGraphistry Documentation

Release 0.56.0+28.gf08c458

Graphistry, Inc.

May 28, 2026

PYGRAPHISTRY DOCUMENTATION

https://github.com/graphistry/pygraphistry/actions?query=workflow%3ACodeQL
https://pygraphistry.readthedocs.io/en/latest/ https://pypi.org/project/graphistry/
https://pypi.org/project/graphistry/ https://pypi.org/project/graphistry/

https://status.graphistry.com/ https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g https://twitter.com/graphistry

PyGraphistry is an open source Python library for data scientists and developers to leverage the power of graph visualization, analytics, AI, including with native GPU acceleration:

- https://pygraphistry.readthedocs.io/en/latest/10min.html Quickly ingest & prepare data in many formats, shapes, and scales as graphs. Use tools like Pandas, Spark, https://www.rapids.ai, and https://arrow.apache.org/.
- https://pygraphistry.readthedocs.io/en/latest/plugins.html Connect to graph databases, data platforms, Python tools, and more.

Category	Connector Tutorials
Data Platforms	https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/databricks_pyspark/graphistry-notebook-dashboard.html https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/splunk/ https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/sql/postgres.html
SQL & Logs	https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/microsoft/kusto/graphistry_ADX https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/spanner/google_spanner_finance_
Graph Datab	https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/neo4j/official/graphistry_bolt_tu https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/neptune/neptune_cypher_viz_us https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/tigergraph/tigergraph_pygraphist https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/arango/arango_tutorial.html https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/memgraph/visualizing_iam_data
Python Tools & Libraries	https://pygraphistry.readthedocs.io/en/latest/demos/upload_csv_miniapp.html https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/gpu_rapids/part_i_cpu_pandas. https://pygraphistry.readthedocs.io/en/latest/performance.html https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/gpu_rapids/cugraph.html https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/networkx/networkx.html https://pygraphistry.readthedocs.io/en/latest/demos/demos_databases_apis/graphviz/graphviz.html

https://pygraphistry.readthedocs.io/en/latest/notebooks/plugins.connectors.html

- https://www.graphistry.com/get-started Prototype from notebooks like Jupyter and Databricks using local CPUs & GPUs, and then power production dashboards & pipelines with Graphistry Hub and your own self-hosted servers.
- https://pygraphistry.readthedocs.io/en/latest/gfql/index.html Use GFQL, the first fully vectorized dataframe-native graph query language with an open-source GPU runtime, to ask relationship questions that are difficult for tabular tools without requiring a database. It supports friendly Cypher syntax and declarative graph semantics through `g.gfql("MATCH ...")`, with the same execution model available on the current bound graph or remotely via `g.gfql_remote([...])`.
- https://pygraphistry.readthedocs.io/en/latest/gfql/combo.html# Call streamlined graph ML & AI methods to benefit from clustering, UMAP embeddings, graph neural networks, automatic feature engineering, and more.

- https://pygraphistry.readthedocs.io/en/latest/visualization/10min.html# In just a few minutes, create stunning interactive visualizations with millions of edges and many point-and-click built-ins like drilldowns, timebars, and filtering. When ready, customize with Python, JavaScript, and REST APIs.
- https://pygraphistry.readthedocs.io/en/latest/performance.html CPU-mode ingestion and wrangling is fast due to native use of Apache Arrow and columnar analytics, and the optional RAPIDS-based GPU mode delivers 100X+ speedups.

From global 10 banks, manufacturers, news agencies, and government agencies, to startups, game companies, scientists, biotechs, and NGOs, many teams are tackling their graph workloads with Graphistry.

CHAPTER

ONE

AI ASSISTANT INTEGRATION

For LLM coding assistants (Claude Code, Cursor, Codex, etc.), install the official https://github.com/graphistry/graphistry-skills package for better PyGraphistry code generation:

```
npx skills add graphistry/graphistry-skills
```

Skills improve AI success rates from ~50% to ~90% on PyGraphistry tasks by providing context-aware guidance for graph ETL, visualization, GFQL queries, and AI workflows.

CHAPTER

TWO

GALLERY

The https://pygraphistry.readthedocs.io/en/latest/demos/for_analysis.html shares many more live visualizations, demos, and integration examples

CHAPTER

THREE

INSTALL

Common configurations:

- **Minimal core**

  Includes: The GFQL dataframe-native graph query language, built-in layouts, Graphistry visualization server client

  ```
  pip install graphistry
  ```

  Does not include `graphistry[ai]`, plugins

- **No dependencies and user-level**

  ```
  pip install --no-deps --user graphistry
  ```

- **GPU acceleration** - Optional

  Local GPU: Install https://www.rapids.ai and/or deploy a GPU-ready https://www.graphistry.com/get-started

  Remote GPU: Use the https://www.graphistry.com/blog/graphistry-2-41-3.

For further options, see the https://pygraphistry.readthedocs.io/en/latest/install/index.html

CHAPTER

FOUR

VISUALIZATION QUICKSTART

Quickly go from raw data to a styled and interactive Graphistry graph visualization:

```
import graphistry
import pandas as pd

# Raw data as Pandas CPU dataframes, cuDF GPU dataframes, Spark, ...
df = pd.DataFrame({
    'src': ['Alice', 'Bob', 'Carol'],
    'dst': ['Bob', 'Carol', 'Alice'],
    'friendship': [0.3, 0.95, 0.8]
})

# Bind
g1 = graphistry.edges(df, 'src', 'dst')

# Override styling defaults
g1_styled = g1.encode_edge_color('friendship', ['blue', 'red'], as_continuous=True)

# Connect: Free GPU accounts and self-hosting @ graphistry.com/get-started
graphistry.register(api=3, username='your_username', password='your_password')

# Upload for GPU server visualization session
g1_styled.plot()
```

Explore https://pygraphistry.readthedocs.io/en/latest/visualization/10min.html for more visualization examples and options

CHAPTER

FIVE

PYGRAPHISTRY[AI] & GFQL QUICKSTART - CPU & GPU

CPU graph pipeline combining graph ML, AI, mining, and visualization:

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

# Graph analytics
g2 = g1.compute_igraph('pagerank')
assert 'pagerank' in g2._nodes.columns

# Graph ML/AI
g3 = g2.umap()
assert ('x' in g3._nodes.columns) and ('y' in g3._nodes.columns)

# Graph querying with GFQL
g4 = g3.gfql([
    n(query='pagerank > 0.1'), e_forward(), n(query='pagerank > 0.1')
])
assert (g4._nodes.pagerank > 0.1).all()

# Upload for GPU server visualization session
g4.plot()
```

The **automatic GPU modes** require almost no code changes:

```
import cudf
from graphistry import n, e, e_forward, e_reverse

# Modified -- Rebind data as a GPU dataframe and swap in a GPU plugin call
g1_gpu = g1.edges(cudf.from_pandas(df))
g2 = g1_gpu.compute_cugraph('pagerank')

# Unmodified -- Automatic GPU mode for all ML, AI, GFQL queries, & visualization APIs
g3 = g2.umap()
g4 = g3.gfql([
    n(query='pagerank > 0.1'), e_forward(), n(query='pagerank > 0.1')
])
g4.plot()
```

Explore https://pygraphistry.readthedocs.io/en/latest/10min.html for a wider variety of graph processing.

CHAPTER

SIX

PYGRAPHISTRY DOCUMENTATION

- https://pygraphistry.readthedocs.io/en/latest/
- 10 Minutes to: https://pygraphistry.readthedocs.io/en/latest/10min.html, https://pygraphistry.readthedocs.io/en/latest/visualization/10min.html, https://pygraphistry.readthedocs.io/en/latest/gfql/about.html
- Get started: https://pygraphistry.readthedocs.io/en/latest/install/index.html, https://hub.graphistry.com/docs/ui/index/, https://pygraphistry.readthedocs.io/en/latest/demos/for_analysis.html
- Performance: https://pygraphistry.readthedocs.io/en/latest/performance.html & https://pygraphistry.readthedocs.io/en/latest/gfql/performance.html
- API References
  - https://pygraphistry.readthedocs.io/en/latest/api/index.html: https://pygraphistry.readthedocs.io/en/latest/visual https://pygraphistry.readthedocs.io/en/latest/cheatsheet.html
  - https://pygraphistry.readthedocs.io/en/latest/gfql/index.html: https://pygraphistry.readthedocs.io/en/latest/gfql/c and https://pygraphistry.readthedocs.io/en/latest/gfql/predicates/quick.html
  - https://pygraphistry.readthedocs.io/en/latest/plugins.html: Databricks, Splunk, Neptune, Neo4j, RAPIDS, and more
  - Web: https://hub.graphistry.com/docs/api/1/rest/url/#urloptions, https://hub.graphistry.com/static/js-docs/index.html?path=/docs/introduction--docs, https://hub.graphistry.com/docs/api/1/rest/auth/

CHAPTER

SEVEN

GRAPHISTRY ECOSYSTEM

- **Graphistry server:**
  - Launch - https://www.graphistry.com/get-started
  - Self-hosting: https://github.com/graphistry/graphistry-cli & https://github.com/graphistry/graphistry-helm
- **Graphistry client APIs:**
  - Web: https://hub.graphistry.com/docs/api/1/rest/url/#urloptions, https://hub.graphistry.com/static/js-docs/index.html?path=/docs/introduction--docs, https://hub.graphistry.com/docs/api/1/rest/auth/
  - https://pygraphistry.readthedocs.io/en/latest/index.html
  - https://hub.graphistry.com/docs/powerbi/pbi/
- **Additional projects**:
  - https://louie.ai/: GenAI-native notebooks & dashboards to talk to your databases & Graphistry
  - https://github.com/graphistry/graph-app-kit: Streamlit Python dashboards with batteries-include graph packages
  - https://chat.openai.com/chat: Automatic GPU feature engineering

CHAPTER

EIGHT

COMMUNITY AND SUPPORT

- https://www.graphistry.com/blog for tutorials, case studies, and updates
- https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g: Join the Graphistry Community Slack for discussions and support
- https://twitter.com/graphistry & https://www.linkedin.com/company/graphistry: Follow for updates
- https://github.com/graphistry/pygraphistry/issues open source support
- https://graphistry.zendesk.com/ dedicated enterprise support

CHAPTER

NINE

CONTRIBUTE

See https://pygraphistry.readthedocs.io/en/latest/CONTRIBUTING.html and https://pygraphistry.readthedocs.io/en/latest/DEVELOP.html for participating in PyGraphistry development, or reach out to our team

CHAPTER

TEN

INDICES AND TABLES

10.1 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.

10.1.1 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.

10.1.2 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.

10.1.3 Installation

10.1.3.1 Install PyGraphistry

```
pip install graphistry
```

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

10.1.3.2 Install cuDF GPU DataFrames (Optional)

For GPU acceleration with DataFrames, install **cuDF** via the https://rapids.ai/.

10.1.3.3 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 https://www.graphistry.com/get-started, 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.

10.1.4 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
...
```

10.1.5 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
```

Example visualization:

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

10.1.6 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.

10.1.7 Adding Visual Encodings

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

10.1.7.1 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',
```

(continues on next page)

(continued from previous page)

```
        'OtherVuln': 'blue',
    },
    default_mapping='gray')

g2.plot()
```

Example visualization:

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

10.1.7.2 Advanced: Collections for layered highlights

Use collections when you want GFQL-driven subsets (nodes, edges, or subgraphs) to override base encodings. This is useful for overlays like alerts or critical paths that take precedence over your normal color rules.

```
from graphistry import collection_set, n

collections = [
    collection_set(
        expr=n({"vip": True}),
        name="VIP",
        node_color="#FF8800",
    )
]
g.collections(collections=collections, show_collections=True).plot()
```

See the *Collections tutorial notebook* and *GFQL docs* for full details.

10.1.8 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.

10.1.9 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()
```

Example visualization:

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

10.1.10 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()
```

Example visualization:

10.1.11 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()
```

Example visualization:

10.1.12 Static Image Export

For documentation, reports, or non-interactive use cases, export to static images with `plot_static()`:

```
# Auto-displays inline in Jupyter, returns SVG object (.data for bytes)
g.plot_static()

# Save to file
g.plot_static(format='png', path='graph.png')

# With styling
g.plot_static(
```

(continues on next page)

(continued from previous page)

```
    graph_attr={'rankdir': 'LR', 'bgcolor': 'white'},
    node_attr={'style': 'filled', 'fillcolor': 'lightblue'}
)
```

Works with any layout source (UMAP, ring, graphviz, manual x/y). For DOT or Mermaid text output:

```
dot_text = g.plot_static(engine='graphviz-dot')
mermaid_text = g.plot_static(engine='mermaid-code')
```

See the static rendering tutorial for styling options and complete examples.

10.1.13 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()
```

Example visualization:

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

g2.plot()
```

Example visualization:

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

10.1.15 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()
```

Example visualization:

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

10.1.16 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.

10.1.17 Rendering Options

10.1.17.1 Inline Rendering

In Jupyter notebooks, you can render the visualization inline.

```
g.plot()
```

Example visualization:

10.1.17.2 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}")
```

Example visualization:

10.1.18 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*

10.1.19 External Resources

- https://hub.graphistry.com/docs/ui/index/
- https://github.com/graphistry/graphistry-js: Node, React, and vanilla JS clients
- https://hub.graphistry.com/docs/api/: Work from any language
- https://hub.graphistry.com/docs/api/1/rest/url/#urloptions: Control visualizations via URL parameters`

Happy graphing!

10.2 Install

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

10.2.1 Installation Guide - Quick Start

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

10.2.1.1 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

10.2.1.2 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__)
```

10.2.1.3 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 https://www.graphistry.com/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!

10.2.2 Installation Guide - Extended

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

10.2.2.1 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.

10.2.2.2 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 https://rapids.ai/start.html.

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]
```

10.2.2.3 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 https://hub.graphistry.com/.
  - **Self-Hosted Server**: Set up your own Graphistry server by following the deployment instructions in the https://github.com/graphistry/graphistry-cli.

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 *ruff*, *pytest*, *sphinx*, etc.

10.2.2.4 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!

10.2.3 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.

10.2.3.1 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.

10.2.3.2 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 https://www.graphistry.com/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 https://aws.amazon.com/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 https://azuremarketplace.microsoft.com/ 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 https://github.com/graphistry/graphistry-cli.
- 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 https://github.com/graphistry/graphistry-cli for Docker-Compose setup instructions.
- Execute the provided Docker-Compose files to launch your Graphistry server locally or on a server.

10.2.3.3 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!

10.3 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()*. For multi-tenant applications or concurrent processing, use isolated client instances (*graphistry.client()*) to safely serve multiple users.

You can set access control policies on all of your uploaded data via *graphistry.privacy()*. Read on for more on both.

10.3.1 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.

10.3.1.1 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.

10.3.1.2 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",
```

(continues on next page)

(continued from previous page)

```
    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"
)
```

10.3.1.3 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.

10.3.1.4 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*).

10.3.1.5 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"
)
```

10.3.1.6 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.

10.3.1.7 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.

10.3.2 Concurrency & Multi-tenancy

To safely use pygraphistry in concurrent and multitenant settings, use client objects. Use of top-level calls like `register()` and `plot()` are unsafe in these settings as they use global variables.

Client objects automatically isolate session state like graphistry server and database tokens. Each client and derived plottable objects are safe for use within a single concurrency context like a thread or event loop, and you can have multiple in different threads and event loops.

10.3.2.1 Creating Client Objects

```
import graphistry

# Create independent client instances
alice_g = graphistry.client()
alice_g.register(api=3, username='alice', password='pw')

bob_g = graphistry.client()
bob_g.register(api=3, username='bob', password='pw')
```

10.3.2.2 Multi-tenant Example

Different users can work with their own isolated sessions:

```
import graphistry

# Alice's client with her credentials and settings
alice_g = graphistry.client()
alice_g.register(api=3, username='alice', password='alice_pw')
alice_g.privacy(mode='public')

# Bob's client with his credentials and settings
```

(continues on next page)

(continued from previous page)

```
bob_g = graphistry.client()
bob_g.register(api=3, username='bob', password='bob_pw')
bob_g.privacy(mode='org')

# Each client creates isolated plottables
alice_plot = alice_g.edges(alice_data).plot(render=False)
bob_plot = bob_g.edges(bob_data).plot(render=False)
```

10.3.2.3 Multiple Servers Example

Connect to different Graphistry servers (e.g., staging vs production):

```
import graphistry

# Production server client
prod_g = graphistry.client()
prod_g.register(
    api=3,
    server='prod.graphistry.com',
    username='user',
    password='pw'
)

# Staging server client
staging_g = graphistry.client()
staging_g.register(
    api=3,
    server='staging.graphistry.com',
    username='user',
    password='pw'
)

# Use different servers for different purposes
prod_url = prod_g.edges(production_data).plot(render=False)
staging_url = staging_g.edges(test_data).plot(render=False)
```

10.3.2.4 Multi-threaded Example

Each thread should use its own client instance:

```
import graphistry
import threading

def process_user_data(username, password, data):
    # Each thread creates its own client
    g = graphistry.client()
    g.register(api=3, username=username, password=password)

    # Process and plot data
    url = g.edges(data).plot(render=False)
```

(continues on next page)

(continued from previous page)

```
    return url

# Launch threads with separate clients
threads = []
for user_info in users:
    t = threading.Thread(
        target=process_user_data,
        args=(user_info['username'], user_info['password'], user_info['data'])
    )
    threads.append(t)
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()
```

10.3.2.5 Transferring Plottables Between Clients

You can transfer ownership of a plottable from one client to another:

```
import graphistry

# Create plottable with Alice's client
alice_g = graphistry.client()
alice_g.register(api=3, username='alice', password='alice_pw')
g = alice_g.edges(data)

# Transfer to Bob's client
bob_g = graphistry.client()
bob_g.register(api=3, username='bob', password='bob_pw')
g_bob = bob_g.set_client_for(g)

# Now the plottable uses Bob's credentials and settings
url = g_bob.plot(render=False)
```

10.3.2.6 Performance Considerations

- Creating a client requires an authentication round trip, unless you set the JWT token manually
- Each `plot()` call may refresh authentication tokens

10.3.3 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 https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/sharing_tutorial.ipynb

10.3.3.1 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*.

10.3.3.2 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.

10.3.3.3 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)
```

(continues on next page)

(continued from previous page)

```
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.

10.3.3.4 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.

10.3.3.5 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.

10.3.3.6 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}")
```

10.3.3.7 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.

10.3.3.8 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.

10.3.3.9 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()*)

10.3.3.10 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

10.3.3.11 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.

10.4 Visualize

We recommend getting started with *10 Minutes to PyGraphistry*, *10 Minutes to Graphistry Visualization*, and the *layout guide*. For advanced, subset-based coloring, see *Layout settings* and the *Collections tutorial notebook*.

For static image export (documentation, reports), see the *static rendering tutorial*.

See also:

10.4.1 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.

10.4.1.1 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 https://www.graphistry.com/get-started).
- **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.

10.4.1.2 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()
  ```

  Example visualization:

- **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.

10.4.1.3 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

10.4.1.4 Static Graphviz render (for docs/notebooks)

When you need a quick static image without an interactive client, render directly with Graphviz. `plot_static` auto-displays in Jupyter and returns an SVG/Image object (use `.data` for raw bytes):

```
# Auto-displays inline in Jupyter notebooks
g.plot_static(format='svg', max_nodes=200, max_edges=400)
```

Example visualization (static):

10.4.1.5 Text-only outputs

Emit DOT or Mermaid DSL for downstream rendering or embedding:

```
dot_text = g.plot_static(engine='graphviz-dot', reuse_layout=True)
mermaid_text = g.plot_static(engine='mermaid-code', reuse_layout=False)
```

Example DOT output:

```
digraph G {
    a -> b;
    b -> c;
    a -> tx1;
}
```

Example Mermaid output:

```
graph LR
  a --> b
  b --> c
  a --> tx1
```

- **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()
```

10.4.1.6 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'}, default_
↪mapping='gray').plot()
```

Example visualization:

Example visualization (static):

- **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()*
- **Collections (advanced coloring)**: Define subsets using GFQL AST helpers and color them consistently:

```
from graphistry import collection_set, n

collections = [
    collection_set(
        expr=n({"subscribed_to_newsletter": True}),
        name="Subscribers",
```

(continues on next page)

(continued from previous page)

```
        node_color="#32CD32",
    )
]
g.collections(collections=collections, show_collections=True).plot()
```

See *Layout settings* and the *Collections tutorial notebook*. Tip: order matters (earlier collections override later ones) and intersections require set IDs.

- **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
)
```

10.4.1.7 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 https://hub.graphistry.com/docs/api/1/rest/url/#urloptions/.

10.4.1.8 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

Static Image Export

For non-interactive outputs (documentation, reports, presentations), use `plot_static()`:

```
# Quick static SVG (auto-displays in Jupyter)
g.plot_static()

# With styling
g.plot_static(
    graph_attr={'rankdir': 'LR', 'bgcolor': 'white'},
    node_attr={'style': 'filled', 'fillcolor': 'lightblue'}
)

# Save to file
g.plot_static(format='png', path='graph.png')
```

Works with any layout source (UMAP, ring, graphviz, or manual x/y positions).

See the static rendering tutorial for styling options, output formats, and complete examples.

10.4.1.9 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*

10.4.1.10 External Resources

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

- https://www.graphistry.com/get-started
- https://github.com/graphistry/graphistry-js
- https://github.com/graphistry/pygraphistry
- https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g

Happy graphing!

10.4.2 UI Guide

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

- https://hub.graphistry.com/docs/ui/index/: How to navigate the Graphistry UI
- https://hub.graphistry.com/docs/ui/basics/: Drilling down with filters and exclusions
- https://hub.graphistry.com/docs/ui/histograms/: Analyze attributes and manipulate colors, sizes, filters, and more
- https://hub.graphistry.com/docs/ui/tips/: Selections to analyze data and work across tools more easily

10.4.3 Maps & Geographic Visualization

PyGraphistry provides multiple approaches for visualizing geographic data, from simple latitude/longitude coordinates to interactive Kepler.gl maps.

10.4.3.1 Overview

Choose the right approach for your use case:

1. **Latitude/Longitude Bindings** - Automatic server-side geographic layout
2. **Mercator Layout** - Client-side projection for local analysis
3. **Kepler.gl Integration** - Full-featured interactive maps with layers and styling

10.4.3.2 Latitude/Longitude Bindings

The simplest approach: bind your lat/lon columns and let Graphistry handle the rest.

When to use:

- You have latitude/longitude data
- You want automatic geographic visualization
- You don't need custom projections

Basic Usage

```
import graphistry
import pandas as pd

# Nodes with geographic coordinates
cities = pd.DataFrame({
    "city": ["NYC", "LA", "London", "Paris", "Tokyo"],
    "latitude": [40.7128, 34.0522, 51.5074, 48.8566, 35.6762],
    "longitude": [-74.0060, -118.2437, -0.1278, 2.3522, 139.6503]
})

# Edges between cities
flights = pd.DataFrame({
    "origin": ["NYC", "LA", "London", "Paris"],
```

(continues on next page)

(continued from previous page)

```
    "destination": ["LA", "London", "Tokyo", "NYC"]
})

# Bind lat/lon columns
g = (graphistry
     .nodes(cities, "city")
     .edges(flights, "origin", "destination")
     .bind(point_latitude="latitude", point_longitude="longitude")
     .layout_settings(play=0))

g.plot()
```

Graphistry automatically applies server-side map layout when geographic bindings are set.

Default bindings: The default `point_latitude` and `point_longitude` bindings are `"latitude"` and `"longitude"`, so explicit binding is not necessary if your columns use these names.

See Also

- *10 Minutes to Graphistry Visualization*
- API: *graphistry.PlotterBase.PlotterBase.bind()*

10.4.3.3 Mercator Layout

Convert lat/lon to 2D Mercator projection coordinates locally.

When to use:

- You need projected coordinates for local analysis
- You want to export coordinates to other tools
- You need GPU-accelerated projection (cuDF support)

Basic Usage

```
import graphistry
import pandas as pd

cities = pd.DataFrame({
    "id": ["NYC", "LA", "London"],
    "latitude": [40.7128, 34.0522, 51.5074],
    "longitude": [-74.0060, -118.2437, -0.1278]
})

# Apply Mercator projection
g = graphistry.nodes(cities, "id").mercator_layout()

# Coordinates now in "x" and "y" columns
print(g._nodes[["id", "x", "y"]])
```

Custom Column Names

```
cities = pd.DataFrame({
    "id": ["NYC", "LA", "London"],
    "lat": [40.7128, 34.0522, 51.5074],
    "lon": [-74.0060, -118.2437, -0.1278]
})

g = (graphistry
     .nodes(cities, "id")
     .bind(point_latitude="lat", point_longitude="lon")
     .mercator_layout())
```

Scaling Options

Scaled mode (default): Optimized for Graphistry visualization

```
g = g.mercator_layout(scale_for_graphistry=True)  # Default
```

Unscaled mode: Standard Web Mercator (EPSG:3857) with Earth radius ~6,378,137 meters for geographic accuracy

```
g = g.mercator_layout(scale_for_graphistry=False)
```

GPU Acceleration

Mercator layout automatically uses GPU acceleration (CuPy) when available:

```
import cudf
import graphistry

# cuDF DataFrame
cities_gpu = cudf.DataFrame({
    "id": ["NYC", "LA", "London"],
    "latitude": [40.7128, 34.0522, 51.5074],
    "longitude": [-74.0060, -118.2437, -0.1278]
})

# Automatically uses GPU-accelerated projection
g = graphistry.nodes(cities_gpu, "id").mercator_layout()
```

See Also

- API: *Mercator Layout API*

10.4.3.4 Kepler.gl Integration

Full-featured interactive maps with multiple layers, styling, and native Kepler.gl controls.

When to use:

- You want rich interactive map visualizations
- You need fine-grained control over map styling
- You're visualizing geographic regions (countries, states)
- You need multiple map layers with different visualizations

What it provides:

- **Full Kepler.gl passthrough**: Direct access to all native Kepler layers and configurations
- **Graphistry dataset shortcuts**: Simplified dataset creation from Graphistry nodes, edges, and geographic data
- **Type-safe configuration**: Use `KeplerDataset`, `KeplerLayer`, and `KeplerEncoding` classes

Quick Start

```
import graphistry
from graphistry import KeplerLayer
import pandas as pd

cities = pd.DataFrame({
    "id": ["NYC", "LA", "London"],
    "latitude": [40.7128, 34.0522, 51.5074],
    "longitude": [-74.0060, -118.2437, -0.1278]
})

g = (graphistry
     .nodes(cities, "id")
     .bind(point_latitude="latitude", point_longitude="longitude")
     .encode_kepler_dataset(id="cities", type="nodes")
     .encode_kepler_layer(KeplerLayer({
         "id": "city-points",
         "type": "point",
         "config": {
             "dataId": "cities",
             "columns": {"lat": "latitude", "lng": "longitude"},
             "visConfig": {"radius": 10, "opacity": 0.8}
         }
     }))
     .layout_settings(play=0))

g.plot()
```

Point Layers

Visualize nodes as points on a map with explicit layer configuration:

```
cities = pd.DataFrame({
    "city": ["NYC", "LA", "London", "Paris", "Tokyo"],
    "latitude": [40.7128, 34.0522, 51.5074, 48.8566, 35.6762],
    "longitude": [-74.0060, -118.2437, -0.1278, 2.3522, 139.6503]
})

g = (graphistry
    .nodes(cities, "city")
    .encode_kepler_dataset(id="nodes", type="nodes", label="Cities")
    .encode_kepler_layer(KeplerLayer({
        "id": "node-layer",
        "type": "point",
        "config": {
            "dataId": "nodes",
            "label": "Cities",
            "color": [255, 0, 0],
            "columns": {"lat": "latitude", "lng": "longitude"}
        }
    }))
    .layout_settings(play=0))
g.plot()
```

Arc Layers for Edges

Visualize edges as arcs between locations:

```
flights = pd.DataFrame({
    "origin": ["NYC", "LA"],
    "destination": ["LA", "London"]
})

g = (graphistry
     .nodes(cities, "id")
     .edges(flights, "origin", "destination")
     .bind(point_latitude="latitude", point_longitude="longitude")
     .encode_kepler_dataset(id="cities", type="nodes")
     .encode_kepler_dataset(id="flights", type="edges", map_node_coords=True)
     .encode_kepler_layer(KeplerLayer({
         "id": "points",
         "type": "point",
         "config": {
             "dataId": "cities",
             "columns": {"lat": "latitude", "lng": "longitude"}
         }
     }))
     .encode_kepler_layer(KeplerLayer({
         "id": "arcs",
         "type": "arc",
```

(continues on next page)

(continued from previous page)

```
        "config": {
            "dataId": "flights",
            "columns": {
                "lat0": "edgeSourceLatitude",
                "lng0": "edgeSourceLongitude",
                "lat1": "edgeTargetLatitude",
                "lng1": "edgeTargetLongitude"
            }
        }
    }))
    .layout_settings(play=0))
g.plot()
```

Hexagon Aggregation

Aggregate points into hexagonal bins for density visualization:

```
locations = pd.DataFrame({
    "location": ["NYC", "LA", "Chicago", "Houston", "Phoenix"],
    "latitude": [40.7128, 34.0522, 41.8781, 29.7604, 33.4484],
    "longitude": [-74.0060, -118.2437, -87.6298, -95.3698, -112.0740]
})

g = (graphistry
    .nodes(locations, "location")
    .encode_kepler_dataset(id="nodes", type="nodes", label="Locations")
    .encode_kepler_layer(KeplerLayer({
        "id": "density-layer",
        "type": "hexagon",
        "config": {
            "dataId": "nodes",
            "label": "Density",
            "columns": {"lat": "latitude", "lng": "longitude"},
            "visConfig": {
                "worldUnitSize": 1,
                "elevationScale": 5
            }
        }
    }))
    .layout_settings(play=0))
g.plot()
```

Geographic Regions

Visualize countries and states with built-in geographic data:

```
countries = pd.DataFrame({
    "country": ["USA", "GBR", "FRA"],
    "gdp": [21.43, 2.83, 2.72]
})

g = (graphistry
     .nodes(countries, "country")
     .encode_kepler_dataset(
         id="countries",
         type="countries",
         resolution=10,  # High resolution
         filter_countries_by_col="country"
     )
     .encode_kepler_layer(KeplerLayer({
         "id": "choropleth",
         "type": "geojson",
         "config": {
             "dataId": "countries",
             "columns": {"geojson": "_geojson"}
         }
     })))
```

Options and Config

Control map behavior and appearance (continuing from examples above):

```
# Method 1: Direct parameters
g = (g
     .encode_kepler_options(center_map=True, read_only=False)
     .encode_kepler_config(cull_unused_columns=True, overlay_blending="additive"))

# Method 2: Using KeplerEncoding builder
from graphistry import KeplerEncoding

encoding = (KeplerEncoding()
    .with_options(center_map=True, read_only=False)
    .with_config(cull_unused_columns=True, overlay_blending="additive"))
g2 = g.encode_kepler(encoding)

# Method 3: Using KeplerOptions/Config objects
from graphistry import KeplerOptions, KeplerConfig

opts = KeplerOptions(center_map=True, read_only=False)
cfg = KeplerConfig(cull_unused_columns=True, overlay_blending="additive")
encoding = KeplerEncoding(options=opts, config=cfg)
g3 = g.encode_kepler(encoding)
```

Complete Configuration

Build full Kepler configuration with multiple datasets, layers, options, and config:

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

cities = pd.DataFrame({
    "city": ["NYC", "LA", "London", "Paris", "Tokyo"],
    "latitude": [40.7128, 34.0522, 51.5074, 48.8566, 35.6762],
    "longitude": [-74.0060, -118.2437, -0.1278, 2.3522, 139.6503]
})

routes = pd.DataFrame({
    "origin": ["NYC", "LA", "London"],
    "destination": ["LA", "London", "Tokyo"]
})

config = (
    KeplerEncoding()
    .with_dataset(KeplerDataset(id="nodes", type="nodes", label="Cities"))
    .with_dataset(KeplerDataset(id="edges", type="edges", label="Routes"))
    .with_layer(KeplerLayer({
        "id": "node-layer",
        "type": "point",
        "config": {
            "dataId": "nodes",
            "columns": {"lat": "latitude", "lng": "longitude"}
        }
    }))
    .with_layer(KeplerLayer({
        "id": "edge-layer",
        "type": "arc",
        "config": {
            "dataId": "edges",
            "columns": {
                "lat0": "edgeSourceLatitude", "lng0": "edgeSourceLongitude",
                "lat1": "edgeTargetLatitude", "lng1": "edgeTargetLongitude"
            }
        }
    }))
    .with_options(center_map=True, read_only=False)
    .with_config(cull_unused_columns=True, overlay_blending="normal")
)

g = (graphistry
     .nodes(cities, "city")
     .edges(routes, "origin", "destination")
     .encode_kepler(config)
     .layout_settings(play=0))
g.plot()
```

See Also

- **API reference:** *Kepler API*
- **Configuration classes:** *graphistry.kepler.KeplerEncoding*, *graphistry.kepler.KeplerDataset*, *graphistry.kepler.KeplerLayer*
- **External docs:** https://docs.kepler.gl/

10.4.3.5 Comparison

Table 1: Geographic Visualization Approaches

Approach	Best For	Complexity	Key Features
Lat/Lon Bindings	Quick geographic viz	Simplest	Automatic server-side layout
Mercator Layout	Local coordinate analysis	Simple	GPU support, exportable coords
Kepler.gl	Rich interactive maps	Advanced	Multiple layers, styling, regions

10.4.3.6 Examples

See the Map Layout Demo notebook for complete examples:

- Map Layout Demo

10.4.3.7 API Reference

- *Mercator Layout API*
- *Kepler API Reference*
- *graphistry.PlotterBase.PlotterBase.bind()* (lat/lon bindings)

10.4.4 Quick Guide to PyGraphistry layouts

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

10.4.4.1 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*

10.4.4.2 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 on page load.

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

  Additionally, you can compute it ahead of time. Unlike the visualization server's pageload-time version, the PyGraphistry version uses the cuGraph (GPU) version, including a subset of the performance and quality improvements.

  ```
  g.fa2_layout().plot()
  ```

  For further details, refer to the *FA2 API*.

- **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* (*notebook*).

- **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()
  ```

(continues on next page)

(continued from previous page)

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

Read more in the *Modularity Layout API* (*notebook*).

- **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* (*notebook*).

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.

10.4.4.3 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*

10.4.5 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.

10.4.5.1 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:

- **Circle** — Positions nodes in a circular layout, useful for ordinal data, or separately laying out singleton nodes. *API info on circle layouts*
- **ForceAtlas2** — Optimized for large, dense graphs. Provides smooth clustering and cluster separation using GPU acceleration. PyGraphistry version gives visual and performance improvements upon other systems, and Graphistry server load-time version provides a different set of features focused on interactivity and additional options. *API info on FA2 layouts*
- **Modularity-Weighted** — Lays out clusters based on modularity, optimizing for visualizing community structures. *API info on modularity-weighted layouts* (*notebook*)
- **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* (*notebook*)
- **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* (*notebook*)

Example:

Visit the *PyGraphistry visualization tutorial*.

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

> **Note**
>
> When building layouts via GFQL or other JSON interfaces, provide `time_start`/`time_end` as ISO-8601 strings. PyGraphistry converts them to `numpy.datetime64` before computing the layout, so the experience matches direct Python usage where you pass Timestamp objects.

10.4.5.2 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()
```

10.4.5.3 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()
```

For static image export (SVG, PNG) instead of interactive visualization, see *`plot_static()`* and the static rendering tutorial.

10.4.5.4 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()
```

10.4.5.5 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
)
```

10.4.5.6 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.

10.4.6 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.

10.4.6.1 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 appearance
    show_arrows=False,
    point_size=1.0,          # Node size (logarithmic scale: 0.1-10.0 → UI 0-100)
    edge_curvature=0.0,      # 0.0 = straight edges
    edge_opacity=0.5,        # 0.0-1.0 range (50% transparent)
    point_opacity=0.9        # 0.0-1.0 range (90% opaque)
).plot()
```

Value Ranges:

- `point_size`: Range 0.1 to 10.0. The UI uses a logarithmic scale (0-100) for display. For example: 0.2 displays as approximately "15", 0.5 as "35", 1.0 as "50", 2.0 as "65", and 5.0 as "85". This logarithmic mapping provides finer control over smaller point sizes.
- `edge_curvature`: Range 0.0 to 1.0 (0.0 for straight edges, displayed as 0-100 in UI)
- `edge_opacity`: Range 0.0 to 1.0 (0.0 fully transparent, 1.0 fully opaque, displayed as 0-100 in UI)
- `point_opacity`: Range 0.0 to 1.0 (0.0 fully transparent, 1.0 fully opaque, displayed as 0-100 in UI)

Encodings (Color, Size, Icons)

Use the `encode_*` methods to style nodes and edges based on columns (for example, color by entity type). See the *Color encodings notebook* for full examples.

Collections

Collections define labeled subsets (nodes, edges, or subgraphs) using full GFQL and apply layered styling that overrides base encodings. Use them to call out alerts or critical paths on top of your standard color encodings, with priority-based overrides when subsets overlap.

For a full walkthrough, see the *Collections tutorial notebook*. For GFQL syntax, see *GFQL documentation*. For schema details, see https://hub.graphistry.com/docs/api/1/rest/url/#url-collections.

```
from graphistry import collection_set, n

collections = [
    collection_set(
        expr=n({"subscribed_to_newsletter": True}),
        id="newsletter_subscribers",
        name="Newsletter Subscribers",
        node_color="#32CD32",
    )
]

g2 = g.collections(
    collections=collections,
    show_collections=True,
    collections_global_node_color="CCCCCC",
    collections_global_edge_color="CCCCCC",
)
g2.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%"]
        ]
    }
})
```

(continues on next page)

(continued from previous page)

```
# 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 https://hub.graphistry.com/docs/api/1/rest/url/#urloptions and https://hub.graphistry.com/docs/api/2/rest/upload/metadata/.

10.4.6.2 HTML/URL-based Configuration

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

- https://hub.graphistry.com/docs/api/1/rest/url/#urloptions

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 https://hub.graphistry.com/docs/api/1/rest/url/#urloptions for more details.

10.5 GFQL: The Dataframe-Native Graph Query Language

Welcome to **GFQL**, the first fully vectorized dataframe-native graph query language with an open-source GPU runtime. GFQL is part of the **PyGraphistry** ecosystem and is designed to make graph analytics easier and faster without requiring a graph database as the execution layer. Whether you're working with **CPUs** or leveraging **GPU acceleration** for massive datasets, GFQL integrates directly into Python dataframe 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 high-performance graph queries directly on your dataframes. It is built to feel familiar to users of Cypher, other graph query languages, and popular dataframe libraries. By being native to accelerated Python data-science technologies such as Apache Arrow, NumPy, NVIDIA RAPIDS, and Graphistry, it can already handle workloads like 100M+ edges in interactive time on a single machine.

If you are new to Cypher: Cypher is a graph query language popularized by Neo4j and related tools. It uses ASCII-art graph patterns such as `(n1)-[e1]->(n2)` to describe traversals from one node to another across an edge. GFQL supports a bounded Cypher surface directly through `g.gfql("MATCH ...")`, so Cypher users can keep familiar `MATCH` / `WHERE` / `RETURN` patterns while moving execution onto GFQL's vectorized columnar engine and open-source GPU runtime. Use `g.gfql_remote([...])` when you want the same GFQL model executed remotely.

For Cypher syntax through `g.gfql("MATCH ...")`, start with *Cypher Syntax In GFQL*, *GFQL Quick Reference*, *GFQL RETURN*, and *Cypher to GFQL Mapping*.

Recommended paths:

- New to GFQL: *Overview of GFQL* -> *GFQL Quick Reference* -> *GFQL WHERE (Same-Path Constraints)* -> *GFQL RETURN (Row Pipelines)*
- Running Cypher syntax in GFQL: *Cypher Syntax In GFQL* -> *GFQL Quick Reference* -> *GFQL RETURN (Row Pipelines)* -> *Cypher to GFQL Python & Wire Protocol Mapping*
- Performance path (intro -> GPU -> remote GPU): *10 Minutes to GFQL* -> *GFQL Performance: Unleashing Vectorization and GPU Power for Scalable Graph Analytics* -> *GFQL Remote Mode*
- Translating existing Cypher to native GFQL: *Cypher to GFQL Python & Wire Protocol Mapping*
- Building agents/integrations: *GFQL Language Specification* + *GFQL Python Embedding* + *GFQL Wire Protocol Specification*

See also:

10.5.1 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.
- Run GFQL and Python on remote GPUs and remote data.

Let's dive in!

10.5.1.1 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.

10.5.1.2 Sample Dataset

Throughout this guide, we'll work with a graph representing people, companies, and transactions with risk indicators:

```
import pandas as pd
import graphistry

nodes_df = pd.DataFrame({
    'id': ['a', 'b', 'c', 'tx1', 'tx2'],
    'type': ['person', 'person', 'company', 'transaction', 'transaction'],
    'risk1': [False, False, False, True, False],
    'risk2': [False, False, False, False, True],
})
edges_df = pd.DataFrame({
    'src': ['a', 'b', 'a', 'tx1', 'tx2'],
    'dst': ['b', 'c', 'tx1', 'tx2', 'c'],
    'e_type': ['knows', 'works_at', 'sent', 'transfer', 'received'],
    'interesting': [True, True, False, False, False],
})

g = graphistry.edges(edges_df, 'src', 'dst').nodes(nodes_df, 'id')
```

10.5.1.3 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.

10.5.1.4 Two Syntax Styles

GFQL supports two syntax styles through the same `g.gfql(...)` entrypoint:

Cypher strings — familiar if you know SQL or Cypher:

```
# Filter nodes - returns a DataFrame
nodes_df = g.gfql("MATCH (n {type: 'person'}) RETURN n")._nodes
```

```
# Extract a subgraph - returns a graph with ._nodes and ._edges
g2 = g.gfql("GRAPH { MATCH (a)-[e]->(b) WHERE e.interesting = true }")
```

Native chain syntax — composable Python objects:

```
from graphistry import n, e_forward

# Same node filter, chain form
nodes_df = g.gfql([ n({"type": "person"}) ])._nodes
```

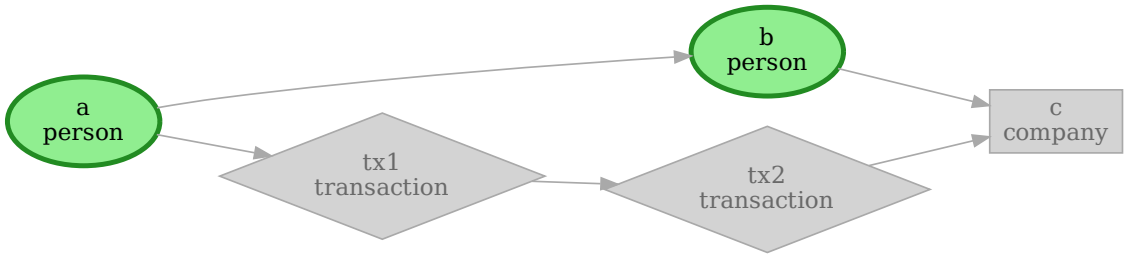
```
# Same subgraph extraction, chain form
g2 = g.gfql([ e_forward({"interesting": True}) ])
```

Both styles run on the same vectorized engine, with the same CPU/GPU acceleration. Use whichever you prefer — or mix them.

10.5.1.5 Examples

1. Find Nodes of a Certain Type

```
# Cypher style - returns a DataFrame of matching nodes
nodes_df = g.gfql("MATCH (n {type: 'person'}) RETURN n")._nodes

# Equivalent chain style
from graphistry import n
nodes_df = g.gfql([ n({"type": "person"}) ])._nodes
# nodes_df: DataFrame with 'a' and 'b' (the person nodes)
```



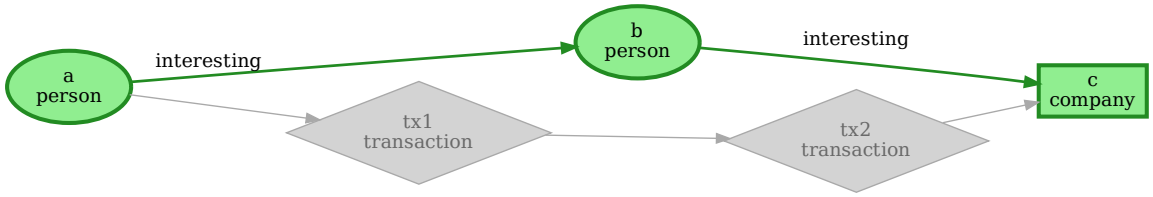
2. Find 2-Hop Edge Sequences with an Attribute

Traverse multiple hops and filter edges based on attributes.

```
# Cypher style - GRAPH { } returns a subgraph with ._nodes and ._edges
g2 = g.gfql("GRAPH { MATCH (a)-[e]->(b) WHERE e.interesting = true }")

# Equivalent chain style
from graphistry import e_forward
g2 = g.gfql([ e_forward({"interesting": True}, hops=2) ])
# g2._edges: edges a->b->c (both marked interesting)
g2.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

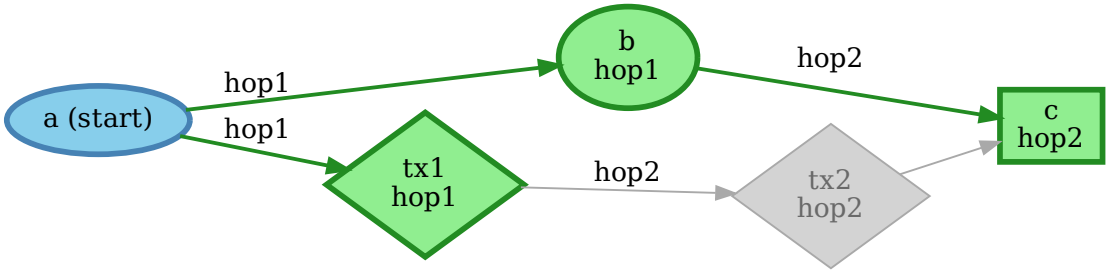
Label hops in your traversal to analyze specific relationships.

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

```
from graphistry import n, e_undirected

g_2_hops = g.gfql([
    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 ]
# first_hop_edges: edges directly connected to 'a' (hop1=True)
```

Explanation:

- `n({g._node: "a"})` starts the traversal from node `"a"` where `g._node` is the identifying column name.
- `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

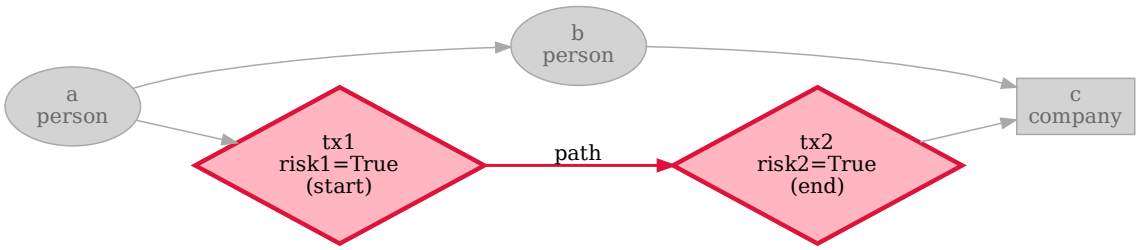
Chain multiple traversals to find patterns between nodes.

Example: Find transaction nodes between two types of risky nodes

```
from graphistry import n, e_forward, e_reverse

g_risky = g.gfql([
    n({"risk1": True}),
    e_forward(to_fixed_point=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed_point=True),
    n({"risk2": True})
])
hits = g_risky._nodes[ g_risky._nodes["hit"] == True ]
# hits: transaction nodes reachable from risk1 nodes and reaching risk2 nodes
```

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 import n, e_forward, e_reverse, is_in

g_filtered = g.gfql([
    n({"type": is_in(["person", "company"])}),
    e_forward({"e_type": is_in(["owns", "reviews"])}, to_fixed_point=True),
    n({"type": is_in(["transaction", "account"])}, name="hit"),
    e_reverse(to_fixed_point=True),
    n({"risk2": True})
])
```

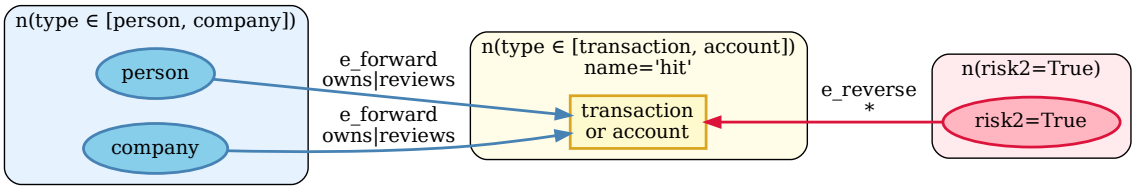
(continues on next page)

(continued from previous page)

```
hits = g_filtered._nodes[ g_filtered._nodes["hit"] == True ]
# hits: transaction/account nodes matching the traversal pattern
```

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`.



10.5.1.6 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.gfql([ ... ])
```

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.gfql([ ... ], 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.

10.5.1.7 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)
# top_nodes[['id', 'pagerank']]: DataFrame with highest PageRank nodes
```

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

```
from graphistry import n, e

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

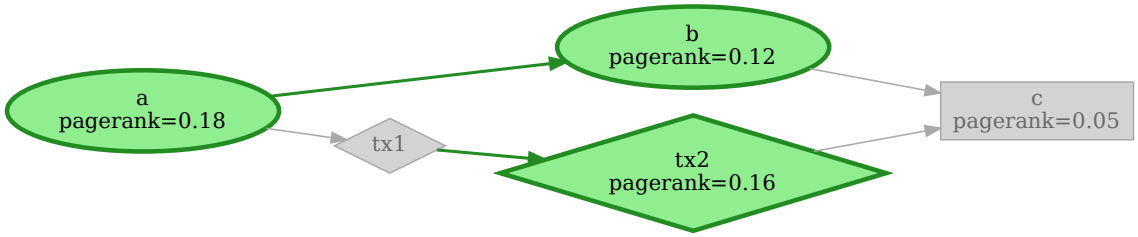
(continues on next page)

(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.



10. Sequencing Programs with Let

GFQL's Let bindings enable you to sequence complex graph programs as directed acyclic graphs (DAGs). This allows you to build sophisticated analysis pipelines with named operations that reference each other:

Example: Multi-stage fraud analysis

```
from graphistry import let, ref, call, n, e_forward, e, gt

result = g.gfql(let({
    # Stage 1: Find suspicious accounts
    'suspicious_accounts': n({'risk_score': gt(80), 'created_recent': True}),

    # Stage 2: Trace money flows from suspicious accounts
    'money_flows': [
        n({'risk_score': gt(80), 'created_recent': True}),
        e_forward({'type': 'transfer', 'amount': gt(10000)}, hops=3),
        n()
    ],

    # Stage 3: Compute PageRank to find central nodes
    'ranked': ref('money_flows', [
        call('compute_cugraph', {'alg': 'pagerank'})
    ]),

    # Stage 4: Identify high-risk clusters
    'high_risk_clusters': ref('ranked', [
        n({'pagerank': gt(0.01)}),
        e(),
        n(),
        call('compute_cugraph', {'alg': 'louvain'})
    ])
```

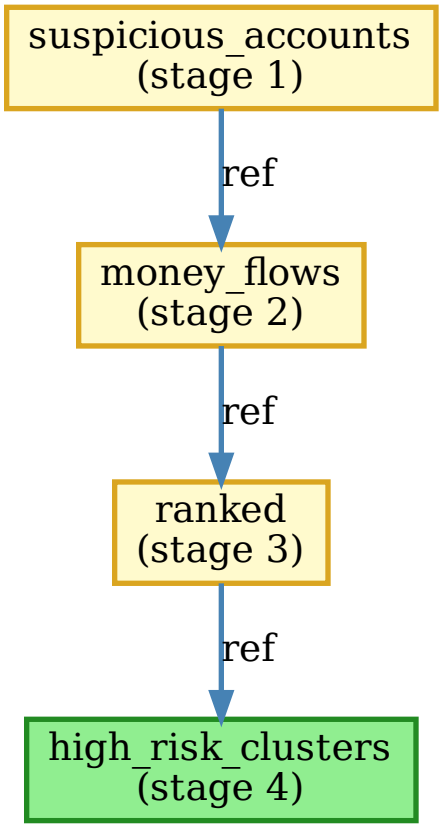
(continues on next page)

(continued from previous page)

```
}))

# Access results from each stage
suspicious = result._nodes[result._nodes['suspicious_accounts']]
clusters = result._nodes[result._nodes['high_risk_clusters']]
# suspicious: nodes flagged in stage 1
# clusters['community']: community assignments from stage 4
```

Key benefits of Let bindings:

- **Declarative DAG**: Express complex multi-stage analysis as a clear computation graph
- **Efficient execution**: All stages execute in a single optimized pass
- **Named results**: Access intermediate results by name for detailed analysis
- **Composability**: Build complex patterns from simpler named operations



11. Run remotely

You may want to run GFQL remotely because the data is remote or a GPU is available remotely:

Example: Run GFQL remotely

```
from graphistry import n, e

g2 = g1.gfql_remote([n(), e(), n()])
```

Example: Run GFQL remotely, and decouple the upload step

```
from graphistry import n, e

g2 = g1.upload()
assert g2._dataset_id is not None, "Uploading sets ``dataset_id`` for subsequent calls"
g3 = g2.gfql_remote([n(), e(), n()])
```

Additional parameters enable controlling options such as the execution `engine` and what is returned

Example: Bind to existing remote data and fetch it

```
import graphistry
from graphistry import n

g2 = graphistry.bind(dataset_id='my-dataset-id')

nodes_df = g2.gfql_remote([n()])._nodes
edges_df = g2.gfql_remote([e()])._edges
```

Example: Run Python on remote GPUs over remote data

```
def compute_shape(g):
    g2 = g.materialize_nodes()
    return {
        'nodes': g2._nodes.shape,
        'edges': g2._edges.shape
    }

g = graphistry.bind(dataset_id='my-dataset-id')
shape_info = g.python_remote_json(compute_shape)
# shape_info: {'nodes': (1000, 5), 'edges': (5000, 3)}
```

Example: Run Python on remote GPUs and return a graph

```
def compute_shape(g):
    g2 = g.materialize_nodes()
    return g2

g = graphistry.bind(dataset_id='my-dataset-id')
g2 = g.python_remote_g(compute_shape)
# g2._nodes: DataFrame returned from remote execution
```

10.5.1.8 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:

- **Try GFQL on Your Data:** Apply what you've learned to your datasets and see the benefits firsthand.
- *Translate Between SQL, Pandas, Cypher, and GFQL*
- *GFQL Quick Reference*
- *10 Minutes to 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!

10.5.2 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.

10.5.2.1 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.

10.5.2.2 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.
- **Architectural Freedom**: Use GFQL with your dataframes on your local CPU/GPU, or offload to a remote GPU cluster.

10.5.2.3 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* .

10.5.2.4 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
- **Cypher strings**: Write queries as Cypher strings — `g.gfql("MATCH (n) WHERE n.score > 5 RETURN n")`
- **Native chains**: Or compose queries as Python objects — `g.gfql([n({"score": gt(5)})])`
- **Predicates**: Apply conditions to filter nodes and edges based on their properties, reusing the optimized native operations of the underlying dataframe engine
- **Same-path constraints (WHERE)**: Relate attributes across steps in a chain using *where*

- **Row pipelines (`MATCH ... RETURN` style)**: Move from graph pattern matches to tabular results with *rows()*, *where_rows()*, *return_()*, *order_by()*, *group_by()*, *skip()*, and *limit()*
- **Result kinds**: Some stages keep you in graph state, while row-pipeline stages and row-returning local Cypher *CALL* queries move you into row state
- **GPU & CPU vectorization**: GFQL automatically leverages GPU acceleration and in-memory columnar processing for massive speedups on your queries
- **Optional remote mode**: Bind to remote data or upload it quickly as Arrow, and run your same Python and GFQL queries on remote GPU resources when available

10.5.2.5 Choosing Entry Points And Result Kinds

Use the entrypoint that matches where the query executes:

- **Local in-memory GFQL / Cypher-style execution**: *g.gfql([...])* or *g.gfql("MATCH ...")* runs on the current *Plottable* in pandas/cuDF.
- **Remote GFQL execution**: *g.gfql_remote([...])* runs the same GFQL chains/DAGs remotely, which is useful for larger datasets and remote GPU execution. See *GFQL Remote Mode*.

> **Warning**
>
> *graphistry.cypher("...")* and *g.cypher("...")* are a separate remote database Cypher path (for example, Neo4j/Neptune integrations), not the GFQL execution surface described on this page. Do not treat them as interchangeable with *g.gfql(...)* or *g.gfql_remote(...)*.

GFQL pipelines also have two practical result kinds:

- **Graph state**: Traversable graph results with meaningful *_nodes* and *_edges*. Matchers, graph-preserving *call(...)* transforms, *let() / ref()* DAG stages, local Cypher *CALL graphistry.*.write()* queries, and local Cypher *GRAPH { MATCH ... }* constructors stay in graph state.
- **Row state**: Tabular results stored in *_nodes*, with *_edges* reduced to an empty placeholder frame. Row-pipeline steps like *rows()*, *with_()*, *select()*, *return_()*, *group_by()*, and row-returning local Cypher *CALL ... YIELD ... RETURN ...* queries move into row state.
- A bare local Cypher procedure call without *.write()* is also row-returning. For example, *CALL graphistry.degree()* materializes the default procedure output columns into *_nodes* and clears *_edges*.

If you need to enrich a graph and keep matching locally, use graph-preserving *call() / let()* composition or a bare local Cypher *CALL graphistry.*.write()*. The local Cypher compiler currently supports *graphistry.degree.write()* plus *graphistry.igraph.<alg>.write()* and *graphistry.cugraph.<alg>.write()* for algorithms exposed through *compute_igraph() / compute_cugraph()*, along with a curated NetworkX subset including *graphistry.nx.pagerank.write()*, *graphistry.nx.betweenness_centrality.write()*, *graphistry.nx.degree_centrality.write()*, *graphistry.nx.closeness_centrality.write()*, *graphistry.nx.eigenvector_centrality.write()*, *graphistry.nx.katz_centrality.write()*, *graphistry.nx.connected_components.write()*, *graphistry.nx.strongly_connected_components.write()*, *graphistry.nx.core_number.write()*, *graphistry.nx.hits.write()*, *graphistry.nx.edge_betweenness_centrality.write()*, and *graphistry.nx.k_core.write()*.

10.5.2.6 Quick Examples

GFQL supports Cypher strings and native Python chains through the same `g.gfql(...)` entrypoint:

Find Nodes of a Certain Type

```
# Cypher string - returns a DataFrame of matching nodes
nodes_df = g.gfql("MATCH (n {type: 'person'}) RETURN n")._nodes

# Equivalent native chain
from graphistry import n
nodes_df = g.gfql([ n({"type": "person"}) ])._nodes
```

Extract a Subgraph

```
# Cypher string - GRAPH { } returns a subgraph with ._nodes and ._edges
g2 = g.gfql(
    "GRAPH { "
    "MATCH (a)-[e]->(b) "
    "WHERE e.interesting = true "
    "}"
)

# Equivalent native chain
from graphistry import n, e_forward
g2 = g.gfql([n(), e_forward({"interesting": True}, hops=2) ])
g2.plot()
```

Same-Path Constraints (WHERE)

Example: Match an account and its owner when both steps share an attribute.

```
from graphistry import n, e_forward, col, compare

g_filtered = g.gfql(
    [
        n({"type": "account"}, name="a"),
        e_forward(),
        n({"type": "user"}, name="c"),
    ],
    where=[compare(col("a", "owner_id"), "==", col("c", "owner_id"))],
)
```

Row-Pipeline `RETURN` Example

Example: Match people, filter rows, project columns, then sort/limit.

```
from graphistry import n, e_forward, gt
from graphistry.compute import rows, where_rows, return_, order_by, limit

top_people = g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person", "score": gt(0)}, name="p"),
    rows(table="nodes", source="p"),
```

(continues on next page)

(continued from previous page)

```
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(10),
])

top_people._nodes
```

Local Cypher `CALL ... .write()` Example

Example: Enrich a graph locally, keep graph state, then run a later *MATCH*.

```
g_enriched = g.gfql("CALL graphistry.degree.write()")
assert not g_enriched._edges.empty
top_degree = g_enriched.gfql(
    "MATCH (n) "
    "WHERE n.degree >= 2 "
    "RETURN n.id AS id, n.degree AS degree "
    "ORDER BY degree DESC, id ASC "
    "LIMIT 10"
)

top_degree._nodes
```

Local Cypher row-returning `CALL` Example

Example: Omit *.write()* when you want procedure rows instead of an enriched graph.

```
degree_rows = g.gfql("CALL graphistry.degree()")
assert degree_rows._edges.empty
degree_rows._nodes
```

This row result uses `nodeId` as the row identifier, stores the projected procedure outputs in `_nodes`, and clears `_edges`. Use `.write()` when the next step needs graph topology.

Example visualization (static):

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.gfql([
    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))
```

Filter by Date/Time

Example: Find recent transactions using temporal predicates.

```
from graphistry import n, e_forward
from graphistry.compute import gt, between
from datetime import datetime, date, time
import pandas as pd

# Find transactions after a specific date
recent = g.gfql([
    n(), e_forward(edge_match={"timestamp": gt(pd.Timestamp("2023-01-01"))}), n()
])

# Find transactions in a date range during business hours
business_hours_txns = g.gfql([
    n(), e_forward(edge_match={
        "date": between(date(2023, 6, 1), date(2023, 6, 30)),
        "time": between(time(9, 0), time(17, 0))
    }), n()
])
```

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.gfql([
    n({"risk1": True}),
    e_forward(to_fixed_point=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed_point=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.gfql([
    n({"type": is_in(["person", "company"])}),
    e_forward({"e_type": is_in(["owns", "reviews"])}, to_fixed_point=True),
    n({"type": is_in(["transaction", "account"])}, name="hit"),
    e_reverse(to_fixed_point=True),
    n({"risk2": True})
])
hits = g_filtered._nodes[ g_filtered._nodes["hit"] == True ]
print('Number of filtered hits:', len(hits))
```

DAG Patterns with Let Bindings

GFQL's Let bindings enable you to compose complex graph analyses by defining named subgraphs and operations that can reference each other. Like variables in programming, Let bindings make it easy to manipulate multiple graphs and subgraphs within a single query, while maintaining all the benefits of GFQL

like GPU acceleration.

Traditional Python approach (manual variable management):

```
# Traditional Python: Manually manage intermediate results
persons = g.gfql([n({'type': 'person'})])
adults = persons.gfql([n({'age': ge(18)})])
friends = adults.gfql([e_forward({'type': 'knows'})])
# Each step requires careful tracking of which graph to operate on
```

GFQL Let approach (declarative DAG with named bindings):

```
from graphistry import let, ref, n, e_forward, ge

# GFQL Let: Define a DAG of named operations
result = g.gfql(let({
    'persons': n({'type': 'person'}),
    'adults': ref('persons', [n({'age': ge(18)})]),  # Reference and filter persons
    'connections': [
        n({'type': 'person', 'age': ge(18)}),
        e_forward({'type': 'knows'}),
        n()  # Find connections from adults
    ]
}))

# Access any named result from the DAG
adults = result._nodes[result._nodes['adults']]
connections = result._edges[result._edges['connections']]
```

Key advantages of GFQL Let: - **Named subgraphs**: Create reusable named graph operations like constants in code - **Dependency management**: Automatically resolves dependencies between operations - **Composability**: Build complex multi-stage analyses from simpler named operations - **GPU preservation**: All operations maintain GPU acceleration when available - **Clean semantics**: Express complex graph analyses as clear, declarative DAGs

10.5.2.7 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')
```

(continues on next page)

(continued from previous page)

```
# Run GFQL query (executes on GPU)
g_result = g_gpu.gfql([ ... ])  # 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.gfql([ ... ], engine='cudf')
```

10.5.2.8 Run Remotely

You may want to run GFQL remotely such as if the data is remote, e.g., in Hub or cloud storage, and you have faster remote GPU servers for acting on it.

Bind to Remote Data and Query

Example: Bind to remote data and run queries on remote GPU resources.

```
import graphistry
from graphistry import n, e

g = graphistry.bind(dataset_id='my-dataset-id')

nodes_df = g.gfql_remote([ n() ])._nodes
```

Upload Data and Run GPU Python Remotely

Example: Upload local data to a remote GPU server and run full GPU Python tasks on it.

```
import graphistry
from graphistry import n, e

# Fully self-contained so can be transferred
def my_remote_trim_graph_task(g):
    # Trick: You can also put database fetch calls here!
    return (g
        .nodes(g._nodes[:10])
        .edges(g._edges[:10])
    )

# Upload any local graph data to the remote server
g2 = g1.upload()
print(g2._dataset_id, g2._nodes_file_id, g2._edges_file_id)

# Compute on it locally
g_result = g2.python_remote_g(my_remote_trim_graph_task)
print('Number of resulting edges:', len(g_result._edges))
```

See also *python_remote_table()* and *python_remote_json()* for returning other types of data.

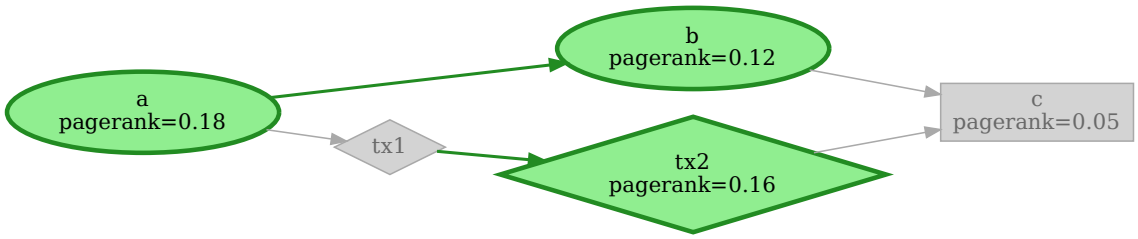
10.5.2.9 Visualizing GFQL Results

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

Example: Visualize high PageRank nodes.

```
from graphistry import n, e

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

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

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

Example visualization (graphviz):



Example visualization (interactive):

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*

10.5.2.10 GFQL APIs

Access detailed documentation of GFQL's API:

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

10.5.3 GFQL Remote Mode

You can run GFQL queries and GPU Python remotely, such as when data is already remote, gets big, or you would like to use a remote GPU

10.5.3.1 Basic Usage

Run chain remotely and fetch results

```
from graphistry import n, e
g2 = g1.gfql_remote([n(), e(), n()])
assert len(g2._nodes) <= len(g1._nodes)
```

`gfql_remote()` accepts the same input types as local `gfql()`:

> **Note**
>
> Collections are visualization URL settings; apply them after GFQL results (for example, `g2.collections(...)`). The GFQL remote/upload APIs do not accept collections payloads yet.

Method `chain_remote` runs chain remotely and fetches the computed graph

- **Chain / List[ASTObject]**: Native GFQL chain syntax (as above).
- **Cypher string**: Compiled locally, sent as wire-protocol JSON.
- **ASTLet / Let dict**: DAG patterns with named bindings.

```
# Cypher string (compiled locally, sent as Chain wire format)
g2 = g1.gfql_remote("MATCH (a)-[r]->(b) WHERE a.score > 10 RETURN a, b")

# Cypher string with params
g2 = g1.gfql_remote(
    "MATCH (n) WHERE n.score > $cutoff RETURN n",
    params={"cutoff": 10},
)

# GRAPH constructor (compiled locally, sent as Chain wire format)
```

(continues on next page)

(continued from previous page)

```
g2 = g1.gfql_remote("GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 10 }")

# Multi-stage pipeline (compiled locally, sent as Let wire format)
g2 = g1.gfql_remote(
    "GRAPH g1 = GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 10 } "
    "GRAPH g2 = GRAPH { USE g1 CALL graphistry.degree.write() } "
    "USE g2 MATCH (n) RETURN n.id, n.degree ORDER BY n.degree DESC"
)
```

Method `gfql_remote` runs the query remotely and fetches the computed graph.

- **chain**: GFQL query — Chain, List[ASTObject], ASTLet, Dict, or Cypher string.
- **output_type**: Defaulting to "all", whether to return the nodes (*'nodes'*), edges (*'edges'*), or both. See `gfql_remote_shape` to return only metadata.
- **node_col_subset**: Optionally limit which node attributes are returned to an allowlist.
- **edge_col_subset**: Optionally limit which edge attributes are returned to an allowlist.
- **engine**: Optional execution engine. Engine is typically not set, defaulting to *'auto'*. Use *'cudf'* for GPU acceleration and *'pandas'* for CPU.
- **validate**: Defaulting to *True*, whether to validate the query and data.

Note

A public `GraphSchema` bound with `g.bind(schema=schema)` is used by local validation APIs such as `g.gfql_validate(...)` and `g.gfql(..., validate=True)`. That schema surface is experimental in this release, and `gfql_remote(...)` does not currently send the schema object to the server. If you want declared-schema checks before a remote run, call `g.gfql_validate(query)` locally first, then call `g.gfql_remote(query)`. Remote schema transport is tracked as a follow-on capability.

10.5.3.2 Manual CPU, GPU engine selection

By default, GFQL will decide which engine to use based on workload characteristics like the dataset size. You can override this default by specifying which engine to use.

GPU

Run on GPU remotely and fetch results

```
from graphistry import n, e
g2 = g1.gfql_remote([n(), e(), n()], engine='cudf')
assert len(g2._nodes) <= len(g1._nodes)
```

CPU

Run on CPU remotely and fetch results

```
from graphistry import n, e
g2 = g1.gfql_remote([n(), e(), n()], engine='pandas')
```

10.5.3.3 Explicit uploads

Explicit uploads via *upload* will bind the field *Plottable::dataset_id*, so subsequent remote calls know to skip re-uploading. Always using explicit uploads can make code more predictable for larger codebases.

```
from graphistry import n, e
g2 = g1.upload()
assert g2._dataset_id is not None, "Uploading sets `dataset_id` for subsequent calls"

g3a = g2.gfql_remote([n()])
g3b = g2.gfql_remote([n(), e(), n()])
assert len(g3a._nodes) >= len(g3b._nodes)
```

10.5.3.4 Bind to existing remote data

If data is already uploaded and your user has access to it, such as from a previous session or shared from another user, you can bind it to a local *Plottable* for remote access.

```
import graphistry
from graphistry import  n, e

g1 = graphistry.bind(dataset_id='abc123')
assert g1._nodes is None, "Binding does not fetch data"

connected_graph_g = g1.gfql_remote([n(), e()])
connected_nodes_df = connected_graph_g._nodes
print(connected_nodes_df.shape)
```

10.5.3.5 Download less

You may not need to download all – or any – of your results, which can significantly speed up execution

Return only nodes

```
g1.gfql_remote([n(), e(), n()], output_type="nodes")
```

Return only nodes and specific columns

```
cols = [g1._node, 'time']
g2b = g1.gfql_remote(
  [n(), e(), n()],
  output_type="nodes",
  node_col_subset=cols)
assert len(g2b._nodes.columns) == len(cols)
```

Return only edges

```
g2a = g1.gfql_remote([n(), e(), n()], output_type="edges")
```

Return only edges and specific columns

```
cols = [g1._source, g1._destination, 'time']
g2b = g1.gfql_remote([n(), e(), n()],
  output_type="edges",
  edge_col_subset=cols)
assert len(g2b._edges.columns) == len(cols)
```

Return metadata but not the actual graph

```
from graphistry import n, e
shape_df = g1.chain_remote_shape([n(), e(), n()])
assert len(shape_df) == 2
print(shape_df)
```

10.5.3.6 Remote Python

You can also run full GPU Python tasks remotely, such as for more complicated code, or if you want the server itself to perform fetching such as from a database.

Run remote python on the current graph

```
import graphistry
from graphistry import n, e

# Fully self-contained so can be transferred
def my_remote_trim_graph_task(g):

    # Trick: You can also put database fetch calls here instead of using 'g'!
    return (g
        .nodes(g._nodes[:10])
        .edges(g._edges[:10])
```

(continues on next page)

(continued from previous page)

```
    )

# Upload any local graph data to the remote server
g2 = g1.upload()

g3 = g2.python_remote_g(my_remote_trim_graph_task)

assert len(g3._nodes) == 10
assert len(g3._edges) == 10
```

Run Python on an existing graph, return a table

```
import graphistry

g = graphistry.bind(dataset_id='ds-abc-123')

def first_n_edges(g):
    return g._edges[:10]

some_edges_df = g.python_remote_table(first_n_edges)

assert len(some_edges_df) == 10
```

Run Python on an existing graph, return JSON

```
import graphistry

g = graphistry.bind(dataset_id='ds-abc-123')

def first_n_edges_shape(g):
    return {'num_edges': len(g._edges[:10])}

obj = g.python_remote_json(first_n_edges_shape)

assert obj['num_edges'] == 10
```

10.5.4 GFQL Performance: Unleashing Vectorization and GPU Power for Scalable Graph Analytics

GFQL, developed by Graphistry, rethinks graph analytics by harnessing vectorization and GPU acceleration. As datasets grow from thousands to billions of rows, traditional tools struggle to keep up without significant infrastructure investment. GFQL is rewriting the story. Start small with a quick *pip install graphistry* on your CPU system, and scale more smoothly by leveraging the power of vectorization and GPUs to handle historically tricky datasets.

10.5.4.1 Built from Real-World Necessity

GFQL was born out of the challenges our team faced across many graph customer projects over the last 10 years. Projects often start with manageable datasets, and as they scale up, require tools that can grow without imposing prohibitive costs or complexities. Likewise, traditional graph solutions often require adding additional storage tier infrastructure and systems of record that duplicate a team's existing standard databases and warehouses: Too many projects died from premature distractions and complexities here.

We have https://gradientflow.com/what-is-graph-intelligence/ and the lack of effective libraries to leverage them for graph analytics. GFQL fills this gap. We designed GFQL to integrate seamlessly with the graph and dataframe ecosystem, providing a much easier, unified, and scalable solution while eliminating the need for hazardous storage tier detours.

10.5.4.2 A New Era of Graph Analytics

Graphistry has a history of award-winning open source data visualization and GPU acceleration engines. With GFQL, we bring our lessons learned to graph querying and analysis for real-time insights on datasets both big and small. Unlike traditional graph databases that process one path at a time, GFQL traverses entire collections simultaneously. Similar to best-of-class analytical CPU databases like Clickhouse and Google BigQuery, our vectorized approach maximizes throughput to drastically reduce query time.

When coupled with GPU acceleration, GFQL's performance reaches Graph 500 levels with even the cheapest cloud GPUs. Modern GPUs execute tens of thousands of threads in parallel, and GFQL is designed to fully saturate this capability. Whether you're traversing graphs with billions of edges or running complex algorithms, GFQL transforms previously impractical tasks into manageable ones.

10.5.4.3 Three Simple Ideas Behind GFQL's Performance

At the core of GFQL's performance are three pioneering techniques:

Collection-Oriented Algorithms

GFQL operates on entire collections of nodes and edges simultaneously, different from older commercial Cypher and Gremlin graph query engines that process one path at a time. The collection-oriented approach, inspired by our research at UC Berkeley and our experience with GPUs, maximizes data throughput and minimizes computational overhead. Small queries stay interactive, and large-scale graph analytics is now more efficient than ever before.

Vectorized Columnar Processing

GFQL processes data in large, parallel batches using columnar data structures. This method optimizes memory usage and computational efficiency, significantly speeding up data handling compared to traditional row-based systems. Natively integrating with cutting-edge technologies like https://arrow.apache.org/, this approach ensures high performance even on CPUs, and unusually fast speeds for moving large data across systems.

Massive Parallelism with GPUs

Designed to saturate the tens of thousands of threads in modern GPUs, GFQL enables rapid processing of complex graph queries. This massive parallelism allows GFQL to handle tasks that are impractical on typical CPU systems, such as real-time traversals that touch hundreds of millions of edges and compute on them.

10.5.4.4 Seamless Scalability from CPUs to GPUs

GFQL allows you to start analyzing graphs on standard CPUs without specialized hardware. As your data grows, you can transition to GPU acceleration without changing your code. GFQL intelligently utilizes available hardware to optimize performance, ensuring efficient resource use whether you're on a single machine or across a cluster.

By eliminating the need for additional infrastructure, GFQL reduces time and expense, allowing you to focus on extracting insights from your data. This seamless scalability ensures that as your projects evolve, GFQL adapts to meet your needs.

10.5.4.5 Optimized for Analytical Workloads

GFQL excels in scenarios requiring deep analytical capabilities. It is designed for:

- **Graph ETL and Analytics**: Efficiently process and transform large volumes of graph data.
- **Machine Learning and AI**: Accelerate graph-based ML and AI tasks, leveraging GPUs for training and inference.
- **Visualization**: Power high-performance graph visualizations, enabling real-time interaction with complex datasets.

By focusing on these areas, GFQL meets the demands of modern data projects, from initial exploration to advanced analysis, without the overhead typically associated with large-scale analytics.

> **Note**
>
> Same-path constraints (`where`) can be more expensive on dense graphs. Prefer selective per-step predicates and see *GFQL WHERE (Same-Path Constraints)* for details.

10.5.4.6 Built on Graphistry's Expertise

Graphistry's reputation for leveraging GPUs and vectorization in data analytics is well-established. GFQL embodies this expertise, filling gaps in the graph and dataframe ecosystem by providing tools that maximize GPU utilization and integrate with open-source technologies like Apache Arrow. Our collaboration with https://www.nvidia.com//, including their investment into our team, ensures that GFQL benefits from optimized kernel methods for top-tier performance.

10.5.4.7 Empower Your Data Journey

With GFQL, you can start quickly, scale more smoothly, and leverage cutting-edge performance. It empowers you to:

- Begin analyzing graphs immediately on your existing hardware
- Grow from CPU to GPU processing without code changes
- Handle datasets ranging from thousands to billions of edges efficiently

Whether you're analyzing social networks, investigating cybersecurity threats, or exploring intricate datasets, GFQL transforms how you work with graph data, making complex analytics accessible and efficient.

10.5.4.8 Join the Graphistry Community

We invite you to become part of our community dedicated to advancing graph analytics through innovation in vectorization and GPU computing. Let's keep pushing the boundaries of what's possible!

—

10.5.4.9 Next Steps

- **Explore GFQL**: Dive deeper into GFQL's capabilities in *10 Minutes to GFQL*.
- **Get Started with PyGraphistry**: Follow the *10 Minutes to PyGraphistry* to setup and experience the performance firsthand.
- **Learn About Vectorization and GPUs**: Understand the partner ecosystem technologies behind GFQL by exploring https://arrow.apache.org/ and https://rapids.ai/.
- **Connect with Us**: Join our *Join the Community* to share insights and collaborate with others pushing the boundaries of graph analytics.

10.5.5 GFQL Cypher Benchmark: CPU/GPU DataFrames vs Neo4j

Run Cypher graph queries and analytics directly on Python dataframes — no database required. This benchmark compares **Graphistry's local Cypher** (CPU and GPU) against **Neo4j + GDS** on the same end-to-end pipeline.

	Neo4j + GDS	GFQL Cypher (CPU)	GFQL Cypher (GPU)	GPU speedup vs Neo4j
Twitter (2.4M edges)	13.83s	2.55s	**0.30s**	**46x**
GPlus (30M edges)	>187s	75.78s	**3.33s**	**>56x**

Warm median of 5 runs, 2 warmup iterations. DGX dgx-spark, GB10 GPU.

10.5.5.1 The pipeline

One `g.gfql(...)` call — search, enrich with PageRank, search again:

```
# pip install graphistry
result = g.gfql("""
    GRAPH g1 = GRAPH {
      MATCH (n)-[e]-(m)
      WHERE n.degree >= $degree_cutoff
    }
    GRAPH g2 = GRAPH {
      USE g1
      CALL graphistry.cugraph.pagerank.write()
    }
    GRAPH {
      USE g2
      MATCH (n)-[e]-(m)
      WHERE n.pagerank >= $pagerank_cutoff
    }
""",
    params={
        "degree_cutoff": degree_cutoff,
        "pagerank_cutoff": pagerank_cutoff,
    },
    engine="cudf",  # or "pandas" with igraph backend
)
```

- `GRAPH g1`: find high-degree nodes and their neighbors
- `GRAPH g2`: enrich `g1` with PageRank scores (igraph on CPU, cugraph on GPU)
- Final `GRAPH`: keep high-PageRank nodes and their neighbors

The same pipeline shape, different backends:

- **CPU**: `engine="pandas"`, `backend="igraph"`
- **GPU**: `engine="cudf"`, `backend="cugraph"`

The Neo4j equivalent requires ~30 lines of Cypher + GDS projection + batched writes (see *Neo4j + GDS analog* below).

10.5.5.2 Twitter (2.4M edges): exact 3-way comparison

Stacked by workload phase: **ETL** (load + shape), **Search** (graph queries), **Analytics** (PageRank).

- Neo4j total lifecycle: ~21.6s (6.0s import + 1.7s prep + 13.8s pipeline)
- GFQL Cypher CPU: ~2.8s — **8x faster than Neo4j**
- GFQL Cypher GPU: ~0.4s — **54x faster than Neo4j**

10.5.5.3 GPlus (30M edges): larger graph

- Neo4j: **>187s** (lower bound — the transaction did not finish)
- GFQL Cypher CPU: ~85.5s — still faster than Neo4j's incomplete run
- GFQL Cypher GPU: ~7.1s — **>26x faster than Neo4j**

10.5.5.4 Why this matters

The CPU path already beats Neo4j without a GPU. You get Cypher-style graph search + PageRank directly on your dataframe, no database to stand up or maintain.

The GPU path accelerates everything — ETL, search, and analytics — because `cudf` and `cugraph` are drop-in replacements for `pandas` and `igraph` under the same GFQL Cypher surface.

10.5.5.5 Neo4j + GDS analog

The Neo4j equivalent of the same pipeline:

```
// 1. Mark seed nodes by degree
MATCH (n:Node)
SET n.seed = n.degree >= $cutoff;

// 2. Expand one hop from seeds
UNWIND $seed_ids AS sid
MATCH (s:Node) WHERE id(s) = sid
MATCH (s)-[r:LINK]-(target:Node)
SET target.in_subgraph = true, r.in_subgraph = true;

// 3. Project subgraph and run PageRank
CALL gds.graph.project.cypher(
  'subgraph',
  'MATCH (n:Node) WHERE n.in_subgraph RETURN id(n) AS id',
  'MATCH (a)-[r:LINK]->(b) WHERE r.in_subgraph
   RETURN id(a) AS source, id(b) AS target
   UNION ALL
   MATCH (a)-[r:LINK]->(b) WHERE r.in_subgraph
   RETURN id(b) AS source, id(a) AS target'
);
CALL gds.pageRank.write('subgraph', {writeProperty: 'pagerank'});

// 4. Keep high-PageRank core + one hop
MATCH (n:Node) WHERE n.pagerank >= $cutoff
SET n.core = true;
UNWIND $core_ids AS cid
MATCH (c:Node) WHERE id(c) = cid
MATCH (c)-[r:LINK]-(target:Node)
SET target.final = true, r.final = true;
```

10.5.5.6 Why the GFQL pipeline is shorter

The difference in pipeline length above is not accidental. It reflects a design difference in how graphs flow through the system:

Graphs as first-class values. GFQL's `GRAPH { }` constructors treat graphs as composable values that flow between pipeline stages. Each stage receives a graph, transforms it, and passes a graph to the next stage. Standard Cypher and GQL are constrained to paths and rows as output values, which forces the Neo4j pipeline into explicit property-flag marking, separate GDS projections, and batched write-back steps.

Multi-language, single engine. The GFQL engine is being designed to support Cypher, and over time additional property graph query languages, all compiled to the same vectorized columnar execution backend. Users write in whichever declarative syntax they prefer; the engine handles CPU/GPU dispatch transparently. See *Cypher Syntax In GFQL* for the current Cypher surface and *Overview of GFQL* for the broader GFQL design.

Modern execution without legacy constraints. Because GFQL does not inherit a database storage layer or a row-at-a-time execution model, it can represent intermediate graph results natively in columnar memory (Arrow / pandas / cuDF). That is what makes the CPU-to-GPU switch a configuration flag (`engine="cudf"`) rather than a rewrite, and what keeps ETL, search, and analytics in the same in-process pipeline.

For more on the GFQL design and supported surface:

- *Cypher Syntax In GFQL* — Cypher syntax through `g.gfql("MATCH ...")`
- *Overview of GFQL* — GFQL design, features, and GPU acceleration
- *10 Minutes to GFQL* — 10-minute introduction to GFQL

10.5.5.7 Benchmark environment

- Host: `dgx-spark`, GPU: `GB10`, driver `580.126.09`
- Container: `graphistry/test-gpu:latest`
- Datasets: https://snap.stanford.edu/data/ Twitter (2.4M edges) and GPlus (30M edges)
- Measurement: median of 5 runs after 2 warmup iterations
- Results rendered from saved JSON — this page does **not** rerun benchmarks

10.5.5.8 Notebook version

See `demos/gfql/benchmark_filter_pagerank_cpu_gpu.ipynb` for a notebook version of this writeup using the same saved DGX results.

10.5.6 Translate Between SQL, Pandas, Cypher, and GFQL

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

10.5.6.1 Introduction

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

GFQL operates on graph DataFrames - graphs represented as node and edge DataFrames. This DataFrame-native approach enables seamless integration with the PyData ecosystem and natural vectorization for both CPU and GPU processing.

GFQL accepts both **native chain syntax** (`g.gfql([n(), e(), n()])`) and **Cypher strings** (`g.gfql("MATCH ...")`). Most examples below show both forms. GFQL also extends Cypher with `GRAPH { }` constructors for graph-state results and composable multi-stage pipelines — see *Cypher Syntax In GFQL* for the full Cypher-in-GFQL guide.

10.5.6.2 Who Is This Guide For?

- **Data Scientists:** Familiar with Pandas or SQL, exploring graph relationships.
- **Engineers:** Integrating graph queries into applications.
- **DBAs:** Understanding how GFQL complements SQL for graph data.
- **Graph Specialists:** Experienced with Cypher, integrating graph queries into Python.

10.5.6.3 Common Graph and Query Tasks

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

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 (chain syntax)

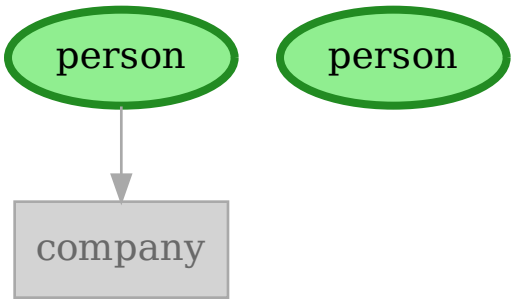
```
from graphistry import n

# df[['id', 'type', ...]]
g.gfql([ n({"type": "person"}) ])._nodes
```

GFQL (Cypher syntax)

```
# Row result - same as standard Cypher MATCH/RETURN
g.gfql("MATCH (n {type: 'person'}) RETURN n")._nodes

# Graph result - GFQL extension, keeps graph state
g.gfql("GRAPH { MATCH (n {type: 'person'}) }")._nodes
```

Explanation:

- **GFQL chain**: `n({"type": "person"})` filters nodes where `type` is `"person"`. `g.gfql([...])` 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).
- **GFQL Cypher**: The same query as a Cypher string. `MATCH ... RETURN` gives row output; `GRAPH { MATCH ... }` keeps graph state (both `_nodes` and `_edges`).



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 (chain syntax)

```
from graphistry import n, e_forward

# df[['src', 'dst', ...]]
```

(continues on next page)

(continued from previous page)

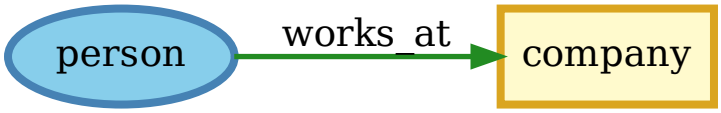
```
g.gfql([
    n({"type": "person"}), e_forward(), n({"type": "company"})
])._edges
```

GFQL (Cypher syntax)

```
# Graph result - keeps matched subgraph with edges
g.gfql(
    "GRAPH { MATCH (n1 {type: 'person'})-[e]->(n2 {type: 'company'}) }"
)._edges

# Row result - returns edge properties as rows
g.gfql(
    "MATCH (n1 {type: 'person'})-[e]->(n2 {type: 'company'}) RETURN e"
)._nodes
```

Explanation:

- **GFQL chain**: Starts from nodes of type `"person"`, traverses forward edges, and reaches nodes of type `"company"`. This version starts to gain the legibility and maintainability benefits of graph query syntax for graph tasks, and maintains the performance benefits of automatically vectorized pandas and GPU-accelerated cuDF.
- **GFQL Cypher**: `GRAPH { MATCH ... }` returns the matched subgraph (graph state with `_edges`); `MATCH ... RETURN e` returns edge properties as rows.
- **Same-path constraints**: Use *where* to relate attributes across steps (same-path scope only; see *GFQL WHERE (Same-Path Constraints)*).



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 (
```

(continues on next page)

(continued from previous page)

```
    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 (chain syntax)

```
from graphistry import n, e_forward

# df[['id', ...]]
g.gfql([
    n({g._node: "Alice"}), e_forward(), e_forward(), n(name='m')
])._nodes.query('m')
```

GFQL (Cypher syntax)

```
# Row result - return 2-hop destinations
g.gfql("MATCH (n {id: 'Alice'})-->()-->(m) RETURN m")._nodes

# Graph result - return the 2-hop subgraph
g.gfql("GRAPH { MATCH (n {id: 'Alice'})-->()-->(m) }")
```

GFQL (bounded hop alternative)

```
# Same intent using hop() with explicit bounds + optional labels
g.hop(
    nodes=pd.DataFrame({g._node: ['Alice']}),
    min_hops=2,
    max_hops=2
)
```

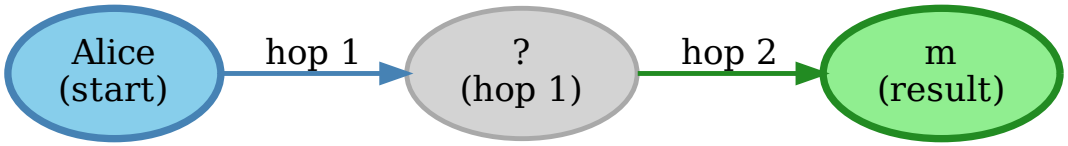
Explanation:

- *min_hops*/*max_hops* express the same bounded traversal intent as a Cypher pattern like *[*2..2]* after translation into native GFQL. Direct *g.gfql("MATCH ... ")* now supports the endpoint-only *[*...]* slice, but *hop()* still gives you the full native GFQL control surface for labels, output slicing, and more complex multihop rewrites. If you set *label_node_hops*/*label_edge_hops*, those column names will store the hop step (nodes = first arrival, edges = traversal step); omit or *None* to skip labels.
- *output_min_hops*/*output_max_hops* (optional) slice the displayed hops after traversal. By default, all traversed hops up to *max_hops* remain visible; set *output_min_hops* if you want to drop early

hops (e.g., traverse 2..4 but only show 3..4). Invalid slices (e.g., *output_min_hops* > *max_hops* or *output_max_hops* < *min_hops*) raise a *ValueError*.

Explanation:

- **GFQL**: Starts at node `"Alice"`, performs two forward hops, and obtains nodes two steps away. Results are in `nodes_df`. Building on the expressive and performance benefits of the previous 1-hop example, it begins adding the parallel path finding benefits of GFQL over Cypher, which benefits both CPU and GPU usage.



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 (chain syntax)

```
from graphistry import e_forward

# df[['src', 'dst', 'weight', ...]]
g.gfql([ e_forward(edge_query='weight > 0.5') ])._edges
```

GFQL (Cypher syntax)

```
g.gfql(
    "MATCH ()-[e]->() WHERE e.weight > 0.5 RETURN e"
)._nodes
```

Explanation:

- **GFQL chain**: Uses `e_forward(edge_query='weight > 0.5')` to filter edges where `weight > 0.5`. This version introduces the string query form that can be convenient. Underneath, it still benefits from the vectorized execution of Pandas and cuDF.
- **GFQL Cypher**: Same filter as a Cypher `WHERE` clause.

Aggregations and Grouping

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

SQL

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

Pandas

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

Cypher

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

GFQL (dataframe)

```python
# df[['src', 'out_degree']]
g._edges.groupby('src').size().reset_index(name='out_degree')
```

GFQL (Cypher syntax)

```python
# Enrich graph with degree columns, then query
g.gfql("CALL graphistry.degree.write()")._nodes

# Or as a single pipeline - enrich then return top nodes
g.gfql(
    "GRAPH g1 = GRAPH { CALL graphistry.degree.write() } "
    "USE g1 "
    "MATCH (n) RETURN n.id AS node_id, n.degree_out AS out_degree "
    "ORDER BY out_degree DESC LIMIT 10"
)._nodes
```

Explanation:

- **GFQL dataframe**: 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.
- **GFQL Cypher**: `CALL graphistry.degree.write()` enriches the graph with degree columns in graph state. Wrap in `GRAPH { }` to compose with subsequent queries in a single expression.

All Paths and Connectivity

Objective: Find all paths between nodes `"Alice"` and `"Bob"` that go through friendships.

SQL

```sql
WITH RECURSIVE path AS (
    -- Base case: Start from "Alice" (no type or edge restrictions)
    SELECT e.src, e.dst, ARRAY[e.src, e.dst] AS full_path, 1 AS hop
    FROM edges e
    WHERE e.src = 'Alice'

    UNION ALL

    -- Recursive case: Expand path where intermediate src/dst are 'people' and edge is
↪'friend'
    SELECT e.src, e.dst, full_path || e.dst, p.hop + 1
    FROM edges e
    JOIN path p ON e.src = p.dst
    JOIN nodes n_src ON e.src = n_src.id  -- Check src type for intermediate nodes
    JOIN nodes n_dst ON e.dst = n_dst.id  -- Check dst type for intermediate nodes
    WHERE n_src.type = 'person' AND n_dst.type = 'person'  -- Intermediate nodes must be
↪'people'
    AND e.type = 'friend'  -- Intermediate edges must be 'friend'
    AND e.dst != ALL(full_path)  -- Avoid cycles (optional)
)
-- Final filter to ensure the path ends with "Bob"
SELECT *
FROM path
WHERE dst = 'Bob';
```

Pandas

```python
def find_paths_fixed_point(edges_df, nodes_df, start_node, end_node):
    # Initialize paths with base case (start with 'Alice')
    paths = [{'path': [start_node], 'last_node': start_node}]
    all_paths = []
    expanded = True  # Continue loop as long as there are paths to expand

    while expanded:
        new_paths = []
        expanded = False

        # Expand each path
        for path in paths:
            last_node = path['last_node']

            # Find all outgoing 'friend' edges from the last node
            valid_edges = edges_df.merge(nodes_df, left_on='dst', right_on='id') \
                                  .merge(nodes_df, left_on='src', right_on='id') \
                                  [(edges_df['src'] == last_node) &
                                  (edges_df['type'] == 'friend') &
                                  (nodes_df['type_x'] == 'person') &  # src is 'person'
                                  (nodes_df['type_y'] == 'person')]   # dst is 'person'
```

(continues on next page)

(continued from previous page)

```
            for _, edge in valid_edges.iterrows():
                new_path = path['path'] + [edge['dst']]

                # If we reached 'Bob', add to all_paths
                if edge['dst'] == end_node:
                    all_paths.append(new_path)
                else:
                    # Otherwise, add to new paths to continue expanding
                    new_paths.append({'path': new_path, 'last_node': edge['dst']})
                    expanded = True  # Mark that we found new paths to expand

        # Stop if no new paths were found (fixed-point behavior)
        paths = new_paths

    return all_paths

# Run the pathfinding function to fixed point
paths = find_paths_fixed_point(edges_df, nodes_df, 'Alice', 'Bob')
```

Cypher

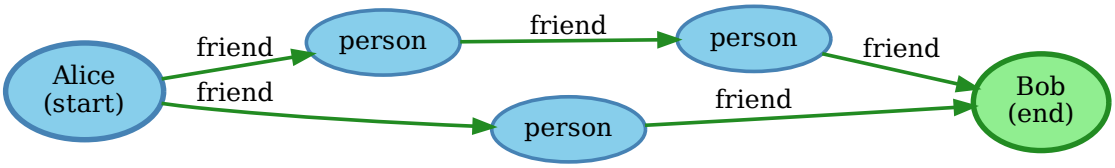
```
MATCH p = (n1 {id: 'Alice'})-[e:friend*]-(n2 {id: 'Bob'})
WHERE ALL(rel IN relationships(p) WHERE type(rel) = 'friend')
AND ALL(node IN NODES(p) WHERE node.type = 'person')
RETURN p;
```

GFQL

```
# g._edges: df[['src', 'dst', ...]]
# g._nodes: df[['id', ...]]
g.gfql([
    n({"id": "Alice"}),
    e_forward(
        source_node_query='type == "person"',
        edge_query='type == "friend"',
        destination_node_query='type == "person"',
        to_fixed_point=True),
    n({"id": "Bob"})
])
```

Explanation:

- **GFQL**: Uses `e(to_fixed_point=True)` to find edge sequences of arbitrary length between nodes `"Alice"` and `"Bob"`. The SQL and Pandas version suffer from syntactic and semantic imepedance mismatch with graph tasks on this example.



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 (Python)

```
# g._nodes: df[['id', 'louvain']]
g.compute_cugraph('louvain')._nodes
```

GFQL (Cypher syntax)

```
# Graph-preserving enrichment via CALL .write()
g.gfql("CALL graphistry.cugraph.louvain.write()")._nodes

# Full pipeline: filter subgraph, enrich, query results
g.gfql(
    "GRAPH g1 = GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 5 } "
    "GRAPH g2 = GRAPH { USE g1 CALL graphistry.cugraph.louvain.write() } "
    "USE g2 "
    "MATCH (n) RETURN n.id AS id, n.louvain AS community "
    "ORDER BY community, id"
)._nodes
```

Explanation:

- **GFQL Python**: Enriches with many algorithms such as the GPU-accelerated `graphistry.plugins.cugraph.compute_cugraph()` for community detection. Any CPU and GPU library can be used, with top plugins already natively supported out-of-the-box.
- **GFQL Cypher**: `CALL graphistry.cugraph.louvain.write()` runs the same algorithm via GFQL's Cypher surface. Wrapping in `GRAPH { }` with `USE` enables single-expression pipelines that filter, enrich, and query.

Time-Windowed Graph Analytics

Objective: Find all edges between nodes `"Alice"` and `"Bob"` that occurred in the last 7 days.

SQL

```
SELECT * FROM edges
WHERE ((src = 'Alice' AND dst = 'Bob') OR (src = 'Bob' AND dst = 'Alice'))
  AND timestamp >= NOW() - INTERVAL '7 days';
```

> **Warning**
>
> This version incorrectly simplifies to a two-hop relationship. For multihop scenarios, refer to *All Paths and Connectivity* for more advanced techniques.

Pandas

```
filtered_edges_df = edges_df[
    ((edges_df['src'] == 'Alice') & (edges_df['dst'] == 'Bob')) |
    ((edges_df['src'] == 'Bob') & (edges_df['dst'] == 'Alice')) &
    (edges_df['timestamp'] >= pd.Timestamp.now() - pd.Timedelta(days=7))
]
```

> **Warning**
>
> This version incorrectly simplifies to a two-hop relationship. For multihop scenarios, refer to *All Paths and Connectivity* for more advanced techniques.

Cypher

```
MATCH path = (a {id: 'Alice'})-[e]-(b {id: 'Bob'})
WHERE e.timestamp >= datetime().subtract(duration({days: 7}))
RETURN e;
```

GFQL

```
from graphistry import n, e_forward, is_in

past_week = pd.Timestamp.now() - pd.Timedelta(7)
g.gfql([
    n({"id": is_in(["Alice", "Bob"])}),
    e_forward(edge_query=f'timestamp >= "{past_week}"'),
    n({"id": is_in(["Alice", "Bob"])})
])._edges
```

Explanation:

- **SQL** and **Pandas**: These versions incorrectly simplify to a two-hop relationships; for multihop scenarios, refer to *All Paths and Connectivity*.
- **GFQL**: Utilizes the `chain` method to filter edges between `"Alice"` and `"Bob"` based on a timestamp within the last 7 days. This approach allows for multihop relationships as it leverages the graph's structure, and further using cuDF for GPU acceleration when available.

Parallel Pathfinding

Objective: Find all paths from `"Alice"` to `"Bob"` and `"Charlie"` in parallel. Parallel pathfinding is particularly interesting because it allows for efficient querying of multiple target nodes at the same time, reducing the time and complexity required to compute multiple independent paths, especially in large graphs.

SQL

- **Not suitable**: SQL is not designed for pathfinding on graphs.

Pandas

- **Not suitable**: Pandas is not designed for pathfinding across graphs.

Cypher

> **Warning**
>
> Cypher is **path-oriented** and does not natively support parallel pathfinding. Each path must be processed individually, which can result in performance bottlenecks for large graphs or multiple targets. Neo4j users can utilize the APOC or GDS libraries to add parallelism, but this is a limited external workaround, rather than a native strength.

```
MATCH (a {id: 'Alice'}), (target)
WHERE target.id IN ['Bob', 'Charlie']
CALL algo.shortestPath.stream(a, target)
YIELD nodeId, cost
RETURN nodeId, cost;
```

GFQL

```
from graphistry import n, e_forward, is_in

# g._nodes: cudf.DataFrame[['src', 'dst', ...]]
g.gfql([
    n({"id": "Alice"}),
    e_forward(to_fixed_point=False),
    n({"id": is_in(["Bob", "Charlie"])})
], engine='cudf')
```

Explanation:

- **Cypher**: Cypher processes paths individually and does not support native parallelism. Libraries like APOC or GDS offer a way to achieve parallel execution, but this adds complexity.
- **GFQL**: GFQL natively supports parallel pathfinding using a bulk wavefront algorithm, processing all paths at once, making it highly efficient in GPU-accelerated environments.

GPU Execution

*Objective**: Execute pathfinding queries on the GPU, computing all paths from `"Alice"` to `"Bob"` and `"Charlie"` simultaneously across hardware resources.

SQL

- **Not suitable**: SQL is not designed for parallel execution of graph queries.

Pandas

- **Not suitable**: Pandas is not designed for parallel execution across graphs.

Cypher

- **Not suitable**: Popular Cypher engines like Neo4j do not natively support GPU execution.

GFQL

```
from graphistry import n, e_forward, is_in

# Executing pathfinding queries in parallel
g.gfql([
    n({"id": "Alice"}),
    e_forward(to_fixed_point=False),
    n({"id": is_in(["Bob", "Charlie"])})
], engine='cudf')
```

Explanation:

This example builds on the previous one, showing how **GFQL** handles parallel execution natively. GFQL benefits from **bulk vector processing**, which boosts performance in both CPU and GPU modes:

- **In CPU environments**, the bulk processing model accelerates query execution algorithmically and takes advantage of hardware parallelism, improving efficiency.
- **In GPU mode**, GFQL **natively parallelizes** pathfinding, further leveraging hardware acceleration to process multiple paths concurrently and quickly, making it highly efficient for large-scale graph traversals.

10.5.6.4 GFQL Functions and Equivalents

Node Matching

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

Edge Matching

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

- **GFQL chain**: `e_forward({ "property": value })` or `e_reverse({ "property": value })` or `e({ "property": value })`
- **GFQL Cypher**: `g.gfql("MATCH ()-[e {property: value}]->() RETURN e")`

Traversal

- **SQL**: Complex joins or recursive queries
- **Pandas**: Multiple merges; not efficient for deep traversals
- **Cypher**: Patterns like `()-[]->()` for traversal
- **GFQL chain**: Chains of `n()`, `e_forward()`, `e_reverse()`, and `e()` functions
- **GFQL Cypher**: `g.gfql("MATCH (a)-[]->(b) RETURN ...")` or `g.gfql("GRAPH { MATCH ... }")`

Graph-State Results (subgraph extraction)

- **SQL**: Not applicable
- **Pandas**: Manual node/edge DataFrame filtering
- **Cypher**: Not supported (Cypher always returns rows)
- **GFQL chain**: `g.gfql([n(...), e_forward(), n()])` — inherently graph-returning
- **GFQL Cypher**: `g.gfql("GRAPH { MATCH (a)-[r]->(b) WHERE ... }")` — GFQL extension

Graph Enrichment (algorithms)

- **SQL**: Not applicable
- **Pandas**: External library calls
- **Cypher**: `CALL algo.pagerank.stream()`
- **GFQL Python**: `g.compute_cugraph('pagerank')` or `g.compute_igraph('pagerank')`
- **GFQL Cypher**: `g.gfql("CALL graphistry.cugraph.pagerank.write()")` or `g.gfql("GRAPH { CALL graphistry.cugraph.pagerank.write() }")`

Multi-Stage Pipelines

- **SQL**: CTEs or temp tables
- **Pandas**: Sequential variable assignment
- **Cypher**: Not supported in standard Cypher (row-only)
- **GFQL chain**: Sequential `g.gfql([...])` calls
- **GFQL Cypher**: `GRAPH g1 = GRAPH { ... } GRAPH g2 = GRAPH { USE g1 CALL ... } USE g2 MATCH ... RETURN ...`

10.5.6.5 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.

10.5.6.6 Additional Resources

- *GFQL Quick Reference*
- *GFQL Operator Reference*: Use predicates for filtering on nodee and edge attributes.
- *10 Minutes to PyGraphistry*: Visualize GFQL queries with GPU-accelerated tools.

10.5.6.7 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!

10.5.7 Combine GFQL with PyGraphistry Loaders, ML, AI, & Visualization

10.5.7.1 Common Graph Visualization Tasks

For an introduction to visualization techniques, see *10 Minutes to Graphistry Visualization*.

Simple Plotting

Objective: Quickly visualize a graph using the *.plot()* method.

Code

```
g2 = g1.gfql(gfql_query)
g2.plot()
```

Explanation:

This creates an interactive graph visualization of the GFQL results using the *graphistry.PlotterBase.PlotterBase.plot()* method.

10.5.7.2 Common Data Loading and Shaping Tasks

We'll cover a range of common tasks related to data loading and shaping for graph structures:

- *Data Loading with Pandas from CSV*
- *GPU Data Loading with cuDF from Parquet*
- *Bind Nodes and Edges*
- *Handle Multiple Node Columns with Hypergraphs*

Data Loading with Pandas from CSV

Objective: Demonstrate loading data from CSV files using *pandas* and converting to graph structures.

Code

```
import pandas as pd
import graphistry

# pd.DataFrame[['src', 'dst', ...]]
df = pd.read_csv('data.csv')

# g._edges: df[['src', 'dst', ...]]
g = graphistry.edges(df, 'src', 'dst')
```

Explanation:

This example illustrates how to load data from a CSV file using *pandas* and bind it into a graph structure via *graphistry.PlotterBase.PlotterBase.edges()* for using **GFQL**. For more information on using *pandas*, refer to the https://pandas.pydata.org/docs/.

GPU Data Loading with cuDF from Parquet

Objective: Demonstrate loading data from Parquet files using GPU-accelerated *cuDF* and converting to graph structures.

Code

```
import cudf
import graphistry

# cudf.DataFrame[['src', 'dst', ...]]
df = cudf.read_parquet('data.parquet')

# g._edges: df[['src', 'dst', ...]]
g = graphistry.edges(df, 'src', 'dst')
```

Explanation:

This example showcases how to load data from a Parquet file using *cuDF* and convert it into a graph structure with **GFQL**. For further details on using *cuDF*, refer to the official https://docs.rapids.ai/api/cudf/stable/ documentation.

Bind Nodes and Edges

Objective: Show how to convert loaded data into graph structures using *.edges()* and *.nodes()* when both are available.

Code

```
# pd.DataFrame[['n_id', ...]]
df1 = pd.read_csv('nodes.csv')

# pd.DataFrame[['src', 'dst', ...]]
df2 = pd.read_csv('edges.csv')

# g._edges: df2[['src', 'dst', ...]]
# g._nodes: df1[['n_id', ...]] <-- optional
g = graphistry.edges(df2, 'src', 'dst').nodes(df1, 'n_id')
```

Explanation:

This example demonstrates how to bind graph data for nodes and edges using **GFQL**. The *graphistry.PlotterBase.PlotterBase.edges()* method is used to load edge data. Binding nodes data is optional, and via method *graphistry.PlotterBase.PlotterBase.nodes*.

Handle Multiple Node Columns with Hypergraphs

Objective: Discuss how to create creates from rows with multiple columns representing nodes via hypergraphs with the default *direct=False* parameter.

Code

```
g = graphistry.hypergraph(df, entity_types=['a', 'b', 'c'])['graph']
# g._node == 'nodeID'
# g._nodes: df[['nodeTitle', 'type', 'category', 'nodeID', 'a', 'b', 'c', 'd', 'e',
↪'EventID']]
# g._source == 'attribID'
# g._destination == 'EventID'
# g._nodes.type.unique() == ['a', 'b', 'c', 'EventID']
# g._edges: df[['EventID', 'attribID', 'a', 'd', 'e', 'c', 'edgeType']]
```

Explanation:

This example explains how to shape graph data into a hypergraph format using the default *direct=False* parameter. In this case, all values in columns *a*, *b*, and *c* become nodes. Additionally, as *direct=False*, each row also becomes a node, with edges to its corresponding values in columns *a*, *b*, and *c*. When *direct=True*, the nodes for columns *a*, *b*, and *c* would be directly connected. Refer to *graphistry.PlotterBase.PlotterBase.hypergraph()* for more variants and advanced usage.

10.5.7.3 Common Graph Machine Learning and Graph AI Tasks

We'll cover a range of common tasks related to graph machine learning and AI you can do on GFQL results:

- *UMAP Cluster & Dimensionality Reduction for Embeddings & Similarity Graphs*
- *UMAP Fit/Transform for Scaling*
- *Anomaly Detection using RGCN Graph Neural Networks*
- *Cluster labeling with DBSCAN*
- *Automated Feature Generation*
- *Semantic Search in Graphs*
- *Knowledge Graph Embeddings*

UMAP Cluster & Dimensionality Reduction for Embeddings & Similarity Graphs

Objective: Show how to apply UMAP for dimensionality reduction, turning wide data into for clustering, embeddings, and similarity graphs.

Code

```
df = pd.DataFrame({
    'a': [0, 1, 2, 3, 4],
    'b': [10, 20, 30, 40, 50],
```

(continues on next page)

(continued from previous page)

```
    'c': [100, 200, 300, 400, 500]
})
g = graphistry.nodes(df).umap()  # Automatically featurizes & embeds into X, Y space
g.plot()

assert set(g._nodes.columns) == {'_n', 'a', 'b', 'c', 'x', 'y'}
assert set(g._edges.columns) == {'_src_implicit', '_dst_implicit', '_weight'}
assert isinstance(g._node_features, (pd.DataFrame, cudf.DataFrame))
```

Explanation:

This example demonstrates how to utilize *graphistry.umap_utils.UMAPMixin.umap()*. See its reference docs for many optional overrides and usage modes, such as defining *X=['col1', 'col2', ...]* to specify which columns to cluster.

UMAP Fit/Transform for Scaling

Objective: Explain how to use UMAP's fit/transform capabilities for scaling features across datasets.

Code

```
# Train: Feature columns X and label column y are optional
g1 = graphistry.nodes(df_sample).umap(X=['col_1', ..., 'col_n'], y='col_m')

# Transform new data under initial UMAP embedding
g2 = g1.transform_umap(batch_df, return_graph=True)

# Visualize new data under initial UMAP embedding
g2.plot()
```

Explanation:

This example illustrates how to fit a UMAP model on one dataset and then use that model to transform another dataset, enabling consistent scaling of features. For more details on using fit/transform with UMAP, consult the *graphistry.umap_utils.UMAPMixin.umap()* documentation.

Anomaly Detection using RGCN Graph Neural Networks

Objective: Introduce the basic concepts of RGCNs and how to build and train a simple graph model.

Code

```
# df: df[['src_ip', 'dst_ip', 'o1', 'dst_port', ...]]

g = graphistry.edges(df, 'src_ip', 'dst_ip')  # graph
g2 = g.embed(  # rerun until happy with quality
    device=dev0,

    #relation='dst_port', # always 22, so runs as a GCN instead of RGCN
```

(continues on next page)

(continued from previous page)

```
    relation='o1', # split by sw type

    #==== OPTIONAL: NODE FEATURES ====
    #requires node feature data, ex: g = graphistry.nodes(nodes_df, node_id_col).edges(..
    #use_feat=True
    #X=[g._node] + good_feats_col_names,
    #cardinality_threshold=len(g._edges)+1, #optional: avoid topic modeling on high-
↪cardinality cols
    #min_words=len(g._edges)+1, #optional: avoid topic modeling on high-cardinality cols

    epochs=10
)

def to_cpu(tensor, is_gpu=True):
    return tensor.cpu() if is_gpu else tensor

score2 = pd.Series(to_cpu(g2._score(g2._triplets)).numpy())

# df[['score', 'is_low_score', ...]]
df.assign(
    score=score2,
    is_low_score=(score2 < (score2.mean() - 2 * score2.std()))
)
```

Explanation:

This example provides an introduction to building and training a basic Relational Graph Convolutional Network (RGCN) using *graphistry.embed_utils.HeterographEmbedModuleMixin.embed()*. See the *SSH logs RGCN demo notebook* for more on this example.

Cluster labeling with DBSCAN

Objective: Discuss using DBSCAN for clustering nodes or edges based on features.

Code

```
g2 = g1.umap().dbscan(eps=0.5, min_samples=5)  # Apply DBSCAN clustering
print('labels: ', g2._nodes['_dbscan'])
```

Explanation:

This example illustrates how to apply DBSCAN clustering using `graphistry.compute.cluster.ClusterMixin.dbscan()` to label graph data after reducing dimensionality with UMAP.

Automated Feature Generation

Objective: Illustrate generating features from raw data for AI applications.

Code

```
g = graphistry.nodes(df).featurize(kind='nodes', X=['raw_feature_1', 'raw_feature_2'])
```

Explanation:

This example demonstrates how to automatically generate features from raw data returned by GFQL queries for use with ML and AI using *graphistry.feature_utils.FeatureMixin.featurize()* methods. Paramter *feature_engine=* (*graphistry.feature_utils.FeatureEngine*) selects the feature generation engine.

Semantic Search in Graphs

Objective: Implement semantic search using graph embeddings and natural language queries.

Code

```
g2 = g1.featurize(
    X = ['text_col_1', 'text_col_n'],  # ... more columns
    kind='nodes',
    model_name = "paraphrase-MiniLM-L6-v2")

results_df, query_vector = g2.search('my natural language query => df hits', ...)

g3 = g2.search_graph('my natural language query => graph', ...)
g3.plot()
```

Explanation:

This example showcases how to perform semantic searches within graph data using embeddings. For further details on implementing semantic search, see the **Semantic Search** section in our documentation.

Knowledge Graph Embeddings

Objective: Explain training models for knowledge graph embeddings and predicting relationships.

Code

```
g2 = g1.embed(relation='relationship_column_of_interest')

g3 = g2.predict_links_all(threshold=0.95)  # score high confidence predicted edges
g3.plot()

# Score over any set of entities and/or relations.
g4 = g2.predict_links(
    source=['entity_k'],
    relation=['relationship_1', 'relationship_4'],  # ... more relationships
```

(continues on next page)

(continued from previous page)

```
    destination=['entity_1', 'entity_m'],  # ... more entities
    threshold=0.9,  # score threshold
    return_dataframe=False)  # return graph vs _edges df
```

Explanation:

This example describes how to train models for knowledge graph embeddings with **GFQL** and how to predict relationships between entities. Shows using both *graphistry.embed_utils.HeterographEmbedModuleMixin.embed()*, *graphistry.embed_utils.HeterographEmbedModuleMixin.predict_links_all()*, and *graphistry.embed_utils.HeterographEmbedModuleMixin.predict_links()*.

10.5.8 GFQL Quick Reference

GFQL is the first fully vectorized dataframe-native graph query language with an open-source GPU runtime. This quick reference page provides short examples of various parameters and usage patterns.

If you are new to Cypher: Cypher is a graph query language popularized by Neo4j and related tools. It uses ASCII-art graph patterns such as `(n1)-[e1]->(n2)` to describe node-edge-node traversals, so GFQL docs use that notation as a familiar shorthand when discussing Cypher syntax through `g.gfql("MATCH ...")`.

10.5.8.1 Basic Usage

Chaining Operations

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

`gfql` sequences multiple matchers for more complex patterns of paths and subgraphs

- **query**: Sequence of graph node/edge matchers and optional row-pipeline call steps (for example, *rows()*, *where_rows()*, *return_()*, *order_by()*, *limit()*), or an equivalent GFQL chain object.
- **engine**: Optional execution engine. Engine is typically not set, defaulting to *'auto'*. Use *'cudf'* for GPU acceleration and *'pandas'* for CPU.

Native GFQL chains are typed Python inputs. Pass the list, dict envelope, or `Chain` object itself; strings passed to `g.gfql(...)` are interpreted as query text for the selected string language.

Use this page as a quick MATCH/chain reference. For row-pipeline RETURN semantics, see *GFQL RETURN (Row Pipelines)*.

10.5.8.2 Choose The Right Entrypoint

- Use `g.gfql([...])` for native GFQL operators and `g.gfql("MATCH ...")` for Cypher syntax on the current graph.
- If a tool receives a serialized native chain such as `"[{'type': ...}]"`, decode it into a real Python list/dict/`Chain` before calling `g.gfql(...)`. `g.gfql(str)` is reserved for Cypher query text by default.
- Use `g.gfql_remote([...])` for remote GFQL when the dataset size or hardware profile calls for remote execution, including remote GPU execution. See *GFQL Remote Mode*.

> **Warning**
>
> *graphistry.cypher("...")* and *g.cypher("...")* are a separate remote database Cypher path (for example, Neo4j/Neptune integrations), not the GFQL execution surface summarized on this page.

10.5.8.3 Graph State Vs Row State

- **Graph state** keeps a traversable graph in *_nodes* and *_edges*. Matchers, graph-preserving *call(...)* transforms, *let()* / *ref()* graph DAG stages, local Cypher *CALL graphistry.*.write()* queries, and local Cypher *GRAPH { MATCH ... }* constructors stay in graph state. (*GRAPH { }* is a GFQL extension — see *Cypher Syntax In GFQL* for details.)
- **Row state** stores tabular results in *_nodes* and uses an empty placeholder *_edges* frame. Row-pipeline steps such as *rows()*, *with_()*, *select()*, *return_()*, *group_by()*, and row-returning local Cypher *CALL ... YIELD ... RETURN ...* queries move into row state.
- A bare local Cypher procedure call without *.write()* also moves into row state. For example, *CALL graphistry.degree()* projects its default output columns into *_nodes* and clears *_edges*.
- If you want to enrich a graph and keep matching locally, use a graph-preserving *call()* / *let()* pattern or a bare local Cypher *CALL graphistry.*.write()*. The local Cypher compiler currently supports *graphistry.degree.write()* plus *graphistry.igraph.<alg>.write()* and *graphistry.cugraph.<alg>.write()* for algorithms exposed through *compute_igraph()* / *compute_cugraph()*, along with a curated NetworkX subset including *graphistry.nx.pagerank.write()*, *graphistry.nx.betweenness_centrality.write()*, *graphistry.nx.degree_centrality.write()*, *graphistry.nx.closeness_centrality.write()*, *graphistry.nx.eigenvector_centrality.write()*, *graphistry.nx.katz_centrality.write()*, *graphistry.nx.connected_components.write()*, *graphistry.nx.strongly_connected_components.write()*, *graphistry.nx.core_number.write()*, *graphistry.nx.hits.write()*, *graphistry.nx.edge_betweenness_centrality.write()*, and *graphistry.nx.k_core.write()*.

10.5.8.4 Cypher Strings Through `g.gfql()`

Use `g.gfql("MATCH ...")` when you want Cypher syntax and declarative graph semantics on a bound graph instead of writing the equivalent GFQL chain by hand:

Do not pass stringified Python or JSON native-chain literals to `g.gfql(...)`. Materialize those serialized values before calling GFQL, or emit Cypher text intentionally.

```
result = g.gfql(
    "MATCH (n1:Person)-[e1:FOLLOWS]->(n2:Person) "
    "RETURN n1.name AS source_name, n2.name AS target_name "
    "ORDER BY source_name, target_name "
    "LIMIT $top_n",
    params={"top_n": 5},
)

result._nodes
```

For the dedicated guide, helper APIs, and direct-vs-translation guidance, see *Cypher Syntax In GFQL*.

10.5.8.5 Node Matchers

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

`n` matches nodes based on their attributes.

- 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'")
```

10.5.8.6 Edge Matchers

```
e_forward(edge_match=None, hops=1, min_hops=None, max_hops=None, output_min_hops=None,␣
↪output_max_hops=None, label_node_hops=None, label_edge_hops=None, label_seeds=False,␣
↪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, min_hops=None, max_hops=None, output_min_hops=None,␣
↪output_max_hops=None, label_node_hops=None, label_edge_hops=None, label_seeds=False,␣
↪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, min_hops=None, max_hops=None, output_min_hops=None,
↪ output_max_hops=None, label_node_hops=None, label_edge_hops=None, label_seeds=False,␣
↪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, min_hops=None, max_hops=None, output_min_hops=None, output_
↪max_hops=None, label_node_hops=None, label_edge_hops=None, label_seeds=False, 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` matches edges based on their attributes (undirected). May also include matching on edge's source and destination nodes.

- 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.
- *min_hops*/*max_hops*: Inclusive traversal bounds (min defaults to 1 unless max_hops is 0; max defaults to *hops*).
- *output_min_hops*/*output_max_hops*: Optional post-filter slice; defaults keep all traversed hops up to *max_hops*.
- *label_node_hops*/*label_edge_hops*: Optional column names for hop numbers; *label_seeds=True* adds hop 0 for seeds.
- *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 up to 2 hops forward on edges where *status* is *'active'*:

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

- Traverse 2..4 hops but show only hops 3..4 with labels:

```
e_forward(
    {"status": "active"},
    min_hops=2,
    max_hops=4,
    output_min_hops=3,
    label_edge_hops="edge_hop"
)
```

- 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`, `e_forward`, and `e` are aliases.

- `e_reverse`: Same as `e_forward`, but traverses in reverse.
- `e`: Traverses edges regardless of direction.

10.5.8.7 Predicates

`graphistry.compute.predicates.ASTPredicate.ASTPredicate`

- 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"])})
```

10.5.8.8 Where (Same-Path Constraints)

Use *where* to relate attributes across named steps in a chain.

```
from graphistry import n, e_forward, col, compare

g.gfql(
    [
        n({"type": "account"}, name="a"),
        e_forward(name="e"),
        n({"type": "user"}, name="c"),
    ],
    where=[
        compare(col("a", "owner_id"), "==", col("c", "owner_id")),
        compare(col("e", "org_id"), "==", col("a", "org_id")),
    ],
)
```

compare() can relate node and edge columns when the column types align. Supported *compare(..., op, ...)* operators: ==, !=, <, <=, >, >=. WHERE works with *g.gfql([...], where=[...])*; *Chain(..., where=[...])* is the equivalent explicit form. Multiple WHERE comparisons are ANDed.

10.5.8.9 Row Pipelines (*MATCH ... RETURN* style)

Use row-pipeline operators to move from pattern matching to tabular Cypher-like *RETURN* processing.

```
from graphistry import n, e_forward, gt
from graphistry.compute import rows, where_rows, return_, order_by, limit

top_people = g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person", "score": gt(10)}, name="p"),
    rows(table="nodes", source="p"),
    where_rows(expr="score >= 50 AND name STARTS WITH 'A'"),
    return_(["id", "name", ("score_bucket", "score / 10")]),
    order_by([("score_bucket", "desc"), ("name", "asc")]),
    limit(25),
])

top_people._nodes
```

Notes:

- *rows(table="nodes" or table="edges", source="alias")* picks the active row table. *source* must be an alias introduced earlier via *name="..."* on a matcher. In the example above, *source="p"* refers to *n(..., name="p")*, so only rows matched by alias *p* are used.
- Edge example: *e_forward(..., name="e")* followed by *rows(table="edges", source="e")* scopes rows to edges matched as *e*.
- If *source* is omitted (for example, *rows(table="nodes")*), the full active nodes/edges table is used.
- *return_(["col"])* is shorthand for *return_([("col", "col")])*.
- *with_(...)* and *select(...)* share projection semantics with *return_(...)*:

  ```
  from graphistry.compute import rows, with_, select, return_

  # Equivalent projections
  rows(table="nodes", source="p")
  return_(["id", ("score2", "score * 2")])
  with_(["id", ("score2", "score * 2")])
  select([("id", "id"), ("score2", "score * 2")])
  ```

  return_(["id"]), *with_(["id"])*, and *select([("id", "id")])* all project the same *id* column.
- *where_rows()* evaluates row expressions and filter dictionaries in a vectorized dataframe execution path (pandas/cuDF engines).
- In *where_rows(expr="...")*, supported comparison operators are =, !=, <>, <, <=, >, >=.

10.5.8.10 Combined Examples

- **MATCH with same-path WHERE constraints:**

```
from graphistry import n, e_forward, col, compare

g.gfql(
    [
        n({"type": "account"}, name="a"),
        e_forward({"status": "active"}, name="e"),
        n({"type": "user"}, name="u"),
    ],
    where=[compare(col("a", "org_id"), "==", col("u", "org_id"))],
)
```

- **MATCH then RETURN (row pipeline):**

```
from graphistry import n, e_forward, gt
from graphistry.compute import rows, where_rows, return_, order_by, limit

g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person", "score": gt(0)}, name="p"),
    rows(table="nodes", source="p"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(10),
])
```

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

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

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

```
g.gfql([
    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.gfql([
    n({"status": "infected"}),
    e_forward(to_fixed_point=True),
```

(continues on next page)

(continued from previous page)

```
    n(name="reachable")
])
```

- **Filter by multiple conditions:**

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

- **Use custom queries in matchers:**

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

10.5.8.11 GPU Acceleration

- **Enable GPU mode:**

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

10.5.8.12 Remote Mode

- **Query existing remote data**

```
g = graphistry.bind(dataset_id='ds-abc-123')

nodes_df = g.gfql_remote([n()])._nodes
```

- **Upload graph and run GFQL**

```
g2 = g1.upload()

g3 = g2.gfql_remote([n(), e(), n()])
```

- **Enforce CPU and GPU mode on remote GFQL**

```
g3a = g2.gfql_remote([n(), e(), n()], engine='pandas')
g3b = g2.gfql_remote([n(), e(), n()], engine='cudf')
```

- **Return only nodes and certain columns**

```
cols = ['id', 'name']
g2b = g1.gfql_remote([n(), e(), n()], output_type="edges", edge_col_subset=cols)
```

- **Return only edges and certain columns**

```
cols = ['src', 'dst']
g2b = g1.gfql_remote([n(), e(), n()], output_type="edges", edge_col_subset=cols)
```

- **Return only shape metadata**

```
shape_df = g1.chain_remote_shape([n(), e(), n()])
```

- **Run remote Python and get back a graph**

```
def my_remote_trim_graph_task(g):
    return (g
        .nodes(g._nodes[:10])
        .edges(g._edges[:10])
    )

g2 = g1.upload()
g3 = g2.python_remote_g(my_remote_trim_graph_task)
```

- **Run remote Python and get back a table**

```
def first_n_edges(g):
    return g._edges[:10]

some_edges_df = g.python_remote_table(first_n_edges)
```

- **Run remote Python and get back JSON**

```
def first_n_edges(g):
    return g._edges[:10].to_json()

some_edges_json = g.python_remote_json(first_n_edges)
```

- **Run remote Python and ensure runs on CPU or GPU**

```
g3a = g2.python_remote_g(my_remote_trim_graph_task, engine='pandas')
g3b = g2.python_remote_g(my_remote_trim_graph_task, engine='cudf')
```

- **Run remote Python, passing as a string**

```
g2 = g1.upload()

# ensure method is called "task" and takes a single argument "g"
g3 = g2.python_remote_g("""
    def task(g):
```

(continues on next page)

(continued from previous page)

```
        return (g
            .nodes(g._nodes[:10])
            .edges(g._edges[:10])
        )
""")
```

10.5.8.13 Let Bindings and DAG Patterns

Use Let bindings to create directed acyclic graph (DAG) patterns with named operations. Lists are treated as implicit Chains.

- **Basic Let with named bindings:**

```
from graphistry import let, ref, n, e_forward, gt

result = g.gfql(let({
    'suspects': n({'risk_score': gt(80)}),
    'connections': [
        n({'risk_score': gt(80)}),
        e_forward({'type': 'transaction'}),
        n()
    ]
}))

# Access results by name
suspects = result._nodes[result._nodes['suspects']]
connections = result._edges[result._edges['connections']]
```

- **Complex DAG with multiple references:**

```
from graphistry import let, ref, n, e_forward, gt

result = g.gfql(let({
    'high_value': n({'balance': gt(100000)}),
    'large_transfers': [
        n({'balance': gt(100000)}),
        e_forward({'type': 'transfer', 'amount': gt(10000)}),
        n()
    ],
    'suspicious': ref('large_transfers', [
        n({'created_recent': True, 'verified': False})
    ])
}))
```

10.5.8.14 Call Operations

Run graph algorithms like PageRank, community detection, and layouts directly within your GFQL queries:

- **Compute PageRank:**

```
from graphistry import call, let, ref, n, e

# Use let() to compose filter + enrichment
result = g.gfql(let({
    'persons': [n({'type': 'person'}), e(), n()],
    'ranked': ref('persons', [call('compute_cugraph', {'alg': 'pagerank', 'damping':
↪ 0.85})])
}))

# Results have pagerank column
top_nodes = result._nodes.sort_values('pagerank', ascending=False).head(10)
```

- **Enrich a graph, then keep matching:**

```
g_enriched = g.gfql("CALL graphistry.degree.write()")
assert not g_enriched._edges.empty
top_degree = g_enriched.gfql(
    "MATCH (n) WHERE n.degree >= 2 RETURN n.id AS id, n.degree AS degree ORDER BY␣
↪degree DESC LIMIT 10"
)
```

  Local note: a bare *g.gfql("CALL graphistry.*.write()")* stays in graph state and can feed later *MATCH* queries. *g.gfql("CALL ... YIELD ... RETURN ...")* still targets row-returning procedure flows.

- **Return procedure rows instead of an enriched graph:**

```
degree_rows = g.gfql("CALL graphistry.degree()")
assert degree_rows._edges.empty

# Row state: _nodes has nodeId/degree columns and _edges is empty
degree_rows._nodes
```

- **Community detection with Louvain:**

```
from graphistry import call, let, ref, n, e_forward

# Use let() to compose traversal + community detection
result = g.gfql(let({
    'reachable': [n({'active': True}), e_forward(to_fixed_point=True), n()],
    'communities': ref('reachable', [call('compute_cugraph', {'alg': 'louvain'})])
}))

# Results have community column
communities = result._nodes.groupby('community').size()
```

- **Filter and compute within Let:**

```
from graphistry import call, let, ref, n, e, gt
```

(continues on next page)

(continued from previous page)

```
# Split mixed chain into separate bindings
result = g.gfql(let({
    'suspects': [n({'flagged': True}), e(), n()],
    'ranked': ref('suspects', [
        call('compute_cugraph', {'alg': 'pagerank'})
    ]),
    'influencers': ref('ranked', [
        n({'pagerank': gt(0.01)})
    ])
}))
```

- **Apply layout algorithms:**

```
from graphistry import call, let, ref, n, e_forward, is_in

# Use let() to compose traversal + layout
result = g.gfql(let({
    'entities': [n({'type': is_in(['person', 'company'])}), e_forward(), n()],
    'positioned': ref('entities', [call('fa2_layout', {'iterations': 100})])
}))

# Results have x, y coordinates for visualization
result.plot()
```

Tip: For subset-based coloring after GFQL, use `result.collections(...)` and see *Layout Settings & Visualization Embedding*.

10.5.8.15 Remote Graph References

Reference graphs on remote servers for distributed computing:

- **Basic remote reference:**

```
from graphistry.compute import remote

result = g.gfql([
    remote(dataset_id='fraud-network-2024'),
    n({'risk_score': gt(90)}),
    e_forward()
])
```

- **Combine remote and local data in Let:**

```
from graphistry.compute import remote

result = g.gfql(let({
    'remote_data': remote(dataset_id='historical-2023'),
    'high_risk': ref('remote_data', [
        n({'risk_score': gt(95)})
    ]),
    'connections': ref('remote_data', [
        n({'risk_score': gt(95)}),
```

(continues on next page)

(continued from previous page)

```
        e_forward({'type': 'transaction'}),
        n()
    ])
}))
```

10.5.8.16 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"
)
```

10.5.8.17 Parameter Summary

- **Common Parameters:**
  - *filter_dict*: Attribute filters (e.g., *{"status": "active"}*)
  - *query*: Custom query string (e.g., *"age > 30"*)
  - *hops*: Max hops to traverse (shorthand for *max_hops*, 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.)

10.5.8.18 Traversal Directions

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

10.5.8.19 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.

10.5.8.20 Examples at a Glance

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

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

- **Custom edge and node queries:**

```
g.gfql([
    n(query="age >= 18"),
    e_forward(edge_query="interaction == 'message'"),
```

(continues on next page)

(continued from previous page)

```
    n(query="location == 'NYC'")
])
```

10.5.9 Cypher Syntax In GFQL

PyGraphistry supports a read-only Cypher surface directly through GFQL on a bound graph. This is the on-ramp for Cypher users who want familiar declarative syntax and graph-pattern semantics, but executed by GFQL's fully vectorized columnar engine and open-source GPU runtime instead of a database-only runtime. Start here when you want to execute a Cypher query through `g.gfql("MATCH ...")` instead of translating it into native GFQL operators by hand.

10.5.9.1 Choose The Right Cypher Entrypoint

- Use `g.gfql("MATCH ...")` or `g.gfql("...", language="cypher")` for Cypher syntax on the current bound `Plottable`.
- Use `g.gfql_remote([...])` for remote GFQL execution when the dataset size or hardware profile calls for running GFQL on remote infrastructure.
- Use `graphistry.cypher("...")` or `g.cypher("...")` for remote database Cypher over Bolt/Neo4j-style integrations. That is a separate execution path.

10.5.9.2 Quickstart

Assume `g` is a bound graph with nodes and edges already attached.

```
top_people = g.gfql(
    "MATCH (p:Person) "
    "RETURN p.name AS name "
    "ORDER BY name DESC "
    "LIMIT 5"
)

top_people._nodes
```

String queries default to `language="cypher"`, so the explicit selector is usually optional:

```
same_result = g.gfql(
    "MATCH (p:Person) RETURN p.name AS name ORDER BY name DESC LIMIT 5",
    language="cypher",
)
```

10.5.9.3 Parameters

Use `params=...` instead of manual string interpolation:

```
limited = g.gfql(
    "MATCH (p:Person) "
    "RETURN p.name AS name "
    "ORDER BY name DESC "
    "LIMIT $top_n",
    params={"top_n": 2},
)

limited._nodes
```

10.5.9.4 Graph Constructors (`GRAPH { }`)

> **Note**
>
> `GRAPH { }` is a **GFQL extension** — not part of openCypher, and GQL's first edition (ISO/IEC 39075:2024) deferred graph constructors to a future revision. Standard Cypher and GQL both force query results through row/path-centric serialization. `GRAPH { }` closes that gap by keeping results in **graph state** (both `_nodes` and `_edges`), enabling composable graph-pipeline workflows.

Use `GRAPH { MATCH ... }` when you want the matched subgraph back in graph state instead of a row table:

```
subgraph = g.gfql(
    "GRAPH { "
    "  MATCH (seed)-[reach]-(nbr) "
    "  WHERE seed.degree >= $degree_cutoff "
    "}",
    params={"degree_cutoff": 10},
)

subgraph._nodes
subgraph._edges
```

Use `GRAPH g = GRAPH { ... }` to bind a named graph, then `USE g` to query it:

```
result = g.gfql(
    "GRAPH g1 = GRAPH { "
    "  MATCH (seed)-[reach]-(nbr) "
    "  WHERE seed.degree >= $degree_cutoff "
    "} "
    "USE g1 "
    "MATCH (n) "
    "RETURN n.id AS id, n.degree AS degree "
    "ORDER BY degree DESC LIMIT 10",
    params={"degree_cutoff": 10},
)
```

Multi-stage graph pipelines chain constructors:

```
g1 = g.gfql(
    "GRAPH { "
    "  MATCH (seed)-[reach]-(nbr) "
    "  WHERE seed.degree >= $degree_cutoff "
    "}",
    params={"degree_cutoff": 10},
)

g2 = g1.gfql("CALL graphistry.cugraph.pagerank.write()", engine="cudf")

g3 = g2.gfql(
    "GRAPH { "
    "  MATCH (core)-[halo]-(nbr) "
    "  WHERE core.pagerank >= $pagerank_cutoff "
    "}",
    params={"pagerank_cutoff": 0.25},
)
```

The graph constructor surface supports:

- `MATCH` and `WHERE` clauses inside `GRAPH { }`
- Graph-preserving `CALL graphistry.*.write()` inside `GRAPH { }`
- `GRAPH g = GRAPH { }` named bindings with lexical scoping
- `USE g` to switch the current graph (works both inside constructors and in the outer query)
- Variable scoping: pattern variables inside `GRAPH { }` do not leak

This enables single-expression pipelines that filter, enrich, and query:

```
result = g.gfql(
    "GRAPH g1 = GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 10 } "
    "GRAPH g2 = GRAPH { USE g1 CALL graphistry.degree.write() } "
    "USE g2 "
    "MATCH (n) RETURN n.id AS id, n.degree AS degree "
    "ORDER BY degree DESC LIMIT 10"
)
```

Inside `GRAPH { }`, only `MATCH`, `WHERE`, `USE`, and graph-preserving `CALL graphistry.*.write()` are allowed. Row-oriented clauses (`RETURN`, `ORDER BY`, `SKIP`, `LIMIT`, `DISTINCT`, `UNWIND`, `WITH`) and row-returning `CALL` (without `.write()`) are not allowed inside the constructor — use them in the outer query after `USE`.

10.5.9.5 Supported Cypher Surface Through `g.gfql()`

The current GFQL Cypher compiler intentionally supports a bounded read-only surface. At a high level, that includes:

- `MATCH` and a bounded `OPTIONAL MATCH` subset
- `WHERE`
- `RETURN` and `WITH`
- `ORDER BY`, `SKIP`, `LIMIT`, and `DISTINCT`
- `UNWIND`

- supported `UNION` / `UNION ALL` and `CALL graphistry.*` flows when executed directly through `g.gfql("...")`

For exact `RETURN` / `WITH` row semantics after pattern matching, see *GFQL RETURN (Row Pipelines)*. For same-path `WHERE` comparisons, see *GFQL WHERE (Same-Path Constraints)*.

10.5.9.6 Support Matrix

Query shape	Status	Notes
`MATCH` / `WHERE` / `RETURN` / `WITH` / `ORDER BY` / `SKIP` / `LIMIT` / `DISTINCT`	Supported	Core read-only Cypher-in-GFQL path.
`GRAPH { MATCH ... }` / `GRAPH g = ...` / `USE g`	Partial	**GFQL extension** (not in openCypher; GQL deferred graph constructors). Returns matched subgraph in graph state. Supports named bindings and scoped graph switching via USE.
`OPTIONAL MATCH`	Partial	Supported for a bounded Cypher-in-GFQL subset, not the full Cypher null-extension surface.
`UNWIND`	Partial	Supported at top level, after `MATCH`, in row-only pipelines, and in the narrow graph-backed `MATCH ... WITH collect([DISTINCT] alias) AS list UNWIND list AS alias MATCH ... RETURN` continuation shape, but not in arbitrary graph/row interleavings.
`UNION` / `UNION ALL` and `CALL graphistry.*`	Partial	Execute directly through `g.gfql("...")`. Helper translation to a single `Chain` is stricter.
Variable-length relationship patterns	Partial	Direct Cypher supports endpoint traversals such as `[*2]`, `[*1..3]`, `[*]`, and typed forms like `[:R*2..4]`; connected multi-relationship variable-length patterns; and bounded/exact/fixed-point variable-length `WHERE` pattern predicates in the current row-shaped subset. Path/list-carrier uses and unsupported path/row-shaping cases still fail fast.
`CREATE` / `DELETE` / `SET`	Not supported	GFQL's Cypher surface is read-only.
Multiple disconnected `MATCH` patterns and arbitrary joins	Not supported	Split the work into separate GFQL / dataframe steps.
Full Cypher expression and function surface in row expressions	Partial	The current row-expression subset is intentionally smaller than full Cypher; finish advanced logic in pandas/cuDF when needed.

10.5.9.7 Supported Syntax Forms

The matrix above is clause-level. This section lists the main user-visible syntax forms on the current Cypher-in-GFQL surface.

Pattern Matching Forms

- Single-pattern `MATCH` queries with node aliases, relationship aliases, inline property maps, and top-level `params=...` binding.
- Node labels and multi-label node patterns such as `(p:Person:Admin)`.
- Relationship direction forms `->`, `<-`, and undirected `-[]-`.
- Relationship type alternation such as `[r:KNOWS|HATES]`.
- Single variable-length relationship patterns, including `[*n]`, `[*m..n]`, `[*]`, and typed forms such as `[:R*2..4]`.
- Connected patterns that mix variable-length and fixed-length relationships, such as `MATCH (a)-[:R*2]->()-[:S]->(c) RETURN c`.
- Connected comma-separated patterns such as `MATCH (a)-[:A]->(b), (b)-[:B]->(c)`.
- Repeated `MATCH` clauses when they stay connected through shared aliases.
- Path variable binding such as `MATCH p = (n)-[r]->(b)` when the path variable itself is not the projected output.

WHERE Forms

- Literal and parameter comparisons on node and edge properties.
- Same-path alias comparisons such as `WHERE p.team = q.team`.
- `IS NULL` and `IS NOT NULL` predicates.
- String predicates `STARTS WITH`, `ENDS WITH`, and `CONTAINS`.
- Label predicates such as `WHERE b:Foo:Bar`.
- Relationship-type predicates such as `WHERE type(r) = 'KNOWS'`.
- Positive relationship-existence pattern predicates such as `WHERE (n)-[:R]->()` and variable-length existence checks such as `WHERE (n)-[*]-()` and `WHERE (n)-[:R*2]->()`.
- Pattern predicates can be combined with row predicates in the current boolean subset, including `AND` / `OR` / `XOR` and `NOT` forms.

Variable-Length Relationship Boundary

Direct Cypher multihop support remains intentionally bounded. The supported direct forms include endpoint traversals, connected multi-relationship patterns, and variable-length `WHERE` pattern predicates where the result stays in the current row-shaping subset, for example:

- `MATCH (a)-[*2]->(b) RETURN b`
- `MATCH (a)-[:R*1..3]->(b) RETURN b`
- `MATCH (a)<-[*2]-(b) RETURN b`
- `MATCH (a)-[:R*1..2]-(b) RETURN b`
- `MATCH (a)-[:R*2]->(b)-[:S]->(c) RETURN c`
- `MATCH (a)-[:R]->(b), (b)-[:S*1..2]->(c) RETURN a.id AS a_id, c.id AS c_id`
- `MATCH (n) WHERE (n)-[:R*2]->() RETURN n`
- `MATCH (n) WHERE NOT (n)-[:R*2]->() RETURN n.id AS id`

The current compiler explicitly rejects these remaining subfamilies with `GFQLValidationError` instead of attempting unsound execution:

- path/list-carrier use of a variable-length relationship alias, such as `RETURN r` or `count(r)`
- shapes that still require unsupported path/relationship-carrier row shaping around a variable-length segment
- connected multi-pattern relationship-alias projection such as `RETURN r` / `r.prop` when it would require unsupported row shaping
- multi-alias `RETURN *` projections that would require unsupported path/multi-source row shaping

Row And Row-Pipeline Forms

- Top-level row-only queries such as `RETURN 1 AS x`.
- `RETURN` / `WITH` projections with aliases, `RETURN *`, `DISTINCT`, `ORDER BY`, `SKIP`, and `LIMIT`.
- Terminal `WITH` queries and multiple `WITH` stages.
- `WITH ... WHERE` row filtering.
- Aggregation/grouping via Cypher projection semantics, including `count`, `count(DISTINCT ...)`, `collect`, `collect(DISTINCT ...)`, `sum`, `max`, and `size(...)`.
- Top-level `UNWIND ... RETURN ...` queries.
- Mixed graph/row queries such as `MATCH ... UNWIND ... RETURN ...`.
- Connected multi-alias scalar projection such as `MATCH (a)-[:R]->(b), (b)-[:S]->(c) RETURN a.id AS a_id, c.id AS c_id`.
- The bounded `MATCH ... WITH ... MATCH ... RETURN` re-entry shape, including connected suffix projections in the current supported row-binding subset.

Procedure And Multi-Branch Forms

- Direct execution of `UNION` row queries through `g.gfql("...")`.
- Direct execution of `UNION ALL` row queries through `g.gfql("...")`.
- Direct execution of row-returning `CALL graphistry.*` procedures, including:
  - `CALL graphistry.degree()`
  - `CALL graphistry.degree YIELD nodeId RETURN nodeId`
  - `CALL graphistry.igraph.pagerank() YIELD nodeId, pagerank RETURN nodeId`
  - `CALL graphistry.cugraph.louvain()`
  - `CALL graphistry.cugraph.edge_betweenness_centrality()`
- Direct execution of standalone graph-preserving `CALL graphistry.*.write()` procedures, including:
  - `CALL graphistry.degree.write()`
  - `CALL graphistry.igraph.pagerank.write()`
  - `CALL graphistry.cugraph.edge_betweenness_centrality.write()`
  - `CALL graphistry.cugraph.k_core.write()`
  - `CALL graphistry.igraph.spanning_tree.write()`
  - `CALL graphistry.nx.edge_betweenness_centrality.write()`
  - `CALL graphistry.nx.k_core.write()`
  - `CALL graphistry.nx.pagerank.write()`
- Bare procedures without `.write()` stay row-returning even when you omit `YIELD ... RETURN ...`. For example, `CALL graphistry.degree()` projects its default outputs into `_nodes` and leaves an empty placeholder `_edges` frame (for example, `assert result._edges.empty`); use `.write()` when you want enrich-then-`MATCH` graph workflows with traversable edges (for example, `assert not result._edges.empty`).
- For `graphistry.igraph.<alg>()` and node-oriented `graphistry.cugraph.<alg>()`, row mode uses `nodeId` plus the algorithm output columns (for example, `pagerank` or `louvain`). For edge-oriented `graphistry.cugraph.<alg>()`, row mode uses `source` / `destination` plus the edge result columns. Topology-returning procedures such as `graphistry.cugraph.k_core()` or `graphistry.igraph.spanning_tree()` require `.write()`.
- `graphistry.nx.*` remains a curated CPU subset rather than the full NetworkX API, but it includes common node, edge, multi-output, and graph-returning forms:
  - `graphistry.nx.pagerank()` / `.write()`
  - `graphistry.nx.betweenness_centrality()` / `.write()`
  - `graphistry.nx.degree_centrality()` / `.write()`
  - `graphistry.nx.closeness_centrality()` / `.write()`
  - `graphistry.nx.eigenvector_centrality()` / `.write()`
  - `graphistry.nx.katz_centrality()` / `.write()`
  - `graphistry.nx.connected_components()` / `.write()`
  - `graphistry.nx.strongly_connected_components()` / `.write()`
  - `graphistry.nx.core_number()` / `.write()`

  - `graphistry.nx.hits()` / `.write()`
  - `graphistry.nx.edge_betweenness_centrality()` / `.write()`
  - `graphistry.nx.k_core.write()`

  Node calls use `nodeId` + the algorithm column, edge calls use `source` / `destination` + the algorithm column, and topology-returning calls such as `k_core` require `.write()`. Multi-output `hits` returns `nodeId`, `hubs`, and `authorities`.

  The same curated NetworkX algorithm subset is available from regular Python as `g.compute_networkx(...)` for users who do not need the local Cypher `CALL` path.

- Local Cypher `CALL` options accept one optional map argument. The top-level keys mirror `compute_igraph()` / `compute_cugraph()` options such as `out_col`, `directed`, `kind`, `use_vids`, and `params`; any extra keys are forwarded into the nested algorithm `params` dictionary.

Component-labeling examples:

```
# Graph-enrichment mode (keeps traversable _edges)
g.gfql(
    "CALL graphistry.cugraph.connected_components.write({out_col: 'wcc_id', directed:␣
↪false})",
    language="cypher",
)
g.gfql(
    "CALL graphistry.cugraph.strongly_connected_components.write({out_col: 'scc_id',␣
↪directed: true})",
    language="cypher",
)

# Row mode (no .write): returns nodeId + output column rows
g.gfql(
    "CALL graphistry.cugraph.connected_components({out_col: 'wcc_row', directed: false})
↪",
    language="cypher",
)
g.gfql(
    "CALL graphistry.nx.connected_components({directed: false})",
    language="cypher",
)
```

- Outside that curated `networkx` subset, `graphistry.nx.*` is not part of the current local Cypher `CALL` surface.
- `cypher_to_gfql()` stays stricter than direct execution and intentionally rejects `UNION` / `UNION ALL` and row-returning `CALL` flows because they are not representable as a single GFQL `Chain`.

Expression Families

- Arithmetic, boolean, comparison, and null-propagation expressions.
- `CASE` expressions.
- Graph introspection helpers such as `labels()`, `type()`, `keys()`, and `properties()`.
- Dynamic graph property lookup such as `n['name']` and `n[$idx]`.
- List predicates such as `all(...)`, `any(...)`, `none(...)`, and `single(...)`.
- Temporal constructors and operations over `date`, `time`, `datetime`, `localtime`, `localdatetime`, and `duration` in the current vectorized subset.

Bounded / Partial Forms

- `OPTIONAL MATCH` works for a bounded subset, including top-level and bound optional rows, but not the full Cypher null-extension surface.
- `UNWIND` works at top level, after `MATCH`, in row-only pipelines, and in the narrow graph-backed `MATCH ... WITH collect([DISTINCT] alias) AS list UNWIND list AS alias MATCH ... RETURN` continuation shape, but not in arbitrary graph/row interleavings.
- `MATCH ... WITH ... MATCH ... RETURN` is limited to the bounded single re-entry shape. Connected suffix projections with whole-row and carried scalar bindings are supported in the current subset, but this still does not generalize to arbitrary re-entry plans.

Not Supported Today

- Variable-length relationship aliases used as path/list carriers, such as `RETURN r` or `count(r)`.
- Connected multihop shapes that still require unsupported path/relationship-carrier row shaping.
- Multiple disconnected `MATCH` patterns used as arbitrary joins.
- Multi-pattern re-entry shapes beyond the bounded single `MATCH ... WITH ... MATCH ... RETURN` form.
- `RETURN *` after unsupported re-entry shapes or when it would require unsupported multi-alias/path projection shaping.
- `CREATE`, `DELETE`, `SET`, and other write clauses.
- Generic database procedures outside `CALL graphistry.*`.
- The full Cypher expression/function surface.

10.5.9.8 Validation And Unsupported Shapes

- Unsupported but syntactically valid query shapes on this Cypher surface raise `GFQLValidationError`, usually before execution starts.
- Invalid Cypher syntax raises `GFQLSyntaxError`.
- Passing a string query with an unsupported `language=...` selector also raises `GFQLValidationError`.

That fail-fast behavior is intentional: the current GFQL Cypher compiler prefers explicit validation over silently returning wrong rows.

10.5.9.9 Static Validation / Preflight Check

If you want to know whether a query fits the current Cypher-in-GFQL subset before execution, start with the bound-graph inline preflight APIs:

```
g.gfql_validate(
    "MATCH (p) RETURN p.name AS name ORDER BY name DESC LIMIT $top_n",
    params={"top_n": 5},
    # strict=True is the default for local bound-graph preflight
)

# On failure:
# - GFQLSyntaxError for invalid syntax
# - GFQLValidationError for unsupported/scheme-invalid shapes
```

- Use `g.gfql_validate(...)` when you want a stable validate-only entrypoint that never executes query operators and raises structured exceptions on invalid queries.
- Use `g.gfql(..., validate=True)` when you want execution guarded by a local preflight check. For Cypher strings, this uses schema-aware strict preflight by default.
- Use `g.gfql_remote(..., validate=True)` when you want remote execution guarded by local preflight before upload/network dispatch. For Cypher strings, remote preflight uses `strict=False` by default because remote schema is authoritative.
- Use `parse_cypher()` when you only want grammar validation and access to the parsed representation.
- Use `compile_cypher()` when you need low-level compiler/lowering output for tooling or whitebox inspection.
- Use `cypher_to_gfql()` only when you specifically need a single GFQL `Chain`. It is intentionally stricter than direct execution through `g.gfql("...")`.

Low-level helper example:

```
from graphistry.compute.exceptions import GFQLSyntaxError, GFQLValidationError
from graphistry.compute.gfql.cypher import parse_cypher, compile_cypher

query = "MATCH (p:Person) RETURN p.name AS name"

try:
    parsed = parse_cypher(query)    # grammar + AST checks
    compiled = compile_cypher(query)  # compiler/lowering checks
except GFQLSyntaxError as exc:
    print("Invalid Cypher syntax for g.gfql(\"MATCH ...\"):", exc)
except GFQLValidationError as exc:
    print("Valid Cypher, but outside the current GFQL Cypher surface:", exc)
```

10.5.9.10 Common Rewrites

- Need remote execution on Graphistry infrastructure instead of running against the current bound graph? Prefer `g.gfql_remote(...)` for remote GFQL, and keep `validate=True` (default) for local preflight before upload.
- Need remote database Cypher against Neo4j/Bolt-style backends instead of remote GFQL? Use `graphistry.cypher("...")` or `g.cypher("...")`.
- Need a pure GFQL chain object? Use `cypher_to_gfql()` when the query fits a single `Chain`.
- Need fixed-length, bounded, or fixed-point endpoint traversal? Direct Cypher already supports `[*2]`, `[*1..3]`, and `[*]` for that endpoint-only slice.
- Need aliasable intermediate hops, output slicing, or mixed connected-pattern multihop control? Rewrite in native GFQL with explicit hop bounds such as `e_forward(min_hops=1, max_hops=3)` or `e_forward(to_fixed_point=True)`, or unroll the traversal into explicit chain steps.
- Need write semantics or arbitrary joins? Keep Cypher syntax for the supported read-only part and finish the rest in a database or in pandas/cuDF.

10.5.9.11 Compiler Helper APIs

Use the helper APIs when you want to inspect or reuse compiler output rather than run the query immediately:

```
from graphistry.compute.gfql.cypher import (
    parse_cypher,
    compile_cypher,
    cypher_to_gfql,
    gfql_from_cypher,
)

parsed = parse_cypher("MATCH (p:Person) RETURN p.name AS name")
compiled = compile_cypher("MATCH (p:Person) RETURN p.name AS name")
chain = cypher_to_gfql("MATCH (p:Person) RETURN p.name AS name")
```

`cypher_to_gfql()` / `gfql_from_cypher()` are intentionally limited to queries that can be represented as a single GFQL `Chain`. If a query requires `UNION` or a row-returning `CALL` flow, execute it directly through `g.gfql("...", language="cypher")` instead.

See *GFQL Cypher Syntax API Reference* for the helper reference.

10.5.9.12 Translation Vs Direct Execution

This page is about **direct Cypher-syntax execution** through `g.gfql("MATCH ...")` on a bound graph.

- If you want to run the query now, use `g.gfql("MATCH ...")`.
- If you want to understand how Cypher maps into GFQL operators and wire protocol, use *Cypher to GFQL Python & Wire Protocol Mapping*.
- If you want native GFQL chain syntax instead of strings, start with *GFQL Quick Reference*.

10.5.10 GFQL WHERE (Same-Path Constraints)

WHERE adds constraints between named steps in a chain. Use it to relate attributes across the same path (for example, start.owner_id equals end.owner_id).

This page documents MATCH-stage *where=[...]* constraints. For RETURN-stage row filtering (*where_rows(...)*), see *GFQL RETURN (Row Pipelines)*.

10.5.10.1 Basic Usage

```
from graphistry import n, e_forward, col, compare

g_filtered = g.gfql(
    [
        n({"type": "account"}, name="a"),
        e_forward(name="e"),
        n({"type": "user"}, name="c"),
    ],
    where=[
        compare(col("a", "owner_id"), "==", col("c", "owner_id")),
        compare(col("e", "org_id"), "==", col("a", "org_id")),
    ],
)
g_filtered.plot()
```

Use *g.gfql([...], where=[...])* for WHERE. *Chain(..., where=[...])* is the equivalent explicit form. WHERE only applies to aliases in the chain (same-path scope), not to unrelated nodes elsewhere in the graph. All WHERE comparisons are ANDed (all must match).

Aliases come from *name=*. Column references use *alias.column*.

10.5.10.2 Boolean Semantics (*where=[...]*)

where is a Python list of comparison clauses. Commas in that list mean logical AND.

```
from graphistry import n, e_forward, col, compare

g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="b")],
    where=[
        compare(col("a", "org_id"), "==", col("b", "org_id")),  # AND
        compare(col("e", "risk"), ">=", col("a", "min_risk")),  # AND
    ],
)
```

- Supported now: conjunction (AND) across entries.
- Not supported yet in same-path WHERE: *OR*, *NOT*, grouping parentheses.

10.5.10.3 Comparator Surface (Same-Path WHERE)

compare(col(...), op, col(...)) supports these operators:

- *==*, *!=*, *<*, *<=*, *>*, *>=*

JSON wire format uses these names:

- *eq*, *neq*, *lt*, *le*, *gt*, *ge*

10.5.10.4 Why predicate helpers are not used in same-path *where*

Predicate helpers (for example *gt(10)*, *between(...)*, *isna()*) are single-column filters, and belong in *n({...}) / e_forward({...}) filter_dict`s or in `where_rows(filter_dict=...)*.

Same-path *where=[...]* is currently restricted to column-vs-column comparisons across aliases so the validator can statically verify aliases and columns before execution in both pandas and cuDF vectorized paths.

```
from graphistry import n, e_forward, col, compare, gt

# Good: single-step predicate helper
g.gfql([n({"score": gt(10)}, name="a"), e_forward(), n(name="b")])

# Good: cross-step column-vs-column comparison
g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="b")],
    where=[compare(col("a", "score"), ">", col("b", "score"))],
)

# Not supported in same-path WHERE (predicate helper inside compare)
g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="b")],
    where=[compare(col("a", "score"), ">", gt(10))],
)
# ValueError: where[...] must use StepColumnRef for left/right ...
```

10.5.10.5 When to use predicates vs WHERE

Predicates live inside *n(...)/e_forward(...)* filter dicts and apply to one step. WHERE compares fields across steps.

```
from graphistry import n, e_forward, col, compare, gt

# Single-step predicate (preferred when you only filter one entity)
g.gfql([n({"a": gt(10)}, name="n1"), e_forward(), n(name="n2")])

# Cross-step comparison (needs WHERE)
g.gfql(
    [n(name="n1"), e_forward(name="e1"), n(name="n2"), e_forward(name="e2"), n()],
    where=[
        compare(col("n1", "a"), ">", col("n2", "b")),
        compare(col("e1", "x"), "==", col("e2", "y")),
    ],
)
```

JSON wire format details live in *GFQL Wire Protocol Specification*. Supported operators: ==, !=, <, <=, >, >= (JSON uses *eq*, *neq*, *lt*, *le*, *gt*, *ge*).

Current scope:

- Same-path column-vs-column comparisons across named aliases
- AND semantics across *where=[...]* entries

In progress:

- Boolean composition (*OR*, *NOT*, grouping)
- Column-vs-literal comparisons
- Predicate/function expressions in *where*
- Computed expressions
- Cross-path/global constraints

10.5.10.6 Validation Behavior

WHERE is validated before same-path execution starts.

Validation checks include:

- **Alias bindings**: Every referenced alias must exist as a *name=* on a node or edge step in the chain.
- **Column visibility**: Each referenced column must exist on the visible schema at that step. This includes columns added by prior safelisted *call(...)* operations whose schema effects are known.
- **Clause shape**: In Python, each *where* entry must be a *compare(col(...), op, col(...))* object (or equivalent dict clause); in JSON, each entry must use exactly one operator key (*eq*, *neq*, *lt*, *le*, *gt*, *ge*) with string *left*/*right* values.

Common failures:

```
from graphistry import n, e_forward, col, compare

# Missing alias binding ("missing" was never introduced via name=)
g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="c")],
    where=[compare(col("missing", "x"), "==", col("c", "owner_id"))],
)
# ValueError: WHERE references aliases with no node/edge bindings: missing

# Missing column on an alias
g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="c")],
    where=[compare(col("a", "missing_col"), "==", col("c", "owner_id"))],
)
# ValueError: WHERE references missing column 'missing_col' on alias 'a' ...

# Invalid where entry type
g.gfql([n(name="a"), e_forward(name="e"), n(name="c")], where=[123])
# ValueError: where[0] must be a WhereComparison or dict clause ...
```

Advanced troubleshooting (opt-in): set environment variables *GRAPHISTRY_WHERE_VALIDATION_IGNORE_ERRORS* and *GRAPHISTRY_WHERE_VALIDATION_IGNORE_CALLS* to selectively suppress specific missing-column validation paths during migration/debugging.

WHERE can compare columns from node or edge steps when the types align. Null handling follows predicate semantics; use *isna()/notna()* in per-step filters when needed (for example, *n({"owner_id": notna()})*).

Use selective per-step filters in *n(...)/e_forward(...)* first; WHERE ties steps together and can be more expensive on dense graphs.

WHERE works with pandas and cuDF; select an engine via *g.gfql(..., engine='cudf')*. For full JSON schema details, see *GFQL Wire Protocol Specification*.

10.5.10.7 Row-Table Filtering with *where_rows(...)*

Use *where_rows(...)* when filtering the active row table selected by *rows(...)* in a *MATCH ... RETURN*-style pipeline.

```
from graphistry import n, e_forward
from graphistry.compute import rows, where_rows, return_

filtered = g.gfql([
    n(name="a"),
    e_forward(name="e"),
    n(name="b"),
    rows(table="nodes", source="b"),
    where_rows(expr="score >= 10 AND name CONTAINS 'alice'"),
    return_(["id", "name", "score"]),
])
```

where and *where_rows* solve different problems:

- *where=[...]*: same-path alias comparisons across chain steps.
- *where_rows(...)*: row-level filtering on the active table (nodes/edges).

where_rows accepts:

- *filter_dict={...}* predicate filters.
- *expr="..."* Cypher-like scalar expressions.
- both together (AND semantics).
- In *expr="..."*, comparison operators are =, *!=*, <>, <, <=, >, >=.
- For temporal/date-time row filtering, *filter_dict* uses the same predicate operators as MATCH filters (for example, *gt*, *ge*, *lt*, *le*, *eq*, *ne*, *between*).

Validation behavior:

- Expression forms outside the supported subset are rejected by validator/runtime.
- Column references are validated against the active row table.
- Execution stays vectorized on pandas/cuDF backends.

10.5.11 GFQL RETURN (Row Pipelines)

Use row-pipeline operators for Cypher-style *MATCH ... RETURN* flows after pattern matching.

10.5.11.1 Scope

- This page covers row-table operations: *rows*, *where_rows*, *with_*, *return_*, *select*, *order_by*, *skip*, *limit*, *distinct*, *unwind*, *group_by*.
- For same-path MATCH constraints, use *GFQL WHERE (Same-Path Constraints)* (*where=[...]*).

10.5.11.2 Minimal Example

```
from graphistry import n, e_forward, gt
from graphistry.compute import rows, where_rows, return_, order_by, limit

top_people = g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person", "score": gt(0)}, name="p"),
    rows(table="nodes", source="p"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(10),
])
```

10.5.11.3 Key Semantics

- *rows(table="nodes" or table="edges", source="alias")* selects the active row table.
- *source* must reference a prior matcher alias from *name="..."*.
- *where_rows(...)* filters the active row table (not chain aliases).
- *return_*, *with_*, and *select* use the same projection shape.

10.5.11.4 *rows(table=..., source=...)* in practice

```
from graphistry import n, e_forward
from graphistry.compute import rows, return_, order_by

# Node rows matched by alias "p"
people_rows = g.gfql([
    n({"type": "Person"}, name="p"),
    e_forward(name="r"),
    n(name="q"),
    rows(table="nodes", source="p"),
    return_(["id", "name", "score"]),
    order_by([("id", "asc")]),
])
```

(continues on next page)

(continued from previous page)

```
# Edge rows matched by alias "r"
edge_rows = g.gfql([
    n(name="a"),
    e_forward({"type": "FOLLOWS"}, name="r"),
    n(name="b"),
    rows(table="edges", source="r"),
    return_(["s", "d", "type", "weight"]),
])
```

- *table="nodes"* switches to node rows; *table="edges"* switches to edge rows.
- *source="p"* (or *"r"*) keeps only rows participating in that named matcher.
- If *source* is omitted (*rows(table="nodes")*), the full active table is used.
- For edge rows, replace *s*/*d* with your graph's configured edge endpoint column names.

10.5.11.5 *where_rows(expr="...")*: expression language

```
from graphistry import n
from graphistry.compute import rows, where_rows, return_

filtered = g.gfql([
    n({"type": "Person"}, name="p"),
    rows(table="nodes", source="p"),
    where_rows(expr="score >= 50 AND name STARTS WITH 'A' AND manager_id IS NOT NULL"),
    return_(["id", "name", "score", "manager_id"]),
])
```

- *expr* uses the GFQL row-expression parser (Cypher-like subset).
- Columns are referenced by active row-table column name (for example, *score*, *name*).
- Common operators: *AND*, *OR*, *NOT*, *=*, *!=*, *<>*, *<*, *<=*, *>*, *>=*, *IS NULL*, *IS NOT NULL*, *IN*, *CONTAINS*, *STARTS WITH*, *ENDS WITH*.
- For predicate helpers like *lt(...)*/*between(...)*, use *where_rows(filter_dict=...)*.
- Unsupported row expressions are rejected by validator/runtime.

10.5.11.6 *with_*, *select*, *return_*: same projection model

```
from graphistry import n
from graphistry.compute import rows, with_, where_rows, return_

ranked = g.gfql([
    n({"type": "Person"}, name="p"),
    rows(table="nodes", source="p"),
    with_([
        "id",                        # shorthand for ("id", "id")
        ("score2", "score * 2"),     # tuple is (output_name, expression)
        ("person_name", "name"),     # rename
```

(continues on next page)

(continued from previous page)

```
    ]),
    where_rows(expr="score2 >= 100"),
    return_(["id", "person_name", "score2"]),
])
```

```
from graphistry import n
from graphistry.compute import rows, select

projected = g.gfql([
    n({"type": "Person"}, name="p"),
    rows(table="nodes", source="p"),
    select([
        ("person_id", "id"),
        ("score2", "score * 2"),
    ]),
])
```

with_(...), *select(...)*, and *return_(...)* all accept:

- Shorthand string: *"col"* means *("col", "col")*.
- Tuple form: *(output_name, expression_or_source_column)*.
- Mixed lists of shorthand + tuples.

10.5.11.7 Notes

- *return_(["id"])* is shorthand for *return_([("id", "id")])*.
- Multiple projection steps are allowed and applied left-to-right: *return_(...)*, *with_(...)*, and *select(...)* each project from the current active row table produced by prior steps. Later projections can only reference columns that still exist after earlier projections.
- *order_by([("col", "asc" | "desc")])* sorts by one or more keys.
- *skip(n)* and *limit(n)* are row offsets/caps.
- In *where_rows(expr="...")*, comparison operators are =, !=, <>, <, <=, >, >=.
- For temporal/date-time row filtering, *where_rows(filter_dict=...)* uses the same predicate operators as MATCH filters (*gt*, *ge*, *lt*, *le*, *eq*, *ne*, *between*).
- Call-step placement rule: row-pipeline calls (*rows*, *where_rows*, *return_*, *with_*, *select*, *order_by*, *skip*, *limit*, *distinct*, *unwind*, *group_by*) are chain-list steps. Do not interleave call steps with *n()/e()* traversals in the chain interior; place calls in boundary prefix/suffix segments around traversal steps.
- Unsupported row expressions are rejected by validator/runtime.

See also: *GFQL Quick Reference*, *GFQL WHERE (Same-Path Constraints)*, *Cypher to GFQL Python & Wire Protocol Mapping*.

10.5.12 GFQL Operator Reference

This reference outlines 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.

10.5.12.1 Operators

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

10.5.12.2 WHERE Operators (Cross-Reference)

This page covers predicate functions used inside step filters like `n({...})` and `e_forward({...})`. WHERE operators are documented separately:

- Same-path MATCH WHERE uses `compare(col(...), op, col(...))` with `op` in `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Row-pipeline WHERE uses `where_rows(expr="...")` with comparators `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`.

See *GFQL WHERE (Same-Path Constraints)* (same-path constraints) and *GFQL RETURN (Row Pipelines)* (`MATCH ... RETURN` row pipelines).

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") })` (supports strings, numeric, temporals; use `isna()/notna()` for nulls)
`ne(value)`	Not equal to `value`.	`n({ "status": ne("inactive") })`
`between(lower, upper)`	Between `lower` and `upper` (inclusive).	`n({ "age": between(18, 65) })`

Note

Null handling and temporal comparisons are separate:

- Use `isna()` / `notna()` for null-safe checks: - `n({ "closed_at": isna() })` - `n({ "created_at": notna() })`
- Use comparison operators for non-null values (including temporal columns): - **DateTime**: `n({ "created_at": gt(pd.Timestamp("2023-01-01 12:00:00")) })` - **Date**: `n({ "event_date": eq(date(2023, 6, 15)) })` - **Time**: `n({ "daily_time": between(time(9, 0), time(17, 0)) })`

See *Working with Dates and Times* for datetime filtering examples.

Categorical Operators

Operator	Description	Example
`is_in(values)`	Value is in `values` list.	`n({ "type": is_in(["person", "company"]) })`
`duplicated(keep='first')`	Marks duplicated values.	`n({ "email": duplicated() })`

String Operators

Operator	Description	Example
`contains(pattern, case=True)`	String contains `pattern`. Case-insensitive if `case=False`.	`n({ "name": contains("smith", case=False) })`
`startswith(prefix, case=True)`	String starts with `prefix`. Case-insensitive if `case=False`.	`n({ "username": startswith( "admin", case=False) })`
`endswith(suffix, case=True)`	String ends with `suffix`. Case-insensitive if `case=False`.	`n({ "email": endswith(".com", case=False) })`
`match(pattern, case=True)`	String matches regex `pattern` from start. Case-insensitive if `case=False`.	`n({ "phone": match(r"^\d{3}-\ d{4}$") })`
`fullmatch(pattern, case=True)`	String matches regex `pattern` entirely. Case-insensitive if `case=False`.	`n({ "code": fullmatch(r"\d{3} ", case=False) })`
`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() })`

10.5.12.3 Usage Examples

Example 1: Filtering Nodes with Numeric Conditions

```
from graphistry 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 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 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 import n, eq, gt

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

Example 5: Same-Path Constraint with WHERE

```
from graphistry import n, e_forward, col, compare

g_filtered = g.gfql(
    [
        n({"type": "account"}, name="a"),
        e_forward(),
        n({"type": "user"}, name="c"),
    ],
    where=[compare(col("a", "owner_id"), "==", col("c", "owner_id"))],
)
```

10.5.12.4 Additional Notes

- **Predicate Functions**: Use predicate instances for filter conditions.

```
n({ "score": gt(50) })
```

  For compound conditions (e.g., `score > 50 AND score is even`), use a `query` string instead:

```
n(query="score > 50 and score % 2 == 0")
```

> **Note**
>
> Lambda functions in `filter_dict` (e.g., `n({"score": lambda x: ...})`) are no longer supported because `filter_dict` values must be JSON-serializable for the wire protocol and remote execution. Use predicates like `gt()`, `between()`, or `query=` strings for compound conditions.

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

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

- **Combining Conditions**: Use range predicates or `query` strings for complex expressions.

```
# Range predicate
n({ "age": between(19, 64) })

# Or equivalently with a query string
n(query="age > 18 and age < 65")
```

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

10.5.13 Working with Dates and Times

GFQL predicates support filtering by datetime, date, and time values. This guide covers common patterns and gotchas when working with temporal data.

10.5.13.1 Required Imports

```
# Core imports
import graphistry
from graphistry import n, e_forward, e_reverse, e_undirected

# Temporal predicates
from graphistry.compute import (
    gt, lt, ge, le, eq, ne, between, is_in,
    DateTimeValue, DateValue, TimeValue
)

# Standard datetime types
import pandas as pd
from datetime import datetime, date, time, timedelta
```

10.5.13.2 Supported Types and Standards

Supported Python Types

- https://pandas.pydata.org/docs/reference/api/pandas.Timestamp.html - Pandas timestamp
- https://docs.python.org/3/library/datetime.html#datetime.datetime - Python datetime
- https://docs.python.org/3/library/datetime.html#datetime.date - Date only (no time)
- https://docs.python.org/3/library/datetime.html#datetime.time - Time only (no date)
- Wire protocol dicts - For ISO strings and JSON compatibility

```
# Use datetime objects
gt(pd.Timestamp("2023-01-01 12:00:00"))
between(datetime(2023, 1, 1), datetime(2023, 12, 31))

# Wire protocol dicts accept ISO strings
gt({"type": "datetime", "value": "2023-01-01T00:00:00", "timezone": "UTC"})

# Raw strings raise ValueError
gt("2023-01-01")  # ValueError
```

Creating from Strings

```
# Timestamps
pd.Timestamp("2023-01-01T12:00:00Z")  # UTC
pd.Timestamp("2023-01-01 12:00:00")   # Naive

# Date/Time objects
date.fromisoformat("2023-01-01")      # date(2023, 1, 1)
time.fromisoformat("14:30:00")        # time(14, 30, 0)
```

Standards

- https://en.wikipedia.org/wiki/ISO_8601 datetime strings: `"2023-01-01T12:00:00Z"`
- https://www.iana.org/time-zones names: `"US/Eastern"`, `"UTC"`

Wire Protocol Types

For JSON serialization and cross-system compatibility:

- **DateTimeWire**: `{"type": "datetime", "value": "ISO-8601-string", "timezone": "IANA-timezone"}`
- **DateWire**: `{"type": "date", "value": "YYYY-MM-DD"}`
- **TimeWire**: `{"type": "time", "value": "HH:MM:SS[.ffffff]"}`

Note: The `timezone` field is optional for DateTimeWire and defaults to "UTC" if omitted.

10.5.13.3 Basic Usage

Datetime Filtering

Filter nodes or edges based on datetime values:

```
import pandas as pd
from datetime import datetime
from graphistry import n, e_forward
from graphistry.compute import gt, lt, between

# Filter nodes created after a specific datetime
recent_nodes = g.gfql([
    n(filter_dict={"created_at": gt(pd.Timestamp("2023-01-01 12:00:00"))})
])

# Filter edges within a date range
date_range_edges = g.gfql([
    n(), e_forward(edge_match={"timestamp": between(
        datetime(2023, 1, 1),
        datetime(2023, 12, 31)
    )}), n()
])
```

Comparison Operators (Python, Row Expressions, Wire)

Temporal comparisons use the same core operators across APIs:

- Python predicate helpers: `gt`, `ge`, `lt`, `le`, `eq`, `ne`
- Wire protocol predicate types: `GT`, `GE`, `LT`, `LE`, `EQ`, `NE`
- `where_rows(expr="...")` comparators: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`

Example mapping:

- Python: `gt(pd.Timestamp("2024-01-01"))`
- Wire: `{"type": "GT", "val": {"type": "datetime", "value": "2024-01-01T00:00:00", "timezone": "UTC"}}`
- Row expression: `where_rows(expr="created_at > created_cutoff")`

WHERE Context (What Works Where)

Temporal filters work in different forms depending on GFQL context:

Context	Temporal form	Notes
Matcher filters (`n(filter_dict=...)`, `e_forward(edge_match=...)`)	Predicate helpers (`gt`, `ge`, `lt`, `le`, `eq`, `ne`, `between`, `is_in`)	Primary place for temporal predicates
Same-path `where=[...]`	`compare(col(...), op, col(...))` with `op` in `==`, `!=`, `<`, `<=`, `>`, `>=`	Alias-column comparisons; predicate helpers are not used here
Row pipeline `where_rows(filter_dict=...)`	Same predicate helpers as matcher filters	Useful after `rows(...)` in `MATCH ... RETURN` flows
Row pipeline `where_rows(expr="...")`	Expression comparators `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`	Expression parser syntax; no predicate helper function calls inside `expr`

See also:

- *GFQL WHERE (Same-Path Constraints)*
- *GFQL RETURN (Row Pipelines)*

Temporal WHERE Examples (Valid vs Invalid)

```
import pandas as pd
from datetime import datetime

from graphistry import n, e_forward, col, compare
from graphistry.compute import rows, where_rows, return_, lt

# 1) Same-path WHERE (MATCH-stage): use compare(..., op, ...)
g.gfql(
    [
        n({"type": "event"}, name="a"),
        e_forward(name="e"),
        n({"type": "event"}, name="b"),
    ],
    where=[compare(col("a", "created_at"), "<", col("b", "created_at"))],
)

# 2) Row-pipeline WHERE with predicate helpers: where_rows(filter_dict=...)
g.gfql([
    n({"type": "event"}, name="evt"),
    rows(table="nodes", source="evt"),
    where_rows(filter_dict={"created_at": lt(pd.Timestamp("2024-01-01T00:00:00Z"))}),
    return_(["id", "created_at"]),
])

# 3) Row-pipeline WHERE with expression comparators: where_rows(expr="...")
g.gfql([
    n({"type": "event"}, name="evt"),
    rows(table="nodes", source="evt"),
    where_rows(expr="created_at < cutoff_ts"),
    return_(["id", "created_at", "cutoff_ts"]),
])

# INVALID for same-path WHERE:
# where=[lt(col("a", "created_at"))]  # predicate helper form is not valid here
```

Date-Only Filtering

For date comparisons (ignoring time):

```
from datetime import date
from graphistry.compute import eq, ge

# Filter nodes by exact date
specific_date = g.gfql([
    n(filter_dict={"event_date": eq(date(2023, 6, 15))})
])

# Filter nodes on or after a date
after_date = g.gfql([
    n(filter_dict={"start_date": ge(date(2023, 1, 1))})
])
```

Time-Only Filtering

Filter based on time of day:

```
from datetime import time
from graphistry.compute import is_in, between

# Filter events at specific times
morning_events = g.gfql([
    n(filter_dict={"event_time": is_in([
        time(9, 0, 0),
        time(9, 30, 0),
        time(10, 0, 0)
    ])})
])

# Filter events in time range
business_hours = g.gfql([
    n(filter_dict={"timestamp": between(
        time(9, 0, 0),
        time(17, 0, 0)
    )})
])
```

10.5.13.4 Timezone Support

```
# Timezone-aware filtering
tz_aware_filter = g.gfql([
    n(filter_dict={
        "timestamp": gt(pd.Timestamp("2023-01-01 12:00:00", tz="US/Eastern"))
    })
])
```

Comparisons automatically handle timezone conversions.

10.5.13.5 Advanced Usage

Mixed Temporal and Non-Temporal Predicates

Combine temporal predicates with other filters:

```
from graphistry.compute import gt, lt, eq

# Complex filter with multiple conditions
complex_filter = g.gfql([
    n(filter_dict={
        "created_at": gt(datetime(2023, 1, 1)),
        "status": eq("active"),
        "priority": gt(5)
    })
])
```

Using Wire Protocol Dictionaries Directly

You can pass wire protocol dictionaries directly to predicates, which is useful for programmatic predicate creation or when working with JSON configurations:

```
# Pass wire protocol dictionaries directly
filter_with_dict = g.gfql([
    n(filter_dict={"timestamp": gt({
        "type": "datetime",
        "value": "2023-01-01T12:00:00",
        "timezone": "UTC"
    })})
])

# Works with all temporal predicates
date_range_filter = g.gfql([
    n(filter_dict={"event_date": between(
        {"type": "date", "value": "2023-01-01"},
        {"type": "date", "value": "2023-12-31"}
    )})
])

# And with is_in for multiple values
time_filter = g.gfql([
    n(filter_dict={"event_time": is_in([
        {"type": "time", "value": "09:00:00"},
        {"type": "time", "value": "12:00:00"},
        {"type": "time", "value": "17:00:00"}
    ])})
])
```

This is the same format used by the wire protocol, making it easy to:

- Store predicate configurations in JSON files
- Build predicates programmatically from external data sources

- Share predicate definitions between Python and other systems

Temporal Predicates in Multi-Hop Queries

Use temporal filters in complex graph traversals:

```
# Find all transactions after a date, then their related accounts
recent_transactions = g.gfql([
    n(filter_dict={"type": eq("transaction"),
                   "date": gt(date(2023, 6, 1))}),
    e_forward(edge_match={"relationship": eq("involves")}),
    n(filter_dict={"type": eq("account")})
])
```

10.5.13.6 Temporal Value Classes

PyGraphistry provides three temporal value classes for precise control:

DateTimeValue

For full datetime with optional timezone:

```
from graphistry.compute import DateTimeValue, gt

# Create datetime value with timezone
dt_value = DateTimeValue("2023-01-01T12:00:00", "US/Eastern")

# Use in predicate
filter_dt = g.gfql([
    n(filter_dict={"timestamp": gt(dt_value)})
])
```

DateValue

For date-only comparisons:

```
from graphistry.compute import DateValue, between

# Create date values
start = DateValue("2023-01-01")
end = DateValue("2023-12-31")

# Use in between predicate
year_filter = g.gfql([
    n(filter_dict={"event_date": between(start, end)})
])
```

TimeValue

For time-of-day comparisons:

```
from graphistry.compute import TimeValue, is_in

# Create time values
morning = TimeValue("09:00:00")
noon = TimeValue("12:00:00")

# Filter by specific times
time_filter = g.gfql([
    n(filter_dict={"daily_event": is_in([morning, noon])})
])
```

10.5.13.7 Best Practices

1. **Use Explicit Types**: Always use `pd.Timestamp`, `datetime`, `date`, or `time` objects instead of strings to avoid ambiguity.
2. **Timezone Awareness**: When working with timestamps across timezones, always specify timezones explicitly.
3. **Performance**: Temporal comparisons are optimized for pandas DataFrames. For large datasets, ensure your datetime columns are properly typed.
4. **JSON Serialization**: When serializing queries, temporal values are automatically converted to tagged dictionaries that preserve type and timezone information.

10.5.13.8 Unsupported Features

Duration/Interval Support

Currently, PyGraphistry does not support duration or interval types (e.g., ISO 8601 durations like "P1D" or "PT2H"). For duration-based queries:

```
# Instead of duration literals, calculate explicit timestamps
from datetime import datetime, timedelta

# Find events within last 7 days
now = datetime.now()
week_ago = now - timedelta(days=7)
recent_events = g.gfql([
    n(filter_dict={"timestamp": gt(pd.Timestamp(week_ago))})
])

# For recurring intervals, use multiple conditions
business_days = g.gfql([
    n(filter_dict={
        "timestamp": between(
            pd.Timestamp("2023-01-01"),
            pd.Timestamp("2023-12-31")
```

(continues on next page)

(continued from previous page)

```
        )
    })
])
```

10.5.13.9 Common Patterns

Filter Recent Data

```
from datetime import datetime, timedelta

# Get data from last 30 days
thirty_days_ago = datetime.now() - timedelta(days=30)
recent_data = g.gfql([
    n(filter_dict={"timestamp": gt(pd.Timestamp(thirty_days_ago))})
])
```

Business Hours Filtering

```
# Filter events during business hours
business_hours = g.gfql([
    n(filter_dict={
        "timestamp": between(time(9, 0, 0), time(17, 0, 0))
    })
])
```

Quarterly Data Analysis

```
# Q1 2023 data
q1_2023 = g.gfql([
    n(filter_dict={
        "date": between(
            date(2023, 1, 1),
            date(2023, 3, 31)
        )
    })
])
```

10.5.13.10 Error Handling

```
# Strings raise ValueError - always use datetime objects
gt("2023-01-01")                # ValueError: Raw string not allowed
gt(pd.Timestamp("2023-01-01"))  # Correct: Use pandas Timestamp
```

10.5.13.11 See Also

- *GFQL Predicates API Reference*
- *GFQL Chain Operations*
- *Wire Protocol Reference*

10.5.14 GFQL Built-in Call Reference

The Call operation in GFQL provides access to a curated set of graph algorithms, transformations, and visualization methods. All methods are validated through a safelist to ensure security and stability.

Table of Contents

10.5.14.1 Overview

Call operations are invoked using the `call()` function within GFQL chains or Let bindings, or using typed builders for better IDE support:

```
from graphistry import call, let, ref, n, e_forward, gt

# Pure call() chains work - filter then enrich
result = g.gfql([
    call('filter_nodes_by_dict', {'filter_dict': {'type': 'person'}}),
    call('get_degrees', {'col': 'degree'})
])

# For filter->enrich->filter patterns, use let()
result = g.gfql(let({
    'persons': n({'type': 'person'}),
    'with_degrees': ref('persons', [call('get_degrees', {'col': 'degree'})]),
    'high_degree': ref('with_degrees', [n({'degree': gt(10)})]),
    'connected': ref('with_degrees', [n({'degree': gt(10)}), e_forward(), n()])
}))
```

All Call operations:

- Validate parameters against type and value constraints
- Return a modified graph (immutable - original is unchanged)
- Can add columns to nodes or edges (schema effects)
- Are restricted to methods in the safelist for security

Call operations stay in graph state: the result remains a traversable graph with meaningful *_edges*, so you can keep matching or compose additional graph stages with *let() / ref()*. If you want row/tabular output, switch into row-pipeline operators such as *rows()*, *with_()*, *select()*, *return_()*, *group_by()*, or use a row-returning local Cypher *CALL ... YIELD ... RETURN ...* query.

Layout-aware consumers can detect layout helper calls without maintaining their own string lists:

```
from graphistry.compute import call
from graphistry.compute.gfql import (
    LAYOUT_FUNCTION_NAMES,
    RADIAL_LAYOUT_FUNCTION_NAMES,
    RADIAL_LAYOUT_KINDS,
    is_layout_chain,
    is_layout_kind,
)

chain = [call('time_ring_layout', {'time_col': 'ts'})]
kind = is_layout_kind(chain)

assert is_layout_chain(chain)
assert kind == 'time_ring'
assert kind in RADIAL_LAYOUT_KINDS
assert 'time_ring_layout' in RADIAL_LAYOUT_FUNCTION_NAMES
assert 'group_in_a_box_layout' in LAYOUT_FUNCTION_NAMES
assert 'tree_layout' in LAYOUT_FUNCTION_NAMES
```

The helpers accept materialized GFQL `Chain` objects, list/tuple chains, Chain wire dictionaries, and direct `Call` objects/dicts. Opaque strings are not parsed, so pass native GFQL structures rather than substring-matching query text. For legitimate pre-parse string diagnostics, use `LAYOUT_FUNCTION_NAMES` or `RADIAL_LAYOUT_FUNCTION_NAMES` as the canonical safelisted function-name registries instead of maintaining local copies. These registries reflect GFQL `call()` safelist names. Legacy `radial_*` names are not safelisted pygraphistry function names; normalize them before calling these helpers.

10.5.14.2 Graph Transformation Methods

hypergraph

Transform event data into entity relationships by connecting entities that appear together in events. This is useful for converting event-based data (logs, transactions, activities) into entity-entity graphs.

Parameters:

Parameter	Type	Required	Description
entity_types	list[string]	No	Column names to use as entity types. If None, uses all columns
opts	dict	No	Configuration options for hypergraph transformation (see below)
drop_na	boolean	No	Whether to drop rows with NA values in entity columns (default: True)
drop_edge_attrs	boolean	No	Whether to drop non-entity attributes from edges (default: True)
verbose	boolean	No	Whether to print verbose output during transformation (default: False)
direct	boolean	No	If True, creates direct entity-to-entity edges. If False, keeps hypernodes to show event connections (default: True)
engine	string	No	Processing engine - 'pandas', 'cudf' (GPU), 'dask' (streaming), or 'auto' (default: 'auto')
npartitions	integer	No	Number of partitions for Dask processing
chunksize	integer	No	Chunk size for streaming processing
from_edges	boolean	No	If True, use edges dataframe as input instead of nodes dataframe (default: False)
return_as	string	No	What to return from hypergraph result: 'graph' (default), 'all', 'entities', 'events', 'edges', 'nodes'

The opts Parameter:

The `opts` dictionary configures advanced hypergraph behavior by controlling how entities are identified and connected. All keys are optional and the dictionary structure is validated to ensure type safety:

Key	Type	Description
TITLE	string	Node title field name (default: 'nodeTitle')
DELIM	string	Delimiter for composite IDs (default: '::')
NODEID	string	Node ID field name (default: 'nodeID')
ATTRIBID	string	Attribute ID field name (default: 'attribID')
EVENTID	string	Event ID field name (default: 'EventID')
EVENTTYPE	string	Event type field name (default: 'event')
SOURCE	string	Source node field name for edges (default: 'src')
DESTINATION	string	Destination node field name for edges (default: 'dst')
CATEGORY	string	Category field name (default: 'category')
NODETYPE	string	Node type field name (default: 'type')
EDGETYPE	string	Edge type field name (default: 'edgeType')
NULLVAL	string	Value representing null (default: 'null')
SKIP	list[string]	Column names to exclude from entity extraction. Each item must be a string
CATEGORIES	dict[str, list[str]]	Maps category names to lists of values for grouping. Keys must be strings, values must be lists of strings
EDGES	dict[str, list[str]]	Defines which entity types can connect to each other. Keys represent source entity types (strings), values are lists of target entity types (strings) that the source can connect to

Examples:

```
# Transform user-product interactions into entity graph
events_df = pd.DataFrame({
    'user': ['alice', 'bob', 'alice'],
    'product': ['laptop', 'phone', 'tablet'],
    'timestamp': [1, 2, 3]
})
g = graphistry.nodes(events_df)

# Simple transformation using typed builder (recommended)
hg = g.gfql(hypergraph(entity_types=['user', 'product']))

# Or using call() directly
hg = g.gfql(call('hypergraph', {'entity_types': ['user', 'product']}))

# Keep hypernodes to show event connections
hg = g.gfql(hypergraph(
    entity_types=['user', 'product'],
    direct=False  # Keep hypernodes
))

# Use GPU acceleration
hg = g.gfql(hypergraph(
    entity_types=['user', 'product'],
    engine='cudf'
))

# Advanced opts configuration with CATEGORIES and EDGES
hg = g.gfql(hypergraph(
    entity_types=['user', 'product', 'category'],
    opts={
        'TITLE': 'Entity Graph',
        'SKIP': ['timestamp', 'metadata'],  # Exclude these columns
        'CATEGORIES': {
            'user_type': ['premium', 'regular', 'trial'],
            'product_type': ['electronics', 'clothing', 'books']
        },
        'EDGES': {
            'user': ['product', 'category'],  # Users connect to products and categories
            'product': ['user', 'category'],  # Products connect back to users and␣
↪categories
            'category': ['product']           # Categories only connect to products
        }
    }
))

# In a DAG with other operations
from graphistry import let, ref, n

result = g.gfql(let({
    'hg': hypergraph(entity_types=['user', 'product']),
    'filtered': ref('hg', [n({'type': 'user'})])
}))
```

(continues on next page)

(continued from previous page)

```
# Use edges dataframe as input
edges_df = pd.DataFrame({
    'src_user': ['alice', 'bob', 'alice'],
    'dst_item': ['laptop', 'phone', 'tablet']
})
g = graphistry.edges(edges_df, 'src_user', 'dst_item')

hg = g.gfql(hypergraph(
    from_edges=True,
    entity_types=['src_user', 'dst_item']
))

# Extract only entities dataframe (not full graph)
entities_df = g.gfql(hypergraph(
    entity_types=['user', 'product'],
    return_as='entities'  # Returns DataFrame instead of Plottable
))

# Extract edges only
edges_df = g.gfql(hypergraph(
    entity_types=['user', 'product'],
    return_as='edges'
))

# Combine both parameters
entity_nodes = g.gfql(hypergraph(
    from_edges=True,
    entity_types=['src_user', 'dst_item'],
    return_as='entities'
))
```

Use Cases:

- **Social Network Analysis**: Transform interaction events (messages, calls) into social graphs
- **Fraud Detection**: Connect accounts, merchants, and devices from transaction events
- **Security Analysis**: Link users, IPs, and resources from access logs
- **Supply Chain**: Connect suppliers, products, and customers from order events

Schema Effects:

Creates a new graph structure where:

- Nodes represent unique entities from the specified columns
- Edges connect entities that appeared in the same event
- Edge attributes can include event metadata (if drop_edge_attrs=False)

Return Value:

By default (`return_as='graph'`), returns a Plottable graph object for method chaining. The `return_as` parameter controls what is returned:

- `'graph'`: Plottable graph (default) - enables chaining like `.plot()`

- `'all'`: Dict with all 5 components (graph, entities, events, edges, nodes) - backward compatible with module-level `graphistry.hypergraph()`
- `'entities'`: DataFrame of entity nodes only
- `'events'`: DataFrame of event/hypernode nodes only
- `'edges'`: DataFrame of edges only
- `'nodes'`: DataFrame of all nodes (entities + events)

Note

Hypergraph transformations cannot be mixed with other operations in chains. Use as a single operation or within Let/DAG constructs for complex compositions.

Note

For large datasets, consider using engine='cudf' for GPU acceleration or engine='dask' for streaming processing.

10.5.14.3 Graph Analysis Methods

compute_cugraph

Run GPU-accelerated graph algorithms using https://github.com/rapidsai/cugraph, part of the https://rapids.ai/ ecosystem.

Parameters:

Parameter	Type	Required	Description
alg	string	Yes	Algorithm name (see supported algorithms below)
out_col	string	No	Output column name (defaults to algorithm name)
params	dict	No	Algorithm-specific parameters
kind	string	No	Graph type hints
directed	boolean	No	Whether to treat graph as directed
G	None	No	Reserved (must be None if provided)

Supported Algorithms:

The exact procedure names mirror `graphistry.plugins.cugraph.compute_algs`. Current categories include:

- **Node-enriching:** `betweenness_centrality`, `bfs`, `bfs_edges`, `connected_components`, `core_number`, `ecg`, `hits`, `katz_centrality`, `leiden`, `louvain`, `pagerank`, `shortest_path`, `shortest_path_length`, `spectralBalancedCutClustering`, `spectralModularityMaximizationClustering`, `sssp`, `strongly_connected_components`
- **Edge-enriching:** `batched_ego_graphs`, `edge_betweenness_centrality`, `jaccard`, `jaccard_w`, `overlap`, `overlap_coefficient`, `overlap_w`, `sorensen`, `sorensen_coefficient`, `sorensen_w`
- **Topology-returning:** `ego_graph`, `k_core`, `minimum_spanning_tree`

Examples:

```
# PageRank with custom parameters
g.gfql([
    call('compute_cugraph', {
        'alg': 'pagerank',
        'out_col': 'pr_score',
        'params': {'alpha': 0.85, 'max_iter': 100}
    })
])

# Community detection
g.gfql([
    call('compute_cugraph', {
        'alg': 'louvain',
        'out_col': 'community'
    })
])

# Weakly connected components (WCC) labels
g.gfql([
    call('compute_cugraph', {
        'alg': 'connected_components',
        'out_col': 'wcc_id',
        'directed': False
    })
])

# Strongly connected components (SCC) labels
g.gfql([
    call('compute_cugraph', {
        'alg': 'strongly_connected_components',
        'out_col': 'scc_id',
        'directed': True
    })
])

# Betweenness centrality
g.gfql([
    call('compute_cugraph', {
        'alg': 'betweenness_centrality',
        'out_col': 'betweenness',
        'directed': True
    })
])
```

Schema Effects: Depends on the algorithm family. Node algorithms add node columns, edge algorithms add edge columns, and topology-returning algorithms return a new graph topology.

Local Cypher Modes:

- **Procedure naming:** `CALL graphistry.cugraph.<alg>()` and `CALL graphistry.cugraph.<alg>.write()` mirror `compute_cugraph(alg=...)` for the supported algorithm names above.
- **Row mode for node algorithms:** `g.gfql("CALL graphistry.cugraph.louvain()")` returns row state with `nodeId` plus the default algorithm output columns in `_nodes` and an empty placeholder `_edges` frame (for example, `assert result._edges.empty`).

- **Row mode for edge algorithms:** `g.gfql("CALL graphistry.cugraph.edge_betweenness_centrality()")` returns row state with `source`, `destination`, and the edge result columns in `_nodes` while leaving `_edges` empty.
- **Graph mode / topology mode:** `g.gfql("CALL graphistry.cugraph.edge_betweenness_centrality.write()")` enriches the graph in place and keeps traversable edges (for example, `assert not result._edges.empty`). Topology-returning algorithms such as `k_core` and `minimum_spanning_tree` require `.write()`.
- **Options map:** Local Cypher procedures accept one optional map argument. `out_col`, `directed`, `kind`, and `params` mirror `compute_cugraph()` directly, and any extra keys are forwarded into the nested algorithm `params` dictionary. For example, `CALL graphistry.cugraph.louvain({resolution: 1.0})` maps to `compute_cugraph('louvain', params={'resolution': 1.0})`.

Parameter Discovery: For detailed algorithm parameters, see the https://docs.rapids.ai/api/cugraph/stable/. Parameters are passed via the `params` dictionary.

> **Note**
>
> For workloads taking 5 seconds to 5 hours on CPU, consider using *GFQL Remote Mode* to offload computation to a GPU-enabled server.

compute_igraph

Run CPU-based graph algorithms using https://igraph.org/, the comprehensive network analysis library.

Parameters:

Parameter	Type	Required	Description
alg	string	Yes	Algorithm name (see supported algorithms below)
out_col	string	No	Output column name (defaults to algorithm name)
params	dict	No	Algorithm-specific parameters
directed	boolean	No	Whether to treat graph as directed
use_vids	boolean	No	Whether to use vertex IDs

Supported Algorithms:

The exact procedure names mirror `graphistry.plugins.igraph.compute_algs`. Current supported names include:

- `articulation_points`, `authority_score`, `betweenness`, `bibcoupling`, `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`, `eccentricity`, `eigenvector_centrality`, `harmonic_centrality`, `hub_score`, `k_core`, `pagerank`, `personalized_pagerank`
- Topology-returning procedures: `gomory_hu_tree` and `spanning_tree`

Examples:

```
# PageRank using igraph
g.gfql([
    call('compute_igraph', {
        'alg': 'pagerank',
        'out_col': 'pagerank',
        'params': {'damping': 0.85}
    })
])

# Community detection
g.gfql([
    call('compute_igraph', {
        'alg': 'community_multilevel',
        'out_col': 'community'
    })
])

# Weakly connected components (WCC) labels
g.compute_igraph('clusters', out_col='wcc_id', params={'mode': 'weak'})

# Strongly connected components (SCC) labels
g.compute_igraph('clusters', out_col='scc_id', params={'mode': 'strong'})
```

Schema Effects: Most algorithms add one node column. Topology-returning algorithms such as `gomory_hu_tree` and `spanning_tree` return a new graph topology instead.

Local Cypher Modes:

- **Procedure naming:** `CALL graphistry.igraph.<alg>()` and `CALL graphistry.igraph.<alg>.write()` mirror `compute_igraph(alg=...)` for the supported algorithm names above.
- **Row mode:** `g.gfql("CALL graphistry.igraph.pagerank()")` returns row state with `nodeId` plus the default algorithm output column in `_nodes` and an empty placeholder `_edges` frame (for example, `assert result._edges.empty`).
- **Graph mode / topology mode:** `g.gfql("CALL graphistry.igraph.pagerank.write()")` keeps the result in graph state with traversable edges (for example, `assert not result._edges.empty`). Topology-returning algorithms such as `spanning_tree` and `gomory_hu_tree` require `.write()`.
- **Options map:** Local Cypher procedures accept one optional map argument. `out_col`, `directed`, `use_vids`, and `params` mirror `compute_igraph()` directly, and any extra keys are forwarded into the nested algorithm `params` dictionary. For example, `CALL graphistry.igraph.pagerank({damping: 0.9, directed: false})` maps to `compute_igraph('pagerank', directed=False, params={'damping': 0.9})`.
- **NetworkX local subset:** The local Cypher compiler also supports a CPU `graphistry.nx.*` subset for parity with common cuGraph-style row outputs:
  - Node-enriching calls: `CALL graphistry.nx.pagerank() / .write()`, `CALL graphistry.nx.betweenness_centrality() / .write()`, `CALL graphistry.nx.degree_centrality() / .write()`, `CALL graphistry.nx.closeness_centrality() / .write()`, `CALL graphistry.nx.eigenvector_centrality() / .write()`, `CALL graphistry.nx.katz_centrality() / .write()`, `CALL graphistry.nx.connected_components() / .write()`, `CALL graphistry.nx.strongly_connected_components() / .write()`, `CALL graphistry.nx.core_number() / .write()`, and `CALL graphistry.nx.hits() / .write()`
  - Edge-enriching calls: `CALL graphistry.nx.edge_betweenness_centrality() / .write()`

- Topology-returning calls: `CALL graphistry.nx.k_core.write()`

They follow the same row-vs-`.write()` contract as the other backends: node calls use `nodeId` + value column rows, edge calls use `source` / `destination` + value column rows, and topology-returning calls require `.write()`. Multi-output `hits` returns `nodeId`, `hubs`, and `authorities` and does not accept `out_col`.

The same curated NetworkX algorithm subset is available from regular Python as `g.compute_networkx(...)` for users who do not need the local Cypher `CALL` path.

Parameter Discovery: For detailed algorithm parameters, see the https://igraph.org/python/. Parameters are passed via the `params` dictionary.

> **Note**
>
> Component IDs (for example, `wcc_id` / `scc_id`) are partition labels and may differ numerically across backends or runs. Use them for grouping and filtering by component membership, not as stable semantic IDs.

> **Note**
>
> For graphs with millions of edges, consider using `compute_cugraph` with a GPU for 10-50x speedup, or *GFQL Remote Mode* if no local GPU is available.

get_degrees

Calculate degree centrality for nodes (in-degree, out-degree, and total degree).

Parameters:

Parameter	Type	Required	Description
col	string	No	Column name for total degree
degree_in	string	No	Column name for in-degree
degree_out	string	No	Column name for out-degree

Examples:

```
# Calculate all degree types
g.gfql([
    call('get_degrees', {
        'col': 'total_degree',
        'degree_in': 'in_degree',
        'degree_out': 'out_degree'
    })
])

# Calculate only total degree
g.gfql([
    call('get_degrees', {'col': 'degree'})
])
```

(continues on next page)

(continued from previous page)

```
# Filter by degree using let()
from graphistry import let, ref, call, n, gt

g.gfql(let({
    'with_degrees': call('get_degrees', {'col': 'degree'}),
    'filtered': ref('with_degrees', [n({'degree': gt(10)})])
}))
```

Schema Effects: Adds up to 3 columns to nodes (based on parameters provided).

Local Cypher Modes:

- **Row mode:** `g.gfql("CALL graphistry.degree()")` returns row state with default `nodeId`, `degree`, `degree_in`, and `degree_out` columns in `_nodes` and an empty placeholder `_edges` frame (for example, `assert result._edges.empty`). Add `YIELD ... RETURN ...` when you want to project or sort those rows explicitly.
- **Graph mode:** `g.gfql("CALL graphistry.degree.write()")` materializes `degree`, `degree_in`, and `degree_out` on nodes while preserving the graph for later matches with traversable edges (for example, `assert not result._edges.empty`).

get_indegrees

Calculate only in-degree for nodes.

Parameters:

Parameter	Type	Required	Description
col	string	No	Column name for in-degree (default: 'in_degree')

Example:

```
g.gfql([
    call('get_indegrees', {'col': 'incoming_connections'})
])
```

Schema Effects: Adds one column to nodes.

get_outdegrees

Calculate only out-degree for nodes.

Parameters:

Parameter	Type	Required	Description
col	string	No	Column name for out-degree (default: 'out_degree')

Example:

```
g.gfql([
    call('get_outdegrees', {'col': 'outgoing_connections'})
])
```

Schema Effects: Adds one column to nodes.

get_topological_levels

Compute topological levels for directed acyclic graphs (DAGs).

Parameters:

Parameter	Type	Required	Description
level_col	string	No	Column name for level (default: 'level')
allow_cycles	boolean	No	Whether to allow cycles (default: True)

Example:

```
# Compute DAG levels
g.gfql([
    call('get_topological_levels', {
        'level_col': 'topo_level',
        'allow_cycles': False
    })
])
```

Schema Effects: Adds one column to nodes.

10.5.14.4 Layout Methods

layout_cugraph

Compute GPU-accelerated graph layouts.

Parameters:

Parameter	Type	Required	Description
layout	string	No	Layout algorithm (default: 'force_atlas2')
params	dict	No	Layout-specific parameters
kind	string	No	Graph type hints
directed	boolean	No	Whether to treat graph as directed
bind_position	boolean	No	Whether to bind positions to nodes
x_out_col	string	No	X coordinate column name
y_out_col	string	No	Y coordinate column name
play	integer	No	Animation frames

Supported Layouts:

- **force_atlas2**: Force-directed layout

Example:

```
g.gfql([
    call('layout_cugraph', {
        'layout': 'force_atlas2',
        'params': {
            'iterations': 500,
            'outbound_attraction_distribution': True,
            'edge_weight_influence': 1.0
        }
    })
])
```

Schema Effects: Modifies node positions or adds position columns.

layout_igraph

Compute CPU-based graph layouts using igraph.

Parameters:

Parameter	Type	Required	Description
layout	string	Yes	Layout algorithm name
params	dict	No	Layout-specific parameters
directed	boolean	No	Whether to treat graph as directed
use_vids	boolean	No	Whether to use vertex IDs
bind_position	boolean	No	Whether to bind positions
x_out_col	string	No	X coordinate column name
y_out_col	string	No	Y coordinate column name
play	integer	No	Animation frames

Supported Layouts:

- **kamada_kawai**: Kamada-Kawai layout
- **fruchterman_reingold**: Fruchterman-Reingold force-directed
- **circle**: Circular layout
- **grid**: Grid layout
- **random**: Random layout
- **drl**: Distributed Recursive Layout
- **lgl**: Large Graph Layout
- **graphopt**: GraphOpt layout
- Many more...

Example:

```
g.gfql([
    call('layout_igraph', {
        'layout': 'fruchterman_reingold',
        'params': {'niter': 500}
```

(continues on next page)

(continued from previous page)

```
    })
])
```

Schema Effects: Modifies node positions or adds position columns.

layout_graphviz

Compute layouts using Graphviz algorithms.

Parameters:

Parameter	Type	Required	Description
prog	string	No	Graphviz program (default: 'dot')
args	string	No	Additional Graphviz arguments
directed	boolean	No	Whether graph is directed
bind_position	boolean	No	Whether to bind positions
x_out_col	string	No	X coordinate column name
y_out_col	string	No	Y coordinate column name
play	integer	No	Animation frames

Supported Programs:

- **dot**: Hierarchical layout
- **neato**: Spring model layout
- **fdp**: Force-directed layout
- **sfdp**: Scalable force-directed
- **circo**: Circular layout
- **twopi**: Radial layout

Example:

```
# Hierarchical layout
g.gfql([
    call('layout_graphviz', {
        'prog': 'dot',
        'directed': True
    })
])

# Circular layout
g.gfql([
    call('layout_graphviz', {'prog': 'circo'})
])
```

Schema Effects: Modifies node positions or adds position columns.

fa2_layout

Apply ForceAtlas2 layout algorithm (CPU-based implementation).

> **Note**
>
> This is a CPU-based ForceAtlas2 implementation. For GPU acceleration, use `call('layout_cugraph', {'layout': 'force_atlas2'})` instead.

Parameters:

Parameter	Type	Required	Description
fa2_params	dict	No	ForceAtlas2 parameters

Example:

```
g.gfql([
    call('fa2_layout', {
        'fa2_params': {
            'iterations': 1000,
            'gravity': 1.0,
            'scaling_ratio': 2.0
        }
    })
])
```

Schema Effects: Modifies node positions.

group_in_a_box_layout

Apply group-in-a-box layout that organizes nodes into rectangular regions by community.

PyGraphistry's implementation is optimized for large graphs on both CPU and GPU.

References: - Paper: https://www.cs.umd.edu/users/ben/papers/Rodrigues2011Group.pdf - Blog post: https://www.graphistry.com/blog/gpu-group-in-a-box-layout-for-larger-social-media-investigations

Parameters:

Parameter	Type	Required	Description
partition_alg	string	No	Community detection algorithm (e.g., 'louvain')
partition_params	dict	No	Parameters for partition algorithm
layout_alg	string/-callable	No	Layout algorithm for each box
layout_params	dict	No	Parameters for layout algorithm
x	number	No	X coordinate of bounding box
y	number	No	Y coordinate of bounding box
w	number	No	Width of bounding box
h	number	No	Height of bounding box
encode_colors	boolean	No	Whether to encode communities as colors
colors	list[string]	No	List of colors for communities
partition_key	string	No	Existing column to use as partition
engine	string	No	Engine ('auto', 'cpu', 'gpu', 'pandas', 'cudf')

Examples:

```
# Basic usage - auto-detect communities
g.gfql([
    call('group_in_a_box_layout')
])

# Use specific partition algorithm
g.gfql([
    call('group_in_a_box_layout', {
        'partition_alg': 'louvain',
        'engine': 'cpu'
    })
])

# Use existing partition column
g.gfql([
    call('group_in_a_box_layout', {
        'partition_key': 'department',
        'encode_colors': True
    })
])

# Full control over layout
g.gfql([
    call('group_in_a_box_layout', {
        'partition_alg': 'louvain',
        'layout_alg': 'force_atlas2',
        'x': 0, 'y': 0, 'w': 1000, 'h': 1000,
        'colors': ['#ff0000', '#00ff00', '#0000ff']
    })
])
```

Schema Effects: Modifies node positions and optionally adds color encoding.

circle_layout

Arrange nodes in a circular layout.

Parameters:

Parameter	Type	Required	Description
bounding_box	list[number]	No	`[cx, cy, width, height]` bounding box. If omitted, the graph must already have `x`/`y` node positions.
ring_spacing	number	No	Spacing between rings
point_spacing	number	No	Spacing between points
partition_by	string or list[string]	No	Node column(s) for partitioned circles
sort_by	string or list[string]	No	Node column(s) for sort order
ascending	boolean or list[boolean]	No	Sort direction
na_position	string	No	`'first'` or `'last'`
ignore_index	boolean	No	Whether to ignore index during sort
engine	string	No	Engine (`'auto'`, `'pandas'`, `'cudf'`, `'dask'`, `'dask_cudf'`)

Example:

```
g.gfql([
    call('circle_layout', {
        'bounding_box': [0, 0, 1000, 1000],
        'engine': 'pandas'
    })
])
```

Schema Effects: Modifies node positions.

tree_layout

Apply the Sugiyama-style tree layout.

Parameters:

Parameter	Type	Required	Description
level_col	string	No	Output level column
level_sort_values_b	string or list[string]	No	Column(s) for ordering nodes within levels
level_sort_values_b	boolean	No	Sort direction for level ordering
width	number	No	Layout width multiplier
height	number	No	Layout height multiplier
rotate	number	No	Rotation in degrees
allow_cycles	boolean	No	Whether to tolerate cyclic inputs
root	string/number/boolean	No	Optional root node id

Example:

```
g.gfql([
    call('tree_layout', {'level_col': 'level'})
])
```

Schema Effects: Modifies node positions and adds a level column.

mercator_layout

Project latitude/longitude node columns into local Mercator coordinates.

Parameters:

Parameter	Type	Required	Description
scale_for_graphistry	boolean	No	Use scaled coordinates suitable for Graphistry visualization

Example:

```
geo_nodes = pd.DataFrame({
    'id': ['nyc', 'sf'],
    'latitude': [40.7128, 37.7749],
    'longitude': [-74.0060, -122.4194],
})

graphistry.nodes(geo_nodes, 'id').gfql([
    call('mercator_layout')
])
```

Schema Effects: Modifies node positions.

modularity_weighted_layout

Prepare a community-weighted layout by weighting edges within and across communities.

Parameters:

Parameter	Type	Required	Description
community_col	string	No	Existing node community column
community_alg	string	No	Community algorithm when `community_col` is omitted
community_params	dict	No	Community algorithm parameters
same_community_w	number	No	Edge weight for within-community edges
cross_community_w	number	No	Edge weight for cross-community edges
edge_influence	number	No	Graphistry edge influence setting
engine	string	No	Engine (`'auto'`, `'pandas'`, `'cudf'`)

Example:

```
g.gfql([
    call('modularity_weighted_layout', {'community_col': 'department'})
])
```

Schema Effects: Adds edge weight columns and binds edge weight. May add a community column when `community_col` is omitted.

10.5.14.5 Filtering and Transformation Methods

filter_nodes_by_dict

Filter nodes based on attribute values.

Parameters:

Parameter	Type	Required	Description
filter_dict	dict	Yes	Dictionary of attribute: value pairs to match

Examples:

```
# Filter by single attribute
g.gfql([
    call('filter_nodes_by_dict', {
        'filter_dict': {'type': 'person'}
    })
])

# Filter by multiple attributes
g.gfql([
    call('filter_nodes_by_dict', {
        'filter_dict': {'type': 'server', 'status': 'active'}
    })
])
```

Schema Effects: None (only filters existing data).

filter_edges_by_dict

Filter edges based on attribute values.

Parameters:

Parameter	Type	Required	Description
filter_dict	dict	Yes	Dictionary of attribute: value pairs to match

Example:

```
g.gfql([
    call('filter_edges_by_dict', {
        'filter_dict': {'weight': 1.0, 'type': 'strong'}
    })
])
```

Schema Effects: None (only filters existing data).

hop

Traverse the graph N steps from current nodes.

Parameters:

Parameter	Type	Required	Description
hops	integer	No*	Number of hops (required unless to_fixed_point=True)
to_fixed_point	boolean	No	Traverse until no new nodes found
direction	string	No	'forward', 'reverse', or 'undirected'
edge_match	dict	No	Filter edges during traversal
source_node_match	dict	No	Filter source nodes
destination_node_match	dict	No	Filter destination nodes
source_node_query	string	No	Query string for source nodes
edge_query	string	No	Query string for edges
destination_node_query	string	No	Query string for destination nodes
return_as_wave_front	boolean	No	Return only new nodes from last hop

Examples:

```
# Simple N-hop traversal
g.gfql([
    n({'id': 'start'}),
    call('hop', {'hops': 2, 'direction': 'forward'})
])

# Traverse to fixed point
g.gfql([
    n({'infected': True}),
    call('hop', {
        'to_fixed_point': True,
        'direction': 'undirected'
    })
])

# Filtered traversal
g.gfql([
    n({'type': 'server'}),
    call('hop', {
        'hops': 3,
        'edge_match': {'protocol': 'ssh'},
        'destination_node_match': {'status': 'active'}
    })
])
```

Schema Effects: None (returns subgraph).

collapse

Merge nodes based on a shared attribute value.

Parameters:

Parameter	Type	Required	Description
column	string	Yes	Column to group nodes by
attribute_columns	list[string]	No	Columns to aggregate
col_aggregations	dict	No	Aggregation functions per column
self_edges	boolean	No	Whether to keep self-edges

Example:

```
# Collapse by department
g.gfql([
    call('collapse', {
        'column': 'department',
        'self_edges': False
    })
])
```

Schema Effects: Modifies node structure based on collapse.

drop_nodes

Remove nodes based on a column value.

Parameters:

Parameter	Type	Required	Description
nodes	list or dict	Yes	Node IDs to drop (list) or filter specification (dict)

Example:

```
# Drop specific nodes by ID
g.gfql([
    call('drop_nodes', {'nodes': ['node_id_1', 'node_id_2']})
])

# Drop nodes matching a filter - use filter_nodes_by_dict first, then drop
inactive = g._nodes[g._nodes['status'] == 'inactive']['id'].tolist()
g.gfql([
    call('drop_nodes', {'nodes': inactive})
])
```

Schema Effects: None (only removes nodes).

keep_nodes

Keep only nodes where a column is True.

Parameters:

Parameter	Type	Required	Description
nodes	list or dict	Yes	Node IDs to keep (list) or filter specification (dict)

Example:

```
# Keep specific nodes by ID
g.gfql([
    call('keep_nodes', {'nodes': ['node_id_1', 'node_id_2']})
])

# Keep nodes matching a filter - use dict form for column-based filtering
g.gfql([
    call('keep_nodes', {'nodes': {'importance': [True]}})
])
```

Schema Effects: None (only filters nodes).

materialize_nodes

Generate a node table from edges when only edges are provided.

Parameters:

Parameter	Type	Required	Description
reuse	boolean	No	Whether to reuse existing node table

Example:

```
# Create nodes from edges
g_edges_only = graphistry.edges(edges, 's', 'd')
g_edges_only.gfql([
    call('materialize_nodes')
])
```

Schema Effects: Creates node table if missing.

prune_self_edges

Remove edges where source equals destination.

Parameters: None

Example:

```
g.gfql([
    call('prune_self_edges')
])
```

Schema Effects: None (only removes edges).

10.5.14.6 Visual Encoding Methods

encode_point_color

Map node attributes to colors.

Parameters:

Parameter	Type	Required	Description
column	string	Yes	Column to encode as color
palette	list	No	Color palette
as_continuous	boolean	No	Treat as continuous scale
as_categorical	boolean	No	Treat as categorical
categorical_mapping	dict	No	Explicit value-to-color mapping
default_mapping	string/int	No	Default color for unmapped values

Example:

```
# Categorical color mapping
g.gfql([
    call('encode_point_color', {
        'column': 'department',
        'categorical_mapping': {
            'sales': 'blue',
            'engineering': 'green',
            'marketing': 'red'
        }
    })
])

# Continuous color scale
g.gfql([
    call('encode_point_color', {
        'column': 'risk_score',
        'palette': ['green', 'yellow', 'red'],
        'as_continuous': True
    })
])
```

Schema Effects: Adds color encoding column.

encode_edge_color

Map edge attributes to colors.

Parameters:

Parameter	Type	Required	Description
column	string	Yes	Column to encode as color
palette	list	No	Color palette
as_continuous	boolean	No	Treat as continuous scale
as_categorical	boolean	No	Treat as categorical
categorical_mapping	dict	No	Explicit value-to-color mapping
default_mapping	string/int	No	Default color for unmapped values

Example:

```
g.gfql([
    call('encode_edge_color', {
        'column': 'relationship_type',
        'categorical_mapping': {
            'friend': 'blue',
            'colleague': 'green',
            'family': 'purple'
        }
    })
])
```

Schema Effects: Adds color encoding column to edges.

encode_point_size

Map node attributes to sizes.

Parameters:

Parameter	Type	Required	Description
column	string	Yes	Column to encode as size
categorical_mapping	dict	No	Value-to-size mapping
default_mapping	number	No	Default size

Example:

```
g.gfql([
    call('encode_point_size', {
        'column': 'importance',
        'categorical_mapping': {
            'low': 10,
            'medium': 20,
            'high': 40
        }
    })
])
```

Schema Effects: Adds size encoding column.

encode_point_icon

Map node attributes to icons.

Parameters:

Parameter	Type	Required	Description
column	string	Yes	Column to encode as icon
categorical_mapping	dict	No	Value-to-icon mapping
default_mapping	string	No	Default icon

Example:

```
g.gfql([
    call('encode_point_icon', {
        'column': 'device_type',
        'categorical_mapping': {
            'server': 'server',
            'laptop': 'laptop',
            'phone': 'mobile'
        }
    })
])
```

Schema Effects: Adds icon encoding column.

10.5.14.7 Utility Methods

name

Set the visualization name.

Parameters:

Parameter	Type	Required	Description
name	string	Yes	Name for the visualization

Example:

```
g.gfql([
    call('name', {'name': 'Network Analysis Results'})
])
```

Schema Effects: None (sets metadata).

description

Set the visualization description.

Parameters:

Parameter	Type	Required	Description
description	string	Yes	Description text

Example:

```
g.gfql([
    call('description', {
        'description': 'PageRank analysis of social network'
    })
])
```

Schema Effects: None (sets metadata).

10.5.14.8 Error Handling

Call operations validate all parameters and will raise specific errors:

```
from graphistry.compute.exceptions import GFQLTypeError, ErrorCode

try:
    # Wrong: function not in safelist
    g.gfql([call('invalid_function')])
```

(continues on next page)

(continued from previous page)

```
except GFQLTypeError as e:
    print(f"Error {e.code}: {e.message}")  # E303: Function not in safelist

try:
    # Wrong: missing required parameter
    g.gfql([call('filter_nodes_by_dict')])
except GFQLTypeError as e:
    print(f"Error {e.code}: {e.message}")  # E105: Missing required parameter

try:
    # Wrong: invalid parameter type
    g.gfql([call('hop', {'hops': 'two'})])
except GFQLTypeError as e:
    print(f"Error {e.code}: {e.message}")  # E201: Type mismatch
```

Common Error Codes:

- **E303**: Function not in safelist
- **E105**: Missing required parameter
- **E201**: Parameter type mismatch
- **E303**: Unknown parameter
- **E301**: Required column not found (runtime)

10.5.14.9 Best Practices

1. **Use Specific Algorithms**: Instead of generic "pagerank", use the appropriate compute method:

   ```
   # Good: Explicit algorithm selection
   call('compute_cugraph', {'alg': 'pagerank'})  # GPU
   call('compute_igraph', {'alg': 'pagerank'})   # CPU

   # Bad: Non-existent generic method
   call('pagerank')  # ERROR: Not in safelist
   ```

2. **Filter Early**: Place filtering operations early in chains:

   ```
   # Good: Filter before expensive operations
   g.gfql([
       call('filter_nodes_by_dict', {'filter_dict': {'active': True}}),
       call('compute_cugraph', {'alg': 'pagerank'})
   ])
   ```

3. **Name Output Columns**: Use descriptive column names:

   ```
   # Good: Clear column naming
   call('compute_cugraph', {
       'alg': 'louvain',
       'out_col': 'community_id'
   })
   ```

4. **Check Schema Effects**: Be aware of columns added by operations:

```python
# After get_degrees, these columns exist - use let() for mixed operations:
from graphistry import let, ref, call, n, gt

g.gfql(let({
    'enriched': call('get_degrees', {
        'col': 'total',
        'degree_in': 'incoming',
        'degree_out': 'outgoing'
    }),
    'filtered': ref('enriched', [n({'total': gt(10)})])  # Filter on degree
}))
```

10.5.14.10 See Also

- *GFQL Quick Reference* - GFQL quick reference
- *GFQL Specifications* - Complete GFQL specification
- *GFQL Operator Reference* - Predicate reference for filtering

10.5.15 GFQL Policy Hooks

Policy hooks provide external control over GFQL query execution, enabling security, resource management, and usage tracking.

10.5.15.1 Quick Start

```python
from graphistry.compute.gfql.policy import PolicyException

def my_policy(context):
    # Deny remote data loading for specific datasets
    if context.get('is_remote'):
        # For remote operations, current_ast is ASTRemoteGraph
        ast = context.get('current_ast')
        if hasattr(ast, 'dataset_id') and ast.dataset_id == 'forbidden':
            raise PolicyException('preload', 'Access denied', code=403)

# Apply policy to query
g.gfql(query, policy={'preload': my_policy})
```

10.5.15.2 Policy Phases

Policies are invoked at these phases:

preload
: Before data is loaded (local or remote). Can prevent data access.

postload
: After data is loaded. Can check size/content and deny further processing.

prelet
: Before `let()` DAG execution starts. Can control entire DAG execution and validate DAG structure.

postlet
: After `let()` DAG execution completes (even on error). Can track DAG-level performance and enforce DAG-level policies.

prechain
: Before chain operations execute. Can control entire chain execution and validate chain structure.

postchain
: After chain operations complete (even on error). Can track chain-level performance and enforce chain-level policies.

preletbinding
: Before each binding execution in `let()` DAGs. Can control per-binding execution and validate dependencies.

postletbinding
: After each binding execution (even on error). Can track binding performance and enforce per-binding policies.

precall
: Before method execution (hop, filter, etc.). Can control operations and validate parameters.

postcall
: After method execution. Can validate result size, track execution time, and log performance.

precompile
: Experimental exact-key hook before local Cypher string queries are compiled. It is not included in `pre` shortcut expansion.

postcompile
: Experimental exact-key hook after local Cypher string-query compilation succeeds or fails. It is not included in `post` shortcut expansion.

10.5.15.3 Context Fields

The context dictionary passed to policy functions contains:

Always present:

- `phase`: Current phase ('preload', 'postload', 'prelet', 'postlet', 'prechain', 'postchain', 'precall', 'postcall', 'preletbinding', 'postletbinding', 'precompile', 'postcompile')
- `hook`: Hook name (same as phase, useful for shared handlers)
- `_policy_depth`: Internal recursion counter

Usually present:

- `query`: Global/original query AST (None in call context)
- `current_ast`: Current sub-AST being executed (None in call context for method calls)
- `query_type`: Type of query ('chain', 'dag', 'single', 'call')

Phase-specific:

- `plottable`: Graph instance (postload/precall/postcall phases)
- `graph_stats`: Data statistics as GraphStats TypedDict (postload/precall/postcall phases)
- `call_op`: Operation name (precall/postcall phases only)
- `call_params`: Operation parameters (precall/postcall phases only)

- `execution_time`: Method execution duration in seconds (postcall phase only)
- `success`: Execution success flag (postcall/postlet/postchain/postletbinding/postcompile phases)
- `error`: Error message string (post* phases when success=False)
- `error_type`: Error type name (post* phases when success=False)

Compiler-specific (`precompile`/`postcompile` phases only):

- `compile_language`: Source language for string-query compilation.
- `compile`: `CompileSummary` for `postcompile` with stable scalar fields such as `language`, `success`, `compiler_phase`, `code`, `context`, and optional error details. Private parser, binder, lowering, and DataFrame objects are not exposed.

Binding-specific (preletbinding/postletbinding phases only):

- `binding_name`: Name of the current binding being executed
- `binding_index`: Execution order of this binding (0-indexed)
- `total_bindings`: Total number of bindings in the let expression
- `binding_dependencies`: List of binding names this binding depends on
- `binding_ast`: The AST object being bound (the value in let({name: ast}))

Hierarchy/Tracing fields (all phases):

- `execution_depth`: Nesting depth (0=query, 1=let/chain, 2=binding/op, 3=call)
- `operation_path`: Unique operation identifier like "query.dag.binding:hg.call:hypergraph"
- `parent_operation`: Parent operation path (for OpenTelemetry span relationships)

Context-specific:

- `is_remote`: True for remote data operations (ASTRemoteGraph)
- `engine`: Current engine value when available

10.5.15.4 GraphStats Type

The `graph_stats` field provides typed statistics:

```
from graphistry.compute.gfql.policy import GraphStats

# GraphStats is a TypedDict with:
# - nodes: int (number of nodes)
# - edges: int (number of edges)
# - node_bytes: int (memory usage)
# - edge_bytes: int (memory usage)
```

10.5.15.5 Examples

Limit Data Size

```
def size_limit_policy(context):
    if context['phase'] == 'postload':
        stats = context.get('graph_stats', {})
        if stats.get('nodes', 0) > 10000:
            raise PolicyException(
                'postload',
                f"Too many nodes: {stats['nodes']}",
                code=413
            )

g.gfql(query, policy={'postload': size_limit_policy})
```

Control Operation Execution and Performance

```
def operation_control_policy(context):
    if context['phase'] == 'precall':
        # Validate operation parameters before execution
        op = context.get('call_op', '')
        params = context.get('call_params', {})

        # Deny expensive operations
        if op == 'hop' and params.get('hops', 0) > 3:
            raise PolicyException(
                'precall',
                f"Too many hops: {params['hops']} > 3",
                code=413
            )

    elif context['phase'] == 'postcall':
        # Track execution performance
        exec_time = context.get('execution_time', 0)
        success = context.get('success', False)

        if not success:
            raise PolicyException(
                'postcall',
                'Operation failed',
                code=500
            )

        # Log slow operations
        if exec_time > 5.0:  # 5 seconds
            print(f"Slow operation detected: {exec_time:.2f}s")

        # Validate result size
        stats = context.get('graph_stats', {})
        if stats.get('nodes', 0) > 50000:
            raise PolicyException(
                'postcall',
```

(continues on next page)

(continued from previous page)

```
                f"Result too large: {stats['nodes']} nodes",
                code=413
            )

g.gfql(query, policy={
    'precall': operation_control_policy,
    'postcall': operation_control_policy
})
```

Control Remote Access

```
def remote_access_policy(context):
    if context.get('is_remote'):
        # Check JWT token for remote operations
        ast = context['current_ast']
        if hasattr(ast, 'token') and not ast.token:
            raise PolicyException(
                'preload',
                'Authentication required',
                code=401
            )

g.gfql(query, policy={'preload': remote_access_policy})
```

Per-Binding Control

```
def binding_policy(context):
    # Control execution of specific bindings
    if context['phase'] == 'preletbinding':
        binding_name = context.get('binding_name')
        deps = context.get('binding_dependencies', [])

        # Deny bindings with too many dependencies
        if len(deps) > 5:
            raise PolicyException(
                'preletbinding',
                f"Binding '{binding_name}' has too many dependencies: {len(deps)}",
                code=413
            )

    elif context['phase'] == 'postletbinding':
        # Track binding performance
        binding_name = context.get('binding_name')
        success = context.get('success', False)

        if not success:
            error = context.get('error', 'Unknown error')
            print(f"Binding '{binding_name}' failed: {error}")

from graphistry.compute.ast import ASTLet, n, call

dag = ASTLet({
```

(continues on next page)

(continued from previous page)

```
    'people': n({'type': 'person'}),
    'orgs': n({'type': 'org'}),
    'connections': call('hypergraph', {})
})

g.gfql(dag, policy={
    'preletbinding': binding_policy,
    'postletbinding': binding_policy
})
```

Track Usage

```
def create_usage_tracker():
    stats = {'calls': 0, 'data_loaded': 0, 'execution_times': []}

    def track(context):
        if context['phase'] == 'precall':
            stats['calls'] += 1
        elif context['phase'] == 'postcall':
            # Track execution performance
            exec_time = context.get('execution_time', 0)
            stats['execution_times'].append(exec_time)
        elif context['phase'] == 'postload':
            data = context.get('graph_stats', {})
            stats['data_loaded'] += data.get('nodes', 0)

    return track, stats

tracker, stats = create_usage_tracker()
g.gfql(query, policy={
    'postload': tracker,
    'precall': tracker,
    'postcall': tracker
})
print(f"Usage: {stats}")
```

Shared Handler

```
def universal_policy(context):
    hook = context['hook']  # Which hook fired

    if hook == 'preload':
        # Pre-execution checks
        pass
    elif hook == 'postload':
        # Data validation
        pass
    elif hook == 'precall':
        # Operation control and parameter validation
        pass
    elif hook == 'postcall':
        # Performance tracking and result validation
```

(continues on next page)

(continued from previous page)

```
        pass

# Use same handler for all phases
g.gfql(query, policy={
    'preload': universal_policy,
    'postload': universal_policy,
    'precall': universal_policy,
    'postcall': universal_policy
})
```

10.5.15.6 Policy Shortcuts

To reduce boilerplate in common patterns, GFQL policies support shortcuts that expand to multiple hooks automatically. This is especially useful for cross-cutting concerns like telemetry, authentication, and resource management.

Shortcuts Reference

Shortcut	Expands To	Use Case
`'pre'`	All 5 pre* hooks (preload, prelet, prechain, preletbinding, precall)	OpenTelemetry span creation, authentication, pre-execution validation
`'post'`	All 5 post* hooks (postload, postlet, postchain, postletbinding, postcall)	OpenTelemetry span cleanup, resource cleanup, post-execution validation
`'load'`	preload + postload	Query-level hooks for data loading control
`'let'`	prelet + postlet	DAG-level hooks for let() execution control
`'chain'`	prechain + postchain	Chain-level hooks for chain operation control
`'binding'`	preletbinding + postletbinding	Binding-level hooks for per-binding control
`'call'`	precall + postcall	Operation-level hooks for method call control

Before/After Comparison

Without shortcuts (10 keys):

```
# Traditional approach - verbose
policy = {
    'preload': create_span,
    'postload': end_span,
    'prelet': create_span,
    'postlet': end_span,
    'prechain': create_span,
    'postchain': end_span,
    'preletbinding': create_span,
    'postletbinding': end_span,
    'precall': create_span,
    'postcall': end_span
}
```

With shortcuts (2 keys):

```
# Shortcuts approach - concise
policy = {
```

(continues on next page)

(continued from previous page)

```
    'pre': create_span,
    'post': end_span
}
```

Both are functionally equivalent and produce the same behavior.

Composition Behavior

When multiple shortcuts apply to the same hook, their handlers automatically compose:

```
from graphistry.compute.gfql.policy import expand_policy, debug_policy

def auth_check(ctx):
    """General authentication check"""
    pass

def rate_limit(ctx):
    """Rate limiting for calls"""
    pass

def validate_params(ctx):
    """Specific parameter validation"""
    pass

policy = {
    'pre': auth_check,        # Applies to ALL pre* hooks
    'call': rate_limit,       # Applies to precall + postcall
    'precall': validate_params  # Applies only to precall
}

# At precall, handlers execute in order: auth_check → rate_limit → validate_params
# At postcall, handlers execute in reverse (LIFO): rate_limit → auth_check
```

Composition Order Rules

- **Pre hooks** execute in forward order: general → scope → specific
- **Post hooks** execute in reverse order (LIFO cleanup): specific → scope → general
- This ensures proper setup/cleanup semantics (like try/finally blocks)

Multi-Policy Server Pattern

Shortcuts compose naturally for scenarios where multiple orthogonal policies need to be applied:

```
# Server scenario: telemetry + security + resource limits
policy = {
    'pre': create_otel_span,       # OpenTelemetry tracing
    'post': end_otel_span,         # Span cleanup
    'postload': check_size_limits,  # Resource limits after data load
    'precall': validate_jwt_token   # Security validation before operations
}

# This composes cleanly:
# - All pre* hooks get telemetry spans
# - postload gets both telemetry cleanup + size checking
```

(continues on next page)

(continued from previous page)

```
# - precall gets telemetry + JWT validation
# - Other post* hooks get just telemetry cleanup
```

Debug Helper

Use `debug_policy()` to see how shortcuts expand:

```
from graphistry.compute.gfql.policy import debug_policy

policy = {
    'pre': auth,
    'call': rate_limit,
    'precall': validate
}

# Show expansion and composition order
debug_policy(policy)
```

Output:

```
preload          [auth (from 'pre')]
prelet           [auth (from 'pre')]
prechain         [auth (from 'pre')]
preletbinding    [auth (from 'pre')]
precall          [auth (from 'pre'), rate_limit (from 'call'), validate (from 'precall')]
postcall         [rate_limit (from 'call'), auth (from 'pre')] ← reversed
postload         [auth (from 'pre')]
postlet          [auth (from 'pre')]
postchain        [auth (from 'pre')]
postletbinding   [auth (from 'pre')]
```

Backward Compatibility

- Full hook names (like `'preload'`) still work and can be mixed with shortcuts
- Shortcuts are entirely optional - use them only when they simplify your code
- No performance overhead - expansion happens once per query

OpenTelemetry Example

Using shortcuts, OpenTelemetry span tracing reduces from 10 hook keys to just 2:

```
from opentelemetry import trace
from opentelemetry.trace import Status, StatusCode

tracer = trace.get_tracer(__name__)
span_map = {}  # operation_path → span

def create_span(ctx):
    """Start span in pre* hooks"""
    # Get parent span using parent_operation
    parent_span = span_map.get(ctx.get('parent_operation'))

    # Create span with unique operation_path as name
```

(continues on next page)

(continued from previous page)

```
    span = tracer.start_span(
        ctx['operation_path'],
        parent=parent_span
    )

    # Add span attributes from context
    span.set_attribute('execution_depth', ctx['execution_depth'])
    span.set_attribute('query_type', ctx.get('query_type', 'unknown'))

    if ctx.get('binding_name'):
        span.set_attribute('binding_name', ctx['binding_name'])
    if ctx.get('call_op'):
        span.set_attribute('call_op', ctx['call_op'])

    # Store span for children and post hook
    span_map[ctx['operation_path']] = span

def end_span(ctx):
    """End span in post* hooks"""
    span = span_map.pop(ctx['operation_path'], None)
    if not span:
        return

    # Add result attributes
    if ctx.get('graph_stats'):
        stats = ctx['graph_stats']
        span.set_attribute('nodes', stats.get('nodes', 0))
        span.set_attribute('edges', stats.get('edges', 0))

    # Handle errors
    if not ctx.get('success', True):
        span.set_status(
            Status(StatusCode.ERROR, ctx.get('error', 'Unknown error'))
        )

    span.end()

# Apply to all hook phases using shortcuts (2 keys instead of 10!)
policy = {
    'pre': create_span,   # Expands to all 5 pre* hooks
    'post': end_span      # Expands to all 5 post* hooks
}

result = g.gfql(my_query, policy=policy)
```

This creates a proper span hierarchy matching the query execution tree, with each operation having a unique `operation_path` and correct parent relationships.

10.5.15.7 PolicyException

Deny operations by raising `PolicyException`:

```
from graphistry.compute.gfql.policy import PolicyException

raise PolicyException(
    phase='preload',      # Which phase denied
    reason='Forbidden',   # Human-readable reason
    code=403,             # HTTP-like status code
    **kwargs              # Additional context
)
```

The exception can be enriched with additional fields for logging/debugging.

10.5.15.8 Thread Safety

Policy execution is thread-safe with built-in recursion prevention. Policies are not invoked recursively when operations trigger internal queries (depth limit of 1).

10.5.15.9 Remote Data Loading

Policies can control remote data operations (`ASTRemoteGraph`). When `is_remote` is True in the context, the operation involves loading data from a remote source:

```
def remote_data_policy(context):
    # Check remote operations in preload phase
    if context['phase'] == 'preload' and context.get('is_remote'):
        ast = context.get('current_ast')

        # For ASTRemoteGraph, check dataset_id
        if hasattr(ast, 'dataset_id'):
            if ast.dataset_id in banned_datasets:
                raise PolicyException('preload', 'Dataset blocked')

            # Check for JWT token
            if hasattr(ast, 'token') and not validate_jwt(ast.token):
                raise PolicyException('preload', 'Invalid token', code=401)

    # Check size after remote data loads
    elif context['phase'] == 'postload' and context.get('is_remote'):
        stats = context.get('graph_stats', {})
        if stats.get('nodes', 0) > remote_limit:
            raise PolicyException('postload', 'Remote data too large')
```

Remote operations trigger both preload and postload hooks, allowing control before and after data transfer.

10.5.15.10 Query Types

Policies work with different GFQL query patterns:

Chain queries - Sequential operations:

```
# query_type will be 'chain'
g.gfql([n(), e(), n()], policy=policy_dict)
```

DAG queries - Named bindings with dependencies:

```
# query_type will be 'dag'
g.gfql({'persons': n({'type': 'person'})}, policy=policy_dict)
```

Call operations - Method invocations:

```
# query_type will be 'single', precall and postcall phases triggered
from graphistry.compute.ast import call
g.gfql(call('hop', {'hops': 2}), policy={
    'precall': my_precall_policy,
    'postcall': my_postcall_policy
})
```

Each query type provides appropriate context to the policy for decision making.

10.5.15.11 Integration with Hub

The policy system is designed for Graphistry Hub integration:

1. Hub creates policies based on user tier/permissions
2. Policies enforce resource limits and feature access
3. Usage tracking for billing/analytics
4. JWT token validation for remote operations

```
# Hub example
def create_tier_policy(tier='free'):
    limits = {
        'free': {'max_nodes': 1000},
        'pro': {'max_nodes': 100000}
    }

    def policy(context):
        if context['phase'] == 'postload':
            stats = context.get('graph_stats', {})
            if stats.get('nodes', 0) > limits[tier]['max_nodes']:
                raise PolicyException(
                    'postload',
                    f'{tier} tier limit exceeded',
                    code=403
                )

    return policy
```

10.5.15.12 Advanced Topics

Policy Composition

Combine multiple policies using composition patterns:

```
def compose_policies(*policies):
    """Compose multiple policies into one."""
    def composed(context):
        for policy in policies:
            policy(context)  # Each can raise PolicyException
    return composed

# Use composed policy
combined = compose_policies(
    size_limit_policy,
    rate_limit_policy,
    tier_policy
)
g.gfql(query, policy={'postload': combined})
```

Stateful Policies with Closures

Track state across multiple queries:

```
def create_rate_limiter(max_per_minute=60):
    from collections import deque
    from time import time

    calls = deque()

    def policy(context):
        if context['phase'] == 'preload':
            now = time()
            # Remove calls older than 1 minute
            while calls and calls[0] < now - 60:
                calls.popleft()

            if len(calls) >= max_per_minute:
                raise PolicyException(
                    'preload',
                    'Rate limit exceeded',
                    code=429
                )
            calls.append(now)

    return policy
```

Testing Policies

Test policies in isolation:

```
def test_policy():
    # Create mock context
    context = {
```

(continues on next page)

(continued from previous page)

```
        'phase': 'postload',
        'graph_stats': {'nodes': 5000},
        '_policy_depth': 0
    }

    # Test acceptance
    my_policy(context)  # Should not raise

    # Test denial
    context['graph_stats']['nodes'] = 50000
    with pytest.raises(PolicyException) as exc:
        my_policy(context)
    assert exc.value.code == 413
```

Performance Considerations

- Policies execute synchronously - keep them lightweight
- Use caching for expensive validations
- Consider async patterns for external calls (future enhancement)
- Recursion prevention adds minimal overhead (depth limit of 1)

Debugging Policies

Use logging to debug policy decisions:

```
import logging
logger = logging.getLogger(__name__)

def debug_policy(context):
    phase = context['phase']
    logger.debug(f"Policy called: phase={phase}")

    if phase == 'postload':
        stats = context.get('graph_stats', {})
        logger.debug(f"Graph stats: {stats}")

        if stats.get('nodes', 0) > limit:
            logger.warning(f"Denying: {stats['nodes']} > {limit}")
            raise PolicyException(...)

    logger.debug(f"Policy accepted in {phase}")
```

10.5.15.13 API Reference

Main Interface

```
# Using full hook names
g.gfql(query, policy={
    'preload': preload_function,              # Optional
    'postload': postload_function,            # Optional
    'prelet': prelet_function,                # Optional
    'postlet': postlet_function,              # Optional
    'prechain': prechain_function,            # Optional
    'postchain': postchain_function,          # Optional
    'preletbinding': preletbinding_function,  # Optional
    'postletbinding': postletbinding_function,# Optional
    'precall': precall_function,              # Optional
    'postcall': postcall_function             # Optional
})

# Or using shortcuts (expands to full hook names)
g.gfql(query, policy={
    'pre': pre_function,     # Expands to all pre* hooks
    'post': post_function,   # Expands to all post* hooks
    'load': load_function,   # Expands to preload + postload
    'let': let_function,     # Expands to prelet + postlet
    'chain': chain_function, # Expands to prechain + postchain
    'binding': binding_fn,   # Expands to preletbinding + postletbinding
    'call': call_function    # Expands to precall + postcall
})

# Shortcuts can be mixed with full hook names
g.gfql(query, policy={
    'pre': general_handler,
    'postload': specific_size_check  # Overrides 'post' for postload
})
```

Imports

```
from graphistry.compute.gfql.policy import (
    PolicyException,  # Exception class for denying operations
    PolicyContext,    # TypedDict for context parameter
    GraphStats,       # TypedDict for graph statistics
    PolicyFunction,   # Type alias for policy functions
    PolicyDict,       # Type alias for policy dictionary
    expand_policy,    # Expand shortcuts to full hook names (internal use)
    debug_policy      # Debug helper to visualize expansion
)
```

PolicyException Parameters

- `phase` (str): Phase where denial occurred ('preload', 'postload', 'prelet', 'postlet', 'prechain', 'postchain', 'preletbinding', 'postletbinding', 'precall', 'postcall')
- `reason` (str): Human-readable explanation
- `code` (int): HTTP-like status code (default: 403)

- `query_type` (str, optional): Type of query being executed
- `data_size` (dict, optional): Graph statistics at time of denial

Common HTTP Status Codes

- `401`: Unauthorized (authentication required)
- `403`: Forbidden (authenticated but not allowed)
- `413`: Payload too large (data size limit exceeded)
- `429`: Too many requests (rate limit exceeded)
- `503`: Service unavailable (resource constraints)

10.5.16 Strict Schema Checks

GFQL can check Cypher queries against the graph schema before the query runs. For Cypher users, this means typos in labels, variables, and properties fail early with a validation error instead of producing confusing empty results or later execution failures.

Use `g.gfql_validate(...)` when you want a report without running the query. Use `g.gfql(..., validate=True)` when you want the same checks before execution.

Local Cypher execution uses these schema checks. Environment variables or keyword arguments do not switch local Cypher execution back to a looser mode.

10.5.16.1 What Gets Checked

For Cypher queries, strict schema checks verify:

- Labels used in `MATCH` exist in the graph schema.
- Variables referenced in `WHERE`, `RETURN`, `UNWIND`, and `CALL` are in scope.
- Property names exist for the node or edge variable they are read from.

Invalid queries raise `GFQLValidationError` before execution. Valid queries run the same as before.

It does **not** check every dataframe value's Python or Arrow type. This page is about Cypher names and schema references.

10.5.16.2 Validate Without Running

`g.gfql_validate(...)` returns structured diagnostics and never executes query operators:

```
report = g.gfql_validate(
    "MATCH (p:Person) RETURN p.name AS name",
    strict=True,
)
if not report["ok"]:
    for diag in report["diagnostics"]:
        print(diag["code"], diag["message"])
```

10.5.16.3 Validate Before Running

Use `validate=True` on `g.gfql(...)` to run the same checks before executing the query:

```
result = g.gfql(
    "MATCH (p:Person) RETURN p.name AS name",
    validate=True,
)
```

These APIs are the recommended way to make validation explicit in request handlers, notebooks, and CI checks.

10.5.16.4 Configuration Notes

Most users do not need to configure these checks directly. Prefer `g.gfql_validate(...)` or `g.gfql(..., validate=True)`.

Code can also set a catalog metadata flag:

```
from graphistry.compute.gfql.ir.compilation import GraphSchemaCatalog
catalog = GraphSchemaCatalog.from_schema_parts(
    node_columns={"id", "label__Person"},
    edge_columns={"src", "dst", "label__KNOWS"},
    metadata={"strict": True},
)
```

or a process-wide environment variable:

```
export GRAPHISTRY_GFQL_STRICT_SCHEMA=true
```

Truthy values: `1`, `true`, `yes`, `on` (case-insensitive). Falsy / unset: anything else (default `false`).

Treat these as opt-in signals, not as switches that disable validation. Setting them to `false` or leaving them unset does not make local Cypher execution looser.

The explicit validation APIs (`g.gfql_validate(strict=True)` and `g.gfql(validate=True)`) are unaffected by these helpers.

10.5.16.5 Error Messages

Schema-check failures raise `GFQLValidationError` with deterministic messages and sorted availability hints:

```
Cypher label is missing from the graph schema.
Use labels that exist in the node schema or extend the schema catalog.
available labels: [Comment, Person, Post]
```

Use the message text to identify the gap, then either fix the query or extend the catalog while iterating.

10.5.16.6 When To Use It

Recommended:

- Production query gates where unknown identifiers should fail closed.
- CI / pre-merge quality bars over a curated catalog.
- Multi-team environments where the graph schema is managed centrally.

Before relying on these checks:

- Exploratory / notebook usage should make sure GFQL knows the labels and properties in the graph being queried.
- Pipelines with intentionally partial schemas should validate only after the schema has enough labels and properties for the queries being checked.

10.5.16.7 Recommended usage

Use explicit validation for the tightest path:

1. **Validate each call explicitly** — for example, in a request handler that should never accept unknown labels, variables, or properties:

```
result = g.gfql(query, validate=True)
```

This is the clearest option for application code that wants strict checks.

Clearing the env var or removing the catalog flag does not make local Cypher execution looser. Explicit validation remains strict when requested.

10.5.16.8 See also

- *GFQL Validation Fundamentals* — preflight + execution-time validation primitives, including `g.gfql_validate(...)`.
- *Cypher Syntax In GFQL* — Cypher syntax reference and preflight examples.

10.5.17 Declarative Graph Schemas

GFQL accepts public schema declarations through the stable `graphistry.schema` import path. Use this when application code owns a graph contract and wants Cypher preflight checks to fail before query execution. The API is experimental in this release: the import path and core declaration objects are intended to be stable, while inference, coercion, remote transport, and planner use are still follow-on surfaces.

The schema is optional. When you provide one, PyGraphistry uses it as the declared contract for local GFQL validation. When you do not provide one, validation falls back to the columns already visible on the bound local `nodes` and `edges` dataframes. If neither a public schema nor local dataframes are available, Cypher validation still parses and compiles the query, but it cannot reject unknown labels or properties because there is no schema to check against.

```
import graphistry
import pandas as pd
import pyarrow as pa
from graphistry.schema import EdgeType, GraphSchema, NodeType
```

(continues on next page)

(continued from previous page)

```
Person = NodeType(
    "Person",
    pa.schema([
        pa.field("id", pa.int64(), nullable=False),
        pa.field("name", pa.large_string()),
    ]),
)
Company = NodeType(
    "Company",
    pa.schema([
        pa.field("id", pa.int64(), nullable=False),
        pa.field("name", pa.large_string()),
    ]),
)
WorksAt = EdgeType(
    "WORKS_AT",
    source=Person,
    destination=Company,
    properties=pa.schema([pa.field("since", pa.int64(), nullable=False)]),
)

schema = GraphSchema(
    node_types=[Person, Company],
    edge_types=[WorksAt],
    node_id_column="id",
    edge_source_column="src",
    edge_destination_column="dst",
)

nodes_df = pd.DataFrame({
    "id": [1, 2],
    "name": ["Ada", "Graphistry"],
    "label__Person": [True, False],
    "label__Company": [False, True],
})
edges_df = pd.DataFrame({
    "src": [1],
    "dst": [2],
    "since": [2024],
    "label__WORKS_AT": [True],
})

g = (
    graphistry
    .edges(edges_df, "src", "dst")
    .nodes(nodes_df, "id")
    .bind(schema=schema)
)

g.gfql_validate("MATCH (p:Person)-[:WORKS_AT]->(c:Company) RETURN p.name")
```

10.5.17.1 Schema Objects

`NodeType(name, properties, labels=None)`
: Declares a node contract. `labels` defaults to `(name,)` and maps to the existing GFQL label-column convention `label__<Label>`. `properties` accepts a `pyarrow.Schema`, a GFQL `RowSchema`, or a mapping shorthand such as `{"id": pa.int64(), "name": pa.large_string()}` or `{"id": int, "name": str}`. Arrow schemas are the preferred declaration path because they preserve dtype and nullability.

`EdgeType(name, source, destination, properties=None)`
: Declares an edge contract and topology. `source` and `destination` accept `NodeType` objects, label strings, or label iterables. Edge properties use the same Arrow-aligned schema inputs as node properties.

`GraphSchema(node_types, edge_types, strict=True, ...)`
: Groups node/edge contracts and adapts them to the internal `GraphSchemaCatalog` used by binder/preflight validation. `strict=False` makes schema-bound `g.gfql_validate(...)` permissive by default; callers can still override per call with `g.gfql_validate(..., strict=True)`. A physical node property column must have the same logical type for every node type that declares it, and a physical edge property column must have the same logical type for every edge type that declares it. Use separate column names when two labels or relationship types need incompatible values under the same property name.

`NodeType.to_arrow()` and `EdgeType.to_arrow()`
: Export declarations as `pyarrow.Schema` objects through GFQL's row-schema bridge. Label/type columns are included by default so exports line up with the table columns used by binder/preflight validation.

`NodeType.from_arrow(...)` and `EdgeType.from_arrow(...)`
: Import explicit Arrow declarations back into public schema objects. This is declaration import, not inference: edge imports still require source and destination labels, and graph-level imports require named node/edge entries.

`GraphSchema.to_arrow()` and `GraphSchema.from_arrow(...)`
: Export/import a declaration payload containing per-node/per-edge Arrow schemas plus merged `nodes` and `edges` table schemas. The merged schemas are useful for dataframe boundary validation; the per-type entries preserve type names and edge topology. When the same column is declared with the same Arrow type but different nullability, merged table schemas mark that column as nullable while the per-type declarations keep their original nullability.

Nullability is type-local. If `Cat.lives` is declared non-nullable and `House` does not declare `lives`, `Cat.lives` remains non-nullable in the `Cat` declaration. Boundary validation against a full fused node table still accounts for which labels are active in each row.

10.5.17.2 What Preflight Checks

When a schema is bound to a graph, Cypher preflight checks validate:

- node labels against declared node types,
- node and edge property names against declared properties,
- relationship types against declared edge types, and
- relationship source/destination labels against declared topology when the query provides enough label information.

Invalid queries raise `GFQLValidationError` with structured context.

This is a correctness and documentation surface first: applications can state what labels, relationship types, properties, and topology they expect, then validate user-authored or generated Cypher before running

it. The same typed contract is also the foundation for later inference, coercion, remote transport, and planner/performance work, but this page covers the declared local contract.

10.5.17.3 Schema Effects

Some graph-growing GFQL calls add properties to an existing graph. For example, `CALL graphistry.degree.write()` adds degree columns to nodes, and PageRank-style `.write()` procedures add score columns. When a graph has a bound `GraphSchema`, PyGraphistry now tracks those successful local effects internally and attaches the updated schema snapshot to the returned graph:

```
enriched = g.gfql("CALL graphistry.degree.write()")
enriched.gfql_validate("MATCH (n:Person) RETURN n.degree")
```

This is not a new public API surface. The effect model is internal while schema inference, remote transport, and planner use continue to evolve. It is scoped to local graph results with an explicitly bound schema; remote GFQL requests still do not serialize schema snapshots or effect history.

10.5.17.4 Arrow Boundary Validation

You can also opt into declared-schema checks at Arrow conversion and upload boundaries. This is off by default so existing `plot()`, `upload()`, and `to_arrow()` calls keep their current behavior.

`schema_validate="strict"`
: Requires every declared node/edge schema column to exist and match the declared Arrow type. Non-nullable declared columns must not contain nulls.

`schema_validate="autofix"`
: Performs the same presence and non-null checks, and casts compatible columns to the declared Arrow type after normal Arrow conversion. Existing `validate="autofix"` mixed-type coercion still runs first.

```
# Debug a bound edge table against the schema.
edges_arrow = g.to_arrow(schema_validate="strict")

# Coerce compatible values such as string-encoded integers to the declared
# Arrow type before local conversion. The same option is accepted by plot()
# and upload().
edges_arrow_autofix = g.to_arrow(schema_validate="autofix")

# Validate the node table explicitly.
nodes_arrow = g.validate_arrow_schema("nodes", validate="strict")
```

10.5.17.5 Provided vs. Inferred Schema

In this release, schemas are **provided**, not inferred. You create `NodeType`, `EdgeType`, and `GraphSchema` objects directly and attach them with `graphistry.bind(..., schema=schema)` or `g.bind(schema=schema)`.

Without an explicit `GraphSchema`:

- `g.gfql_validate(...)` can still use local dataframe columns already bound on `g._nodes` and `g._edges` for schema-aware checks.
- It does not infer node types, edge types, Arrow dtypes, nullability, or topology from data.

- A remote-only graph such as `graphistry.bind(dataset_id="...")` has no local dataframe columns, so local validation is limited to syntax, compile, and structural checks unless you also bind a declared schema.

Schema inference from existing plottables is tracked separately from this declared-schema API.

10.5.17.6 Local vs. Remote GFQL

The public schema is consumed by local validation APIs, including:

- `g.gfql_validate("MATCH ...")`
- `g.gfql(..., validate=True)`

`gfql_remote(...)` is different. It compiles Cypher strings locally and sends the resulting GFQL wire payload to the server, but this release does **not** serialize a bound `GraphSchema` into remote GFQL requests. Remote execution therefore still depends on the server-side dataset schema and GFQL support. If you want declared schema checks before a remote call, run `g.gfql_validate(query)` locally first, then call `g.gfql_remote(query)`.

Remote schema transport is planned as a follow-on after the local schema contract and serialization boundary are stable.

10.5.17.7 Compatibility Notes

The public import path is stable:

```
from graphistry.schema import NodeType, EdgeType, GraphSchema
```

Top-level imports are also available:

```
from graphistry import NodeType, EdgeType, GraphSchema
```

This lane exposes declaration, Arrow row-schema import/export, binder/preflight integration, opt-in Arrow boundary validation/coercion, and internal local schema-effect propagation for graph-growing calls. Inference from existing plottables and remote schema transport remain separate follow-on surfaces.

10.5.18 Temporal Predicates Wire Protocol Reference

This document provides a comprehensive reference for how temporal predicates serialize to JSON in the GFQL wire protocol. The wire protocol enables interoperability between Python and other systems.

10.5.18.1 Overview

The wire protocol uses tagged dictionaries to preserve type information during JSON serialization. This enables:

- Cross-language compatibility
- Configuration-driven predicate creation
- Network transport of queries
- Storage of predicate definitions

Key Concept: Wire protocol dictionaries can be used directly in the Python API:

```
# These are equivalent:
pred1 = gt(100)
pred2 = gt(pd.Timestamp("2023-01-01"))
pred3 = gt({"type": "datetime", "value": "2023-01-01T00:00:00", "timezone": "UTC"})
```

10.5.18.2 Comparator / Operator Mapping

- Python predicate helpers: `gt`, `ge`, `lt`, `le`, `eq`, `ne`
- Wire predicate types: `GT`, `GE`, `LT`, `LE`, `EQ`, `NE`
- Same-path chain `where` JSON operators: `gt`, `ge`, `lt`, `le`, `eq`, `neq`
- Row-expression comparators (`where_rows(expr="...")`): `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`

10.5.18.3 WHERE Contexts in Wire JSON

`WHERE`-style filtering appears in three distinct wire shapes:

1. **Same-path chain WHERE (`where=[...]`)** uses lower-case operator keys and alias-column references:

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "name": "a"},
    {"type": "Node", "name": "b"}
  ],
  "where": [
    {"ge": {"left": "a.created_at", "right": "b.created_at"}}
  ]
}
```

2. **Matcher predicates (`filter_dict` / `edge_match`)** use predicate envelopes (`GT/GE/LT/LE/EQ/NE/...`) and can carry typed temporal values:

```
{
  "type": "Node",
  "filter_dict": {
    "created_at": {
      "type": "GT",
      "val": {"type": "datetime", "value": "2024-01-01T00:00:00", "timezone": "UTC"}
    }
  }
}
```

3. **Row-pipeline WHERE** after `rows(...)` supports either:
   - expression string: `{"function": "where_rows", "params": {"expr": "..."} }`
   - predicate dict: `{"function": "where_rows", "params": {"filter_dict": {...}} }`

```
{
  "type": "Call",
  "function": "where_rows",
```

(continues on next page)

(continued from previous page)

```
  "params": {
    "filter_dict": {
      "created_at": {
        "type": "GE",
        "val": {"type": "datetime", "value": "2024-01-01T00:00:00", "timezone": "UTC"}
      }
    }
  }
}
```

10.5.18.4 1. DateTime Comparisons

Python API

```
import pandas as pd
from datetime import datetime
from graphistry import n, e_forward
from graphistry.compute import gt, between

# Using pandas Timestamp
filter1 = n(filter_dict={
    "created_at": gt(pd.Timestamp("2023-01-01 12:00:00"))
})

# Using Python datetime
filter2 = e_forward(edge_match={
    "timestamp": between(
        datetime(2023, 1, 1),
        datetime(2023, 12, 31, 23, 59, 59)
    )
})
```

Wire Protocol (JSON)

```
// GT with datetime
{
    "type": "ASTNode",
    "filter_dict": {
        "created_at": {
            "type": "GT",
            "val": {
                "type": "datetime",
                "value": "2023-01-01T12:00:00",
                "timezone": "UTC"
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
// Between with datetime range
{
    "type": "ASTNode",
    "edge_match": {
        "timestamp": {
            "type": "Between",
            "lower": {
                "type": "datetime",
                "value": "2023-01-01T00:00:00",
                "timezone": "UTC"
            },
            "upper": {
                "type": "datetime",
                "value": "2023-12-31T23:59:59",
                "timezone": "UTC"
            },
            "inclusive": true
        }
    }
}
```

Round-trip Example

```
# Create predicate
from graphistry.compute import gt
pred = gt(pd.Timestamp("2023-01-01 12:00:00"))

# Serialize to JSON
json_data = pred.to_json()
print(json_data)
# Output: {
#     'type': 'GT',
#     'val': {
#         'type': 'datetime',
#         'value': '2023-01-01T12:00:00',
#         'timezone': 'UTC'
#     }
# }

# Deserialize from JSON
from graphistry.compute.predicates.numeric import GT
pred2 = GT.from_json(json_data)
# pred2 is functionally equivalent to pred
```

10.5.18.5 2. Date-Only Comparisons

Python API

```python
from datetime import date
from graphistry.compute import eq, ge

# Date equality
filter1 = n(filter_dict={
    "event_date": eq(date(2023, 6, 15))
})

# Date range check
filter2 = n(filter_dict={
    "start_date": ge(date(2023, 1, 1))
})
```

Wire Protocol (JSON)

```json
// Date equality
{
    "type": "ASTNode",
    "filter_dict": {
        "event_date": {
            "type": "EQ",
            "val": {
                "type": "date",
                "value": "2023-06-15"
            }
        }
    }
}

// Date greater than or equal
{
    "type": "ASTNode",
    "filter_dict": {
        "start_date": {
            "type": "GE",
            "val": {
                "type": "date",
                "value": "2023-01-01"
            }
        }
    }
}
```

10.5.18.6 3. Time-Only Comparisons

Python API

```python
from datetime import time
from graphistry import n, e_forward
from graphistry.compute import is_in, between

# Specific times
filter1 = n(filter_dict={
    "event_time": is_in([
        time(9, 0, 0),
        time(12, 0, 0),
        time(17, 0, 0)
    ])
})

# Time range
filter2 = e_forward(edge_match={
    "daily_schedule": between(
        time(9, 0, 0),
        time(17, 30, 0)
    )
})
```

Wire Protocol (JSON)

```json
// IsIn with times
{
    "type": "ASTNode",
    "filter_dict": {
        "event_time": {
            "type": "IsIn",
            "options": [
                {"type": "time", "value": "09:00:00"},
                {"type": "time", "value": "12:00:00"},
                {"type": "time", "value": "17:00:00"}
            ]
        }
    }
}

// Time range
{
    "type": "ASTNode",
    "edge_match": {
        "daily_schedule": {
            "type": "Between",
            "lower": {"type": "time", "value": "09:00:00"},
            "upper": {"type": "time", "value": "17:30:00"},
            "inclusive": true
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

10.5.18.7 4. Timezone-Aware DateTime

Python API

```python
from graphistry.compute import DateTimeValue, gt

# Using timezone-aware timestamp
filter1 = n(filter_dict={
    "timestamp": gt(
        pd.Timestamp("2023-01-01 09:00:00", tz="US/Eastern")
    )
})

# Using DateTimeValue with explicit timezone
dt_val = DateTimeValue("2023-01-01T09:00:00", "US/Eastern")
filter2 = n(filter_dict={
    "timestamp": gt(dt_val)
})
```

Wire Protocol (JSON)

```
// Timezone-aware datetime
{
    "type": "ASTNode",
    "filter_dict": {
        "timestamp": {
            "type": "GT",
            "val": {
                "type": "datetime",
                "value": "2023-01-01T09:00:00",
                "timezone": "US/Eastern"
            }
        }
    }
}
```

10.5.18.8 5. Complex Chain with Temporal Predicates

Python API

```python
from graphistry import n, e_forward
from graphistry.compute import gt, eq, between
from datetime import datetime, timedelta

# Multi-hop query with temporal filters
chain = g.gfql([
    # Recent transactions
    n(),
    e_forward(edge_match={
        "timestamp": gt(datetime.now() - timedelta(days=7)),
        "amount": gt(1000)
    }),
    # To active accounts
    n(filter_dict={
        "status": eq("active"),
        "last_login": between(
            datetime.now() - timedelta(days=30),
            datetime.now()
        )
    }),
    # Outgoing transfers
    e_forward(edge_match={
        "type": eq("transfer"),
        "timestamp": gt(datetime.now() - timedelta(days=1))
    })
])
```

Wire Protocol (JSON)

```json
{
    "type": "Chain",
    "queries": [
        {
            "type": "ASTNode",
            "edge_match": {
                "timestamp": {
                    "type": "GT",
                    "val": {
                        "type": "datetime",
                        "value": "2023-12-18T10:30:00",
                        "timezone": "UTC"
                    }
                },
                "amount": {
                    "type": "GT",
                    "val": 1000
                }
```

(continues on next page)

(continued from previous page)

```
            }
        },
        {
            "type": "ASTNode",
            "filter_dict": {
                "status": {
                    "type": "EQ",
                    "val": "active"
                },
                "last_login": {
                    "type": "Between",
                    "lower": {
                        "type": "datetime",
                        "value": "2023-11-25T10:30:00",
                        "timezone": "UTC"
                    },
                    "upper": {
                        "type": "datetime",
                        "value": "2023-12-25T10:30:00",
                        "timezone": "UTC"
                    },
                    "inclusive": true
                }
            }
        },
        {
            "type": "ASTEdge",
            "direction": "forward",
            "edge_match": {
                "type": {
                    "type": "EQ",
                    "val": "transfer"
                },
                "timestamp": {
                    "type": "GT",
                    "val": {
                        "type": "datetime",
                        "value": "2023-12-24T10:30:00",
                        "timezone": "UTC"
                    }
                }
            }
        }
    ]
}
```

10.5.18.9 6. Temporal Value Classes Direct Usage

Python API

```python
from graphistry.compute import (
    DateTimeValue, DateValue, TimeValue,
    temporal_value_from_json, gt
)

# Create temporal values
dt_val = DateTimeValue("2023-06-15T14:30:00", "Europe/London")
date_val = DateValue("2023-06-15")
time_val = TimeValue("14:30:00")

# Use in predicates
filter1 = n(filter_dict={"timestamp": gt(dt_val)})
filter2 = n(filter_dict={"event_date": eq(date_val)})
filter3 = n(filter_dict={"daily_time": eq(time_val)})

# Create from JSON
json_dt = {
    "type": "datetime",
    "value": "2023-06-15T14:30:00",
    "timezone": "Europe/London"
}
dt_from_json = temporal_value_from_json(json_dt)
```

Wire Protocol (JSON)

```
// DateTimeValue serialization
{
    "type": "datetime",
    "value": "2023-06-15T14:30:00",
    "timezone": "Europe/London"
}

// DateValue serialization
{
    "type": "date",
    "value": "2023-06-15"
}

// TimeValue serialization
{
    "type": "time",
    "value": "14:30:00"
}
```

10.5.18.10 7. Full Round-Trip Example

```
# 1. Create a complex query with temporal predicates
from graphistry import n, Chain
from graphistry.compute import gt, between, is_in
from datetime import datetime, date, time
import pandas as pd

query = Chain([
    n(filter_dict={
        "created": gt(pd.Timestamp("2023-01-01")),
        "event_date": between(date(2023, 6, 1), date(2023, 6, 30)),
        "event_time": is_in([time(9, 0), time(12, 0), time(17, 0)])
    })
])

# 2. Serialize to JSON
json_query = query.to_json()
print(json_query)

# 3. Send over wire (simulated)
import json
wire_data = json.dumps(json_query)
received_data = json.loads(wire_data)

# 4. Deserialize on receiving end
from graphistry import Chain
reconstructed_query = Chain.from_json(received_data)

# 5. Apply to graph data
result = g.gfql(reconstructed_query.chain)
```

10.5.18.11 Wire Protocol Structure

Temporal Value Types

All temporal values in the wire protocol follow this pattern:

```
// DateTime with timezone
interface DateTimeWire {
    type: "datetime";
    value: string;      // ISO 8601 format
    timezone?: string;  // IANA timezone (default: "UTC")
}

// Date only
interface DateWire {
    type: "date";
    value: string;      // YYYY-MM-DD format
}
```

(continues on next page)

(continued from previous page)

```
// Time only
interface TimeWire {
    type: "time";
    value: string;      // HH:MM:SS[.ffffff] format
}
```

Predicate Structure

Predicates containing temporal values serialize as:

```
interface TemporalPredicate {
    type: "GT" | "LT" | "GE" | "LE" | "EQ" | "NE";
    val: DateTimeWire | DateWire | TimeWire;
}

interface BetweenPredicate {
    type: "Between";
    lower: DateTimeWire | DateWire | TimeWire;
    upper: DateTimeWire | DateWire | TimeWire;
    inclusive: boolean;
}

interface IsInPredicate {
    type: "IsIn";
    options: Array<DateTimeWire | DateWire | TimeWire>;
}
```

10.5.18.12 Key Points

1. **Type Safety**: Raw strings are rejected in the Python API to avoid ambiguity
2. **Automatic Conversion**: Python datetime objects are automatically converted to appropriate temporal values
3. **Timezone Preservation**: Timezone information is preserved through serialization
4. **Tagged Format**: JSON uses tagged dictionaries to preserve type information
5. **Direct Usage**: Wire protocol dicts can be passed directly to Python predicates

10.5.18.13 Error Handling

```
# Raw strings raise ValueError
try:
    filter_raw = n(filter_dict={"date": gt("2023-01-01")})
except ValueError as e:
    print(e)
    # Output: Raw string '2023-01-01' is ambiguous. Use:
    #   - gt(pd.Timestamp('2023-01-01')) for datetime
    #   - gt({'type': 'datetime', 'value': '2023-01-01T00:00:00'}) for explicit type
```

(continues on next page)

(continued from previous page)

```
# Valid approaches
filter1 = n(filter_dict={"date": gt(pd.Timestamp("2023-01-01"))})
filter2 = n(filter_dict={"date": gt(datetime(2023, 1, 1))})
filter3 = n(filter_dict={"date": gt({"type": "datetime", "value": "2023-01-01T00:00:00",
↪"timezone": "UTC"})})
```

10.5.18.14 Performance Considerations

- Temporal predicates leverage pandas' optimized datetime operations
- Timezone conversions are handled efficiently
- For large datasets, ensure datetime columns are properly typed (not object dtype)
- Use `pd.Timestamp` for best performance when creating many predicates programmatically

10.5.19 GFQL Specifications

This section contains formal specifications for GFQL (Graph Frame Query Language), designed to support both human understanding and automated tooling, including LLM-based code synthesis.

10.5.19.1 GFQL Language Specification

Introduction

GFQL (Graph Frame Query Language) is a DataFrame-native graph query language designed for expressing graph patterns and traversals on tabular data. It operates on node and edge DataFrames, providing a functional, composable approach to graph querying with native GPU acceleration support.

Design Principles

- **Dataframe-native**: Type-safe functional bulk operations over dataframe libraries like pandas, cuDF
- **Declarative**: Focus on what to retrieve, and give the engine freedom to optimize how
- **Accessible**: Designed for both human readability and machine generation, and building on intuitions from popular tabular and graph systems
- **Performance-oriented**: Vectorized operations by default, including GPU acceleration
- **Embeddable**: Similar to DuckDB, can be embedded in different languages, and initially focused on Python data ecosystem
- **Computer-tier**: Decoupling from storage enables flexible execution - embedded locally or via remote acceleration servers

Language Forms

GFQL exists in three complementary forms:

1. **Core Language**: Abstract graph pattern matching language defined by this specification
2. **Embedded DSL**: Host language implementations (currently Python with pandas/cuDF)
3. **Wire Protocol**: JSON serialization for client-server communication (see Wire Protocol spec)

This specification focuses on the core language concepts. Examples use Python syntax for concreteness, but the patterns apply to any embedding.

Language Overview

Core Concepts

Graph Model

Graphs consist of node and edge dataframes:

- Edges: DataFrame with source and destination columns
- Nodes: DataFrame with unique identifier column
- Column names are user-defined globals for the graph:
  - Node ID attribute: `g._node` (e.g., "node_id", "id")
  - Edge source attribute: `g._source` (e.g., "source", "from")
  - Edge destination attribute: `g._destination` (e.g., "destination", "to")
- GFQL infers nodes from edge references when only edges are provided

GFQL Programs

GFQL programs are declarative graph-to-graph transformations:

- Enable use cases like search, filter, enrich, and traverse
- Express *what* to find (ex: Cypher), not *how* to find it (ex: Gremlin)

Chains

Path pattern expressions for matching graph structures:

- Express graph patterns as sequences of node and edge matching operations
- Similar to Cypher patterns but decomposed into composable steps
- Define paths through the graph: start nodes → edges → end nodes
- Each operation refines the pattern match based on previous results

WHERE (Same-Path Constraints)

WHERE ties attributes across named steps in a chain. Use it when you need to enforce relationships between nodes/edges on the same path (for example, start.owner_id equals end.owner_id). Multiple WHERE comparisons are conjunctive (AND).

Python example:

```
from graphistry import n, e_forward, col, compare

g.gfql(
    [n({"type": "account"}, name="a"), e_forward(), n({"type": "user"}, name="c")],
    where=[compare(col("a", "owner_id"), "==", col("c", "owner_id"))],
)
```

Wire format (JSON):

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"type": "account"}, "name": "a"},
    {"type": "Edge", "direction": "forward"},
    {"type": "Node", "filter_dict": {"type": "user"}, "name": "c"}
  ],
  "where": [{"eq": {"left": "a.owner_id", "right": "c.owner_id"}}]
}
```

WHERE context boundaries:

- Same-path `where=[...]` uses `compare(col(...), op, col(...))` with `op` in `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Predicate helper calls (for example, `gt(...)`, `between(...)`) are not used inside same-path `where=[...]`.
- Row-table filtering after `rows(...)` uses `where_rows(...)`:
  - `where_rows(filter_dict=...)` supports predicate helpers.
  - `where_rows(expr="...")` uses expression comparators `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`.

Operations

GFQL supports two operation families:

- Graph matchers act on graph entities (nodes and edges).
- Row-pipeline operators act on tabular outputs from matched graph entities.
- `g.gfql([...], where=[...])` filters same-path alias relationships.
- `where_rows(...)` filters the active row table in RETURN/WITH-style pipelines.

Predicates

Act on attributes of nodes and edges:

- Filter based on property values
- Comparison, membership, string matching, temporal checks
- Composable within operations to build complex conditions

Values

Type system matching modern data formats:

- Scalars: numbers, strings, booleans, null
- Temporal: ISO datetimes, dates, times with timezone support
- Collections: lists for membership tests
- Compatible with JSON, Arrow, and DataFrame type systems

Formal Grammar

Listing 1: GFQL Grammar in Extended Backus-Naur Form

```
(* Entry point *)
query ::= chain

(* Chain - path pattern expression *)
chain ::= "[" step ("," step)* "]"
step ::= operation | row_operation | call_operation

(* Graph operations *)
operation ::= node_matcher | edge_matcher

(* Node Matcher *)
node_matcher ::= "n(" node_params? ")"
node_params ::= filter_dict ("," name_param)? ("," query_param)?
              | name_param ("," query_param)?
              | query_param

(* Edge Matchers *)
edge_matcher ::= edge_forward | edge_reverse | edge_undirected
edge_forward ::= "e_forward(" edge_params? ")"
edge_reverse ::= "e_reverse(" edge_params? ")"
edge_undirected ::= ("e" | "e_undirected") "(" edge_params? ")"

(* WHERE (same-path constraints) *)
where_clause ::= "where=" where_list
where_list ::= "[" where_expr ("," where_expr)* "]"
where_expr ::= "compare(" column_ref "," compare_op "," column_ref ")"
compare_op ::= "'=='" | "'!='" | "'<'" | "'<='" | "'>'" | "'>='"
column_ref ::= alias "." column
```

(continues on next page)

(continued from previous page)

```
alias ::= identifier
column ::= identifier

(* Row operations - Cypher RETURN/WITH-style pipeline *)
row_operation ::= rows_op | where_rows_op | select_op | with_op | return_op
                | order_by_op | skip_op | limit_op | distinct_op
                | unwind_op | group_by_op
call_operation ::= "call(" string ("," params_object)? ")"
params_object ::= "{" (string ":" value_or_expr ("," string ":" value_or_expr)*)? "}"
rows_op ::= "rows(" (rows_arg ("," rows_arg)*)? ")"
rows_arg ::= "table=" ("'nodes'" | "'edges'") | "source=" string
where_rows_op ::= "where_rows(" (where_rows_arg ("," where_rows_arg)*)? ")"
where_rows_arg ::= "filter_dict=" filter_dict | "expr=" string
select_op ::= "select(" (projection_items | "items=" projection_items) ")"
with_op ::= "with_(" (projection_items | "items=" projection_items) ")"
return_op ::= "return_(" (projection_items | "items=" projection_items) ")"
projection_items ::= "[" projection_item ("," projection_item)* "]"
projection_item ::= string | "(" string "," value_or_expr ")"
value_or_expr ::= value | string
order_by_op ::= "order_by(" (order_keys | "keys=" order_keys) ")"
order_keys ::= "[" "(" value_or_expr "," ("'asc'" | "'desc'") ")" ("," "(" value_or_expr
↪"," ("'asc'" | "'desc'") ")")* "]"
skip_op ::= "skip(" (integer | "value=" integer) ")"
limit_op ::= "limit(" (integer | "value=" integer) ")"
distinct_op ::= "distinct()"
unwind_op ::= "unwind(" "expr=" value_or_expr ("," "as_=" string)? ")"
group_by_op ::= "group_by(" "keys=" "[" string ("," string)* "]" "," "aggregations=" "["␣
↪aggregation_spec ("," aggregation_spec)* "]" ")"
aggregation_spec ::= "(" string "," string ")" | "(" string "," string "," value_or_expr
↪")"

(* Parameters *)
edge_params ::= edge_match_params ("," hop_params)? ("," node_filter_params)? ("," name_
↪param)?

filter_dict ::= "{" (property_filter ("," property_filter)*)? "}"
property_filter ::= string ":" (value | predicate)

hop_params ::= hop_bound_params | hop_slice_params | hop_label_params | "hops=" integer␣
↪| "to_fixed_point=True"
hop_bound_params ::= "min_hops=" integer | "max_hops=" integer
hop_slice_params ::= "output_min_hops=" integer | "output_max_hops=" integer
hop_label_params ::= "label_node_hops=" string | "label_edge_hops=" string | "label_
↪seeds=True"
node_filter_params ::= source_filter ("," dest_filter)?
source_filter ::= "source_node_match=" filter_dict | "source_node_query=" string
dest_filter ::= "destination_node_match=" filter_dict | "destination_node_query=" string

name_param ::= "name=" string
query_param ::= "query=" string
edge_query_param ::= "edge_query=" string
edge_match_params ::= filter_dict | edge_query_param
```

(continues on next page)

(continued from previous page)

```
(* Predicates *)
predicate ::= comparison | membership | range | null_check | string_pred | temporal_pred

comparison ::= ("gt" | "lt" | "ge" | "le" | "eq" | "ne") "(" value ")"
membership ::= "is_in(" "[" value ("," value)* "]" ")"
range ::= "between(" value "," value ("," "inclusive=" boolean)? ")"
null_check ::= "isnull()" | "notnull()" | "isna()" | "notna()"
string_pred ::= string_match | string_check
string_match ::= "contains(" string ("," "case=" boolean)? ("," "regex=" boolean)? ")"
              | "match(" string ("," "case=" boolean)? ("," "flags=" integer)? ")"
              | "fullmatch(" string ("," "case=" boolean)? ("," "flags=" integer)? ")"
              | ("startswith" | "endswith") "(" string ("," "case=" boolean)? ")"
string_check ::= ("isalpha" | "isnumeric" | "isdigit" | "isalnum"
               | "isupper" | "islower") "()"
temporal_pred ::= temporal_check "()"
temporal_check ::= "is_month_start" | "is_month_end" | "is_quarter_start"
                 | "is_quarter_end" | "is_year_start" | "is_year_end" | "is_leap_year"

(* Values *)
value ::= scalar | temporal_value | collection
scalar ::= number | string | boolean | null
temporal_value ::= datetime_value | date_value | time_value
datetime_value ::= "pd.Timestamp(" string ("," "tz=" string)? ")"
                 | "datetime(" datetime_args ")"
date_value ::= "date(" date_args ")"
time_value ::= "time(" time_args ")"
collection ::= "[" (value ("," value)*)? "]"

(* Primitives *)
string ::= '"' [^"]* '"' | "'" [^']* "'"
number ::= integer | float
integer ::= ["-"]? [0-9]+
float ::= ["-"]? [0-9]+ "." [0-9]+
boolean ::= "True" | "False"
null ::= "None"
identifier ::= [A-Za-z_][A-Za-z0-9_]*
datetime_args ::= integer ("," integer)*
date_args ::= integer "," integer "," integer
time_args ::= integer "," integer ("," integer)?
```

Operations

Node Matcher: `n()`

Filters nodes based on attributes.

Syntax: `n(filter_dict?, name?, query?)`

Parameters:

- `filter_dict`: Dictionary of attribute filters

- `name`: Optional string label for results
- `query`: Pandas query string expression

Examples:

```
n()                                    # All nodes
n({"type": "person"})                  # Nodes where type='person'
n({"age": gt(30)})                     # Nodes where age > 30
n(name="important")                    # Label matching nodes
n(query="age > 30 and status == 'active'")  # Query string
```

Edge Matchers

Forward Traversal: `e_forward()`

Traverses edges in forward direction (source → destination).

Syntax: `e_forward(edge_match?, hops?, min_hops?, max_hops?, output_min_hops?, output_max_hops?, label_node_hops?, label_edge_hops?, label_seeds?, to_fixed_point?, source_node_match?, destination_node_match?, name?)`

Parameters:

- `edge_match`: Edge attribute filters
- `hops`: Number of hops (default: 1; shorthand for `max_hops`)
- `min_hops`/`max_hops`: Inclusive traversal bounds (default min=1 unless max=0; max defaults to hops)
- `output_min_hops`/`output_max_hops`: Optional post-filter slice; defaults keep all traversed hops up to `max_hops`
- `label_node_hops`/`label_edge_hops`: Optional hop-number columns; `label_seeds=True` writes hop 0 for seeds when labeling
- `to_fixed_point`: Continue until no new nodes (default: False)
- `source_node_match`: Filters for source nodes
- `destination_node_match`: Filters for destination nodes
- `name`: Optional label

Examples:

```
e_forward()                            # One hop forward
e_forward(hops=2)                      # Two hops forward
e_forward(min_hops=2, max_hops=4, output_min_hops=3, label_edge_hops="edge_hop")  #␣
↪bounded + sliced + labeled
e_forward(to_fixed_point=True)         # All reachable nodes
e_forward({"type": "follows"})         # Only 'follows' edges
e_forward(source_node_match={"active": True})  # From active nodes
```

Reverse Traversal: `e_reverse()`

Traverses edges in reverse direction (destination → source).

Syntax: Same as `e_forward()`

Undirected Traversal: `e()` or `e_undirected()`

Traverses edges in both directions.

Syntax: Same as `e_forward()`

Row-Pipeline Operations

These operations are encoded as call steps in the chain and are used for Cypher-style `MATCH ... RETURN` processing:

- `rows(table=..., source=...)`: select active row table (nodes/edges; optional alias scope)
- `where_rows(filter_dict=..., expr=...)`: row-level filtering on active row table
- `select(...)` / `with_(...)` / `return_(...)`: projection and expression shaping
- `order_by(...)`, `skip(...)`, `limit(...)`, `distinct()`: row sorting/paging/dedup
- `unwind(...)`: expand list-valued expressions into rows
- `group_by(...)`: grouped vectorized aggregations

Row-pipeline operators are part of the chain list itself (not top-level `g.gfql()` keyword arguments):

```
from graphistry import n, e_forward
from graphistry.compute import rows, where_rows, return_, order_by, limit

g.gfql([
    n({"type": "Person"}, name="p"),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person"}, name="q"),
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_([("id", "id"), ("name", "name"), ("score", "score")]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(25),
])
```

Equivalent explicit `Chain` form:

```
from graphistry.compute.chain import Chain

query = Chain([
    n({"type": "Person"}, name="p"),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person"}, name="q"),
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
```

(continues on next page)

(continued from previous page)

```
])
g.gfql(query)
```

`where=[...]` and `where_rows(...)` are intentionally different:

- `where=[...]` compares values across named path aliases in the MATCH pattern.
- `where_rows(...)` evaluates scalar expressions against the active row table.

Predicates

Comparison Predicates

```
gt(value)    # Greater than
lt(value)    # Less than
ge(value)    # Greater than or equal
le(value)    # Less than or equal
eq(value)    # Equal
ne(value)    # Not equal
```

Membership Predicate

```
is_in([value1, value2, ...])  # Value in list
```

Range Predicate

```
between(lower, upper, inclusive=True)  # Value in range
```

String Predicates

Pattern matching predicates:

```
contains(pat, case=True, regex=True)     # Contains pattern (substring or regex)
startswith(prefix, case=True)            # Starts with prefix
endswith(suffix, case=True)              # Ends with suffix
match(pat, case=True, flags=0)           # Matches regex from start of string
fullmatch(pat, case=True, flags=0)       # Matches regex against entire string
```

String type checking predicates:

```
isalpha()    # Alphabetic characters only
isnumeric()  # Numeric characters only
isdigit()    # Digits only
isalnum()    # Alphanumeric
isupper()    # All uppercase
islower()    # All lowercase
```

Null Predicates

```
isnull()     # Is null/None
notnull()    # Is not null/None
isna()       # Is NaN (numeric)
notna()      # Is not NaN
```

Temporal Predicates

```
is_month_start()    # First day of month
is_month_end()      # Last day of month
is_quarter_start()  # First day of quarter
is_quarter_end()    # Last day of quarter
is_year_start()     # First day of year
is_year_end()       # Last day of year
is_leap_year()      # Is leap year
```

Call Operations and Security

Call Operations

GFQL supports calling Plottable methods through the `call()` operation, providing controlled access to graph transformation and analysis capabilities:

```
call(function: str, params: dict) -> ASTCall
```

Call operations enable:

- Graph algorithms (PageRank, community detection)
- Layout computations (ForceAtlas2, Graphviz)
- Data transformations (filtering, collapsing)
- Visual encodings (color, size, icons)
- Row-pipeline operations (`rows`, `where_rows`, `select`, `with_`, `return_`, `order_by`, `skip`, `limit`, `distinct`, `unwind`, `group_by`)

Safelist Architecture

For security and stability, Call operations are restricted to a predefined safelist of methods. This prevents:

- Arbitrary code execution
- Access to filesystem or network operations
- Modification of global state
- Unsafe graph operations

Safelist Categories

Graph Analysis

- `get_degrees`, `get_indegrees`, `get_outdegrees`: Calculate node degrees
- `compute_cugraph`: Run GPU algorithms (pagerank, louvain, etc.)
- `compute_igraph`: Run CPU algorithms
- `get_topological_levels`: Analyze DAG structure

Filtering & Transformation

- `filter_nodes_by_dict`, `filter_edges_by_dict`: Filter by attributes
- `hop`: Traverse graph with conditions
- `drop_nodes`, `keep_nodes`: Node selection
- `collapse`: Merge nodes by attribute
- `prune_self_edges`: Remove self-loops
- `materialize_nodes`: Generate node table

Layout

- `layout_cugraph`: GPU-accelerated layouts
- `layout_igraph`: CPU-based layouts
- `layout_graphviz`: Graphviz layouts
- `fa2_layout`: ForceAtlas2 layout
- `ring_continuous_layout`: Radial layout driven by numeric attributes
- `ring_categorical_layout`: Radial layout grouping by categories
- `time_ring_layout`: Time-series radial layout (accepts ISO timestamp bounds)
- `group_in_a_box_layout`: Group-in-a-box community layout
- `circle_layout`: Circular node layout
- `tree_layout`: Sugiyama-style tree layout
- `mercator_layout`: Mercator projection for latitude/longitude node coordinates
- `modularity_weighted_layout`: Community-weighted edge layout preparation

> **Note**
>
> `time_ring_layout` accepts ISO-8601 strings for `time_start` / `time_end` when sent over the wire. GFQL converts them to `numpy.datetime64` before use so the behavior matches direct Plotter calls.

Visual Encoding

- `encode_point_color`: Color nodes/edges
- `encode_point_size`: Size nodes
- `encode_point_icon`: Set icons
- `bind`: Attach visual attributes

Embeddings & Dimensionality Reduction

- `umap`: UMAP dimensionality reduction for graph embeddings

Validation

Call operations undergo multiple validation stages:

1. **Safelist Check**: Function name must be in the safelist
2. **Parameter Validation**: Parameters validated against method signature
3. **Type Checking**: Runtime type validation
4. **Schema Validation**: Compatibility with graph schema

Error Codes

- **E104**: Function not in safelist
- **E105**: Missing required parameter
- **E201**: Parameter type mismatch
- **E303**: Unknown parameter
- **E301**: Required column not found (runtime)

Type System

Value Types

1. **Scalars**
   - `number`: int, float
   - `string`: Text values
   - `boolean`: True/False
   - `null`: None
2. **Temporal Types**
   - `datetime`: Timestamp with optional timezone
   - `date`: Calendar date
   - `time`: Time of day
3. **Collections**
   - `list`: Ordered sequence of values

Type Coercion

GFQL performs automatic type coercion:

- Python datetime → pandas Timestamp
- Numeric types → appropriate precision
- Collections → lists for `is_in()`

Execution Model

Declarative Pattern Matching

GFQL follows a declarative execution model similar to Neo4j's Cypher:

1. **Pattern Declaration**: Chains express path patterns in the graph
   - Users declare graph patterns as sequences of node and edge constraints
   - Patterns specify *what* paths to match, not *how* to find them
   - The engine optimizes pattern matching based on data characteristics
2. **Row-Pipeline Transformation**: Optional call steps shape tabular outputs
   - `rows(...)` chooses active table (`nodes` or `edges`, optionally alias-scoped)
   - `where_rows(...)`, projections, sorting, grouping, and paging transform rows
   - Expressions are validated before execution and unsupported forms fail fast
3. **Set-Based Operations**: Graph and row operations run in bulk
   - No explicit user-managed iteration or traversal order
   - Results include all matching paths/rows satisfying constraints
   - Execution is vectorized in supported engines (pandas/cuDF)
4. **Lazy Evaluation**: Chains define transformations without immediate execution
   - Allows engines to optimize path finding and row-table transformations

Result Access

Query execution returns graph and/or row-tabular outputs according to the embedding implementation.

```
result = g.gfql([...])
# accessors are embedding-specific
```

For Python accessor details (including row-pipeline result materialization), see *GFQL Python Embedding*.

Named Results

Operations with `name` parameter add boolean columns to mark matched entities:

```
result = g.gfql([
    n({"type": "person"}, name="people"),
    e_forward(name="connections"),
    n({"active": True}, name="active_targets")
])

# Access all matched nodes and edges:
all_nodes = result._nodes
all_edges = result._edges

# Access specific matched nodes/edges using pandas filtering:
people_nodes = result._nodes[result._nodes["people"]]
connection_edges = result._edges[result._edges["connections"]]
active_nodes = result._nodes[result._nodes["active_targets"]]

# Or using standard pandas query syntax:
people_nodes = result._nodes.query("people == True")
```

This pattern is essential for extracting specific subsets from complex graph traversals.

Best Practices

1. **Use specific filters early**: Filter nodes before traversing edges
2. **Limit hops**: Use reasonable hop limits to avoid explosion
3. **Name important results**: Use `name` parameter for analysis
4. **Prefer filter_dict**: More efficient than query strings
5. **Use appropriate predicates**: Match predicate to column type

See Also

- *GFQL Python Embedding* - Python implementation details
- *GFQL Wire Protocol Specification* - JSON serialization format
- *Cypher to GFQL Python & Wire Protocol Mapping* - Cypher to GFQL translation with wire protocol
- *GFQL Quick Reference* - Comprehensive examples and usage patterns
- *GFQL Validation Guide* - Learn validation basics

10.5.19.2 GFQL Python Embedding

This document describes the Python-specific implementation of GFQL using pandas and cuDF dataframes.

Graph Construction

In Python, graphs are created with user-defined column names:

```
import graphistry
assert 'src_col' in df.columns and 'dst_col' in df.columns
g = graphistry.edges(df, source='src_col', destination='dst_col')

# Optional; GFQL infers node existence when only edges are provided
assert 'node_col' in df.columns
g2 = graphistry.nodes(df, node='node_col')
```

Schema Access

The graph schema is accessible via attributes:

- `g._node`: Node ID column name
- `g._source`: Edge source column name
- `g._destination`: Edge destination column name

Graph nodes can be generically accessed using these attributes:

- `g._nodes`: Node DataFrame
- `g._nodes[g._node]`: Node ID column
- `g._nodes[[attr for attr in g._nodes.columns if attr != g._node]]`: All other node attributes

Graph edges can be accessed similarly:

- `g._edges`: Edge DataFrame
- `g._edges[g._source]`: Edge source column
- `g._edges[g._destination]`: Edge destination column
- `g._edges[[attr for attr in g._edges.columns if attr not in [g._source, g._destination]]]`: All other edge attributes

Query Execution

```
from graphistry import n, e_forward

# Execute a chain
result = g.gfql([
    n({"type": "person"}),
    e_forward(),
    n()
```

(continues on next page)

(continued from previous page)

```
])

# Access results
nodes_df = result._nodes  # Filtered nodes DataFrame
edges_df = result._edges  # Filtered edges DataFrame
```

Native GFQL chains are Python-embedded values, not source strings. Pass actual Python `list`, `dict` envelope, or `Chain` objects to `g.gfql(...)`; a `str` query is treated as Cypher text and is not decoded as a Python literal or JSON chain. Tooling that receives serialized native chains should decode them before calling `g.gfql(...)`, or intentionally emit Cypher query text.

Row-Pipeline Query Execution (`MATCH ... RETURN` style)

```
from graphistry import n, e_forward
from graphistry.compute import rows, where_rows, return_, order_by, limit

result = g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person"}, name="q"),
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(25),
])
```

Row-pipeline results use the active row table as `result._nodes`. `result._edges` is an empty placeholder frame in row mode.

Same-Path Constraints (WHERE)

```
from graphistry import n, e_forward, col, compare

result = g.gfql(
    [
        n({"type": "account"}, name="a"),
        e_forward(),
        n({"type": "user"}, name="c"),
    ],
    where=[compare(col("a", "owner_id"), "==", col("c", "owner_id"))],
)
```

Multiple WHERE comparisons are ANDed.

Common WHERE Validation Errors

WHERE is validated before same-path execution starts, so invalid references fail early with clean errors.

```
from graphistry import n, e_forward, col, compare

# Missing alias binding in WHERE
g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="c")],
    where=[compare(col("missing", "x"), "==", col("c", "owner_id"))],
)
# ValueError: WHERE references aliases with no node/edge bindings: missing

# Missing column on a bound alias
g.gfql(
    [n(name="a"), e_forward(name="e"), n(name="c")],
    where=[compare(col("a", "missing_col"), "==", col("c", "owner_id"))],
)
# ValueError: WHERE references missing column 'missing_col' on alias 'a' ...

# Invalid WHERE entry class
g.gfql([n(name="a"), e_forward(name="e"), n(name="c")], where=[123])
# ValueError: where[0] must be a WhereComparison or dict clause ...
```

Advanced troubleshooting (migration/debugging): you can set `GRAPHISTRY_WHERE_VALIDATION_IGNORE_ERRORS` and `GRAPHISTRY_WHERE_VALIDATION_IGNORE_CALLS` to suppress specific missing-column validation branches when needed.

Common Row-Pipeline Validation Errors

`where_rows(...)` and related row operations fail fast when expressions or payloads are unsupported:

Exception class depends on validation phase:

- expression/shape checks usually raise `GFQLTypeError`
- schema/column checks usually raise `GFQLSchemaError`

```
from graphistry.compute import rows, where_rows, return_

# Missing column in expression
g.gfql([rows(), where_rows(expr="missing_col > 1"), return_(["id"])])
# -> Validation error for missing required column on active row table

# Unsupported function in expression subset
g.gfql([rows(), where_rows(expr="reverse(name) = 'x'"), return_(["id"])])
# -> GFQLTypeError (unsupported row expression/function)

# Invalid rows table selector
g.gfql([rows(table="invalid_table")])
# -> Validation error (table must be 'nodes' or 'edges')
```

Engine Selection

GFQL supports multiple execution engines:

- **pandas**: CPU execution (default)
- **cudf**: GPU acceleration
- **auto**: Automatic selection based on data type

```
# Force specific engine
g.gfql([...], engine='cudf')  # GPU execution
g.gfql([...], engine='pandas')  # CPU execution
g.gfql([...], engine='auto')  # Auto-select
```

Python-Specific Values

Temporal Values

```
import pandas as pd

# Timestamps
pd.Timestamp('2023-01-01')
pd.Timestamp.now()

# Time deltas
pd.Timedelta(days=30)
pd.Timedelta(hours=24)
```

DataFrame Operations

Results can be further processed using standard pandas operations:

```
# Using boolean columns from named operations
people_nodes = result._nodes[result._nodes["people"]]

# Using pandas query
active_nodes = result._nodes.query("active == True")

# Standard pandas operations
result._nodes.groupby('type').size()
```

Validation

GFQL provides comprehensive validation to catch errors early:

Syntax Validation

Chains validate on construction by default. Nodes, edges, predicates, refs, calls, and remote graphs are validated when a parent `Chain`/`Let` validates them or when you call `.validate()` directly. Schema validation is a separate, data-aware pass.

```
from graphistry.compute.chain import Chain
from graphistry.compute.ast import n, e_forward

# Automatic validation on construction
chain = Chain([
    n({'type': 'person'}),
    e_forward(hops=-1)  # Raises GFQLTypeError: hops must be positive
])
```

For advanced flows (large/nested ASTs or staged assembly), you can defer structural validation and run it once after assembly:

```
# Defer validation while building
chain = Chain([
    n({'type': 'person'}),
    e_forward(hops=-1)
], validate=False)  # No validation yet

# Later, validate once (or let g.gfql validate it)
chain.validate()  # Raises GFQLTypeError: hops must be positive
```

Use deferred validation to avoid re-validating nested `Chain`/`Let` wrappers during assembly; keep the defaults for typical workflows so mistakes surface immediately.

Validation Phases

- **Constructor defaults:** `Chain([...])` and `Let(...)` validate immediately; pass `validate=False` to defer.
- **Parent-driven checks:** AST operations (`Node`, `Edge`, predicates, `Ref`, `Call`, `RemoteGraph`) validate when their parent validates, or via explicit `.validate()`.
- **JSON defaults:** `to_json` / `from_json` default to `validate=True`, which runs structural validation during serialization/deserialization.
- **Schema validation:** Use `validate_chain_schema(g, chain)` or `g.gfql(..., validate_schema=True)` to verify column/type compatibility before execution.

Schema Validation

You have two options for validating queries against your data schema:

1. **Validate-only** (no execution): Use `validate_chain_schema()` to check compatibility without running the query
2. **Runtime validation** (automatic): `g.gfql(...)` validates columns during execution and raises `GFQLSchemaError` for missing or mismatched columns

```
# Method 1: Validate-only (no execution)
from graphistry import Chain
from graphistry.compute.exceptions import GFQLSchemaError
from graphistry.compute.validate_schema import validate_chain_schema

chain = Chain([n({'missing_column': 'value'})])
try:
    validate_chain_schema(g, chain)  # Only validates, doesn't execute
    print("Chain is valid for this graph")
except GFQLSchemaError as e:
    print(f"Schema incompatibility: {e}")
    print("No query was executed")

# Method 2: Validate-and-run (automatic - gfql() validates before executing)
try:
    result = g.gfql([
        n({'missing_column': 'value'})
    ])  # Schema is validated automatically; raises GFQLSchemaError before execution
except GFQLSchemaError as e:
    print(f"Pre-execution validation failed: {e}")
    print("Query was not executed")
```

> **Note:** `g.gfql()` always validates the schema before executing. There is no need for a separate `validate_schema=True` flag — validation is built in. Use Method 1 (`validate_chain_schema`) when you want to check validity without executing at all.

Error Types

GFQL uses structured exceptions with error codes:

- **GFQLSyntaxError** (E1xx): Structural issues
  - E101: Invalid type (e.g., chain not a list)
  - E103: Invalid parameter value (e.g., negative hops)
  - E104: Invalid direction
  - E105: Missing required field
- **GFQLTypeError** (E2xx): Type mismatches
  - E201: Wrong value type (e.g., string instead of dict)
  - E202: Predicate type mismatch
  - E204: Invalid name type
- **GFQLSchemaError** (E3xx): Data-related issues

- E301: Column not found
- E302: Incompatible column type (e.g., numeric predicate on string column)

Validation Modes

```
# Fail-fast mode (default) - raises on first error
chain.validate()

# Collect-all mode - returns list of all errors
errors = chain.validate(collect_all=True)
for error in errors:
    print(f"[{error.code}] {error.message}")
    if error.suggestion:
        print(f"  Suggestion: {error.suggestion}")

# Pre-validate schema without execution
from graphistry.compute.validate_schema import validate_chain_schema

# Check schema compatibility
errors = validate_chain_schema(g, chain, collect_all=True)
```

Example: Handling Validation Errors

```
from graphistry.compute.exceptions import GFQLValidationError, GFQLSchemaError

try:
    result = g.gfql([
        n({'age': 'twenty-five'})  # Type mismatch
    ])
except GFQLSchemaError as e:
    print(f"Schema error [{e.code}]: {e.message}")
    print(f"Field: {e.context.get('field')}")
    print(f"Suggestion: {e.context.get('suggestion')}")
    # Output:
    # Schema error [E302]: Type mismatch: column "age" is numeric but filter value is␣
↪string
    # Field: age
    # Suggestion: Use a numeric value like age=25
```

Common Errors and Validation

Type Mismatches

```
# Wrong - String predicate on numeric column
n({"age": contains("3")})

# Correct - Use numeric predicate
```

(continues on next page)

(continued from previous page)

```
n({"age": gt(30)})

# Wrong - String comparison on datetime
n({"created": gt("2024-01-01")})

# Correct - Use proper datetime type
n({"created": gt(pd.Timestamp("2024-01-01"))})
```

Schema Validation

```
# Check available columns before querying
print(g._nodes.columns)  # ['id', 'type', 'name']

# Wrong - Column doesn't exist
g.gfql([n({"username": "Alice"})])  # KeyError

# Correct - Use existing column
g.gfql([n({"name": "Alice"})])
```

Unsupported Operations

```
# Supported in row pipeline - grouped aggregation
from graphistry.compute import rows, group_by
g.gfql([
    rows(),
    group_by(keys=["type"], aggregations=[("cnt", "count")]),
])

# Pure GFQL list/Chain syntax still has no direct OPTIONAL MATCH operator.
# For the bounded Cypher surface through g.gfql(), execute a Cypher string instead:
g.gfql(
    "MATCH (n:Person) "
    "OPTIONAL MATCH (n)-[r:KNOWS]->(m) "
    "RETURN n.name AS name, type(r) AS rel_type"
)

# Or handle optionality explicitly in post-processing:
result = g.gfql([n(), e_forward()])
# Check for nodes without edges
nodes_with_edges = result._nodes[result._nodes[g._node].isin(result._edges[g._source])]

# Wrong - Arbitrary row function outside supported expression subset
# g.gfql([rows(), where_rows(expr="custom_fn(score)")])
# Correct - Use supported row-expression operators, or post-process DataFrame
```

Cypher String Execution Through `g.gfql()`

For supported Cypher strings on a bound graph, `g.gfql()` defaults string queries to `language="cypher"`.

`g.gfql("MATCH ...")` still returns a `Plottable`, but current Cypher `RETURN` output is usually consumed as rows from `result._nodes`:

- scalar/property projections such as `RETURN p.name AS name` produce a table in `result._nodes`
- whole-entity projections such as `RETURN p` also surface entity-valued rows in `result._nodes`
- `result._edges` is typically an empty placeholder frame for these row-shaped Cypher results

If you want a traversable graph/subgraph back in both `_nodes` and `_edges`, use native GFQL chain syntax or the `GRAPH { }` constructor (a GFQL extension to Cypher that keeps results in graph state instead of flattening to rows).

```
from graphistry import n, e_forward

# Cypher syntax through g.gfql() returns a Plottable, with row output exposed in _nodes.
result = g.gfql("MATCH (p:Person) RETURN p.name AS name")
df = result._nodes

entity_rows = g.gfql("MATCH (p:Person) RETURN p")
entity_df = entity_rows._nodes

# If you want a graph/subgraph back, use native GFQL chain syntax...
g2 = g.gfql([n({"type": "Person"}), e_forward(), n()])

# ...or the GRAPH { } constructor (GFQL extension).
g3 = g.gfql(
    "GRAPH { "
    "  MATCH (p:Person)-[r]->(q) "
    "  WHERE p.score >= 10 "
    "}"
)

limited = g.gfql(
    "MATCH (p:Person) RETURN p.name AS name ORDER BY name DESC LIMIT $top_n",
    params={"top_n": 10},
)

same_limited = g.gfql("MATCH (p:Person) RETURN p.name AS name", language="cypher")
```

Use `params=...` instead of manual string interpolation, and expect unsupported but syntactically valid query shapes on this Cypher surface to raise `GFQLValidationError`.

Use the compiler helpers when you need parse/compile/translation output instead of immediate execution:

```
from graphistry.compute.gfql.cypher import (
    parse_cypher,
    compile_cypher,
    cypher_to_gfql,
    gfql_from_cypher,
)
```

See the Cypher-in-GFQL guide for the execution-first path and entrypoint selection: *Cypher Syntax In GFQL*.

Best Practices

Query Construction

```
# Good: Build queries programmatically
node_filters = {"type": "User"}
if min_age:
    node_filters["age"] = gt(min_age)
g.gfql([n(node_filters)])

# Avoid: Hardcoded query strings
g.gfql([n(query=f"type == 'User' and age > {min_age}")])  # SQL injection risk
```

Memory Efficiency

```
# Good: Filter early and use named results
result = g.gfql([
    n({"active": True}, name="active_users"),  # Filter first
    e_forward({"recent": True})
])
# Only access what you need
active_users = result._nodes[result._nodes["active_users"]]

# Avoid: Loading everything then filtering
all_nodes = g._nodes
active = all_nodes[all_nodes["active"] == True]  # Loads entire graph
```

GPU Best Practices

```
# Check GPU memory before large operations
if engine == 'cudf':
    import cudf
    print(f"GPU memory: {cudf.cuda.cuda.get_memory_info()}")

# Convert results back to pandas if needed for compatibility
result_pandas = result._nodes.to_pandas() if hasattr(result._nodes, 'to_pandas') else␣
↪result._nodes
```

DAG Patterns with Let Bindings

GFQL supports directed acyclic graph (DAG) patterns using Let bindings, which allow you to define named graph operations that can reference each other.

Let Bindings

```
from graphistry import let, ref, n, e_forward, ge

# Define DAG patterns with named bindings
result = g.gfql(let({
    'persons': n({'type': 'person'}),
    'adults': ref('persons', [n({'age': ge(18)})]),
    'connections': [
        n({'type': 'person', 'age': ge(18)}),
        e_forward({'type': 'knows'}),
        n({'type': 'person', 'age': ge(18)})
    ]
}))

# Access individual binding results
persons_df = result._nodes[result._nodes['persons']]
adults_df = result._nodes[result._nodes['adults']]
connection_edges = result._edges[result._edges['connections']]
```

Ref (Reference to Named Bindings)

The `ref()` function creates references to named bindings within a Let. Ref chains run on the referenced graph; bindings created by `n()` contain nodes only, so edge traversals need a binding that preserves edges (for example, via a list or `Chain([...])`).

```
# Basic reference - just the binding result
result = g.gfql(let({
    'base': n({'status': 'active'}),
    'extended': ref('base', [n()])  # Just references 'base'
}))

# Reference with additional operations (node-only refinements)
result = g.gfql(let({
    'suspects': n({'risk_score': gt(80)}),
    'verified': ref('suspects', [
        n({'verified': True})
    ])
}))

# For traversals, inline the seed filter into a list or Chain binding
result = g.gfql(let({
    'lateral_movement': [
        n({'risk_score': gt(80)}),
        e_forward({'type': 'ssh', 'failed_attempts': gt(5)}),
        n({'type': 'server'})
    ]
}))
```

Complex DAG Patterns

```
# Multi-level analysis pattern
result = g.gfql(let({
    # Find high-value accounts
    'high_value': n({'balance': gt(100000)}),

    # Find transactions from high-value accounts
    'large_transfers': [
        n({'balance': gt(100000)}),
        e_forward({'type': 'transfer', 'amount': gt(10000)}),
        n()
    ],

    # Find suspicious patterns
    'suspicious': ref('large_transfers', [
        n({'created_recent': True, 'verified': False})
    ])
}))
```

Remote Graph References

For distributed computing, `remote()` allows referencing graphs on remote servers:

```
from graphistry.compute import remote

# Reference a remote dataset
result = g.gfql([
    remote(dataset_id='fraud-network-2024'),
    n({'risk_score': gt(90)}),
    e_forward()
])
```

Call Operations with Let Bindings

Call operations can be used within Let bindings for complex workflows:

```
result = g.gfql(let({
    # Initial filtering with edges preserved for graph algorithms
    'suspects': Chain([
        n({'flagged': True}),
        e_undirected(),
        n({'flagged': True})
    ]),

    # Compute PageRank on subgraph
    'ranked': ref('suspects', [
        call('compute_cugraph', {'alg': 'pagerank'})
    ]),
```

(continues on next page)

(continued from previous page)

```
    # Find high PageRank nodes
    'influencers': ref('ranked', [
        n({'pagerank': gt(0.01)})
    ])
}))
```

See Also

- *GFQL Language Specification* - Core language specification
- *GFQL Quick Reference* - Python API examples

10.5.19.3 GFQL Wire Protocol Specification

Introduction

The GFQL Wire Protocol defines the JSON serialization format for GFQL queries, enabling:

- Client-server communication
- Query persistence and storage
- Cross-language interoperability between Python, JavaScript, and other clients
- Configuration-driven query generation

Design Principles

- **Type Safety**: Tagged dictionaries preserve type information
- **Self-Describing**: Each object includes type metadata
- **Extensible**: Schema supports future additions
- **Round-Trip Safe**: Lossless serialization/deserialization

Protocol Overview

Message Structure

All GFQL wire protocol messages are JSON objects with a `type` field:

```
{
  "type": "MessageType",
  "payload": {}
}
```

Supported Message Types

- `Chain`: Complete query chain
- `Let`: DAG pattern with named bindings
- `Ref`: Reference to Let binding with optional chain
- `RemoteGraph`: Reference to remote dataset
- `Call`: Algorithm/transformation invocation
- `Node`: Node matcher operation
- `Edge`: Edge traversal operation
- Predicates: `GT`, `LT`, `EQ`, `IsIn`, `Between`, etc.
- Temporal values: `datetime`, `date`, `time`

Message Structure

All GFQL wire protocol messages are JSON objects with a `type` field that identifies the message type. The protocol uses discriminated unions for polymorphic types.

Type Identification

Each object includes a `type` field:

- Operations: `"Node"`, `"Edge"`, `"Chain"`, `"Let"`, `"Ref"`, `"RemoteGraph"`, `"Call"`
- Predicates: `"GT"`, `"LT"`, `"IsIn"`, etc.
- Temporal values: `"datetime"`, `"date"`, `"time"`

This enables unambiguous deserialization and validation.

Operation Serialization

Node Operation

Python:

```
n({"type": "person", "age": gt(30)}, name="adults")
```

Wire Format:

```
{
  "type": "Node",
  "filter_dict": {
    "type": "person",
    "age": {
      "type": "GT",
      "val": 30
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "name": "adults"
}
```

Edge Operation

Python:

```
e_forward(
    {"type": "transaction"},
    min_hops=2,
    max_hops=4,
    output_min_hops=3,
    label_edge_hops="edge_hop",
    source_node_match={"active": True},
    name="txns"
)
```

Wire Format:

```
{
  "type": "Edge",
  "direction": "forward",
  "edge_match": { "type": "transaction" },
  "min_hops": 2,
  "max_hops": 4,
  "output_min_hops": 3,
  "label_edge_hops": "edge_hop",
  "source_node_match": { "active": true },
  "name": "txns"
}
```

Optional fields:

- `hops` (shorthand for `max_hops`)
- `output_min_hops`
- `output_max_hops`
- `label_node_hops`, `label_edge_hops`, `label_seeds`
- `to_fixed_point`

Chain

Python:

```
from graphistry import n, e_forward

g.gfql([
    n({"id": "Alice"}),
    e_forward({"type": "friend"}),
```

(continues on next page)

(continued from previous page)

```
    n({"status": "active"})
])
```

Wire Format:

```
{
  "type": "Chain",
  "chain": [
    {
      "type": "Node",
      "filter_dict": {"id": "Alice"}
    },
    {
      "type": "Edge",
      "direction": "forward",
      "edge_match": {"type": "friend"}
    },
    {
      "type": "Node",
      "filter_dict": {"status": "active"}
    }
  ]
}
```

Optional fields:

- `where`: list of same-path comparisons using `eq`, `neq`, `lt`, `le`, `gt`, `ge` with `left`/`right` as `alias.column` strings. Multiple entries are ANDed. Operator mapping:
  - `eq` maps to `==`
  - `neq` maps to `!=`
  - `lt` maps to `<`
  - `le` maps to `<=`
  - `gt` maps to `>`
  - `ge` maps to `>=`

Chain with WHERE (wire format):

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"type": "account"}, "name": "a"},
    {"type": "Edge", "direction": "forward"},
    {"type": "Node", "filter_dict": {"type": "user"}, "name": "c"}
  ],
  "where": [{"eq": {"left": "a.owner_id", "right": "c.owner_id"}}]
}
```

WHERE Validation Errors

The parser and same-path validator reject malformed or unresolved WHERE clauses before execution.

Unsupported operator key:

```
{
  "type": "Chain",
  "chain": [{"type": "Node", "name": "a"}, {"type": "Node", "name": "c"}],
  "where": [{"lte": {"left": "a.owner_id", "right": "c.owner_id"}}]
}
```

Expected error: `Unsupported WHERE operator 'lte'`.

Missing required keys:

```
{
  "type": "Chain",
  "chain": [{"type": "Node", "name": "a"}, {"type": "Node", "name": "c"}],
  "where": [{"eq": {"left": "a.owner_id"}}]
}
```

Expected error: `WHERE clause must have 'left' and 'right' keys`.

Alias not bound in the chain:

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "name": "a"},
    {"type": "Edge", "direction": "forward", "name": "e"},
    {"type": "Node", "name": "c"}
  ],
  "where": [{"eq": {"left": "missing.owner_id", "right": "c.owner_id"}}]
}
```

Expected error: `WHERE references aliases with no node/edge bindings: missing`.

Let Operation

Python:

```
let({
    'persons': n({'type': 'Person'}),
    'adults': ref('persons', [n({'age': ge(18)})])
})
```

Wire Format:

```
{
  "type": "Let",
  "bindings": {
    "persons": {
      "type": "Node",
```

(continues on next page)

(continued from previous page)

```
      "filter_dict": {"type": "Person"}
    },
    "adults": {
      "type": "Ref",
      "ref": "persons",
      "chain": [{
        "type": "Node",
        "filter_dict": {
          "age": {"type": "GE", "val": 18}
        }
      }]
    }
  }
}
```

Nested Let (Scope Isolation)

A `Let` binding value may itself be a `Let`. The inner `Let` executes as an opaque unit: its internal bindings are **not** visible in the outer scope. The outer `Let` sees only the binding name and the inner DAG's result.

Python:

```
let({
    'stage1': let({
        'people': n({'type': 'Person'}),
        'friends': ref('people', [e_forward(), n()])
    }),
    'stage2': ref('stage1', [e_forward(), n()])
})
```

Wire Format:

```
{
  "type": "Let",
  "bindings": {
    "stage1": {
      "type": "Let",
      "bindings": {
        "people": {"type": "Node", "filter_dict": {"type": "Person"}},
        "friends": {
          "type": "Ref", "ref": "people",
          "chain": [{"type": "Edge", "direction": "forward"}, {"type": "Node"}]
        }
      }
    },
    "stage2": {
      "type": "Ref", "ref": "stage1",
      "chain": [{"type": "Edge", "direction": "forward"}, {"type": "Node"}]
    }
  }
}
```

Scope rules (lexical scoping):

- `stage2` can reference `stage1` (an outer binding)
- `stage2` **cannot** reference `people` or `friends` (inner bindings — they do not leak upward)
- Inner bindings **can** read outer bindings (e.g., `people` could use `ref('stage2')` if `stage2` had already executed)
- Sibling inner `Let` blocks may reuse the same binding names without collision
- If an inner binding has the same name as an outer binding, the inner shadows the outer within its scope without corrupting the outer value
- The inner `Let` result is the last executed binding in its own scope

Ref Operation

Ref executes on the referenced graph; bindings used for edge traversal should retain edges (for example, from an `Edge` or `Chain` binding).

Python:

```
ref('base_graph', [
    e_forward({'weight': gt(0.5)}),
    n({'status': 'active'})
])
```

Wire Format:

```
{
  "type": "Ref",
  "ref": "base_graph",
  "chain": [
    {
      "type": "Edge",
      "direction": "forward",
      "edge_match": {"weight": {"type": "GT", "val": 0.5}}
    },
    {
      "type": "Node",
      "filter_dict": {"status": "active"}
    }
  ]
}
```

RemoteGraph Operation

Python:

```
remote(dataset_id='fraud-network-2024')
```

Wire Format:

```
{
  "type": "RemoteGraph",
  "dataset_id": "fraud-network-2024"
}
```

Call Operation

Python:

```
call('compute_cugraph', {'alg': 'pagerank', 'damping': 0.85})
```

Wire Format:

```
{
  "type": "Call",
  "function": "compute_cugraph",
  "params": {
    "alg": "pagerank",
    "damping": 0.85
  }
}
```

Note

For the complete list of safelisted layout calls—including the radial variants—refer to *GFQL Built-in Call Reference*.

Row-Pipeline Call Serialization

Row-pipeline operators use the same existing `Call` envelope. There is no wire-format envelope change for row pipelines; only `function`/`params` values vary by operator.

`rows`:

```
{"type": "Call", "function": "rows", "params": {"table": "nodes", "source": "q"}}
```

`where_rows`:

```
{"type": "Call", "function": "where_rows", "params": {"expr": "score >= 50"}}
```

`where_rows.expr` supports comparison operators: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`. `where_rows` can also use predicate dictionaries on the active row table:

```
{"type": "Call", "function": "where_rows", "params": {"filter_dict": {"score": {"type":
↪"GE", "val": 50}}}}
```

WHERE context summary:

- Chain-level same-path `where` uses lower-case operator keys (`eq`, `neq`, `lt`, `le`, `gt`, `ge`) with `left`/`right` alias-column references.

- Row-level `where_rows(filter_dict=...)` uses predicate envelopes like `GT`, `GE`, `LT`, `LE`, `EQ`, `NE` on active row-table columns.

`select`:

```
{"type": "Call", "function": "select", "params": {"items": [["id", "id"], ["score",
↪"score"]]}}
```

`with_`:

```
{"type": "Call", "function": "with_", "params": {"items": [["id", "id"]]}}
```

`order_by`:

```
{"type": "Call", "function": "order_by", "params": {"keys": [["score", "desc"], ["name",
↪"asc"]]}}
```

`skip`:

```
{"type": "Call", "function": "skip", "params": {"value": 20}}
```

`limit`:

```
{"type": "Call", "function": "limit", "params": {"value": 10}}
```

`distinct`:

```
{"type": "Call", "function": "distinct", "params": {}}
```

`unwind`:

```
{"type": "Call", "function": "unwind", "params": {"expr": "tags", "as_": "tag"}}
```

`group_by`:

```
{"type": "Call", "function": "group_by", "params": {"keys": ["category"], "aggregations":
↪ [["cnt", "count"], ["total", "sum", "amount"]]}}
```

`return_(...)` is serialized as `function: "select"` with equivalent `items`.

Row-Call Validation Errors

Row-call payloads are validated before execution. Invalid payloads fail fast.

Invalid `rows.table` enum:

```
{"type": "Call", "function": "rows", "params": {"table": "invalid"}}
```

Expected error: parameter validation failure (`table` must be `"nodes"` or `"edges"`).

Invalid `where_rows.expr` type:

```
{"type": "Call", "function": "where_rows", "params": {"expr": 123}}
```

Expected error: parameter validation failure (`expr` must be a non-empty string).

Invalid `order_by` direction:

```
{"type": "Call", "function": "order_by", "params": {"keys": [["score", "up"]]}}
```

Expected error: parameter validation failure (`direction` must be `"asc"` or `"desc"`).

Invalid `group_by` payload shape:

```
{"type": "Call", "function": "group_by", "params": {"keys": [], "aggregations": []}}
```

Expected error: parameter validation failure (non-empty keys and valid aggregation specs required).

Predicate Serialization

Comparison Predicates

```
{"type": "GT", "val": 100}
{"type": "LT", "val": 50.5}
{"type": "GE", "val": "2024-01-01"}
{"type": "LE", "val": true}
{"type": "EQ", "val": "active"}
{"type": "NE", "val": null}
```

Between Predicate

```
{
  "type": "Between",
  "lower": 10,
  "upper": 20,
  "inclusive": true
}
```

IsIn Predicate

```
{
  "type": "IsIn",
  "options": ["A", "B", "C"]
}
```

String Predicates

Basic forms (defaults: `case=true`, `na=null`, `flags=0`):

```
{"type": "Contains", "pat": "search", "case": true, "flags": 0, "na": null, "regex":␣
↪true}
{"type": "Startswith", "pat": "prefix", "case": true, "na": null}
{"type": "Endswith", "pat": "suffix", "case": true, "na": null}
{"type": "Match", "pat": "^[A-Z]+\\d+$", "case": true, "flags": 0, "na": null}
{"type": "Fullmatch", "pat": "^[A-Z]+$", "case": true, "flags": 0, "na": null}
```

Case-insensitive matching (using `case=false`):

```
{"type": "Startswith", "pat": "prefix", "case": false, "na": null}
{"type": "Fullmatch", "pat": "^test$", "case": false, "flags": 0, "na": null}
```

Tuple patterns (OR logic - match any):

```
{"type": "Startswith", "pat": ["app", "ban"], "case": true, "na": null}
{"type": "Endswith", "pat": [".jpg", ".png", ".gif"], "case": true, "na": null}
```

NA handling (fill value for missing data):

```
{"type": "Startswith", "pat": "test", "case": true, "na": false}
{"type": "Endswith", "pat": "end", "case": true, "na": true}
```

Notes:

- `pat`: Pattern string or array of strings (array uses OR logic)
- `case`: Case-sensitive if `true` (default: `true`)
- `na`: Fill value for null/missing values (default: `null` preserves NA)
- `flags`: Regex flags for `Match`/`Fullmatch` (default: `0`)
- `regex`: Whether pattern is regex for `Contains` (default: `true`)

Null Predicates

```
{"type": "IsNull"}
{"type": "NotNull"}
{"type": "IsNA"}
{"type": "NotNA"}
```

Temporal Check Predicates

```
{"type": "IsMonthStart"}
{"type": "IsYearEnd"}
{"type": "IsLeapYear"}
```

Type Serialization

Scalar Types

```
"hello world"        // string
42                   // integer
3.14159              // float
true                 // boolean
null                 // null
```

Temporal Types

DateTime

```
{
  "type": "datetime",
  "value": "2024-01-15T10:30:00",
  "timezone": "America/New_York"  // Optional, defaults to "UTC"
}
```

Date

```
{
  "type": "date",
  "value": "2024-01-15"
}
```

Time

```
{
  "type": "time",
  "value": "14:30:00.123456"
}
```

Temporal comparisons use standard predicate envelopes over these typed temporal values:

- `GT`, `GE`, `LT`, `LE`, `EQ`, `NE`

Example:

```
{
  "type": "GE",
  "val": {
    "type": "date",
    "value": "2024-01-01"
  }
}
```

Note: The `timezone` field is optional for DateTime values and defaults to "UTC" if omitted. This ensures consistent behavior across systems while allowing explicit timezone specification when needed.

Collections Payloads

Collections are Graphistry visualization overlays that use GFQL wire protocol operations to define subsets of nodes, edges, or subgraphs. They are applied in priority order, with earlier collections overriding later ones for styling.

Collection Set

Collection sets wrap GFQL operations in a `gfql_chain` object:

```
{
  "type": "set",
  "id": "purchasers",
  "name": "Purchasers",
  "node_color": "#00BFFF",
  "expr": {
    "type": "gfql_chain",
    "gfql": [
      {"type": "Node", "filter_dict": {"status": "purchased"}}
    ]
  }
}
```

Collection Intersection

Intersections reference previously defined set IDs:

```
{
  "type": "intersection",
  "name": "High Value Purchasers",
  "node_color": "#AA00AA",
  "expr": {
    "type": "intersection",
    "sets": ["purchasers", "vip"]
  }
}
```

For Python examples and helper constructors, see the :doc:`Collections tutorial notebook </demos/more_examples/graphistry_features/collections>`.

Examples

`MATCH ... RETURN` Row Pipeline

Python:

```
g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person"}, name="q"),
```

(continues on next page)

(continued from previous page)

```
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(25),
])
```

Wire Format:

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"type": "Person"}},
    {"type": "Edge", "direction": "forward", "edge_match": {"type": "FOLLOWS"}},
    {"type": "Node", "filter_dict": {"type": "Person"}, "name": "q"},
    {"type": "Call", "function": "rows", "params": {"table": "nodes", "source": "q"}},
    {"type": "Call", "function": "where_rows", "params": {"expr": "score >= 50"}},
    {"type": "Call", "function": "select", "params": {"items": [["id", "id"], ["name",
↪"name"], ["score", "score"]]}},
    {"type": "Call", "function": "order_by", "params": {"keys": [["score", "desc"], [
↪"name", "asc"]]}},
    {"type": "Call", "function": "limit", "params": {"value": 25}}
  ]
}
```

User 360 Query

Python:

```
g.gfql([
    n({"customer_id": "C123"}),
    e_forward({
        "type": "purchase",
        "timestamp": gt(pd.Timestamp("2024-01-01"))
    })
])
```

Wire Format:

```
{
  "type": "Chain",
  "chain": [
    {
      "type": "Node",
      "filter_dict": {
        "customer_id": "C123"
      }
    },
    {
      "type": "Edge",
      "direction": "forward",
```

(continues on next page)

(continued from previous page)

```
      "edge_match": {
        "type": "purchase",
        "timestamp": {
          "type": "GT",
          "val": {
            "type": "datetime",
            "value": "2024-01-01T00:00:00",
            "timezone": "UTC"
          }
        }
      }
    }
  ]
}
```

Cyber Security Pattern

Python:

```
g.gfql([
    n({"ip": is_in(["192.168.1.100", "192.168.1.101"])}),
    e_forward(
        edge_query="port IN [22, 23, 3389]",
        to_fixed_point=True
    ),
    n({"type": "server", "critical": True})
])
```

Wire Format:

```
{
  "type": "Chain",
  "chain": [
    {
      "type": "Node",
      "filter_dict": {
        "ip": {
          "type": "IsIn",
          "options": ["192.168.1.100", "192.168.1.101"]
        }
      }
    },
    {
      "type": "Edge",
      "direction": "forward",
      "edge_query": "port IN [22, 23, 3389]",
      "to_fixed_point": true
    },
    {
      "type": "Node",
      "filter_dict": {
```

(continues on next page)

(continued from previous page)

```
        "type": "server",
        "critical": true
      }
    }
  ]
}
```

Graph Constructors and the Wire Protocol

GFQL's Cypher extensions (`GRAPH { }` constructors, `GRAPH g = ...` bindings, `USE g` graph switching) serialize using the existing `Let`, `Chain`, `Call`, and `Ref` wire-protocol primitives. No new message types are needed.

Serialization

A multi-stage graph pipeline maps to a `Let` whose bindings are `Chain` or `Call` values, with `Ref` for `USE` references:

```
GRAPH g1 = GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 10 }
GRAPH g2 = GRAPH { USE g1 CALL graphistry.degree.write() }
USE g2 MATCH (n) RETURN n.id, n.degree ORDER BY n.degree DESC
```

```
{
  "type": "Let",
  "bindings": {
    "g1": {
      "type": "Chain",
      "chain": [
        {"type": "Node", "filter_dict": {"score": {"type": "GT", "val": 10}}, "name": "a
↪"},
        {"type": "Edge", "direction": "forward", "name": "r"},
        {"type": "Node", "name": "b"}
      ]
    },
    "g2": {
      "type": "Ref",
      "ref": "g1",
      "chain": [
        {"type": "Call", "function": "graphistry.degree.write", "params": {}}
      ]
    },
    "__result__": {
      "type": "Ref",
      "ref": "g2",
      "chain": [
        {"type": "Node", "name": "n"},
        {"type": "Call", "function": "rows", "params": {"table": "nodes", "source": "n"}}
↪,
        {"type": "Call", "function": "select", "params": {"items": [["id", "n.id"], [
```

(continues on next page)

(continued from previous page)

```
↪"degree", "n.degree"]]}},
        {"type": "Call", "function": "order_by", "params": {"keys": [["degree", "desc"]]}
↪}
      ]
    }
  }
}
```

The entire pipeline is a single `Let` message — one request, server-side evaluation.

Desugaring Reference

GFQL Extension	Wire Equivalent
`GRAPH { MATCH ... WHERE ... }`	`{"type": "Chain", "chain": [...], "where": [...]}`
`GRAPH { CALL graphistry.*.write() }`	`{"type": "Call", "function": "...", "params": {}}`
`GRAPH g = GRAPH { ... }`	Named `Let` binding — body is a `Chain` or `Call`
`USE g`	`Ref` with `"ref": "g"` — subsequent operations execute against g's result
`USE g MATCH ... RETURN ...`	`Ref` with `"ref": "g"` and the query chain as its body

Best Practices

1. **Always include type fields**: Every object must have a `type`
2. **Use ISO formats**: Dates and times in ISO 8601
3. **Handle timezones consistently**: Include timezone for datetime values when precision matters (defaults to UTC)
4. **Validate before sending**: Use JSON Schema validation
5. **Handle unknown fields**: Ignore unrecognized fields for compatibility

See Also

- *GFQL Language Specification* - Language specification
- *Cypher to GFQL Python & Wire Protocol Mapping* - Cypher to GFQL translation with wire protocol examples

10.5.19.4 Cypher to GFQL Python & Wire Protocol Mapping

GFQL supports Cypher syntax out of the box for a bounded read-only surface on bound graphs, while executing through GFQL's columnar engine with optional GPU acceleration. This page explains how to translate familiar Cypher patterns into native GFQL Python and wire protocol forms when you want more explicit control.

Introduction

Cypher is a graph query language popularized by Neo4j and related tools. In PyGraphistry, you can often start with a Cypher string directly through `g.gfql("MATCH ...")`, then translate that query into native GFQL when you want direct operator control, *Wire Protocol* JSON generation, migration from Cypher-centric systems, language-agnostic API integration, or secure query generation without code execution.

Direct `g.gfql("MATCH ...")` Note

If you want to **run** a supported Cypher string through `g.gfql("MATCH ...")` on a bound graph, use `g.gfql("MATCH ...")` (or `g.gfql("...", language="cypher")`) and start with *Cypher Syntax In GFQL*. This page stays translation-first: it explains how to express Cypher semantics in native GFQL operators and wire protocol, not the primary quickstart for direct Cypher syntax execution.

What Maps 1-to-1

When translating from Cypher, you'll encounter three scenarios:

1. **Direct Translation**: Most pattern matching maps cleanly to pure GFQL.
2. **Row-Pipeline Translation**: `RETURN/WITH/ORDER BY/SKIP/LIMIT/DISTINCT/GROUP BY` map to GFQL row operators.
3. **GFQL Advantages**: Some capabilities go beyond what Cypher offers.

Direct Translations

- Graph patterns: `(a)-[r]->(b)` → chain operations
- Property filters: WHERE clauses embed into operations
- Path traversals: direct `g.gfql("MATCH ...")` supports single and connected variable-length relationship forms such as `[*2]`, `[*1..3]`, and `[*]`, including bounded/exact variable-length `WHERE` pattern predicates in the current row-shaped subset. Native GFQL still gives you the full explicit hop surface (output slicing, intermediate-hop aliasing, and custom rewrites).
- Pattern composition: Multiple patterns become sequential operations
- Same-path constraints: `WHERE` across steps → `g.gfql([...], where=[...])`

Row-Pipeline Translation (`MATCH ... RETURN`)

- Row source selection: `rows(table=..., source=...)`
- Row filtering: `where_rows(filter_dict=..., expr=...)`
- Projection: `return_(...)` / `with_(...)` / `select(...)`
- Sorting/paging: `order_by(...)`, `skip(...)`, `limit(...)`
- Deduplication: `distinct()`
- Aggregation: `group_by(keys=[...], aggregations=[...])`

These row-pipeline operators are call steps inside the same chain list passed to `g.gfql([...])` (or to `Chain([...])`), not top-level `g.gfql()` keyword args:

```
from graphistry import n, e_forward
from graphistry.compute import rows, where_rows, return_, order_by, limit

g.gfql([
    n({"type": "Person"}, name="p"),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person"}, name="q"),
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_([("id", "id"), ("name", "name"), ("score", "score")]),
    order_by([("score", "desc")]),
    limit(25),
])
```

```
from graphistry.compute.chain import Chain

query = Chain([
    n({"type": "Person"}, name="p"),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person"}, name="q"),
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
])
g.gfql(query)
```

Projection sequencing and placement rules:

- Multiple `return_(...)` / `with_(...)` / `select(...)` steps are valid and execute in list order; each step projects from the current row table produced by previous steps.
- Interior mixing is invalid: do not place call steps between traversal steps (`n()/e_*()`), e.g. `[n(...), return_(...), e_forward(...)]`. Keep call steps in boundary prefix/suffix segments around traversal blocks.

When You Still Need DataFrames

- Translation targets outside the current pure GFQL operator surface, such as some `OPTIONAL MATCH` null-extension flows
- Arbitrary joins across disconnected intermediate result sets
- Custom functions outside the current row-expression subset

GFQL-Only Super-Powers

- **Edge properties**: Query edges as first-class entities
- **Dataframe-native**: Zero-cost transitions between graph and tabular operations
- **GPU acceleration**: Parallel execution on NVIDIA hardware
- **Heterogeneous graphs**: No schema constraints on types or properties
- **Integrated visualization**: Layouts like `group_in_a_box_layout` for community visualization
- **Algorithm chaining**: Combine community detection with layout algorithms

Quick Example

Cypher:

```
MATCH (p:Person)-[r:FOLLOWS]->(q:Person)
WHERE p.age > 30
```

Python:

```
g.gfql([
    n({"type": "Person", "age": gt(30)}, name="p"),
    e_forward({"type": "FOLLOWS"}, name="r"),
    n({"type": "Person"}, name="q")
])
```

Wire Protocol:

```
{"type": "Chain", "chain": [
  {"type": "Node", "filter_dict": {"type": "Person", "age": {"type": "GT", "val": 30}},
↪"name": "p"},
  {"type": "Edge", "direction": "forward", "edge_match": {"type": "FOLLOWS"}, "name": "r
↪"},
  {"type": "Node", "filter_dict": {"type": "Person"}, "name": "q"}
]}
```

Translation Tables

Node Patterns

Cypher	Python	Wire Protocol
`(n)`	`n()`	`{"type": "Node"}`
`(n:Label)`	`n({"type": "Label"})`	`{"type": "Node", "filter_dict": {"type": "Label "}}`
`(n {prop: val})`	`n({"prop": val})`	`{"type": "Node", "filter_dict": {"prop": val}}`
`(n) WHERE n.a > 10`	`n({"a": gt(10)})`	`{"type": "Node", "filter_dict": {"a": {"type": "GT", "val": 10}}}`
`(n:Person) WHERE n.age > 30`	`n({"type": "Person", "age": gt(30)})`	`{"type": "Node", "filter_dict": {"type": "Person", "age": {"type": "GT", "val": 30}}}`

Row-Pipeline Translation Tables

Use these as chain steps inside `g.gfql([...])` / `Chain([...])`.

Cypher	Python chain step	Wire Protocol call (compact)
`RETURN q.id, q.name`	`return_(["id", "name"])`	`{"type":"Call","function":"select","params":{"items":[["id","id"],["name","name"]]}}`
`RETURN q.id AS person_id`	`return_([("person_id", "id")])`	`{"type":"Call","function":"select","params":{"items":[["person_id","id"]]}}`
`WITH q.id AS id, q.score AS s`	`with_([("id", "id"), ("s", "score")])`	`{"type":"Call","function":"with_","params":{"items":[["id","id"],["s","score"]]}}`
`WHERE <row expr>` after `MATCH`	`where_rows(expr="score >= 50")`	`{"type":"Call","function":"where_rows", "params":{"expr":"score >= 50"}}`
`WHERE` with predicate helpers in row stage	`where_rows(filter_dict={ "created_at": gt(ts)})`	`{"type":"Call","function":"where_rows", "params":{"filter_dict":{"created_at":{ "type":"GT","val":...}}}}`
`ORDER BY score DESC, name ASC`	`order_by([("score", "desc"), ("name", "asc")])`	`{"type":"Call","function":"order_by", "params":{"keys":[["score","desc"],["name", "asc"]]}}`
`SKIP 20`	`skip(20)`	`{"type":"Call","function":"skip","params":{ "value":20}}`
`LIMIT 10`	`limit(10)`	`{"type":"Call","function":"limit","params": {"value":10}}`
`RETURN DISTINCT ...`	`distinct()`	`{"type":"Call","function":"distinct", "params":{}}`
`GROUP BY category` with `count(*)`	`group_by(keys=["category"], aggregations=[("cnt", "count")])`	`{"type":"Call","function":"group_by", "params":{"keys":["category"],"aggregations":[["cnt","count"]]}}`
Scope rows to alias `q`	`rows(table="nodes", source="q")`	`{"type":"Call","function":"rows","params":{ "table":"nodes","source":"q"}}`

Same-Path WHERE Predicates

Use `g.gfql([...], where=[...])` when the predicate compares multiple steps.

Cypher:

```
MATCH (n1)-[e1]->(n2)-[e2]->(n3)
WHERE n1.a > n2.b AND e1.x = e2.y
```

Python:

```
from graphistry import n, e_forward, col, compare

g.gfql(
    [n(name="n1"), e_forward(name="e1"), n(name="n2"), e_forward(name="e2"), n(name="n3
↪")],
    where=[
        compare(col("n1", "a"), ">", col("n2", "b")),
        compare(col("e1", "x"), "==", col("e2", "y")),
    ],
)
```

Wire Protocol:

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "name": "n1"},
    {"type": "Edge", "direction": "forward", "name": "e1"},
    {"type": "Node", "name": "n2"},
    {"type": "Edge", "direction": "forward", "name": "e2"},
    {"type": "Node", "name": "n3"}
  ],
  "where": [
    {"gt": {"left": "n1.a", "right": "n2.b"}},
    {"eq": {"left": "e1.x", "right": "e2.y"}}
  ]
}
```

`MATCH ... RETURN` Row Pipeline

Use GFQL call steps after the pattern match to encode Cypher `RETURN` behavior.

Cypher:

```
MATCH (p:Person)-[:FOLLOWS]->(q:Person)
WHERE q.score >= 50
RETURN q.id AS id, q.name AS name, q.score AS score
ORDER BY score DESC, name ASC
LIMIT 25
```

Python:

```python
from graphistry import n, e_forward, gt
from graphistry.compute import rows, where_rows, return_, order_by, limit

g.gfql([
    n({"type": "Person"}),
    e_forward({"type": "FOLLOWS"}),
    n({"type": "Person", "score": gt(0)}, name="q"),
    rows(table="nodes", source="q"),
    where_rows(expr="score >= 50"),
    return_(["id", "name", "score"]),
    order_by([("score", "desc"), ("name", "asc")]),
    limit(25),
])
```

Wire Protocol:

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"type": "Person"}},
    {"type": "Edge", "direction": "forward", "edge_match": {"type": "FOLLOWS"}},
    {"type": "Node", "filter_dict": {"type": "Person", "score": {"type": "GT", "val": 0}}
↪, "name": "q"},
    {"type": "Call", "function": "rows", "params": {"table": "nodes", "source": "q"}},
    {"type": "Call", "function": "where_rows", "params": {"expr": "score >= 50"}},
    {"type": "Call", "function": "select", "params": {"items": [["id", "id"], ["name",
↪"name"], ["score", "score"]]}},
    {"type": "Call", "function": "order_by", "params": {"keys": [["score", "desc"], [
↪"name", "asc"]]}},
    {"type": "Call", "function": "limit", "params": {"value": 25}}
  ]
}
```

Edge Patterns

Rows using `[*...]` below show the native GFQL rewrite for the same traversal intent. Direct `g.gfql("MATCH ...")` accepts these variable-length forms in the supported direct-Cypher subset, while native GFQL remains the more explicit option when you need intermediate-hop control or advanced path/list-carrier semantics.

Cypher / intent	Python	Wire Protocol (compact)
`-[]->`	`e_forward()`	`{"type": "Edge", "direction": "forward"}`
`-[r:KNOWS]->`	`e_forward({"type": "KNOWS"}, name="r")`	`{"type": "Edge", "direction": "forward", "edge_match": {"type": "KNOWS"}, "name": "r"}`
`<-[r]-`	`e_reverse(name="r")`	`{"type": "Edge", "direction": "reverse", "name": "r"}`
`-[r]-`	`e(name="r")`	`{"type": "Edge", "direction": "undirected", "name": "r"}`
`(n1)-[*2]->(n2`	`e_forward(min_hops=2, max_hops=2)`	`{"type": "Edge", "direction": "forward", "min_hops": 2, "max_hops": 2}`
`(n1)-[*1..3]->(n2)`	`e_forward(min_hops=1, max_hops=3)`	`{"type": "Edge", "direction": "forward", "min_hops": 1, "max_hops": 3}`
`(n1)-[*3..3]->(n2)`	`e_forward(min_hops=3, max_hops=3)`	`{"type": "Edge", "direction": "forward", "min_hops": 3, "max_hops": 3}`
`(n1)-[*2..4]->(n2)` but only show hops 3..4	`e_forward(min_hops=2, max_hops=4, output_min_hops=3, label_edge_hops="edge_hop")`	`{"type": "Edge", "direction": "forward", "min_hops": 2, "max_hops": 4, "output_min_hops": 3, "label_edge_hops": "edge_hop"}`
`(n1)-[*]->(n2)`	`e_forward(to_fixed_point=Tru`	`{"type": "Edge", "direction": "forward", "to_fixed_point": true}`
`-[r:BOUGHT {amount: gt(100)}]->`	`e_forward({"type": "BOUGHT", "amount": gt(100)}, name="r")`	`{"type": "Edge", "direction": "forward", "edge_match": {"type": "BOUGHT", "amount": {"type": "GT", "val": 100}}, "name": "r"}`

When you need constraints on intermediate hops, path/list-carrier semantics, or advanced row-shaping beyond the current direct-Cypher subset, use repeated single-hop GFQL steps with aliases instead of collapsing the traversal into one multihop edge operator.

Predicates

Cypher	Python	Wire Protocol
`n.status = 'active'`	`"active"`	`"active"`
`n.age > 30`	`gt(30)`	`{"type": "GT", "val": 30}`
`n.age >= 50`	`ge(50)`	`{"type": "GE", "val": 50}`
`n.age < 100`	`lt(100)`	`{"type": "LT", "val": 100}`
`n.age <= 50`	`le(50)`	`{"type": "LE", "val": 50}`
`n.status <> 'deleted'`	`ne("deleted")`	`{"type": "NE", "val": "deleted"}`
`n.id IN [1,2,3]`	`is_in([1,2,3])`	`{"type": "IsIn", "options": [1,2,3]}`
`n.score BETWEEN 0 AND 100`	`between(0, 100)`	`{"type": "Between", "lower": 0, "upper": 100}`
`n.name =~ '^A.*'`	`match("^A.*")`	`{"type": "Match", "pattern": "^A.*"}`
`n.text CONTAINS 'search'`	`contains("search")`	`{"type": "Contains", "pattern": "search"}`
`n.name STARTS WITH 'Dr'`	`startswith("Dr")`	`{"type": "Startswith", "pattern": "Dr"}`
`n.email ENDS WITH '.com'`	`endswith(".com")`	`{"type": "Endswith", "pattern": ".com"}`
`n.val IS NULL`	`isnull()`	`{"type": "IsNull"}`
`n.val IS NOT NULL`	`notnull()`	`{"type": "NotNull"}`

Complete Examples

Friend of Friend

Cypher:

```
MATCH (u:User {name: 'Alice'})-[:FRIEND*2]->(fof:User)
WHERE fof.active = true
```

Python:

```
g.gfql([
    n({"type": "User", "name": "Alice"}),
    e_forward({"type": "FRIEND"}, min_hops=2, max_hops=2),
    n({"type": "User", "active": True}, name="fof")
])
```

Wire Protocol:

```
{"type": "Chain", "chain": [
  {"type": "Node", "filter_dict": {"type": "User", "name": "Alice"}},
  {"type": "Edge", "direction": "forward", "edge_match": {"type": "FRIEND"}, "min_hops":␣
↪2, "max_hops": 2},
  {"type": "Node", "filter_dict": {"type": "User", "active": true}, "name": "fof"}
]}
```

Same-Path Constraint

Cypher:

```
MATCH (a:Account)-[:TRANSFER]->(c:User)
WHERE a.owner_id = c.owner_id
```

Python:

```
from graphistry import n, e_forward, col, compare

g.gfql(
    [n({"type": "Account"}, name="a"), e_forward(), n({"type": "User"}, name="c")],
    where=[compare(col("a", "owner_id"), "==", col("c", "owner_id"))],
)
```

Wire Protocol:

```
{"type": "Chain", "chain": [
  {"type": "Node", "filter_dict": {"type": "Account"}, "name": "a"},
  {"type": "Edge", "direction": "forward"},
  {"type": "Node", "filter_dict": {"type": "User"}, "name": "c"}
], "where": [{"eq": {"left": "a.owner_id", "right": "c.owner_id"}}]}
```

Fraud Detection

Cypher:

```
MATCH (a:Account)-[t:TRANSFER]->(b:Account)
WHERE t.amount > 10000 AND t.date > date('2024-01-01')
```

Python:

```
g.gfql([
    n({"type": "Account"}),
    e_forward({
        "type": "TRANSFER",
        "amount": gt(10000),
        "date": gt(date(2024,1,1))
    }, name="t"),
    n({"type": "Account"})
])
```

Wire Protocol:

```
{"type": "Chain", "chain": [
  {"type": "Node", "filter_dict": {"type": "Account"}},
  {"type": "Edge", "direction": "forward", "edge_match": {
    "type": "TRANSFER",
    "amount": {"type": "GT", "val": 10000},
    "date": {"type": "GT", "val": {"type": "date", "value": "2024-01-01"}}
  }, "name": "t"},
  {"type": "Node", "filter_dict": {"type": "Account"}}
]}
```

Complex Aggregation Example

Cypher:

```
MATCH (u:User)-[t:TRANSACTION]->(m:Merchant)
WHERE t.date > date('2024-01-01')
RETURN t.category AS category, count(*) as cnt, sum(t.amount) as total
ORDER BY total DESC
LIMIT 10
```

Python:

```
from datetime import date
from graphistry import n, e_forward, gt
from graphistry.compute import rows, where_rows, group_by, return_, order_by, limit

analysis = g.gfql([
    n({"type": "User"}),
    e_forward({"type": "TRANSACTION", "date": gt(date(2024, 1, 1))}, name="t"),
    rows(table="edges", source="t"),
    where_rows(expr="amount IS NOT NULL"),
```

(continues on next page)

(continued from previous page)

```
    group_by(
        keys=["category"],
        aggregations=[
            ("cnt", "count", "amount"),
            ("total", "sum", "amount"),
        ],
    ),
    return_(["category", "cnt", "total"]),
    order_by([("total", "desc")]),
    limit(10),
])
```

Note: If the aggregation/function you need is outside the supported `group_by` subset, fall back to dataframe post-processing.

Row-Pipeline Operations Mapping

Cypher Feature	GFQL Python Row Operation	Notes
`RETURN a, b, c`	`return_(["a", "b", "c"])`	String item shorthand maps `a -> ("a", "a")`
`WITH a, b`	`with_(["a", "b"])`	Same projection semantics as `return_`
`RETURN DISTINCT`	`distinct()`	Deduplicate active row table
`ORDER BY x DESC`	`order_by([("x", "desc")])`	Multi-key sorting supported
`SKIP 20`	`skip(20)`	Row offset
`LIMIT 10`	`limit(10)`	Row cap
`WHERE <row expr>`	`where_rows(expr="...")`	Scalar expression subset
`count(*)`	`group_by(keys=[...], aggregations=[("cnt", "count")])`	Grouped count
`sum(n.val)`	`group_by(..., aggregations=[("total", "sum", "val")])`	Grouped sum
`collect(n.x)`	`group_by(..., aggregations=[("xs", "collect", "x")])`	Nulls excluded from collection
Named patterns	`rows(source="alias")`	Scope row table to a named match alias

Key Differences

Feature	Python	Wire Protocol
Temporal values	`pd.Timestamp()`, `date()`	`{"type": "date", "value": "..."}`
Direct equality	`"active"`	`"active"` (same)
Comparisons	`gt(30)`	`{"type": "GT", "val": 30}`
Collections	`is_in([...])`	`{"type": "IsIn", "options": [...]}`

GFQL Extension: Graph Constructors (`GRAPH { }`)

Standard Cypher and GQL's first edition (ISO/IEC 39075:2024) have no way to return a graph from a query — every result is a flat table of binding rows. GFQL extends Cypher with graph constructors that keep results in graph state, enabling composable graph-in / graph-out pipelines.

Syntax and Desugaring

Graph constructors compile down to the same Chain/Call wire-protocol primitives as regular queries — **no new wire types are needed**.

Cypher (GFQL extension)	Desugars to
`GRAPH { MATCH (a)-[r]->(b) WHERE a.x > 10 }`	`Chain([n("a"), e_forward("r"), n("b")], where=[...])` — executed in graph state
`GRAPH { CALL graphistry.degree.write() }`	`Call("graphistry.degree.write")` — graph-preserving procedure
`GRAPH g = GRAPH { ... }`	Named binding (like `Let`) whose value is the Chain/Call result
`USE g MATCH ... RETURN ...`	Execute the following query against binding `g`'s graph

Examples

Standalone graph constructor (graph state out):

```
# Cypher extension
g2 = g.gfql("GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 10 }")

# Equivalent native GFQL (inherently graph-returning)
g2 = g.gfql([n({"score": ge(10)}), e_forward(), n()])
```

Multi-stage pipeline in one expression:

```
result = g.gfql(
    "GRAPH g1 = GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 10 } "
    "GRAPH g2 = GRAPH { USE g1 CALL graphistry.degree.write() } "
    "USE g2 MATCH (n) RETURN n.id, n.degree ORDER BY n.degree DESC"
)
```

Wire protocol — the pipeline above compiles locally to a sequence of Chain + Call executions against resolved graph bindings. The server sees ordinary Chain/Call messages, not graph-constructor-specific wire types.

Design Notes

- `GRAPH { }` is a **closed scope** — pattern variables inside do not leak
- `USE` is **lexically scoped** — bindings must be defined before USE
- Only graph-preserving `CALL ... .write()` procedures are allowed inside constructors (row-returning CALL is rejected)
- `cypher_to_gfql()` rejects multi-graph pipelines (they can't be represented as a single Chain); use `g.gfql("...")` for direct execution instead

Not Supported

- `CREATE`, `DELETE`, `SET`: GFQL is read-only.
- `OPTIONAL MATCH`: direct `g.gfql("MATCH ...")` execution supports a bounded subset, but pure GFQL translation still has no single general operator for full outer-join/null-extension semantics.
- Full Cypher expression/function surface in row expressions: current vectorized subset only.
- Multiple disconnected `MATCH` patterns in one query: use separate GFQL chains and explicit dataframe joins.

Practical fallback: keep pattern traversal and row-pipeline stages in GFQL, then apply final custom dataframe logic in pandas/cuDF when needed.

Best Practices

1. **Direct Translation First**: Try pure GFQL before adding DataFrame operations
2. **Use Named Patterns**: Label important results with `name=` for easy access
3. **Filter Early**: Apply selective node filters before traversing edges
4. **Type Consistency**: Ensure wire protocol types match expected column types
5. **Validate JSON**: Test wire protocol against schema before sending

LLM Integration Guide

When building translators:

```
Given Cypher: {cypher_query}

Generate both:
1. Python: Human-readable GFQL code
2. Wire Protocol: JSON for API calls

Rules:
- (n:Label) → Python: n({"type": "Label"}) → JSON: {"type": "Node", "filter_dict": {
↪"type": "Label"}}
- Cross-step WHERE → `g.gfql([...], where=[compare(col(...), op, col(...))])`
- RETURN/WITH/ORDER BY/SKIP/LIMIT/DISTINCT/GROUP BY → row-pipeline call steps (`rows`,␣
↪`where_rows`, `return_`, `order_by`, `skip`, `limit`, `distinct`, `group_by`)
- Unsupported expressions/functions → explicitly mark as unsupported instead of␣
↪silently rewriting
```

See Also

- *GFQL Wire Protocol Specification* - Full wire protocol specification
- *GFQL Language Specification* - Language specification
- *GFQL Python Embedding* - Python implementation details

10.5.19.5 GFQL JSON Generation Guide for LLMs

What is GFQL?

GFQL (GraphFrame Query Language) is a Cypher-like JSON AST for graph queries with three purposes:

1. **Graph Search**: Pattern matching with node/edge chains (filtering, traversal, etc.)
2. **Graph Algorithms**: Examples: PageRank, Louvain, UMAP, hypergraph, and more
3. **Visualization**: Encodings for color, icon, size, and more

---

Table of Contents

Quick Example: Fraud Detection

Dense: `let({'suspicious': n({'risk_score': gt(80)}), 'flows': [n({'risk_score': gt(80)}), e_forward(min_hops=1, max_hops=3), n()], 'ranked': ref('flows', [call('compute_cugraph', {'alg': 'pagerank'})]), 'viz': ref('ranked', [call('encode_point_color', {...}), call('encode_point_icon', {...})])})`

JSON:

```
{
  "type": "Let",
  "bindings": {
    "suspicious": {
      "type": "Chain",
      "chain": [{"type": "Node", "filter_dict": {"risk_score": {"type": "GT", "val": 80}}
↪}]
    },
    "flows": {
      "type": "Chain",
      "chain": [
        {"type": "Node", "filter_dict": {"risk_score": {"type": "GT", "val": 80}}},
        {"type": "Edge", "direction": "forward", "min_hops": 1, "max_hops": 3, "to_fixed_
↪point": false,
         "edge_match": {"amount": {"type": "GT", "val": 10000}}},
        {"type": "Node", "filter_dict": {}}
      ]
    },
    "ranked": {
      "type": "Ref",
      "ref": "flows",
      "chain": [{"type": "Call", "function": "compute_cugraph", "params": {"alg":
↪"pagerank"}}]
    },
    "viz": {
      "type": "Ref",
      "ref": "ranked",
      "chain": [
        {"type": "Call", "function": "encode_point_color",
         "params": {"column": "risk_score", "palette": ["green", "yellow", "red"], "as_
↪continuous": true}},
        {"type": "Call", "function": "encode_point_icon",
         "params": {"column": "type", "categorical_mapping": {"account": "credit-card"}}}
      ]
    }
  }
}
```

Core Types

Chain (Container)

```
{"type": "Chain", "chain": [Node|Edge|Call, ...]}
```

Node (Filter nodes)

```
{
  "type": "Node",
  "filter_dict": {"col": value | predicate},  // required
  "name": "label"                              // optional
}
```

Edge (Traverse)

```
{
  "type": "Edge",
  "direction": "forward|reverse|undirected",   // required
  "max_hops": 1,                               // default: 1 (hops shorthand)
  "min_hops": 1,                               // optional; default 1 unless max_hops is␣
↪0
  "output_min_hops": 1,                        // optional post-filter slice; omit to␣
↪keep min_hops..max_hops
  "output_max_hops": 1,                        // optional post-filter cap; omit to keep␣
↪max_hops
  "label_node_hops": "hop",                    // optional; omit/null to skip node hop␣
↪labels
  "label_edge_hops": "edge_hop",               // optional; omit/null to skip edge hop␣
↪labels
  "label_seeds": false,                        // optional; when true, label seeds at␣
↪hop 0
  "to_fixed_point": false,                     // default: false
  "edge_match": {filters},                     // optional
  "source_node_match": {filters},              // optional
  "destination_node_match": {filters},         // optional
  "name": "label"                              // optional
}
```

Call (Operation)

```
{"type": "Call", "function": "name", "params": {...}}
```

Let (Multi-step)

```
{"type": "Let", "bindings": {"name": Chain | Ref}}
```

Bindings may themselves be `Let` (nested let). Scope follows lexical rules:

- Inner bindings do **not** leak upward (outer cannot see `a` or `b` below)
- Inner bindings **can** read outer bindings (lexical closure)
- Sibling inner `Let` blocks may reuse names without collision
- Shadowing: inner name same as outer does not corrupt the outer value

```
{"type": "Let", "bindings": {
  "stage1": {"type": "Let", "bindings": {"a": ..., "b": ...}},
  "stage2": {"type": "Ref", "ref": "stage1", "chain": [...]}
}}
```

Ref (Reference)

```
{"type": "Ref", "ref": "name", "chain": [operations]}
```

Predicates

Comparison:

```
{"type": "GT|LT|GE|LE|EQ|NE", "val": value}
```

Membership:

```
{"type": "IsIn", "options": [values]}
{"type": "Between", "lower": 10, "upper": 100}
```

String:

```
{"type": "Contains|Startswith|Endswith", "pat": "text", "case": true, "regex": false}
```

Null:

```
{"type": "IsNull|NotNull"}
```

Temporal:

```
{"type": "datetime", "value": "2024-01-15T10:30:00", "timezone": "UTC"}
{"type": "date", "value": "2024-01-15"}
```

Note: Raw values work for equality: `{"age": 30}` equals `{"age": {"type": "EQ", "val": 30}}`

Simple Search

Basic pattern matching:

```
# Dense: [n({'type': 'Person', 'name': 'Alice'}), e_forward(), n({'type': 'Person'})]
```

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"type": "Person", "name": "Alice"}},
    {"type": "Edge", "direction": "forward", "hops": 1, "to_fixed_point": false},
    {"type": "Node", "filter_dict": {"type": "Person"}}
  ]
}
```

Multi-hop (friends of friends only):

```
# Dense: [n({'name': 'Alice'}), e_forward(min_hops=2, max_hops=2), n()]
```

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"name": "Alice"}},
    {"type": "Edge", "direction": "forward", "min_hops": 2, "max_hops": 2, "to_fixed_
↪point": false},
    {"type": "Node", "filter_dict": {}}
  ]
}
```

Fixed-point (traverse until no new nodes):

```
# Dense: [n({'compromised': True}), e_forward(to_fixed_point=True), n({'critical': True}
↪)]
```

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"compromised": true}},
    {"type": "Edge", "direction": "forward", "to_fixed_point": true},
    {"type": "Node", "filter_dict": {"critical": true}}
  ]
}
```

Reverse edges (follow edges backward):

```
# Dense: [n({'type': 'product'}), e_reverse(), n({'type': 'supplier'})]
```

```
{
  "type": "Chain",
  "chain": [
    {"type": "Node", "filter_dict": {"type": "product"}},
    {"type": "Edge", "direction": "reverse", "hops": 1, "to_fixed_point": false},
    {"type": "Node", "filter_dict": {"type": "supplier"}}
  ]
}
```

Filter After Enrichment

Pattern: Compute metric → Filter by result

```
# Dense: let({'enriched': call('get_degrees', {'col': 'degree'}), 'filtered': ref(
↪'enriched', [n({'degree': gt(10)})])})
```

```
{
  "type": "Let",
  "bindings": {
    "enriched": {
      "type": "Chain",
      "chain": [{"type": "Call", "function": "get_degrees", "params": {"col": "degree"}}]
    },
    "filtered": {
      "type": "Ref",
      "ref": "enriched",
      "chain": [{"type": "Node", "filter_dict": {"degree": {"type": "GT", "val": 10}}}]
    }
  }
}
```

Graph Algorithms

Centrality

PageRank: `call('compute_cugraph', {'alg': 'pagerank', 'out_col': 'score'})`

```
{"type": "Call", "function": "compute_cugraph", "params": {"alg": "pagerank", "out_col":
↪"score"}}
```

Betweenness Centrality: `call('compute_cugraph', {'alg': 'betweenness_centrality', 'out_col': 'bc'})`

```
{"type": "Call", "function": "compute_cugraph", "params": {"alg": "betweenness_centrality
↪", "out_col": "bc"}}
```

Katz Centrality: `call('compute_cugraph', {'alg': 'katz_centrality', 'out_col': 'katz'})`

```
{"type": "Call", "function": "compute_cugraph", "params": {"alg": "katz_centrality",
↪"out_col": "katz"}}
```

Community Detection

Louvain: `call('compute_cugraph', {'alg': 'louvain', 'out_col': 'community'})`

```
{"type": "Call", "function": "compute_cugraph", "params": {"alg": "louvain", "out_col":
↪"community"}}
```

Similarity

Jaccard Coefficient: `call('compute_cugraph', {'alg': 'jaccard', 'out_col': 'similarity'})`

```
{"type": "Call", "function": "compute_cugraph", "params": {"alg": "jaccard", "out_col":
↪"similarity"}}
```

Graph Transforms

Hypergraph (Events → Entities): `call('hypergraph', {'entity_types': ['user', 'product'], 'direct': True})`

```
{"type": "Call", "function": "hypergraph", "params": {"entity_types": ["user", "product
↪"], "direct": true}}
```

UMAP (2D Embedding): `call('umap', {'kind': 'nodes', 'n_neighbors': 15, 'n_components': 2})`

```
{"type": "Call", "function": "umap", "params": {"kind": "nodes", "n_neighbors": 15, "n_
↪components": 2}}
```

Collapse Nodes: `call('collapse', {'node': 'category', 'attribute': 'type'})`

```
{"type": "Call", "function": "collapse", "params": {"node": "category", "attribute":
↪"type"}}
```

Materialize Nodes: `call('materialize_nodes', {'column': 'relationship'})`

```
{"type": "Call", "function": "materialize_nodes", "params": {"column": "relationship"}}
```

Degree Operations

All Degrees: `call('get_degrees', {'col': 'degree', 'col_in': 'in_deg', 'col_out': 'out_deg'})`

```
{"type": "Call", "function": "get_degrees", "params": {"col": "degree", "col_in": "in_deg
↪", "col_out": "out_deg"}}
```

In-Degrees Only: `call('get_indegrees', {'col': 'in_degree'})`

```
{"type": "Call", "function": "get_indegrees", "params": {"col": "in_degree"}}
```

Out-Degrees Only: `call('get_outdegrees', {'col': 'out_degree'})`

```
{"type": "Call", "function": "get_outdegrees", "params": {"col": "out_degree"}}
```

Topological Levels: `call('get_topological_levels', {'level_col': 'level'})`

```
{"type": "Call", "function": "get_topological_levels", "params": {"level_col": "level"}}
```

Traversals & Filters

Hop (Multi-step): `call('hop', {'min_hops': 1, 'max_hops': 3, 'direction': 'forward'})`

```
{"type": "Call", "function": "hop", "params": {"min_hops": 1, "max_hops": 3, "direction":
↪ "forward"}}
```

Filter Nodes: `call('filter_nodes_by_dict', {'query': {'type': 'Person', 'age': {'type': 'GT', 'val': 30}}})`

```
{"type": "Call", "function": "filter_nodes_by_dict", "params": {"query": {"type": "Person
↪", "age": {"type": "GT", "val": 30}}}}
```

Filter Edges: `call('filter_edges_by_dict', {'query': {'amount': {'type': 'GT', 'val': 1000}}})`

```
{"type": "Call", "function": "filter_edges_by_dict", "params": {"query": {"amount": {
↪"type": "GT", "val": 1000}}}}
```

Drop Nodes: `call('drop_nodes', {'nodes': ['node_1', 'node_2']})`

```
{"type": "Call", "function": "drop_nodes", "params": {"nodes": ["node_1", "node_2"]}}
```

Keep Nodes: `call('keep_nodes', {'nodes': ['important_1', 'important_2']})`

```
{"type": "Call", "function": "keep_nodes", "params": {"nodes": ["important_1",
↪"important_2"]}}
```

igraph Algorithms

igraph Wrapper: `call('compute_igraph', {'alg': 'community_leiden', 'out_col': 'community'})`

```
{"type": "Call", "function": "compute_igraph", "params": {"alg": "community_leiden",
↪"out_col": "community"}}
```

Visualization

Gradient Color (continuous):

```
# Dense: call('encode_point_color', {'column': 'risk', 'palette': ['green','yellow','red
↪'], 'as_continuous': True})
```

```
{"type": "Call", "function": "encode_point_color",
 "params": {"column": "risk", "palette": ["green", "yellow", "red"], "as_continuous":␣
↪true}}
```

Categorical Color:

```
# Dense: call('encode_point_color', {'column': 'dept', 'categorical_mapping': {'sales':
↪'blue', 'eng': 'green'}})
```

```
{"type": "Call", "function": "encode_point_color",
 "params": {"column": "dept", "categorical_mapping": {"sales": "blue", "eng": "green"}}}
```

Icons (FontAwesome 4):

```
# Dense: call('encode_point_icon', {'column': 'type', 'categorical_mapping': {'server':
↪'server', 'laptop': 'laptop'}})
```

```
{"type": "Call", "function": "encode_point_icon",
 "params": {"column": "type", "categorical_mapping": {"server": "server", "laptop":
↪"laptop"}}}
```

Size:

```
# Dense: call('encode_point_size', {'column': 'importance', 'categorical_mapping': {'low
↪': 10, 'high': 40}})
```

```
{"type": "Call", "function": "encode_point_size",
 "params": {"column": "importance", "categorical_mapping": {"low": 10, "high": 40}}}
```

Edge Color: `call('encode_edge_color', {'column': 'weight', 'palette': ['#ccc', '#000'], 'as_continuous': True})`

```
{"type": "Call", "function": "encode_edge_color",
 "params": {"column": "weight", "palette": ["#ccc", "#000"], "as_continuous": true}}
```

Recommended Palettes:

Continuous Gradients:

- Risk/Heat: `["#00b894", "#fdcb6e", "#d63031"]` (green→yellow→red)
- Cyber Threat: `["#2E86AB", "#A23B72", "#F18F01"]` (cold→warm)
- Influence: `["#95a5a6", "#3498db", "#e74c3c"]` (gray→blue→red)
- Cool→Warm: `["#3498db", "#9b59b6", "#e67e22"]` (blue→purple→orange)

Categorical (Colorblind-Safe):

- Okabe-Ito: `["#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79A7"]`
- Safe 4-color: `["#4477AA", "#EE6677", "#228833", "#CCBB44"]`

Cool Linking (Same Hue):

- Blues: `["#EBF5FB", "#85C1E9", "#2E86C1", "#1B4F72"]` (varying saturation)
- Greens: `["#E8F8F5", "#7DCEA0", "#27AE60", "#145A32"]`

See *Domain Guidance* for domain-specific palette recommendations.

Layouts

Default: ForceAtlas2 (FA2)

Graphistry uses ForceAtlas2 by default, which produces excellent force-directed layouts for most graph structures.

Recommendation: Do NOT override layout in most cases. FA2 handles general graphs well.

When to Override:

- Time-series data → Use ring layouts
- Hierarchical data → Use tree/graphviz layouts
- Grouped communities → Use group_in_a_box_layout

Time-Based Ring Layout: `call('time_ring_layout', {'time_col': 'timestamp', 'num_rings': 10})`

```
{"type": "Call", "function": "time_ring_layout",
 "params": {"time_col": "timestamp", "num_rings": 10}}
```

> **Tip:** Supply `time_start` / `time_end` as ISO strings (e.g., `"2024-01-01T00:00:00"`) when needed. The executor converts them to `numpy.datetime64`, matching the Python Plotter API.

Categorical Ring Layout: `call('ring_categorical_layout', {'ring_col': 'category', 'num_rings': 5})`

```
{"type": "Call", "function": "ring_categorical_layout",
 "params": {"ring_col": "category", "num_rings": 5}}
```

Continuous Ring Layout: `call('ring_continuous_layout', {'ring_col': 'score', 'num_rings': 8})`

```
{"type": "Call", "function": "ring_continuous_layout",
 "params": {"ring_col": "score", "num_rings": 8}}
```

Other Layouts:

- `layout_cugraph` - GPU-accelerated cuGraph layouts
- `layout_igraph` - igraph layouts (CPU)
- `layout_graphviz` - Graphviz layouts (hierarchical)
- `group_in_a_box_layout` - Group-in-a-box with community partitioning

Multi-Step (Let/Ref)

Pattern: Named DAG stages - `let({'step1': ..., 'step2': ref('step1', [...]), 'step3': ref('step2', [...])})`

See *Quick Example* for full JSON example.

Domain Guidance

Cyber/IT Security:

- Icons: server, laptop, mobile, shield, lock, bug, exclamation-triangle, database, firewall, cloud
- Colors: Blues/grays (trusted), reds/oranges (threats), yellows (warnings)
- Palettes: `["#2E86AB", "#A23B72", "#F18F01"]` (cold→warm threat levels)
- Patterns: Lateral movement (fixed-point), privilege escalation, kill chains

Fraud/Finance:

- Icons: credit-card, bank, shopping-cart, dollar-sign, exclamation-triangle, flag, money, exchange
- Colors: Greens (legitimate), yellows (suspicious), reds (fraudulent)
- Palettes: `["#00b894", "#fdcb6e", "#d63031"]` (risk gradients)
- Patterns: Transaction chains, velocity analysis, clustering by behavior

Government/Enforcement/IC/Military:

- Icons: shield, flag, star, building, briefcase, user-secret, crosshairs, certificate
- Colors: Blues (official), reds (adversary), greens (allied)
- Palettes: Classification levels, threat actors by affiliation
- Patterns: Attribution chains, actor networks, timeline analysis

Social Media:

- Icons: user, users, comment, heart, share, camera, envelope, bell, rss, hashtag
- Colors: Cool blues (low engagement) → warm oranges (high engagement)
- Palettes: `["#95a5a6", "#3498db", "#e74c3c"]` (influence levels)
- Patterns: Influence ranking, community detection, virality tracking

Supply Chain/Logistics:

- Icons: truck, plane, ship, warehouse, box, industry, shopping-bag, map-marker, cog

- Colors: Greens (on-time), yellows (delayed), reds (critical)
- Palettes: `["#27ae60", "#f39c12", "#c0392b"]` (status indicators)
- Patterns: Recall tracing (reverse), bottleneck detection, route optimization

Event/People/Digital:

- Icons: calendar, clock, user, desktop, mobile, tablet, globe, sitemap
- Colors: Time-based gradients, entity type differentiation
- Palettes: Blues (past) → purples (present) → oranges (future)
- Patterns: Timeline layouts (ring), entity relationships, digital footprints

---

Call Functions

Algorithms: `compute_cugraph` (pagerank, louvain, betweenness), `compute_igraph`, `get_degrees`, `get_indegrees`, `get_outdegrees`

Transforms: `hypergraph`, `umap`, `collapse`, `materialize_nodes`

Filters: `filter_nodes_by_dict`, `filter_edges_by_dict`, `hop`, `drop_nodes`, `keep_nodes`

Layouts: `layout_cugraph`, `layout_igraph`, `layout_graphviz`, `fa2_layout`, `ring_continuous_layout`, `ring_categorical_layout`, `time_ring_layout`, `group_in_a_box_layout`, `circle_layout`, `tree_layout`, `mercator_layout`, `modularity_weighted_layout`

Encodings: `encode_point_color`, `encode_edge_color`, `encode_point_size`, `encode_point_icon`

Generation Rules

1. **Always include `type` field** in every object
2. **Chain wraps operations** - use `{"type": "Chain", "chain": [...]}`
3. **Edge defaults:** `direction: "forward"`, `max_hops: 1` (`hops` shorthand), `min_hops: 1` unless `max_hops` is 0, `to_fixed_point: false`
4. **Output slice defaults:** If `output_min_hops`/`output_max_hops` are omitted, results keep all traversed hops up to `max_hops`; set them to post-filter displayed hops.
5. **Empty filters:** Use `{}` for match-all
6. **Predicates:** Wrap comparisons: `{"type": "GT", "val": 100}`
7. **Temporal:** Tag values: `{"type": "datetime", "value": "...", "timezone": "UTC"}`
8. **Ref:** Reference bindings: `{"type": "Ref", "ref": "name", "chain": [...]}`

Common Mistakes

Wrong: Missing type: `{"filter_dict": {...}}` **Correct:** `{"type": "Node", "filter_dict": {...}}`

Wrong: Raw datetime: `{"timestamp": "2024-01-01"}` **Correct:** `{"timestamp": {"type": "GT", "val": {"type": "datetime", "value": "2024-01-01T00:00:00"}}}`

Wrong: Forgot to_fixed_point: `{"max_hops": 999}` for "traverse all" **Correct:** `{"to_fixed_point": true}`

Wrong: Using `"backward"` instead of `"reverse"` **Correct:** `{"direction": "reverse"}`

10.5.19.6 Overview

- *GFQL Language Specification* - Complete formal grammar, operations, predicates, and type system
- *GFQL Python Embedding* - Python-specific implementation with pandas/cuDF
- *GFQL Wire Protocol Specification* - JSON serialization format for client-server communication
- *Cypher to GFQL Python & Wire Protocol Mapping* - Cypher to GFQL translations with both Python and wire protocol
- *GFQL JSON Generation Guide for LLMs* - LLM-optimized guide for generating valid GFQL JSON (Claude, GPT, etc.)

These specifications are optimized for text-to-GFQL synthesis, Cypher-to-GFQL pipelines, query validation, and schema-aware code generation.

10.5.20 GFQL Validation Guide

Learn how to validate GFQL queries for syntax correctness, schema compatibility, and production use.

10.5.20.1 GFQL Validation Fundamentals

Learn how to use GFQL's built-in validation system to catch errors early and build robust graph applications.

> **Note**
>
> This guide is accompanied by an interactive Jupyter notebook. To run the examples yourself, see GFQL Validation Fundamentals notebook.

What You'll Learn

- How GFQL automatically validates queries
- Understanding structured error messages with error codes
- Schema validation against your data
- Pre-execution validation for performance
- Collecting all errors vs fail-fast mode

Prerequisites

- Basic Python knowledge
- PyGraphistry installed (`pip install graphistry[ai]`)

Quick Start

```
from graphistry.compute.chain import Chain
from graphistry.compute.ast import n, e_forward
from graphistry.compute.exceptions import GFQLValidationError

# Automatic validation during construction
try:
    chain = Chain([
        n({'type': 'customer'}),
        e_forward(),
        n()
    ])
    print("Valid chain created!")
except GFQLValidationError as e:
    print(f"Error: [{e.code}] {e.message}")
```

Key Concepts

Built-in Validation

GFQL validates automatically - no separate validation calls needed:

- **Syntax validation**: Happens during chain construction
- **Schema validation**: Happens by default during `g.chain()` execution
- **Structured errors**: Error codes (E1xx, E2xx, E3xx) for programmatic handling

Error Types

- **GFQLSyntaxError** (E1xx): Structural issues in query
- **GFQLTypeError** (E2xx): Type mismatches and invalid values
- **GFQLSchemaError** (E3xx): Missing columns, incompatible types

Common Errors and Fixes

Invalid Parameters

```
# Wrong - negative hops
try:
    chain = Chain([n(), e_forward(hops=-1)])
except GFQLTypeError as e:
    print(f"Error: {e.message}")  # "hops must be a positive integer"

# Correct
chain = Chain([n(), e_forward(hops=2)])
```

Missing Columns

```
# Wrong - column doesn't exist
try:
    result = g.chain([n({'category': 'VIP'})])
except GFQLSchemaError as e:
    print(f"Error: {e.message}")  # Column "category" does not exist
    print(f"Suggestion: {e.context.get('suggestion')}")

# Correct - use existing columns
result = g.chain([n({'type': 'customer'})])
```

Type Mismatches

```
# Wrong - string value on numeric column
try:
    result = g.chain([n({'score': 'high'})])
except GFQLSchemaError as e:
    print(f"Error: {e.message}")  # Type mismatch

# Correct - use numeric predicate
from graphistry.compute.predicates.numeric import gt
result = g.chain([n({'score': gt(80)})])
```

Temporal Comparisons

```
import pandas as pd
from graphistry.compute.predicates.numeric import gt, lt

# Compare datetime columns
result = g.chain([
    n({'created_at': gt(pd.Timestamp('2024-01-01'))})
])
```

(continues on next page)

(continued from previous page)

```
# Find recent activity (last 7 days)
result = g.chain([
    e_forward({
        'timestamp': gt(pd.Timestamp.now() - pd.Timedelta(days=7))
    })
])
```

How Validation Works

Default Behavior

GFQL validates automatically - just write your queries and run them:

```
# Validation happens automatically
result = g.chain([n({'type': 'customer'})])

# Errors are caught and reported clearly
try:
    result = g.chain([n({'invalid_column': 'value'})])
except GFQLSchemaError as e:
    print(f"Error: {e.message}")
```

Pre-Execution Validation Options

Use the inline GFQL entrypoints first:

1. `g.gfql_validate(...)` for validate-only preflight (no execution)
2. `g.gfql(..., validate=True)` for preflight + execution
3. `validate_chain_schema()` for low-level chain-schema checks only

`g.gfql_validate(...)` (validate-only, no execution) supports:

- **Input forms**: Cypher strings, GFQL JSON payloads, and GFQL Python objects (for example `Chain(...)`, `[n(), e(), n()]`, and `ASTLet(...)`) String inputs are always validated as Cypher (no separate string-shape precheck).
- **Predicate + structural validation**: yes
- **Schema validation**:
  - GFQL JSON and GFQL Python chain-like forms: yes (default `schema=True`)
  - GFQL Let/DAG forms: DAG structure + schema checks for direct graph-bound steps; reference-based steps stay structural-only
  - Cypher strings: syntax/compile + schema-aware name checks against the bound graph schema by default (`strict=True`); pass `strict=False` for syntax/compile-only preflight

```
# Chain / JSON-style GFQL
g.gfql_validate([n({'type': 'customer'})], collect_all=True)
```

(continues on next page)

(continued from previous page)

```
# Cypher
g.gfql_validate("MATCH (c) RETURN c.id AS id LIMIT $n", params={"n": 10})
```

Validation failures raise `GFQLValidationError` / `GFQLSyntaxError` with structured, inspectable context:

```
from graphistry.compute.exceptions import GFQLValidationError

try:
    g.gfql_validate([n({"missing_col": "x"})], collect_all=True)
except GFQLValidationError as exc:
    payload = exc.to_dict()
    # LM-friendly payload:
    # {
    #   "code": "...",
    #   "message": "...",
    #   "query_type": "chain",
    #   "language": "gfql",
    #   "diagnostics": [...]
    # }
    print(payload)
```

`g.gfql(..., validate=True)` accepts the same query inputs as `g.gfql(...)` (Cypher string, GFQL JSON, GFQL Python objects), runs local preflight first, and executes only when preflight passes. Its preflight uses `g.gfql_validate(...)` defaults, so local bound-graph execution runs schema-aware checks by default.

```
# Run preflight first; execute only if preflight passes
result = g.gfql(
    "MATCH (c) RETURN c.id AS id LIMIT $n",
    params={"n": 10},
    validate=True,
)
```

Use `validate_chain_schema()` when you specifically want the low-level chain-schema helper. It is intentionally narrower than `g.gfql_validate(...)`:

- validates chain operations against currently bound node/edge dataframe columns
- does **not** parse/compile Cypher strings
- does **not** run Let/DAG orchestration validation
- does **not** execute query operators

```
from graphistry.compute.validate_schema import validate_chain_schema

# Step 1: Validate (no execution)
try:
    validate_chain_schema(g, chain)  # Only validates, doesn't execute
    print("Chain is valid for this graph schema")
except GFQLSchemaError as e:
    print(f"Schema incompatibility: {e}")

# Step 2: Execute (after validation passes)
result = g.gfql(chain.chain)
print(f"Query executed: {len(result._nodes)} nodes")
```

Execution-time Preflight Toggles

For remote execution, `g.gfql_remote(..., validate=True)` runs local query prevalidation before implicit upload/network execution, so invalid queries fail before data upload when possible. For Cypher strings, remote prevalidation uses `strict=False` by default because the authoritative schema is on the remote dataset.

Grounded vs Ungrounded Validation

Schema checks are most useful when local graph tables are bound on `g`. If local node/edge tables are missing, GFQL JSON/AST chain validation can only do structural/predicate checks, and column-existence checks are effectively ungrounded.

Error Collection

Choose between fail-fast and collect-all modes:

```
# Fail-fast (default)
try:
    chain = Chain([problematic_operations])
except GFQLValidationError as e:
    print(f"First error: {e}")

# Collect all errors
errors = chain.validate(collect_all=True)
for error in errors:
    print(f"[{error.code}] {error.message}")
```

Next Steps

- *GFQL Validation for LLMs* - AI integration patterns
- *GFQL Validation in Production* - Production deployment patterns

See Also

- *GFQL Language Specification* - Complete language specification
- *Overview of GFQL* - GFQL overview

10.5.20.2 GFQL Validation for LLMs

Learn how to integrate GFQL's built-in validation with Large Language Models and automation pipelines.

> **Note**
>
> Explore the complete examples in GFQL Validation Fundamentals notebook.

Target Audience

- AI/ML Engineers building GFQL generation systems
- Developers integrating LLMs with graph queries
- Teams building automated query generation pipelines

JSON Integration

GFQL queries use JSON for LLM integration:

```
{
    "type": "Chain",
    "chain": [
        {"type": "Node", "filter_dict": {"type": "user"}},
        {"type": "Edge", "direction": "forward", "hops": 2},
        {"type": "Node", "filter_dict": {"score": {"type": "GT", "val": 80}}}
    ]
}
```

Validation Workflow

Parse and Validate

```
from graphistry.compute.exceptions import GFQLValidationError
from graphistry.compute.chain import Chain

def json_to_chain(json_data):
    """Parse JSON from LLM into query object."""
    try:
        return Chain.from_json(json_data, validate=True)
    except GFQLValidationError as e:
        # Handle parse errors
        return None, e

def chain_to_json(chain):
    """Convert query to JSON for LLM training/examples."""
    return chain.to_json(validate=False)  # Already validated

def validation_error_to_dict(error: GFQLValidationError) -> dict:
    """Convert validation error to LLM-friendly format."""
    return {
        "code": error.code,
        "message": error.message,
        "field": error.context.get("field"),
        "value": str(error.context.get("value")) if error.context.get("value") else None,
        "suggestion": error.context.get("suggestion"),
        "operation_index": error.context.get("operation_index"),
        "error_type": error.__class__.__name__
    }
```

Validate Query Syntax

```
# Validate syntax and structure
syntax_errors = chain.validate(collect_all=True)

if syntax_errors:
    print(f"Found {len(syntax_errors)} syntax errors")
    for error in syntax_errors:
        print(f"  [{error.code}] {error.message}")
```

Validate Against Schema

```
from graphistry.compute.validate_schema import validate_chain_schema

# Validate against actual data schema
schema_errors = []
if g:  # Your Plottable instance with data
    schema_errors = validate_chain_schema(g, chain, collect_all=True) or []

    if schema_errors:
        print(f"Found {len(schema_errors)} schema errors")
        for error in schema_errors:
            print(f"  [{error.code}] {error.message}")
```

Combined Validation

```
# Complete validation pipeline
def validate_llm_query(json_data, graph=None):
    """Full validation with detailed feedback."""
    # Parse
    result = json_to_chain(json_data)
    if isinstance(result, tuple):
        return {"success": False, "parse_errors": [validation_error_to_dict(result[1])]}

    chain = result

    # Validate syntax
    syntax_errors = chain.validate(collect_all=True)

    # Validate schema if graph provided
    schema_errors = []
    if graph:
        schema_errors = validate_chain_schema(graph, chain, collect_all=True) or []

    # Return results
    if syntax_errors or schema_errors:
        return {
            "success": False,
            "syntax_errors": [validation_error_to_dict(e) for e in syntax_errors],
```

(continues on next page)

(continued from previous page)

```
            "schema_errors": [validation_error_to_dict(e) for e in schema_errors]
        }

    return {"success": True, "chain": chain}
```

Direct Preflight For Retry Loops

For generate-validate-repair loops, you can run `g.gfql_validate(...)` and convert raised exceptions into structured payloads:

```
from graphistry.compute.exceptions import GFQLValidationError, GFQLSyntaxError

def preflight_payload(g, query):
    try:
        g.gfql_validate(query, collect_all=True)
        return {"ok": True}
    except (GFQLValidationError, GFQLSyntaxError) as exc:
        payload = exc.to_dict()
        return {
            "ok": False,
            "error": payload,  # includes code/message + diagnostics context
        }
```

Automated Fix Suggestions

Generate actionable suggestions using structured error context:

```
def generate_fix_suggestions(errors):
    """Generate fix suggestions from validation errors."""
    fixes = []

    for error in errors:
        fix = {
            "error_code": error.code,
            "operation_index": error.context.get("operation_index"),
            "field": error.context.get("field"),
            "current_value": error.context.get("value"),
            "suggested_action": error.context.get("suggestion")
        }

        # Add specific fix actions based on error code
        if error.code == ErrorCode.E103:  # Invalid parameter value (e.g., negative hops)
            fix["action"] = "replace_parameter"
            # Extract valid value from suggestion if present
            if "positive integer" in error.message:
                fix["fix_hint"] = "Use a positive integer value"
        elif error.code == ErrorCode.E301:  # Column not found
            fix["action"] = "replace_column"
            # Available columns are in the suggestion text
```

(continues on next page)

(continued from previous page)

```
            if error.context.get("suggestion") and "Available columns:" in error.context.
↪get("suggestion"):
                fix["available_columns_hint"] = error.context.get("suggestion")
        elif error.code == ErrorCode.E302:  # Type mismatch
            fix["action"] = "fix_type_mismatch"
            fix["column_type"] = error.context.get("column_type")

        fixes.append(fix)

    return fixes
```

Best Practices

1. **Built-in Validation**: Use GFQL's automatic validation during construction
2. **Error Codes**: Leverage structured error codes (E1xx, E2xx, E3xx) for programmatic handling
3. **Collect-All Mode**: Use `collect_all=True` for comprehensive error reporting to LLMs
4. **Schema Context**: Provide available columns and types in LLM prompts
5. **Pre-execution Validation**: Validate schema before expensive operations

See Also

- *GFQL Validation in Production* - Production patterns
- *GFQL Language Specification* - Language specification
- *Cypher to GFQL Python & Wire Protocol Mapping* - Cypher to GFQL mapping

10.5.20.3 GFQL Validation in Production

Production-ready patterns for GFQL built-in validation in platform engineering and DevOps contexts.

> **Note**
>
> See complete implementation examples in GFQL Validation Fundamentals notebook.

Target Audience

- Platform Engineers
- DevOps Teams
- Backend Developers
- System Architects

Performance & Caching

Schema Caching

```
from functools import lru_cache
import time

class CachedSchemaValidator:
    def __init__(self, cache_size=1000, ttl_seconds=3600):
        self.cache_size = cache_size
        self.ttl_seconds = ttl_seconds
        self._cache = {}
        self._cache_times = {}

        # Cache the actual validation function
        self._validate_uncached = lru_cache(maxsize=cache_size)(
            self._validate_impl
        )

    def _validate_impl(self, operations_hash, plottable_id):
        """Actual validation implementation."""
        # Implementation depends on having stable plottable references
        pass

    def validate(self, operations, plottable):
        """Validate with caching."""
        # Use built-in validation with caching layer
        cache_key = (hash(str(operations)), id(plottable))

        if cache_key in self._cache:
            cache_time = self._cache_times.get(cache_key, 0)
            if time.time() - cache_time < self.ttl_seconds:
                return self._cache[cache_key]

        # Perform validation
        from graphistry.compute.validate.validate_schema import validate_chain_schema
        result = validate_chain_schema(plottable, operations)

        # Cache result
        self._cache[cache_key] = result
        self._cache_times[cache_key] = time.time()

        return result
```

Batch Validation

```
from graphistry.compute.validate.validate_schema import validate_chain_schema

def batch_validate_queries(operation_sets, plottable):
    """Validate multiple queries efficiently with built-in validation."""
    results = []
    for operations in operation_sets:
        try:
            errors = validate_chain_schema(plottable, operations)
            results.append({
                "valid": len(errors) == 0,
                "errors": [
                    {
                        "code": e.code,
                        "message": e.message,
                        "field": e.context.get("field"),
                        "suggestion": e.context.get("suggestion")
                    }
                    for e in errors
                ]
            })
        except Exception as e:
            results.append({
                "valid": False,
                "error": str(e)
            })

    return results
```

Testing Patterns

pytest Fixtures

```
import pytest
import pandas as pd
import graphistry
from graphistry.compute.chain import Chain
from graphistry.compute.ast import n, e_forward
from graphistry.compute.predicates.comparison import eq
from graphistry.compute.exceptions import GFQLValidationError

@pytest.fixture
def sample_plottable():
    nodes = pd.DataFrame({
        'id': [1, 2, 3],
        'type': ['A', 'B', 'A']
    })
    edges = pd.DataFrame({
        'src': [1, 2],
```

(continues on next page)

(continued from previous page)

```
        'dst': [2, 3]
    })
    g = graphistry.nodes(nodes, node='id').edges(edges, source='src', destination='dst')
    return g

def test_valid_query(sample_plottable):
    operations = [n({'type': eq('A')})]

    # Test syntax validation
    chain = Chain(operations)  # Should not raise

    # Test schema validation
    result = sample_plottable.chain(operations)  # Should not raise
    assert len(result._nodes) > 0

def test_invalid_query_syntax(sample_plottable):
    with pytest.raises(GFQLValidationError) as exc_info:
        chain = Chain([n({'type': eq('A')}, name=123)])  # Invalid name type
    assert exc_info.value.code.startswith('E2')  # Type error

def test_invalid_query_schema(sample_plottable):
    operations = [n({'missing_column': eq('value')})]

    with pytest.raises(GFQLValidationError) as exc_info:
        result = sample_plottable.chain(operations)  # Schema validation fails
    assert exc_info.value.code == 'E301'  # Column not found
```

API Integration

Flask Example

```
from flask import Flask, request, jsonify
from graphistry.compute.chain import Chain
from graphistry.compute.exceptions import GFQLValidationError
from graphistry.compute.ast import from_json

app = Flask(__name__)

@app.route('/api/v1/validate', methods=['POST'])
def validate_gfql():
    data = request.get_json()
    operations_json = data.get('operations')

    try:
        # Parse operations from JSON
        operations = [from_json(op) for op in operations_json]

        # Validate syntax (automatic during Chain construction)
        chain = Chain(operations)
        syntax_errors = chain.validate(collect_all=True)
```

(continues on next page)

(continued from previous page)

```
        # Prepare response
        response = {
            'valid': len(syntax_errors) == 0,
            'errors': [
                {
                    'code': e.code,
                    'message': e.message,
                    'field': e.context.get('field'),
                    'suggestion': e.context.get('suggestion')
                }
                for e in syntax_errors
            ]
        }

        return jsonify(response)

    except Exception as e:
        return jsonify({
            'valid': False,
            'error': str(e)
        }), 400

@app.route('/api/v1/validate-with-schema', methods=['POST'])
def validate_gfql_with_schema():
    data = request.get_json()
    operations_json = data.get('operations')
    plottable_data = data.get('plottable')  # Serialized plottable

    try:
        # Parse operations from JSON
        operations = [from_json(op) for op in operations_json]

        # Would need to reconstruct plottable from data
        # and use validate_chain_schema
        from graphistry.compute.validate_schema import validate_chain_schema

        # This is a placeholder - actual implementation would need
        # to deserialize plottable_data into a plottable instance
        # errors = validate_chain_schema(plottable, operations, collect_all=True)

        return jsonify({
            'valid': True,
            'message': 'Schema validation endpoint placeholder'
        })

    except Exception as e:
        return jsonify({
            'valid': False,
            'error': str(e)
        }), 500
```

Security Considerations

GFQL is designed with security in mind to prevent arbitrary code execution:

Safe Query Generation

Instead of generating Python code directly, generate JSON and use GFQL's validation:

```
# DON'T: Generate Python code (security risk)
# query = f"g.chain([n({{'user_id': '{user_input}'}})])"
# eval(query)  # NEVER DO THIS

# DO: Generate JSON and validate
query_json = {
    "type": "Chain",
    "chain": [{
        "type": "Node",
        "filter_dict": {"user_id": user_input}
    }]
}

# Safe parsing with validation
from graphistry.compute.chain import Chain
chain = Chain.from_json(query_json, validate=True)
result = g.chain(chain.chain)
```

Key Security Features

1. **No Code Execution**: GFQL operations are data structures, not executable code
2. **Input Validation**: All inputs are validated against strict schemas
3. **Type Safety**: Strong typing prevents injection attacks
4. **Bounded Operations**: Queries have defined limits (e.g., max hops)

Rate Limiting Example

```
from collections import defaultdict
import time

class RateLimiter:
    def __init__(self, requests_per_minute=100):
        self.requests_per_minute = requests_per_minute
        self.request_times = defaultdict(list)

    def check_rate_limit(self, user_id):
        now = time.time()
        user_requests = self.request_times[user_id]

        # Clean old requests
        user_requests[:] = [t for t in user_requests if now - t < 60]

        if len(user_requests) >= self.requests_per_minute:
            return False, f"Rate limit exceeded. Try again in {60 - (now - user_
↪requests[0]):.0f} seconds"
```

(continues on next page)

(continued from previous page)

```
        user_requests.append(now)
        return True, None
```

Production Checklist

- **Built-in Validation**: Use GFQL's automatic validation system
- **Caching**: Implement validation result caching
- **Batch Processing**: Validate multiple queries efficiently
- **Testing**: Comprehensive test coverage with pytest
- **API Design**: RESTful endpoints with structured error responses
- **Security**: Generate JSON instead of Python code, use validation
- **Rate Limiting**: Implement per-user request limits
- **Error Codes**: Use structured error codes for programmatic handling

Performance Guidelines

1. **Schema Validation**: Use validate_schema=True (default) for production safety
2. **Pre-execution Validation**: Validate before expensive operations
3. **Caching**: Cache validation results with appropriate TTL
4. **Batch Processing**: Use collect_all=True for multiple error reporting
5. **Rate Limiting**: Set reasonable per-user request limits

Next Steps

- Implement production validation service
- Create runbooks for common issues
- Establish performance benchmarks

See Also

- *GFQL Wire Protocol Specification* - Wire protocol specification
- https://docs.graphistry.com/api/
- https://docs.graphistry.com/deployment/

10.5.20.4 Overview

The GFQL validation framework provides comprehensive tools for ensuring query correctness:

- **Syntax Validation** - Check query structure without data
- **Schema Validation** - Verify queries against actual data schemas
- **Combined Validation** - Full validation with helpful error messages
- **LLM Integration** - Structured formats for AI code generation
- **Production Patterns** - Caching, monitoring, and CI/CD integration

Start with *GFQL Validation Fundamentals* to learn the basics, then explore advanced topics based on your needs.

10.6 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.

> **Note**
>
> `graphistry.cypher("...")` / `g.cypher("...")` below refer to the remote database Cypher integrations. For Cypher syntax through GFQL on a bound graph, use `g.gfql("MATCH ...")`; for remote GFQL execution, use `g.gfql_remote([...])`. See *Cypher Syntax In GFQL*.

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

10.6.1 Databases

10.6.1.1 Graph

- https://aws.amazon.com/neptune (*graphistry.gremlin.NeptuneMixin*)
- https://www.arangodb.com (external link)
- https://tinkerpop.apache.org (*graphistry.gremlin.GremlinMixin*)
- https://memgraph.com (external link) (*graphistry.PlotterBase.PlotterBase.cypher()*)
- https://neo4j.com (external link) (*graphistry.PlotterBase.PlotterBase.cypher()*)
- https://cloud.google.com/spanner/docs/graph/overview (`graphistry.pygraphistry.PyGraphistry.spanner_gql()`)
- https://www.tigergraph.com (external link) (*graphistry.PlotterBase.PlotterBase.gsql()*)
- https://trovares.com (external link)

10.6.1.2 Document, Key-Value, Log, Text, and SIEM

- https://aws.amazon.com/dynamodb
- https://azure.microsoft.com/en-us/services/cosmos-db (*graphistry.gremlin.CosmosMixin*)
- https://azure.microsoft.com/en-us/services/data-explorer
- https://cassandra.apache.org
- https://www.elastic.co
- https://opensearch.org
- https://redis.io
- https://www.splunk.com (external link)

10.6.1.3 SQL

Typically accessed via dataframe bindings

When available, we recommend exploring for accelerated bindings via https://arrow.apache.org/docs/format/ADBC.html

- https://aws.amazon.com/athena
- https://databricks.com (external link)
- https://opensearch.org
- https://www.postgresql.org
- https://aws.amazon.com/redshift
- https://cloud.google.com/bigquery
- https://www.snowflake.com (external link)
- https://www.microsoft.com/en-us/sql-server

10.6.2 Compute engines

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

- https://spark.apache.org
- https://pandas.pydata.org/
- https://docs.rapids.ai/api/cudf/stable/

Partial native support:

- https://docs.rapids.ai/api/cuml/stable/
- https://www.dask.org/
- https://docs.rapids.ai/api/cudf/stable/dask-cudf.html

Accelerated interop via https://arrow.apache.org/ or Parquet:

- https://duckdb.org/
- https://www.pola.rs/
- https://spark.apache.org/

10.6.3 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

10.6.4 Tools

We are constantly experimenting, feel free to add:

- OWASP Amass

10.6.5 Storage engines and file formats

GPU-accelerated readers via https://docs.rapids.ai/api/cudf/stable/ (in-memory single-GPU) and https://docs.rapids.ai/api/cudf/stable/dask-cudf.html (bigger-than-memory, multi-GPU):

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

Others, often via https://filesystem-spec.readthedocs.io/en/latest/:

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

10.7 CPU & GPU Acceleration in PyGraphistry

10.7.1 Why PyGraphistry is Fast

PyGraphistry is designed for speed. By focusing on **vectorized processing**, it outperforms most graph libraries on standard CPUs. When you leverage GPUs and AI models, PyGraphistry can become **100X+ faster**, enabling real-time analytics and machine learning at scale. We regularly use it on datasets with millions and billions of rows.

Just as Apache Spark used in-memory processing to replace racks of Hadoop servers with faster and smaller multicore ones, the PyGraphistry ecosystem uses GPU acceleration to increase speeds and decrease costs even further.

10.7.2 Flexible GPU Use: Client and Server

Strictly optional, PyGraphistry lets you harness GPUs where they make the most sense for your workflow. For smaller datasets, you can run PyGraphistry on your local GPU. Graph loading, shaping, computing, querying, ML, AI, and visualization tasks all become much more interactive and immediate, making PyGraphistry great for exploration in Jupyter notebooks and dashboards.

For larger datasets and team projects, you can offload PyGraphistry tasks like **GFQL queries** and **visualization ETL**, and even full GPU Python scripts, to shared Graphistry GPU servers. This setup handles enterprise-grade workloads, helping deliver consistent performance across web apps, dashboards, and AI pipelines.

10.7.3 Where PyGraphistry Accelerates with Vector Processing and GPUs

PyGraphistry uses vector processing and GPU acceleration throughout your data workflow.

In data processing, it integrates with **Apache Arrow** to seamlessly transition between **pandas** for algorithmic and hardware acceleration of datasets even on CPUs, and **cuDF** (via https://rapids.ai/) for large, GPU-accelerated workloads, keeping your data pipelines efficient at any size. Graphistry is typically used on GPUs with 12-80 GB single-GPU RAM, and we increasingly work with teams experimenting with multi-GPU nodes (128-640 GB GPU RAM) and clusters of them.

For graph querying, **GFQL** leverages GPUs to speed up queries on massive graph datasets, delivering results in seconds on a single GPU even when traversal steps touch hundreds of millions of rows.

In visualization, GPUs enable PyGraphistry to render large, complex graphs in real time. Whether you're investigating cybersecurity threats, monitoring supply chains, or analyzing clickstreams, you get responsive visuals at any scale, locally or via shared servers.

For AI and machine learning, **PyGraphistry[AI]** uses GPUs to accelerate model training and inference for tasks like **UMAP** and **GNNs**, unlocking rapid insights from large graph datasets in areas like security and commercial analytics. When running on real-time data and billions of rows, the combination of GPU training and GPU inferencing unlocks significant velocity.

10.7.4 Easy Deployment Anywhere

The Graphistry ecosystem fits into your existing infrastructure.

You can https://www.graphistry.com/get-started on any modern cloud platform (https://aws.amazon.com/, https://cloud.google.com/, https://azure.microsoft.com/en-us/), and on-premises using **Docker Compose** or **Kubernetes**. PyGraphistry works with any NVIDIA GPU that are https://rapids.ai/.

If you don't have a GPU, no problem. PyGraphistry is a quick *pip install graphistry* away, giving performance optimized for CPU hardware through vectorized columnar processing concepts similar to https://clickhouse.com/ and https://spark.apache.org/. You can also offload heavy tasks to remote Graphistry shared GPUs, including Graphistry Hub visualization servers.

10.7.5 Trusted Security & Compliance

Many top organizations with sensitive environments — including global banks and air-gapped government systems — trust PyGraphistry. Regular security practices such as periodic penetration testing ensure systems meets strict security requirements, making it safe for some of the most stringent teams.

10.7.6 Next Steps

Get started with PyGraphistry:

- **Installation Guide**: Set up PyGraphistry .
- **Visualization**: Explore *10 Minutes to PyGraphistry*.
- **GFQL Documentation**: Start with *10 Minutes to GFQL*.

10.8 Notebook Tutorials

10.8.1 Getting Started

10.8.1.1 Tutorial: Data Analysis in Graphistry

1. Register
2. Load table
3. Plot:
    - Simple: input is a list of edges
    - Arbitrary: input is a table (*hypergraph* transform)
4. Advanced plotting
5. Further reading
    - https://github.com/graphistry/pygraphistry
    - https://github.com/graphistry/pygraphistry/tree/master/demos/demos_databases_apis
    - https://github.com/graphistry/graph-app-kit
    - https://hub.graphistry.com/docs/ui/index/
    - https://github.com/graphistry/pygraphistry/tree/master/demos/upload_csv_miniapp.ipynb

1. Register

[101]:
```
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

2. Load table

Graphistry works seamlessly with dataframes like https://pandas.pydata.org/ and GPU https://www.rapids.ai

[94]:
```
import pandas as pd

df = pd.read_csv('./data/honeypot.csv')

df.sample(3)
```

[94]:
```
         attackerIP      victimIP  victimPort          vulnName  count  \
64    178.77.190.33  172.31.14.66       445.0  MS08067 (NetAPI)      6
7    112.209.78.240  172.31.14.66       445.0  MS08067 (NetAPI)     10
182  79.140.174.193  172.31.14.66       445.0  MS08067 (NetAPI)      2

        time(max)     time(min)
64   1.419968e+09  1.419967e+09
7    1.414516e+09  1.414514e+09
182  1.422062e+09  1.422062e+09
```

3. Plot

A. Simple graphs

- Build up a set of bindings. Simple graphs are:
  - required: edge table, with src+dst ID columns, and optional additional property columns
  - optional: node table, with matching node ID column
- See https://hub.graphistry.com/docs/ui/index/ for in-tool activity

Demo graph schema:

- **Input table**: Above alerts `df` with columns `| attackerIP | victimIP |`
- **Edges**: Link `df`'s columns `attackerIP -> victimIP`
- **Nodes**: Unspecified; Graphistry defaults to generating based on the edges
- **Node colors**: Graphistry defaults to inferring the commmunity
- **Node sizes**: Graphistry defaults to the number of edges ("degree")

[16]:
```
g = graphistry.edges(df, 'attackerIP', 'victimIP')
```

[17]:
```
g.plot()
```

[17]:
```
<IPython.core.display.HTML object>
```

B. Hypergraphs – Plot arbitrary tables

The hypergraph transform is a convenient method to transform tables into graphs:

- It extracts entities from the table and links them together
- Entities get linked together when they are from the same row

Approach 1: Treat each row as a node, and link it to each cell value in it

Demo graph schema: * **Edges**: row -> attackerIP, row -> victimIP, row -> victimPort, row -> volnName * **Nodes**: row, attackerIP, victimIP, victimPort, vulnName * **Node colors**: Automatic based on inferred commmunity * **node sizes**: Number of edges

[93]:
```
hg1 = graphistry.hypergraph(
    df,

    # Optional: Subset of columns to turn into nodes; defaults to all
    entity_types=['attackerIP', 'victimIP', 'victimPort', 'vulnName'],

    # Optional: merge nodes when their IDs appear in multiple columns
    # ... so replace nodes attackerIP::1.1.1.1 and victimIP::1.1.1.1
    # ... with just one node ip::1.1.1.1
    opts={
        'CATEGORIES': {
            'ip': ['attackerIP', 'victimIP']
        }
    })

hg1_g = hg1['graph']
hg1_g.plot()
```

```
# links 880
# events 220
# attrib entities 221
```

[93]:
```
<IPython.core.display.HTML object>
```

Approach 2: Link values from column entries

For more advanced hypergraph control, we can skip the row node, and control which edges are generated, by enabling `direct`.

Demo graph schema: * Edges: * attackerIP -> victimIP, attackerIP -> victimPort, attackerIP -> vulnName * victimPort -> victimIP * vulnName -> victimIP * Nodes: attackerIP, victimIP, victimPort, vulnName * Default colors: Automatic based on inferred commmunity * Default node size: Number of edges

[102]:
```
hg2 = graphistry.hypergraph(
    df,
    entity_types=['attackerIP', 'victimIP', 'victimPort', 'vulnName'],
    direct=True,
    opts={
        # Optional: Without, creates edges that are all-to-all for each row
```

(continues on next page)

(continued from previous page)

```
        'EDGES': {
            'attackerIP': ['victimIP', 'victimPort', 'vulnName'],
            'victimPort': ['victimIP'],
            'vulnName': ['victimIP']
        },

        # Optional: merge nodes when their IDs appear in multiple columns
        # ... so replace nodes attackerIP::1.1.1.1 and victimIP::1.1.1.1
        # ... with just one node ip::1.1.1.1
        'CATEGORIES': {
            'ip': ['attackerIP', 'victimIP']
        }
    })

hg2_g = hg2['graph']
hg2_g.plot()
```

```
# links 1100
# events 220
# attrib entities 221
```

[102]:
```
<IPython.core.display.HTML object>
```

3. Advanced plotting

You can then drive visual styles based on node and edge attributes

This demo starts by computing a node table. By default, you do not need to explictly provide a table of nodes, but then you may lack data for node properties:

- Regular inferred graph nodes will only have id and degree
- Hypergraph edges and row nodes will have many properties, but hypergraph entity nodes will only have id, type/category, and degree

Demo schema:

- **Node table:** `| node_id | type | attacks |`
- **Point size**: number of attacks
- **Point icon & color**: attacker vs victim
- **Edge color**: based on first attack

[62]:
```
# Cell:
# Compute nodes_df by combining entities in attackerIP and victimIP
# As part of this, compute attack counts for each node

targets_df = (
    df
    [['victimIP']]
    .drop_duplicates()
    .rename(columns={'victimIP': 'node_id'})
    .assign(type='victim')
```

(continues on next page)

(continued from previous page)

```
)

attackers_df = (
    df
    .groupby(['attackerIP'])
    .agg(attacks=pd.NamedAgg(column="attackerIP", aggfunc="count"))
    .reset_index()
    .rename(columns={'attackerIP': 'node_id'}).assign(type='attacker')
)

nodes_df = pd.concat([targets_df, attackers_df])

nodes_df.sort_values(by='attacks', ascending=False)[:5]
```

[62]:

	node_id	type	attacks
31	125.64.35.67	attacker	6.0
32	125.64.35.68	attacker	4.0
95	198.204.253.101	attacker	2.0
78	188.225.73.153	attacker	2.0
79	188.44.107.239	attacker	2.0

[86]:

```
# Cell:
# Add


# New encodings features requires api=3: `graphistry.register(api=3, username='...',␣
↪password='...')

g2 = (g
      .nodes(nodes_df, 'node_id')

      # 'red', '#f00', '#ff0000'
      .encode_point_color('type', categorical_mapping={
          'attacker': 'red',
          'victim': 'white'
      }, default_mapping='gray')

      # Icons: https://fontawesome.com/v4.7/cheatsheet/
      .encode_point_icon('type', categorical_mapping={
          'attacker': 'bomb',
          'victim': 'laptop'
      })

      # Gradient
      .encode_edge_color('time(min)', palette=['blue', 'purple', 'red'], as_
↪continuous=True)

      .encode_point_size('attacks')

      .addStyle(bg={'color': '#eee'}, page={'title': 'My Graph'})

      # Options: https://hub.graphistry.com/docs/api/1/rest/url/
```

(continues on next page)

(continued from previous page)

```
      .settings(url_params={'play': 1000, 'pointSize': 0.5})
)

g2.plot(as_files=False)
```

[86]:
```
<IPython.core.display.HTML object>
```

Advanced bindings work with hypergraphs too

Hypergraphs precompute a lot of values on nodes and edges, which we can use to drive clearer visualizations

[104]:
```
hg2_g._nodes.sample(3)
```

[104]:
```
        attackerIP      nodeTitle        type category             nodeID  \
159    77.52.11.94    77.52.11.94  attackerIP       ip    ip::77.52.11.94
162  78.187.242.78  78.187.242.78  attackerIP       ip  ip::78.187.242.78
170  81.47.128.144  81.47.128.144  attackerIP       ip  ip::81.47.128.144

    victimIP  victimPort vulnName EventID
159      NaN         NaN      NaN     NaN
162      NaN         NaN      NaN     NaN
170      NaN         NaN      NaN     NaN
```

[103]:
```
hg2_g._edges.sample(3)
```

[103]:
```
         edgeType              category          vulnName  \
516  ip::vulnName  attackerIP::vulnName  MS08067 (NetAPI)
932  vulnName::ip    vulnName::victimIP  MS08067 (NetAPI)
983  vulnName::ip    vulnName::victimIP  MS08067 (NetAPI)

                           dst     time(max)     time(min)  \
516  vulnName::MS08067 (NetAPI)  1.416885e+09  1.416881e+09
932            ip::172.31.14.66  1.423515e+09  1.423515e+09
983            ip::172.31.14.66  1.423932e+09  1.423932e+09

                           src  victimPort      victimIP       EventID  \
516           ip::186.149.87.94       445.0  172.31.14.66   EventID::76
932  vulnName::MS08067 (NetAPI)       445.0  172.31.14.66   EventID::52
983  vulnName::MS08067 (NetAPI)       445.0  172.31.14.66  EventID::103

     count       attackerIP
516      3    186.149.87.94
932      1    176.119.227.9
983      2  192.110.160.227
```

[113]:
```
(hg2_g

 .encode_point_color('type', categorical_mapping={
     'attackerIP': 'yellow',
     'victimIP': 'blue'
 }, default_mapping='gray')
```

(continues on next page)

(continued from previous page)

```
 .encode_point_icon('type', categorical_mapping={
      'attackerIP': 'bomb',
      'victimIP': 'laptop'
 }, default_mapping='')

 .encode_edge_color('time(min)', palette=['blue', 'purple', 'red'], as_continuous=True)

 .settings(url_params={'pointsOfInterestMax': 10})

).plot()
```

[113]: <IPython.core.display.HTML object>

Further reading:

- https://github.com/graphistry/pygraphistry
- https://github.com/graphistry/pygraphistry/demos/demos_databases_apis
- https://github.com/graphistry/graph-app-kit
- https://hub.graphistry.com/docs/ui/index/
- https://github.com/graphistry/pygraphistry/demos/upload_csv_miniapp.ipynb

10.8.1.2 Tutorial: Graphistry for Developers

Start by generating interactive graphs in thehttps://github.com/graphistry/pygraphistry/demos/for_analysis.ipynb

Graphistry is a client/server system:

- Graphs are akin to live documents: they are created on the server (a `dataset`), and then users can interact with them
- Uploads may provide some settings
- Users may dynamically create settings, such as filters: these are `workbooks`. Multiple `workbooks` may reuse the same `dataset`

APIs:

- Backend APIs
  - https://github.com/graphistry/pygraphistry
  - https://hub.graphistry.com/docs/api/
  - https://graphistry.github.io/graphistry-js/node-tsdocs/
- Frontend APIs
  - iframe
  - React
  - JavaScript

[4]:
```
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

1. Backend APIs

Graphistry provides a REST upload API, and you can reuse the Python client for more conveniently using it.

Python

- Use the PyGraphistry API as in the https://github.com/graphistry/pygraphistry/demos/for_analysis.ipynb
- Instead of plotting, get the plot URL for embedding

[34]:
```
edges = [{'src': 0, 'dst': 1}, {'src': 1, 'dst': 0}]

g = graphistry.edges(pd.DataFrame(edges)).bind(source='src', destination='dst').
↪settings(url_params={'play': 1000})

url = g.plot(render=False)

url
```

[34]:
```
u'https://hub.graphistry.com/graph/graph.html?dataset=PyGraphistry/D1I2OZFIZR&
↪type=vgraph&viztoken=1b70c86e6b6f357d435dce80bf37c42ce284c6ae&usertag=86f11264-
↪pygraphistry-0.9.63&splashAfter=1554622313&info=true&play=1000'
```

REST

- Sample CURL below
- Get API key either from your profile page, or for admins, by https://github.com/graphistry/graphistry-cli

[1]:
```
json_data = {
    "name": "myUniqueGraphName",
    "type": "edgelist",
    "bindings": {
        "sourceField": "src",
        "destinationField": "dst",
        "idField": "node"
    },
    "graph": [
      {"src": "myNode1", "dst": "myNode2",
       "myEdgeField1": "I'm an edge!", "myCount": 7},
      {"src": "myNode2", "dst": "myNode3",
        "myEdgeField1": "I'm also an edge!", "myCount": 200}
```

(continues on next page)

(continued from previous page)

```
    ],
    "labels": [
      {"node": "myNode1",
       "myNodeField1": "I'm a node!",
       "pointColor": 5},
      {"node": "myNode2",
       "myNodeField1": "I'm a node too!",
       "pointColor": 4},
      {"node": "myNode3",
       "myNodeField1": "I'm a node three!",
       "pointColor": 4}
    ]
}

import json
with open('./data/samplegraph.json', 'w') as outfile:
    json.dump(json_data, outfile)
```

[2]:
```
! curl -H "Content-type: application/json" -X POST -d @./data/samplegraph.json https://
↪hub.graphistry.com/etl?key=YOUR_API_KEY_HERE
```

```
{"success":true,"dataset":"myUniqueGraphName"}
```

NodeJS

See [@graphistry/node-api](https://graphistry.github.io/graphistry-js/node-tsdocs/) docs

2. Frontend APIs

Graphistry supports 3 frontend APIs: iframe, React, and JavaScript

iframe

[27]:
```
from IPython.display import HTML, display

#skip splash screen
url = url.replace('splashAfter', 'zzz')

display(HTML('<iframe src="' + url + '" style="width: 100%; height: 400px"></iframe>'))
```

```
<IPython.core.display.HTML object>
```

JavaScript - Browser vanilla JS

- https://www.npmjs.com/package/@graphistry/client-api
- `npm install --save "@graphistry/client-api"`
- See https://hub.graphistry.com/static/js-docs/examples/toggles.html]
- Try with https://npm.runkit.com/@graphistry/client-api

The JavaScript API uses RxJS Observables. Icons are via https://fontawesome.com/v4.7.0/icons/.

```
var pointIconEncoding = {
    attribute: 'type',
    mapping: {
       categorical: {
           fixed: {
               'Page': 'file-text-o',
               'Phone': 'mobile',
               'Email': 'envelope-o',
               'Instagram': 'instagram',
               'Snapchat': 'snapchat',
               'Twitter': 'twitter',
               'Location': 'map-marker',
           }
       }
    }
};

var pointColorEncoding = {
    attribute: 'type',
    mapping: {
       categorical: {
           fixed: {
               'Page': '#777777',
               'Phone': '#f8999e',
               'Email': '#c05c61',
               'Username': '#00AA00',
               'Instagram': '#a43cb1',
               'Snapchat': '#fffb2e',
               'Twitter': '#46a6e4',
               'Location': '#9c4e00',
               'Facebook': '#3f5692',
               'Telegram': '#37aee2'
           },
          'other': '#cccccc'
       }
    }
};

document.addEventListener("DOMContentLoaded", function () {

    GraphistryJS(document.getElementById('viz'))
        .flatMap(function (g) {
            window.g = g;
```

(continues on next page)

(continued from previous page)

```
            g.updateSetting('pointSize', 3);
            g.encodeIcons('point', 'type', pointIconEncoding.mapping);
            return g.encodeColor('point', 'type', 'categorical', pointColorEncoding.
↪mapping);
        })
        .subscribe(function (result) {
            console.log('all columns: ', result);
        }, function(result) {
            console.log('error', result);
        });
});
```

JavaScript React

The React API wraps the JavaScript API.

- https://www.npmjs.com/package/@graphistry/client-api-react
- `npm install --save "@graphistry/client-api-react"`
- See https://hub.graphistry.com/static/js-docs/index.html
- See https://github.com/graphistry/graphistry-js/tree/master/projects/cra-test-18
- See https://github.com/graphistry/graphistry-js/blob/master/projects/client-api-react/src/bindings.js

```
import { Graphistry } from '@graphistry/client-api-react';

<Graphistry
    key={react_nonce}
    className={'my_class'}
    vizClassName={'my_class_2'}

    defaultPointsOfInterestMax={20}
    defaultPruneOrphans={true}
    defaultShowPointsOfInterest={true}
    defaultShowPointsOfInterestLabel={false}
    backgroundColor={'#EEEEEE'}
    defaultPointSize={1}
    defaultDissuadeHubs={true}
    defaultGravity={8}
    defaultScalingRatio={12}
    defaultEdgeOpacity={0.5}
    defaultShowArrows={false}
    defaultEdgeCurvature={0.02}
    defaultShowLabelPropertiesOnHover={true}
    ...
    play={2000}
    showIcons={true}
    defaultShowToolbar={true}
    axes={axes}
    controls={controls}
    type={datasetType}
```

(continues on next page)

(continued from previous page)

```
    dataset={datasetName}
    graphistryHost={'http://...'}
    loadingMessage={'loading..'}
    showLoadingIndicator={true}
/>
```

[ ]:

10.8.1.3 Visualize CSV Mini-App

- Jupyter: `File` -> `Make a copy` Colab: `File` -> `Save a copy in Drive`
- Run notebook cells by pressing `shift-enter`
- Either edit annd run top cells one-by-one, or edit and run the self-contained version at the bottom

[1]:
```
#!pip install graphistry -q
```

[3]:
```
import pandas as pd
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

1. Upload csv

Use a file by uploading it or via URL.

Run `help(pd.read_csv)` for more options.

File Upload: Jupyter Notebooks

- If circle on top right not green, click `kernel` -> `reconnect`
- Go to file directory (`/tree`) by clicking the Jupyter logo
- Navigate to the directory page containing your notebook
- Press the `upload` button on the top right

File Upload: Google Colab

- Open the left sidebar by pressing the right arrow on the left
- Go to the `Files` tab
- Press `UPLOAD`
- Make sure goes into `/content`

File Upload: URL

- Uncomment below line and put in the actual data url
- Run `help(pd.read_csv)` for more options

[4]:
```
file_path = './data/honeypot.csv'
df = pd.read_csv(file_path)

print('# rows', len(df))
df.sample(min(len(df), 3))
```

```
('# rows', 220)
```

[4]:
```
          attackerIP      victimIP  victimPort          vulnName  count  \
145   41.230.211.128  172.31.14.66       445.0  MS08067 (NetAPI)      2
25   122.121.202.157  172.31.14.66       445.0  MS08067 (NetAPI)      8
75    182.68.160.230  172.31.14.66       445.0  MS08067 (NetAPI)      9

        time(max)     time(min)
145  1.421730e+09  1.421729e+09
25   1.423612e+09  1.423611e+09
75   1.417438e+09  1.417436e+09
```

2. Optional: Clean up CSV

[5]:
```
df = df.rename(columns={
#    'attackerIP': 'src_ip',
#    'victimIP': 'dest_ip'
})

df.sample(3)
```

[5]:
```
         attackerIP      victimIP  victimPort          vulnName  count  \
70   182.161.224.84  172.31.14.66       139.0  MS08067 (NetAPI)      4
10   115.115.227.82  172.31.14.66       445.0  MS08067 (NetAPI)      2
152    46.130.76.13  172.31.14.66       445.0  MS08067 (NetAPI)      7

        time(max)     time(min)
70   1.419954e+09  1.419952e+09
10   1.413569e+09  1.413569e+09
152  1.421093e+09  1.421092e+09
```

3. Configure: Visualize with 3 kinds of graphs

Set `mode` and the corresponding values:

Mode "A". See graph from table of (src,dst) edges

Mode "B". See hypergraph: Draw row as node and connect it to entities in same row

- Pick which cols to make nodes
- If multiple cols share same type (e.g., "src_ip", "dest_ip" are both "ip"), unify them

Mode "C". See by creating multiple nodes, edges per row

- Pick how different column values point to other column values
- If multiple cols share same type (e.g., "src_ip", "dest_ip" are both "ip"), unify them

[6]:
```
#Pick 'A', 'B', or 'C'
mode = 'B'
max_rows = 1000


### 'A' == mode
my_src_col = 'attackerIP'
my_dest_col = 'victimIP'



### 'B' == mode
node_cols = ['attackerIP', 'victimIP', 'vulnName']
categories = { #optional
    'ip': ['attacker_IP', 'victimIP']
    #, 'user': ['owner', 'seller'],
}



### 'C' == mode
edges = {
      'attackerIP': [ 'victimIP', 'victimPort', 'vulnName'],
      'victimIP': [ 'victimPort'],
      'vulnName': [ 'victimIP' ]
}
categories = { #optional
      'ip': ['attackerIP', 'victimIP']
       #, user': ['owner', 'seller'], ...
}
```

4. Plot: Upload & render!

- See https://hub.graphistry.com/docs/ui/index/

[75]:
```
g = None
hg = None
num_rows = min(max_rows, len(df))
if mode == 'A':
    g = graphistry.edges(df.sample(num_rows)).bind(source=my_src_col, destination=my_
↪dest_col)
elif mode == 'B':
    hg = graphistry.hypergraph(df.sample(num_rows), node_cols, opts={'CATEGORIES':␣
↪categories})
    g = hg['graph']
elif mode == 'C':
    nodes = list(edges.keys())
    for dests in edges.values():
        for dest in dests:
            nodes.append(dest)
    node_cols = list(set(nodes))
    hg = graphistry.hypergraph(df.sample(num_rows), node_cols, direct=True, opts={
↪'CATEGORIES': categories, 'EDGES': edges})
    g = hg['graph']

#hg
print(len(g._edges))

g.plot()
```

```
('# links', 1100)
('# events', 220)
('# attrib entities', 221)
1100
```

[75]:
```
<IPython.core.display.HTML object>
```

Alternative: Combined

Split into data loading and cleaning/configuring/plotting.

[59]:
```
#!pip install graphistry -q
import pandas as pd
import graphistry
#graphistry.register(key='MY_KEY', server='hub.graphistry.com')


##########
#1. Load
file_path = './data/honeypot.csv'
df = pd.read_csv(file_path)

print(df.columns)
```

(continues on next page)

(continued from previous page)

```
print('rows:', len(df))
print(df.sample(min(len(df),3)))
```

```
Index([u'attackerIP', u'victimIP', u'victimPort', u'vulnName', u'count',
       u'time(max)', u'time(min)'],
      dtype='object')
('rows:', 220)
         attackerIP      victimIP  victimPort             vulnName  count  \
81  187.143.247.231  172.31.14.66       445.0      MS04011 (LSASS)      1
47   151.252.204.92  172.31.14.66       139.0     MS08067 (NetAPI)      1
41     125.64.35.68  172.31.14.66      9999.0  MaxDB Vulnerability      6

       time(max)     time(min)
81  1.420657e+09  1.420657e+09
47  1.422929e+09  1.422929e+09
41  1.420915e+09  1.417479e+09
```

[79]:
```
##########
#2. Clean
#df = df.rename(columns={'attackerIP': 'src_ip', 'victimIP: 'dest_ip', 'victimPort':
↪'protocol'})


##########
#3. Config - Pick 'A', 'B', or 'C'
mode = 'C'
max_rows = 1000


### 'A' == mode
my_src_col = 'attackerIP'
my_dest_col = 'victimIP'

### 'B' == mode
node_cols = ['attackerIP', 'victimIP', 'victimPort', 'vulnName']
categories = { #optional
    'ip': ['src_ip', 'dest_ip']
    #, 'user': ['owner', 'seller'],
}

### 'C' == mode
edges = {
    'attackerIP': [ 'victimIP', 'victimPort', 'vulnName'],
    'victimIP': [ 'victimPort' ],
    'vulnName': ['victimIP' ]
}
categories = { #optional
    'ip': ['attackerIP', 'victimIP']
    #, 'user': ['owner', 'seller'], ...
}

##########
```

(continues on next page)

(continued from previous page)

```
#4. Plot
g = None
hg = None
num_rows = min(max_rows, len(df))
if mode == 'A':
    g = graphistry.edges(df.sample(num_rows)).bind(source=my_src_col, destination=my_
↪dest_col)
elif mode == 'B':
    hg = graphistry.hypergraph(df.sample(num_rows), node_cols, opts={'CATEGORIES':␣
↪categories})
    g = hg['graph']
elif mode == 'C':
    nodes = list(edges.keys())
    for dests in edges.values():
        for dest in dests:
            nodes.append(dest)
    node_cols = list(set(nodes))
    hg = graphistry.hypergraph(df.sample(num_rows), node_cols, direct=True, opts={
↪'CATEGORIES': categories, 'EDGES': edges})
    g = hg['graph']


g.plot()
```

```
('# links', 1100)
('# events', 220)
('# attrib entities', 221)
```

[79]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
```

10.8.1.4 Visually analyze any table as a graph: Our three favorite Graphistry shapings

Our 3 favorite ways to shape tables into a graph and then visualize it, each with just one line of code!

1. **Simple property graphs:** Edge tables
2. **Advanced property graphs:** Hypergraphs for more control
3. **AI - UMAP**: Automatically link entities with similar properties

Install

Local pip install to run the shaping and analytics locally, CPU or GPU

For the GPU cloud visualization sessons, and GPU analytics offloading of bigger graphs, get a free username/password or api key at https://hub.graphistry.com (external link)

[ ]:
```
! pip install -q graphistry[umap_learn]
```

[15]:
```
import graphistry
print(graphistry.__version__)

# Make API key at https://hub.graphistry.com/users/personal/key/ (create free account␣
↪first)
graphistry.register(api=3, personal_key_id=FILL_ME_IN, personal_key_secret=FILL_ME_IN)
```

```
0.35.4+66.g9a3a886
```

Data

Sample logs

CPU mode is great for < 10K rows, and consider GPU and AI modes for 10K-1B rows

[16]:
```
import pandas as pd
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/refs/heads/
↪master/demos/data/honeypot.csv')
df
```

[16]:
```
          attackerIP      victimIP  victimPort          vulnName  count  \
0       1.235.32.141  172.31.14.66       139.0  MS08067 (NetAPI)      6
1     105.157.235.22  172.31.14.66       445.0  MS08067 (NetAPI)      4
2    105.186.127.152  172.31.14.66       445.0   MS04011 (LSASS)      1
3      105.227.98.90  172.31.14.66       445.0  MS08067 (NetAPI)      7
4     105.235.44.218  172.31.14.66       445.0  MS08067 (NetAPI)      4
..               ...           ...         ...               ...    ...
215    94.153.13.180  172.31.14.66       445.0  MS08067 (NetAPI)      1
216     94.243.32.41  172.31.14.66       445.0  MS08067 (NetAPI)     10
217    95.234.253.23  172.31.14.66       445.0  MS08067 (NetAPI)      2
218    95.68.116.216  172.31.14.66       445.0  MS08067 (NetAPI)     20
219    95.74.232.188  172.31.14.66       445.0  MS08067 (NetAPI)      6

        time(max)     time(min)
0    1.421434e+09  1.421423e+09
1    1.422498e+09  1.422495e+09
2    1.419966e+09  1.419966e+09
3    1.421742e+09  1.421740e+09
4    1.416686e+09  1.416684e+09
..            ...           ...
215  1.423904e+09  1.423904e+09
216  1.412510e+09  1.412508e+09
217  1.421355e+09  1.421354e+09
218  1.420813e+09  1.414762e+09
219  1.418149e+09  1.418148e+09

[220 rows x 7 columns]
```

1. Simple property graph: Edge tables with attributes

Each table row represents an edge with properties: * One column to use as the edge source * One column as the edge destination * Remaining as edge attributes

Optionally add a nodes table by chaining `.nodes(nodes_df, 'my_id_column')`

[21]:
```
g1 = graphistry.edges(df, source='attackerIP', destination='victimIP')
g1.plot()
```

[21]:
```
<IPython.core.display.HTML object>
```

2. Advanced property graphs: Hypergraphs for more control

We commonly want a table row to yield multiple edges between multiple columns, not just a src/dst column pair

The first version simply links entities 3 columns to one another, so each table row forms a triangle:

[18]:
```
g2 = graphistry.hypergraph(df, ['attackerIP', 'victimIP', 'vulnName'], direct=True)[
↪'graph']
g2.plot()
```

```
# links 660
# events 220
# attrib entities 212
```

[18]:
```
<IPython.core.display.HTML object>
```

We can control many aspects. In this case:

- Causally directed edges: attackerIP->victimIP, attackerIP->vulnName, vulnName->attackerIP
- Combine name spaces: When an IP appears both as a victimIP and attackerIP, collapse into one node, vs treating those columns as distinct node ID namespaces

[6]:
```
g2b = graphistry.hypergraph(df, ['attackerIP', 'victimIP', 'vulnName'], direct=True,␣
↪opts={
    'EDGES': {
        'attackerIP': ['victimIP', 'vulnName'],
        'vulnName': ['victimIP']
    },
    'CATEGORIES': {
        'ip': ['attackerIP', 'victimIP']
    }
})['graph']
g2b = g2b.encode_point_color('category', categorical_mapping={'ip': 'grey', 'vulnName':
↪'orange'}, as_categorical=True)
g2b.plot()
```

```
# links 660
# events 220
# attrib entities 212
```

[6]:
```
<IPython.core.display.HTML object>
```

3. AI - UMAP: Automatically link entities with similar properties

[19]:
```
#g3.reset_caches()
g3 = graphistry.nodes(df).umap(X=['attackerIP',     'victimIP',     'victimPort',
↪'vulnName',     'count',        'time(max)',    'time(min)'])
g3.plot()
```

[19]:
```
<IPython.core.display.HTML object>
```

Next steps

- Learn:
  - https://github.com/graphistry/pygraphistry:
    - https://pygraphistry.readthedocs.io/en/latest/10min.html
    - https://pygraphistry.readthedocs.io/en/latest/gfql/about.html
    - https://pygraphistry.readthedocs.io/en/latest/demos/talks/infosec_jupyterthon2022/rgcn_login_anomaly_det story.html
  - https://hub.graphistry.com/docs/
- Try:
  - https://pypi.org/project/graphistry/ the pygraphistry client
  - https://www.graphistry.com/get-started a free Graphistry Hub GPU account and even https://www.graphistry.com/get-started
  - ... Then login and https://www.graphistry.com/blog/no-code-file-uploader-hypergraph!
- Explore more of the Graphistry ecosystem:
  - https://www.graphistry.com/get-started
  - https://louie.ai/: GenAI-first notebooks, dashboards, and pipelines, including for working with Graphistry
  - Dashboards: Use in Snowflake's https://github.com/graphistry/graph-app-kit, https://github.com/graphistry/pygraphistry/tree/master/demos/demos_databases_apis/databricks_pyspark, https://www.graphistry.com/graphistry-for-powerbi, and more
  - https://pygraphistry.readthedocs.io/en/latest/gfql/about.html, our new open source system, including optional GPU acceleration and ability to switch between local & remote execution

10.8.2 Visualization

10.8.2.1 Encodings

Color encodings tutorial

See the examples below for common ways to map data to node/edge color in Graphistry.

Colors are often used with node size, icon, label, and badges to provide more visual information. Most encodings work both for points and edges. The https://github.com/graphistry/pygraphistry makes it easier to use the https://hub.graphistry.com/docs/api/1/rest/url/ and the

https://hub.graphistry.com/docs/api/2/rest/upload/. For dynamic control, you can use also use the https://hub.graphistry.com/docs/.

Setup

Mode `api=3` is used. It supports both `complex_encodings` (ex: `.encode_point_color(...)`) and the simpler `.bind(point_color='col_a')` form.

[ ]:
```
# ! pip install --user graphistry
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

graphistry.__version__
```

[ ]:
```
import datetime, pandas as pd
e_df = pd.DataFrame({
    's': ['a', 'b', 'c', 'a', 'd', 'e'],
    'd': ['b', 'c', 'a', 'c', 'e', 'd'],
    'time': [datetime.datetime(1987, 10, 1), datetime.datetime(1987, 10, 2), datetime.
↪datetime(1987, 10, 3),
             datetime.datetime(1988, 10, 1), datetime.datetime(1988, 10, 2), datetime.
↪datetime(1988, 10, 3)]
})
n_df = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'score': [ 0, 30, 50, 70, 90 ],
    'palette_color_int32': pd.Series(
        [0, 1, 2, 3, 4],
        dtype='int32'),
    'hex_color_int64': pd.Series(
        [0xFF000000, 0xFFFF0000, 0xFFFFFF00, 0x00FF0000, 0x0000FF00],
        dtype='int64'),
    'type': ['mac', 'macbook', 'mac', 'macbook', 'sheep']
})
g = graphistry.edges(e_df, 's', 'd').nodes(n_df, 'n')
```

Colors

Options: default, built-in palette, RGBA, continuous palette, and categorical mapping

Applies to both nodes and edges

- Use the `.encode_point_color()` and `.encode_edge_color()` calls
- For palette and RGBA bindings (non-complex), you can also use the shorthand `.bind(point_color='col_a', edge_color='col_b')`.

Default

- Node: Graphistry looks at the local graph structure to auto-color nodes
- Edges: Gradient from the src/dst node color to reinforce the node color decision

[ ]:
```
g.plot()
```

Built-in palette

Bind an int32 column where values are intepreted by the https://hub.graphistry.com/docs/api/api-color-palettes/

[ ]:
```
print(g._nodes['palette_color_int32'].dtype)
g.encode_point_color('palette_color_int32').plot()
```

RGBA colors

[ ]:
```
print(g._nodes['hex_color_int64'].dtype)
g.encode_point_color('hex_color_int64').plot()
```

Continuous colors

Create a gradient effect by linearly mapping the input column to an evenly-spaced palette.

Great for tasks like mapping timestamps, counts, and scores to low/high and low/medium/high intervals.

[ ]:
```
g.encode_point_color('score', palette=['silver', 'maroon', '#FF99FF'], as_
↪continuous=True).plot()
```

Categorical colors

Map distinct values to specific colors. Optionally, set a default, else black.

[ ]:
```
g.encode_point_color(
    'type',
    categorical_mapping={
        'mac': '#F99',
        'macbook': '#99F'
    },
    default_mapping='silver'
).plot()
```

Edge colors

Edge colors work the same as node colors by switching to call `.encode_edge_color()`:

[ ]:
```
g.encode_edge_color('time', palette=['blue', 'red'], as_continuous=True).plot()
```

Legend support

Categorical node colors will appear in legend when driven by column `type`:

[ ]:
```
g.encode_point_color(
    'type',
    categorical_mapping={
        'mac': '#F99',
        'macbook': '#99F'
    },
    default_mapping='silver'
).plot()
```

[ ]:
```

```

Size encodings tutorial

See the examples below for common ways to map data to node size in Graphistry.

Size refers to point radius. This tutorial covers two kinds of size controls:

- Node size setting, which is a global scaling factor
- Node size encoding, used for mapping a node data column to size

Sizes are often used with node color, label, icon, and badges to provide more visual information. Most encodings work both for points and edges. The https://github.com/graphistry/pygraphistry makes it easier to use the https://hub.graphistry.com/docs/api/1/rest/url/ and the https://hub.graphistry.com/docs/api/2/rest/upload/. For dynamic control, you can use also use the https://hub.graphistry.com/docs/.

Setup

Mode `api=3` is used. It supports both `complex_encodings` (ex: `.encode_point_size(...)`) and the simpler `.bind(point_size='col_a')` form.

[ ]:
```
# ! pip install --user graphistry
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

graphistry.__version__
```

[ ]:
```
import datetime, pandas as pd
e_df = pd.DataFrame({
    's': ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'd', 'e'],
    'd': ['b', 'c', 'a', 'b', 'c', 'a', 'c', 'e', 'd'],
    'time': [datetime.datetime(1987, 10, 1), datetime.datetime(1987, 10, 2), datetime.
↪datetime(1987, 10, 3),
             datetime.datetime(1988, 10, 1), datetime.datetime(1988, 10, 2), datetime.
↪datetime(1988, 10, 3),
             datetime.datetime(1989, 10, 1), datetime.datetime(1989, 10, 2), datetime.
↪datetime(1989, 10, 3)]
})
n_df = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'score': [ 1, 30, 50, 70, 90 ],
    'palette_color_int32': pd.Series(
        [0, 1, 2, 3, 4],
        dtype='int32'),
    'hex_color_int64': pd.Series(
        [0xFF000000, 0xFFFF0000, 0xFFFFFF00, 0x00FF0000, 0x0000FF00],
        dtype='int64'),
    'type': ['mac', 'macbook', 'mac', 'macbook', 'sheep']
})

g = graphistry.edges(e_df, 's', 'd').nodes(n_df, 'n')
```

Default Size

Graphistry uses the 'degree' of a node, so nodes with more edges appear bigger

[ ]:
```
g.plot()
```

Size Setting

You can tune the scaling factor:

[ ]:
```
g.settings(url_params={'pointSize': 0.5}).plot()
```

Size encodings

Options: continuous mapping, categorical mapping

Continuous size encodings

Use an input column as relative sizes. Graphistry automatically normalizes them.

[ ]:
```
g.settings(url_params={'pointSize': 0.3}).encode_point_size('score').plot()
```

Categorical size encodings

Map distinct values to specific sizes. Optionally, set a default, else black.

[ ]:
```
g.settings(url_params={'pointSize': 0.3})\
 .encode_point_size(
    'type',
    categorical_mapping={
        'mac': 50,
        'macbook': 100
    },
    default_mapping=20
).plot()
```

Legend support

Categorical node sizes will appear in legend when driven by column `type`:

[ ]:
```
g.settings(url_params={'pointSize': 0.3})\
 .encode_point_size(
    'type',
    categorical_mapping={
        'mac': 50,
        'macbook': 100
    },
    default_mapping=20
).plot()
```

[ ]:
```

```

Icons encodings tutorial

See the examples below for common ways to map data to node icon in Graphistry.

You can add a main icon. The glyph system supports text, icons, flags, and images, as well as multiple mapping and style controls. When used with column `type`, the icon will also appear in the legend.

Icons are often used with node color, label, size, and badges to provide more visual information. Most encodings work both for points and edges. The https://github.com/graphistry/pygraphistry makes it easier to use the https://hub.graphistry.com/docs/api/1/rest/url/ and the https://hub.graphistry.com/docs/api/2/rest/upload/. For dynamic control, you can use also use the https://hub.graphistry.com/docs/.

Setup

Mode `api=3` is used. It supports both `complex_encodings` (ex: `.encode_point_icon(...)`) and the simpler `.bind(point_icon='col_a')` form.

[ ]:
```
# ! pip install --user graphistry
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

graphistry.__version__
```

[ ]:
```
import datetime, pandas as pd
e_df = pd.DataFrame({
    's': ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'd', 'e'],
    'd': ['b', 'c', 'a', 'b', 'c', 'a', 'c', 'e', 'd'],
    'time': [datetime.datetime(1987, 10, 1), datetime.datetime(1987, 10, 2), datetime.
↪datetime(1987, 10, 3),
             datetime.datetime(1988, 10, 1), datetime.datetime(1988, 10, 2), datetime.
↪datetime(1988, 10, 3),
             datetime.datetime(1989, 10, 1), datetime.datetime(1989, 10, 2), datetime.
↪datetime(1989, 10, 3)]
})
n_df = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'score': [ 1, 30, 50, 70, 90 ],
    'palette_color_int32': pd.Series(
        [0, 1, 2, 3, 4],
        dtype='int32'),
    'hex_color_int64': pd.Series(
        [0xFF000000, 0xFFFF0000, 0xFFFFFF00, 0x00FF0000, 0x0000FF00],
        dtype='int64'),
    'type': ['mac', 'macbook', 'mac', 'macbook', 'sheep'],
    'assorted': ['Canada', 'mac', 'macbook', 'embedded_smile', 'external_logo'],
    'origin': ['Canada', 'England', 'Russia', 'Mexico', 'China']

})

g = graphistry.edges(e_df, 's', 'd').nodes(n_df, 'n')
```

Icons as categorical mappings + glyph types

The most common form is mapping distinct values to icons.

Graphistry supports built-in + custom glyphs:

- Built-in general glyphs: Use values from https://fontawesome.com/v4.7.0/ or, more explicitly, `fa-thename`
- Built-in flag icons: Use ISO3611-Alpha-2 values
- Custom image URL
- Custom image data URI (embedded)

[ ]:
```
g.encode_point_icon(
  'assorted',
  shape="circle", #clip excess
  categorical_mapping={
      'macbook': 'laptop', #https://fontawesome.com/v4.7.0/icons/
      'Canada': 'flag-icon-ca', #ISO3611-Alpha-2: https://github.com/datasets/country-
↪codes/blob/master/data/country-codes.csv
      'embedded_smile': '
↪iVBORw0KGgoAAAANSUhEUgAAARkAAAC0CAMAAACXO6ihAAAAgVBMVEUAAAD////T09P4+PiYmJihoaHf39/7+/
↪vm5uYLCwsRERHy8vJ+fn7v7+83Nze2trbZ2dlRUVFGRka9vb1xcXEYGBhbW1tWVlbq6uq7u7usrKxgYGA8PDwWFhbNzc14eHhpaWm
↪AS9L06ZlkZm5I37i+acUSA9dkjRNJal/
↪JPNYjfxRBlvW1guzb3leBPupJo8wLGfat0wvgaOrjEQo0b1vsfrGTKvSUiLtW7RekaRNvGRQV32L1x9it5mX0Wh86lu+3uCM24jJEI
↪Wwud3WUy3ihpq+pewBB9yTnPWMXjivfURYjxL2hIDV3j+KFx3Wwwan8TFixl5VpdtvswuWHEXGKOhBuD4xZS0mntRcPxvRYT1Lkvt
↪eIEVW4mRuIMhKH0pztM2MvmMjtMzEh+nnC9QiP1jb6ayzgcM0PpTsSOVA4tZfQhMgNmQWsX4dwTI/
↪1ZsvUL8G7GbYV8jplhuGjupNJKayl+7en8JNn6hUpq67WWWmJitk8SrWfArPNAR0Hdybg9R7KecQYVbtFebsuI2TxHsr5xh1mnvSE
↪SbDesezCDLYN5k8TrWdQZprNyT0mRm/
↪tde8E2psaF9hmnJbX4vR7M2zo2NrQnea8MTmYARi1iKj2uhBMMx7QAiVbmnSqHSXmXHkZjAEFpF1ZtR3hUlgNpqlvWO+JBLle3DW6
↪e2GVyjibe4b8n/GBVHja9H14Wgxu3n6dbV+SQA2/
↪deutxvBVq2aksU0QkpO9Z7O272HsdLdHigvSGXznuPMgmek6wILXOb12S2u1VtatAFlfh5UvaAL0QMzhoyWV4D2bdqx1g1shU7Ct7
↪xzqBbnEa2EHwFoUeVve3DwJkGQlfmX2hMw1rRpmCjb0WBIZxZLdvC3hg0WK9mmCVX3PdWchuwAp1NqTaMBTDTg1z9YwomUI0rAaIB
↪qYCaS1bNRVi3fW/
↪DqAFzaDJx9doKvBLPFuolcABdrmYXytJqbk7vD+gwdRrLEfTip0v1CiCVrwv1ncGsNZzlJASTVL5uVzIoOv7u6WK9AEJS+7otpmCAI
↪XwTN7QLMKXlwoQ4FnMY2swQf+EDtbND/K1sK6MQ0TGWG7RfURbcl3ZAwqOAYhITUfywkjKZB98ZgI/
↪GAAZdbA6AG+JsvJrWBLmVvGTUTlRIzSP9DCZax04XmsWZrLNaAI4MXLL5BMZw4uasyS3YwTMcMoGOXz7Dz+M/
↪qg2by+Sq49C1cv9g0MeMMnBghWRNrMQOeligOdpUYf7iKDMbGE4lxBpK16SH2sY7Cy8bRepCLkg0w1Uh2DUOPInWgScfbcFkshpLi
↪wWTF0FrwHERe/
↪ByZXgNbxQa07vc6Fou90ZunPsDAqVZaikGgljycEJtuzFxo7OdwNnJ3Y+ZOywzlQFMuqUgLMwdaZiCHU2ZII5ZmsLnUENuMeTlPrY
↪lDbIRdmHGTIssI1634HtlQMxIOnyapnE3beY4GhQz3bH2/4iZr83ykGE5b0m2tciKLDeVeMxbXNwZtpwceAmLW/
↪EmUp6ZW/72uGP0XhIWL7zj0WdGdqq2MbMp7ur2DoDmRb4yHo8VxZA9Jy7/3J1UQP7PMtKzIooRBZiCdeDp9ji/
↪1dY9TyV1ozeq+W6LtRPZ5a0aqztm5hS5+dtt2UPxwXP2EBX3p+lWtgtZ9chLIT50Cfsxo2txg5Y9aEZv13KJr5FR3nXMnhayZ5Pax
↪6HAjB3YpV2HAHtvPGkxKPRfJaNF717gtZfi3jlL13z8rakrlFZey4yswdJVpU2L5lljAMn5sR2viN4zJboWFXkg1P8Z6qUcrtCIlM
↪tGpktzHzVvPEq3WqkCHlmQvzxZU4JJ8ygw7fK04C+6s/
↪IUXhmZG4PDmHmLFduy6FxzCjiYSqQzYkyI+5G0JqZ2dclXD6yFFoIc44Zh/
↪s83ZhpODxIYEZIuV8yU3eAzEhkpgr9xjNTefWlkZna87G0tbvKFhwzfGfrxAzK/
↪qfngJx3OTPLmr0l5cWCGfb9jOJWlMhqX+n2mNpTlRnbMFDXbWQGfupemgYukfUo1bRda4cNUgEyxON1nMmZgaxkI8Ur9zLeSH1zZm
↪3hvSqoa/
↪iBupJY0T6e4EZVwtXyXIL7BHz8a4GHrxjxnGllTNAaBBmrk4QwOPdoEA+myWnq4Y+rr0NiBmbvWCtasAnYSZk/
↪+TMgPhjmpVBZcwUwzP9lO5yBc05Z+YMnPmgIEBRMk7MqEhBkVjmwtrAjGcmIolnoHI6NQZWEb4DCthwOSDM5ICxRtBnWEeL5hM4d7
↪C0bPVY4xX66ZGZSmCOppccyw63I3ZmgrXJ7SdLrcdWHGK4emoJC/
↪bGVtzNygM6Gd5A3MkL69uu1myfnC3G0oOchjZo7kL9+sZ8ZrZ4bNQV41JdYDZkj1z4X8q4fMwG0+8n80MCOm1AQ1CLkhHzMD05ny9
```

(continues on next page)

(continued from previous page)

```
↪GTP8vmY3wPt2/
↪19mYKbDSa47MgMNGyUueMzMhPReZfMzZiTh8AwF7Wl+OWbQCNWBGTjZ9qfMzCrnihggwKswQ6eJ7zFj/
↪JIZaVMxtuzwT5lBvrS/ZEaC3nT7KTPSLhUPT9P/ghkYgW3kAezGjPkjZjaQ1HX/Y2YyxWGq85au8wfMUNUZHSB/
↪7MQM9UCgxImPmYHvXShHP2VGyt2w24iZ5tHld8ywI35iG5jZgXBo+oOD0Tsyg5IqPmYGTP8mHbgrMxnOMVVuSgfqz5mJqLIKpzkb0
↪3gT3y/jwO2Ys8MpT+Qx05DWsBl9gNn7EDPjLdDatpcg/U8sM9STFP2NmWrzAoCsXnE71fWZuY3y/
↪hNybGTN3egZQWnS2TQSlHzFzgQ/GVD3hTPUKM3TUHp1/
↪yEzRKHXKDLxwipkRcqa2MJNQD1Ph9dgwp17uuaLCjg1HDVC6MmAmVOqZAVt75MKMP0fezlpmqBOp/
↪KcmL3uVGei0CWKG2ktwgE+MmZH5lUT4uy4zEm0jlh5F1M9LmBH1bYGZVQKi6bNFZqSix1L3s15M+UlIbzUWixliZuQu57fkNqcj5j
↪4x6u+XzjXmhp96Gi90KBrKg/BYJ+AijYuWTvj/
↪YTcxZKBf0r2jCVG+XyS+sjx+azpUVPOQ8Cr9a6jE3sG3rMjb+Z+z8mKPKXKgF/
↪PwYkAn6w5DRqoa3ynh12A3ZkzMLc+NQp29+HPA1q6vj0TZs2caJDvGuli87UtFKfyTVQEa+4+TX8sMdPSrk2PZdWWg4l8oajWIxZq
↪mkMZSyCZ3FaqbUe0ph88xIc6HR2Gn5KUpmpBR1sNHYIaNBwYy4WsGdFSPWjlvUMRuZGRceCIf/
↪s5YZoTlvG5gBLVxcdHKllVjeF0eaBVfCvUkcM9LX1iXc+d4SmoIfPmKm0hhdJFsjM+Vw+UNmasYFdkbWTmjhVmw+ZEa60cVLK8o68
↪gejzMiyKxu5A7QwJU5iO5Xc+HB4qMem9KcFsQjcHSkAhpLmB007slupoXG/D/
↪XPPCD+ydTnFaLqz5mDUGJQi6J2OVAFzrHbnyRJrz0svvwEA6y6kxUlFVyx1YJId7tmrjsNj0etWVN9jekz2w0x3GC6f+TNvqyWJ6O
↪pvIxsDVhZrCSZ/zZenhnSQHDnIHPKoY4Zzhb5Ff4VZlDsIKz6bGqY2YDW+fsDK/4VZtisAiamsaPMGNQBDbP7/
↪3CWXfKPMEPlm4ORlk9hVP93r/E6VJlj/
↪rdHKd3mB1CtDe00fclza8EGVaJ0vY6P1BQoViMqlhGB89scrOIDXzELcFhXcxhbGpjRf33GlPjEVzyBqIGZclauZ8b/
↪fdv/d5nZksqLMdI5L9v/4a3/
↪AjMLt+q2c5nSG6p8qGykrlse1hniG19xnMksMy3CLcOmmwNKmLv5Oj7k0frTeN22H+M72PBYv+LclONrsQnjqVpW/
↪kbV4f8AHefeSC51gZgAAAAASUVORK5CYII=',
      'external_logo': 'https://awsmp-logos.s3.amazonaws.com/4675c3b9-6053-4a8c-8619-
↪6519b83bbbfd/536ec8b5c79de08fcac1086fdf74f91b.png'
  },
  default_mapping="question").plot()
```

Icons as continuous mappings and text

You can also use value ranges to pick the glyph, and use text as the glyph

[ ]:
```
g.encode_point_icon(
   'score',
    as_text=True,
    continuous_binning=[
        [33, 'low'],
        [66, 'mid'],
        [200, 'high']
    ]).plot()
```

Special continuous bins

- For values bigger than the last bin, use `None`
- For nulls, use the default mapping

[ ]:
```
g.encode_point_icon(
   'score',
    as_text=True,
    continuous_binning=[
        [33, 'low'],
        [66, 'mid'],
        [None, 'high']
    ],
    default_mapping='?'
).plot()
```

Flag inference

The below code generates ISO3166 mappings from different conventions to Alpha-2

[ ]:
```
codes = pd.read_csv('https://raw.githubusercontent.com/datasets/country-codes/master/
↪data/country-codes.csv')
codes.columns
```

[ ]:
```
country_to_iso_flag = {
    o['CLDR display name']: 'flag-icon-' + o['ISO3166-1-Alpha-2'].lower()
    for o in codes[['CLDR display name', 'ISO3166-1-Alpha-2']].dropna().to_dict('records
↪')
}

g.encode_point_icon(
  'origin',
  shape="circle",
  categorical_mapping=country_to_iso_flag,
  default_mapping="question").plot()
```

[ ]:
```

```

Badge encodings tutorial

See the examples below for common ways to map data to node badges in Graphistry. Icons appear in the main area of a node, while badges circle them (`TopRight`, `BottomLeft`, `Right`, etc.). They can be used together.

Badges are quite configurable. You can set the glyph, color, border, and define their encoding to be based on the ndoe data.

The https://github.com/graphistry/pygraphistry makes it easier to use the https://hub.graphistry.com/docs/api/1/rest/url/ and the https://hub.graphistry.com/docs/api/2/rest/upload/.

For dynamic control, you can use also use the https://hub.graphistry.com/docs/.

Setup

Mode `api=3` is used. It supports both `complex_encodings` (ex: `.encode_point_badge(...)`) and the simpler `.bind(point_color='col_a')` form.

[ ]:
```
# ! pip install --user graphistry
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

graphistry.__version__
```

[ ]:
```
import datetime, pandas as pd
e_df = pd.DataFrame({
    's': ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'd', 'e'],
    'd': ['b', 'c', 'a', 'b', 'c', 'a', 'c', 'e', 'd'],
    'time': [datetime.datetime(1987, 10, 1), datetime.datetime(1987, 10, 2), datetime.
↪datetime(1987, 10, 3),
             datetime.datetime(1988, 10, 1), datetime.datetime(1988, 10, 2), datetime.
↪datetime(1988, 10, 3),
             datetime.datetime(1989, 10, 1), datetime.datetime(1989, 10, 2), datetime.
↪datetime(1989, 10, 3)]
})
n_df = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'score': [ 1, 30, 50, 70, 90 ],
    'palette_color_int32': pd.Series(
        [0, 1, 2, 3, 4],
        dtype='int32'),
    'hex_color_int64': pd.Series(
        [0xFF000000, 0xFFFF0000, 0xFFFFFF00, 0x00FF0000, 0x0000FF00],
        dtype='int64'),
    'type': ['mac', 'macbook', 'mac', 'macbook', 'sheep'],
    'assorted': ['Canada', 'mac', 'macbook', 'embedded_smile', 'external_logo'],
    'origin': ['Canada', 'England', 'Russia', 'Mexico', 'China']

})

g = graphistry.edges(e_df, 's', 'd').nodes(n_df, 'n')
```

Badges as categorical mappings + glyph types

The most common form is mapping distinct values to glyphs.

Graphistry supports built-in + custom glyphs:

- Built-in general glyphs: Use values from https://fontawesome.com/v4.7.0/ or, more explicitly, `fa-thename`
- Built-in flag icons: Use ISO3611-Alpha-2 values
- Custom image URL
- Custom image data URI (embedded)

[ ]:
```
g.encode_point_badge(
  'assorted',
  'TopRight',
  shape="circle", #clip excess
  categorical_mapping={
      'macbook': 'laptop', #https://fontawesome.com/v4.7.0/icons/
      'Canada': 'flag-icon-ca', #ISO3611-Alpha-2: https://github.com/datasets/country-
↪codes/blob/master/data/country-codes.csv
      'embedded_smile': '
↪iVBORw0KGgoAAAANSUhEUgAAARkAAAC0CAMAAACXO6ihAAAAgVBMVEUAAAD////T09P4+PiYmJihoaHf39/7+/
↪vm5uYLCwsRERHy8vJ+fn7v7+83Nze2trbZ2dlRUVFGRka9vb1xcXEYGBhbW1tWVlbq6uq7u7usrKxgYGA8PDwWFhbNzc14eHhpaWm
↪AS9L06ZlkZm5I37i+acUSA9dkjRNJal/
↪JPNYjfxRBlvW1guzb3leBPupJo8wLGfat0wvgaOrjEQo0b1vsfrGTKvSUiLtW7RekaRNvGRQV32L1x9it5mX0Wh86lu+3uCM24jJEI
↪Wwud3WUy3ihpq+pewBB9yTnPWMXjivfURYjxL2hIDV3j+KFx3Wwwan8TFixl5VpdtvswuWHEXGKOhBuD4xZS0mntRcPxvRYT1Lkvt
↪eIEVW4mRuIMhKH0pztM2MvmMjtMzEh+nnC9QiP1jb6ayzgcM0PpTsSOVA4tZfQhMgNmQWsX4dwTI/
↪1ZsvUL8G7GbYV8jplhuGjupNJKayl+7en8JNn6hUpq67WWWmJitk8SrWfArPNAR0Hdybg9R7KecQYVbtFebsuI2TxHsr5xh1mnvSE
↪SbDesezCDLYN5k8TrWdQZprNyT0mRm/
↪tde8E2psaF9hmnJbX4vR7M2zo2NrQnea8MTmYARi1iKj2uhBMMx7QAiVbmnSqHSXmXHkZjAEFpF1ZtR3hUlgNpqlvWO+JBLle3DW6
↪e2GVyjibe4b8n/GBVHja9H14Wgxu3n6dbV+SQA2/
↪deutxvBVq2aksU0QkpO9Z7O272HsdLdHigvSGXznuPMgmek6wILXOb12S2u1VtatAFlfh5UvaAL0QMzhoyWV4D2bdqx1g1shU7Ct7
↪xzqBbnEa2EHwFoUeVve3DwJkGQlfmX2hMw1rRpmCjb0WBIZxZLdvC3hg0WK9mmCVX3PdWchuwAp1NqTaMBTDTg1z9YwomUI0rAaIBI
↪qYCaS1bNRVi3fW/
↪DqAFzaDJx9doKvBLPFuolcABdrmYXytJqbk7vD+gwdRrLEfTip0v1CiCVrwv1ncGsNZzlJASTVL5uVzIoOv7u6WK9AEJS+7otpmCAI
↪XwTN7QLMKXlwoQ4FnMY2swQf+EDtbND/K1sK6MQ0TGWG7RfURbc13ZAwqOAYhITUfywkjKZB98ZgI/
↪GAAZdbA6AG+JsvJrWBLmVvGTUTlRIzSP9DCZax04XmsWZrLNaAI4MXLL5BMZw4uasyS3YwTMcMoGOXz7Dz+M/
↪qg2by+Sq49C1cv9g0MeMMnBghWRNrMQOeligOdpUYf7iKDMbGE4lxBpK16SH2sY7Cy8bRepCLkg0w1Uh2DUOPInWgScfbcFkshpLi
↪wWTF0FrwHERe/
↪ByZXgNbxQa07vc6Fou90ZunPsDAqVZaikGgljycEJtuzFxo7OdwNnJ3Y+ZOywzlQFMuqUgLMwdaZiCHU2ZII5ZmsLnUENuMeTlPrY
↪lDbIRdmHGTIssI1634HtlQMxIOnyapnE3beY4GhQz3bH2/4iZr83ykGE5bOm2tciKLDeVeMxbXNwZtpwceAmLW/
↪EmUp6ZW/72uGP0XhIWL7zj0WdGdqq2MbMp7ur2DoDmRb4yHo8VxZA9Jy7/3J1UQP7PMtKzIooRBZiCdeDp9ji/
↪1dY9TyV1ozeq+W6LtRPZ5a0aqztm5hS5+dtt2UPxwXP2EBX3p+lWtgtZ9chLIT50Cfsxo2txg5Y9aEZv13KJr5FR3nXMnhayZ5Pax
↪6HAjB3YpV2HAHtvPGkxKPRfJaNF717gtZfi3jlL13z8rakrlFZey4yswdJVpU2L5lljAMn5sR2viN4zJboWFXkg1P8Z6qUcrtCIlM
↪tGpktzHzVvPEq3WqkCHlmQvzxZU4JJ8ygw7fK04C+6s/
↪IUXhmZG4PDmHmLFduy6FxzCjiYSqQzYkyI+5G0JqZ2dclXD6yFFoIc44Zh/
↪s83ZhpODxIYEZIuV8yU3eAzEhkpgr9xjNTefWlkZna87G0tbvKFhwzfGfrxAzK/
↪qfngJx3OTPLmr0l5cWCGfb9jOJWlMhqX+n2mNpTlRnbMFDXbWQGfupemgYukfUo1bRda4cNUgEyx0N1nMmZgaxkI8Ur9zLeSH1zZm
↪3hvSqoa/
↪iBupJY0T6e4EZVwtXyXIL7BHz8a4GHrxjxnGllTNAaBBmrk4QwOPdoEA+myWnq4Y+rr0NiBmbvWCtasAnYSZk/
↪+TMgPhjmpVBZcwUwzP9lO5yBc05Z+YMnPmgIEBRMk7MqEhBkVjmwtrAjGcmIolnoHI6NQZWEb4DCthwOSDM5ICxRtBnWEeL5hM4d7
↪C0bPVY4xX66ZGZSmCOppccyw63I3ZmgrXJ7SdLrcdWHGK4emoJC/
```

(continues on next page)

(continued from previous page)

```
↪bGVtzNygM6Gd5A3MkL69uu1myfnC3G0oOchjZo7kL9+sZ8ZrZ4bNQV41JdYDZkj1z4X8q4fMwG0+8n80MCOm1AQ1CLkhHzMD05ny9
↪GTP8vmY3wPt2/
↪19mYKbDSa47MgMNGyUueMzMhPReZfMzZiTh8AwF7Wl+OWbQCNWBGTjZ9qfMzCrnihggwKswQ6eJ7zFj/
↪JIZaVMxtuzwT5lBvrS/ZEaC3nT7KTPSLhUPT9P/ghkYgW3kAezGjPkjZjaQ1HX/Y2YyxWGq85au8wfMUNUZHSB/
↪7MQM9UCgxImPmYHvXShHP2VGyt2w24iZ5tHld8ywI35iG5jZgXBo+oOD0Tsyg5IqPmYGTP8mHbgrMxnOMVVuSgfqz5mJqLIKpzkb0
↪3gT3y/jwO2Ys8MpT+Qx05DWsBl9gNn7EDPjLdDatpcg/U8sM9STFP2NmWrzAoCsXnE71fWZuY3y/
↪hNybGTN3egZQWnS2TQSlHzFzgQ/GVD3hTPUKM3TUHp1/
↪yEzRKHXKDLxwipkRcqa2MJNQD1Ph9dgwp17uuaLCjg1HDVC6MmAmVOqZAVt75MKMP0fezlpmqBOp/
↪KcmL3uVGei0CWKG2ktwgE+MmZH5lUT4uy4zEm0jlh5F1M9LmBH1bYGZVQKi6bNFZqSix1L3s15M+UlIbzUWixliZuQu57fkNqcj5j
↪4x6u+XzjXmhp96Gi90KBrKg/BYJ+AijYuWTvj/
↪YTcxZKBf0r2jCVG+XyS+sjx+azpUVPOQ8Cr9a6jE3sG3rMjb+Z+z8mKPKXKgF/
↪PwYkAn6w5DRqoa3ynh12A3ZkzMLc+NQp29+HPA1q6vj0TZs2caJDvGuli87UtFKfyTVQEa+4+TX8sMdPSrk2PZdWWg4l8oajWIxZq
↪mkMZSyCZ3FaqbUe0ph88xIc6HR2Gn5KUpmpBR1sNHYIaNBwYy4WsGdFSPWjlvUMRuZGRceCIf/
↪s5YZoTlvG5gBLVxcdHKllVjeF0eaBVfCvUkcM9LX1iXc+d4SmoIfPmKm0hhdJFsjM+Vw+UNmasYFdkbWTmjhVmw+ZEa60cVLK8o68
↪gejzMiyKxu5A7QwJU5iO5Xc+HB4qMem9KcFsQjcHSkAhpLmB007slupoXG/D/
↪XPPCD+ydTnFaLqz5mDUGJQi6J2OVAFzrHbnyRJrz0svvwEA6y6kxUlFVyx1YJId7tmrjsNj0etWVN9jekz2w0x3GC6f+TNvqyWJ6O
↪pvIxsDVhZrCSZ/zZenhnSQHDnIHPKoY4Zzhb5Ff4VZlDsIKz6bGqY2YDW+fsDK/4VZtisAiamsaPMGNQBDbP7/
↪3CWXfKPMEPlm4ORlk9hVP93r/E6VJlj/
↪rdHKd3mB1CtDe00fclza8EGVaJ0vY6P1BQoViMqlhGB89scrOIDXzELcFhXcxhbGpjRf33GlPjEVzyBqIGZclauZ8b/
↪fdv/d5nZksqLMdI5L9v/4a3/
↪AjMLt+q2c5nSG6p8qGykrlse1hniG19xnMksMy3CLcOmmwNKmLv5Oj7k0frTeN22H+M72PBYv+LclONrsQnjqVpW/
↪kbV4f8AHefeSC51gZgAAAAASUVORK5CYII=',
      'external_logo': 'https://awsmp-logos.s3.amazonaws.com/4675c3b9-6053-4a8c-8619-
↪6519b83bbbfd/536ec8b5c79de08fcac1086fdf74f91b.png'
  },
  default_mapping="question").plot()
```

[ ]:
```
g.encode_point_badge(
  'assorted',
  'TopRight',
  shape="circle", #clip excess
  categorical_mapping={
      'macbook': 'laptop', #https://fontawesome.com/v4.7.0/icons/
      'Canada': 'flag-icon-ca', #ISO3611-Alpha-2: https://github.com/datasets/country-
↪codes/blob/master/data/country-codes.csv
      'embedded_smile': '
↪iVBORw0KGgoAAAANSUhEUgAAARkAAAC0CAMAAACXO6ihAAAAgVBMVEUAAAD////T09P4+PiYmJihoaHf39/7+/
↪vm5uYLCwsRERHy8vJ+fn7v7+83Nze2trbZ2dlRUVFGRka9vb1xcXEYGBhbW1tWVlbq6uq7u7usrKxgYGA8PDwWFhbNzc14eHhpaWm
↪AS9L06ZlkZm5I37i+acUSA9dkjRNJal/
↪JPNYjfxRBlvW1guzb3leBPupJo8wLGfat0wvgaOrjEQo0b1vsfrGTKvSUiLtW7RekaRNvGRQV32L1x9it5mX0Wh86lu+3uCM24jJEI
↪Wwud3WUy3ihpq+pewBB9yTnPWMXjivfURYjxL2hIDV3j+KFx3Wwwan8TFixl5VpdtvswuWHEXGKOhBuD4xZS0mntRcPxvRYT1Lkvt
↪eIEVW4mRuIMhKH0pztM2MvmMjtMzEh+nnC9QiP1jb6ayzgcM0PpTsSOVA4tZfQhMgNmQWsX4dwTI/
↪1ZsvUL8G7GbYV8jplhuGjupNJKayl+7en8JNn6hUpq67WWWmJitk8SrWfArPNAR0Hdybg9R7KecQYVbtFebsuI2TxHsr5xh1mnvSE
↪SbDesezCDLYN5k8TrWdQZprNyT0mRm/
↪tde8E2psaF9hmnJbX4vR7M2zo2NrQnea8MTmYARi1iKj2uhBMMx7QAiVbmnSqHSXmXHkZjAEFpF1ZtR3hUlgNpqlvWO+JBLle3DW6
↪e2GVyjibe4b8n/GBVHja9H14Wgxu3n6dbV+SQA2/
↪deutxvBVq2aksU0QkpO9Z7O272HsdLdHigvSGXznuPMgmek6wILXOb12S2u1VtatAFlfh5UvaAL0QMzhoyWV4D2bdqx1g1shU7Ct7
↪xzqBbnEa2EHwFoUeVve3DwJkGQlfmX2hMw1rRpmCjb0WBIZxZLdvC3hg0WK9mmCVX3PdWchuwAp1NqTaMBTDTg1z9YwomUI0rAaIB
↪qYCaS1bNRVi3fW/
↪DqAFzaDJx9doKvBLPFuolcABdrmYXytJqbk7vD+gwdRrLEfTip0v1CiCVrwv1ncGsNZzlJASTVL5uVzIoOv7u6WK9AEJS+7otpmCAI
```

(continues on next page)

(continued from previous page)

```
↪XwTN7QLMKXlwoQ4FnMY2swQf+EDtbND/K1sK6MQ0TGWG7RfURbcl3ZAwqOAYhITUfywkjKZB98ZgI/
↪GAAZdbA6AG+JsvJrWBLmVvGTUTlRIzSP9DCZax04XmsWZrLNaAI4MXLL5BMZw4uasyS3YwTMcMoGOXz7Dz+M/
↪qg2by+Sq49C1cv9g0MeMMnBghWRNrMQOeligOdpUYf7iKDMbGE4lxBpK16SH2sY7Cy8bRepCLkg0w1Uh2DUOPInWgScfbcFkshpLi
↪wWTF0FrwHERe/
↪ByZXgNbxQa07vc6Fou90ZunPsDAqVZaikGgljycEJtuzFxo7OdwNnJ3Y+ZOywzlQFMuqUgLMwdaZiCHU2ZII5ZmsLnUENuMeTlPrY
↪lDbIRdmHGTIssI1634HtlQMxIOnyapnE3beY4GhQz3bH2/4iZr83ykGE5bOm2tciKLDeVeMxbXNwZtpwceAmLW/
↪EmUp6ZW/72uGP0XhIWL7zj0WdGdqq2MbMp7ur2DoDmRb4yHo8VxZA9Jy7/3J1UQP7PMtKzIooRBZiCdeDp9ji/
↪1dY9TyV1ozeq+W6LtRPZ5a0aqztm5hS5+dtt2UPxwXP2EBX3p+lWtgtZ9chLIT50Cfsxo2txg5Y9aEZv13KJr5FR3nXMnhayZ5Paxu
↪6HAjB3YpV2HAHtvPGkxKPRfJaNF717gtZfi3jlL13z8rakrlFZey4yswdJVpU2L5lljAMn5sR2viN4zJboWFXkg1P8Z6qUcrtCIlMi
↪tGpktzHzVvPEq3WqkCHlmQvzxZU4JJ8ygw7fK04C+6s/
↪IUXhmZG4PDmHmLFduy6FxzCjiYSqQzYkyI+5G0JqZ2dclXD6yFFoIc44Zh/
↪s83ZhpODxIYEZIuV8yU3eAzEhkpgr9xjNTefWlkZna87GOtbvKFhwzfGfrxAzK/
↪qfngJx3OTPLmr0l5cWCGfb9jOJWlMhqX+n2mNpTlRnbMFDXbWQGfupemgYukfUo1bRda4cNUgEyx0N1nMmZgaxkI8Ur9zLeSH1zZm
↪3hvSqoa/
↪iBupJY0T6e4EZVwtXyXIL7BHz8a4GHrxjxnGllTNAaBBmrk4QwOPdoEA+myWnq4Y+rr0NiBmbvWCtasAnYSZk/
↪+TMgPhjmpVBZcwUwzP9lO5yBc05Z+YMnPmgIEBRMk7MqEhBkVjmwtrAjGcmIolnoHI6NQZWEb4DCthwOSDM5ICxRtBnWEeL5hM4d7
↪C0bPVY4xX66ZGZSmCOppccyw63I3ZmgrXJ7SdLrcdWHGK4emoJC/
↪bGVtzNygM6Gd5A3MkL69uu1myfnC3G0oOchjZo7kL9+sZ8ZrZ4bNQV41JdYDZkj1z4X8q4fMwG0+8n80MCOm1AQ1CLkhHzMD05ny9
↪GTP8vmY3wPt2/
↪19mYKbDSa47MgMNGyUueMzMhPReZfMzZiTh8AwF7Wl+OWbQCNWBGTjZ9qfMzCrnihggwKswQ6eJ7zFj/
↪JIZaVMxtuzwT5lBvrS/ZEaC3nT7KTPSLhUPT9P/ghkYgW3kAezGjPkjZjaQ1HX/Y2YyxWGq85au8wfMUNUZHSB/
↪7MQM9UCgxImPmYHvXShHP2VGyt2w24iZ5tHld8ywI35iG5jZgXBo+oOD0Tsyg5IqPmYGTP8mHbgrMxnOMVVuSgfqz5mJqLIKpzkb0
↪3gT3y/jwO2Ys8MpT+Qx05DWsBl9gNn7EDPjLdDatpcg/U8sM9STFP2NmWrzAoCsXnE71fWZuY3y/
↪hNybGTN3egZQWnS2TQSlHzFzgQ/GVD3hTPUKM3TUHp1/
↪yEzRKHXKDLxwipkRcqa2MJNQD1Ph9dgwp17uuaLCjg1HDVC6MmAmVOqZAVt75MKMP0fezlpmqBOp/
↪KcmL3uVGei0CWKG2ktwgE+MmZH5lUT4uy4zEm0jlh5F1M9LmBH1bYGZVQKi6bNFZqSix1L3s15M+UlIbzUWixliZuQu57fkNqcj5ji
↪4x6u+XzjXmhp96Gi90KBrKg/BYJ+AijYuWTvj/
↪YTcxZKBf0r2jCVG+XyS+sjx+azpUVPOQ8Cr9a6jE3sG3rMjb+Z+z8mKPKXKgF/
↪PwYkAn6w5DRqoa3ynh12A3ZkzMLc+NQp29+HPA1q6vj0TZs2caJDvGuli87UtFKfyTVQEa+4+TX8sMdPSrk2PZdWWg4l8oajWIxZql
↪mkMZSyCZ3FaqbUe0ph88xIc6HR2Gn5KUpmpBR1sNHYIaNBwYy4WsGdFSPWjlvUMRuZGRceCIf/
↪s5YZoTlvG5gBLVxcdHKllVjeF0eaBVfCvUkcM9LX1iXc+d4SmoIfPmKm0hhdJFsjM+Vw+UNmasYFdkbWTmjhVmw+ZEa60cVLK8o68
↪gejzMiyKxu5A7QwJU5iO5Xc+HB4qMem9KcFsQjcHSkAhpLmBO07slupoXG/D/
↪XPPCD+ydTnFaLqz5mDUGJQi6J2OVAFzrHbnyRJrz0svvwEA6y6kxUlFVyx1YJId7tmrjsNj0etWVN9jekz2w0x3GC6f+TNvqyWJ6O
↪pvIxsDVhZrCSZ/zZenhnSQHDnIHPKoY4Zzhb5Ff4VZlDsIKz6bGqY2YDW+fsDK/4VZtisAiamsaPMGNQBDbP7/
↪3CWXfKPMEPlm4ORlk9hVP93r/E6VJlj/
↪rdHKd3mB1CtDe00fclza8EGVaJ0vY6P1BQoViMqlhGB89scr0IDXzELcFhXcxhbGpjRf33GlPjEVzyBqIGZclauZ8b/
↪fdv/d5nZksqLMdI5L9v/4a3/
↪AjMLt+q2c5nSG6p8qGykrlse1hniG19xnMksMy3CLcOmmwNKmLv5Oj7k0frTeN22H+M72PBYv+LclONrsQnjqVpW/
↪kbV4f8AHefeSC51gZgAAAAASUVORK5CYII=',
      'external_logo': 'https://awsmp-logos.s3.amazonaws.com/4675c3b9-6053-4a8c-8619-
↪6519b83bbbfd/536ec8b5c79de08fcac1086fdf74f91b.png'
  },
  default_mapping="question",
  bg={'color': {'mapping': {'categorical': {'fixed': {}, 'other': 'white'}}}})\
.plot()
```

Badges as continuous mappings and styling

You can also use value ranges to pick the glyph, and use text as the glyph

[ ]:
```
g.encode_point_badge(
    'score',
   'TopRight',
    continuous_binning=[
        [33, None],
        [66, 'info-circle'],
        [None, 'exclamation-triangle']
    ],
    border={'width': 2, 'color': 'black', 'stroke': 'solid'},
    color={'mapping': {'categorical': {'fixed': {}, 'other': 'black'}}},
    bg={'color': {'mapping': {'continuous': {'bins': [
        [33, '#ccc'],
        [66, 'yellow'],
        [200, 'red']
    ], 'other': 'black'}}}}).plot(render=False)
```

Flag inference

The below code generates ISO3166 mappings from different conventions to Alpha-2

[ ]:
```
codes = pd.read_csv('https://raw.githubusercontent.com/datasets/country-codes/master/
↪data/country-codes.csv')
codes.columns
```

[ ]:
```
country_to_iso_flag = {
    o['CLDR display name']: 'flag-icon-' + o['ISO3166-1-Alpha-2'].lower()
    for o in codes[['CLDR display name', 'ISO3166-1-Alpha-2']].dropna().to_dict('records
↪')
}

# also try with   shape="circle" and   border={'width': 2, 'color': 'black', 'stroke':
↪'solid'}
g.encode_point_badge('origin', 'BottomRight', categorical_mapping=country_to_iso_flag).
↪plot()
```

[ ]:
```

```

Collections in PyGraphistry

Collections define labeled subsets of a graph (nodes, edges, or subgraphs) using full GFQL. They enable advanced, layered styling that overrides base encodings when you need precise highlights.

Use collections when you want: - baseline encodings (for example, by entity type) plus overlays for alerts or critical paths - multiple overlapping highlights with a priority order - a UI panel for toggling focused subsets on and off

Collections are evaluated in priority order, with higher priority collections overriding lower ones for styling.

In this notebook, we build sets using GFQL AST helpers, combine them with intersections, and apply node and edge colors. Collections can be based on nodes, edges, or multi-step graph traversals (Chain).

[ ]:
```
from pathlib import Path
import pandas as pd
import graphistry
from graphistry import collection_set, collection_intersection, n, e_forward, Chain

edges = pd.read_csv(Path('demos/data/honeypot.csv'))
g = graphistry.edges(edges, "attackerIP", "victimIP")
```

[ ]:
```
# Use Chain to select subgraphs (nodes + edges) by edge attributes
collections = [
    collection_set(
        expr=Chain([n(), e_forward({"vulnName": "MS08067 (NetAPI)"}), n()]),
        id='netapi',
        name='MS08067 (NetAPI)',
        node_color='#00BFFF',
        edge_color='#00BFFF',
    ),
    collection_set(
        expr=Chain([n(), e_forward({"victimPort": 445.0}), n()]),
        id='port445',
        name='Port 445',
        node_color='#32CD32',
        edge_color='#32CD32',
    ),
    collection_intersection(
        sets=['netapi', 'port445'],
        name='NetAPI + 445',
        node_color='#AABBCC',
        edge_color='#AABBCC',
    ),
]

g2 = g.collections(
    collections=collections,
    show_collections=True,
    collections_global_node_color='CCCCCC',
    collections_global_edge_color='CCCCCC',
)
```

(continues on next page)

(continued from previous page)

```
g2._url_params
```

```
[ ]: # Render (requires graphistry.register(...))
g2.plot()
```

Notes and validation

- Order matters: earlier collections override later ones.
- Use collections for priority-based subsets and overlaps; use `encode_*` for simple column-driven colors.
- Helper constructors: `graphistry.collection_set(...)` and `graphistry.collection_intersection(...)` return JSON-friendly dicts (AST inputs wrap to `gfql_chain`).
- Provide `id` for sets used by intersections.
- Global colors apply to nodes/edges not in any collection; `#` is optional.
- Use `validate='strict'` to raise, or `warn=False` to silence warnings.

Wire protocol and pre-encoded strings:

```
collections_wire = [
    {
        "type": "set",
        "name": "Wire Protocol Example",
        "node_color": "#AA00AA",
        "expr": {
            "type": "gfql_chain",
            "gfql": [
                {"type": "Node", "filter_dict": {"status": "purchased"}}
            ]
        }
    }
]
g.collections(collections=collections_wire)

g.collections(collections=encoded_collections, encode=False)
```

Run `g2.plot()` in a notebook session with valid credentials to render inline.

Overlap priority example

Earlier collections override later ones when they overlap.

```
collections_priority = [
    collection_set(
        expr=Chain([n(), e_forward({"vulnName": "MS08067 (NetAPI)"}), n()]),
        id="netapi",
        name="MS08067 (NetAPI)",
```

(continues on next page)

(continued from previous page)

```
        node_color="#FFAA00",
        edge_color="#FFAA00",
    ),
    collection_set(
        expr=Chain([n(), e_forward({"victimPort": 445.0}), n()]),
        id="port445",
        name="Port 445",
        node_color="#00BFFF",
        edge_color="#00BFFF",
    ),
]
g.collections(collections=collections_priority)
```

For more on color encodings, see the *Color encodings notebook*.

10.8.2.2 Geographic (Kepler.gl)

Geospatial Network Visualization with Kepler.gl Integration

Visualizing geospatial networks combines the power of graph analysis with geographic context. This tutorial demonstrates how to create interactive map-based visualizations using PyGraphistry's Kepler.gl integration to analyze company networks with real-world geographic coordinates.

PyGraphistry's Kepler.gl integration enables you to overlay network data on maps, making it ideal for analyzing location-based relationships like supply chains, business partnerships, and regional connections.

Key Benefits

- **Geographic Context**: Visualize networks with real-world coordinates, making spatial patterns and regional clusters immediately visible.
- **Interactive Mapping**: Leverage Kepler.gl's powerful map visualization capabilities including multiple layer types (points, arcs, hexbins) and dynamic filtering.
- **Flexible Data Integration**: Seamlessly combine network topology with geospatial attributes from sources like Wikidata, enriching your analysis with location-based insights.

Tutorial

Follow this tutorial to create geospatial network visualizations:

- Fetch company data with geographic coordinates from Wikidata
- Create a network of company relationships
- Visualize the network on an interactive map using Kepler.gl layers

[ ]:
```
import graphistry
graphistry.__version__
```

[ ]:
```
# API key page (free GPU account): https://hub.graphistry.com/users/personal/key/
# graphistry.register(
#     api=3,
#     personal_key_id=FILL_ME_IN,
#     personal_key_secret=FILL_ME_IN
# )
```

Data: Fetching Company Information from Wikidata

We'll use a SPARQL query to fetch 5,000 companies with headquarters coordinates and related business attributes from Wikidata. The query retrieves:

- Company names and headquarters locations
- Geographic coordinates (latitude/longitude)
- Business metrics (employees, revenue, market cap)
- Additional metadata (industry, country, website, stock ticker)

[ ]:
```
import requests
import pandas as pd

# SPARQL query to fetch data from Wikidata
sparql_query = """
# Companies with HQ coordinates + commonly populated attributes
SELECT ?company ?companyLabel ?hq ?hqLabel
       ?countryLabel ?hqCountryLabel ?industryLabel
       ?inception ?employees ?revenue ?marketCap ?netIncome
       ?ticker ?isin ?website
       (xsd:decimal(STRBEFORE(STRAFTER(STR(?coord), "Point("), " ")) AS ?longitude)
       (xsd:decimal(STRAFTER(STRBEFORE(STR(?coord), ")"), " ")) AS ?latitude)
WHERE {
  # Company w/ headquarters
  ?company wdt:P31 wd:Q4830453 ;     # instance of: business/enterprise
           wdt:P159 ?hq .           # headquarters location

  # Require coordinates (ensures non-sparse lat/long)
  ?hq wdt:P625 ?coord .

  # Frequently populated company attributes
  OPTIONAL { ?company wdt:P17   ?country . }    # country
  OPTIONAL { ?hq      wdt:P17   ?hqCountry . }  # HQ country
  OPTIONAL { ?company wdt:P452  ?industry . }   # industry
  OPTIONAL { ?company wdt:P571  ?inception . }  # inception (founded)
  OPTIONAL { ?company wdt:P1128 ?employees . }  # number of employees
  OPTIONAL { ?company wdt:P2139 ?revenue . }    # revenue
  OPTIONAL { ?company wdt:P2295 ?marketCap . }  # market cap
  OPTIONAL { ?company wdt:P2293 ?netIncome . }  # net income
  OPTIONAL { ?company wdt:P249  ?ticker . }     # stock ticker
  OPTIONAL { ?company wdt:P946  ?isin . }       # ISIN
  OPTIONAL { ?company wdt:P856  ?website . }    # website
```

(continues on next page)

(continued from previous page)

```
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en" .
  }
}
LIMIT 5000
"""

# Wikidata SPARQL endpoint URL
wikidata_endpoint = "https://query.wikidata.org/sparql"

# Set up headers for the request
headers = {
    "Accept": "application/sparql-results+json"  # Request JSON format
}

# Send the GET request to the SPARQL endpoint
response = requests.get(wikidata_endpoint, headers=headers, params={"query": sparql_
↪query})

# Check if the request was successful
if response.status_code == 200:
    # Parse the JSON response
    data = response.json()

    # Extract the results
    results = data["results"]["bindings"]

    # Create a list of dictionaries for the DataFrame
    processed_results = []
    for item in results:
        row = {}
        for key, value in item.items():
            row[key] = value["value"]
        processed_results.append(row)

    # Create a pandas DataFrame
    companies_df = pd.DataFrame(processed_results)

    print("Data successfully fetched")
else:
    print(f"Error fetching data: {response.status_code}")
    print(response.text)
```

[14]:
```
# Convert compatible columns to float type
def convert_to_float(df):
    """Convert columns that can be interpreted as float to float type."""
    df_converted = df.copy()

    for col in df_converted.columns:
        # Skip if already float
        if df_converted[col].dtype == "float64":
```

(continues on next page)

(continued from previous page)

```
            continue

        try:
            # Try to convert to float
            df_converted[col] = pd.to_numeric(df_converted[col], errors="coerce")

            # If all values became NaN, revert to original
            if df_converted[col].isna().all():
                df_converted[col] = df[col]
        except:
            # Keep original if conversion fails
            pass

    return df_converted

# Apply float conversion
companies_df = convert_to_float(companies_df)
```

[15]:
```
len(companies_df)
```

[15]:
```
5000
```

Data Validation

[16]:
```
companies_df.dtypes
```

[16]:
```
company            object
hq                 object
inception          object
companyLabel      float64
hqLabel            object
countryLabel       object
hqCountryLabel     object
industryLabel      object
longitude         float64
latitude          float64
employees         float64
isin               object
website            object
revenue           float64
marketCap         float64
dtype: object
```

[17]:
```
invalid_lat_long_count = companies_df[(companies_df["latitude"].isna()) |
                                      (companies_df["longitude"].isna()) |
                                      (companies_df["latitude"] == 0) |
                                      (companies_df["longitude"] == 0)].shape[0]

print(f"Number of records with invalid latitude or longitude: {invalid_lat_long_count}")
```

```
Number of records with invalid latitude or longitude: 0
```

Creating Synthetic Relationships

For demonstration purposes, we'll create synthetic relationships between companies with timestamps and relationship types.

[18]:
```
import numpy as np

# Create pairs of companies. For simplicity, let's create random pairs.
# In a real scenario, you would likely have a specific way to determine relationships.
num_edges = 20000  # You can adjust the number of edges
src_companies = np.random.choice(companies_df["company"], num_edges)
dest_companies = np.random.choice(companies_df["company"], num_edges)

# Generate random timestamps between 2010 and 2025
start_date = pd.to_datetime("2010-01-01")
end_date = pd.to_datetime("2025-12-31")
time_range = end_date - start_date
random_seconds = np.random.rand(num_edges) * time_range.total_seconds()
random_timestamps = start_date + pd.to_timedelta(random_seconds, unit="s")

# Create a list of possible relationship types
relationship_types = ["acquisition", "partnership", "investment", "merger",
↪"collaboration"] # Add more types as needed

# Assign random relationship types
random_types = np.random.choice(relationship_types, num_edges)

# Create the edges_df DataFrame
edges_df = pd.DataFrame({
    "src": src_companies,
    "dest": dest_companies,
    "timestamp": random_timestamps,
    "type": random_types
})

display(edges_df.head(3))
```

```
                                    src  \
0  http://www.wikidata.org/entity/Q391795
1  http://www.wikidata.org/entity/Q403685
2  http://www.wikidata.org/entity/Q320111

                                   dest                       timestamp  \
0  http://www.wikidata.org/entity/Q429877 2011-03-07 05:44:15.732827432
1  http://www.wikidata.org/entity/Q476185 2011-03-04 15:24:34.197431505
2  http://www.wikidata.org/entity/Q181697 2012-09-19 04:47:02.804731473

          type
0  acquisition
1       merger
```

(continues on next page)

(continued from previous page)

```
2       merger
```

Standard Graph Visualization

The bare minimum: When your data contains `latitude` and `longitude` columns, Graphistry automatically enables map-based visualization with a simple call to `.plot()`.

This is the easiest way to get started with geographic visualization - no explicit Kepler configuration needed. Simply: 1. Have geographic coordinates in your data (`latitude` and `longitude` columns) 2. Set `layout_settings(play=0)` to disable auto-layout 3. Call `.plot()`

Graphistry automatically detects the geographic columns and renders your network on an interactive map with default Kepler.gl settings.

[30]:
```
g = graphistry.bind(source="src", destination="dest", node="company") \
    .nodes(companies_df) \
    .edges(edges_df) \
    .layout_settings(play=0)
```

[32]:
```
g.plot()
```

[32]:
```
<IPython.core.display.HTML object>
```

Configure: Custom Datasets and Layers

Taking control: While the default visualization is convenient, you often want precise control over the data available to kepler and how your data appears on the map.

This section demonstrates how to **explicitly populate Kepler datasets and layers** using PyGraphistry's encoding methods:

- **Datasets** (`.encode_kepler_dataset()`): Define which data to make available to Kepler (nodes, edges, etc.)
- **Layers** (`.encode_kepler_layer()`): Configure how each dataset appears on the map (point layers, line layers, colors, visibility, etc.)

By explicitly configuring datasets and layers, you gain fine-grained control over: - Which data appears in which layer - Visual properties (colors, opacity, thickness) - Layer visibility (toggle layers on/off by default) - Layer types (points, lines, arcs, hexbins, etc.)

Note how the edge coordinates (`edgeSourceLatitude`, `edgeSourceLongitude`, etc.) are automatically created when an edge type dataset is specified.

[34]:
```
# Create Kepler encoding with datasets and layers
g2 = g \
    .encode_kepler_dataset(
        id="companies-dataset",
        type="nodes",
        label="Companies"
    ) \
    .encode_kepler_dataset(
```

(continues on next page)

(continued from previous page)

```
        id="relationships-dataset",
        type="edges",
        label="Relationships",
    ) \
    .encode_kepler_layer({
        "id": "companies-layer",
        "type": "point",
        "config": {
            "dataId": "companies-dataset",
            "label": "Company Headquarters",
            "columns": {
                "lat": "latitude",
                "lng": "longitude"
            }
        }
    }) \
    .encode_kepler_layer({
        "id": "relationships-layer",
        "type": "line",
        "config": {
            "dataId": "relationships-dataset",
            "label": "Company Relationships",
            "columns": {
              "lat0": "edgeSourceLatitude",
              "lng0": "edgeSourceLongitude",
              "lat1": "edgeTargetLatitude",
              "lng1": "edgeTargetLongitude"
            },
            "isVisible": False,
            "color": [100, 200, 200],
            "visConfig": {
                "opacity": 0.01,
                "thickness": 1
            }
        }
    })

print("Kepler encoding applied successfully")
```

```
Kepler encoding applied successfully
```

[35]:
```
g2.plot()
```

[35]:
```
<IPython.core.display.HTML object>
```

Configure: Choropleth Maps with Computed Columns

The power of maps: This final example demonstrates the full capabilities of PyGraphistry's Kepler integration by combining multiple advanced features.

What This Example Demonstrates

1. **Choropleth Maps**: Geographic regions (countries) colored by aggregated metrics - a powerful way to visualize regional patterns
2. **Computed Columns**: Dynamic data transformations that aggregate company-level data (market cap, revenue) to country-level statistics without an external pipeline
3. **Multi-layer Visualization**: Combining point layers (companies), arc layers (relationships), and geojson layers (countries) in a single view

The Computed Column Magic

The `computed_columns` feature calculates a **Price-to-Sales (P/S) ratio** for each country by: - **Aggregating** market cap values by country (mean of `marketCap`) - **Normalizing** by revenue (mean of `revenue`) - **Binning** results into categories for color encoding - Computing: `(avg marketCap by country) / (avg revenue by country)`

This creates a choropleth showing which countries have companies with higher valuation multiples relative to their revenue - a key financial metric for investors.

Why This Matters

This example shows how geographic visualization can transform complex raw financial data into intuitive visual insights: - **Spatial patterns**: See which regions have overvalued vs undervalued companies - **Multi-scale analysis**: View individual companies and country-level aggregates simultaneously - **Interactive exploration**: Toggle layers to focus on different aspects (companies, relationships, regional metrics)

This progression from simple geographic plotting → explicit layer control → computed aggregations demonstrates how PyGraphistry's Kepler integration enables sophisticated geospatial analytics with relatively simple Python code.

[36]:
```
from graphistry.kepler import KeplerDataset, KeplerLayer, KeplerEncoding

# Create visualization with countries colored by price-to-sales ratio
kepler_ps_encoding = (
    KeplerEncoding()

    # Nodes dataset with company data
    .with_dataset(
        KeplerDataset(
            id="companies",
            type="nodes",
            label="Companies"
        )
    )
```

(continues on next page)

(continued from previous page)

```
    # Edges dataset with mapped coordinates
    .with_dataset(
        KeplerDataset(
            id="relationships",
            type="edges",
            label="Relationships",
            map_node_coords=True
        )
    )

    # Countries dataset with computed price-to-sales ratio
    .with_dataset(
        KeplerDataset(
            id="countries",
            type="countries",
            label="Countries by P/S Ratio",
            resolution=110,
            boundary_lakes=False,
            computed_columns={
                "avg_price_to_sales": {
                    "type": "aggregate",
                    "computeFromDataset": "companies",
                    "sourceKey": "hqCountryLabel",
                    "targetKey": "name",
                    "aggregate": "mean",
                    "aggregateCol": "marketCap",
                    "normalizer": "mean",
                    "normalizerCol": "revenue",
                    "bins": [0, 0.05, 0.1, 0.15, 0.2, 0.3, 0.5, 1, 999999],
                    "right": False,
                    "includeLowest": True
                }
            }
        )
    )

    # # Company point layer
    .with_layer(
        KeplerLayer({
            "id": "nodes",
            "type": "point",
            "config": {
                "dataId": "companies",
                "label": "Company Headquarters",
                "columns": {
                    "lat": "latitude",
                    "lng": "longitude"
                },
                "isVisible": False,
                "color": [255, 140, 0],
                "visConfig": {
                    "radius": 5,
```

(continues on next page)

(continued from previous page)

```
                    "fixedRadius": False,
                    "opacity": 0.6,
                    "outline": True,
                    "thickness": 1,
                    "strokeColor": [255, 255, 255],
                    "colorRange": {
                        "name": "Global Warming",
                        "type": "sequential",
                        "category": "Uber",
                        "colors": ["#5A1846", "#900C3F", "#C70039", "#E3611C", "#F1920E",
↪ "#FFC300"]
                    }
                }
            }
        })
    )

    # Edge arc layer (initially hidden)
    .with_layer(
        KeplerLayer({
            "id": "relationship-arcs",
            "type": "arc",
            "config": {
                "dataId": "relationships",
                "label": "Company Relationships",
                "columns": {
                "lat0": "edgeSourceLatitude",
                "lng0": "edgeSourceLongitude",
                "lat1": "edgeTargetLatitude",
                "lng1": "edgeTargetLongitude"
                },
                "isVisible": False,
                "color": [100, 200, 200],
                "visConfig": {
                    "opacity": 0.01,
                    "thickness": 1
                }
            }
        })
    )

    # Countries geojson layer with color encoding by P/S ratio
    .with_layer(
        KeplerLayer({
            "id": "countries-ps-layer",
            "type": "geojson",
            "config": {
                "dataId": "countries",
                "label": "Countries by Price-to-Sales Ratio",
                "columns": {
                    "geojson": "_geometry"
                },
```

(continues on next page)

(continued from previous page)

```
                "isVisible": True,
                "visConfig": {
                    "opacity": 0.7,
                    "strokeOpacity": 0.8,
                    "thickness": 0.5,
                    "strokeColor": [60, 60, 60],
                    "colorRange": {
                        "name": "Custom P/S Gradient",
                        "type": "sequential",
                        "category": "Custom",
                        "colors": [
                            "#000000",   # Black for lowest P/S (0-0.5)
                            "#1a0a00",   # Very dark brown (0.5-1)
                            "#331400",   # Dark brown (1-2)
                            "#4d1f00",   # Brown (2-3)
                            "#802d00",   # Dark orange-brown (3-5)
                            "#b34000",   # Medium orange (5-7)
                            "#e65c00",   # Bright orange (7-10)
                            "#ff8c1a"    # Vibrant orange for highest P/S (10+)
                        ]
                    },
                    "filled": True,
                    "outline": True,
                    "extruded": False,
                    "wireframe": False
                }
            },
            "visualChannels": {
                "colorField": {
                    "name": "avg_price_to_sales",
                    "type": "string"
                },
                "colorScale": "ordinal",
                "sizeField": None,
                "sizeScale": "linear"
            }
        })
    )

    # Configure options
    .with_options(
        center_map=True,
        read_only=False
    )

    # Configure settings
    .with_config(
        cull_unused_columns=False
    )
)

# Apply kepler encoding to graph
```

(continues on next page)

(continued from previous page)

```
g3 = g.encode_kepler(kepler_ps_encoding)

print("Price-to-Sales Ratio Visualization Created")
print(f"Datasets: {len(kepler_ps_encoding.datasets)}")
print(f"Layers: {len(kepler_ps_encoding.layers)}")
print("\nComputed column calculates: marketCap / revenue aggregated by country")
print("P/S Ratio bins: [0-0.5, 0.5-1, 1-2, 2-3, 3-5, 5-7, 7-10, 10+]")
print("Color scale: Black (low P/S) → Orange (high P/S)")
```

```
Price-to-Sales Ratio Visualization Created
Datasets: 3
Layers: 3

Computed column calculates: marketCap / revenue aggregated by country
P/S Ratio bins: [0-0.5, 0.5-1, 1-2, 2-3, 3-5, 5-7, 7-10, 10+]
Color scale: Black (low P/S) → Orange (high P/S)
```

[ ]:
```
g3.plot()
```

10.8.2.3 Static Export

Static Graph Rendering with plot_static()

Render graphs as static images (SVG, PNG) for documentation, reports, and presentations.

Key point: `plot_static()` works with **any layout** - UMAP, ring, graphviz, or manual positions.

[1]:
```
import pandas as pd
import graphistry

# Sample graph
edges_df = pd.DataFrame({'src': ['a', 'a', 'b', 'c', 'd'], 'dst': ['b', 'c', 'c', 'd', 'e
↪']})
nodes_df = pd.DataFrame({
    'id': ['a', 'b', 'c', 'd', 'e'],
    'type': ['start', 'middle', 'middle', 'middle', 'end'],
    'label': ['Start', 'Step 1', 'Step 2', 'Step 3', 'End']
})
g = graphistry.edges(edges_df, 'src', 'dst').nodes(nodes_df, 'id')
```

Basic Usage

Auto-displays in Jupyter, returns SVG object (use `.data` for bytes):

[2]:
```
g.plot_static()
```

[2]:

Styling with graph_attr, node_attr, edge_attr

[3]:
```
g.plot_static(
    graph_attr={'rankdir': 'LR', 'bgcolor': 'white'},
    node_attr={'shape': 'box', 'style': 'rounded,filled', 'fillcolor': 'lightblue'},
    edge_attr={'color': 'gray'}
)
```

[3]:

Per-Node Styling (Data-Driven)

Add graphviz attribute columns to your dataframe:

[4]:
```
nodes_styled = nodes_df.copy()
nodes_styled['fillcolor'] = nodes_styled['type'].map({
    'start': 'lightgreen', 'middle': 'lightblue', 'end': 'lightyellow'
})
nodes_styled['shape'] = 'box'

g_styled = graphistry.edges(edges_df, 'src', 'dst').nodes(nodes_styled, 'id')
g_styled.plot_static(graph_attr={'rankdir': 'LR'}, node_attr={'style': 'filled'})
```

[4]:

Layout Programs

Program	Best For
`dot`	Hierarchies, DAGs
`neato`	Small undirected graphs
`circo`	Circular layouts

[5]:
```
g.plot_static(prog='circo', node_attr={'style': 'filled', 'fillcolor': 'lightblue'})
```

[5]:

Output Formats

[6]:
```
# DOT source text
dot = g.plot_static(engine='graphviz-dot')
print(dot[:200])
```

```
digraph "" {
        graph [bb="0,0,115.89,324"];
        node [label="\N"];
        a       [height=0.5,
                id=a,
                label=Start,
                pos="74.946,306",
                width=0.95686];
        b       [height=0.5,
```

(continues on next page)

(continued from previous page)

```
                id=b,
                label="Step 1",
                pos="40.946,234"
```

[7]:
```
# Mermaid DSL
mermaid = g.plot_static(engine='mermaid-code')
print(mermaid)
```

```
graph LR
    a --> b
    a --> c
    b --> c
    c --> d
    d --> e
```

Finding More Options

Discover available attributes:

[8]:
```
from graphistry.plugins_types.graphviz_types import GRAPH_ATTRS, NODE_ATTRS, EDGE_ATTRS,␣
↪PROGS
print('Progs:', PROGS[:6])
print('Node attrs (sample):', NODE_ATTRS[:8])
```

```
Progs: ['acyclic', 'ccomps', 'circo', 'dot', 'fdp', 'gc']
Node attrs (sample): ['area', 'class', 'color', 'colorscheme', 'comment', 'distortion',
↪'fillcolor', 'fixedsize']
```

External references: - https://graphviz.org/doc/info/attrs.html - https://graphviz.org/doc/info/shapes.html - https://graphviz.org/doc/info/colors.html

See Also

- *graphviz.ipynb* - Layout algorithms for interactive Graphistry viz

10.8.2.4 Layout

Guiding Layout with Edge Weights

We can use edge attributes to guide the layout by having how much the nodes of an edge get attracted to one another be influenced by that attribute. This is useful in several scenarios: * An edge has a natural property, such as `affinity` * An edge represents multiple edges and thus represents a non-uniform weight such as `count` * Algorithms provide edge properties, such as `relevance`

By binding such an edge column to **edge_weight** and optionally tuning how much to factor in that column with the **edgeInfluence** control, we can guide the clustering to factor in the edge weight.

1. By default, every edge contributes a weight of `1` on how much to pull its nodes together.

- Multiple edges between the same 2 nodes will thus cause those nodes to be closer together

2. Activate edge weights in `api=3` (2.0): Edges with high edge weights bring their nodes closer together; edges with low weight allow their nodes to move further appart

- Set via binding `edge_weight` (`.bind(edge_weight='my_col')`)
- Edge weight values automatically normalize between 0 and 1 starting with v2.30.25

2. The edge influence control guides whether to ignore edge weight (`0`) and use it primarily (`7+`)

- Set via the UI (`Layout Controls -> Edge Influence`) or via url parameter `edgeInfluence` (`.settings(url_params={'edgeInfluence': 2})`)

[ ]:
```
import pandas as pd, graphistry
```

[ ]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Demo: Strongly connected graph of 20 nodes

- No edge weight: Appears as a regular mesh
- Same edge weights: Appears as a regular mesh
- Edge weight `1` for edges (`i`, `i+1`), defining a chain, and the other edges set to weight `0`:
  - `'edgeInfluence': 0`: Appears as a regular mesh
  - `'edgeInfluence': 1`: Still a mesh, but start to see a chain interleaved
  - `'edgeInfluence': 2`: The chain starts to form a circle around the mesh
  - `'edgeInfluence': 7`: The chain starts to become a straight line; the other edges have little relative impact (no more mesh)
- Edge weight `100` instead of `1` for the chain: same as edge weight `1` due to normalization
- Edge weight `1` for the chain's edges and `-1` for the rest: Same due to normalization

[ ]:
```
edges = []
n = 20
k = 2

edges = pd.DataFrame({
    's': [i for i in range(0,n) for j in range(0,n) if i != j],
    'd': [j for i in range(0,n) for j in range(0,n) if i != j]
})
edges['1_if_neighbor'] = edges.apply(
    lambda r: \
            1 \
                if (r['s'] == r['d'] - 1) \
                or (r['s'] == r['d'] + 1) \
            else 0,
    axis=1).astype('float32')
edges['100_if_neighbor'] = (edges['1_if_neighbor'] * 100).astype('int64')
edges['ec'] = edges['1_if_neighbor'].apply(lambda v: round(v) * 0xFF000000)
edges.head(20)
```

[ ]:
```
URL_PARAMS = {'play': 5000, 'edgeCurvature': 0.1, 'precisionVsSpeed': -3}
g = graphistry.edges(edges).bind(source='s', destination='d', edge_color='ec').
↪settings(url_params=URL_PARAMS)
```

Edge Influence 0: No weights – a mesh

[ ]:
```
g.bind(edge_weight='1_if_neighbor').settings(url_params={**URL_PARAMS, 'edgeInfluence':␣
↪0}).plot(render=True)
```

Edge influence 1: Some weight – chain interleaved into the mesh

[ ]:
```
g.bind(edge_weight='1_if_neighbor').settings(url_params={**URL_PARAMS, 'edgeInfluence':␣
↪1}).plot(render=True)
```

Edge influence 2: Strong weight – chain becomes circumference of mesh

[ ]:
```
g.bind(edge_weight='1_if_neighbor').settings(url_params={**URL_PARAMS, 'edgeInfluence':␣
↪2}).plot(render=True)
```

Edge influence 7: Non-chain edges lose relative influence – chain becomes a straight line (no more mesh)

[ ]:
```
g.bind(edge_weight='1_if_neighbor').settings(url_params={**URL_PARAMS, 'edgeInfluence':␣
↪7}).plot(render=True)
```

Edge weights -1 to 1, and 0 to 100: Same as if edge weights were between 0 and 1

Graphistry automatically normalizes edge weights in version 2.30.25+

[ ]:
```
g.edges(g._edges.assign(with_negative=\
        g._edges['1_if_neighbor'].apply(lambda v: \
            -1 if v == 0 else 1 )))\
    .bind(edge_weight='1_if_neighbor').settings(url_params={**URL_PARAMS, 'edgeInfluence
↪': 1}).plot(render=True)
```

[ ]:
```
g.bind(edge_weight='100_if_neighbor').settings(url_params={**URL_PARAMS, 'edgeInfluence':
↪ 2}).plot(render=True)
```

[ ]:
```

```

Categorical ring layout tutorial

Graphs where nodes have a categorical attribute may be layed out radially with the new time ring layout.

Example values might be names, IDs, colors, and small multiples.

The tutorial overviews:

- Continuous coloring
- Automated use with smart defaults given just the `ring_col: str` value dimension
- `order: Optional[List[Any]]`: Sort the axis
- `drop_empty`, `combine_unhandled`, `append_unhandled`: Handle missing values and axis
- `r_min`, `r_max`: Control the ring radius ranges
- `axis: Optional[Dict[Any,str]]`: Pass in axis labels
- `format_axis: Callable, format_label: Callable`: Changing the labels
- `reverse: bool`: Reversing the axis

For larger graphs, we also describe automatic GPU acceleration support

Setup

[1]:
```
import os
os.environ['LOG_LEVEL'] = 'INFO'
```

[2]:
```
from typing import Any, Dict, List
import numpy as np
import pandas as pd
import graphistry

graphistry.register(
    api=3,
    username=FILL_ME_IN,
    password=FILL_ME_IN,
    protocol='https',
    server='hub.graphistry.com',
    client_protocol_hostname='https://hub.graphistry.com'
)
```

Data

- Edges: Load a table of IDS network events for our edges
- Nodes: IP addresses, computing for each IP the time of the first and last events it was seen in

[3]:
```
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/honeypot.csv')
df = df.assign(t= pd.Series(pd.to_datetime(df['time(max)'] * 1000000000)))
print(df.dtypes)
```

(continues on next page)

(continued from previous page)

```
print(len(df))
df.sample(3)
```

```
attackerIP            object
victimIP              object
victimPort           float64
vulnName              object
count                  int64
time(max)            float64
time(min)            float64
t             datetime64[ns]
dtype: object
220
```

[3]:
```
         attackerIP      victimIP  victimPort           vulnName  count  \
179   78.140.56.110  172.31.14.66       445.0   MS08067 (NetAPI)     12
79     186.23.87.31  172.31.14.66       445.0   MS08067 (NetAPI)     11
90   188.225.73.153  172.31.14.66       443.0  IIS Vulnerability      1

        time(max)     time(min)                   t
179  1.414243e+09  1.414241e+09 2014-10-25 13:08:50
79   1.420662e+09  1.420661e+09 2015-01-07 20:24:19
90   1.418287e+09  1.418287e+09 2014-12-11 08:42:18
```

[4]:
```
ip_times = pd.concat([
    df[['attackerIP', 't', 'count', 'time(min)', 'vulnName']].rename(columns={'attackerIP
↪': 'ip'}),
    df[['victimIP', 't', 'count', 'time(min)', 'vulnName']].rename(columns={'victimIP':
↪'ip'})
])

def most_frequent(series):
    return series.mode().iloc[0] if not series.mode().empty else None

ip_times = ip_times.groupby('ip').agg({
    't': ['min', 'max'],
    'count': ['sum'],
    'time(min)': ['min'],
    'vulnName': [most_frequent, lambda x: str(list(x.unique()))]
}).reset_index()
ip_times.columns = ['ip', 't_min', 't_max', 'count', 'time_min', 'vuln_top', 'vuln_all']

print(ip_times.dtypes)
print(ip_times.shape)
ip_times.sample(5)
```

```
ip                    object
t_min       datetime64[ns]
t_max       datetime64[ns]
count                int64
time_min           float64
vuln_top            object
vuln_all            object
```

(continues on next page)

(continued from previous page)

```
dtype: object
(203, 7)
```

[4]:
```
                  ip               t_min               t_max  count  \
160   77.232.152.116 2015-02-08 05:48:25 2015-02-09 00:17:13      2
146   49.149.168.197 2014-12-05 10:46:55 2014-12-05 10:46:55      8
42     176.103.22.19 2014-12-06 19:23:06 2014-12-06 19:23:06     13
23    119.157.215.18 2014-12-19 20:50:11 2014-12-19 20:50:11      4
112  220.128.136.237 2014-09-30 08:43:16 2014-09-30 08:43:16      2

         time_min           vuln_top  \
160  1.423375e+09  IIS Vulnerability
146  1.417775e+09   MS08067 (NetAPI)
42   1.417892e+09   MS08067 (NetAPI)
23   1.419021e+09   MS08067 (NetAPI)
112  1.412066e+09   MS08067 (NetAPI)

                                      vuln_all
160  ['IIS Vulnerability', 'MaxDB Vulnerability']
146                          ['MS08067 (NetAPI)']
42                           ['MS08067 (NetAPI)']
23                           ['MS08067 (NetAPI)']
112                          ['MS08067 (NetAPI)']
```

[5]:
```
g = graphistry.edges(df, 'attackerIP', 'victimIP').nodes(ip_times, 'ip')
```

Visualization

Default

The default layout will scan for a numeric column and try to infer reasonable layout settings

[6]:
```
g.ring_categorical_layout('vuln_top').plot(render=False)
```

[6]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=6cf9007b825e47eab9b7855cd407c7d3&
↪type=arrow&viztoken=1c6e0c33-2c29-4a00-870a-1ddeaeb2b623&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721028603&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control axis order

[8]:
```
order = sorted(list(g._nodes['vuln_top'].unique()))

g.ring_categorical_layout(
    ring_col='vuln_top',
    order=order,
    reverse=True
).plot(render=False)
```

[8]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=77130f55096c4a71a4226e4557e1ac6c&
↪type=arrow&viztoken=2c585098-cd4f-43e6-b2a6-8cc0c2097451&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721027155&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Handle missing values and axis labels

When passed in axis labels do not cover all observed values in the data, we can:

- Put all unexpected values in one ring "Other", or a ring per unique value
- Put the new rings before or after the other rings

[10]:
```
order = sorted(list(g._nodes['vuln_top'].unique()))
order = order[:3] + order[6:]
missing_labels = set(g._nodes['vuln_top'].unique()) - set(order)

print('showing', order)
print('combining into Other', missing_labels)

g.ring_categorical_layout(
    ring_col='vuln_top',
    order=order,
    reverse=True,
    combine_unhandled=True,  # put into 1 ring
    append_unhandled=False,  # put after other items
).plot(render=False)
```

```
showing ['DCOM Vulnerability', 'HTTP Vulnerability', 'IIS Vulnerability', 'MaxDB␣
↪Vulnerability', 'SYMANTEC Vulnerability', 'TIVOLI Vulnerability']
combining into Other {'MYDOOM Vulnerability', 'MS04011 (LSASS)', 'MS08067 (NetAPI)'}
```

[10]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=e897ebdfa58e4229afaad3a768400b26&
↪type=arrow&viztoken=79ba2bd2-1ab9-442e-b6aa-534caadebd8a&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721027203&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

[12]:
```
g.ring_categorical_layout(
    ring_col='vuln_top',
    order=order,
    reverse=True,
    combine_unhandled=True,  # put into 1 ring
    append_unhandled=True,  # put after other items
).plot(render=False)
```

[12]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=9e63c4b30bde4555a314fc0e4953553e&
↪type=arrow&viztoken=e5e38bef-3992-4cd4-b521-b7f632a671c4&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721027264&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

When axis cover data not seen in the data, we can drop it (default) or keep it as an unpopulated ring

[15]:
```
order_excessive = list(g._nodes['vuln_top'].unique()) + ['a value that never occurs']
g.ring_categorical_layout(
    ring_col='vuln_top',
    order=order_excessive,
```

(continues on next page)

(continued from previous page)

```
    drop_empty=False,
).plot(render=False)
```

[15]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=e562ab9d9df94593be7e75632d10989e&
↪type=arrow&viztoken=5c835399-d4c3-45b5-821b-cd37ff9f64f4&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721027656&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control sizes

- Control the radius of the first, last rings via `min_r`, `max_r`

[16]:
```
g.ring_categorical_layout(
    ring_col='vuln_top',
    min_r=500,
    max_r=1000,
).plot(render=False)
```

[16]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=18611f4354ae4c68b87b073d1e2fc916&
↪type=arrow&viztoken=2f2844e2-a154-4e29-9ff8-0008b2dbe7d7&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721027856&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control axis labels

Label each axis as `"ring: <lower case value>"`

[20]:
```
axis: Dict[Any, str] = {
    v: f'ring: {v.lower()}'
    for v in list(g._nodes['vuln_top'].unique())
}
print('axis', axis)

g.ring_categorical_layout(
    ring_col='vuln_top',
    axis=axis
).plot(render=False)
```

```
axis {'MS08067 (NetAPI)': 'ring: ms08067 (netapi)', 'MS04011 (LSASS)': 'ring: ms04011␣
↪(lsass)', 'MaxDB Vulnerability': 'ring: maxdb vulnerability', 'IIS Vulnerability':
↪'ring: iis vulnerability', 'SYMANTEC Vulnerability': 'ring: symantec vulnerability',
↪'MYDOOM Vulnerability': 'ring: mydoom vulnerability', 'TIVOLI Vulnerability': 'ring:␣
↪tivoli vulnerability', 'DCOM Vulnerability': 'ring: dcom vulnerability', 'HTTP␣
↪Vulnerability': 'ring: http vulnerability'}
```

[20]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=27e512bb757c41a4ae8f1037bd934969&
↪type=arrow&viztoken=bec9ee71-fa6b-4122-b553-f20d41899d6f&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721027971&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Compute a custom label based on the value

[23]:
```
def axis_to_title(v: str, step: int, radius: float) -> str:
    lbl = f'ring: {v.lower()}'
```

(continues on next page)

(continued from previous page)

```
    return lbl

g.ring_categorical_layout(
    ring_col='vuln_top',
    format_labels=axis_to_title
).plot(render=False)
```

[23]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=6b910e61013d40f5a6238ab2acb71f00&
↪type=arrow&viztoken=b678d9bb-8d6b-49bb-b509-2af9b45d6de0&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721028107&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control more aspects of the axis, like border style

[24]:
```
def fancy_axis_transform(axis: List[Dict]) -> List[Dict]:
    """
      - same radii
      - add "Ring ..." to labels
      - color radial axis based on ring number
          * ring 3: internal (blue axis style)
          * ring 6: external (orange axis style)
          * other rings: space (default gray axis style)
    """
    out = []
    print('sample input axis[0]:', axis[0])
    for i, ring in enumerate(axis):
        out.append({
            'r': ring['r'],
            'label': f'Ring {ring["label"]}',
            'internal': i == 3,  # blue
            'external': i == 6,  # orange
            'space': i != 3 and i != 6  # gray
        })
    print('sample output axis[0]:', out[0])
    return out

g.ring_categorical_layout(
    ring_col='vuln_top',
    min_r=400,
    max_r=1000,
    format_axis=fancy_axis_transform
).plot(render=False)
```

```
sample input axis[0]: {'label': 'MS08067 (NetAPI)', 'r': 400.0, 'internal': True}
sample output axis[0]: {'r': 400.0, 'label': 'Ring MS08067 (NetAPI)', 'internal': False,
↪'external': False, 'space': True}
```

[24]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=14effed027694ed59e2ac56114b23edf&
↪type=arrow&viztoken=8c566e91-36ae-4ea2-b49d-0598260edbea&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721028166&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

GPU Acceleration

For larger graphs, automatic GPU acceleration triggers when `g._nodes` is a `cudf.DataFrame`.

To ensure GPU acceleration is used, set `engine='cudf'`

[7]:
```
import cudf

(g
 .nodes(cudf.from_pandas(g._nodes))
 .ring_categorical_layout('vuln_top', engine='cudf')
).plot(render=False)
```

[7]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=a337d895c28e4fd2b84bd57475942c5a&
↪type=arrow&viztoken=427551fa-b454-4325-b6df-fc038fc685c2&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721028625&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Continuous layout tutorial

Graphs where nodes have a continuous attribute may be layed out radially with the new continuous ring layout.

Examples of continuous values are ranges like weight and price.

The tutorial overviews:

- Continuous coloring
- Automated use with smart defaults
- `ring_col: str`: Specifying the value dimension
- `reverse: bool`: Reversing the axis
- `v_start`, `v_end`, `r_min`, `r_max`: Control the value and ring radius ranges
- `normalize_ring_col`: Whether to normalize values or pass them through
- `num_rings: int`: Picking the number of rings
- `ring_step: float`: Changing the ring step size
- `axis: List[str] | Dict[float,str]`: Pass in axis labels
- `format_axis: Callable, format_label: Callable`: Changing the labels

For larger graphs, we also describe automatic GPU acceleration support

Setup

[1]:
```
import os
os.environ['LOG_LEVEL'] = 'INFO'
```

[2]:
```
from typing import Dict, List
import numpy as np
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import graphistry

graphistry.register(
    api=3,
    username=FILL_ME_IN,
    password=FILL_ME_IN,
    protocol='https',
    server='hub.graphistry.com',
    client_protocol_hostname='https://hub.graphistry.com'
)
```

Data

- Edges: Load a table of IDS network events for our edges
- Nodes: IP addresses, computing for each IP the time of the first and last events it was seen in

[3]:
```
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/honeypot.csv')
df = df.assign(t= pd.Series(pd.to_datetime(df['time(max)'] * 1000000000)))
print(df.dtypes)
print(len(df))
df.sample(3)
```

```
attackerIP            object
victimIP              object
victimPort           float64
vulnName              object
count                  int64
time(max)            float64
time(min)            float64
t             datetime64[ns]
dtype: object
220
```

[3]:

	attackerIP	victimIP	victimPort	vulnName	count	time(max)	time(min)	t
144	41.227.194.80	172.31.14.66	445.0	MS08067 (NetAPI)	1	1.417556e+09	1.417556e+09	2014-12-02 21:28:47
116	201.221.119.171	172.31.14.66	445.0	MS08067 (NetAPI)	7	1.414022e+09	1.414021e+09	2014-10-22 23:49:50
31	124.123.70.99	172.31.14.66	445.0	MS08067 (NetAPI)	3	1.413567e+09	1.413566e+09	2014-10-17 17:26:09

[4]:
```
ip_times = pd.concat([
    df[['attackerIP', 't', 'count', 'time(min)']].rename(columns={'attackerIP': 'ip'}),
    df[['victimIP', 't', 'count', 'time(min)']].rename(columns={'victimIP': 'ip'})
])
ip_times = ip_times.groupby('ip').agg({
    't': ['min', 'max'],
    'count': ['sum'],
```

(continues on next page)

(continued from previous page)

```
    'time(min)': ['min']
}).reset_index()
ip_times.columns = ['ip', 't_min', 't_max', 'count', 'time_min']

print(ip_times.dtypes)
print(ip_times.shape)
ip_times.sample(3)
```

```
ip                  object
t_min       datetime64[ns]
t_max       datetime64[ns]
count                int64
time_min           float64
dtype: object
(203, 5)
```

[4]:
```
                ip               t_min               t_max  count  \
106    212.0.209.7 2014-12-31 03:54:42 2014-12-31 03:54:42      1
40    172.31.14.66 2014-09-30 08:43:16 2015-03-03 00:54:49   1321
30   124.123.70.99 2014-10-17 17:26:09 2014-10-17 17:26:09      3

         time_min
106  1.419998e+09
40   1.412066e+09
30   1.413566e+09
```

[5]:
```
g = graphistry.edges(df, 'attackerIP', 'victimIP').nodes(ip_times, 'ip')
```

Visualization

Continuous coloring

Coloring nodes and edges by the value domain can help visual interpretation, so we encode smallest as cold (blue) and biggest as hot (red):

[6]:
```
g = g.encode_point_color('count', ['blue', 'blue', 'blue', 'yellow', 'yellow', 'yellow',
↪'red'], as_continuous=True)
```

Default

The default layout will scan for a numeric column and try to infer reasonable layout settings

[8]:
```
g.ring_continuous_layout().plot(render=False)
```

[8]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=363ec1cb22ef4e9dafb474d1add25ed6&
↪type=arrow&viztoken=58585ee0-4a23-4765-81b2-457e7c29380f&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721015632&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Pick the value column and reverse direction

[10]:
```
g.ring_continuous_layout(
    ring_col='count',
    reverse=True
).plot(render=False)
```

[10]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=45d4daf6ce4347818e005e05c218e9f1&
↪type=arrow&viztoken=2b5b95bf-290d-4fa5-a221-50fa53de00d4&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721015645&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control the number of rings

- Control the number of rings via `num_rings`
- Control which values map to the first, last rings via `v_start` and `v_end` in terms of ring value column `g._nodes['count']`
  - Let the layout algorithm determine the distance between rings based on `num_rings`, `v_start`, and `v_end`
- Control the radius of the first, last rings via `min_r`, `max_r`

[13]:
```
g.ring_continuous_layout(
    ring_col='count',
    min_r=500,
    max_r=1000,
    v_start=100,
    v_end=1400,
    num_rings=13
).plot(render=False)
```

[13]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=c864818290f64e22a5333c5d4567824f&
↪type=arrow&viztoken=feabda1a-8411-4a68-9c93-206182f9a9ca&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721015794&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control sizes

- Control which values map to the first, last rings via `v_start` andn `v_end`
- Control the ring step size via `v_step` (in terms of input column `g._nodes['count']`)
  - Let the layout algorithm determine the `num_rings` based on `v_start`, `v_end`, and `v_step`
- Control the radius of the first, last rings via `min_r`, `max_r`

[14]:
```
import math

def round_up_to_nearest_100(x: float) -> int:
    return math.ceil(x / 100) * 100

g.ring_continuous_layout(
    ring_col='count',
    min_r=500,
```

(continues on next page)

(continued from previous page)

```
    max_r=1000,
    v_start=100,
    v_end=round_up_to_nearest_100(g._nodes['count'].max()),
    v_step=100
).plot(render=False)
```

[14]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=78cd72af75b74e5280d3c644be9aa768&
↪type=arrow&viztoken=9ac50462-1d05-487c-bec7-303d24a60847&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721015950&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control axis labels

Pass in a value for each radial axis

[18]:
```
axis: List[str] = [
    f'ring: {v}'
    for v in
    ['a', 'b', 'c', 'd', 'e', 'f']  # one more than rings
]
print('axis', axis)

g.ring_continuous_layout(
    ring_col='count',
    num_rings=5,
    axis=axis
).plot(render=False)
```

```
axis ['ring: a', 'ring: b', 'ring: c', 'ring: d', 'ring: e', 'ring: f']
```

[18]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=fd1009384627494a8d983bc7df79d4c3&
↪type=arrow&viztoken=9e1032cd-b2c7-47c6-b912-32b6d7b18a78&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721016118&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Compute a custom label based on the value

[22]:
```
def axis_to_title(value: float, step: int, ring_width: float) -> str:
    lbl = int(value)
    return f'Count: {lbl}'

g.ring_continuous_layout(
    ring_col='count',
    num_rings=5,
    format_labels=axis_to_title
).plot(render=False)
```

[22]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=b76a773409c9487d913995b9586460d0&
↪type=arrow&viztoken=fbfedc8c-3d15-43ea-b2e4-5e19743e5856&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721016584&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Control more aspects of the axis, like border style

[30]:
```
def fancy_axis_transform(axis: List[Dict]) -> List[Dict]:
    """
```

(continues on next page)

(continued from previous page)

```
      - same radii
      - add "Ring ..." to labels
      - color radial axis based on ring number
          * ring 3: internal (blue axis style)
          * ring 6: external (orange axis style)
          * other rings: space (default gray axis style)
    """
    out = []
    print('sample input axis[0]:', axis[0])
    for i, ring in enumerate(axis):
        out.append({
            'r': ring['r'],
            'label': f'Ring {ring["label"]}',
            'internal': i == 3,  # blue
            'external': i == 6,  # orange
            'space': i != 3 and i != 6  # gray
        })
    print('sample output axis[0]:', out[0])
    return out

g.ring_continuous_layout(
    ring_col='count',
    num_rings=10,
    min_r=500,
    max_r=1000,
    format_axis=fancy_axis_transform
).plot(render=False)
```

```
sample input axis[0]: {'label': '1.0', 'r': 500.0, 'internal': True}
sample output axis[0]: {'r': 500.0, 'label': 'Ring 1.0', 'internal': False, 'external':␣
↪False, 'space': True}
```

[30]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=63980e32d60d4cebb6b5007ae914e586&
↪type=arrow&viztoken=d60a6b8b-958a-4f9a-a6f9-f3be428d390c&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721016963&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

GPU Acceleration

For larger graphs, automatic GPU acceleration triggers when `g._nodes` is a `cudf.DataFrame`.

To ensure GPU acceleration is used, set `engine='cudf'`

[32]:
```
import cudf

(g
 .nodes(cudf.from_pandas(g._nodes))
 .ring_continuous_layout(engine='cudf')
).plot(render=False)
```

[32]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=2731e5857dd541048c3754c902707dcb&
↪type=arrow&viztoken=fee07e68-cf25-4814-9e17-d95bf96b59d4&usertag=6c2f6dc1-pygraphistry-
↪0+unknown&splashAfter=1721017010&info=true&play=0&lockedR=True&bg=%23E2E2E2'
```

Time ring layout tutorial

Graphs where nodes have a time attribute may be layed out radially with the new time ring layout.

The tutorial overviews:

- Temporal coloring
- Automated use with smart defaults
- `time_col: str`: Specifying the time dimension
- `reverse: bool`: Reversing the axis
- `time_unit: TimeUnit`: Changing the ring step time interval
- `num_rings: int`: Picking the number of rings
- `time_start: np.datetime64, time_end: np.datetime64`: Clipping the time interval
- `min_r: float, max_r: float`: Changing chart sizes
- `format_axis: Callable, format_label: Callable`: Changing the labels

For larger graphs, we also describe automatic GPU acceleration support

Setup

[2]:
```
import os
os.environ['LOG_LEVEL'] = 'INFO'
```

[ ]:
```
from typing import Dict, List
import numpy as np
import pandas as pd
import graphistry

graphistry.register(
    api=3,
    username=FILL_ME_IN,
    password=FILL_ME_IN,
    protocol='https',
    server='hub.graphistry.com',
    client_protocol_hostname='https://hub.graphistry.com'
)
```

Data

- Edges: Load a table of IDS network events for our edges
- Nodes: IP addresses, computing for each IP the time of the first and last events it was seen in

[4]:
```
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/honeypot.csv')
df = df.assign(t= pd.Series(pd.to_datetime(df['time(max)'] * 1000000000)))
print(df.dtypes)
```

(continues on next page)

(continued from previous page)

```
print(len(df))
df.sample(3)
```

```
attackerIP            object
victimIP              object
victimPort           float64
vulnName              object
count                  int64
time(max)            float64
time(min)            float64
t              datetime64[ns]
dtype: object
220
```

[4]:

```
          attackerIP      victimIP  victimPort          vulnName  count  \
133       27.51.48.2  172.31.14.66       445.0  MS08067 (NetAPI)     10
120  217.172.247.126  172.31.14.66       139.0  MS08067 (NetAPI)     13
158     46.175.85.19  172.31.14.66       445.0  MS08067 (NetAPI)      8

        time(max)     time(min)                   t
133  1.423648e+09  1.423647e+09 2015-02-11 09:54:42
120  1.424391e+09  1.424389e+09 2015-02-20 00:16:47
158  1.419202e+09  1.419201e+09 2014-12-21 22:48:14
```

[5]:

```
ip_times = pd.concat([
    df[['attackerIP', 't']].rename(columns={'attackerIP': 'ip'}),
    df[['victimIP', 't']].rename(columns={'victimIP': 'ip'})
])
ip_times = ip_times.groupby('ip').agg({'t': ['min', 'max']}).reset_index()
ip_times.columns = ['ip', 't_min', 't_max']

print(ip_times.dtypes)
print(len(ip_times))
ip_times.sample(3)
```

```
ip              object
t_min    datetime64[ns]
t_max    datetime64[ns]
dtype: object
203
```

[5]:

```
                ip               t_min               t_max
5   106.201.227.134 2014-11-21 14:38:07 2014-11-21 14:38:07
25  122.121.202.157 2015-02-10 23:53:52 2015-02-10 23:53:52
59   179.25.208.154 2015-01-05 23:22:45 2015-01-05 23:22:45
```

[6]:

```
g = graphistry.edges(df, 'attackerIP', 'victimIP').nodes(ip_times, 'ip')
```

Visualization

Temporal coloring

Coloring nodes and edges by time can help visual interpretation, so we encode old as cold (blue) and new as hot (red):

[7]:
```
g = g.encode_point_color('t_min', ['blue', 'yellow', 'red'], as_continuous=True)
```

Default

The default layout will scan for a time column and try to infer reasonable layout settings

[8]:
```
g.time_ring_layout().plot(render=False)
```

[8]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=df1e3c96e94b4770adb7bd3195f3c5e4&
↪type=arrow&viztoken=2512035f-15a0-4284-aed4-8418e1152826&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920413&info=true&play=2000&lockedR=True&bg=%23E2E2E2'
```

Pick the time column and reverse direction

[9]:
```
g.time_ring_layout(
    time_col='t_min',
    reverse=True
).plot(render=False)
```

[9]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=7455256f9d5446e1bf379e9c751be4f2&
↪type=arrow&viztoken=091ebc73-a500-4e71-a162-d1f4c5d57c6e&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920419&info=true&play=2000&lockedR=True&bg=%23E2E2E2'
```

Use alternate units

Available units:

- 's': seconds
- 'm': minutes
- 'h': hours
- 'D': days
- 'W': weeks
- 'M': months
- 'Y': years
- 'C': centuries

[10]:
```
g.time_ring_layout(
    time_col='t_min',
    time_unit='W',
```

(continues on next page)

(continued from previous page)

```
    num_rings=30
).plot(render=False)
```

[10]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=97ecd3d87acf406f9eba87bfec53e634&
↪type=arrow&viztoken=e1553a4c-9e4f-4771-bb21-90a262044c14&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920423&info=true&play=2000&lockedR=True&bg=%23E2E2E2E2'
```

Control the ring size, radius, and time interval

[11]:
```
g.time_ring_layout(
    time_unit='Y',
    num_rings=2,
    play_ms=0,
    min_r=700,
    max_r=1000
).plot(render=False)
```

[11]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=e96440fb19a64bddb8c29e9a7cc80ec4&
↪type=arrow&viztoken=e9c0203e-1d7d-42ff-bb9e-b5b86075d49d&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920426&info=true&play=0&lockedR=True&bg=%23E2E2E2E2'
```

[12]:
```
g.time_ring_layout(
    time_unit='Y',
    time_start=np.datetime64('2013'),
    play_ms=0,
    min_r=700,
    max_r=1000
).plot(render=False)
```

[12]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=30daf0734b61454f80c89606af5ae24a&
↪type=arrow&viztoken=b8adf44c-2942-49de-a622-bc858031a169&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920428&info=true&play=0&lockedR=True&bg=%23E2E2E2E2'
```

Control labels

[13]:
```
def custom_label(time: np.datetime64, ring: int, step: np.timedelta64) -> str:
    date_str = pd.Timestamp(time).strftime('%Y-%m-%d')
    return f'Ring {ring}: {date_str}'

g.time_ring_layout(
    format_label=custom_label
).plot(render=False)
```

[13]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=88adff411bb04c95843c5eee61ec1b97&
↪type=arrow&viztoken=18f922c2-bbd3-4972-8f5b-1d6d93ea31f3&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920432&info=true&play=2000&lockedR=True&bg=%23E2E2E2E2'
```

[14]:
```
def custom_axis(axis: List[Dict]) -> List[Dict]:
    """
    Axis with reversed label text
```

(continues on next page)

(continued from previous page)

```
    """
    print('axis item keys', {k: type(axis[0][k]) for k in axis[0].keys()})
    return [
        {**o, 'label': o['label'][::-1]}
        for o in axis
    ]

g.time_ring_layout(
    format_axis=custom_axis
).plot(render=False)
```

```
axis item keys {'label': <class 'str'>, 'r': <class 'numpy.float64'>, 'internal': <class
↪'bool'>}
```

[14]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=1403b7ba4c4b4678904546555d656060&
↪type=arrow&viztoken=60e14303-a0bb-4f97-a76a-a68f5ee1bcab&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920438&info=true&play=2000&lockedR=True&bg=%23E2E2E2'
```

GPU Acceleration

For larger graphs, automatic GPU acceleration triggers when `g._nodes` is a `cudf.DataFrame`.

To ensure GPU acceleration is used, set `engine=

[15]:
```
import cudf

(g
 .nodes(cudf.from_pandas(g._nodes))
 .time_ring_layout()
).plot(render=False)
```

[15]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=d9e58ab3967f4003b7ef292406ba1a47&
↪type=arrow&viztoken=80d707a5-cb03-48f8-897f-5e56e127630b&usertag=1c11b3a4-pygraphistry-
↪0+unknown&splashAfter=1720920444&info=true&play=2000&lockedR=True&bg=%23E2E2E2'
```

GPU-Accelerated Group-in-a-Box Layout for Analyzing Large Graphs

Visualizing interactions within large, complex datasets can be challenging. The Group-in-a-Box layout simplifies this by arranging communities into grids, making it easier to analyze relationships within and between groups. It's especially useful for deeper insights in social media analysis.

PyGraphistry extends the https://www.cs.umd.edu/users/ben/papers/Rodrigues2011Group.pdf with high-speed performance, flexible customization, and integration into the PyGraphistry ecosystem.

Key Benefits

- **Faster Insights**: GPU support accelerates commercial workloads, reducing runtime for a 3M+ edge social network from 18 minutes to just 26 seconds. CPU mode likewise benefits from algorithmic and vector optimizations. The result is rapid analysis iterations that unblocks workflows and achieves previously out-of-reach results.
- **Customizable Layouts**:
  - **Flexible Partitioning**: Choose from built-in algorithms or custom keys to align partitions with data structures like regional clusters or demographics.
  - **Adaptive Community Layouts**: Focus on tightly connected groups or highlight outliers to uncover hidden patterns.
- **Clear Visualization of Isolated Nodes**: The PyGraphistry variant additionally arranges isolated nodes in circles around their connected counterparts to handle the common case of noisey nodes within a partition.

Tutorial

Follow this tutorial to master PyGraphistry's Group-in-a-Box layout:

- Load a 45,000-edge blockchain transaction graph
- Layout in seconds on CPU or 1 second on GPU
- Customize partitioning and group layouts.

[31]:
```
import pandas as pd
import graphistry
graphistry.__version__
```

[31]:
```
'0+unknown'
```

[2]:
```
# API key page (free GPU account): https://hub.graphistry.com/users/personal/key/
graphistry.register(
    api=3,
    personal_key_id=FILL_ME_IN,
    personal_key_secret=FILL_ME_IN
)
```

Data

[12]:
```
e_df = pd.read_csv(
    'https://raw.githubusercontent.com/graphistry/pygraphistry/refs/heads/master/demos/
↪data/transactions.csv',
)

print(e_df.shape)
e_df.head()
```

```
(45117, 6)
```

[12]:
```
        Amount $          Date                            Destination  \
0    3223.975200  1.385240e+12  84a0b53e1ac008b8dd0fd6212d4b7fa2...
1    3708.021600  1.401500e+12  3b62a891b99969042d4e6ac8158d0a18...
2       2.480000  1.398560e+12  3b62a891b99969042d4e6ac8158d0a18...
3  991986.487600  1.385540e+12  e52baeb69fbd7a24f3ef825bc4e20973...
4     902.063544  1.387850e+12  a209cf1b3dc79338896ffa773b4249ff...

                                Source  \
0  2dd13954e18508bb8b3a41d96a022be9...
1  7c74d3afb41e536e26948a1d2455a7c7...
2  50dced19b8ee41114916bf3ca894f455...
3  4289f81f7ce5dfba9bf6e20794c76a9f...
4  7f0e2244e41718b68e36ed0c810d084b...

                                     Transaction ID  isTainted
0  b6eb8ba20df31fa74fbe7755f58c18f82a599d6bb5fa79...           0
1  60df3c67063e136a0c9715edcd12ae717e6f9ed492afe2...           0
2  a6aafd3d85600844536b8a5f2c255686c33dc4969e68a4...           0
3  1cea981590e8a3ac67ba872f9411412c6b6f4dc7358071...           0
4  a399e3920b1e6a1e487b0559821f823065970499e74cbc...           0
```

[13]:
```
g = graphistry.edges(e_df, 'Source', 'Destination')
```

[14]:
```
g._edges.shape
```

[14]:
```
(45117, 6)
```

[17]:
```
g.materialize_nodes()._nodes.shape
```

[17]:
```
(28832, 1)
```

Regular layout

[18]:
```
g.plot()
```

[18]:
```
<IPython.core.display.HTML object>
```

Group-in-a-box: CPU Mode

Passing in a pandas dataframe defaults to using igraph rf layout (CPU) within each partition. Even in CPU mode, it is significantly faster than published group-in-a-box algorithms through a combination of asymptotic and machine-oriented optimizations.

[19]:
```
g2 = g.group_in_a_box_layout()
```

```
edge index g._edge not set so using edge index as ID; set g._edge via g.edges(), or␣
↪change merge_if_existing to Falseedge index g._edge __edge_index__ missing as␣
↪attribute in ig; using ig edge order for IDsPandas engine detected. FA2 falling back␣
↪to igraph fredge index g._edge not set so using edge index as ID; set g._edge via g.
↪edges(), or change merge_if_existing to Falseedge index g._edge __edge_index__ missing␣
↪as attribute in ig; using ig edge order for IDs
```

[20]:
```
g2.plot()
```

[20]:
```
<IPython.core.display.HTML object>
```

GPU Mode

Switching the input to GPU dataframes automatically transitions execution to GPU mode, which is dramatically faster.

The GPU mode defaults to ForceAtlas2, which generally shows a superior layout within a group when zoomed in.

[26]:
```
g_gpu = g.to_cudf()

g2_gpu = g_gpu.group_in_a_box_layout()
```

[27]:
```
g2_gpu.plot()
```

[27]:
```
<IPython.core.display.HTML object>
```

Configure: Precomputed partition and alternate layout

- Use an existing node attribute to predetermine box membership
- Control the layout algorithm and its parameters

[28]:
```
g_louvain = g.to_cudf().compute_cugraph('louvain', directed=False)
assert 'louvain' in g_louvain._nodes
```

[29]:
```
from graphistry.plugins.igraph import layout_algs as igraph_layouts
from graphistry.plugins.cugraph import layout_algs as cugraph_layouts

{
    'igraph_layout_algs': ', '.join(igraph_layouts),
    'cugraph_layout_algs': ', '.join(cugraph_layouts)
}
```

[29]:
```
{'igraph_layout_algs': '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',
 'cugraph_layout_algs': 'force_atlas2'}
```

[30]:
```
(g_louvain
     .group_in_a_box_layout(
         partition_key='louvain',
         layout_alg='force_atlas2',
         layout_params={
             'lin_log_mode': True
         }
```

(continues on next page)

(continued from previous page)

```
    )
).plot()
```

[30]:
```
<IPython.core.display.HTML object>
```

[ ]:

[ ]:

Modularity weighted layout

When community labels are known, modularity weighting helps force the layout to better hightlight the community structure. The layout algorithm will weight edges based on whether they are same-community vs cross-community. If no community labels are provided, default to using Louivain.

[ ]:
```
# ! pip install -q graphistry igraph
```

[24]:
```
import graphistry
import pandas as pd
graphistry.register(api=3, username=FILL_ME_IN, password=FILL_ME_IN)
```

[4]:
```
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/transactions.csv')
g = graphistry.edges(df, 'Destination', 'Source')
```

[27]:
```
# Before
g.compute_igraph('community_multilevel', directed=False).plot()
```

```
WARNING:graphistry.plugins.igraph:edge index g._edge not set so using edge index as ID;␣
↪set g._edge via g.edges(), or change merge_if_existing to FalseWARNING:graphistry.
↪plugins.igraph:edge index g._edge __edge_index__ missing as attribute in ig; using ig␣
↪edge order for IDs
```

[27]:
```
<IPython.core.display.HTML object>
```

[19]:
```
# After
g2 = g.modularity_weighted_layout(g)
```

```
WARNING:graphistry.plugins.igraph:edge index g._edge not set so using edge index as ID;␣
↪set g._edge via g.edges(), or change merge_if_existing to FalseWARNING:graphistry.
↪plugins.igraph:edge index g._edge __edge_index__ missing as attribute in ig; using ig␣
↪edge order for IDs
```

[21]:
```
g2.plot()
```

[21]:
```
<IPython.core.display.HTML object>
```

[ ]:

Layered Hierarchical Layout

Note: We strongly recommend considering alternate layouts like the ring layouts (pygraphistry) and the igraph plugin's dot engine

[ ]:
```
from graphistry.layout.sugiyama import SugiyamaLayout
from graphistry.layout.graph import Graph, Vertex, Edge
import pandas as pd
import networkx as nx
import matplotlib
import matplotlib.pyplot as plt
```

[ ]:
```
def from_networkx(nxg):
    """
        Converts a networkx graph to a sugiyama graph.
    """
    vertices = []
    data_to_v = {}
    for x in nxg.nodes():
        vertex = Vertex(x)
        vertices.append(vertex)
        data_to_v[x] = vertex
    E = [Edge(data_to_v[xy[0]], data_to_v[xy[1]], data = xy) for xy in nxg.edges()]
    g = Graph(vertices, E)
    return g


def to_networkx(g):
    """
        Converts a sugiyama graph to a networkx graph.
    """
    from networkx import MultiDiGraph

    nxg = MultiDiGraph()
    for v in g.vertices():
        nxg.add_node(v.data)
    for e in g.edges():
        # todo: this leads to issues when the data is more than an id
        nxg.add_edge(e.v[0].data, e.v[1].data)
    return nxg


def draw_graph(g, layout_direction = 0, source_column = "source", target_column = "target
↪", root=None):
    """
    Renders the given graph after applying the layered layout.
    :param g: Graphistry Graph or NetworkX Graph.
    """
    if isinstance(g, nx.Graph):
        gg = from_networkx(g)
        nxg = g
    elif isinstance(g, Graph):
```

(continues on next page)

(continued from previous page)

```
        gg = g
        nxg = to_networkx(g)
    elif isinstance(g, pd.DataFrame):
        gg = SugiyamaLayout.graph_from_pandas(g, source_column = source_column, target_
↪column = target_column)
        nxg = to_networkx(gg)
    else:
        raise ValueError
    # apply layout
    positions = SugiyamaLayout.arrange(gg, layout_direction = layout_direction,␣
↪root=root)
    nx.draw(nxg, pos = positions, with_labels = True, verticalalignment = 'bottom',␣
↪arrowsize = 3, horizontalalignment = "left", font_size = 20)
    plt.show()


def scatter_graph(g, root=None):
    """
    Renders the given graph as a scatter plot after applying the layered layout.
    :param g: Graphistry Graph or NetworkX Graph.
    """
    if isinstance(g, nx.Graph):
        gg = from_networkx(g)
        nxg = g
    elif isinstance(g, Graph):
        gg = g
        nxg = to_networkx(g)
    # apply layout
    coords = list(SugiyamaLayout.arrange(gg, root=root).values())
    x = [c[0] for c in coords]
    y = [c[1] for c in coords]
    fig, ax = plt.subplots()
    ax.scatter(x, y)

    for i, v in enumerate(gg.vertices()):
        ax.annotate(v.data, (x[i], y[i]))

    plt.axis('off')
    plt.show()


def arrange(g, layout_direction = 0, source_column = "source", target_column = "target",␣
↪topological_coordinates = False):
    """
    Returns the positions of the given graph after applying the layered layout.
    :param g: Graphistry Graph, Pandas frame or NetworkX Graph.
    """
    if isinstance(g, nx.Graph):
        gg = from_networkx(g)
        nxg = g
    elif isinstance(g, Graph):
        gg = g
```

(continues on next page)

(continued from previous page)

```
        nxg = to_networkx(g)
    elif isinstance(g, pd.DataFrame):
        gg = SugiyamaLayout.graph_from_pandas(g, source_column = source_column, target_
↪column = target_column)
        nxg = to_networkx(gg)
    else:
        raise ValueError
    # apply layout
    positions = SugiyamaLayout.arrange(gg, layout_direction = layout_direction,␣
↪topological_coordinates = topological_coordinates)
    return positions
```

Explicit Simple Graph

The `Graph` object can be used to create an explicit graph:

[ ]:
```
matplotlib.rc('figure', figsize = [8, 5])
g = Graph()

bosons = Vertex("Boson")
higgs = Vertex("Higgs")
pions = Vertex("Pions")
kaons = Vertex("Kaons")
hadrons = Vertex("Hadrons")

e1 = Edge(bosons, higgs)
e2 = Edge(bosons, kaons)
e3 = Edge(bosons, pions)
e4 = Edge(pions, hadrons)
e5 = Edge(kaons, hadrons)

g.add_edges([e1, e2, e3, e4, e5])
scatter_graph(g)
```

Pandas Graph

[ ]:
```
g = nx.generators.balanced_tree(2, 3)
df = nx.to_pandas_edgelist(g, "source", "target")
df.head()
```

[ ]:
```
matplotlib.rc('figure', figsize = [5, 5])
draw_graph(df, 3)
```

You can set the root like so

[ ]:
```
matplotlib.rc('figure', figsize = [5, 5])
draw_graph(df, 3, root=[3,12])
```

Trees

A genuine tree will be arranged as expected:

[ ]:
```
matplotlib.rc('figure', figsize = [120, 30])
g = nx.generators.balanced_tree(5, 3)
draw_graph(g, 2)
```

Real-world Graphs

https://en.wikipedia.org/wiki/Barabási–Albert_model represent scale-free networks mimicing biological and other realworld netowrks:

[ ]:
```
matplotlib.rc('figure', figsize = [120, 90])
g = nx.generators.barabasi_albert_graph(500, 3)
draw_graph(g)
```

Layout Direction

- 0: top-to-bottom
- 1: right-to-left
- 2: bottom-to-top
- 3: left-to-right

[ ]:
```
matplotlib.rc('figure', figsize = [10, 5])
g = nx.generators.balanced_tree(3, 2)
draw_graph(g, layout_direction = 2)
```

Topological coordinates

Instead of absolute coordinates you can also request the topological coordinates which correspond to the layer index for the vertical axis and a value in the unit interval for the horizontal axis. Multiplying these values with an actual total width and height results in coordinates within the given (width, height) rectangle. Note that the layering is according to the standard coordinate system, ie. upwards and to the right. When setting the layout direction to horizontal (layout_direction equal to 1 or 3), the first coordinate is the layer index and the second will be in the unit interval.

[ ]:
```
g = nx.from_edgelist([(1,2),(1,3),(3,4)])
positions = arrange(g, topological_coordinates=True, layout_direction=0)
print(positions)
```

The topological coordinates can be used as-is with NetworX as well:

[ ]:
```
nx.draw(g, pos = positions, with_labels = True, verticalalignment = 'bottom', arrowsize␣
↪= 3, horizontalalignment = "left", font_size = 20)
```

Complete Graph

The layered layout attempts to minimize crossings but with something like a complete graph you have an extreme example where, no matter how many times the algorithm tries to adjust, the crossings persist.

[ ]:
```
matplotlib.rc('figure', figsize = [120, 90])
g = nx.generators.complete_graph(10)
draw_graph(g,root=4)
```

Only Positions

You can fetch the positions of the nodes without the visualization via the `arrange` method. You can also just plot the points without the edges like so:

[ ]:
```
matplotlib.rc('figure', figsize = [20, 30])
g = nx.generators.random_lobster(100, 0.3, 0.3)
scatter_graph(g)
```

Watts Strogatz

The https://en.wikipedia.org/wiki/Watts–Strogatz_model model is another kind of real-world graph exhibiting the so-called small world phenomenon:

[ ]:
```
matplotlib.rc('figure', figsize = [120, 70])
g = nx.generators.connected_watts_strogatz_graph(1000, 2, 0.3)
draw_graph(g)
```

[ ]:
```

```

Demo: Externally Defined Layouts

This notebook computes a layout using NetworkX and sends it to Graphistry for subsequent interactive analysis.

[1]:
```
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

import networkx as nx
import pandas as pd
```

Make a NetworkX Graph

[2]:
```
num_nodes = 1000
G=nx.Graph()
G.add_nodes_from(range(num_nodes))
Edges = [(x, (x * 2) % (num_nodes/5)) for x in range(num_nodes)] + [(x, (x + 1) % 20)␣
↪for x in range(20)]
G.add_edges_from(Edges)
G
```

[2]:
```
<networkx.classes.graph.Graph at 0x10de7e150>
```

Use a NetworkX layout

[3]:
```
%time
pos=nx.fruchterman_reingold_layout(G)
pos[0]
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 9.06 µs
```

[3]:
```
array([0.3766924 , 0.36150154], dtype=float32)
```

Combine into node, edge dataframes

[4]:
```
def pos2df (pos):
    nodes = pd.DataFrame({'key': pos.keys(), 'pos_0': [pos[k][0] for k in pos.keys()],
↪'pos_1': [pos[k][1] for k in pos.keys()]})
    nodes.columns = ['key', 'x', 'y']
    return nodes
```

[5]:
```
nodes = pos2df(pos)
nodes[:3]
```

[5]:

	key	x	y
0	0	0.376692	0.361502
1	1	0.483908	0.427637
2	2	0.588375	0.491546

[6]:
```
edges = pd.DataFrame(Edges)
edges.columns = ['src', 'dst']
edges[:3]
```

[6]:

	src	dst
0	0	0
1	1	2
2	2	4

Autolayout mode

[7]:
```
g = graphistry.nodes(nodes).edges(edges).bind(source='src', destination='dst', node='key
↪')
g.plot()
```

[7]:
```
<IPython.core.display.HTML object>
```

Predefined layout mode

As the node table provides "x" and "y" columns, they will be automatically used as starting positions. To prevent the automatic layout algorithm from moving the nodes on load, we also set the URL parameter "play" to 0 seconds. (Both settings will likely change.)

[8]:
```
g2 = g.settings(url_params={'play':0})
g.plot()
```

[8]:
```
<IPython.core.display.HTML object>
```

For fun, here's a circular layout

[9]:
```
pos2=nx.circular_layout(G, scale=100)
nodes2 = pos2df(pos2)
g2.nodes(nodes2).plot()
```

[9]:
```
<IPython.core.display.HTML object>
```

[ ]:
```

```

How to pass in layout positions to the visualization

1. Add x, y columns to the node table
2. Add the url parameter "play=0" to prevent the page load to run clustering.

[1]:
```
import pandas as pd
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

1. Nodes should have columns "x", "y"

[2]:
```
edges = pd.DataFrame({'s': [0,1,2, 3], 'd': [1,2,3, 0]})
nodes = pd.DataFrame({'n': [0,1,2, 3], 'x': [0, 0, 1, 1], 'y': [0, 5, 5, 0]})
```

2. The bindings should include ".settings(url_params={'play': 0})"

[3]:
```
graphistry\
    .settings(url_params={'play': 0})\
    .nodes(nodes).edges(edges)\
    .bind(source='s', destination='d', node='n')\
    .plot()
```

[3]:
```
<IPython.core.display.HTML object>
```

[ ]:
```

```

10.8.2.5 Accounts and Sharing

Sharing Tutorial: Securely Collaborating in Graphistry

Investigtions are better together. This tutorial walks through the new PyGraphistry method `.privacy()`, which enables API control of the new sharing features

We walk through:

- Switching between organization uploads and personal accounts
- Global defaults for `graphistry.privacy(mode='private', ...)`
- Compositional per-visualization settings via `g.privacy(...)`
- Inviting and notifying via `privacy(invited_users=[{...])`

Setup

You need pygraphistry 0.20.0+ for a corresponding Graphistry server (2.37.20+)

[1]:
```
#! pip install --user -q graphistry pandas
```

[2]:
```
import graphistry, pandas as pd
graphistry.__version__
```

[2]:
```
'0.20.0'
```

[3]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Optional: Share within an organization

When you login, optional parameter "org_name" specifies an organization to upload into instead of your personal account.

Subsequent sections show further attenuating how the data may be shared through your organization.

[ ]:
```
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com', org_name="my-org")
```

[4]:
```
#demo data
g = graphistry.edges(pd.DataFrame({
    's': ['a', 'b', 'c'],
    'd': ['b', 'c', 'a'],
    'v': [1, 1, 2]
}), 's', 'd')

g = g.settings(url_params={'play': 0})
```

Safe default: Unlisted & owner-editable

When creating a plot, Graphistry creates a dedicated URL with the following rules:

- Viewing: Unlisted - Only those given the link can access it
- Editing: Owner-only

The URL is unguessable, and the only webpage it is listed at is the creator's private gallery: https://hub.graphistry.com/datasets/ . That means it is as private as whomever the owner shares the URL with.

[12]:
```
public_url = g.plot(render=False)
```

Switching to fully private by default

Call `graphistry.privacy()` to default to stronger privacy. It sets:

- `mode='private'` - viewing only by owners and invitees (and not other organization members)
- `invited_users=[]` - no invitees by default
- `notify=False` - no email notifications during invitations
- `mode_action='20'` - only used when `mode="organization"`, action for sharing within organization, "10" (view) or "20" (edit)
- `message=''`

By default, this means an explicit personal invitation is necessary for viewing. Subsequent plots in the session will default to this setting.

You can also explicitly set or override those as optional parameters.

[14]:
```
graphistry.privacy()
# or equivaently, graphistry.privacy(mode='private', invited_users=[], notify=False,␣
↪mode_action='20', message='')

owner_only_url = g.plot(render=False)
```

Local overrides

We can locally override settings, such as opting back in to public sharing for some visualizations:

[15]:
```
public_g = g.privacy(mode='public')

public_url1 = public_g.plot(render=False)

#Ex: Inheriting public_g's mode='public'
public_g2 = public_g.name('viz2')
public_url2 = public_g.plot(render=False)

#Ex: Global default was still via .privacy()
still_private_url = g.plot(render=False)
```

Invitations and notifications

As part of the settings, we can permit specific individuals as viewers or editors, and optionally, send them an email notification

[18]:
```
VIEW = '10'
EDIT = '20'

shared_g = g.privacy(
    mode='private',
    notify=True,
    invited_users=[{'email': 'partner1@site1.com', 'action': VIEW},
                   {'email': 'partner2@site2.org', 'action': EDIT}],
    message='Check out this graph!')
shared_url = shared_g.plot(render=False)
```

The options can be configured globally or locally, just as we did with `mode`. For example, we might not want to send emails by default, just on specific plots:

[19]:
```
graphistry.privacy(
    mode='private',
    notify=False,
    invited_users=[{'email': 'partner1@site1.com', 'action': VIEW},
                   {'email': 'partner2@site2.org', 'action': EDIT}])


shared_url = g.plot(render=False)
notified_and_shared_url = g.privacy(notify=True).plot(render=False)
```

Even if we do not explicitly notify recipients, the objects will still appear in their gallery at https://hub.graphistry.com/datasets/

Share within an organization

You can share within an organization. Register within the organization as per above (`register(..., org_name=my_org)`) then set the privacy mode to 'organization':

[ ]:
```
# Share to all members within organization as EDIT mode (default)
VIEW = '10'
EDIT = '20'

shared_org_g = g.privacy(
    mode='organization',
    message='Check out this graph! Only for our organization')
shared_url = shared_g.plot(render=False)
```

[ ]:
```
# Share to all members within organization as VIEW mode
VIEW = '10'
EDIT = '20'

shared_org_g = g.privacy(
    mode='organization',
    mode_action=VIEW,
    message='Check out this graph! Only for our organization')
shared_url = shared_g.plot(render=False)
```

10.8.3 GFQL Graph queries

10.8.3.1 Hop ranges, slices, and labels

GFQL examples for bounded hops, output slices, and labeling, including the same traversal intent often written in Cypher as `(a)-[*2..4]->(b)` after translation into native GFQL.

- Exact and ranged hops (`min_hops`/`max_hops`)
- Post-filtered output slices (`output_min_hops`/`output_max_hops`)
- Optional hop labels on nodes/edges; seeds = starting nodes (label seeds at hop 0 when requested)

Visual of the toy branching chain used below (seed at 'a').

[1]:
```
import pandas as pd
import graphistry
from graphistry.compute.ast import is_in, n, e_forward

edges = pd.DataFrame({
    's': ['a', 'b1', 'c1', 'd1', 'a', 'b2'],
    'd': ['b1', 'c1', 'd1', 'e1', 'b2', 'c2']
})

nodes = pd.DataFrame({'id': ['a', 'b1', 'c1', 'd1', 'e1', 'b2', 'c2']})

g = graphistry.edges(edges, 's', 'd').nodes(nodes, 'id')
seed_ids = ['a']
```

(continues on next page)

(continued from previous page)

```
print('Edges:')
print(edges)
print()
print('Nodes:')
print(nodes)
print()
print('Seed ids:', seed_ids)
```

```
Edges:
    s   d
0   a  b1
1  b1  c1
2  c1  d1
3  d1  e1
4   a  b2
5  b2  c2

Nodes:
   id
0   a
1  b1
2  c1
3  d1
4  e1
5  b2
6  c2

Seed ids: ['a']
```

[2]:
```
from html import escape
import pandas as pd

# Fixed layout positions for the toy graph
pos_lookup = {
    'a': (50, 100),
    'b1': (150, 70),
    'c1': (250, 70),
    'd1': (350, 70),
    'e1': (450, 70),
    'b2': (150, 130),
    'c2': (250, 130),
}

_y_min = min(y for _, y in pos_lookup.values())
_y_max = max(y for _, y in pos_lookup.values())

def _flip_y(y):
    return _y_min + _y_max - y

def _format_label(val):
    if pd.isna(val):
```

(continues on next page)

(continued from previous page)

```
        return None
    if isinstance(val, (int, float)) and float(val).is_integer():
        val = int(val)
    return str(val)

def _with_fixed_layout(g_out, label_col=None):
    if g_out._nodes is None or g_out._edges is None:
        return g_out
    node_col = g_out._node
    src_col, dst_col = g_out._source, g_out._destination
    nodes_df = g_out._nodes.copy()
    coords = nodes_df[node_col].map(pos_lookup)
    nodes_df['x'] = coords.map(lambda v: v[0] if v is not None else None)
    nodes_df['y'] = coords.map(lambda v: _flip_y(v[1]) if v is not None else None)
    nodes_df = nodes_df.dropna(subset=['x', 'y'])
    if label_col and label_col in nodes_df.columns:
        label_vals = nodes_df[label_col].map(_format_label)
        labels = []
        for node, label_val in zip(nodes_df[node_col], label_vals):
            if label_val is None:
                labels.append(escape(str(node)))
            else:
                labels.append(f"{escape(str(node))}:{escape(str(label_val))}")
        nodes_df = nodes_df.assign(label=labels)
    node_ids = set(nodes_df[node_col])
    edges_df = g_out._edges
    edges_df = edges_df[edges_df[src_col].isin(node_ids) & edges_df[dst_col].isin(node_
↪ids)]
    return g_out.nodes(nodes_df).edges(edges_df)

def render_static(g_out, label_col=None, title=None):
    g_fixed = _with_fixed_layout(g_out, label_col=label_col)
    graph_attr = None
    if title:
        graph_attr = {
            'label': escape(title),
            'labelloc': 't',
            'fontsize': '14',
        }
    node_attr = {
        'shape': 'circle',
        'style': 'filled',
        'fillcolor': '#eef3ff',
        'color': '#1f4b99',
        'fontcolor': '#0a1a2f',
        'fontsize': '14',
    }
    edge_attr = {
        'color': '#1f4b99',
        'penwidth': '2',
    }
    return g_fixed.plot_static(
```

(continues on next page)

(continued from previous page)

```
        engine='graphviz-svg',
        reuse_layout=True,
        graph_attr=graph_attr,
        node_attr=node_attr,
        edge_attr=edge_attr,
    )

# Fill missing hop labels from edge hops (keeps seeds unlabeled when excluded)
def fill_hops(g_out, label_col='hop', edge_label_col='edge_hop', seeds=None):
    if g_out._nodes is None or g_out._edges is None:
        return g_out
    if label_col not in g_out._nodes.columns or edge_label_col not in g_out._edges.
↪columns:
        return g_out
    nodes_df = g_out._nodes.copy()
    edges_df = g_out._edges
    hop_map = pd.concat([
        edges_df[[g._source, edge_label_col]].rename(columns={g._source: g._node}),
        edges_df[[g._destination, edge_label_col]].rename(columns={g._destination: g._
↪node}),
    ], ignore_index=True, sort=False).groupby(g._node)[edge_label_col].min()
    mask = nodes_df[label_col].isna()
    if seeds is not None:
        mask = mask & ~nodes_df[g._node].isin(seeds)
    if mask.any():
        nodes_df.loc[mask, label_col] = nodes_df.loc[mask, g._node].map(hop_map)
    try:
        nodes_df[label_col] = pd.to_numeric(nodes_df[label_col], errors='coerce').astype(
↪'Int64')
    except Exception:
        pass
    return g_out.nodes(nodes_df)
```

[3]:
```
render_static(g, label_col=None, title='Toy graph layout')
```

[3]:

Exact hops

Exact-hop GFQL edge calls for quick sanity checks before slicing or labeling.

Exactly 1 hop from seed 'a' to its immediate neighbors (no labels).

[4]:
```
# Exactly 1 hop (no labels)
hop_1 = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=1, max_hops=1),
])

print("1-hop nodes from 'a' (first hop):")
```

(continues on next page)

(continued from previous page)

```
print(hop_1._nodes.sort_values(g._node))
print()
print('1-hop edges:')
print(hop_1._edges.sort_values([g._source, g._destination]))
render_static(hop_1, label_col=None, title='1 hop')
```

```
1-hop nodes from 'a' (first hop):
   id  __gfql_output_node_hop__
0   a                      <NA>
1  b1                         1
2  b2                         1

1-hop edges:
   __gfql_output_edge_hop__  s   d
0                         1  a  b1
1                         1  a  b2
```

[4]:

Exactly 3 hops bounded to three steps; default output keeps the earlier hops for context (no labels).

[5]:

```
# Exactly 3 hops (no labels, default keeps path)
hop_3 = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=3, max_hops=3),
])

print("3-hop nodes from 'a' (path to third hop):")
print(hop_3._nodes.sort_values(g._node))
print()
print('3-hop edges:')
print(hop_3._edges.sort_values([g._source, g._destination]))
render_static(hop_3, label_col=None, title='3 hops')
```

```
3-hop nodes from 'a' (path to third hop):
   id  __gfql_output_node_hop__
0   a                      <NA>
1  b1                         1
2  c1                         2
3  d1                         3

3-hop edges:
   __gfql_output_edge_hop__   s   d
0                         1   a  b1
1                         2  b1  c1
2                         3  c1  d1
```

[5]:

Hop ranges

Variable-length traversal with full-path output (keep hops up to the bound).

Range 1..3 hops from 'a' (unlabeled), expressing the same traversal intent as a Cypher pattern like `(a)-[*1..3]->(?)` after translation into native GFQL.

[6]:
```
# Range 1..3 hops (combined)
hop_range = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=1, max_hops=3),
])

print("Nodes within 1..3 hops from 'a':")
print(hop_range._nodes.sort_values(g._node))
print()
print('Edges within 1..3 hops:')
print(hop_range._edges.sort_values([g._source, g._destination]))
render_static(hop_range, label_col=None, title='Hops 1..3')
```

```
Nodes within 1..3 hops from 'a':
   id  __gfql_output_node_hop__
0   a                      <NA>
1  b1                         1
3  b2                         1
2  c1                         2
5  c2                         2
4  d1                         3

Edges within 1..3 hops:
   __gfql_output_edge_hop__   s   d
0                         1   a  b1
3                         1   a  b2
1                         2  b1  c1
4                         2  b2  c2
2                         3  c1  d1
```

[6]:

Output slicing

Post-filter the traversal results without changing the traversal itself.

Traverse 2..4 hops but only display hops 3..4, with hop numbers on nodes/edges.

[7]:
```
# Traverse 2..4 hops; output slice 3..4 with hop labels
hop_slice = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=2, max_hops=4, output_min_hops=3, output_max_hops=4, label_node_
↪hops='hop', label_edge_hops='edge_hop'),
])
hop_slice = fill_hops(hop_slice, seeds=seed_ids)

print('Nodes in hops 3..4 (after traversing 2..4):')
```

(continues on next page)

(continued from previous page)

```
print(hop_slice._nodes.sort_values(['hop', g._node]))
print()
print('Edges in hops 3..4:')
print(hop_slice._edges.sort_values(['edge_hop', g._source, g._destination]))
render_static(hop_slice, label_col='hop', title='Slice hops 3..4')
```

```
Nodes in hops 3..4 (after traversing 2..4):
   id  hop
2  c1    3
0  d1    3
1  e1    4

Edges in hops 3..4:
   edge_hop   s   d
0         3  c1  d1
1         4  d1  e1
```

[7]:

Output_min below min_hops: keep the lead-in hops and label the seed at hop 0.

[8]:
```
# Output slice below traversal min (keeps earlier hops)
hop_slice_below = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=3, max_hops=3, output_min_hops=1, label_node_hops='hop', label_
↪edge_hops='edge_hop', label_seeds=True),
])
hop_slice_below = fill_hops(hop_slice_below, seeds=seed_ids)

print('Nodes when output_min < min_hops (labels on, seeds labeled):')
print(hop_slice_below._nodes.sort_values(['hop', g._node]))
print()
print('Edges when output_min < min_hops:')
print(hop_slice_below._edges.sort_values(['edge_hop', g._source, g._destination]))
render_static(hop_slice_below, label_col='hop', title='Output min < min_hops')
```

```
Nodes when output_min < min_hops (labels on, seeds labeled):
   id  hop
0   a    0
1  b1    1
2  c1    2
3  d1    3

Edges when output_min < min_hops:
   edge_hop   s   d
0         1   a  b1
1         2  b1  c1
2         3  c1  d1
```

[8]:

Output_max above traversal: slice is capped at traversal depth; edge labels show the cap.

[9]:
```
# Output slice max above traversal max (allowed, capped by traversal)
```

(continues on next page)

(continued from previous page)

```
hop_slice_above = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=2, max_hops=2, output_max_hops=5, label_edge_hops='edge_hop'),
])

print('Edges when output_max > traversal max (still capped at traversal):')
print(hop_slice_above._edges.sort_values(['edge_hop', g._source, g._destination]))
render_static(hop_slice_above, label_col=None, title='Output max > traversal')
```

```
Edges when output_max > traversal max (still capped at traversal):
   edge_hop   s   d
0         1   a  b1
2         1   a  b2
1         2  b1  c1
3         2  b2  c2
```

[9]:

Invalid output slices: examples of raised validation errors for mismatched bounds.

[10]:
```
# Invalid output slice (output_min > max_hops)
bad_output_min_chain = [
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=2, max_hops=3, output_min_hops=5),
]

# Invalid output slice (output_max < min_hops)
bad_output_max_chain = [
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=2, max_hops=3, output_max_hops=1),
]
try:
    g.gfql(bad_output_max_chain)
except Exception as e:
    print('Invalid output_max < min_hops:', e)
```

```
Invalid output_max < min_hops: [invalid-hops-value] output_max_hops cannot be below min_
↪hops traversal bound | field: output_max_hops | value: 1 | suggestion: Raise output_
↪max_hops or lower min_hops
```

Labels

Compare hop labels with and without labeling the seeds.

Labels without seeds: hop numbers start at 1 for new nodes/edges; seed 'a' stays unlabeled.

[11]:
```
# Hop labels without seed labels
hop_labels_off = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=1, max_hops=3, label_node_hops='hop', label_edge_hops='edge_hop',␣
↪label_seeds=False),
])
hop_labels_off = fill_hops(hop_labels_off, seeds=seed_ids)
```

(continues on next page)

(continued from previous page)

```
print('Nodes with hop labels (seeds not labeled):')
print(hop_labels_off._nodes.sort_values(['hop', g._node]))
print()
print('Edges with hop labels (seeds not labeled):')
print(hop_labels_off._edges.sort_values(['edge_hop', g._source, g._destination]))
render_static(hop_labels_off, label_col='hop', title='Labels (no seeds)')
```

```
Nodes with hop labels (seeds not labeled):
   id   hop
1  b1     1
3  b2     1
2  c1     2
5  c2     2
4  d1     3
0   a  <NA>

Edges with hop labels (seeds not labeled):
   edge_hop   s   d
0         1   a  b1
3         1   a  b2
1         2  b1  c1
4         2  b2  c2
2         3  c1  d1
```

[11]:

Labels with seeds: seed labeled hop 0; downstream hops increment from there.

[12]:
```
# Hop labels with seed labels
hop_labels_on = g.gfql([
    n({g._node: is_in(seed_ids)}),
    e_forward(min_hops=1, max_hops=3, label_node_hops='hop', label_edge_hops='edge_hop',␣
↪label_seeds=True),
])
hop_labels_on = fill_hops(hop_labels_on, seeds=seed_ids)

print('Nodes with hop labels (seeds labeled hop=0):')
print(hop_labels_on._nodes.sort_values(['hop', g._node]))
print()
print('Edges with hop labels (seeds labeled hop=0):')
print(hop_labels_on._edges.sort_values(['edge_hop', g._source, g._destination]))
render_static(hop_labels_on, label_col='hop', title='Labels (seeds=0)')
```

```
Nodes with hop labels (seeds labeled hop=0):
   id  hop
0   a    0
1  b1    1
2  b2    1
3  c1    2
4  c2    2
5  d1    3
```

(continues on next page)

(continued from previous page)

```
Edges with hop labels (seeds labeled hop=0):
   edge_hop   s   d
0         1   a  b1
3         1   a  b2
1         2  b1  c1
4         2  b2  c2
2         3  c1  d1
```

[12]:

10.8.3.2 GFQL Validation Fundamentals

Learn how to use GFQL's built-in validation system to catch errors early and build robust graph applications.

What You'll Learn

- How GFQL automatically validates queries
- Understanding structured error messages with error codes
- Schema validation against your data
- Pre-execution validation for performance
- Collecting all errors vs fail-fast mode

Prerequisites

- Basic Python knowledge
- PyGraphistry installed (`pip install graphistry[ai]`)

Setup and Imports

First, let's import the necessary modules and create sample data.

[ ]:
```python
# Core imports
import pandas as pd
import graphistry
from graphistry.compute.chain import Chain
from graphistry.compute.ast import n, e_forward, e_reverse

# Exception types for error handling
from graphistry.compute.exceptions import (
    GFQLValidationError,
    GFQLSyntaxError,
    GFQLTypeError,
    GFQLSchemaError,
    ErrorCode
)
```

(continues on next page)

(continued from previous page)

```
# Check version
print(f"PyGraphistry version: {graphistry.__version__}")
print("\nValidation is now built-in to GFQL operations!")
```

Automatic Syntax Validation

GFQL validates operations automatically when you create them. No need to call separate validation functions!

[ ]:
```
# Example 1: Valid chain creation
try:
    chain = Chain([
        n({'type': 'customer'}),
        e_forward(),
        n()
    ])
    print("Valid chain created successfully!")
    print(f"Chain has {len(chain.chain)} operations")
except GFQLValidationError as e:
    print(f"Validation error: {e}")
```

[ ]:
```
# Example 2: Invalid parameter - negative hops
try:
    chain = Chain([
        n(),
        e_forward(hops=-1),  # Invalid: negative hops
        n()
    ])
except GFQLTypeError as e:
    print(f"Caught validation error!")
    print(f"   Error code: {e.code}")
    print(f"   Message: {e.message}")
    print(f"   Field: {e.context.get('field')}")
    print(f"   Suggestion: {e.context.get('suggestion')}")
```

Understanding Error Codes

GFQL uses structured error codes for programmatic handling:

[ ]:
```
# Display available error codes
print("Error Code Categories:")
print("\nE1xx - Syntax Errors:")
print(f"  {ErrorCode.E101}: Invalid type (e.g., chain not a list)")
print(f"  {ErrorCode.E103}: Invalid parameter value")
print(f"  {ErrorCode.E104}: Invalid direction")
print(f"  {ErrorCode.E105}: Missing required field")

print("\nE2xx - Type Errors:")
print(f"  {ErrorCode.E201}: Type mismatch")
print(f"  {ErrorCode.E204}: Invalid name type")
```

(continues on next page)

(continued from previous page)

```
print("\nE3xx - Schema Errors:")
print(f"  {ErrorCode.E301}: Column not found")
print(f"  {ErrorCode.E302}: Incompatible column type")
```

Create Sample Data

[ ]:
```
# Create sample data
nodes_df = pd.DataFrame({
    'id': ['a', 'b', 'c', 'd', 'e'],
    'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'type': ['customer', 'customer', 'product', 'product', 'customer'],
    'score': [100, 85, 95, 120, 110],
    'active': [True, True, False, True, False]
})

edges_df = pd.DataFrame({
    'src': ['a', 'b', 'c', 'd', 'e'],
    'dst': ['c', 'd', 'a', 'b', 'c'],
    'weight': [1.0, 2.5, 0.8, 1.2, 3.0],
    'edge_type': ['buys', 'buys', 'recommends', 'recommends', 'buys']
})

# Create graph using canonical graphistry.edges() and graphistry.nodes()
g = graphistry.edges(edges_df, 'src', 'dst').nodes(nodes_df, 'id')

print("Graph created with:")
print(f"  Nodes: {len(g._nodes)} (columns: {list(g._nodes.columns)})")
print(f"  Edges: {len(g._edges)} (columns: {list(g._edges.columns)})")
```

Schema Validation (Runtime)

When you execute a chain, GFQL automatically validates against your data schema:

[ ]:
```
# Valid query - columns exist
try:
    result = g.gfql([
        n({'type': 'customer'}),
        e_forward({'edge_type': 'buys'}),
        n({'type': 'product'})
    ])
    print(f"Query executed successfully!")
    print(f"   Found {len(result._nodes)} nodes")
    print(f"   Found {len(result._edges)} edges")
except GFQLSchemaError as e:
    print(f"Schema error: {e}")
```

[ ]:
```
# Invalid query - column doesn't exist
try:
```

(continues on next page)

(continued from previous page)

```
    result = g.gfql([
        n({'category': 'VIP'})  # 'category' column doesn't exist
    ])
except GFQLSchemaError as e:
    print(f"Schema validation caught the error!")
    print(f"   Error code: {e.code}")
    print(f"   Message: {e.message}")
    print(f"   Suggestion: {e.context.get('suggestion')}")
```

Type Mismatch Detection

GFQL detects when you use the wrong type of value or predicate for a column:

[ ]:
```
# Type mismatch: string value on numeric column
try:
    result = g.gfql([
        n({'score': 'high'})  # 'score' is numeric, not string
    ])
except GFQLSchemaError as e:
    print(f"Type mismatch detected!")
    print(f"   {e}")
    print(f"\n   Column type: {e.context.get('column_type')}")
```

[ ]:
```
# Using predicates
from graphistry.compute.predicates.numeric import gt
from graphistry.compute.predicates.str import contains

# Correct: numeric predicate on numeric column
try:
    result = g.gfql([n({'score': gt(90)})])
    print(f"Valid: Found {len(result._nodes)} high-scoring nodes")
except GFQLSchemaError as e:
    print(f"Error: {e}")

# Wrong: string predicate on numeric column
try:
    result = g.gfql([n({'score': contains('9')})])
except GFQLSchemaError as e:
    print(f"\nPredicate type mismatch caught!")
    print(f"   {e.message}")
    print(f"   Suggestion: {e.context.get('suggestion')}")
```

Pre-Execution Validation

For better performance, you can validate queries before execution:

[ ]:
```
# Pre-validate to catch errors early
chain_to_test = Chain([
    n({'missing_col': 'value'}),
    e_forward({'also_missing': 'value'})
])

# Method 1: Use validate_schema parameter
try:
    result = g.gfql(chain_to_test.chain, validate_schema=True)
except GFQLSchemaError as e:
    print("Pre-execution validation caught error!")
    print(f"   Error: {e}")
    print("   (No graph operations were performed)")
```

[ ]:
```
# Method 2: Validate chain object directly
from graphistry.compute.validate_schema import validate_chain_schema

# Check if chain is compatible with graph schema
try:
    validate_chain_schema(g, chain_to_test)
    print("Chain is valid for this graph schema")
except GFQLSchemaError as e:
    print(f"Schema incompatibility: {e}")
```

Collect All Errors vs Fail-Fast

By default, validation fails on the first error. You can collect all errors instead:

[ ]:
```
# Create a chain with multiple errors
problematic_chain = Chain([
    n({'missing1': 'value', 'score': 'not-a-number'}),  # 2 errors
    e_forward({'missing2': 'value'}),  # 1 error
    n({'type': gt(5)})  # 1 error: numeric predicate on string column
])

# Fail-fast mode (default)
print("Fail-fast mode:")
try:
    problematic_chain.validate()
except GFQLValidationError as e:
    print(f"  Stopped at first error: {e}")

# Collect-all mode
print("\nCollect-all mode:")
errors = problematic_chain.validate(collect_all=True)
print(f"  Found {len(errors)} syntax/type errors")
```

(continues on next page)

(continued from previous page)

```
# For schema validation
schema_errors = validate_chain_schema(g, problematic_chain, collect_all=True)
print(f"  Found {len(schema_errors)} schema errors:")
for i, error in enumerate(schema_errors):
    print(f"\n  Error {i+1}: [{error.code}] {error.message}")
    if error.context.get('suggestion'):
        print(f"    Suggestion: {error.context['suggestion']}")
```

Error Handling Best Practices

[ ]:
```
# Comprehensive error handling example
def safe_chain_execution(g, operations):
    """Execute chain with proper error handling."""
    try:
        # Create chain
        chain = Chain(operations)

        # Pre-validate if desired
        # errors = chain.validate_schema(g, collect_all=True)
        # if errors:
        #     print(f"Warning: {len(errors)} schema issues found")

        # Execute
        result = g.gfql(operations)
        return result

    except GFQLSyntaxError as e:
        print(f"Syntax Error [{e.code}]: {e.message}")
        if e.context.get('suggestion'):
            print(f"  Try: {e.context['suggestion']}")
        return None

    except GFQLTypeError as e:
        print(f"Type Error [{e.code}]: {e.message}")
        print(f"  Field: {e.context.get('field')}")
        print(f"  Value: {e.context.get('value')}")
        return None

    except GFQLSchemaError as e:
        print(f"Schema Error [{e.code}]: {e.message}")
        if e.code == ErrorCode.E301:
            print("  Column not found in data")
        elif e.code == ErrorCode.E302:
            print("  Type mismatch between query and data")
        return None

# Test with valid query
print("Valid query:")
result = safe_chain_execution(g, [
    n({'type': 'customer'}),
```

(continues on next page)

(continued from previous page)

```
    e_forward()
])
if result:
    print(f"  Success! Found {len(result._nodes)} nodes")

# Test with invalid query
print("\nInvalid query:")
result = safe_chain_execution(g, [
    n({'invalid_column': 'value'})
])
```

Summary

Key Takeaways

1. **Automatic Validation**: GFQL validates automatically - no separate validation calls needed
2. **Structured Errors**: Error codes (E1xx, E2xx, E3xx) help with programmatic handling
3. **Helpful Messages**: Errors include suggestions for fixing issues
4. **Two Validation Stages**:
   - Syntax/Type: During chain construction
   - Schema: During execution (or pre-execution)
5. **Flexible Modes**: Choose between fail-fast or collect-all errors

Quick Reference

```
# Automatic syntax validation
chain = Chain([...])  # Validates syntax/types

# Runtime schema validation
result = g.gfql([...])  # Validates against data

# Pre-execution schema validation
result = g.gfql([...], validate_schema=True)

# Validate chain against graph schema
from graphistry.compute.validate_schema import validate_chain_schema
validate_chain_schema(g, chain)  # Throws GFQLSchemaError if invalid

# Collect all syntax errors
errors = chain.validate(collect_all=True)

# Collect all schema errors
schema_errors = validate_chain_schema(g, chain, collect_all=True)
```

Next Steps

- Explore more complex query patterns
- Learn about GFQL predicates for advanced filtering
- Use validation in production applications

10.8.3.3 GFQL DateTime Filtering Examples

This notebook shows how to filter graph data by dates and times using GFQL predicates.

Table of Contents

Key Temporal Filtering Concepts:

1. **Basic DateTime Filtering** - Filter by specific dates and times
2. **Date-Only Filtering** - Ignore time components
3. **Time-of-Day Filtering** - Filter by time patterns
4. **Complex Temporal Queries** - Combine with other predicates
5. **Temporal Value Classes** - Explicit temporal objects
6. **Timezone-Aware Filtering** - Handle timezone conversions
7. **Chain Operations** - Multi-hop temporal queries
8. **Wire Protocol Dicts** - JSON-compatible configuration

Quick Reference: - `gt()`, `lt()`, `ge()`, `le()` - Greater/less than comparisons - `between()` - Range queries - `is_in()` - Match specific values - `DateTimeValue`, `DateValue`, `TimeValue` - Explicit temporal types

[ ]:
```
# Standard Python datetime imports
import pandas as pd
import numpy as np
from datetime import datetime, date, time, timedelta

# Graphistry imports
import graphistry
from graphistry import n, e_forward, e_reverse, e_undirected

# Temporal predicates
from graphistry.compute import (
    gt, lt, ge, le, eq, ne, between, is_in,
    DateTimeValue, DateValue, TimeValue
)
```

Setup: Create Sample Data

Let's create a sample dataset representing a transaction network with temporal data.

[2]:
```
# Generate sample transaction data
np.random.seed(42)

# Create nodes (accounts)
n_accounts = 100
accounts_df = pd.DataFrame({
    'account_id': [f'ACC_{i:04d}' for i in range(n_accounts)],
    'account_type': np.random.choice(['checking', 'savings', 'business'], n_accounts),
    'created_date': pd.date_range('2020-01-01', periods=n_accounts, freq='W'),
    'last_active': pd.date_range('2023-01-01', periods=n_accounts, freq='D') +
                   pd.to_timedelta(np.random.randint(0, 365, n_accounts), unit='D')
})

# Create edges (transactions)
n_transactions = 500
transactions_df = pd.DataFrame({
    'transaction_id': [f'TXN_{i:06d}' for i in range(n_transactions)],
    'source': np.random.choice(accounts_df['account_id'], n_transactions),
    'target': np.random.choice(accounts_df['account_id'], n_transactions),
    'amount': np.random.exponential(100, n_transactions).round(2),
    'timestamp': pd.date_range('2023-01-01', periods=n_transactions, freq='h') +
                 pd.to_timedelta(np.random.randint(0, 8760, n_transactions), unit='h'),
    'transaction_time': [time(np.random.randint(0, 24), np.random.randint(0, 60))
                         for _ in range(n_transactions)],
    'transaction_type': np.random.choice(['transfer', 'payment', 'deposit'], n_
↪transactions)
})

print(f"Created {len(accounts_df)} accounts and {len(transactions_df)} transactions")
print(f"\nTransaction date range: {transactions_df['timestamp'].min()} to {transactions_
↪df['timestamp'].max()}")
```

[3]:
```
# Create graphistry instance
g = graphistry.edges(transactions_df, 'source', 'target').nodes(accounts_df, 'account_id
↪')
print(f"Graph: {len(g._nodes)} nodes, {len(g._edges)} edges")
```

1. Basic DateTime Filtering

Filter transactions based on datetime values using edge predicates.

[4]:
```
# Filter transactions after a specific date
# First, filter the edges directly
cutoff_date = datetime(2023, 7, 1)
recent_edges = g._edges[gt(pd.Timestamp(cutoff_date))(g._edges['timestamp'])]
recent_g = g.edges(recent_edges)
```

(continues on next page)

(continued from previous page)

```
print(f"Transactions after {cutoff_date}: {len(recent_g._edges)}")
recent_g._edges[['transaction_id', 'timestamp', 'amount']].head()
```

[5]:
```
# Alternative: Use chain with edge operations
# Start from all nodes, then follow edges with temporal filter
recent_chain = g.gfql([
    n(),  # Start with all nodes
    e_forward({
        "timestamp": gt(pd.Timestamp(cutoff_date))
    })
])

print(f"Transactions after {cutoff_date} (chain): {len(recent_chain._edges)}")
```

[6]:
```
# Filter transactions in a specific month
march_edges = g._edges[
    between(
        datetime(2023, 3, 1),
        datetime(2023, 3, 31, 23, 59, 59)
    )(g._edges['timestamp'])
]
march_g = g.edges(march_edges)

print(f"Transactions in March 2023: {len(march_g._edges)}")
march_g._edges[['transaction_id', 'timestamp', 'amount']].head()
```

2. Date-Only Filtering

Filter nodes based on dates, ignoring time components.

[7]:
```
# Filter accounts created after a specific date
new_accounts = g.gfql([
    n(filter_dict={
        "created_date": ge(date(2021, 1, 1))
    })
])

print(f"Accounts created after 2021: {len(new_accounts._nodes)}")
new_accounts._nodes[['account_id', 'created_date', 'account_type']].head()
```

[8]:
```
# Find accounts active in the last 90 days
ninety_days_ago = datetime.now().date() - timedelta(days=90)
active_accounts = g.gfql([
    n(filter_dict={
        "last_active": gt(pd.Timestamp(ninety_days_ago))
    })
])

print(f"Recently active accounts: {len(active_accounts._nodes)}")
```

3. Time-of-Day Filtering

Filter transactions based on time of day.

[9]:
```
# Find transactions during business hours (9 AM - 5 PM)
business_hours_edges = g._edges[
    between(
        time(9, 0, 0),
        time(17, 0, 0)
    )(g._edges['transaction_time'])
]
business_hours_g = g.edges(business_hours_edges)

print(f"Business hour transactions: {len(business_hours_g._edges)}")
print(f"Percentage of total: {len(business_hours_g._edges) / len(g._edges) * 100:.1f}%")
```

[10]:
```
# Find transactions at specific times (e.g., on the hour)
on_the_hour_times = [time(h, 0, 0) for h in range(24)]
on_hour_edges = g._edges[
    is_in(on_the_hour_times)(g._edges['transaction_time'])
]
on_hour_g = g.edges(on_hour_edges)

print(f"Transactions on the hour: {len(on_hour_g._edges)}")
```

4. Complex Temporal Queries

Combine temporal predicates with other filters for complex queries.

[11]:
```
# Find large transactions (>$500) in Q4 2023
q4_mask = between(
    datetime(2023, 10, 1),
    datetime(2023, 12, 31, 23, 59, 59)
)(g._edges['timestamp'])
large_mask = gt(500)(g._edges['amount'])

q4_large_edges = g._edges[q4_mask & large_mask]
q4_large_g = g.edges(q4_large_edges)

print(f"Large Q4 2023 transactions: {len(q4_large_g._edges)}")
if len(q4_large_g._edges) > 0:
    print(f"Total value: ${q4_large_g._edges['amount'].sum():,.2f}")
    print(f"Average: ${q4_large_g._edges['amount'].mean():,.2f}")
```

[12]:
```
# Multi-hop query: Find accounts that received money recently
# and then sent money to business accounts
thirty_days_ago = datetime.now() - timedelta(days=30)

# First, find recent transactions
recent_edges = g._edges[gt(pd.Timestamp(thirty_days_ago))(g._edges['timestamp'])]
recent_g = g.edges(recent_edges)
```

(continues on next page)

(continued from previous page)

```
# Use chain to find money flow pattern
money_flow = recent_g.gfql([
    # Start with any node
    n(),
    # Follow incoming edges (as destination)
    e_reverse(),
    # Go to source nodes
    n(),
    # Follow outgoing edges
    e_forward(),
    # To business accounts
    n(filter_dict={"account_type": "business"})
])

print(f"Money flow pattern found: {len(money_flow._nodes)} business accounts")
```

5. Using Temporal Value Classes

Use explicit temporal value classes for more control.

[13]:
```
# Create temporal values with specific properties
dt_value = DateTimeValue("2023-06-15T14:30:00", "UTC")
date_value = DateValue("2023-06-15")
time_value = TimeValue("14:30:00")

# Use in predicates
specific_edges = g._edges[gt(dt_value)(g._edges['timestamp'])]
specific_g = g.edges(specific_edges)

print(f"Transactions after {dt_value.value}: {len(specific_g._edges)}")
```

6. Timezone-Aware Filtering

Handle timezone-aware datetime comparisons.

[14]:
```
# Add timezone info to our data for this example
transactions_df_tz = transactions_df.copy()
transactions_df_tz['timestamp_utc'] = pd.to_datetime(transactions_df_tz['timestamp']).dt.
↪tz_localize('UTC')
transactions_df_tz['timestamp_eastern'] = transactions_df_tz['timestamp_utc'].dt.tz_
↪convert('US/Eastern')

g_tz = graphistry.edges(transactions_df_tz, 'source', 'target')

# Filter using Eastern time
eastern_cutoff = pd.Timestamp("2023-07-01 09:00:00", tz="US/Eastern")  # 9 AM Eastern

eastern_morning_edges = g_tz._edges[
```

(continues on next page)

(continued from previous page)

```
    gt(pd.Timestamp(eastern_cutoff))(g_tz._edges['timestamp_eastern'])
]
eastern_morning_g = g_tz.edges(eastern_morning_edges)

print(f"Transactions after 9 AM Eastern on July 1, 2023: {len(eastern_morning_g._edges)}
↪")
```

7. Chain Operations with Temporal Edge Filters

Demonstrate using temporal predicates in chain operations with proper edge filtering.

[15]:
```
# Find paths through recent high-value transactions
recent_high_value = g.gfql([
    # Start from all nodes
    n(),
    # Follow edges with temporal and amount filters
    e_forward({
        "timestamp": gt(datetime.now() - timedelta(days=7)),
        "amount": gt(200)
    }),
    # Reach destination nodes
    n()
])

print(f"Recent high-value transaction paths:")
print(f"  Nodes: {len(recent_high_value._nodes)}")
print(f"  Edges: {len(recent_high_value._edges)}")
```

[ ]:
```
# Wire protocol dicts in is_in predicates
# Useful for checking against multiple specific timestamps

important_dates = [
    {"type": "datetime", "value": "2023-01-01T00:00:00", "timezone": "UTC"},  # New Year
    {"type": "datetime", "value": "2023-07-04T00:00:00", "timezone": "UTC"},  # July 4th
    {"type": "datetime", "value": "2023-12-25T00:00:00", "timezone": "UTC"},  # Christmas
]

# Note: This checks for exact timestamp matches
# For date matching, you'd need to extract the date portion
holiday_pred = is_in(important_dates)

# For demonstration, let's check if any transactions happened exactly at midnight on␣
↪these days
# (In real data, you'd probably want to check date ranges instead)
print(f"Checking for transactions at midnight on holidays...")
print(f"(This is likely 0 unless transactions were specifically created at midnight)")
```

Summary

This notebook demonstrated:

1. **DateTime filtering** with `gt`, `lt`, `between` predicates on edges
2. **Date-only filtering** for day-level granularity on nodes
3. **Time-of-day filtering** for patterns like business hours
4. **Complex queries** combining temporal and non-temporal predicates
5. **Multi-hop queries** with temporal constraints using chain operations
6. **Temporal value classes** for explicit control
7. **Timezone-aware** filtering
8. **Wire protocol dictionaries** for JSON-compatible predicate configuration
9. **Proper chain syntax** with edge filters in `e_forward()` and node filters in `n()`

Key takeaways: - Temporal predicates work seamlessly with pandas datetime types - Wire protocol dicts enable configuration-driven filtering: `gt({"type": "datetime", "value": "2023-01-01T00:00:00", "timezone": "UTC"})` - Timezone awareness is built-in for accurate cross-timezone comparisons - Complex temporal patterns can be expressed through chain operations

Temporal predicates in GFQL provide a powerful way to analyze time-series aspects of graph data, enabling complex temporal queries while maintaining the expressiveness of graph traversals.

[ ]:
```
# Build predicates programmatically with wire protocol dicts
def create_date_filter(year, month, day, comparison="gt"):
    """Create a date filter using wire protocol format"""
    date_dict = {
        "type": "date",
        "value": f"{year:04d}-{month:02d}-{day:02d}"
    }

    if comparison == "gt":
        return gt(date_dict)
    elif comparison == "lt":
        return lt(date_dict)
    elif comparison == "ge":
        return ge(date_dict)
    elif comparison == "le":
        return le(date_dict)
    else:
        raise ValueError(f"Unknown comparison: {comparison}")

# Use the programmatic filter
filter_2023 = create_date_filter(2023, 1, 1, "ge")
accounts_2023 = g.gfql([
    n(filter_dict={
        "created_date": filter_2023
    })
])

print(f"Accounts created in 2023 or later: {len(accounts_2023._nodes)}")
```

[ ]:
```
# Example: Load predicate configuration from JSON
import json

# Simulate loading from a JSON config file
config_json = '''
{
    "filters": {
        "recent_transactions": {
            "timestamp": {
                "type": "gt",
                "value": {
                    "type": "datetime",
                    "value": "2023-10-01T00:00:00",
                    "timezone": "UTC"
                }
            }
        },
        "business_hours": {
            "transaction_time": {
                "type": "between",
                "start": {"type": "time", "value": "09:00:00"},
                "end": {"type": "time", "value": "17:00:00"}
            }
        }
    }
}
'''

config = json.loads(config_json)

# Use the wire protocol dict directly
recent_filter = config["filters"]["recent_transactions"]["timestamp"]["value"]
recent_edges = g._edges[gt(recent_filter)(g._edges['timestamp'])]
recent_g = g.edges(recent_edges)

print(f"Recent transactions (from JSON config): {len(recent_g._edges)}")
```

[ ]:
```
# Wire protocol dictionaries work directly in Python
# These are equivalent:
pred1 = gt(pd.Timestamp("2023-07-01"))
pred2 = gt({"type": "datetime", "value": "2023-07-01T00:00:00", "timezone": "UTC"})

# Test they produce the same results
result1 = pred1(g._edges['timestamp'])
result2 = pred2(g._edges['timestamp'])
print(f"Results are identical: {result1.equals(result2)}")
print(f"Transactions after July 1, 2023: {result1.sum()}")
```

8. Using Wire Protocol Dictionaries

You can pass wire protocol dictionaries directly to temporal predicates. This is useful for: - Loading predicate configurations from JSON files - Building predicates programmatically - Sharing predicate definitions between systems

What's Next?

- Datetime Filtering Guide - Full temporal predicate reference
- Wire Protocol Reference - JSON serialization examples
- GFQL Documentation - Complete GFQL reference

[16]:
```
# Complex multi-hop with temporal constraints
# Find 2-hop paths through recent transactions
two_hop_recent = g.gfql([
    # Start from business accounts
    n(filter_dict={"account_type": "business"}),
    # First hop: recent outgoing transactions
    e_forward({
        "timestamp": gt(datetime.now() - timedelta(days=30))
    }, name="hop1"),
    # Intermediate nodes
    n(),
    # Second hop: any transaction
    e_forward(name="hop2"),
    # Final nodes
    n()
])

print(f"2-hop paths from business accounts through recent transactions:")
print(f"  Total edges: {len(two_hop_recent._edges)}")
print(f"  Hop 1 edges: {two_hop_recent._edges['hop1'].sum()}")
print(f"  Hop 2 edges: {two_hop_recent._edges['hop2'].sum()}")
```

Summary

This notebook demonstrated:

1. **DateTime filtering** with `gt`, `lt`, `between` predicates on edges
2. **Date-only filtering** for day-level granularity on nodes
3. **Time-of-day filtering** for patterns like business hours
4. **Complex queries** combining temporal and non-temporal predicates
5. **Multi-hop queries** with temporal constraints using chain operations
6. **Temporal value classes** for explicit control
7. **Timezone-aware** filtering
8. **Proper chain syntax** with edge filters in `e_forward()` and node filters in `n()`

Temporal predicates in GFQL provide a powerful way to analyze time-series aspects of graph data, enabling complex temporal queries while maintaining the expressiveness of graph traversals.

10.8.3.4 GFQL CPU, GPU Benchmark

This notebook examines GFQL progerty graph query performance on 1-8 hop queries using CPU + GPU modes on various real-world 100K - 100M edge graphs. The data comes from a variety of popular social networks. The single-threaded CPU mode benefits from GFQL's novel dataframe engine, and the GPU mode further adds single-GPU acceleration. Both the `chain()` and `hop()` methods are examined.

The benchmark does not examine bigger-than-memory and distributed scenarios. The provided results here are from running on a free Google Colab T4 runtime, with a 2.2GHz Intel CPU (12 GB CPU RAM) and T4 Nvidia GPU (16 GB GPU RAM).

Data

From https://snap.stanford.edu/data/

Network	Nodes	Edges
Facebook	4,039	88,234
Twitter	81,306	2,420,766
GPlus	107,614	30,494,866
Orkut	3,072,441	117,185,082

Results

Definitions:

- GTEPS: Giga (billion) edges traversed per second
- T edges / $: Estimated trillion edges traversed for 1$ USD based on observed GTEPS and a 3yr AWS reservation (as of 12/2023)

Tasks:

1. `chain()` - includes complex pre/post processing

Task: `g.gfql([n({'id': some_id}), e_forward(hops=some_n)])`

Dataset	Max GPU Speedup	CPU GTEPS	GPU GTEPS	T CPU edges / $ (t3.l)	T GPU edges / $ (g4dn.xl)
Facebook	1.1X	0.66	0.61	65.7	10.4
Twitter	17.4X	0.17	2.81	16.7	48.1
GPlus	43.8X	0.09	2.87	8.5	49.2
Orkut	N/A	N/A	12.15	N/A	208.3
AVG	20.7X	0.30	4.61	30.3	79.0
MAX	43.8X	0.66	12.15	65.7	208.3

2. `hop()` - core property search primitive similar to BFS

Task: `g.hop(nodes=[some_id], direction='forward', hops=some_n)`

Dataset	Max GPU Speedup	CPU GTEPS	GPU GTEPS	T CPU edges / $ (t3.l)	T GPU edges / $ (g4dn.xl)
Face-book	3X	0.47	1.47	47.0	25.2
Twitter	42X	0.50	10.51	50.2	180.2
GPlus	21X	0.26	4.11	26.2	70.4
Orkut	N/A	N/A	41.50	N/A	711.4
AVG	22X	0.41	14.4	41.1	246.8
MAX	42X	0.50	41.50	50.2	711.4

Optional: GPU setup - Google Colab

[1]:
```
# Report GPU used when GPU benchmarking
# ! nvidia-smi
```

```
Tue Dec 26 00:50:30 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   54C    P8              10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

[8]:
```
# if in google colab
#!git clone https://github.com/rapidsai/rapidsai-csp-utils.git
#!python rapidsai-csp-utils/colab/pip-install.py
```

[3]:
```
import cudf
cudf.__version__
```

[3]:
```
'23.12.01'
```

1. Install & configure

[2]:
```
#! pip install graphistry[igraph]
```

```
  Preparing metadata (setup.py) ... done
```

Imports

[3]:
```
import pandas as pd

import graphistry

from graphistry import (

    # graph operators
    n, e_undirected, e_forward, e_reverse,

    # attribute predicates
    is_in, ge, startswith, contains, match as match_re
)
graphistry.__version__
```

[3]:
```
'0.32.0+12.g72e778c'
```

[6]:
```
import cudf
```

[7]:
```
#work around google colab shell encoding bugs

import locale
locale.getpreferredencoding = lambda: "UTF-8"
```

2. Perf benchmarks

Facebook: 88K edges

[10]:
```
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/facebook_combined.txt', sep=' ', names=['s', 'd'])
print(df.shape)
df.head(5)
```

```
(88234, 2)
```

[10]:

	s	d
0	0	1
1	0	2
2	0	3
3	0	4
4	0	5

[11]:
```
fg = graphistry.edges(df, 's', 'd').materialize_nodes()
print(fg._nodes.shape, fg._edges.shape)
fg._nodes.head(5)
```

```
(4039, 1) (88234, 2)
```

[11]:

	id
0	0
1	1
2	2
3	3
4	4

[12]:
```
%%time
for i in range(100):
  fg2 = fg.gfql([n({'id': 0}), e_forward(hops=2)])
```

```
CPU times: user 13.6 s, sys: 2.08 s, total: 15.7 s
Wall time: 18 s
```

[ ]:
```
%%time
fg_gdf = fg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
for i in range(100):
  fg2 = fg_gdf.gfql([n({'id': 0}), e_forward(hops=2)])
print(fg._nodes.shape, fg._edges.shape)
print(fg2._nodes.shape, fg2._edges.shape)
del fg_gdf
del fg2
```

```
(4039, 1) (88234, 2)
(1519, 1) (4060, 2)
CPU times: user 11.8 s, sys: 28.1 ms, total: 11.8 s
Wall time: 11.9 s
```

[ ]:
```
%%time
for i in range(50):
  fg2 = fg.gfql([n({'id': 0}), e_forward(hops=5)])
print(fg._nodes.shape, fg._edges.shape)
print(fg2._nodes.shape, fg2._edges.shape)
```

```
(4039, 1) (88234, 2)
(3829, 1) (86074, 2)
CPU times: user 15.4 s, sys: 808 ms, total: 16.2 s
Wall time: 16.2 s
```

[ ]:
```
%%time
fg_gdf = fg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
for i in range(50):
  fg2 = fg_gdf.gfql([n({'id': 0}), e_forward(hops=5)])
print(fg._nodes.shape, fg._edges.shape)
print(fg2._nodes.shape, fg2._edges.shape)
del fg_gdf
```

(continues on next page)

(continued from previous page)

```
del fg2
```

```
(4039, 1) (88234, 2)
(3829, 1) (86074, 2)
CPU times: user 9.82 s, sys: 133 ms, total: 9.95 s
Wall time: 10.1 s
```

[ ]:
```
%%time
for i in range(100):
  fg2 = fg.gfql([e_forward(source_node_match={'id': 0}, hops=5)])
```

```
CPU times: user 11.8 s, sys: 377 ms, total: 12.1 s
Wall time: 13.1 s
```

[ ]:
```
%%time
fg_gdf = fg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
for i in range(100):
  fg2 = fg_gdf.gfql([e_forward(source_node_match={'id': 0}, hops=5)])
print(fg._nodes.shape, fg._edges.shape)
print(fg2._nodes.shape, fg2._edges.shape)
del fg_gdf
del fg2
```

```
(4039, 1) (88234, 2)
(348, 1) (347, 2)
CPU times: user 14.1 s, sys: 48.5 ms, total: 14.2 s
Wall time: 14.2 s
```

[17]:
```
%%time
start_nodes = pd.DataFrame({fg._node: [0]})
for i in range(100):
  fg2 = fg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
print(fg2._nodes.shape, fg2._edges.shape)
```

```
(1519, 1) (4060, 2)
CPU times: user 4.5 s, sys: 1.35 s, total: 5.85 s
Wall time: 6.09 s
```

[18]:
```
%%time
start_nodes = cudf.DataFrame({fg._node: [0]})
fg_gdf = fg.nodes(cudf.from_pandas(fg._nodes)).edges(cudf.from_pandas(fg._edges))
for i in range(100):
  fg2 = fg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
print(fg2._nodes.shape, fg2._edges.shape)
```

(continues on next page)

(continued from previous page)

```
del start_nodes
del fg_gdf
del fg2
```

```
(1519, 1) (4060, 2)
CPU times: user 2.58 s, sys: 6.75 ms, total: 2.59 s
Wall time: 2.58 s
```

[19]:
```
%%time
start_nodes = pd.DataFrame({fg._node: [0]})
for i in range(100):
  fg2 = fg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=5)
print(fg2._nodes.shape, fg2._edges.shape)
```

```
(3829, 1) (86074, 2)
CPU times: user 13.2 s, sys: 2 s, total: 15.2 s
Wall time: 18.3 s
```

[20]:
```
%%time
start_nodes = cudf.DataFrame({fg._node: [0]})
fg_gdf = fg.nodes(cudf.from_pandas(fg._nodes)).edges(cudf.from_pandas(fg._edges))
for i in range(100):
  fg2 = fg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=5)
print(fg2._nodes.shape, fg2._edges.shape)
del start_nodes
del fg_gdf
del fg2
```

```
(3829, 1) (86074, 2)
CPU times: user 5.72 s, sys: 159 ms, total: 5.88 s
Wall time: 5.86 s
```

Twitter

- edges: 2420766
- nodes: 81306

[21]:
```
#! wget 'https://snap.stanford.edu/data/twitter_combined.txt.gz'
```

```
--2023-12-25 21:58:27--  https://snap.stanford.edu/data/twitter_combined.txt.gz
Resolving snap.stanford.edu (snap.stanford.edu)... 171.64.75.80
Connecting to snap.stanford.edu (snap.stanford.edu)|171.64.75.80|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10621918 (10M) [application/x-gzip]
Saving to: ‘twitter_combined.txt.gz’
```

(continues on next page)

(continued from previous page)

```
twitter_combined.tx 100%[===================>]  10.13M  3.00MB/s    in 4.0s

2023-12-25 21:58:32 (2.52 MB/s) - 'twitter_combined.txt.gz' saved [10621918/10621918]
```

[22]:
```
#! gunzip twitter_combined.txt.gz
```

[24]:
```
#! head -n 5 twitter_combined.txt
```

```
214328887 34428380
17116707 28465635
380580781 18996905
221036078 153460275
107830991 17868918
```

[25]:
```
%%time
te_df = pd.read_csv('twitter_combined.txt', sep=' ', names=['s', 'd'])
te_df.shape
```

```
CPU times: user 474 ms, sys: 61.9 ms, total: 536 ms
Wall time: 534 ms
```

[25]:
```
(2420766, 2)
```

[26]:
```
import graphistry
```

[27]:
```
%%time
g = graphistry.edges(te_df, 's', 'd').materialize_nodes()
g._nodes.shape
```

```
CPU times: user 86.4 ms, sys: 106 ms, total: 193 ms
Wall time: 191 ms
```

[27]:
```
(81306, 1)
```

[29]:
```
%%time
for i in range(10):
  g2 = g.gfql([n({'id': 17116707}), e_forward(hops=1)])
g2._nodes.shape, g2._edges.shape
```

```
CPU times: user 11.8 s, sys: 8.4 s, total: 20.2 s
Wall time: 23 s
```

[29]:
```
((140, 1), (615, 2))
```

[30]:
```
%%time
g_gdf = g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
for i in range(10):
  out = g_gdf.gfql([n({'id': 17116707}), e_forward(hops=1)])._nodes
print(out.shape)
del g_gdf
del out
```

```
(140, 1)
CPU times: user 1.33 s, sys: 46.6 ms, total: 1.38 s
Wall time: 1.63 s
```

[31]:
```
%%time
for i in range(10):
  out = g.gfql([n({'id': 17116707}), e_forward(hops=2)])
print(out._nodes.shape, out._edges.shape)
```

```
(2345, 1) (68536, 2)
CPU times: user 13.3 s, sys: 8.05 s, total: 21.4 s
Wall time: 21.6 s
```

[36]:
```
%%time
g_gdf = g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
for i in range(10):
  out = g_gdf.gfql([n({'id': 17116707}), e_forward(hops=2)])._nodes
print(out.shape)
del g_gdf
del out
```

```
(2345, 1)
CPU times: user 1.67 s, sys: 55.8 ms, total: 1.72 s
Wall time: 1.75 s
```

[37]:
```
%%time
for i in range(10):
  out = g.gfql([n({'id': 17116707}), e_forward(hops=8)])
print(out._nodes.shape, out._edges.shape)
```

```
(81304, 1) (2417796, 2)
CPU times: user 1min 56s, sys: 17.1 s, total: 2min 13s
Wall time: 2min 22s
```

[38]:
```
%%time
g_gdf = g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
for i in range(10):
  out = g_gdf.gfql([n({'id': 17116707}), e_forward(hops=8)])._nodes
print(out.shape)
del g_gdf
del out
```

```
(81304, 1)
CPU times: user 5.3 s, sys: 1.48 s, total: 6.78 s
Wall time: 7.89 s
```

[39]:
```
%%time
start_nodes = pd.DataFrame({g._node: [17116707]})
for i in range(10):
  g2 = g.hop(
      nodes=start_nodes,
```

(continues on next page)

(continued from previous page)

```
      direction='forward',
      hops=1)
print(g2._nodes.shape, g2._edges.shape)
```

```
(0, 1) (0, 2)
CPU times: user 2.58 s, sys: 1.59 s, total: 4.17 s
Wall time: 6.02 s
```

[44]:
```
%%time
start_nodes = cudf.DataFrame({g._node: [17116707]})
g_gdf = g.nodes(cudf.from_pandas(g._nodes)).edges(cudf.from_pandas(g._edges))
for i in range(10):
  g2 = g_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=5)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del g_gdf
del g2
```

```
(61827, 1) (1473599, 2)
CPU times: user 822 ms, sys: 179 ms, total: 1 s
Wall time: 997 ms
```

[40]:
```
%%time
start_nodes = pd.DataFrame({g._node: [17116707]})
for i in range(10):
  g2 = g.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
print(g2._nodes.shape, g2._edges.shape)
```

```
(2345, 1) (68536, 2)
CPU times: user 8.93 s, sys: 5.92 s, total: 14.9 s
Wall time: 15.8 s
```

[41]:
```
%%time
start_nodes = cudf.DataFrame({g._node: [17116707]})
g_gdf = g.nodes(cudf.from_pandas(g._nodes)).edges(cudf.from_pandas(g._edges))
for i in range(10):
  g2 = g_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del g_gdf
del g2
```

```
(2345, 1) (68536, 2)
CPU times: user 374 ms, sys: 6.92 ms, total: 381 ms
```

(continues on next page)

(continued from previous page)

```
Wall time: 379 ms
```

[42]:
```
%%time
start_nodes = pd.DataFrame({g._node: [17116707]})
for i in range(10):
  g2 = g.hop(
      nodes=start_nodes,
      direction='forward',
      hops=8)
print(g2._nodes.shape, g2._edges.shape)
```

```
(81304, 1) (2417796, 2)
CPU times: user 38.8 s, sys: 8.7 s, total: 47.5 s
Wall time: 48.2 s
```

[43]:
```
%%time
start_nodes = cudf.DataFrame({g._node: [17116707]})
g_gdf = g.nodes(cudf.from_pandas(g._nodes)).edges(cudf.from_pandas(g._edges))
for i in range(10):
  g2 = g_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=8)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del g_gdf
del g2
```

```
(81304, 1) (2417796, 2)
CPU times: user 1.8 s, sys: 506 ms, total: 2.3 s
Wall time: 2.3 s
```

GPlus

- edges: 30494866
- nodes: 107614

[4]:
```
#! wget https://snap.stanford.edu/data/gplus_combined.txt.gz
```

```
--2023-12-26 18:36:29--  https://snap.stanford.edu/data/gplus_combined.txt.gz
Resolving snap.stanford.edu (snap.stanford.edu)... 171.64.75.80
Connecting to snap.stanford.edu (snap.stanford.edu)|171.64.75.80|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 398930514 (380M) [application/x-gzip]
Saving to: ‘gplus_combined.txt.gz’

gplus_combined.txt. 100%[===================>] 380.45M  34.7MB/s    in 9.6s

2023-12-26 18:36:39 (39.7 MB/s) - ‘gplus_combined.txt.gz’ saved [398930514/398930514]
```

[5]:
```
#! gunzip gplus_combined.txt.gz
```

[6]:
```
%%time
ge_df = pd.read_csv('gplus_combined.txt', sep=' ', names=['s', 'd'])
print(ge_df.shape)
ge_df.head(5)
```

```
(30494866, 2)
CPU times: user 16 s, sys: 1.45 s, total: 17.5 s
Wall time: 22.5 s
```

[6]:

	s	d
0	116374117927631468606	101765416973555767821
1	112188647432305746617	107727150903234299458
2	116719211656774388392	100432456209427807893
3	117421021456205115327	101096322838605097368
4	116407635616074189669	113556266482860931616

[7]:
```
%%time
gg = graphistry.edges(ge_df, 's', 'd').materialize_nodes()
gg = graphistry.edges(ge_df, 's', 'd').nodes(gg._nodes, 'id')
print(gg._edges.shape, gg._nodes.shape)
gg._nodes.head(5)
```

```
(30494866, 2) (107614, 1)
CPU times: user 4.49 s, sys: 1.25 s, total: 5.74 s
Wall time: 5.97 s
```

[7]:

	id
0	116374117927631468606
1	112188647432305746617
2	116719211656774388392
3	117421021456205115327
4	116407635616074189669

[49]:
```
%%time
gg.gfql([ n({'id': '116374117927631468606'})])._nodes
```

```
CPU times: user 534 ms, sys: 598 ms, total: 1.13 s
Wall time: 1.65 s
```

[49]:

	id
0	116374117927631468606

[ ]:
```
%%time
out = gg.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=1)])._nodes
out.shape
```

```
CPU times: user 27.5 s, sys: 11.1 s, total: 38.5 s
Wall time: 39.5 s
```

```
(1473, 1)
```

[ ]:
```
%%time
gg_gdf = gg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
```

(continues on next page)

(continued from previous page)

```
↪edges))
out = gg_gdf.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=1)])
print(out._nodes.shape, out._edges.shape)
del gg_gdf
del out
```

```
(1473, 1) (13375, 2)
CPU times: user 4.57 s, sys: 2.11 s, total: 6.68 s
Wall time: 7.63 s
```

[ ]:
```
%%time
out = gg.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=2)])._nodes
out.shape
```

```
CPU times: user 45.8 s, sys: 17 s, total: 1min 2s
Wall time: 1min 5s
```

```
(44073, 1)
```

[ ]:
```
%%time
gg_gdf = gg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
out = gg_gdf.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=2)])
print(out._nodes.shape, out._edges.shape)
del gg_gdf
del out
```

```
(44073, 1) (2069325, 2)
CPU times: user 4.97 s, sys: 2.36 s, total: 7.34 s
Wall time: 10.6 s
```

[ ]:
```
%%time
out = gg.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=3)])._nodes
out.shape
```

```
CPU times: user 3min 45s, sys: 1min 5s, total: 4min 50s
Wall time: 4min 52s
```

```
(102414, 1)
```

[ ]:
```
%%time
gg_gdf = gg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
out = gg_gdf.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=3)])
print(out._nodes.shape, out._edges.shape)
del gg_gdf
del out
```

```
(102414, 1) (24851333, 2)
CPU times: user 6.95 s, sys: 2.63 s, total: 9.57 s
Wall time: 9.84 s
```

[8]:
```
%%time
out = gg.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=4)])
print(out._nodes.shape, out._edges.shape)
```

```
(105479, 1) (30450354, 2)
CPU times: user 4min 36s, sys: 1min 25s, total: 6min 2s
Wall time: 6min 4s
```

[ ]:
```
%%time
gg_gdf = gg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
out = gg_gdf.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=4)])
print(out._nodes.shape, out._edges.shape)
del gg_gdf
del out
```

```
(105479, 1) (30450354, 2)
CPU times: user 7.44 s, sys: 2.45 s, total: 9.88 s
Wall time: 9.9 s
```

[9]:
```
%%time
out = gg.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=5)])
print(out._nodes.shape, out._edges.shape)
```

```
(105604, 1) (30468335, 2)
CPU times: user 5min 36s, sys: 1min 39s, total: 7min 16s
Wall time: 7min 15s
```

[ ]:
```
%%time
gg_gdf = gg.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g._
↪edges))
out = gg_gdf.gfql([ n({'id': '116374117927631468606'}), e_forward(hops=5)])
print(out._nodes.shape, out._edges.shape)
del gg_gdf
del out
```

```
(105604, 1) (30468335, 2)
CPU times: user 8.82 s, sys: 2.71 s, total: 11.5 s
Wall time: 11.9 s
```

[50]:
```
%%time
start_nodes = pd.DataFrame({gg._node: ['116374117927631468606']})
for i in range(1):
  g2 = gg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=1)
print(g2._nodes.shape, g2._edges.shape)
```

```
(1473, 1) (13375, 2)
CPU times: user 19.9 s, sys: 9.36 s, total: 29.2 s
Wall time: 41.8 s
```

[52]:
```
%%time
start_nodes = cudf.DataFrame({gg._node: ['116374117927631468606']})
gg_gdf = gg.nodes(cudf.from_pandas(gg._nodes)).edges(cudf.from_pandas(gg._edges))
for i in range(1):
```

(continues on next page)

(continued from previous page)

```
  g2 = gg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=1)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del gg_gdf
del g2
```

```
(1473, 1) (13375, 2)
CPU times: user 3.71 s, sys: 2.09 s, total: 5.8 s
Wall time: 6.05 s
```

[53]:
```
%%time
start_nodes = pd.DataFrame({gg._node: ['116374117927631468606']})
for i in range(1):
  g2 = gg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
print(g2._nodes.shape, g2._edges.shape)
```

```
(44073, 1) (2069325, 2)
CPU times: user 27.8 s, sys: 13.2 s, total: 41 s
Wall time: 43.9 s
```

[54]:
```
%%time
start_nodes = cudf.DataFrame({gg._node: ['116374117927631468606']})
gg_gdf = gg.nodes(cudf.from_pandas(gg._nodes)).edges(cudf.from_pandas(gg._edges))
for i in range(1):
  g2 = gg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del gg_gdf
del g2
```

```
(44073, 1) (2069325, 2)
CPU times: user 4.26 s, sys: 2.37 s, total: 6.63 s
Wall time: 7.91 s
```

[55]:
```
%%time
start_nodes = pd.DataFrame({gg._node: ['116374117927631468606']})
for i in range(1):
  g2 = gg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=3)
print(g2._nodes.shape, g2._edges.shape)
```

```
(102414, 1) (24851333, 2)
```

(continues on next page)

(continued from previous page)

```
CPU times: user 1min 3s, sys: 22.7 s, total: 1min 26s
Wall time: 1min 35s
```

[56]:
```
%%time
start_nodes = cudf.DataFrame({gg._node: ['116374117927631468606']})
gg_gdf = gg.nodes(cudf.from_pandas(gg._nodes)).edges(cudf.from_pandas(gg._edges))
for i in range(1):
  g2 = gg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=3)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del gg_gdf
del g2
```

```
(102414, 1) (24851333, 2)
CPU times: user 3.96 s, sys: 2.11 s, total: 6.07 s
Wall time: 6.05 s
```

[57]:
```
%%time
start_nodes = pd.DataFrame({gg._node: ['116374117927631468606']})
for i in range(1):
  g2 = gg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=4)
print(g2._nodes.shape, g2._edges.shape)
```

```
(105479, 1) (30450354, 2)
CPU times: user 1min 34s, sys: 30.6 s, total: 2min 5s
Wall time: 2min 5s
```

[58]:
```
%%time
start_nodes = cudf.DataFrame({gg._node: ['116374117927631468606']})
gg_gdf = gg.nodes(cudf.from_pandas(gg._nodes)).edges(cudf.from_pandas(gg._edges))
for i in range(1):
  g2 = gg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=4)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del gg_gdf
del g2
```

```
(105479, 1) (30450354, 2)
CPU times: user 5.25 s, sys: 2.41 s, total: 7.67 s
Wall time: 7.69 s
```

[59]:
```
%%time
start_nodes = pd.DataFrame({gg._node: ['116374117927631468606']})
```

(continues on next page)

(continued from previous page)

```
for i in range(1):
  g2 = gg.hop(
      nodes=start_nodes,
      direction='forward',
      hops=5)
print(g2._nodes.shape, g2._edges.shape)
```

```
(105604, 1) (30468335, 2)
CPU times: user 2min 16s, sys: 39.1 s, total: 2min 55s
Wall time: 2min 58s
```

[60]:
```
%%time
start_nodes = cudf.DataFrame({gg._node: ['116374117927631468606']})
gg_gdf = gg.nodes(cudf.from_pandas(gg._nodes)).edges(cudf.from_pandas(gg._edges))
for i in range(1):
  g2 = gg_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=5)
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del gg_gdf
del g2
```

```
(105604, 1) (30468335, 2)
CPU times: user 5.79 s, sys: 2.51 s, total: 8.3 s
Wall time: 8.29 s
```

Orkut

- 117M edges
- 3M nodes

[8]:
```
#! wget https://snap.stanford.edu/data/bigdata/communities/com-orkut.ungraph.txt.gz
```

```
--2023-12-26 00:55:52--  https://snap.stanford.edu/data/bigdata/communities/com-orkut.
↪ungraph.txt.gz
Resolving snap.stanford.edu (snap.stanford.edu)... 171.64.75.80
Connecting to snap.stanford.edu (snap.stanford.edu)|171.64.75.80|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 447251958 (427M) [application/x-gzip]
Saving to: 'com-orkut.ungraph.txt.gz'

com-orkut.ungraph.t 100%[===================>] 426.53M  45.1MB/s    in 9.7s

2023-12-26 00:56:02 (44.0 MB/s) - 'com-orkut.ungraph.txt.gz' saved [447251958/447251958]
```

[9]:
```
#! gunzip com-orkut.ungraph.txt.gz
```

[10]:
```
#! head -n 7 com-orkut.ungraph.txt
```

```
# Undirected graph: ../../data/output/orkut.txt
# Orkut
# Nodes: 3072441 Edges: 117185083
# FromNodeId   ToNodeId
1       2
1       3
1       4
```

[11]:
```
import pandas as pd

import graphistry

from graphistry import (

    # graph operators
    n, e_undirected, e_forward, e_reverse,

    # attribute predicates
    is_in, ge, startswith, contains, match as match_re
)

import cudf

#work around google colab shell encoding bugs
import locale
locale.getpreferredencoding = lambda: "UTF-8"

cudf.__version__, graphistry.__version__
```

[11]:
```
('23.12.01', '0.32.0+12.g72e778c')
```

[12]:
```
#! nvidia-smi
```

```
Tue Dec 26 00:56:27 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   47C    P0              27W /  70W |    103MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

[13]:
```
%%time
co_df = cudf.read_csv('com-orkut.ungraph.txt', sep='\t', names=['s', 'd'], skiprows=5).
↪to_pandas()
print(co_df.shape)
print(co_df.head(5))
print(co_df.dtypes)
#del co_df
```

```
(117185082, 2)
   s  d
0  1  3
1  1  4
2  1  5
3  1  6
4  1  7
s    int64
d    int64
dtype: object
CPU times: user 2.56 s, sys: 4.2 s, total: 6.76 s
Wall time: 6.76 s
```

[14]:
```
%%time
co_g = graphistry.edges(cudf.DataFrame(co_df), 's', 'd').materialize_nodes(engine='cudf')
co_g = co_g.nodes(lambda g: g._nodes.to_pandas()).edges(lambda g: g._edges.to_pandas())
print(co_g._nodes.shape, co_g._edges.shape)
co_g._nodes.head(5)
```

```
(3072441, 1) (117185082, 2)
CPU times: user 1.96 s, sys: 2.95 s, total: 4.91 s
Wall time: 4.92 s
```

[14]:
```
   id
0   1
1   2
2   3
3   4
4   5
```

[15]:
```
#! nvidia-smi
```

```
Tue Dec 26 00:56:39 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   49C    P0              27W /  70W |   2819MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
```

(continues on next page)

(continued from previous page)

```
| Processes:                                                                                |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|===========================================================================================|
+-------------------------------------------------------------------------------------------+
```

[ ]:
```
%%time
# crashes
if False:
  out = co_g.gfql([ n({'id': 1}), e_forward(hops=1)])._nodes
  print(out.shape)
  del out
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 47.7 µs
```

[ ]:
```
%%time
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  out = co_gdf.gfql([ n({'id': 1}), e_forward(hops=1)])
#! nvidia-smi
print(out._nodes.shape, out._edges.shape)
del co_gdf
del out
```

```
Mon Dec 25 06:23:46 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   63C    P0              30W /  70W |   1925MiB / 15360MiB |     35%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Mon Dec 25 06:23:49 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
```

(continues on next page)

(continued from previous page)

```
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   66C    P0              72W /  70W |   2845MiB / 15360MiB |     84%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(12, 1) (11, 2)
CPU times: user 4.42 s, sys: 131 ms, total: 4.55 s
Wall time: 4.42 s
```

[ ]:
```
%%time
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  out = co_gdf.gfql([ n({'id': 1}), e_forward(hops=2)])
#! nvidia-smi
print(out._nodes.shape, out._edges.shape)
del co_gdf
del out
```

```
Mon Dec 25 06:24:52 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P0              29W /  70W |   1925MiB / 15360MiB |     22%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Mon Dec 25 06:24:58 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
```

(continues on next page)

(continued from previous page)

```
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   64C    P0              71W /  70W |   2845MiB / 15360MiB |     57%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(391, 1) (461, 2)
CPU times: user 5.34 s, sys: 132 ms, total: 5.47 s
Wall time: 6.13 s
```

[ ]:
```
%%time
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  out = co_gdf.gfql([ n({'id': 1}), e_forward(hops=3)])
#! nvidia-smi
print(out._nodes.shape, out._edges.shape)
del co_gdf
del out
```

```
Mon Dec 25 06:25:25 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P0              29W /  70W |   1925MiB / 15360MiB |     31%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Mon Dec 25 06:25:31 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
```

(continues on next page)

(continued from previous page)

```
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   65C    P0              71W /  70W |   2849MiB / 15360MiB |     58%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(21767, 1) (28480, 2)
CPU times: user 6.25 s, sys: 100 ms, total: 6.35 s
Wall time: 6.37 s
```

[ ]:
```
%%time
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  out = co_gdf.gfql([ n({'id': 1}), e_forward(hops=4)])
#! nvidia-smi
print(out._nodes.shape, out._edges.shape)
del co_gdf
del out
```

```
Mon Dec 25 06:26:04 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P0              29W /  70W |   1927MiB / 15360MiB |     36%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Mon Dec 25 06:26:13 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
```

(continues on next page)

(continued from previous page)

```
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   65C    P0              71W /  70W |   2931MiB / 15360MiB |     90%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(718640, 1) (2210961, 2)
CPU times: user 9.01 s, sys: 1.03 s, total: 10 s
Wall time: 9.84 s
```

[ ]:
```
%%time
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  out = co_gdf.gfql([ n({'id': 1}), e_forward(hops=5)])
#! nvidia-smi
print(out._nodes.shape, out._edges.shape)
del co_gdf
del out
```

```
Mon Dec 25 06:27:18 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   60C    P0              29W /  70W |   1927MiB / 15360MiB |     28%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Mon Dec 25 06:27:57 2023
+---------------------------------------------------------------------------------------+
```

(continues on next page)

(continued from previous page)

```
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   72C    P0              43W /  70W |   4351MiB / 15360MiB |    100%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(3041556, 1) (47622917, 2)
CPU times: user 34.9 s, sys: 4.76 s, total: 39.6 s
Wall time: 39.2 s
```

[ ]:
```
%%time
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
out = co_gdf.gfql([ n({'id': 1}), e_forward(hops=6)])._nodes
print(out.shape)
del co_gdf
del out
```

[ ]:
```
#!lscpu
```

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         46 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  2
  On-line CPU(s) list:   0,1
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU @ 2.20GHz
    CPU family:          6
    Model:               79
    Thread(s) per core:  2
    Core(s) per socket:  1
    Socket(s):           1
    Stepping:            0
    BogoMIPS:            4399.99
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov␣
↪pat pse36 clf
                         lush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm␣
↪constant_tsc rep_
                         good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni␣
```

(continues on next page)

(continued from previous page)

```
↪pclmulqdq ssse3 fm
                        a cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx␣
↪f16c rdrand hyp
                        ervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb␣
↪stibp fsgsb
                        ase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed␣
↪adx smap xsa
                        veopt arat md_clear arch_capabilities
Virtualization features:
  Hypervisor vendor:    KVM
  Virtualization type:  full
Caches (sum of all):
  L1d:                  32 KiB (1 instance)
  L1i:                  32 KiB (1 instance)
  L2:                   256 KiB (1 instance)
  L3:                   55 MiB (1 instance)
NUMA:
  NUMA node(s):         1
  NUMA node0 CPU(s):    0,1
Vulnerabilities:
  Gather data sampling: Not affected
  Itlb multihit:        Not affected
  L1tf:                 Mitigation; PTE Inversion
  Mds:                  Vulnerable; SMT Host state unknown
  Meltdown:             Vulnerable
  Mmio stale data:      Vulnerable
  Retbleed:             Vulnerable
  Spec rstack overflow: Not affected
  Spec store bypass:    Vulnerable
  Spectre v1:           Vulnerable: __user pointer sanitization and usercopy barriers␣
↪only; no swap
                        gs barriers
  Spectre v2:           Vulnerable, IBPB: disabled, STIBP: disabled, PBRSB-eIBRS: Not␣
↪affected
  Srbds:                Not affected
  Tsx async abort:      Vulnerable
```

[ ]:
```
#!free -h
```

```
               total        used        free      shared  buff/cache   available
Mem:            12Gi       717Mi       8.0Gi       1.0Mi       3.9Gi        11Gi
Swap:             0B          0B          0B
```

[ ]:
```
%%time
start_nodes = pd.DataFrame({'id': [1]})
#! nvidia-smi
for i in range(1):
  g2 = co_g.hop(
      nodes=start_nodes,
      direction='forward',
      hops=1)
```

(continues on next page)

(continued from previous page)

```
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
#del start_nodes
#del co_gdf
#del g2
```

```
Tue Dec 26 01:01:43 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   64C    P0              30W /  70W |   2821MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

[16]:
```
%%time
start_nodes = cudf.DataFrame({'id': [1]})
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  g2 = co_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=1)
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del co_gdf
del g2
```

```
Tue Dec 26 00:56:45 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   49C    P0              28W /  70W |   1923MiB / 15360MiB |     37%      Default |
|                                         |                      |                  N/A |
```

(continues on next page)

(continued from previous page)

```
+---------------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                             GPU Memory |
|        ID   ID                                                              Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
Tue Dec 26 00:56:47 2023
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   52C    P0              70W /  70W |   2819MiB / 15360MiB |     79%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                             GPU Memory |
|        ID   ID                                                              Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
(12, 1) (11, 2)
CPU times: user 1.6 s, sys: 37.3 ms, total: 1.64 s
Wall time: 1.84 s
```

[17]:
```
%%time
start_nodes = cudf.DataFrame({'id': [1]})
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  g2 = co_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=2)
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del co_gdf
del g2
```

```
Tue Dec 26 00:56:47 2023
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
```

(continues on next page)

(continued from previous page)

```
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   51C    P0              35W /  70W |   1923MiB / 15360MiB |     59%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Tue Dec 26 00:56:49 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   53C    P0              59W /  70W |   2821MiB / 15360MiB |     86%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(391, 1) (461, 2)
CPU times: user 2.32 s, sys: 58.5 ms, total: 2.38 s
Wall time: 2.51 s
```

[18]:
```
%%time
start_nodes = cudf.DataFrame({'id': [1]})
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  g2 = co_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=3)
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del co_gdf
del g2
```

```
Tue Dec 26 00:56:50 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   52C    P0              36W /  70W |   1925MiB / 15360MiB |     55%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Tue Dec 26 00:56:53 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   54C    P0              75W /  70W |   2825MiB / 15360MiB |     74%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(21767, 1) (28480, 2)
CPU times: user 3.04 s, sys: 63.6 ms, total: 3.1 s
Wall time: 3.25 s
```

[19]:
```
%%time
start_nodes = cudf.DataFrame({'id': [1]})
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  g2 = co_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=4)
```

(continues on next page)

(continued from previous page)

```
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del co_gdf
del g2
```

```
Tue Dec 26 00:56:53 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   54C    P0              36W /  70W |   1927MiB / 15360MiB |     54%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Tue Dec 26 00:56:58 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   56C    P0              38W /  70W |   2907MiB / 15360MiB |     89%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(718640, 1) (2210961, 2)
CPU times: user 4.58 s, sys: 309 ms, total: 4.89 s
Wall time: 5.02 s
```

[20]:
```
%%time
start_nodes = cudf.DataFrame({'id': [1]})
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
```

(continues on next page)

(continued from previous page)

```
#! nvidia-smi
for i in range(10):
  g2 = co_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=5)
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del co_gdf
del g2
```

```
Tue Dec 26 00:56:58 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   55C    P0              37W /  70W |   1925MiB / 15360MiB |     59%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Tue Dec 26 00:57:10 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   60C    P0              48W /  70W |   4325MiB / 15360MiB |     99%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
(3041556, 1) (47622917, 2)
CPU times: user 10.8 s, sys: 1.29 s, total: 12.1 s
Wall time: 12 s
```

[21]:
```
%%time
start_nodes = cudf.DataFrame({'id': [1]})
co_gdf = co_g.nodes(lambda g: cudf.DataFrame(g._nodes)).edges(lambda g: cudf.DataFrame(g.
↪_edges))
#! nvidia-smi
for i in range(10):
  g2 = co_gdf.hop(
      nodes=start_nodes,
      direction='forward',
      hops=6)
#! nvidia-smi
print(g2._nodes.shape, g2._edges.shape)
del start_nodes
del co_gdf
del g2
```

```
Tue Dec 26 00:57:10 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   59C    P0              38W /  70W |   1925MiB / 15360MiB |     44%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
Tue Dec 26 00:57:38 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   68C    P0              55W /  70W |   6445MiB / 15360MiB |     95%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
```

(continues on next page)

(continued from previous page)

```
+-----------------------------------------------------------------------------------------+
(3071927, 1) (117032738, 2)
CPU times: user 23.5 s, sys: 2.68 s, total: 26.2 s
Wall time: 28.2 s
```

[ ]:

10.8.3.5 Tutorial: GFQL remote mode

Running GFQL on remote servers helps with scenarios like large workloads benefiting from GPU acceleration depite no local GPU, when the data is already on a remote Graphistry server, and other team and production setting needs.

The following examples walk through several common scenarios:

- Uploading data and running GFQL remotely on it
- Binding to existing remote data and running GFQL remotely on it
- Control how much data is returned and in what format
- Control CPU vs GPU execution

See also the sibling tutorial for running arbitrary GPU Python remotely for even more powerful scenarios.

Setup

Note: Ensure https://hub.graphistry.com/docs/GFQL/gfql-api/ for the API user

Imports

[ ]:
```
import pandas as pd
import graphistry
from graphistry import n, e_undirected, e_forward

# Import Python API for cleaner syntax with let bindings
from graphistry.compute.ast import ref, let, call

graphistry.__version__
```

[2]:
```
graphistry.register(api=3, username='FILL_ME_IN', password='FILL_ME_IN', protocol='https
↪', server='hub.graphistry.com')
```

Data

Create an edge table. For simplicity, we will leave the nodes table implicit:

[3]:
```
e_df = pd.DataFrame({
    's': ['a', 'b', 'c'],
    'd': ['b', 'c', 'd'],
    'v': ['x', 'y', 'z'],
    'u': [2, 4, 6]
})

g = graphistry.edges(e_df, 's', 'd')
```

Upload data

Uploaded datasets have a nodes File, edges File, and combined graph Dataset. You can inspect these on your `Plottable` objects.

Remote-mode GFQL calls will automatically upload your graph if not already sent. If a table has already been recently in the session, the PyGraphistry client is smart enough to detect this and reuse the File ID handle instead of reuploading the data. However, in application code, we recommend explicitly uploading in your code flow to enable easier reuse and more predictable code flow.

[4]:
```
%%time
g2 = g.upload()

{
    'dataset_id': g2._dataset_id,
    'nodes_file_id': g2._nodes_file_id,
    'edges_file_id': g2._edges_file_id
}
```

```
CPU times: user 84.2 ms, sys: 13.2 ms, total: 97.4 ms
Wall time: 1.47 s
```

[4]:
```
{'dataset_id': '3a479d960595447e9e4f1b83ace969ed',
 'nodes_file_id': None,
 'edges_file_id': 'cd5bf7c37f1b4ced85a4d23b6f841be6'}
```

The edge table does not need to get re-uploaded

[5]:
```
%%time

# Much faster as g._edges is not re-uploaded, and instead g2._edges_file_id is reused
g2b = g.upload()

assert g2b._dataset_id != g2._dataset_id, "Each upload is a new Dataset object"
assert g2b._edges_file_id == g2._edges_file_id, "Dataframe files get automatically reused
↪"

{
    'dataset_id': g2._dataset_id,
    'nodes_file_id': g2._nodes_file_id,
```

(continues on next page)

(continued from previous page)

```
    'edges_file_id': g2._edges_file_id
}
```

```
CPU times: user 45 ms, sys: 1.61 ms, total: 46.6 ms
Wall time: 605 ms
```

[5]:
```
{'dataset_id': '3a479d960595447e9e4f1b83ace969ed',
 'nodes_file_id': None,
 'edges_file_id': 'cd5bf7c37f1b4ced85a4d23b6f841be6'}
```

Query remote data

Regular chain calls can be called in remote mode, and return back the resulting graph

[6]:
```
two_hop_query = [
    n({'id': 'a'}),
    e_forward(hops=2),
    n()
]
```

[7]:
```
%%time

two_hop_g = g2.gfql_remote(two_hop_query)
```

```
CPU times: user 37.9 ms, sys: 9.9 ms, total: 47.8 ms
Wall time: 613 ms
```

[8]:
```
two_hop_g._edges
```

[8]:

	s	d	v	u
0	a	b	x	2
1	b	c	y	4

[9]:
```
two_hop_g._nodes
```

[9]:

	id
0	a
1	b
2	c

[10]:
```
assert len(two_hop_g._edges) == len(g.gfql(two_hop_query)._edges), "Remote result should␣
↪match local results"
```

Ensure GPU mode in remote execution

Explicitly set the remote `engine=` configuration to `"cudf"` (GPU) or `"pandas"` (CPU), or leave unconfigured to let the runtime decide

[11]:
```
%%time
two_hop_g_gpu1 = g2.gfql_remote(two_hop_query, engine='cudf')
```

```
CPU times: user 48.4 ms, sys: 0 ns, total: 48.4 ms
Wall time: 598 ms
```

[12]:
```
%%time
two_hop_g_cpu1 = g2.gfql_remote(two_hop_query, engine='pandas')
```

```
CPU times: user 50 ms, sys: 744 µs, total: 50.8 ms
Wall time: 590 ms
```

You can move the results to a local if available:

[13]:
```
try:
    two_hop_g_gpu1 = two_hop_g_gpu1.to_cudf()
    print(type(two_hop_g_gpu1._edges))
except Exception as e:
    print('Error moving to a local GPU, do you have a GPU and is cudf configured?')
    print(e)
```

```
<class 'cudf.core.dataframe.DataFrame'>
```

Fetch only subsets of the data

You can fetch only subsets of the remote data:

Shape: Check result counts without downloading the graph

Often the important aspect is whether or not a search had hits, and you rather not pay the performance penalty of transfering all the hits. In these cases, switch to `gfql_remote_shape()`:

[14]:
```
g2.chain_remote_shape(two_hop_query)
```

[14]:

	kind	rows	cols
0	nodes	3	1
1	edges	2	4

Return only nodes

[15]:
```
%%time

two_hops_nodes = g2.gfql_remote(two_hop_query, output_type="nodes")

assert two_hops_nodes._edges is None, "No edges returned"

two_hops_nodes._nodes
```

```
CPU times: user 51.8 ms, sys: 74 µs, total: 51.9 ms
Wall time: 637 ms
```

[15]:
```
  id
0  a
1  b
2  c
```

Return only edges

[16]:
```
%%time

two_hops_edges = g2.gfql_remote(two_hop_query, output_type="edges")

assert two_hops_edges._nodes is None, "No nodes returned"

two_hops_edges._edges
```

```
CPU times: user 54.1 ms, sys: 3.58 ms, total: 57.6 ms
Wall time: 609 ms
```

[16]:
```
   s  d  v  u
0  a  b  x  2
1  b  c  y  4
```

Return subset of attributes

Whether returning both nodes and edges, or only one type of these, you can also pick a subset of the columns to fetch back. For example, you may only desire the IDs, as the full data may be prohibitively large, and you may already have the relevant data locally.

[17]:
```
%%time

two_hops_IDs_g = g2.gfql_remote(two_hop_query, node_col_subset=['id'], edge_col_subset=[
↪'s', 'd'])
```

```
CPU times: user 47.3 ms, sys: 7.85 ms, total: 55.1 ms
Wall time: 609 ms
```

[18]:
```
two_hops_IDs_g._nodes
```

[18]:
```
  id
0  a
1  b
2  c
```

[19]:
```
assert 'v' not in two_hops_IDs_g._edges.columns, "Only columns in the subset are returned
↪"

two_hops_IDs_g._edges
```

[19]:
```
   s  d
0  a  b
1  b  c
```

Bind, use, and fetch existing remote data

When a remote graph dataset ID is already known, bind to it and use it

Locally bind to remote data

[20]:
```
%%time

g3_bound = graphistry.bind(dataset_id=g2._dataset_id)

{
    'dataset_id': g3_bound._dataset_id,
    'has local nodes': g3_bound._nodes is not None,
    'has local edges': g3_bound._edges is not None
}
```

```
CPU times: user 125 µs, sys: 34 µs, total: 159 µs
Wall time: 161 µs
```

[20]:
```
{'dataset_id': '5990e1142056407ea3b13639521ffb56',
 'has local nodes': False,
 'has local edges': False}
```

Remotely query remote data

Use `chain_remote()` and `gfql_remote_shape()` as usual:

[21]:
```
g3_bound.chain_remote_shape(two_hop_query)
```

[21]:
```
    kind  rows  cols
0  nodes     3     1
1  edges     2     4
```

Fetch remote data

Use `gfql_remote()` to fetch the nodes and edges table. Note that the below takes care to fetch nodes that are not connected to any edges.

[22]:
```
%%time

remote_g_nodes = g3_bound.gfql_remote([n()], output_type='nodes')
remote_g_edges = g3_bound.gfql_remote([e_undirected()], output_type='edges')

g3_fetched_g = (graphistry
    .nodes(remote_g_nodes._nodes, 'id')
```

(continues on next page)

(continued from previous page)

```
    .edges(remote_g_edges._edges,  's', 'd')
)
```

```
CPU times: user 116 ms, sys: 10.5 ms, total: 127 ms
Wall time: 1.33 s
```

[23]:
```
print('Node ID column:', g3_fetched_g._node)
g3_fetched_g._nodes
```

```
Node ID column: id
```

[23]:

	id
0	a
1	b
2	c
3	d

[24]:
```
print('Edge src/dst columns:', g3_fetched_g._source, g3_fetched_g._destination)
g3_fetched_g._edges
```

```
Edge src/dst columns: s d
```

[24]:

	s	d	v	u
0	a	b	x	2
1	b	c	y	4
2	c	d	z	6

[ ]:
```
```

Combining Let Bindings with Call Operations

Let bindings in GFQL allow you to create named intermediate results and compose complex operations. When combined with call operations in remote mode, you can orchestrate sophisticated graph analyses entirely on the server, minimizing data transfer and leveraging server-side GPU acceleration.

Example 1: PageRank Analysis with Filtering

This example demonstrates using let bindings to: 1. Compute PageRank scores 2. Filter high-value nodes 3. Extract subgraphs around important nodes 4. Return results for visualization

[ ]:
```
# Create a more complex graph for demonstration
complex_edges = pd.DataFrame({
    's': ['a', 'b', 'c', 'd', 'e', 'f', 'a', 'b', 'c', 'd'],
    'd': ['b', 'c', 'd', 'e', 'f', 'a', 'c', 'd', 'e', 'f'],
    'weight': [1, 2, 1, 3, 1, 2, 1, 2, 1, 1],
    'type': ['follow', 'mention', 'follow', 'follow', 'mention', 'follow', 'mention',
↪'follow', 'follow', 'mention']
})

g_complex = graphistry.edges(complex_edges, 's', 'd').upload()
print(f"Uploaded graph with {len(complex_edges)} edges")
```

[ ]:
```
%%time

# Define a complex query using Python API for cleaner syntax
pagerank_analysis_query = let({
    # Step 1: Compute PageRank scores
    'with_pagerank': call('compute_pagerank', {}),

    # Step 2: Filter nodes with high PageRank scores
    'important_nodes': ref('with_pagerank', [
        n({'filter': {'gte': [{'col': 'pagerank'}, 0.15]}})
    ]),

    # Step 3: Get 1-hop neighborhoods of important nodes
    'important_neighborhoods': ref('with_pagerank', [
        n({'filter': {'gte': [{'col': 'pagerank'}, 0.15]}}),
        e_undirected({'hops': 1}),
        n()
    ])
})

# Note: The 'in' clause is automatically the last binding when using Python let()
# To specify a different output, pass it as second argument: let(bindings, 'output_name')

# Execute the query remotely - chain_remote accepts Python objects directly!
result = g_complex.gfql_remote([pagerank_analysis_query])

print(f"Result has {len(result._nodes)} nodes and {len(result._edges)} edges")
print("\nNodes with PageRank scores:")
print(result._nodes)
```

Example 2: Multi-Stage Analysis with Different Edge Types

This example shows how to use let bindings to analyze different edge types separately and combine the results:

Python API vs JSON Format Comparison

The examples above use the clean Python API. For reference, here's what the equivalent JSON format looks like:

[ ]:
```
# Comparison: Python API vs JSON format

# Clean Python API (what we use above):
python_query = let({
    'data': call('compute_pagerank', {}),
    'filtered': ref('data', [
        n({'filter': {'gte': [{'col': 'pagerank'}, 0.15]}})
    ])
})
```

(continues on next page)

(continued from previous page)

```
# Equivalent verbose JSON format:
json_query = {
    'let': {
        'data': {
            'type': 'Call',
            'function': 'compute_pagerank',
            'params': {}
        },
        'filtered': {
            'type': 'Ref',
            'ref': 'data',
            'chain': [{
                'type': 'Node',
                'filter_dict': {
                    'filter': {'gte': [{'col': 'pagerank'}, 0.15]}
                }
            }]
        }
    },
    'in': {'type': 'Ref', 'ref': 'filtered', 'chain': []}
}

# Both work with chain_remote:
# result = g.gfql_remote([python_query])  # Clean!
# result = g.gfql_remote([json_query])    # Verbose but equivalent

print("Python object converts to JSON:")
print(python_query.to_json())
```

[ ]:
```
%%time

# Analyze different edge types using clean Python API
edge_type_analysis = let({
    # Analyze follow edges
    'follow_network': e_undirected({
        'filter': {'eq': [{'col': 'type'}, 'follow']}
    }),

    # Compute centrality on follow network
    'follow_centrality': ref('follow_network', [
        call('compute_degree_centrality', {})
    ]),

    # Get nodes that are highly connected in the follow network
    'influential_nodes': ref('follow_centrality', [
        n({'filter': {'gte': [{'col': 'degree_centrality'}, 0.5]}}),
        e_undirected({'hops': 1}),
        n()
    ])
})
```

(continues on next page)

(continued from previous page)

```
# Execute remotely
influential_result = g_complex.gfql_remote([edge_type_analysis])

print(f"Found {len(influential_result._nodes)} influential nodes")
print(f"Connected by {len(influential_result._edges)} edges")
print("\nInfluential nodes with centrality scores:")
print(influential_result._nodes)
```

Example 3: Conditional Analysis with Let Bindings

This example demonstrates using let bindings to perform conditional analysis based on graph properties:

[ ]:
```
%%time

# Complex analysis with multiple algorithms using Python API
comprehensive_analysis = let({
    # Base graph with PageRank computation
    'enriched_graph': call('compute_pagerank', {}),

    # Add centrality metrics
    'with_centrality': ref('enriched_graph', [
        call('compute_degree_centrality', {})
    ]),

    # Find bridge nodes (high PageRank, low-medium centrality)
    'bridge_nodes': ref('with_centrality', [
        n({
            'filter': {
                'and': [
                    {'gte': [{'col': 'pagerank'}, 0.1]},
                    {'lte': [{'col': 'degree_centrality'}, 0.7]}
                ]
            }
        })
    ]),

    # Find hub nodes (high degree centrality)
    'hub_nodes': ref('with_centrality', [
        n({'filter': {'gte': [{'col': 'degree_centrality'}, 0.7]}})
    ]),

    # Get connections between bridges and hubs
    'critical_paths': ref('with_centrality', [
        n({
            'filter': {
                'and': [
                    {'gte': [{'col': 'pagerank'}, 0.1]},
                    {'lte': [{'col': 'degree_centrality'}, 0.7]}
                ]
```

(continues on next page)

(continued from previous page)

```
            }
        }),
        e_forward(),
        n({'filter': {'gte': [{'col': 'degree_centrality'}, 0.7]}})
    ])
})

# Execute remotely with GPU acceleration
critical_paths_result = g_complex.gfql_remote([comprehensive_analysis], engine='cudf')

print(f"Critical paths network: {len(critical_paths_result._nodes)} nodes and
↪{len(critical_paths_result._edges)} edges")

# Check if we got results
if len(critical_paths_result._nodes) > 0:
    print("\nCritical path nodes:")
    print(critical_paths_result._nodes)
else:
    print("\nNo critical paths found with current thresholds")
```

Example 4: Visualization-Ready Analysis

This example shows how to prepare data for visualization by enriching it with multiple metrics and creating a focused subgraph:

```
[ ]: %%time

# Prepare visualization-ready data with all enrichments
viz_prep_query = {
    'let': {
        # Compute all metrics - sequential operations
        'with_pagerank': {
            'call': {'method': 'compute_pagerank', 'args': [], 'kwargs': {}}
        },

        'with_metrics': {
            'type': 'Ref',
            'ref': 'with_pagerank',
            'chain': [
                {'call': {'method': 'compute_degree_centrality', 'args': [], 'kwargs': {}
↪}},
                # Add node colors based on PageRank
                {
                    'call': {
                        'method': 'nodes',
                        'args': [],
                        'kwargs': {
                            'assign': {
                                'node_color': {
```

(continues on next page)

(continued from previous page)

```
                                'case': [
                                    {
                                        'when': {'gte': [{'col': 'pagerank'}, 0.2]},
                                        'then': 65280  # Green for high PageRank
                                    },
                                    {
                                        'when': {'gte': [{'col': 'pagerank'}, 0.15]},
                                        'then': 16776960  # Yellow for medium
                                    }
                                ],
                                'else': 16711680  # Red for low
                            },
                            'node_size': {
                                'mul': [
                                    {'col': 'degree_centrality'},
                                    50  # Scale factor
                                ]
                            }
                        }
                    }
                }
            }
        ]
    },

    # Add edge styling based on type and weight
    'styled_graph': {
        'type': 'Ref',
        'ref': 'with_metrics',
        'chain': [
            {
                'call': {
                    'method': 'edges',
                    'args': [],
                    'kwargs': {
                        'assign': {
                            'edge_color': {
                                'case': [
                                    {
                                        'when': {'eq': [{'col': 'type'}, 'follow']},
                                        'then': 255  # Blue for follows
                                    }
                                ],
                                'else': 16711935  # Magenta for mentions
                            },
                            'edge_weight': {
                                'col': 'weight'
                            }
                        }
                    }
                }
            }
```

(continues on next page)

(continued from previous page)

```
            ]
        },

        # Focus on top nodes and their connections
        'viz_subgraph': {
            'type': 'Ref',
            'ref': 'styled_graph',
            'chain': [
                {
                    'n': {
                        'filter': {
                            'or': [
                                {'gte': [{'col': 'pagerank'}, 0.15]},
                                {'gte': [{'col': 'degree_centrality'}, 0.6]}
                            ]
                        }
                    }
                },
                {'e_undirected': {'hops': 1}},
                {'n': {}}
            ]
        }
    },

    'in': {'type': 'Ref', 'ref': 'viz_subgraph', 'chain': []}
}

# Get visualization-ready data
viz_result = g_complex.gfql_remote([viz_prep_query])

print(f"Visualization subgraph: {len(viz_result._nodes)} nodes, {len(viz_result._edges)}
↪ edges")
print("\nNodes with visualization attributes:")
print(viz_result._nodes)
print("\nEdges with styling:")
print(viz_result._edges)

# Ready to visualize
# viz_result.plot()  # Uncomment to create visualization
```

Key Benefits of Let Bindings with Remote Calls

1. **Server-Side Orchestration**: All operations happen on the server, minimizing data transfer
2. **Named Intermediate Results**: Create readable, reusable steps in complex analyses
3. **GPU Acceleration**: Leverage server GPU for compute-intensive operations like PageRank
4. **Composability**: Build complex workflows from simple building blocks
5. **Efficiency**: Avoid redundant computations by reusing named results

When working with large graphs, this approach is particularly powerful as it allows you to: - Perform multiple analyses without downloading intermediate results - Chain together different algorithms and filters - Prepare

visualization-ready data entirely on the server - Return only the final, filtered results you need

10.8.3.6 Tutorial: GPU Python remote mode

Running GPU Python on remote servers helps with scenarios like large workloads benefiting from GPU acceleration depite no local GPU, when the data is already on a remote Graphistry server, and other team and production setting needs.

The following examples walk through several common scenarios:

- Uploading data and running Python remotely on it
- Binding to existing remote data and running Python remotely on it
- Control how much data is returned
- Control CPU vs GPU execution

See also the sibling tutorial for running pure GFQL queries remotely for typical scenarios. When viable, we recommend sticking to GFQL for safety, clarity, and performance reasons.

Setup

Note: Ensure the remote Python endpoint is enabled on the server, and https://hub.graphistry.com/docs/Python/python-api/

Imports

[6]:
```
import pandas as pd
import graphistry
from graphistry import n, e_undirected, e_forward
graphistry.__version__
```

[6]:
```
'0+unknown'
```

[23]:
```
graphistry.register(api=3, username='FILL_ME_IN', password='FILL_ME_IN', protocol='https
↪', server='hub.graphistry.com')
```

Data

[8]:
```
e_df = pd.DataFrame({
    's': ['a', 'b', 'c'],
    'd': ['b', 'c', 'd'],
    'v': ['x', 'y', 'z'],
    'u': [2, 4, 6]
})

g = graphistry.edges(e_df, 's', 'd')
```

Upload data

We will upload the graph.

See the GFQL remote mode tutorial for how to use `g2 = graphistry.bind(dataset_id=my_id)` for existing remote data.

[9]:
```
%%time
g2 = g.upload()

{
    'dataset_id': g2._dataset_id,
    'nodes_file_id': g2._nodes_file_id,
    'edges_file_id': g2._edges_file_id
}
```

```
CPU times: user 70.1 ms, sys: 1.24 ms, total: 71.3 ms
Wall time: 2.03 s
```

[9]:
```
{'dataset_id': '0a56aa27ec1e4112b1458e960dc6f674',
 'nodes_file_id': None,
 'edges_file_id': '271a00f639a748fcaaaf620437bcd0f2'}
```

Remotely query the data

Define your remote function as a top-level method `def task(g): ...`, or pass in a named method (`Callable`). If the passed-in `Callable` does not have name `task`, the Python client will try to rename it to `task` for you.

The remote Python endpoint can return graphs, dataframes, and JSON objects in a way that plays nicely with Python type checking. Hint which by using the different calling forms:

- `python_remote_g()`: For returning a `Plottable` (graph)
- `python_remote_json()`: For returning JSON values
- `python_remote_table()`: For returning a `pd.DataFrame`

By default, the parquet data format is used for safely and efficiently transporting graphs and dataframes return types, and JSON format transport for JSON return types.

Return a graph

The below shows two aspects:

- Code provided as a Python source string defining a top-level function `def task(g: Plottable) -> Plottable`
- Remote invocation `python_remote_g()` that implies that `task()` will return a `Plottable` (graph)

[10]:
```
g3 = g2.python_remote_g("""

from graphistry import Plottable

def task(g: Plottable) -> Plottable:
  '''
  Fill in the nodes table based on the edges table and return the combined
```

(continues on next page)

(continued from previous page)

```
  '''

  return g.materialize_nodes()

""")

g3._edges
```

[10]:
```
   s  d  v  u
0  a  b  x  2
1  b  c  y  4
2  c  d  z  6
```

[11]:
```
g3._nodes
```

[11]:
```
  id
0  a
1  b
2  c
3  d
```

Run a local Callable remotely

You can also pass self-contained python functions for code that is easier to read and works with your developer and automation tools

Note that only the source code is transferred to the server; there should be no associated local references

[12]:
```
def materialize_nodes(g):
    return g.materialize_nodes()

g3b = g2.python_remote_g(materialize_nodes)

g3b._nodes
```

[12]:
```
  id
0  a
1  b
2  c
3  d
```

Return a table

For remotely calling functions that return dataframes, instead call `python_remote_table()`:

[13]:
```
nodes_df = g2.python_remote_table("""

import pandas as pd
from graphistry import Plottable

def task(g: Plottable) -> pd.DataFrame:
```

(continues on next page)

(continued from previous page)

```
  '''
  Fill in the nodes table based on the edges table and return it
  '''

  return g.materialize_nodes()._nodes

""")

nodes_df
```

[13]:
```
  id
0  a
1  b
2  c
3  d
```

And as before, you can also pass in a self-contained Python function:

[14]:
```
def g_to_materialized_nodes(g):
    return g.materialize_nodes()._nodes

nodes_df = g2.python_remote_table(g_to_materialized_nodes)

nodes_df
```

[14]:
```
  id
0  a
1  b
2  c
3  d
```

Return arbitrary JSON

The remote Python endpoint also supports returning arbitrary JSON-format data via `python_remote_json()`:

[15]:
```
shape = g2.python_remote_json("""

from typing import Dict
from graphistry import Plottable

def task(g: Plottable) -> Dict[str, int]:
  '''
  Fill in the nodes table based on the edges table and return it
  '''

  return {'num_edges': len(g._edges), 'num_nodes': len(g.materialize_nodes()._nodes)}
""")

shape['num_nodes'], shape['num_edges']
```

[15]:
```
(4, 3)
```

And by passing in a self-contained Python function:

[16]:
```
def g_to_shape(g):
  """
  Fill in the nodes table based on the edges table and return it
  """

  return {'num_edges': len(g._edges), 'num_nodes': len(g.materialize_nodes()._nodes)}


g2.python_remote_json(g_to_shape)
```

[16]:
```
{'num_edges': 3, 'num_nodes': 4}
```

Enforce GPU mode

Override `engine="cudf"` for GPU mode and `engine="pandas"` for CPU mode:

[20]:
```
def report_types(g):
    return {
        'edges': str(type(g._edges)),
        'nodes': str(type(g.materialize_nodes()._nodes))
    }

g2.python_remote_json(report_types)
```

[20]:
```
{'edges': "<class 'cudf.core.dataframe.DataFrame'>",
 'nodes': "<class 'cudf.core.dataframe.DataFrame'>"}
```

[21]:
```
def report_types(g):
    return {
        'edges': str(type(g._edges)),
        'nodes': str(type(g.materialize_nodes()._nodes))
    }

g2.python_remote_json(report_types, engine='pandas')
```

[21]:
```
{'edges': "<class 'pandas.core.frame.DataFrame'>",
 'nodes': "<class 'pandas.core.frame.DataFrame'>"}
```

[22]:
```
def report_types(g):
    return {
        'edges': str(type(g._edges)),
        'nodes': str(type(g.materialize_nodes()._nodes))
    }

g2.python_remote_json(report_types, engine='cudf')
```

[22]:
```
{'edges': "<class 'cudf.core.dataframe.DataFrame'>",
 'nodes': "<class 'cudf.core.dataframe.DataFrame'>"}
```

[ ]:

[ ]:

10.8.4 GPU

10.8.4.1 Visual GPU Log Analytics Part I: CPU Baseline in Python Pandas

Graphistry is great – Graphistry and RAPIDS/BlazingDB is better!

This tutorial series visually analyzes Zeek/Bro network connection logs using different compute engines:

- Part I: *CPU Baseline in Python Pandas*
- Part II: *GPU Dataframes with RAPIDS Python cudf bindings*
- Part III: GPU SQL - deprecated as Dask-SQL replaced BlazingSQL in the RAPIDS ecosystem
- Part IV: *GPU ML with RAPIDS cuML UMAP and PyGraphistry*
- *Graphistry cuGraph bindings*

Part I Contents:

Time using CPU-based Python Pandas and Graphistry for a full ETL & visual analysis flow:

1. Load data
2. Analyze data
3. Visualize data

[ ]:
```
#!pip install graphistry -q


import pandas as pd

import graphistry
graphistry.__version__

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')

# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

1. Load data

[ ]:
```
%%time
# download data
#!if [ ! -f conn.log ]; then \
#    curl https://www.secrepo.com/maccdc2012/conn.log.gz | gzip -d > conn.log; \
#fi
```

[ ]:
```
#!head -n 3 conn.log
```

[ ]:
```
# OPTIONAL: For slow or limited devices, work on a subset:
LIMIT = 1200000
```

[ ]:
```
%%time
df = pd.read_csv("./conn.log", sep="\t", header=None,
                 names=["time", "uid", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_p
↪", "proto", "service",
                        "duration", "orig_bytes", "resp_bytes", "conn_state", "local_orig
↪", "missed_bytes",
                        "history", "orig_pkts", "orig_ip_bytes", "resp_pkts", "resp_ip_
↪bytes", "tunnel_parents"],
                 na_values=['-'], index_col=False, nrows=LIMIT)
```

[ ]:
```
df.sample(3)
```

2. Analyze Data

Summarize network activities between every communicating src/dst IP, split by connection state

[ ]:
```
df_summary = df\
.assign(
    sum_bytes=df.apply(lambda row: row['orig_bytes'] + row['resp_bytes'], axis=1))\
.groupby(['id.orig_h', 'id.resp_h', 'conn_state'])\
.agg({
    'time': ['min', 'max', 'size'],
    'id.resp_p':  ['nunique'],
    'uid': ['nunique'],
    'duration':   ['min', 'max', 'mean'],
    'orig_bytes': ['min', 'max', 'sum', 'mean'],
    'resp_bytes': ['min', 'max', 'sum', 'mean'],
    'sum_bytes':  ['min', 'max', 'sum', 'mean']
}).reset_index()
```

[ ]:
```
df_summary.columns = [' '.join(col).strip() for col in df_summary.columns.values]
df_summary = df_summary\
.rename(columns={'time size': 'count'})\
.assign(
    conn_state_uid=df_summary.apply(lambda row: row['id.orig_h'] + '_' + row['id.resp_h
↪'] + '_' + row['conn_state'], axis=1))
```

[ ]:
```
print ('# rows', len(df_summary))
df_summary.sample(3)
```

3. Visualize data

- Nodes:
  - IPs
  - Bigger when more sessions (split by connection state) involving them
- Edges:
  - src_ip -> dest_ip, split by connection state

[ ]:
```
hg = graphistry.hypergraph(
    df_summary,
    ['id.orig_h', 'id.resp_h'],
    direct=True,
    opts={
        'CATEGORIES': {
            'ip': ['id.orig_h', 'id.resp_h']
        }
    })
```

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

Next Steps

- Part I: *CPU Baseline in Python Pandas*
- Part II: *GPU Dataframe with RAPIDS Python cudf bindings*
- Part III: GPU SQL - deprecated as Dask-SQL replaced BlazingSQL in the RAPIDS ecosystem
- Part IV: *GPU ML with RAPIDS cuML UMAP and PyGraphistry*
- *Graphistry cuGraph bindings*

[ ]:

[ ]:

10.8.4.2 Visual GPU Log Analytics Part II: GPU dataframes with RAPIDS Python cudf bindings

Graphistry is great – Graphistry and RAPIDS/BlazingDB is better!

This tutorial series visually analyzes Zeek/Bro network connection logs using different compute engines:

- Part I: *CPU Baseline in Python Pandas*
- Part II: *GPU Dataframe with RAPIDS Python cudf bindings*
- Part III: GPU SQL - deprecated as Dask-SQL replaced BlazingSQL in the RAPIDS ecosystem
- Part IV: *GPU ML with RAPIDS cuML UMAP and PyGraphistry*
- *Graphistry cuGraph bindings*

Part II Contents:

Time using GPU-based RAPIDS Python cudf bindings and Graphistry for a full ETL & visual analysis flow:

1. Load data
2. Analyze data
3. Visualize data

TIP: If you get out of memory errors, you usually must restart the kernel & refresh the page

[1]:
```
#!pip install graphistry -q

import pandas as pd
import numpy as np
import cudf

import graphistry
graphistry.__version__

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')

# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

[2]:
```
# check nvidia config
!nvidia-smi
```

```
Tue Nov 22 06:20:14 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.141.03   Driver Version: 470.141.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   38C    P0    27W /  70W |   8037MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

1. Load data

[16]:
```
%%time
# download data
#!if [ ! -f conn.log ]; then \
#    curl https://www.secrepo.com/maccdc2012/conn.log.gz | gzip -d > conn.log; \
#fi
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  523M  100  523M    0     0  31.5M      0  0:00:16  0:00:16 --:--:-- 30.0M
CPU times: user 240 ms, sys: 105 ms, total: 345 ms
Wall time: 17.3 s
```

[4]:
```
!head -n 3 conn.log
```

```
1331901000.000000       CCUIP21wTjqkj8ZqX5      192.168.202.79  50463   192.168.229.251␣
↪80      tcp     -       -       -       -       SH      -       0       Fa      1       ␣
↪ 52      1       52      (empty)
1331901000.000000       Csssjd3tX0yOTPDpng      192.168.202.79  46117   192.168.229.254␣
↪443     tcp     -       -       -       -       SF      -       0       dDafFr  3       ␣
↪ 382     9       994     (empty)
1331901000.000000       CHEt7z3AzG4gyCNgci      192.168.202.79  50465   192.168.229.251␣
↪80      tcp     http    0.010000        166     214     SF      -       0       ␣
↪ShADfFa 4       382     3       382     (empty)
```

[5]:
```
# OPTIONAL: For slow or limited devices, work on a subset:
LIMIT = 12000000
```

[6]:
```
%%time
cdf = cudf.read_csv("./conn.log", sep="\t", header=None,
                    names=["time", "uid", "id.orig_h", "id.orig_p", "id.resp_h", "id.resp_
↪p", "proto", "service",
                           "duration", "orig_bytes", "resp_bytes", "conn_state", "local_orig
↪", "missed_bytes",
                           "history", "orig_pkts", "orig_ip_bytes", "resp_pkts", "resp_ip_
↪bytes", "tunnel_parents"],
                    dtype=['date', 'str', 'str', 'int', 'str', 'int', 'str', 'str',
                           'int', 'int', 'int', 'str', 'str', 'int',
                           'str', 'int', 'int', 'int', 'int', 'str'],
                    na_values=['-'], index_col=False, nrows=LIMIT)
```

```
CPU times: user 2.02 s, sys: 676 ms, total: 2.69 s
Wall time: 2.69 s
```

[7]:
```
#fillna
for c in cdf.columns:
    if c in ['uid', 'id.orig_h', 'id.resp_h', 'proto', 'service', 'conn_state', 'history
↪', 'tunnel_parents', 'local_orig']:
        continue
    cdf[c] = cdf[c].fillna(0)
```

[8]:
```
print('# rows', len(cdf))
cdf.head(3)
```

```
# rows 12000000
```

[8]:
```
                          time                 uid       id.orig_h  \
0 1753-11-29 22:43:41.128654848  CCUIP21wTjqkj8ZqX5  192.168.202.79
1 1753-09-09 22:43:41.128654848  Csssjd3tX0yOTPDpng  192.168.202.79
2 1753-10-06 22:43:41.128654848  CHEt7z3AzG4gyCNgci  192.168.202.79

   id.orig_p        id.resp_h  id.resp_p proto service  duration  orig_bytes  \
0      50463  192.168.229.251         80   tcp    <NA>         0           0
1      46117  192.168.229.254        443   tcp    <NA>         0           0
2      50465  192.168.229.251         80   tcp    http         0         166

   resp_bytes conn_state local_orig  missed_bytes  history  orig_pkts  \
0           0         SH       <NA>             0       Fa          1
1           0         SF       <NA>             0   dDafFr          3
2         214         SF       <NA>             0  ShADfFa          4

   orig_ip_bytes  resp_pkts  resp_ip_bytes tunnel_parents
0             52          1             52        (empty)
1            382          9            994        (empty)
2            382          3            382        (empty)
```

2. Analyze Data

- Summarize network activities between every communicating src/dst IP, split by connection state
- RAPIDS currently fails when exceeding GPU memory, so limit workload size as needed

[9]:
```
%%time
cdf_summary=(cdf
    .pipe(lambda df: df.assign(sum_bytes=df.orig_bytes + df.resp_bytes))
    .groupby(['id.orig_h', 'id.resp_h', 'conn_state'])
    .agg({
        'time': ['min', 'max', 'count'],
        'id.resp_p':  ['count'],
        'uid': ['count'],
        'duration':   ['min', 'max', 'mean'],
        'orig_bytes': ['min', 'max', 'sum', 'mean'],
        'resp_bytes': ['min', 'max', 'sum', 'mean'],
        'sum_bytes':  ['min', 'max', 'sum', 'mean']
}))
```

```
CPU times: user 1.19 s, sys: 68.2 ms, total: 1.26 s
Wall time: 1.26 s
```

[10]:
```
print('# rows', len(cdf_summary))
cdf_summary.head(3).to_pandas()
```

```
# rows 50140
```

[10]:

```
                                                                time  \
                                                                 min
id.orig_h   id.resp_h       conn_state
0.0.0.0     255.255.255.255 S0         1753-12-19 22:43:41.128654848
10.10.10.10 10.255.255.255  S0         1755-03-06 22:43:41.128654848
            192.168.202.78  OTH        1753-11-09 22:43:41.128654848

                                                                      \
                                                                max count
id.orig_h   id.resp_h       conn_state
0.0.0.0     255.255.255.255 S0         1774-05-02 22:43:41.128654848    87
10.10.10.10 10.255.255.255  S0         1769-12-03 22:43:41.128654848    27
            192.168.202.78  OTH        1773-12-17 22:43:41.128654848    34

                                    id.resp_p   uid duration       \
                                        count count      min  max
id.orig_h   id.resp_h       conn_state
0.0.0.0     255.255.255.255 S0                87    87        0  142
10.10.10.10 10.255.255.255  S0                27    27        0   52
            192.168.202.78  OTH               34    34        0   95

                                                 orig_bytes               \
                                               mean        min   max    sum
id.orig_h   id.resp_h       conn_state
0.0.0.0     255.255.255.255 S0           17.620690          0  9323  74099
10.10.10.10 10.255.255.255  S0            5.296296          0  1062   4558
            192.168.202.78  OTH          10.352941          0     0      0

                                                      resp_bytes               \
                                                    mean        min max sum mean
id.orig_h   id.resp_h       conn_state
0.0.0.0     255.255.255.255 S0           851.712644          0   0   0  0.0
10.10.10.10 10.255.255.255  S0           168.814815          0   0   0  0.0
            192.168.202.78  OTH            0.000000          0   0   0  0.0

                                    sum_bytes
                                          min   max    sum        mean
id.orig_h   id.resp_h       conn_state
0.0.0.0     255.255.255.255 S0                0  9323  74099  851.712644
10.10.10.10 10.255.255.255  S0                0  1062   4558  168.814815
            192.168.202.78  OTH               0     0      0    0.000000
```

3. Visualize data

- Nodes:
  - IPs
  - Bigger when more sessions (split by connection state) involving them
- Edges:
  - src_ip -> dest_ip, split by connection state

[13]:
```
# flatten multi-index
cdfs2 = cdf_summary.copy(deep=False).reset_index()
cdfs2.columns = [''.join(c) for c in cdfs2.columns]
cdfs2.columns
```

[13]:
```
Index(['id.orig_h', 'id.resp_h', 'conn_state', 'timemin', 'timemax',
       'timecount', 'id.resp_pcount', 'uidcount', 'durationmin', 'durationmax',
       'durationmean', 'orig_bytesmin', 'orig_bytesmax', 'orig_bytessum',
       'orig_bytesmean', 'resp_bytesmin', 'resp_bytesmax', 'resp_bytessum',
       'resp_bytesmean', 'sum_bytesmin', 'sum_bytesmax', 'sum_bytessum',
       'sum_bytesmean'],
      dtype='object')
```

[14]:
```
hg = graphistry.hypergraph(
    cdfs2,
    ['id.orig_h', 'id.resp_h'],
    direct=True,
    opts={
        'CATEGORIES': {
            'ip': ['id.orig_h', 'id.resp_h']
        }
    }, engine='cudf') # or use default of 'pandas' with cdfs2.to_pandas() above
```

```
# links 50140
# events 50140
# attrib entities 5048
```

[15]:
```
hg['graph'].plot()
```

[15]:
```
<IPython.core.display.HTML object>
```

Next Steps

- Part I: *CPU Baseline in Python Pandas*
- Part II: *GPU Dataframe with RAPIDS Python cudf bindings*
- Part III: GPU SQL - deprecated as Dask-SQL replaced BlazingSQL in the RAPIDS ecosystem
- Part IV: *GPU ML with RAPIDS cuML UMAP and PyGraphistry*
- *Graphistry cuGraph bindings*

[ ]:

10.8.4.3 GPU UMAP

UMAP is a popular method of dimensionality reduction, a helpful technique for meaningful analysis of large, complex datasets. Graphistry provides convenient bindings for working with `cuml.UMAP`.

UMAP is: * interested in the number of nearest numbers * non-linear, unlike longstanding methods such as PCA * non-scaling, which keep calculation fast * stochastic and thus non-deterministic – and different libraries handle this differently as you will see in this notebook * `umap-learn` states that https://umap-learn.readthedocs.io/en/latest/reproducibility.html * `cuml` currently uses https://docs.rapids.ai/api/cuml/stable/api.html?highlight=umap#cuml.UMAP. This may chance in https://github.com/rapidsai/cuml/issues/1653#issuecomment-584357155

Further reading:

- Part I: *CPU Baseline in Python Pandas*
- Part II: *GPU Dataframe with RAPIDS Python cudf bindings*
- Part III: GPU SQL - deprecated as Dask-SQL replaced BlazingSQL in the RAPIDS ecosystem
- Part IV: *GPU ML with RAPIDS cuML UMAP and PyGraphistry*
- *Graphistry cuGraph bindings*

clone and install graphistry, print version

[9]:
```
import pandas as pd, networkx as nx
# !git clone https://github.com/graphistry/pygraphistry.git

from time import time
!pip install -U pygraphistry/ --quiet

import graphistry
graphistry.register(api=3,protocol="https", server="hub.graphistry.com", username='***',␣
↪password='***')
graphistry.__version__
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting␣
↪behaviour with the system package manager. It is recommended to use a virtual␣
↪environment instead: https://pip.pypa.io/warnings/venv
```

[9]:
```
'0.27.2+4.ga674343.dirty'
```

[2]:
```
import pandas as pd, numpy as np
start_u = pd.to_datetime('2016-01-01').value//10**9
end_u = pd.to_datetime('2021-01-01').value//10**9
samples=1000
# df = pd.DataFrame(np.random.randint(,100,size=(samples, 1)), columns=['user_id', 'age',
↪ 'profile'])
df = pd.DataFrame(np.random.randint(18,75,size=(samples, 1)), columns=['age'])
df['user_id'] = np.random.randint(0,200,size=(samples, 1))
df['profile'] = np.random.randint(0,1000,size=(samples, 1))
df['date']=pd.to_datetime(np.random.randint(start_u, end_u, samples), unit='s').date

# df[['lat','lon']]=(np.round(np.random.uniform(, 180,size=(samples,2)), 5))
```

(continues on next page)

(continued from previous page)

```
df['lon']=np.round(np.random.uniform(20, 24,size=(samples)), 2)
df['lat']=np.round(np.random.uniform(110, 120,size=(samples)), 2)
df['location']=df['lat'].astype(str) +","+ df["lon"].astype(str)
df.drop(columns=['lat','lon'],inplace=True)
df = df.applymap(str)
df
```

[2]:

	age	user_id	profile	date	location
0	32	185	357	2017-06-16	117.81,22.87
1	66	86	84	2020-03-30	110.07,20.52
2	28	26	862	2019-05-12	116.16,23.02
3	69	193	607	2019-03-11	112.21,23.25
4	34	27	4	2019-08-06	114.56,20.99
..	..	...	...	...	...
995	52	128	435	2016-10-19	115.3,23.67
996	67	116	97	2016-04-24	117.69,23.92
997	32	55	915	2018-11-07	113.63,22.74
998	72	68	148	2020-05-23	116.39,21.25
999	56	19	932	2016-04-23	116.2,23.54

[1000 rows x 5 columns]

[3]:
```
g = graphistry.nodes(df)
t=time()
g2 = g.umap()
min=(time()-t)/60
lin=df.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
g2.plot()
```

```
! Failed umap speedup attempt. Continuing without memoization speedups.* Ignoring target␣
↪column of shape (1000, 0) in UMAP fit, as it is not one dimensional
```

```
['time: 0.14064184427261353 line/min: 7110.259433612426']
```

Parameters: `X` and `y`, `feature_engine`, etc

[4]:
```
g = graphistry.nodes(df)
t=time()
g2 = g.umap(X=['user_id'],y=['date','location'])
min=(time()-t)/60
lin=df.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
g2.plot()
```

```
! Failed umap speedup attempt. Continuing without memoization speedups.* Ignoring target␣
↪column of shape (1000, 14) in UMAP fit, as it is not one dimensional
```

```
['time: 0.0287002166112264 line/min: 34842.94260026035']
```

[5]:
```
g = graphistry.nodes(df)
t=time()
```

(continues on next page)

(continued from previous page)

```
g2 = g.umap(X=['user_id'],y=['date','location'], feature_engine='torch')
min=(time()-t)/60
lin=df.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
g2.plot()
```

```
! Failed umap speedup attempt. Continuing without memoization speedups.* Ignoring target column of shape (1000, 14) in UMAP fit, as it is not one dimensional
```

```
['time: 0.0024895787239074705 line/min: 401674.38386140653']
```

testing various other parameters

[6]:
```
g = graphistry.nodes(df)
t=time()
g2 = g.umap(X=['user_id'],y=['date','location'], feature_engine='torch', n_neighbors= 2, min_dist=.1, spread=.1, local_connectivity=2, n_components=5,metric='hellinger')
min=(time()-t)/60
lin=df.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
g2.plot(render=False)
```

```
! Failed umap speedup attempt. Continuing without memoization speedups.* Ignoring target column of shape (1000, 14) in UMAP fit, as it is not one dimensional
```

```
['time: 0.0022179365158081056 line/min: 450869.5325013168']
```

test `engine` flag to see speed boost

[7]:
```
g = graphistry.nodes(df)
t=time()
g2 = g.umap(engine='cuml')
min=(time()-t)/60
lin=df.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
```

```
! Failed umap speedup attempt. Continuing without memoization speedups.* Ignoring target column of shape (1000, 0) in UMAP fit, as it is not one dimensional
```

```
['time: 0.00446544885635376 line/min: 223941.65338544376']
```

[8]:
```
g = graphistry.nodes(df)
t=time()
g2 = g.umap(engine='umap_learn') ## note this will take appreciable time depending on sample count defined above
min=(time()-t)/60
lin=df.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
```

```
* Ignoring target column of shape (1000, 0) in UMAP fit, as it is not one dimensional
```

```
['time: 0.11818180878957113 line/min: 8461.539134001174']
```

Now lets look at some real data:

[12]:
```
G=pd.read_csv('pygraphistry/demos/data/honeypot.csv')

g = graphistry.nodes(G)
t=time()
g3 = g.umap(engine='cuml')#-learn')
min=(time()-t)/60
lin=G.shape[0]/min
print(['time: '+str(min)+' line/min: '+str(lin)])
```

```
! Failed umap speedup attempt. Continuing without memoization speedups.* Ignoring target␣
↪column of shape (220, 0) in UMAP fit, as it is not one dimensional
```

```
['time: 0.008098324139912924 line/min: 27166.11439590581']
```

[13]:
```
print(g3._edges.info())
g3._edges.sample(5)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2410 entries, 0 to 2821
Data columns (total 3 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   _src_implicit  2410 non-null   int32
 1   _dst_implicit  2410 non-null   int32
 2   _weight        2410 non-null   float32
dtypes: float32(1), int32(2)
memory usage: 47.1 KB
None
```

[13]:

	_src_implicit	_dst_implicit	_weight
671	51	123	0.017956
2123	167	194	0.663975
1761	139	78	0.113361
2444	191	3	0.999991
2441	190	152	0.544303

[16]:
```
#g3.plot()
```

Next steps

- Part I: *CPU Baseline in Python Pandas*
- Part II: *GPU Dataframe with RAPIDS Python cudf bindings*
- Part III: GPU SQL - deprecated as Dask-SQL replaced BlazingSQL in the RAPIDS ecosystem
- Part IV: *GPU ML with RAPIDS cuML UMAP and PyGraphistry*
- *Graphistry cuGraph bindings*

10.8.4.4 Graphistry cuGraph bindings

Graphistry simplifies working with cuGraph. This tutorial demonstrates:

- Converting data between cpu/gpu dataframes and cuGraph
- Enriching dataframes with cuGraph algorithm results
- Visualization

An upcoming release will be adding remote GPU execution when your immediate Python environment does not have a GPU.

Import

[1]:
```
import graphistry, pandas as pd
print(graphistry.__version__)
graphistry.register(api=3, username='...', password='...')
```

```
/opt/conda/envs/rapids/lib/python3.8/site-packages/torch/cuda/__init__.py:83:␣
↪UserWarning: HIP initialization: Unexpected error from hipGetDeviceCount(). Did you␣
↪run some cuda functions before calling NumHipDevices() that might have already set an␣
↪error? Error 101: hipErrorInvalidDevice (Triggered internally at  ../c10/hip/
↪HIPFunctions.cpp:110.)
  return torch._C._cuda_getDeviceCount() > 0
/opt/conda/envs/rapids/lib/python3.8/site-packages/huggingface_hub/snapshot_download.py:
↪6: FutureWarning: snapshot_download.py has been made private and will no longer be␣
↪available from version 0.11. Please use `from huggingface_hub import snapshot_
↪download` to import the only public function in this module. Other members of the file␣
↪may be changed without a deprecation notice.
  warnings.warn(
```

[1]:
```
'refs/pull/360/head'
```

[22]:
```
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/lesmiserables.csv')
g = graphistry.edges(df, df.columns[0], df.columns[1])
g._edges.sample(5)
```

[22]:

	source	target	value
44	Fantine	Dahlia	4
206	Gueulemer	Valjean	1
225	Claquesous	Enjolras	1
49	Mme.Thenardier	Valjean	7
110	Gavroche	Thenardier	1

Enrich graphs with cuGraph algorithm results and visualize the results

[23]:
```
g2 = g.compute_cugraph('pagerank')
g2._nodes.sample(5)
```

```
/opt/conda/envs/rapids/lib/python3.8/site-packages/cudf/core/indexed_frame.py:2271:␣
↪FutureWarning: append is deprecated and will be removed in a future version. Use␣
↪concat instead.
  warnings.warn(
```

[23]:

	id	pagerank
24	Labarre	0.027145
25	Listolier	0.024138
37	Mme.Pontmercy	0.004004
56	Champtercier	0.004004
49	BaronessT	0.004004

[24]:
```
g2.encode_point_color('pagerank', ['blue', 'yellow', 'red'], as_continuous=True).plot()
```

[24]:
```
<IPython.core.display.HTML object>
```

Use cuGraph provided layouts

Run cuGraph's implementation of force_atlas2, inspect results, use custom parameters, and render.

Note that Graphistry's default layout algorithm is already a GPU accelerated FA2 and with additional settings.

[25]:
```
g3 = g2.layout_cugraph('force_atlas2')
g3._nodes.sample(5)
```

```
/opt/conda/envs/rapids/lib/python3.8/site-packages/cudf/core/indexed_frame.py:2271:␣
↪FutureWarning: append is deprecated and will be removed in a future version. Use␣
↪concat instead.
  warnings.warn(
```

[25]:

	id	pagerank	x	y
63	Mme.Burgon	0.010177	100.112083	116.192375
36	Eponine	0.014482	4.713388	30.839472
38	Fantine	0.043900	-38.890110	-113.019806
67	Mme.Thenardier	0.036174	-15.617190	-7.400091
24	Champtercier	0.004004	169.118210	-157.776154

[29]:
```
g3b = g2.layout_cugraph('force_atlas2', params={'lin_log_mode': True})
g3b._nodes.sample(5)
```

[29]:

	id	pagerank	x	y
34	Mme.Burgon	0.010177	341.922455	-116.781532
56	Listolier	0.024138	312.646667	-599.275085
13	Geborand	0.004004	-17.671986	85.812553
35	Mme.Hucheloup	0.004004	325.224762	235.983505
73	Feuilly	0.006593	259.452240	416.950470

[26]:
```
g3.plot()
```

[26]:
```
<IPython.core.display.HTML object>
```

Convert between dataframes and cuGraph

[30]:
```
G = g3.to_cugraph()
```

```
/opt/conda/envs/rapids/lib/python3.8/site-packages/cudf/core/indexed_frame.py:2271:␣
↪FutureWarning: append is deprecated and will be removed in a future version. Use␣
↪concat instead.
  warnings.warn(
```

[31]:
```
G
```

[31]:
```
<cugraph.structure.graph_classes.Graph at 0x7f122b43f2e0>
```

[36]:
```
G.number_of_edges()
```

[36]:
```
254
```

[40]:
```
g4 = graphistry.from_cugraph(G)

{
    'edges': g4._edges.shape,
    'nodes': g4.materialize_nodes()._nodes.shape
}
```

[40]:
```
{'edges': (254, 2), 'nodes': (77, 1)}
```

[ ]:
```

```

[ ]:
```

```

10.8.4.5 How much GPU RAM do you need and how much data fits into a GPU task?

GPU memory size planning & data ratios for Parquet, Arrow, RAPIDS/cuDF, and Graphistry/GFQL

Put too much data into a GPU or use a GPU without enough memory and things fall apart. Whatever GPU you pick, you may then want to partition your data to make sure it fits, but make partitions too small and now you risk only getting a fraction of the available GPU speedups.

Achieving high performance with your GPUs often starts with navigating these questions.

It is surprisingly simple in practice to stay within your GPU memory budget once you understand some common data ratios that occur at basic data pipeline phases.

Using a representative activity logs dataset, we will work through a typical GPU ETL & analytics pipeline that starts all the way from disk:

- Parquet (disk, compressed): 0.1-0.5X
- Arrow (CPU, in-memory): 0.2-1X

- **Pandas (CPU, in-memory): 1X <– baseline**
- cuDF (GPU, in-memory): 0.2-1X
- **GPU compute operations (GPU): 0.2-1X <– includes cuDF tabular queries and GFQL graph queries**
- Overall Peak Usage: 1-2X
- Variants: **Multi-GPU**, **multi-node**, and **AI+ML**

Even before we begin, note that the above ratios already show GPU libraries typically consume a small fraction of the memory required by popular CPU-based libraries like Pandas: They're built with better performance in mind in general, not just because of GPU processing.

10.8.4.6 Phase 1: Setup and Data Creation

(Skip ahead to **The data** if you're just skimming)

Installs & imports

Pandas (CPU), RAPIDS cuDF (GPU), PyGraphistry

[9]:
```
! pip install -q graphistry
```

[ ]:
```
# For freely testing on colab.research.google.com:

# RAPIDS for Google Colab
# This get the RAPIDS-Colab install files and test check your GPU.  Run this and the␣
↪next cell only.
# Please read the output of this cell.  If your Colab Instance is not RAPIDS compatible,␣
↪it will warn you and give you remediation steps.
! git clone -q https://github.com/rapidsai/rapidsai-csp-utils.git
! python rapidsai-csp-utils/colab/pip-install.py > /dev/null 2>&1
```

[11]:
```
import cudf
cudf.__version__
```

[11]:
```
'24.10.01'
```

[12]:
```
# Initialize RMM with a managed memory pool; this will automatically apply to cuDF␣
↪allocations.
import cudf
import rmm
import rmm.statistics
rmm.reinitialize(pool_allocator=True, managed_memory=True)
rmm.statistics.enable_statistics()

# Initialize NVML for direct GPU memory measurement
import pynvml
pynvml.nvmlInit()
handle = pynvml.nvmlDeviceGetHandleByIndex(0)
```

[13]:
```
import pandas as pd
import numpy as np
import pyarrow as pa
import pyarrow.parquet as pq
import cudf
import graphistry
import matplotlib.pyplot as plt
import os
from graphistry import e, n
```

The data

One million simulated network traffic connection events with timestamped events (src_ip, dst_ip) representing graph edges

[14]:
```
rows = 1_000_000
data = {
    "timestamp": pd.date_range(start="2023-01-01", periods=rows, freq="S"),
    "src_ip": np.random.choice([f"192.168.1.{i}" for i in range(1, 256)], rows),
    "dst_ip": np.random.choice([f"10.0.0.{i}" for i in range(1, 256)], rows),
    "event_type": np.random.choice(["connect", "disconnect", "data_transfer"], rows),
    "bytes_transferred": np.random.randint(0, 1000, rows),
}
df = pd.DataFrame(data)
df.head()
```

```
<ipython-input-14-4e913c230e95>:3: FutureWarning: 'S' is deprecated and will be removed␣
↪in a future version, please use 's' instead.
  "timestamp": pd.date_range(start="2023-01-01", periods=rows, freq="S"),
```

[14]:

	timestamp	src_ip	dst_ip	event_type	bytes_transferred
0	2023-01-01 00:00:00	192.168.1.6	10.0.0.216	disconnect	595
1	2023-01-01 00:00:01	192.168.1.247	10.0.0.73	connect	754
2	2023-01-01 00:00:02	192.168.1.244	10.0.0.32	connect	630
3	2023-01-01 00:00:03	192.168.1.204	10.0.0.207	disconnect	348
4	2023-01-01 00:00:04	192.168.1.121	10.0.0.219	connect	710

Phase 2: Measure space usage on-disk with Parquet and CPU in-memory with Pandas

4X CPU in-memory compaction with Arrow

The Apache Arrow in-memory computing table format makes analytics fast by **packing data into typed columns** (vs typical row-wise SQL, KV, graph, and log databases). A typical benefit is data also getting smaller

20X disk compaction with Parquet

Parquet adds **compression algorithms for each column**, giving another 5X multiple over Arrow

Both Parquet and Arrow have their place. Arrow avoids compression for in-memory use to enable faster in-memory access. Parquet prioritizes compression for better disk storage.

[15]:
```
# Pandas Dataframe size
pandas_memory = df.memory_usage(index=True, deep=True).sum() / (1024**2)

# Arrow Table size
arrow_table = pa.Table.from_pandas(df)
arrow_size = arrow_table.nbytes / (1024**2)

# Parquet compressed size
pq_file_path = "compressed_data.parquet"
pq.write_table(arrow_table, pq_file_path, compression="SNAPPY")
parquet_size = os.path.getsize(pq_file_path) / (1024**2)

print(f"Pandas in-memory size: {pandas_memory:.2f} MB")
print(f"Arrow in-memory size: {arrow_size:.2f} MB")
print(f"Parquet compressed size on disk: {parquet_size:.2f} MB")
```

```
Pandas in-memory size: 209.00 MB
Arrow in-memory size: 57.36 MB
Parquet compressed size on disk: 9.57 MB
```

Phase 3: Load data into the GPU with cuDF and PyGraphistry

4X GPU compaction with cuDF

`cuDF` is an open source GPU-based dataframe library that matches the Pandas API. Note that cuDF is Arrow-native, so the estimated GPU memory consumption exactly matches Apache Arrow. It maintains the 4X improvement over Pandas even without doing any compute.

4X GPU compaction with PyGraphistry

Graph users can automate transfering a graph's tables to the GPU via https://pygraphistry.readthedocs.io/en/latest/api/compute.html#graphistry.compute.ComputeMixin.ComputeMixin.to_cudf, reaping the same benefits over a Pandas-based approach.

[16]:
```
# Convert DataFrame to cuDF for operations
gdf = cudf.from_pandas(df)

# Calculate the size of the gdf in memory
gdf_size_bytes = gdf.memory_usage(deep=True).sum()
gdf_size_mb = gdf_size_bytes / (1024**2)  # Convert bytes to MB
print(f"Total gdf size in memory: {gdf_size_mb:.2f} MB")
```

```
Total gdf size in memory: 57.36 MB
```

Pack in 10X+ more data for real workloads with GPU projections and higher CPU RAM

It is convenient to move the entire dataframe to the GPU when there is a lot of room, so we recommend doing that during prototyping.

However, 10X+ bigger workloads can often be easily handled on the same GPU just by mindful of which columns to use at the beginning:

```
# Only transfer 2 columns from df to the GPU
df2 = cudf.from_pandas(df[['src_ip', 'dst_ip']])
```

CPU RAM is often cheaper than GPU RAM, so you may want your CPU to have 1-4X more RAM than your GPUs

Off-GPU IO Speeds

To handle bigger-than-memory datasets, it helps to keep in mind that data travels through different speed devices as it goes through disk to GPU:

It helps to pair your GPU RAM with even more (cheaper) CPU RAM or disk: * Individual SSDs can do 1-5 GB/s, and arrays of them can do 100GB+/s * Consumer speeds for disk->CPU and CPU->GPU are around 32 GB/s per 1-2 GPUs via PCIe 4.0 * Server-grade are often PCIe 5.0 at 64 GB/s per 1-2 GPUs

For advanced setups, such as for going at 100 GB/s on 1-2 GPUs, see our recorded Dask Summit talk on https://www.youtube.com/watch?v=8ZMzsTbfImU. It reviews broad concepts, architecture, and tricks like skipping the convoluted CPU path via https://developer.nvidia.com/gpudirect.

Phase 4: GPU Computation - Simple Task and GFQL Traversal

CPU and GPU programs need extra memory on top of the input data structure memory in order to create intermediate data structures. This is often 1-5X the input data size.

Step A: Simple GPU computation for memory baseline

We see simple cuDF dataframe methods like filtering and joining are optimized, so both take < 1X the original input size. Later, however, we will see peek is multiples higher.

[20]:
```
# Synchronize to ensure a clean memory state before starting
cudf.cuda.current_context().synchronize()

with rmm.statistics.profiler(name="Filter and Sum Operation"):
    filtered = gdf[gdf["event_type"] == "data_transfer"]
    total_bytes = filtered["bytes_transferred"].sum()

subset_gdf = gdf.head(10000)  # Smaller subset to avoid large memory requirements
with rmm.statistics.profiler(name="Join Operation on Subset"):
    joined = subset_gdf.merge(subset_gdf, on="src_ip", how="inner")


filter_sum_stats = rmm.statistics.default_profiler_records.records["Filter and Sum␣
↪Operation"]
filter_sum_peak_mb = filter_sum_stats.memory_peak / (1024**2)
print(f"Filter & Sum Operation Memory Peak: {filter_sum_peak_mb: .2f} MB")

join_stats = rmm.statistics.default_profiler_records.records["Join Operation on Subset"]
join_peak_mb = join_stats.memory_peak / (1024**2)
print(f"Join Operation on Subset Memory Peak: {join_peak_mb: .2f} MB")
```

```
Filter & Sum Operation Memory Peak:  31.16 MB
Join Operation on Subset Memory Peak:  47.92 MB
```

Step B: GPU Graph analytics with GFQL - 2-hop Traversal

The example below is a 2-hop traversal in PyGraphistry's GFQL in cuDF GPU engine mode, including filtering for "data_transfer" events and >500 bytes

Graph queries are more like a sequence of database operators, so unsurprisingly, so we see not only the speed benefits of cuDF but the memory benefits too. The memory is essentially the sum of the optimized operators used.

[21]:
```
# Step 4: Profile the GFQL 2-hop Traversal
g1 = graphistry.edges(gdf, 'src_ip', 'dst_ip')  # Example edge specification for␣
↪Graphistry
with rmm.statistics.profiler(name="GFQL 2-hop Traversal"):
    g2 = g1.gfql([
        n(),
        e(edge_match={'event_type': 'data_transfer'},
          edge_query="bytes_transferred > 500"),
        n()
    ])

gfql_stats = rmm.statistics.default_profiler_records.records["GFQL 2-hop Traversal"]
gfql_peak_mb = gfql_stats.memory_peak / (1024**2)
print(f"GFQL 2-hop Traversal Memory Peak: {gfql_peak_mb: .2f} MB")
```

```
GFQL 2-hop Traversal Memory Peak:  80.58 MB
```

Comparison chart

Let's put it all together to see each phase side by side: dataset sizes, and extra intermediate memory

Fascinatingly, the GPU version was able to do both store the data and and compute on it while taking less memory than the Pandas needed to just make the initial data structure without yet doing anything on top

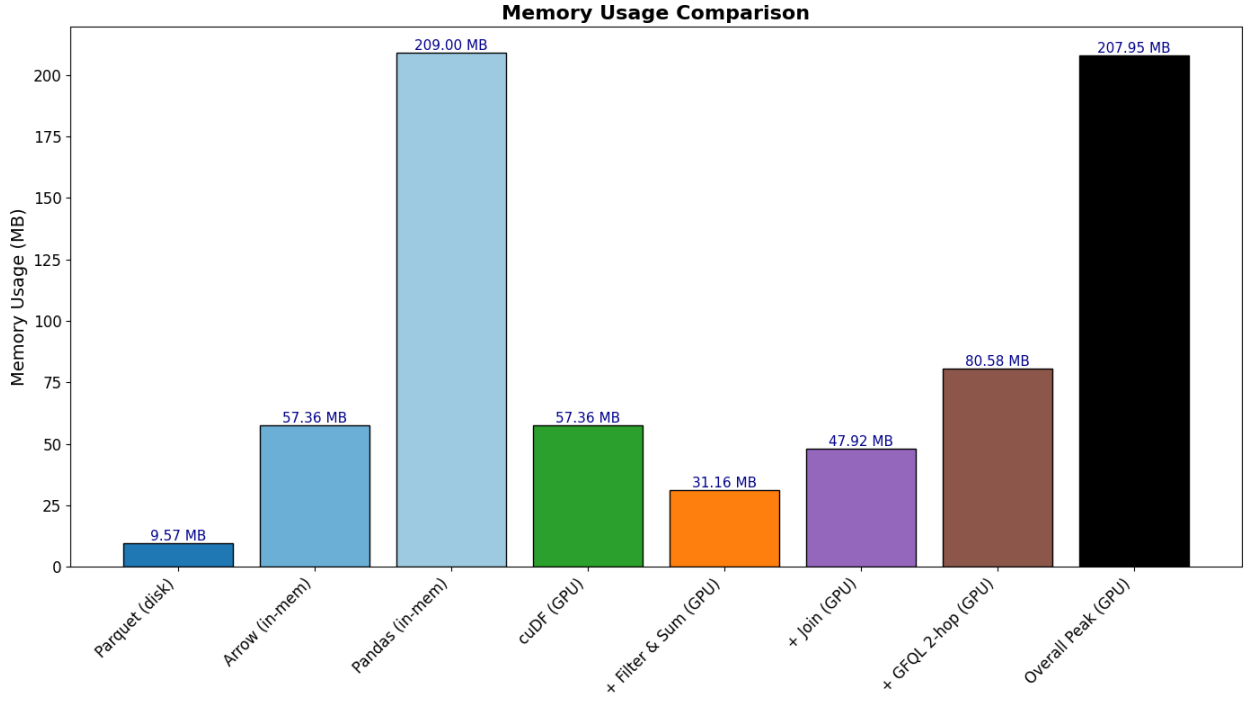
In a large-scale production scenario, we would likely aim for another 10X+ by being targeted on which columns to put on the GPU and when to retire intermediate structures

[23]:
```
import matplotlib.pyplot as plt


# Labels and sizes for the bar chart
labels = [
    'Parquet (disk)',
    'Arrow (in-mem)',
    'Pandas (in-mem)',
    'cuDF (GPU)',
    '+ Filter & Sum (GPU)',
    '+ Join (GPU)',
    '+ GFQL 2-hop (GPU)',
    'Overall Peak (GPU)'
]
sizes = [
    parquet_size,
    arrow_size,
    pandas_memory,
    gdf_size_mb,
    filter_sum_peak_mb,
    join_peak_mb,
    gfql_peak_mb,
    overall_peak_mb
]

colors = ['#1f77b4', '#6baed6', '#9ecae1', '#2ca02c', '#ff7f0e', '#9467bd', '#8c564b', '
↪#000000']

plt.figure(figsize=(14, 8))
bars = plt.bar(labels, sizes, color=colors, edgecolor='black')

# Add labels and title with a modern font size
plt.ylabel('Memory Usage (MB)', fontsize=14)
plt.title('Memory Usage Comparison', fontsize=16, fontweight='bold')
plt.xticks(rotation=45, ha="right", fontsize=12)  # Rotate and size labels for␣
↪readability
plt.yticks(fontsize=12)  # Increase y-axis label font size for consistency
plt.tight_layout()

# Add value labels on top of each bar with a cleaner font style
```

(continues on next page)

(continued from previous page)

```
for bar, size in zip(bars, sizes):
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        bar.get_height(),
        f'{size:.2f} MB',
        ha='center',
        va='bottom',
        fontsize=11,
        fontweight='medium',
        color='darkblue'  # Softer label color for contrast
    )

# Display the plot
plt.show()
```



Takeaways, multi-GPU/multi-node, and ML+AI

The chart reveals several key understandings:

Parquet is a great on-disk format

It is great at compressing big tables, taking a fraction of the in-memory dataset representations

The RAPIDS (cuDF) is a great in-memory format

Its initial GPU memory allocation sizes matches the Apache Arrow CPU in-memory size

Memory consumption is a multiple over the input data siz

Computing takes extra space. 1X-5X additional GPU RAM was needed to compute on the dataframe than just storing it.

Single-GPU

We recommend assuming 10X+ in-memory size needed than the size of compressed Parquet on disk

Multi-GPU, bigger-than-memory, & dask-cudf

When manually chunking big datasets, such as for bigger-than-memory compute or spreading data across multiple GPUs, or automatically via Dask, we generally recommend 1GB+ chunks. This is~10X bigger than CPU Dask tasks because GPUs are more throughput-oriented in general. You can see our Dask Distributed Summit talk on https://www.youtube.com/watch?v=8ZMzsTbfImU for more methodology here

AI/ML Workloads

Modern data science libraries like PyGraphistry's https://pygraphistry.readthedocs.io/en/latest/gfql/combo.html#umap-fit-transform-for-scaling use GPUs and learning to scale:

Training

Often called `fit()`, GPU systems can often make AI/ML training phases handle 10X more data within your time budgets. As you generally do not train on all your data, this means a 10X+ bigger sample set for a higher-fidelity and more representative model.

Inferencing

Often called `transform()`, inference applies a trained model to the rest of your data. This is more scalable than fitting your entire data, so a massive speedup. With GPUs, this goes faster too, essentially matching your GPU budget.

Next steps

We're preparing follow-on articles on more performance intuitions in general and deeper on the technologies discussed here, including how to more carefully measure your own workloads

Meanwhile, you may find these useful as well:

- https://www.youtube.com/watch?v=8ZMzsTbfImU recorded talk at Dask Distributed Summit
- https://pygraphistry.readthedocs.io/en/latest/10min.html GPU-accelerated visual graph analytics
- https://pygraphistry.readthedocs.io/en/latest/gfql/combo.html#umap-fit-transform-for-scaling for visual graph AI
- The open source https://pygraphistry.readthedocs.io/en/latest/gfql/index.html with optional GPU mode
- Try for yourself at https://www.graphistry.com/get-started

10.8.5 AI

10.8.5.1 Your first graph neural network: Detecting suspicious logins with link prediction

http://github.com/graphistry/pygraphistry - Leo Meyerovich, Alex Morrise, Tanmoy Sarkar

https://infosecjupyterthon.com/2022/agenda.html, December 2022

Alert on & visualize anomalous identity events * Demo dataset: 1.6B windows events over 58 days => logins by 12K user over 14K systems * adapt to any identity system with logins * => Can we identify accounts & computers acting anomalously? Resources being oddly accessed? * => Can we spot the red team? * => Operations: Identity incident alerting + identity data investigations * Community/contact for help handling bigger-than-memory & additional features * Techniques explored: Graph AI - * RGCN (primary) - powerful with tweaking and in a pipeline * UMAP (secondary) - surprisingly effective with little tweaking * Runs on both CPU + multi-GPU * Tools: http://github.com/graphistry/pygraphistry, https://www.dgl.ai/ + https://pytorch.org/, and https://rapids.ai/ / https://github.com/lmcinnes/umap

---

1. Graphs are awesome

- https://github.com/JohnLaTwC/Shared/blob/master/Defenders%20think%20in%20lists.%20Attackers%20think%20in%
- Network graphs & event graphs & kill chains & ..: https://hub.graphistry.com/graph/graph.html?dataset=7c2234aa98274
- **Today:** Two techniques for the graph AI era, focusing on **identity graphs**
- **=> Caught 96% of red team's logins (400+ out of millions) with only 10% FPs**
- Graph neural networks (GNNs) + UMAP

2. Graphs for identity data

Sample attacks * Fake account * Account takeover: Malware, credential stuffing, ... * Insider threat: Helpdesk, rogue admin, ... * Abnormal resource access patterns

Data & user activities (UEBA): - Entity resolution: You, your assets, your contexts, .. - Authentication - Authorization - **Money BagMoney BagMoney Bag** Did I mention zero-trust identity protection ? **Money BagMoney BagMoney BagMoney Bag**

Goals: Empower - * Identity detection * Identity investigation

3. AI era of graph: GNNs + UMAP

- GNN's: https://www.science.org/content/article/breakthrough-2021https://www.science.org/content/article/breakthrou 2021 - https://news.mit.edu/sites/default/files/images/202112/MIT-Molecular-Shapes-01-press.jpg
- Combines network thinking (interesting connectivity) with tabular (time, $, etc. features)
- Primitives:
  - Classify nodes ("bot")
  - Predict links ("recommendation", "violation") <– **TODAY**
  - Classify graphs ("motif mining")
- Compose into tools:
  - Anomaly detection <– **today**
  - abuse scoring
  - feeding into combined methods: today we're looking for graph shapes, but temporal cool too (RNN)
  - if model can do well at some task, good chance of reuse on other bits

4. RGCNs - Relational graph convolutional networks

https://hub.graphistry.com/graph/graph.html?dataset=Twitter&play=5000

- GNN - Graph neural network: Label prop
  - "if all their friends are bots, ..."
  - multiple dimensions: bytes, region, ...
- GCNs - Graph convolutional network: Multiple layers
  - "even if know little about them, but their friends.."
  - shallow!
- RGCNs - Relational GCNs:
  - multiple relationship types - follow vs block vs ...
  - ex: remote desktop vs regular login

Watch 2 youtube videos at end for theoretical intuitions

5. Try it yourself

See:

- SSH logs RGCN anomaly detector in a few cells: *simple-ssh-logs-rgcn-anomaly-detector.ipynb*
- In-depth RGCN: *advanced-identity-protection-40m.ipynb*

6. Taking it to production

Watch the repo / contact to join us on:

- Daily batch / real-time alerting => Splunk
- Scaling & autonomous operation
- Tuning: Time data, common FPs (new IPs, ..), ...
- Use for correlation ID generation for investigation context (see tmw's UMAP talk!)

Next steps

- SSH logs RGCN anomaly detector in a few cells: *simple-ssh-logs-rgcn-anomaly-detector.ipynb*
- In-depth RGCN: *advanced-identity-protection-40m.ipynb*
- UMAP demo for 97% alert volume reduction & alert correlation
- http://github.com/graphistry/pygraphistryhttp://github.com/graphistry/pygraphistry (py, oss) + https://hub.graphistry.com/https://hub.graphistry.com/ (free)
  - Dashboarding with https://github.com/graphistry/graph-app-kithttps://github.com/graphistry/graph-app-kit
- Happy to help:
  - https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6ghttps://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g
  - email and let's chat! info@graphistry.com

Resource

- http://github.com/graphistry/pygraphistryhttp://github.com/graphistry/pygraphistry
- https://gradientflow.com/what-is-graph-intelligence/https://gradientflow.com/what-is-graph-intelligence/
- GNN Videos:
  - GCN - https://www.youtube.com/watch?v=2KRAOZIULzw
  - RGCN - https://www.youtube.com/watch?v=wJQQFUcHO5U
  - Euler (combining RNN + GNN)- https://www.youtube.com/watch?v=1t124vguwJ8

[ ]:

10.8.5.2 Identity data anomaly detection: SSH session anomaly detection with RGCNs

- SSH logs from https://www.secrepo.com/: Replace with any event data
- Detects and visualizes anomalous connections based on communication topology & event type
- Unsupervised graph neural network: RGCN
- Runs on both CPU + GPU: Toggle `is_gpu`

For background, so the RGCN intro: *intro-story.ipynb*

Dependencies & data

[1]:
```
#! pip cache remove graphistry
#! pip install --no-cache --user https://github.com/graphistry/pygraphistry/archive/
↪heteroembed.zip

#! pip install --user --no-input "torch==1.11.0" -f https://download.pytorch.org/whl/
↪cu113/torch_stable.html
#! pip install --user dgl-cu113 dglgo -f https://data.dgl.ai/wheels/repo.html
! python -c "import torch; print(torch.cuda.is_available())"
```

[2]:
```
#! wget https://www.secrepo.com/maccdc2012/ssh.log.gz
#! gunzip ssh.log.gz
#! ls -alh ssh*
! head -n 5 ssh.log
```

```
1331901011.840000       CTHcOo3BARDOPDjYue      192.168.202.68  53633   192.168.28.254 ␣
↪22      failure INBOUND SSH-2.0-OpenSSH_5.0     SSH-1.99-Cisco-1.25     -       -       ␣
↪-       -       -
1331901030.210000       CBHpSz2Zi3rdKbAvwd      192.168.202.68  35820   192.168.23.254 ␣
↪22      failure INBOUND SSH-2.0-OpenSSH_5.0     SSH-1.99-Cisco-1.25     -       -       ␣
↪-       -       -
1331901032.030000       C2h6wz2S5MWTiAk6Hb      192.168.202.68  36254   192.168.26.254 ␣
↪22      failure INBOUND SSH-2.0-OpenSSH_5.0     SSH-1.99-Cisco-1.25     -       -       ␣
↪-       -       -
1331901034.340000       CeY76r1JXPbjJS8yKb      192.168.202.68  37764   192.168.27.102 ␣
↪22      failure INBOUND SSH-2.0-OpenSSH_5.0     SSH-2.0-OpenSSH_5.8p1 Debian-1ubuntu3  -
↪       -       -       -       -
1331901041.920000       CPJHML3uGn4IV2MGWi      192.168.202.68  40244   192.168.27.101 ␣
↪22      failure INBOUND SSH-2.0-OpenSSH_5.0     SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1  -
↪       -       -       -       -
```

Imports

[3]:
```
import pandas as pd
import graphistry
graphistry.register(
    #Free gpu server API key: https://www.graphistry.com/get-started
    api=3, username='***', password='***',
    protocol='https', server='hub.graphistry.com', client_protocol_hostname='https://hub.
```

(continues on next page)

(continued from previous page)

```
↪graphistry.com'
)
```

Load data

[4]:
```
df = pd.read_csv(
    './ssh.log', sep='\t',
    names=['time', 'key', 'src_ip', 'src_port', 'dst_ip', 'dst_port', 'msg', 'dir',
           'o1', 'o2', 'o3', 'o4', 'o5', 'o6', 'o7']
)
df.sample(5)
```

[4]:

	time	key	src_ip	src_port	\
6174	1.332014e+09	CA7Epl2hovHB7Zm4a9	192.168.202.141	7200	
1554	1.331919e+09	CwL1tJHLLzytUAaH2	192.168.202.110	49584	
4032	1.332000e+09	C40EOw3sbeRoypxQKi	192.168.202.140	48131	
4691	1.332011e+09	CGSKwo4O56EzNTUqN2	192.168.202.90	48951	
1460	1.331918e+09	CY0O0Q2vWPnFKJAJNe	192.168.204.45	58408	

	dst_ip	dst_port	msg	dir	\
6174	192.168.229.101	22	failure	INBOUND	
1554	192.168.229.101	22	failure	INBOUND	
4032	192.168.25.203	22	undetermined	INBOUND	
4691	192.168.23.254	22	failure	INBOUND	
1460	192.168.25.253	22	failure	INBOUND	

	o1	\
6174	-	
1554	SSH-2.0-OpenSSH_5.0	
4032	SSH-2.0-OpenSSH_5.0	
4691	SSH-2.0-OpenSSH_5.3p1 Debian-3ubuntu6	
1460	SSH-2.0-OpenSSH_5.0	

	o2	o3	o4	o5	o6	7
6174	SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1	-	-	-	-	-
1554	SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1	-	-	-	-	-
4032	SSH-2.0-OpenSSH_5.8p1 Debian-1ubuntu3	-	-	-	-	-
4691	SSH-1.99-Cisco-1.25	-	-	-	-	-
1460	SSH-2.0-OpenSSH_4.5	-	-	-	-	-

Train

- `help(g.embed)` for options
- `relation`: pick an edge column to guide learning to weight differently on
- See other notebooks for adding node features

[5]:
```
is_gpu = True
dev0 = 'cpu'
```

(continues on next page)

(continued from previous page)

```
if is_gpu:
    dev0 = 'cuda'

g = graphistry.edges(df, 'src_ip', 'dst_ip')  # graph
```

[ ]:
```
g2 = g.embed(  # rerun until happy with quality
    device=dev0,

    #relation='dst_port', # always 22, so runs as a GCN instead of RGCN
    relation='o1', # split by sw type

    #==== OPTIONAL: NODE FEATURES ====
    #requires node feature data, ex: g = graphistry.nodes(nodes_df, node_id_col).edges(..
    #use_feat=True
    #X=[g._node] + good_feats_col_names,
    #cardinality_threshold=len(g._edges)+1, #optional: avoid topic modeling on high-
↪cardinality cols
    #min_words=len(g._edges)+1, #optional: avoid topic modeling on high-cardinality cols

    epochs=10
)
```

[ ]:
```
```

Score

- `score`: prediction score from RGCN
- `low_score`: `True` when 2 stdev below the average score

[10]:
```
%%time
def to_cpu(tensor):
    """
    Helper for switching between is_gpu=True/False to avoid coercion errors
    """
    if is_gpu:
        return tensor.cpu()
    else:
        return tensor

score2 = pd.Series(to_cpu(g2._score(g2._triplets)).numpy())

df2 = df.assign(
    score=score2,
    low_score=(score2 < (score2.mean() - 2 * score2.std())) # True for unusually low␣
↪prediction scores
)
df2[['score', 'low_score'] + list(df2.columns[:10])].sort_values(by=['score'])[:5]
```

```
CPU times: user 36.6 ms, sys: 0 ns, total: 36.6 ms
Wall time: 6.87 ms
```

```
/home/graphistry/.local/lib/python3.8/site-packages/graphistry/embed_utils.py:459:␣
↪UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.
↪clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than␣
↪torch.tensor(sourceTensor).
  emb = torch.tensor(self._embeddings)
```

[10]:

```
         score  low_score          time                 key           src_ip  \
4273  0.017218       True  1.332001e+09  CvpN0F4oRP5Pc895fc  192.168.202.136
4369  0.018677       True  1.332006e+09   COqmtb2K1yl9ptmBC  192.168.202.143
2847  0.023266       True  1.331931e+09  CH5EtE1xtwQmyxf5s1   192.168.203.63
2844  0.023587       True  1.331931e+09   CsO3K9zNNojTSGFhk  192.168.202.110
2977  0.023587       True  1.331931e+09   CObILv2xfzVJkUXY6  192.168.202.110

      src_port           dst_ip  dst_port           msg      dir  \
4273     47495  192.168.229.101        22  undetermined  INBOUND
4369     37624  192.168.229.156        22  undetermined  INBOUND
2847     53667   192.168.23.101        22  undetermined  INBOUND
2844     36493  192.168.229.101        22       failure  INBOUND
2977     36511  192.168.229.101        22       failure  INBOUND

                                       o1  \
4273                                    -
4369                                    -
2847                                    -
2844  SSH-2.0-OpenSSH_5.3p1 Debian-3ubuntu6
2977  SSH-2.0-OpenSSH_5.3p1 Debian-3ubuntu6

                                       o2
4273  SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1
4369                   SSH-2.0-OpenSSH_4.3
2847  SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1
2844  SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1
2977  SSH-2.0-OpenSSH_5.8p1 Debian-7ubuntu1
```

Visualize

Color edges red when low prediction score

[9]:

```
(g2
 .edges(df2)
 .encode_edge_color('low_score', categorical_mapping={'true': 'red', 'false': 'blue'})
 .settings(url_params={'strongGravity': 'true', 'play': 0})
).plot()
```

[9]:

```
<IPython.core.display.HTML object>
```

Next steps

- RGCN intro: *intro-story.ipynb*
- In-depth RGCN: *advanced-identity-protection-40m.ipynb*
- UMAP demo for 97% alert volume reduction & alert correlation
- http://github.com/graphistry/pygraphistryhttp://github.com/graphistry/pygraphistry (py, oss) + https://hub.graphistry.com/https://hub.graphistry.com/ (free)
  - Dashboarding with https://github.com/graphistry/graph-app-kithttps://github.com/graphistry/graph-app-kit
- Happy to help:
  - https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g
  - email and let's chat! info@graphistry.com

10.8.5.3 Your first graph neural network: Detecting suspicious logins with link prediction

http://github.com/graphistry/pygraphistry - Leo Meyerovich, Alex Morrise, Tanmoy Sarkar

https://infosecjupyterthon.com/2022/agenda.html, December 2022

Alert on & visualize anomalous identity events * Demo dataset: 1.6B windows events over 58 days => logins by 12K user over 14K systems * adapt to any identity system with logins * => Can we identify accounts & computers acting anomalously? Resources being oddly accessed? * => Can we spot the red team? * => Operations: Identity incident alerting + identity data investigations * Community/contact for help handling bigger-than-memory & additional features * Techniques explored: Graph AI - * RGCN (primary) - powerful with tweaking and in a pipeline * UMAP (secondary) - surprisingly effective with little tweaking * Runs on both CPU + multi-GPU * Tools: http://github.com/graphistry/pygraphistry, https://www.dgl.ai/ + https://pytorch.org/, and https://rapids.ai/ / https://github.com/lmcinnes/umap

---

Dependencies

Installs

- installs below
- ... or docker (free): https://hub.docker.com/r/graphistry/graphistry-nvidia
- .... or aws/azure/enterprise (paid) https://www.graphistry.com/get-started

[4]:
```
#! pip install --user --no-input "torch==1.11.0" -f https://download.pytorch.org/whl/
↪cu113/torch_stable.html
```

[2]:
```
# ! python -c "import torch; print(torch.cuda.is_available())"
```

```
True
```

[3]:
```
#! pip install --user dgl-cu113 dglgo -f https://data.dgl.ai/wheels/repo.html
```

[4]:
```
#! pip cache remove graphistry
#! pip install --no-cache --user https://github.com/graphistry/pygraphistry/archive/
↪heteroembed.zip

#! pip install --user git+https://github.com/graphistry/pygraphistry.
↪git@fe894b556d078a35d1109c5cb78150818950d66d
#! pip install --user git+https://github.com/graphistry/pygraphistry.
↪git@d7e87c0faf762be5c8779c71caf8da13fae48041
#! pip install --user git+https://github.com/graphistry/pygraphistry.
↪git@6f47e034dcf8e96f241be66aca6a6a6374e7bf83
```

Imports

[5]:
```
import pandas as pd
import os
from joblib import load, dump
from collections import Counter
import torch
print('torch', torch.__version__)

import numpy as np
import random
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED)

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

```
torch 1.11.0+cu113
graphistry 0.28.4+78.g6f47e03
```

graphistry

[6]:
```
#graphistry.register(
# api=3,
#  protocol="https", server="hub.graphistry.com", client_protocol_hostname='https://hub.
↪graphistry.com',
#  username = '***', password='***'
#)
```

Data

Winlogs demo

Tested on 40M row sample (2 x RTX), 4M sample (3080 TI), and on a 32 GB CPU box

- https://csr.lanl.gov/data/cyber1/
- `auth`: auth.txt.gz (extractable from winlogs)
- optional: red team data (749 events flagged from auth.txt.gz)

Custom

Use any auth data like winlogs

- load as dataframe `auth`
- key fields: `auth_type` (arbitrary values), `src_computer` (arbitrary values), `dst_computer` (arbitrary values)
- other fields nice but not necessary
- some evaluation steps benefit from a red team but not necessary

Load identity events + optional red team data

[7]:
```
%%time
# total size -> 1051430459/1.05B
auth = pd.read_parquet('data/auth40m.parquet')
print(auth.shape)
auth.head(5)
```

```
(40000000, 9)
CPU times: user 11.7 s, sys: 1.88 s, total: 13.6 s
Wall time: 6.14 s
```

[7]:

	time	src_domain	dst_domain	src_computer	dst_computer	\
0	1	ANONYMOUS LOGON@C586	ANONYMOUS LOGON@C586	C1250	C586	
1	1	ANONYMOUS LOGON@C586	ANONYMOUS LOGON@C586	C586	C586	
2	1	C101$@DOM1	C101$@DOM1	C988	C988	
3	1	C1020$@DOM1	SYSTEM@C1020	C1020	C1020	
4	1	C1021$@DOM1	C1021$@DOM1	C1021	C625	

	auth_type	logontype	authentication_orientation	success_or_failure
0	NTLM	Network	LogOn	Success
1	?	Network	LogOff	Success
2	?	Network	LogOff	Success
3	Negotiate	Service	LogOn	Success
4	Kerberos	Network	LogOn	Success

Red team comparison data (optional)

[8]:
```
if True:
    red_team = pd.read_csv('data/redteam.txt', header=None)
else:
    # or whatever comparison set
    # unsupervised method so just for reporting
    red_team = auth.head(749)

print(red_team.shape)
red_team.head(5)
```

```
(749, 4)
```

[8]:

	0	1	2	3
0	150885	U620@DOM1	C17693	C1003
1	151036	U748@DOM1	C17693	C305
2	151648	U748@DOM1	C17693	C728
3	151993	U6115@DOM1	C17693	C1173
4	153792	U636@DOM1	C17693	C294

[9]:
```
cols = auth.columns
anom_label = auth[
    auth[cols[0]].isin(red_team[0]) &
    auth[cols[1]].isin(red_team[1]) &
    auth[cols[2]].isin(red_team[1]) &
    auth[cols[3]].isin(red_team[2]) &
    auth[cols[4]].isin(red_team[3])
].reset_index()
anom_label
```

[9]:

	index	time	src_domain	dst_domain	src_computer	dst_computer	\
0	29627157	150885	U620@DOM1	U620@DOM1	C17693	C1003	
1	29657341	151036	U748@DOM1	U748@DOM1	C17693	C305	
2	29776464	151648	U748@DOM1	U748@DOM1	C17693	C728	
3	29850298	151993	U6115@DOM1	U6115@DOM1	C17693	C1173	
4	30188660	153792	U636@DOM1	U636@DOM1	C17693	C294	
5	30425163	155219	U748@DOM1	U748@DOM1	C17693	C5693	
6	30458238	155399	U748@DOM1	U748@DOM1	C17693	C152	
7	30468055	155460	U748@DOM1	U748@DOM1	C17693	C2341	
8	30489739	155591	U748@DOM1	U748@DOM1	C17693	C332	
9	30662347	156658	U748@DOM1	U748@DOM1	C17693	C4280	
10	39964825	210086	U748@DOM1	U748@DOM1	C18025	C1493	

	auth_type	logontype	authentication_orientation	success_or_failure
0	NTLM	Network	LogOn	Success
1	NTLM	Network	LogOn	Success
2	NTLM	Network	LogOn	Success
3	NTLM	Network	LogOn	Success
4	NTLM	Network	LogOn	Success
5	NTLM	Network	LogOn	Success
6	NTLM	Network	LogOn	Success
7	NTLM	Network	LogOn	Success
8	NTLM	Network	LogOn	Success

(continues on next page)

(continued from previous page)

```
9       NTLM   Network                       LogOn           Success
10      NTLM   Network                       LogOn           Success
```

[10]:
```
anom_label['RED'] = 1

# enrich the edges with the red team data
ind = auth.index.isin(anom_label['index'])
auth.loc[ind, 'RED'] = 1
auth.loc[~ind, 'RED'] = 0

print('# red', auth['RED'].sum())

# let's see the bad infiltration point
auth[auth.src_computer == 'C17693'].sample(10)
```

```
# red 11.0
```

[10]:
```
            time   src_domain   dst_domain src_computer dst_computer  \
39201345  206645   U8350@DOM1   U8350@DOM1       C17693         C467
29776464  151648    U748@DOM1    U748@DOM1       C17693         C728
38906146  205356    U748@DOM1    U748@DOM1       C17693        C3007
30188660  153792    U636@DOM1    U636@DOM1       C17693         C294
28845530  147076   U8009@C926   U8009@C926       C17693         C926
28860620  147141  U8009@C1759  U8009@C1759       C17693        C1759
39208487  206678   U8350@DOM1   U8350@DOM1       C17693         C529
39126261  206295   U8350@DOM1   U8350@DOM1       C17693         C529
29620541  150847    U620@DOM1    U620@DOM1       C17693        C1759
28359627  145015  U1723@C1759  U1723@C1759       C17693        C1759

         auth_type logontype authentication_orientation success_or_failure  \
39201345      NTLM   Network                       LogOn            Success
29776464      NTLM   Network                       LogOn            Success
38906146      NTLM   Network                       LogOn            Success
30188660      NTLM   Network                       LogOn            Success
28845530      NTLM   Network                       LogOn               Fail
28860620      NTLM   Network                       LogOn               Fail
39208487      NTLM   Network                       LogOn            Success
39126261      NTLM   Network                       LogOn            Success
29620541      NTLM   Network                       LogOn            Success
28359627      NTLM   Network                       LogOn               Fail

          RED
39201345  0.0
29776464  1.0
38906146  0.0
30188660  1.0
28845530  0.0
28860620  0.0
39208487  0.0
39126261  0.0
29620541  0.0
28359627  0.0
```

Quick viz

https://hub.graphistry.com/graph/graph.html?dataset=7f5cd746e6944a739a8627bd3a8892fb&play=0

[11]:
```
graphistry.edges(
    pd.concat([anom_label, auth.sample(100000)], ignore_index=True, sort=False),
    'src_computer', 'dst_computer'
).plot()
```

[11]:
```
<IPython.core.display.HTML object>
```

Down-sample (optional)

[12]:
```
if True:
    SAMPLE_PERCENT = 0.10
    print(f'sampling down {SAMPLE_PERCENT * 100}%')
    all_auth = auth
    all_auth_unred = all_auth[ all_auth.RED == 0 ]
    all_auth_red = all_auth[ all_auth.RED == 1 ]
    auth = pd.concat(
        [
            all_auth_unred.sample(frac=SAMPLE_PERCENT, random_state=RANDOM_SEED),
            all_auth_red
        ],
        ignore_index=True,
        sort=False)
    print(auth.shape)
```

```
sampling down 10.0%
(4000010, 10)
```

[13]:
```
# see the data and if everything was flagged (since this is timelike..)
auth[
    auth['src_computer'].isin(set(red_team[2])) &
    auth['dst_computer'].isin(set(red_team[3]))
].reset_index()
```

[13]:

	index	time	src_domain	dst_domain	src_computer	dst_computer	\
0	1467943	206678	U8350@DOM1	U8350@DOM1	C17693	C529	
1	3999999	150885	U620@DOM1	U620@DOM1	C17693	C1003	
2	4000000	151036	U748@DOM1	U748@DOM1	C17693	C305	
3	4000001	151648	U748@DOM1	U748@DOM1	C17693	C728	
4	4000002	151993	U6115@DOM1	U6115@DOM1	C17693	C1173	
5	4000003	153792	U636@DOM1	U636@DOM1	C17693	C294	
6	4000004	155219	U748@DOM1	U748@DOM1	C17693	C5693	
7	4000005	155399	U748@DOM1	U748@DOM1	C17693	C152	
8	4000006	155460	U748@DOM1	U748@DOM1	C17693	C2341	
9	4000007	155591	U748@DOM1	U748@DOM1	C17693	C332	
10	4000008	156658	U748@DOM1	U748@DOM1	C17693	C4280	
11	4000009	210086	U748@DOM1	U748@DOM1	C18025	C1493	

	auth_type	logontype	authentication_orientation	success_or_failure	RED

(continues on next page)

(continued from previous page)

```
0       NTLM   Network                LogOn           Success  0.0
1       NTLM   Network                LogOn           Success  1.0
2       NTLM   Network                LogOn           Success  1.0
3       NTLM   Network                LogOn           Success  1.0
4       NTLM   Network                LogOn           Success  1.0
5       NTLM   Network                LogOn           Success  1.0
6       NTLM   Network                LogOn           Success  1.0
7       NTLM   Network                LogOn           Success  1.0
8       NTLM   Network                LogOn           Success  1.0
9       NTLM   Network                LogOn           Success  1.0
10      NTLM   Network                LogOn           Success  1.0
11      NTLM   Network                LogOn           Success  1.0
```

Configure

Identify src vs dst columns

The RGCN connects homogeneous nodes (ex: computers) through heterogeneous relationships (ex: event types)

[14]:
```
SRC = 'src_computer'
DST = 'dst_computer'
#SRC = 'src_domain'
#DST = 'dst_domain'

NODE = 'computer'


##DST_OPP = 'dst_domain'
#SRC_OPP = 'src_computer'
#DST_OPP = 'dst_computer'
```

Feature engineering

Node features - optional enrichment & event summaries

- Provided: If node features are available, handle here
- Aggregated: We gather some here from the incoming vs outgoing edges
- Graph analytics: Often useful to also add graph statistics here too (pagerank, centrality, ...)

For bigger-than-memory, use dask (cpu) or dask_cudf (gpu)

[15]:
```
mode = pd.Series.mode
outgoing_conn_stats = auth[:1000].groupby([SRC]).agg({
    'time': ['min', 'max', 'mean'],
    'dst_domain': [mode, 'count'],
    'dst_computer': [mode, 'count'],
    'auth_type': [mode, 'count'],
```

(continues on next page)

(continued from previous page)

```
    'logontype': [mode, 'count'],
    'authentication_orientation': [mode, 'count'],
    'success_or_failure': [mode, 'count']
})
outgoing_conn_stats.columns = [f'out_{c[0]}_{c[1]}' for c in outgoing_conn_stats.
↪columns]

mode = pd.Series.mode
incoming_conn_stats = auth[:1000].groupby([DST]).agg({
    'time': ['min', 'max', 'mean'],
    'src_domain': [mode, 'count'],
    'src_computer': [mode, 'count'],
    'auth_type': [mode, 'count'],
    'logontype': [mode, 'count'],
    'authentication_orientation': [mode, 'count'],
    'success_or_failure': [mode, 'count']
})
incoming_conn_stats.columns = [f'in_{c[0]}_{c[1]}' for c in incoming_conn_stats.columns]


# outgoing_conn_stats
# incoming_conn_stats

nodes = pd.DataFrame({NODE: pd.concat([auth[SRC], auth[DST]]).unique()})
nodes = nodes.merge(outgoing_conn_stats, how='left', left_on=NODE, right_on=SRC)
nodes = nodes.merge(incoming_conn_stats, how='left', left_on=NODE, right_on=DST)
for c in ['min', 'max', 'mean']:
    nodes[f'out_time_{c}'] = nodes[f'out_time_{c}'].fillna(0.)
    nodes[f'in_time_{c}'] = nodes[f'in_time_{c}'].fillna(0.)


def flatten_lst(o):
    if isinstance(o, list):
        return o[0]
    if isinstance(o, str):
        return o
    return ''


for c in ['mode']:
    for k in ['dst_domain', 'dst_computer', 'auth_type', 'logontype', 'authentication_
↪orientation', 'success_or_failure']:
        nodes[f'out_{k}_{c}'] = nodes[f'out_{k}_{c}'].apply(flatten_lst)
    for k in ['src_domain', 'src_computer', 'auth_type', 'logontype', 'authentication_
↪orientation', 'success_or_failure']:
        nodes[f'in_{k}_{c}'] = nodes[f'in_{k}_{c}'].apply(flatten_lst)

for c in ['count']:
    for k in ['dst_domain', 'dst_computer', 'auth_type', 'logontype', 'authentication_
↪orientation', 'success_or_failure']:
        nodes[f'out_{k}_{c}'] = nodes[f'out_{k}_{c}'].fillna(0)
    for k in ['src_domain', 'src_computer', 'auth_type', 'logontype', 'authentication_
```

(continues on next page)

(continued from previous page)

```
↪orientation', 'success_or_failure']:
        nodes[f'in_{k}_{c}'] = nodes[f'in_{k}_{c}'].fillna(0)

print(nodes.shape)
nodes.sample(10)
```

```
(10887, 31)
```

[15]:

```
      computer  out_time_min  out_time_max  out_time_mean out_dst_domain_mode  \
1157     C7249           0.0           0.0            0.0
10714   C10703           0.0           0.0            0.0
3872     C6943           0.0           0.0            0.0
8832     C7472           0.0           0.0            0.0
10732    C9999           0.0           0.0            0.0
2885     C1555           0.0           0.0            0.0
1701     C5796           0.0           0.0            0.0
7597    C11731           0.0           0.0            0.0
5051     C7485           0.0           0.0            0.0
8477     C1548           0.0           0.0            0.0

       out_dst_domain_count out_dst_computer_mode  out_dst_computer_count  \
1157                    0.0                                           0.0
10714                   0.0                                           0.0
3872                    0.0                                           0.0
8832                    0.0                                           0.0
10732                   0.0                                           0.0
2885                    0.0                                           0.0
1701                    0.0                                           0.0
7597                    0.0                                           0.0
5051                    0.0                                           0.0
8477                    0.0                                           0.0

      out_auth_type_mode  out_auth_type_count  ... in_src_computer_mode  \
1157                                      0.0  ...
10714                                     0.0  ...
3872                                      0.0  ...
8832                                      0.0  ...
10732                                     0.0  ...
2885                                      0.0  ...
1701                                      0.0  ...
7597                                      0.0  ...
5051                                      0.0  ...
8477                                      0.0  ...

       in_src_computer_count in_auth_type_mode  in_auth_type_count  \
1157                     0.0                                   0.0
10714                    0.0                                   0.0
3872                     0.0                                   0.0
8832                     0.0                                   0.0
10732                    0.0                                   0.0
2885                     0.0                                   0.0
1701                     0.0                                   0.0
7597                     0.0                                   0.0
```

(continues on next page)

(continued from previous page)

```
5051                    0.0                                  0.0
8477                    0.0                                  0.0

      in_logontype_mode  in_logontype_count  \
1157                                    0.0
10714                                   0.0
3872                                    0.0
8832                                    0.0
10732                                   0.0
2885                                    0.0
1701                                    0.0
7597                                    0.0
5051                                    0.0
8477                                    0.0

       in_authentication_orientation_mode  \
1157
10714
3872
8832
10732
2885
1701
7597
5051
8477

       in_authentication_orientation_count  in_success_or_failure_mode  \
1157                                    0.0
10714                                   0.0
3872                                    0.0
8832                                    0.0
10732                                   0.0
2885                                    0.0
1701                                    0.0
7597                                    0.0
5051                                    0.0
8477                                    0.0

      in_success_or_failure_count
1157                          0.0
10714                         0.0
3872                          0.0
8832                          0.0
10732                         0.0
2885                          0.0
1701                          0.0
7597                          0.0
5051                          0.0
8477                          0.0

[10 rows x 31 columns]
```

Node features - auto-generation & compression

[16]:
```
gg = graphistry.nodes(nodes, NODE)
g_sample = gg.featurize(
    cardinality_threshold=len(auth)+1,  # avoid topic modeling on high-cardinality cols
    min_words=len(auth)+1,  # avoid topic modeling on high-cardinality cols
)

X = g_sample._node_features
#print('memory', X.memory_usage())
print('sample low cardinality')
{
    c: X[c].nunique()
    for c in X.columns[:10] #sampled, good to manually inspect for nice columns
    if X[c].nunique() < 10
}
```

```
sample low cardinality
```

[16]:
```
{'out_dst_domain_mode_': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C1065': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C1628': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C17899': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C457': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C467': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C528': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C529': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C586': 2,
 'out_dst_domain_mode_ANONYMOUS LOGON@C625': 2}
```

[17]:
```
sums = X.sum(0)
good_feats = [c for c in X.columns if sums[c] > 10 or 'type' in c]

print('#total feats', len(X.columns))
print('#good feats (decent cardinality)', len(good_feats))
print(X[good_feats].sum(0).astype(int))
X[good_feats].sample(5)
```

```
#total feats 889
#good feats (decent cardinality) 74
out_dst_domain_mode_                        10411
out_dst_domain_mode_ANONYMOUS LOGON@C586       16
out_dst_computer_mode_                      10408
out_dst_computer_mode_C1065                    25
out_dst_computer_mode_C2106                    16
                                             ...
in_src_computer_count                        1000
in_auth_type_count                           1000
in_logontype_count                           1000
in_authentication_orientation_count          1000
in_success_or_failure_count                  1000
Length: 74, dtype: int64
```

[17]:
```
      out_dst_domain_mode_  out_dst_domain_mode_ANONYMOUS LOGON@C586  \
```

(continues on next page)

(continued from previous page)

```
3579                  1.0                                      0.0
6966                  1.0                                      0.0
4599                  1.0                                      0.0
409                   0.0                                      0.0
5072                  1.0                                      0.0

      out_dst_computer_mode_  out_dst_computer_mode_C1065  \
3579                     1.0                          0.0
6966                     1.0                          0.0
4599                     1.0                          0.0
409                      0.0                          0.0
5072                     1.0                          0.0

      out_dst_computer_mode_C2106  out_dst_computer_mode_C457  \
3579                          0.0                         0.0
6966                          0.0                         0.0
4599                          0.0                         0.0
409                           0.0                         0.0
5072                          0.0                         0.0

      out_dst_computer_mode_C467  out_dst_computer_mode_C528  \
3579                         0.0                         0.0
6966                         0.0                         0.0
4599                         0.0                         0.0
409                          0.0                         0.0
5072                         0.0                         0.0

      out_dst_computer_mode_C529  out_dst_computer_mode_C586  ...  \
3579                         0.0                         0.0  ...
6966                         0.0                         0.0  ...
4599                         0.0                         0.0  ...
409                          0.0                         0.0  ...
5072                         0.0                         0.0  ...

      out_success_or_failure_count  in_time_min  in_time_max  in_time_mean  \
3579                           0.0          0.0          0.0           0.0
6966                           0.0          0.0          0.0           0.0
4599                           0.0          0.0          0.0           0.0
409                            1.0     199787.0     199787.0      199787.0
5072                           0.0          0.0          0.0           0.0

      in_src_domain_count  in_src_computer_count  in_auth_type_count  \
3579                  0.0                    0.0                 0.0
6966                  0.0                    0.0                 0.0
4599                  0.0                    0.0                 0.0
409                   1.0                    1.0                 1.0
5072                  0.0                    0.0                 0.0

      in_logontype_count  in_authentication_orientation_count  \
3579                 0.0                                 0.0
6966                 0.0                                 0.0
4599                 0.0                                 0.0
```

(continues on next page)

(continued from previous page)

```
409                  1.0                                  1.0
5072                 0.0                                  0.0

      in_success_or_failure_count
3579                          0.0
6966                          0.0
4599                          0.0
409                           1.0
5072                          0.0

[5 rows x 74 columns]
```

Edge features - Multi-column relationship types (optional)

Untyped

Useful for emulating a vanilla GCN via RGCN

[18]:
```
df = auth
df2a = df.assign(constant_edge_label=1)
```

Combos: reln as combined (`auth_type` x `logontype`)

[19]:
```
df2 = df2a.assign(reln=df.auth_type + '::' + df.logontype)
df2.reln.sample(5)
```

[19]:
```
243922     Kerberos::Network
835852     Kerberos::Network
865963                  ?::?
3562087           ?::Network
1788835    Kerberos::Network
Name: reln, dtype: object
```

Compression: reln as topic model over many columns or high-cardinality columns

To prevent too many relationship types blowing out memory, we can use tricks like topic models to cut down to a more reasonable set

[20]:
```
g_runiq = graphistry.nodes(df2.drop_duplicates(subset=['reln']))
g_featurized = g_runiq.featurize(X=['reln'], min_words=1000000, n_topics=15, cardinality_
↪threshold=2)
X = g_featurized._node_features
X.sample(5)
```

[20]:
```
        reln: ntlm, wave, batch  reln: batch, ntlm, wave  \
339375                 0.148557                 0.056838
7814                   0.052674                 0.055471
683631                 0.050558                 0.053184
```

(continues on next page)

(continued from previous page)

```
237141                0.206005                0.056884
4217                  0.054613                0.074897

       reln: negotiate, ntlm, wave  reln: ntlm, wave, interactive  \
339375                     0.057913                       0.152160
7814                      11.718799                       1.717323
683631                     0.058496                       0.053717
237141                     0.057404                       0.057193
4217                       0.057616                       0.056144

       reln: microsoft_authentication_packa, microsoft_authentication_pack, microsoft_
↪authentication_pac  \
339375                                          0.185454
7814                                            0.053624
683631                                          0.050803
237141                                          8.189937
4217                                            0.056909

       reln: remoteinteractive, interactive, ntlm  \
339375                                    0.174556
7814                                     25.518031
683631                                    0.051884
237141                                    0.054009
4217                                      0.056405

       reln: microsoft_authentication_package_v1, microsoft_authentication_package_v,␣
↪microsoft_authentication_package_  \
339375                                          0.280331
7814                                            0.052828
683631                                          0.050711
237141                                         51.363086
4217                                            0.055506

       reln: kerberos, network, ntlm  reln: newcredentials, ntlm, wave  \
339375                       0.054524                         0.060744
7814                         0.054558                         0.062141
683631                       0.051522                         0.050903
237141                       0.262343                         0.072549
4217                         0.066759                        22.429283

       reln: microsoft_authentication_package_v1_0, microsoft_authentication_package_v1_
↪, microsoft_authentication_package_v1  \
339375                                          70.747278
7814                                            0.060254
683631                                          0.050871
237141                                          0.211270
4217                                            0.053986

       reln: networkcleartext, ntlm, wave  reln: service, ntlm, wave  \
339375                           0.053815                  0.058804
7814                             0.071018                  0.085292
683631                           0.051656                  0.052714
```

(continues on next page)

(continued from previous page)

```
237141                            0.085901                    0.055594
4217                              0.059091                    0.056387

        reln: unlock, wave, ntlm  reln: cachedinteractive, interactive, ntlm  \
339375                  0.056780                                     0.522876
7814                    0.057539                                     0.137766
683631                 15.020143                                     0.052309
237141                  0.054310                                     0.057548
4217                    0.054664                                     0.058439

        reln: microsoft_authentica, microsoft_authentication_pa, microsoft_
↪authentication_pac
339375                                              0.139371
7814                                                0.052682
683631                                              0.050528
237141                                              1.465967
4217                                                0.059300
```

[21]:
```
X_df2, y_df2 = g_featurized.transform(df2, ydf=None, kind='nodes')
reln_topic = pd.Series([ X_df2.columns[k] for k in X_df2.values.argmax(1) ])
df3 = df2.assign(reln_topic=reln_topic)
reln_topic.sample(10)
```

[21]:
```
392537     reln: kerberos, network, ntlm
3497381    reln: kerberos, network, ntlm
387017     reln: kerberos, network, ntlm
2356261    reln: kerberos, network, ntlm
1169846    reln: kerberos, network, ntlm
3038088    reln: kerberos, network, ntlm
2420173    reln: kerberos, network, ntlm
2441144          reln: batch, ntlm, wave
3933167    reln: kerberos, network, ntlm
2362703    reln: kerberos, network, ntlm
dtype: object
```

Comparison

Any of these may be interesting as feature columns

[22]:
```
print({
    v: df3[v].nunique()
    for v in ['constant_edge_label', 'auth_type', 'logontype', 'reln', 'reln_topic']
})

df3[['constant_edge_label', 'auth_type', 'logontype', 'reln', 'reln_topic']].sample(10)
```

```
{'constant_edge_label': 1, 'auth_type': 17, 'logontype': 10, 'reln': 38, 'reln_topic':␣
↪15}
```

[22]:
```
         constant_edge_label auth_type logontype              reln  \
3181846                    1         ?         ?             ?::?
3637183                    1      NTLM   Network     NTLM::Network
```

(continues on next page)

(continued from previous page)

```
3527822                    1  Kerberos   Network  Kerberos::Network
2972171                    1  Kerberos   Network  Kerberos::Network
1844173                    1         ?   Network         ?::Network
3405729                    1         ?   Network         ?::Network
703185                     1  Kerberos   Network  Kerberos::Network
3827554                    1         ?   Network         ?::Network
2057222                    1         ?   Network         ?::Network
2072045                    1  Kerberos   Network  Kerberos::Network

                          reln_topic
3181846       reln: batch, ntlm, wave
3637183  reln: kerberos, network, ntlm
3527822  reln: kerberos, network, ntlm
2972171  reln: kerberos, network, ntlm
1844173  reln: kerberos, network, ntlm
3405729  reln: kerberos, network, ntlm
703185   reln: kerberos, network, ntlm
3827554  reln: kerberos, network, ntlm
2057222  reln: kerberos, network, ntlm
2072045  reln: kerberos, network, ntlm
```

Model training

Graph: `(computer)-[event]->(computer)`

Edge relationship variants: untyped, `auth_type`, `auth_type::logontype`, topic model `auth_type::logontype` Node feature variants: none, sampled

Graph shape

[23]:
```
# simple: graphistry.edges(df3, SRC, DST)

g = (
    graphistry
    .nodes(pd.concat([nodes, X], axis=1), NODE) ## adds computed node features
    .edges(df3, SRC, DST)
)
```

Node & edge data

[24]:
```
print(g._nodes.shape)
g._nodes.sample(5)
```

```
(10904, 46)
```

[24]:
```
      computer  out_time_min  out_time_max  out_time_mean out_dst_domain_mode  \
554      C1476           0.0           0.0            0.0
8324    C15964           0.0           0.0            0.0
10540    C5684           0.0           0.0            0.0
```

(continues on next page)

(continued from previous page)

```
1933    C12857           0.0           0.0           0.0
2428    C14040           0.0           0.0           0.0

       out_dst_domain_count out_dst_computer_mode  out_dst_computer_count  \
554                     0.0                                           0.0
8324                    0.0                                           0.0
10540                   0.0                                           0.0
1933                    0.0                                           0.0
2428                    0.0                                           0.0

      out_auth_type_mode  out_auth_type_count  ...  \
554                                      0.0  ...
8324                                     0.0  ...
10540                                    0.0  ...
1933                                     0.0  ...
2428                                     0.0  ...

      reln: remoteinteractive, interactive, ntlm  \
554                                          NaN
8324                                         NaN
10540                                        NaN
1933                                         NaN
2428                                         NaN

       reln: microsoft_authentication_package_v1, microsoft_authentication_package_v,␣
↪microsoft_authentication_package_  \
554                                                  NaN
8324                                                 NaN
10540                                                NaN
1933                                                 NaN
2428                                                 NaN

      reln: kerberos, network, ntlm  reln: newcredentials, ntlm, wave  \
554                             NaN                                NaN
8324                            NaN                                NaN
10540                           NaN                                NaN
1933                            NaN                                NaN
2428                            NaN                                NaN

      reln: microsoft_authentication_package_v1_0, microsoft_authentication_package_v1_,␣
↪microsoft_authentication_package_v1  \
554                                                  NaN
8324                                                 NaN
10540                                                NaN
1933                                                 NaN
2428                                                 NaN

       reln: networkcleartext, ntlm, wave  reln: service, ntlm, wave  \
554                                   NaN                        NaN
8324                                  NaN                        NaN
10540                                 NaN                        NaN
1933                                  NaN                        NaN
```

(continues on next page)

(continued from previous page)

```
2428                                   NaN                                   NaN

      reln: unlock, wave, ntlm  reln: cachedinteractive, interactive, ntlm  \
554                        NaN                                          NaN
8324                       NaN                                          NaN
10540                      NaN                                          NaN
1933                       NaN                                          NaN
2428                       NaN                                          NaN

     reln: microsoft_authentica, microsoft_authentication_pa, microsoft_authentication_
↪pac
554                                                    NaN
8324                                                   NaN
10540                                                  NaN
1933                                                   NaN
2428                                                   NaN

[5 rows x 46 columns]
```

[25]:
```
print(g._edges.shape)
g._edges.sample(5)
```

```
(4000010, 13)
```

[25]:
```
           time             src_domain             dst_domain src_computer  \
2919189  165394            C1620$@DOM1            C1620$@DOM1        C1621
3926399   45204   ANONYMOUS LOGON@C586   ANONYMOUS LOGON@C586        C2654
390234    14983                U6@DOM1                U6@DOM1          C61
2017626   47099            C1085$@DOM1            C1085$@DOM1         C467
3126304  138945  ANONYMOUS LOGON@C2106  ANONYMOUS LOGON@C2106        C8416

        dst_computer auth_type logontype authentication_orientation  \
2919189         C528         ?         ?                        TGS
3926399         C586      NTLM   Network                      LogOn
390234           C61         ?   Network                     LogOff
2017626         C467         ?   Network                     LogOff
3126304        C2106      NTLM   Network                      LogOn

        success_or_failure  RED  constant_edge_label           reln  \
2919189            Success  0.0                    1          ?::?
3926399            Success  0.0                    1  NTLM::Network
390234             Success  0.0                    1     ?::Network
2017626            Success  0.0                    1     ?::Network
3126304            Success  0.0                    1  NTLM::Network

                        reln_topic
2919189       reln: batch, ntlm, wave
3926399  reln: kerberos, network, ntlm
390234   reln: kerberos, network, ntlm
2017626  reln: kerberos, network, ntlm
3126304  reln: kerberos, network, ntlm
```

Train

[26]:
```
# If GPU/CPU RAM low, try to free up
if True:
    g.reset_caches()
    torch.cuda.empty_cache()
```

Resumable – want to run for more epochs? Just run the cell again **(rocket)**

[ ]:
```
is_gpu = True

# CPU + GPU both work!
# GPUs tested:
#   20-40M rows: 2 x GPUs (24 GB GPU RAM ea)
#   4M rows: 1 x 3080 Ti GPU  (12 GB GPU RAM)
# CPU: 32 GB RAM

dev0 = 'cpu'
if is_gpu:
    dev0 = 'cuda'

g2 = g.embed(

    #relation='constant_edge_label',
    #relation='auth_type',
    #relation='reln',
    #relation='reln_topic'

    #proto='RotatE',
    #proto='DistMult', ##default
    #proto='TransE',

    #if nodes: g.nodes(...), then g.embed(use_feats=True, X=['col1', ..], and optionally,
↪ y=['col10', ..]
    use_feat=True, #nodes with attributes
    X=[NODE] + good_feats,
    #cardinality_threshold=len(g._edges)+1, #avoid topic modeling on high-cardinality␣
↪cols
    #min_words=len(g._edges)+1,  #avoid topic modeling on high-cardinality cols

    ##n_topics=21,

    train_split=0.99,
    batch_size=32,
    embedding_dim=32,
    evaluate=True,
    lr=0.004,  # 0.002, ..
    device=dev0,

    epochs=20,  # 20 if no features & 4M samples, 60+ if node features:
                # increase/rerun till score 85%+
)
```

Inspect model

[27]:
```
g2._embed_model
```

[27]:
```
HeteroEmbed(
  (rgcn): RGCNEmbed(
    (emb): Embedding(10888, 32)
    (rgc1): RelGraphConv(
      (linear_r): TypedLinear(in_size=32, out_size=32, num_types=1, regularizer=bdd, num_
↪bases=32)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (rgc2): RelGraphConv(
      (linear_r): TypedLinear(in_size=32, out_size=32, num_types=1)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (dropout): Dropout(p=0.2, inplace=False)
  )
)
```

[28]:
```
g2._embed_model.relational_embedding.shape
```

[28]:
```
torch.Size([1, 32])
```

[29]:
```
(
    g2._embed_model.relational_embedding.min(),
    g2._embed_model.relational_embedding.mean(),
    g2._embed_model.relational_embedding.max()
)
```

[29]:
```
(tensor(-0.4868, grad_fn=<MinBackward1>),
 tensor(0.0236, grad_fn=<MeanBackward0>),
 tensor(0.4513, grad_fn=<MaxBackward1>))
```

Explore hits

[28]:
```
def to_cpu(tensor):
    """
    Helper for switching between is_gpu to avoid coercion errors
    """
    if is_gpu:
        return tensor.cpu()
    else:
        return tensor
```

[37]:
```
%%time
score2 = g2._score(g2._triplets)
df['score'] = to_cpu(score2).numpy()
type(score2), to_cpu(score2).numpy().min(), to_cpu(score2).numpy().max()
```

```
CPU times: user 998 ms, sys: 27.8 ms, total: 1.03 s
Wall time: 744 ms
```

[37]:
```
(torch.Tensor, 3.049316e-08, 0.99901354)
```

[38]:
```
anom_edges2 = df[to_cpu(score2).numpy() < 0.03]

print('number within threshold', anom_edges2.shape)

anom_edges2[
    anom_edges2['time'].isin(set(red_team[0])) &
    anom_edges2['src_computer'].isin(set(red_team[2])) &
    anom_edges2['dst_computer'].isin(set(red_team[3]))
].reset_index()
```

```
number within threshold (924, 11)
```

[38]:

	index	time	src_domain	dst_domain	src_computer	dst_computer	auth_type	\
0	4000000	151036	U748@DOM1	U748@DOM1	C17693	C305	NTLM	
1	4000001	151648	U748@DOM1	U748@DOM1	C17693	C728	NTLM	
2	4000003	153792	U636@DOM1	U636@DOM1	C17693	C294	NTLM	
3	4000004	155219	U748@DOM1	U748@DOM1	C17693	C5693	NTLM	
4	4000005	155399	U748@DOM1	U748@DOM1	C17693	C152	NTLM	
5	4000007	155591	U748@DOM1	U748@DOM1	C17693	C332	NTLM	
6	4000008	156658	U748@DOM1	U748@DOM1	C17693	C4280	NTLM	
7	4000009	210086	U748@DOM1	U748@DOM1	C18025	C1493	NTLM	

	logontype	authentication_orientation	success_or_failure	RED	score
0	Network	LogOn	Success	1.0	0.006004
1	Network	LogOn	Success	1.0	0.003731
2	Network	LogOn	Success	1.0	0.020784
3	Network	LogOn	Success	1.0	0.023513
4	Network	LogOn	Success	1.0	0.000488
5	Network	LogOn	Success	1.0	0.006469
6	Network	LogOn	Success	1.0	0.005589
7	Network	LogOn	Success	1.0	0.009605

[31]:
```
srcindx = df.src_computer.isin(anom_label.src_computer)
dstindx = df.dst_computer.isin(anom_label.dst_computer)
srcscore = score2.cpu().numpy()[srcindx]
dstscore = score2.cpu().numpy()[dstindx]
```

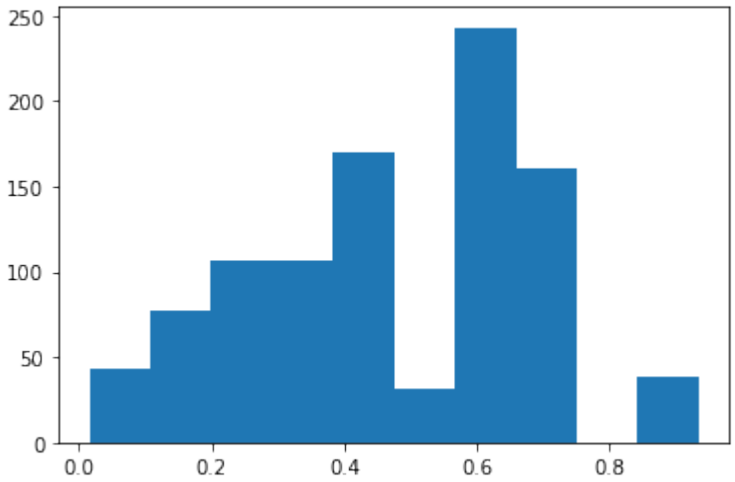
[ ]:
```
sdf = df[srcindx | dstindx]
sdf
```

[33]:
```
# one can see from this graph that it is not entirely obvious that red team nodes are␣
↪where they are, and certianly
gsmall = graphistry.edges(sdf, 'src_computer','dst_computer')
url = gsmall.plot(render=False)
url
```

[33]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=3f5ec495b70449abad38834abafbccfc&
↪type=arrow&viztoken=b274cd2e-e255-49bb-a5ce-34c6c9d54357&usertag=23693246-pygraphistry-
↪0.28.4+78.g6f47e03&splashAfter=1669998005&info=true'
```

[35]:
```
import matplotlib.pyplot as plt
import numpy as np
s = np.r_[srcscore, dstscore]
plt.hist(s)
```

[35]:
```
(array([ 43.,  77., 107., 107., 170.,  31., 243., 160.,   0.,  39.]),
 array([0.01555263, 0.10742634, 0.19930004, 0.29117373, 0.38304743,
        0.47492114, 0.5667948 , 0.6586685 , 0.7505422 , 0.8424159 ,
        0.93428963], dtype=float32),
 <BarContainer object of 10 artists>)
```



Are simpler methods better?

Maybe conditional probability `p(dst_computer | src_computer)` can give us good histograms to tease out red team nodes?

[53]:
```
def 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'
    """
    return df.groupby([given])[x].apply(lambda g: g.value_counts()/len(g))
```

[54]:
```
auth.src_computer.nunique(), auth.dst_computer.nunique()
```

[54]:
```
(10475, 10313)
```

[55]:
```
pd.concat([auth.src_computer, auth.dst_computer]).nunique()
```

[55]:
```
10887
```

[56]:
```
cnp = conditional_probability('dst_computer', given='src_computer', df=auth)
cnp
```

[56]:
```
src_computer
C1            C1       0.438462
              C529     0.130769
              C625     0.092308
              C586     0.080769
              U25      0.076923
                         ...
C9997         C1065    0.010256
              C585     0.010256
              U7       0.005128
              C801     0.005128
              C2162    0.005128
Name: dst_computer, Length: 136119, dtype: float64
```

Conditional Graph

Demo

[57]:
```
x=SRC
given=DST
condprobs = conditional_probability(x, given, df=auth)

cprob = pd.DataFrame(list(condprobs.index), columns=[given, x])
cprob['_probs'] = condprobs.values
```

[58]:
```
# now enrich the edges dataframe with the redteam data
indx = cprob.src_computer.isin(anom_label[SRC]) & cprob[DST].isin(anom_label[DST])
cprob.loc[indx, 'red'] = 1
cprob.loc[~indx, 'red'] = 0
```

[42]:
```
cg = graphistry.edges(cprob, x, given).bind(edge_weight='_probs')
cg.plot(render=False)
```

[42]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=03bcc2f71a054a9c84fa5ce881f462bd&
↪type=arrow&viztoken=abe69b16-c979-4ee5-b954-f6214d2d6b91&usertag=23693246-pygraphistry-
↪refs/pull/408/head&splashAfter=1669918110&info=true'
```

Learning

The conditional graph shows that most of the edge probabilities are between 4e-7 and 0.03, whose bucket contains 91k edges. Thus the chances of finding the red team edges are 10/91000 ~ 1e-4 – slim indeed

[43]:
```
# let's repeat but with reverse conditional
x='src_computer'
given='dst_computer'
condprobs2 = conditional_probability(x, given, df=auth)
```

(continues on next page)

(continued from previous page)

```
cprob2 = pd.DataFrame(list(condprobs2.index), columns=[given, x])
cprob2['_probs'] = condprobs2.values
```

[44]:
```
# now enrich the edges dataframe with the redteam data
indx = cprob2.src_computer.isin(anom_label.src_computer) & cprob2.dst_computer.isin(anom_
↪label.dst_computer)
cprob2.loc[indx, 'red'] = 1
cprob2.loc[~indx, 'red'] = 0
```

Can we do simple std analysis on red vs not?

Not really

[56]:
```
red=X[:11].std(0).values
red
```

[56]:
```
array([6.69291932e+00, 6.94233955e+00, 4.03406261e+00, 1.24375581e+00,
       8.19816652e+00, 5.34599137e+00, 8.55273272e+00, 8.82849580e+00,
       5.97048788e+00, 7.98474697e-01, 7.14580528e-03, 4.22057963e-02,
       2.00462442e-02, 2.38532804e+00, 8.04956952e-02])
```

[57]:
```
import numpy as np
indx = np.random.choice(range(11, len(X)), 11)
rand = X.iloc[indx].std(0).values
rand
```

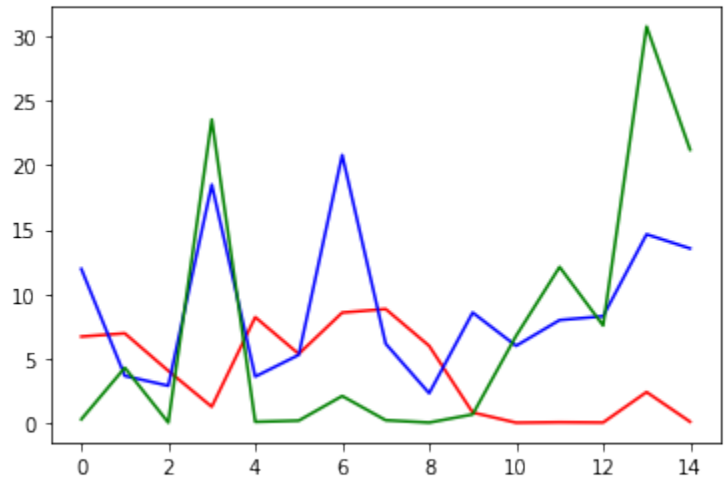
[57]:
```
array([2.72630555e-01, 4.27446082e+00, 1.50943836e-02, 2.35438425e+01,
       7.47794986e-02, 1.66500055e-01, 2.07862744e+00, 1.97366326e-01,
       1.64508509e-02, 6.49012753e-01, 6.75271878e+00, 1.20960431e+01,
       7.55023721e+00, 3.07596829e+01, 2.12004662e+01])
```

[58]:
```
blue=X[11:].std(0).values
blue
```

[58]:
```
array([11.94419565,  3.63960234,  2.87543166, 18.48754619,  3.58461871,
        5.26787065, 20.78975583,  6.13758242,  2.29442998,  8.55447084,
        5.97436329,  7.97298694,  8.26999525, 14.63120211, 13.53291024])
```

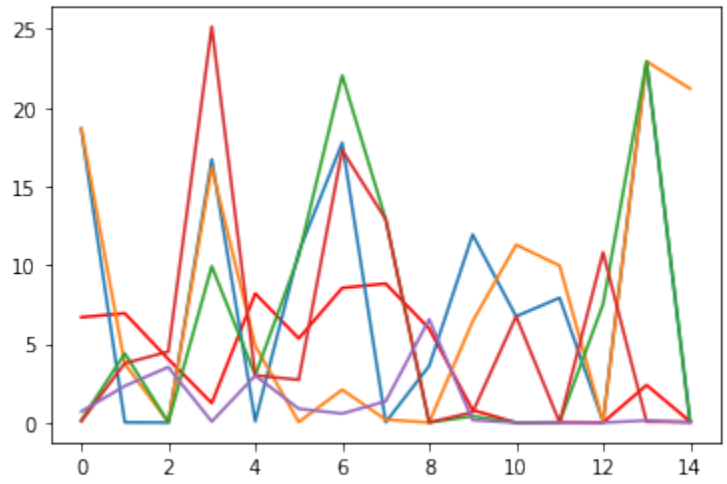
[61]:
```
from matplotlib import pyplot as plt


#plt.plot(red/blue)
plt.plot(red, c='r')
plt.plot(blue, c='b')
plt.plot(rand, c='g')
```

[61]:
```
[<matplotlib.lines.Line2D at 0x7f954901ef70>]
```



[62]:
```
plt.plot(red, c='r')
rands = []
for k in range(4):
    indx = np.random.choice(range(11, len(X)), 11)
    rand = X.iloc[indx].std(0).values
    plt.plot(rand)
    rands.append(rand)

plt.plot(red/np.mean(rands, axis=0))
```

[62]:
```
[<matplotlib.lines.Line2D at 0x7f953806d0d0>]
```



Resource

- http://github.com/graphistry/pygraphistryhttp://github.com/graphistry/pygraphistry
  - Dashboarding with https://github.com/graphistry/graph-app-kit
- https://gradientflow.com/what-is-graph-intelligence/https://gradientflow.com/what-is-graph-intelligence/
- GNN Videos:
  - GCN - https://www.youtube.com/watch?v=2KRAOZIULzw
  - RGCN - https://www.youtube.com/watch?v=wJQQFUcHO5U
  - Euler (combining RNN + GNN)- https://www.youtube.com/watch?v=1t124vguwJ8

[ ]:

10.8.5.4 The age old question, Bot or Not? Attacker or Friendly?

We load a botnet dataset and produce features from the raw data as well as a GNN model that predicts bot traffic. Attacking bots make up less than a percent of the total data.

Feature engineering the data might take a form like this https://github.com/NagabhushanS/Machine-Learning-Based-Botnet-Detection/blob/master/src/dataset_load.py

We create features and models automatically using the `g.featurize` and `g.umap` APIs, demonstrating fast ML pipelines over data that may be complex and multimodal with little more effort than setting some parameters.

[ ]:
```
#! pip install --upgrade graphistry[ai]
```

[ ]:
```
# cd ..
```

[ ]:
```
import os
import graphistry
from graphistry.features import ModelDict

import torch
import numpy as np
import pandas as pd
import matplotlib.pylab as plt

import os
from collections import Counter
from importlib import reload

import warnings
warnings.filterwarnings('ignore')
```

[ ]:
```
## add your username and password here
graphistry.register(api=3, protocol="https", server="hub.graphistry.com", username=os.
↪environ['USERNAME'], password=os.environ['GRAPHISTRY_PASSWORD'])
```

[ ]:
```
# some plot helpers
def fast_plot(g, attr, mask=None, cols=None, interpolation=None):
    plt.figure(figsize=(17,10))
    if cols is None:
        cols = np.arange(getattr(g, attr).shape[1])
    if mask is not None:
        plt.imshow(getattr(g, attr)[mask].values[:,cols], aspect='auto', cmap='hot',␣
↪interpolation=interpolation)
    else:
        plt.imshow(getattr(g, attr).values[:,cols], aspect='auto', cmap='hot',␣
↪interpolation=interpolation)
```

We import the CTU-13 malware dataset

You can find a number of datasets here https://www.stratosphereips.org/datasets-ctu13

[ ]:
```
edf = pd.read_csv('https://gist.githubusercontent.com/silkspace/
↪33bde3e69ae24fee1298a66d1e00b467/raw/dc66bd6f1687270be7098f94b3929d6a055b4438/malware_
↪bots.csv', index_col=0)
```

[ ]:
```
edf
```

[ ]:
```
# let's find the Botnet vs not
T = edf.Label.apply(lambda x: True if 'Botnet' in x else False)
```

[ ]:
```
T
```

[ ]:
```
bot = edf[T]
nbot = edf[~T]
print(f'Botnet abundance: {100*len(bot)/len(edf):0.2f}%')# so botnet traffic makes up a␣
↪tiny fraction of total

# let's balance the dataset in a 10-1 ratio, for speed and demonstrative purposes
negs = nbot.sample(10*len(bot))
edf = pd.concat([bot, negs])  # top part of arrays are bot traffic, then all non-bot␣
↪traffic
edf = edf.drop_duplicates()

# some useful indicators for later that predict Botnet as Bool and Int
Y = edf.Label.apply(lambda x: 1 if 'Botnet' in x else 0)  # np.array(T)

# Later we will use and exploit any meaning shared between the labels in a latent␣
↪distribution

# add it to the dataframe
edf['bot'] = Y
```

[ ]:
```
# name some columns for edges and features
src = 'SrcAddr'
```

(continues on next page)

(continued from previous page)

```
dst = 'DstAddr'
good_cols_with_edges = ['Dur', 'Proto', 'Sport',
       'Dport', 'State', 'TotPkts', 'TotBytes', 'SrcBytes', src, dst]

good_cols_without_edges = ['Dur', 'Proto', 'Sport',
       'Dport', 'State', 'TotPkts', 'TotBytes', 'SrcBytes']

## some encoding parameters
n_topics = 20
n_topics_target = 7
```

Fast Incident Response

An Incident Responder needs to quickly find which IP is the attacker.

If, say, a predictive model enriched the data, responders could repeat the pipeline on new data drastically reducing the search space.

They can see affected computers and log, manage, escalate, and triage alerts using Graphistry playbooks integrations.

We will use Graphistry[ai] to generate such a predictive pipeline, that finds offending nodes (find attacker IPs), as well as the *systems* and *patterns* they exploit, detecting deviations from benign behaviors and instances of known attack behaviors.

[ ]:
```
# load the data using the edges API
g = graphistry.edges(edf, src, dst)
```

[ ]:
```
g.plot()
```

[ ]:
```
# Let's featurize and reduce the dimensionality of the dataset
```

[ ]:
```
# lets umap the data
g2 = g.umap(kind='edges',
            X=good_cols_with_edges,
            y = ['bot'],
            use_scaler='quantile',
            use_scaler_target=None,
            cardinality_threshold=20,
            cardinality_threshold_target=2,
            n_topics=n_topics,
            n_topics_target=n_topics_target,
            n_bins=n_topics_target,
            metric='euclidean',
            n_neighbors=12)
```

[ ]:
```
x, y = g2._edge_features, g2._edge_target
x
```

```
[ ]: # can spot edge features that are bot vs not
fast_plot(g2, '_edge_features', mask=[True]*1500 + [False]*(len(x)-1500))
```

Do we have a predictive model?

Using the x, y's we get from autofeaturization, we fit two RandomForest models

```
[ ]: from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.model_selection import train_test_split
```

```
[ ]: clf = RandomForestClassifier()
rlf = RandomForestRegressor()
```

```
[ ]: X_train, x_test, y_train, y_test = train_test_split(x, y)
rlf.fit(X_train, y_train)
rlf.score(x_test, y_test)
```

```
[ ]: X_train, x_test, y_train, y_test = train_test_split(x, y)
clf.fit(X_train, y_train).score(x_test, y_test)
```

```
[ ]: lengthy_computation = False

if lengthy_computation: # if you have patience or GPUs
    from sklearn.inspection import permutation_importance
    r = permutation_importance(clf, x_test, y_test,
                               n_repeats=10,
                               random_state=0)

    for i in r.importances_mean.argsort()[::-1]:
        if r.importances_mean[i] - 2 * r.importances_std[i] > 0:
            print(f"{x.columns[i]:<8}"
                  f"{r.importances_mean[i]:.3f}"
                  f" +/- {r.importances_std[i]:.3f}")
    tops = r.importances_mean.argsort()[::-1][:10]
else:
    tops = clf.feature_importances_.argsort()[::-1][:10]

tops
```

```
[ ]: # top features that predict bot or not -- and since we used the ip address, we easily␣
↪find the 'feature' (ie target)
x.columns[tops]
```

Let's remove edges and see if there is a model of just 'common features' (ie no ip addresses)

Given learnings, we want to see if there is a model that does not use edge information (ie, no IP addresses, only connection metadata)

[ ]:
```
g3 = g.nodes(edf)#  treat edf as ndf to featurize, since we aren't using the src/dst␣
↪data, no need to add it to .edges(..)
g3 = g3.umap(kind='nodes',
             X=good_cols_without_edges,
             y = 'bot',
             scale =0.1,
             use_scaler='quantile',
             use_scaler_target=None,
             cardinality_threshold=20,
             cardinality_threshold_target=20,
             n_topics=n_topics,
             n_topics_target=n_topics_target,
             n_bins=n_topics,
             metric='euclidean',
             n_neighbors=20)
```

[ ]:
```
X = g3._node_features
y = g3._node_target
```

[ ]:
```
X
```

[ ]:
```
y
```

[ ]:
```
fast_plot(g3, '_node_features')  # one can clearly see that bot vs non-bot features are␣
↪different
```

[ ]:
```
X_train, x_test, y_train, y_test = train_test_split(X, y)
clf.fit(X_train, y_train).score(x_test, y_test)
```

[ ]:
```
# if interested to find sensitivities
X_train, x_test, y_train, y_test = train_test_split(X, y)
rlf.fit(X_train, y_train).score(x_test, y_test)
```

[ ]:
```
lengthy_computation = False

if lengthy_computation: # if you have patience or GPUs
    from sklearn.inspection import permutation_importance
    r = permutation_importance(clf, x_test, y_test,
                               n_repeats=10,
                               random_state=0)

    for i in r.importances_mean.argsort()[::-1]:
        if r.importances_mean[i] - 2 * r.importances_std[i] > 0:
            print(f"{x.columns[i]:<8}"
                  f"{r.importances_mean[i]:.3f}"
                  f" +/- {r.importances_std[i]:.3f}")
```

(continues on next page)

(continued from previous page)

```
    tops = r.importances_mean.argsort()[::-1][:10]
else:
    tops = clf.feature_importances_.argsort()[::-1][:10]

tops
```

[ ]:
```
topcols = X.columns[tops]
topcols
```

[ ]:
```
nres = X[y.values==0][topcols].describe() #not bot
res = X[y.values==1][topcols].describe() #bot
res
```

Hence we see that including just common features clusters botnet traffic together under featurization and UMAP

[ ]:
```
a=res.loc['mean']/nres.loc['mean']
a.plot(kind='bar')
print('Bot Mean divided by Non-Bot Mean')
```

[ ]:
```
b=nres.loc['mean']/res.loc['mean']
b.plot(kind='bar', rot=77)
print('Not-Bot Mean divided by Bot Mean')
```

Now we dive deeper

Let's encode the graph as a DGL graph for use in Machine Learning

[ ]:
```
from graphistry.networks import LinkPredModelMultiOutput, train_link_pred
```

[ ]:
```
# first, let's examine the cardinality of labels and get a sense of the different flows
cnt = Counter(g2._edges['Label'])
cnt.most_common()
```

[ ]:
```
## can we learn a better representation of these labels?
len(cnt)
```

[ ]:
```
# let's build a GNN model with the 'Label' being reduced to a n_topics_target < 70+␣
↪dimensional representation
g4 = g.build_gnn(y_edges = 'Label',
            use_node_scaler='quantile',
            use_node_scaler_target=None,
            cardinality_threshold=2,
            cardinality_threshold_target=2,
            n_topics=n_topics,
            n_topics_target=n_topics_target,
            )
```

[ ]:
```
g4._edge_target  # it identifies `Label: *, download, botnet` strongly over bot vs not
```

[ ]:
```
fast_plot(g4, '_edge_target')  # clear to see bot vs not as a regressive label
```

[ ]:
```
# the deep learning graph
G = g4._dgl_graph

# define the model from the data
node_features = G.ndata["feature"].float()
n_feat = node_features.shape[1]
# we are predicting edges
edge_label = G.edata["target"]
labels = edge_label.argmax(1)  # turn regressive label into

n_targets = edge_label.shape[1]
train_mask = G.edata["train_mask"]
test_mask = G.edata["test_mask"]

latent_dim = 32
n_output_feats = 16  # this is equal to the latent dim output of the SAGE net

model = LinkPredModelMultiOutput(n_feat, latent_dim, n_output_feats, n_targets)

pred = model(G, node_features)  # the untrained graph

assert G.num_edges() == pred.shape[0], "something went wrong"

print(f"output of model has same length as the number of edges: {pred.shape[0]}")
print(f"number of edges: {G.num_edges()}\n")

# Train model
train_link_pred(model, G, epochs=2900)
```

[ ]:
```
# trained comparison
logits = model(G, node_features)
pred = logits.argmax(1)

accuracy = sum(pred[test_mask] == labels[test_mask]) / len(
    pred[test_mask]
)
print("-" * 30)
print(f"Final Accuracy: {100 * accuracy:.2f}%")
```

Take away – * we encode targets in the latent space of messy multi targets * we can do inductive prediction on new graphs using GNN model

[ ]:
```
cnts = Counter(np.array(labels)).most_common()
cnts
```

[ ]:
```
# most common classifier (that always predicts one class), would score:
print(f'{100*cnts[0][1]/len(labels):.2f}%')
```

[ ]:
```
# get node features after training of the GNN model
enc = model.sage(G, node_features.float())
enc
```

[ ]:
```
plt.figure(figsize=(15,7))
plt.imshow(enc.detach().numpy(), aspect='auto')
```

Contributions

Now we know how to take raw data and turn them into actionable features and models using the Graphistry[ai] API.

Integrate them into your pipelines and https://join.slack.com/t/graphistry-community!

[ ]:
```

```

10.8.6 Plugins - Data Providers

10.8.6.1 AlientVault OTX <> Graphistry: Industry threat map

[ ]:
```
#!pip install graphistry -q
#!pip install OTXv2 -q
```

[ ]:
```
import graphistry
import pandas as pd
from OTXv2 import OTXv2, IndicatorTypes
from gotx import G_OTX
```

[ ]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

otx = OTXv2("MY_KEY")
g_otx = G_OTX(graphistry, otx)
```

[ ]:
```
#May take a couple minutes

pulses = otx.getall(author_name='AlienVault')
pulses_df = g_otx.pulses_to_df(pulses)
```

V1: Country<>Industry

```
[ ]: g_otx.industrymap(pulses_df).plot()
```

V2: Country <- Threat Report -> Industry

```
[ ]: g_otx.industrymap(pulses_df, include_indicators=True).plot()
```

```
[ ]:
```

10.8.6.2 AlientVault OTX <> Graphistry: LockerGoga investigation

```
[ ]: #!pip install graphistry -q
     #!pip install OTXv2 -q
```

```
[ ]: import graphistry
     import pandas as pd
     from OTXv2 import OTXv2, IndicatorTypes
     from gotx import G_OTX
```

```
[ ]: # To specify Graphistry account & server, use:
     # graphistry.register(api=3, username='...', password='...', protocol='https', server=
     ↪'hub.graphistry.com')
     # For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

     otx = OTXv2("MY_KEY")
     g_otx = G_OTX(graphistry, otx)
```

Start: rough hits

We find there are 3 clusters of activity

```
[ ]: lockergoga_pulses = otx.search_pulses('LockerGoga').get('results')
```

```
[ ]: lockergoga_pulses_df = g_otx.pulses_to_df(lockergoga_pulses)
     lockergoga_indicators_df = g_otx.pulses_to_indicators_df(lockergoga_pulses)

     g = g_otx.indicatormap(lockergoga_pulses_df, lockergoga_indicators_df)

     g.plot()
```

Continue: Expand on IPv4 hits

Let's expand the small cluster related to "Powershell Backdoor calling back on port 443". Use the OTX API to get other pulses containing the same IP address and then expand them and create a new graph

[ ]:
```
ip_pulses = otx.get_indicator_details_by_section(IndicatorTypes.IPv4, lockergoga_
↪indicators_df[lockergoga_indicators_df['indicator_type'] == 'IPv4'].values[0][0])
```

[ ]:
```
ip_pulses_df = g_otx.indicator_details_by_section_to_pulses_df(ip_pulses)
ip_indicators_df = g_otx.indicator_details_by_section_to_indicators_df(ip_pulses)

g_otx.indicatormap(ip_pulses_df, ip_indicators_df).plot()
```

10.8.6.3 AlienVault USM event visualizer

- Search
- Graphistry helper
- Plot

Plug in your credentials in the first cells, put in a custom search, then plot.

[ ]:
```
!pip install requests_oauthlib -q
```

[ ]:
```
from unimatrix import UnimatrixClient
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

client = UnimatrixClient("my_subdomain", "client_id", "client_secret")
```

Search

[ ]:
```
params = {'plugin': 'AlienVault Agent - Windows EventLog', 'event_name':'Network␣
↪connection'}
events_df = client.get_events_df(num_items=25000, params=params)

print('# rows', len(events_df))
events_df.sample(3)
```

Helper

[ ]:

```
def graph(events_df):
    red_events = events_df
    red_events = red_events.assign(source_process= red_events['source_process'].
↪apply(lambda v: v.split('\\')[-1]),
                                   destination_organisation=red_events.destination_
↪organisation.fillna('INTERNAL'))
    red_events = red_events.assign(source_process_id=red_events.apply(lambda row:␣
↪str(row['source_process_id']) + '::' + str(row['source_canonical']), axis=1)).reset_
↪index()

    edges = {
        'source_username': ['source_canonical', 'source_process_id', 'source_process_id
↪'],
        'source_process_id': ['source_process', 'iocs', 'event_category', 'watchlist'],
        'source_canonical': ['source_address', 'source_process', 'source_name', 'source_
↪process_id',
                             'destination_name', 'destination_canonical', 'destination_
↪address'],
        'destination_canonical': ['destination_name', 'destination_address']
    }

    node_types = list(edges.keys())
    for dsts in edges.values():
        node_types.extend(dsts)
    node_types = list(set(node_types))

    g = graphistry.hypergraph(
        red_events.astype(str),
        entity_types=node_types,
        direct=True,
        opts={
            'EDGES': edges,
            'CATEGORIES': {
                'asset': ['source_canonical', 'source_address', 'source_canonical',
↪'destination_name', 'destination_canonical', 'destination_address' ]
            }
        })['graph']

    types = list(g._nodes['category'].unique())
    type_to_color = {t: types.index(t) % 10 for t in types}

    events = list(g._edges['event_category'].unique())
    event_to_color = {t: events.index(t) % 10 for t in events}

    g = g.nodes(g._nodes.assign(p_color=g._nodes['category'].apply(lambda t: type_to_
↪color[t]))).bind(point_color='p_color')
    g = g.edges(g._edges[~(g._edges.dst.str.match('.*::nan') | g._edges.src.str.match('.
↪*::nan'))])
    g = g.edges(g._edges.assign(e_color=g._edges['event_category'].apply(lambda t: event_
```

(continues on next page)

(continued from previous page)

```
↪to_color[t]))).bind(edge_color='e_color')

    return g
```

Visualize

```
[ ]: graph(events_df).plot()
```

```
[ ]:
```

10.8.6.4 Graphistry Neptune Gremlin identity graph demo

PyGraphistry helps connect to graph data sources, wrangle them with Python dataframe tools, and visualize them with Graphistry. It's often used in notebooks, data apps, and dashboards.

This notebook uses PyGraphistry to quickly: * **Connect to**https://aws.amazon.com/neptune/ * **Run**http://tinkerpop.apache.org/**queries** via built-in bindings over https://pypi.org/project/gremlinpython/ * **Convert to dataframes** for data wrangling: CPU via https://pandas.pydata.org/ and GPU via https://rapids.ai/ * **Visualize** by automatically generating rich, interactive, & GPU-accelerated https://www.graphistry.com (external link) graph visualization sessions * **Share & embed** your beautiful results

For any API used below, run `help(graphistry.the_method)` for a quick view of its docs

The demo is on AWS Neptune's identity graph data sample from our joint https://aws.amazon.com/blogs/database/enabling-low-code-graph-data-apps-with-amazon-neptune-and-graphistry/. If you have your own dataset, including non-identity data, the example queries should still work.

Setup

Optional - Quicklaunch via https://github.com/graphistry/graph-app-kit/blob/master/docs/neptune.md: * **Neptune**: It is tested on Neptune's https://github.com/graphistry/graph-app-kit/blob/master/docs/neptune.md, and you can swap in your own * **Graphistry**: Use your own, get a https://www.graphistry.com/get-started, or https://github.com/graphistry/graph-app-kit/blob/master/docs/neptune.md * **Notebook**: Use your own, or https://github.com/graphistry/graph-app-kit/blob/master/docs/neptune.md

If you hit `gremlinpython` event runtime bugs, try https://gist.github.com/lmeyerov/459f6f0360abea787909c7c8c8f04cee

Install

Already provided in graphistry envs

[1]:
```
# ! pip install -u gremlinpython graphistry
# ! pip install -u pandas
# see https://rapids.ai/ if trying GPU dataframes
```

Imports

[2]:
```
! pip show gremlinpython graphistry | grep 'Name\|Version'
```

```
Name: gremlinpython
Version: 3.4.10
Name: graphistry
Version: 0.19.0+5.g5ce1d3fb0
```

[3]:
```
import graphistry
graphistry.__version__
```

[3]:
```
'0.19.0+5.g5ce1d3fb0'
```

Configure

[4]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

[29]:
```
NEPTUNE_READER_PROTOCOL='wss'
NEPTUNE_READER_HOST='neptunedbcluster-abc.cluster-ro-xyz.us-east-1.neptune.amazonaws.com'
NEPTUNE_READER_PORT='8182'

endpoint = f'{NEPTUNE_READER_PROTOCOL}://{NEPTUNE_READER_HOST}:{NEPTUNE_READER_PORT}/
↪gremlin'
endpoint
```

[29]:
```
'wss://neptunedbcluster-abc.cluster-ro-xyz.us-east-1.neptune.amazonaws.com:8182/gremlin'
```

[6]:
```
#import logging
#logging.basicConfig(level=logging.DEBUG)
```

Connect

[7]:
```
graphistry.register(**GRAPHISTRY_CFG)

g = graphistry.neptune(endpoint=endpoint)

g._gremlin_client
```

[7]:
```
<gremlin_python.driver.client.Client at 0x7fdfc230e3d0>
```

Query & plot

- PyGraphistry automatically converts gremlin results into node/edge dataframes
- Edge queries typically only return node IDs; call `fetch_nodes()` to enrich your `g._nodes` dataframe
- PyGraphistry plots dataframes

[25]:
```
%%time

g2 = g.gremlin('g.E().limit(10000)')
```

```
CPU times: user 4.96 s, sys: 27.9 ms, total: 4.99 s
Wall time: 4.95 s
```

[26]:
```
print('NODES:')
g2._nodes.info()
g2._nodes.sample(3)
```

```
NODES:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8106 entries, 0 to 8105
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      8106 non-null   object
 1   label   8106 non-null   object
dtypes: object(2)
memory usage: 126.8+ KB
```

[26]:

	id	label
4102	ed95a9a5be30e4c8/e212d4b4d4a865a/7e3e41e09dfe6...	website
6496	6ea77fc3ea42bd5b/87be29bd5615083/d4392e74543e413	website
7540	4c980617e02858a4/7de2f069da3a3655/30591f4d8c71...	website

[27]:
```
print('EDGES:')
print(g2._edges.info())

g2._edges.sample(3)
```

```
EDGES:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
```

(continues on next page)

(continued from previous page)

```
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      10000 non-null  object
 1   label   10000 non-null  object
 2   src     10000 non-null  object
 3   dst     10000 non-null  object
dtypes: object(4)
memory usage: 312.6+ KB
None
```

[27]:

```
                                          id    label  \
2814  f7803bf0ac187592421c0695792b698f43b596ce  visited
8081  fe80cddfec97a7dd802cf93cf277da01d9b5fb65  visited
2046  4e5290971de41c1e1bcb7433e53ffc6321e410cf  visited

                                                  src  \
2814  556de63e26686d50/95263499b67bbda1?f300c39f4f33...
8081                  3ccec85ce35ea661?fa76e6024017220f
2046  6ea77fc3ea42bd5b/9c280de73bf0fb32/bb555a4d63de...

                                    dst
2814  48e740025e70e4e38dc87928cd45357c
8081  23c31ea91be100fd224dff1499939851
2046  9e77c2a52fdf9f9b7416e85cabaf7c76
```

[28]:

```
%%time

# Enrich nodes dataframe with any available server property data

g3 = g2.fetch_nodes()

print(g3._nodes.info())

g3._nodes.sample(3)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8106 entries, 0 to 8105
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      8106 non-null   object
 1   label   8106 non-null   object
dtypes: object(2)
memory usage: 126.8+ KB
None
CPU times: user 4.32 s, sys: 43.9 ms, total: 4.37 s
Wall time: 4.33 s
```

[28]:

```
                                                    id    label
1242  4c980617e02858a4/7de2f069da3a3655/30591f4d8c71...  website
3190  493a46bbfd2029ae/4a0cad2f071a71ce/f9ba18598922...  website
6782  4c980617e02858a4/7de2f069da3a3655/30591f4d8c71...  website
```

[19]:
```
%%time

g3.plot()
```

```
CPU times: user 59.8 ms, sys: 4 ms, total: 63.8 ms
Wall time: 1.68 s
```

[19]:
```
<IPython.core.display.HTML object>
```

Customize your visuals & Embed

Graphistry visualizes data with smart defaults: community-based coloring, degree-based sizing, force-directed layout, auto-zoom, and built-in visual analytics. However, it often helps to configure your visuals ahead of time.

Example: * Enable legend on new column 'type' * Color nodes by node column 'type' * Pick icons based on node type * Set background color to match notebook * Use a tighter layout

See further examples at the https://github.com/graphistry/pygraphistry

[24]:
```
%%time

g4 = (g3

      # Add node column 'type' based on gremlin-provided column 'label'
      # The legend auto-detects this column and appears
      .nodes(lambda g: g._nodes.assign(type=g._nodes['label']))

      .encode_point_color('type', categorical_mapping={
          'website': 'blue',
          'transientId': 'green'
      })

      .encode_point_icon('type', categorical_mapping ={
          'website': 'link',
          'transientId': 'barcode'
      })

      .addStyle(bg={'color': '#eee'}, page={'title': 'My Graph'})

      # More: https://hub.graphistry.com/docs/api/1/rest/url/
      .settings(url_params={'play': 2000})
)

g4.plot()
```

```
CPU times: user 63.5 ms, sys: 3.88 ms, total: 67.3 ms
Wall time: 1.62 s
```

[24]:
```
<IPython.core.display.HTML object>
```

Generate URL for other systems

[23]:
```
%%time

url = g4.plot(render=False)

url
```

```
CPU times: user 64.8 ms, sys: 0 ns, total: 64.8 ms
Wall time: 1.67 s
```

[23]:
```
'https://hub.graphistry.com/graph/graph.html?dataset=7405d0ac396a47ea9ee84acab7b0b31d&
↪type=arrow&viztoken=c5e68946-e922-487e-9484-ef8fc9e2c8f9&usertag=5bf3845f-pygraphistry-
↪0.19.0+5.g5ce1d3fb0&splashAfter=1625879227&info=true&strongGravity=False&play=2000'
```

Next steps

- Go deeper with https://github.com/graphistry/pygraphistry: Examples for customization, GPU graph analytics, and more
- Explore https://pypi.org/project/gremlinpython/
- Dashboarding with https://github.com/graphistry/graph-app-kit's https://github.com/graphistry/graph-app-kit/blob/master/docs/neptune.md
  - Amazon Neptune's https://aws.amazon.com/blogs/database/enabling-low-code-graph-data-apps-with-amazon-neptune-and-graphistry/
- https://www.graphistry.com/get-started
- https://hub.graphistry.com/docs: REST, React, JS, ...

[ ]:

10.8.6.5 Graphistry for Neptune using pygraphistry bolt connector

This example uses pygraphistry bolt helper class to run queries against AWS Neptune and retrieve query results as graph, then the bolt helper function extracts all the nodes and edges into the dataframes automatically. Then visualize the resulting datasets using Graphistry.

[ ]:
```
!pip install --user neo4j
```

[ ]:
```
!pip install --user awswrangler
```

[ ]:
```
!pip install --user graphistry
```

make sure to restart kernel after pip install

[ ]:
```
import awswrangler as wr
import pandas as pd
import graphistry
graphistry.__version__
```

Configure graphistry connnection

[ ]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')

# To run from a graphistry-host jupyter notebook:
# graphistry.register(api=3, username="...", password="...", protocol="http", server=
↪"nginx")

# to use personal keys:
# graphistry.register(api=3, protocol="...", server="...", personal_key_id='pkey_id',␣
↪personal_key_secret='pkey_secret')   # Key instead of username+password+org_name

# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

graphistry.register(api=3, username="...", password="...", protocol="...", server="...")
```

Configure Neptune connection endpoint:

[ ]:
```
# update with your Neptune endpoint name:
url='NEPTUNE_NAME.REGION.neptune.amazonaws.com'
```

[ ]:
```
iam_enabled = False             # Set to True/False based on the configuration of your␣
↪cluster
neptune_port = 8182             # Set to the Neptune Cluster Port, Default is 8182
neptune_region = 'us-east-1'    # Set to neptune region

client = wr.neptune.connect(url, neptune_port, iam_enabled=iam_enabled, region=neptune_
↪region)
```

[ ]:
```
# check status of neptune connection:
client.status()
```

Connect to Neptune using pygraphistry bolt connector

[ ]:
```
from neo4j import GraphDatabase
uri = f"bolt://{url}:8182"
driver = GraphDatabase.driver(uri, auth=("ignored", "ignored"), encrypted=True)

graphistry.register(bolt=driver)
g = graphistry.cypher("MATCH (a)-[r]->(b) return a, r, b limit 10000")
```

[ ]:
```
g.plot()
```

[ ]:
```

```

10.8.6.6 ArangoDB with Graphistry

We explore Game of Thrones data in `ArangoDB` to show how Arango's graph support interops with Graphistry pretty quickly.

This tutorial shares two sample transforms: * Visualize the full graph * Visualize the result of a traversal query

Each runs an AQL query via `python-arango`, automatically converts to `pandas`, and plots with `graphistry`.

Setup

[ ]:
```
!pip install python-arango --user -q
```

[1]:
```
from arango import ArangoClient
import pandas as pd
import graphistry
```

[3]:
```
def paths_to_graph(paths, source='_from', destination='_to', node='_id'):
    nodes_df = pd.DataFrame()
    edges_df = pd.DataFrame()
    for graph in paths:
        nodes_df = pd.concat([ nodes_df, pd.DataFrame(graph['vertices']) ], ignore_
↪index=True)
        edges_df = pd.concat([ edges_df, pd.DataFrame(graph['edges']) ], ignore_
↪index=True)
    nodes_df = nodes_df.drop_duplicates([node])
    edges_df = edges_df.drop_duplicates([node])
    return graphistry.bind(source=source, destination=destination, node=node).
↪nodes(nodes_df).edges(edges_df)

def graph_to_graphistry(graph, source='_from', destination='_to', node='_id'):
    nodes_df = pd.DataFrame()
    for vc_name in graph.vertex_collections():
        nodes_df = pd.concat([nodes_df, pd.DataFrame([x for x in graph.vertex_
↪collection(vc_name)])], ignore_index=True)
    edges_df = pd.DataFrame()
```

(continues on next page)

(continued from previous page)

```
    for edge_def in graph.edge_definitions():
        edges_df = pd.concat([edges_df, pd.DataFrame([x for x in graph.edge_
↪collection(edge_def['edge_collection'])])], ignore_index=True)
    return graphistry.bind(source=source, destination=destination, node=node).
↪nodes(nodes_df).edges(edges_df)
```

Connect

[ ]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

[4]:
```
client = ArangoClient(protocol='http', host='localhost', port=8529)
db = client.db('GoT', username='root', password='1234')
```

Demo 1: Traversal viz

- Use `python-arango`'s `traverse()` call to descendants of Ned Stark
- Convert result paths to pandas and Graphistry
- Plot, and instead of using raw Arango vertex IDs, use the first name

[7]:
```
paths = db.graph('theGraph').traverse(
    start_vertex='Characters/4814',
    direction='outbound',
    strategy='breadthfirst'
)['paths']
```

[8]:
```
g = paths_to_graph(paths)
g.bind(point_title='name').plot()
```

[8]:
```
<IPython.core.display.HTML object>
```

Demo 2: Full graph

- Use `python-arango` on a graph to identify and download the involved vertex/edge collections
- Convert the results to pandas and Graphistry
- Plot, and instead of using raw Arango vertex IDs, use the first name

[11]:
```
g = graph_to_graphistry( db.graph('theGraph') )
g.bind(point_title='name').plot()
```

[11]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
```

10.8.6.7 Databricks <> Graphistry Tutorial: Notebooks & Dashboards on IoT data

This tutorial visualizes a set of sensors by clustering them based on lattitude/longitude and overlaying summary statistics

We show how to load the interactive plots both with Databricks notebook and dashboard modes. The general flow should work in other PySpark environments as well.

Steps:

- Install Graphistry
- Prepare IoT data
- Plot in a notebook
- Plot in a dashboard
- Plot as a shareable URL

Install & authenticate with graphistry server

[ ]:
```
# Uncomment and run first time or
#  have databricks admin install graphistry python library:
#  https://docs.databricks.com/en/libraries/package-repositories.html#pypi-package

#%pip install graphistry
```

[ ]:
```
# Required to run after pip install to pick up new python package:
dbutils.library.restartPython()
```

[ ]:
```
import graphistry  # if not yet available, install pygraphistry and/or restart Python␣
↪kernel using the cells above
graphistry.__version__
```

Use databricks secrets to retrieve graphistry creds and pass to register

[ ]:
```

# As a best practice, use databricks secrets to store graphistry personal key (access␣
↪token)
# create databricks secrets: https://docs.databricks.com/en/security/secrets/index.html
# create graphistry personal key: https://hub.graphistry.com/account/tokens

graphistry.register(api=3,
                    personal_key_id=dbutils.secrets.get(scope="my-secret-scope", key=
↪"graphistry-personal_key_id"),
                    personal_key_secret=dbutils.secrets.get(scope="my-secret-scope", key=
↪"graphistry-personal_key_secret"),
                    protocol='https',
                    server='hub.graphistry.com')

# Alternatively, use username and password:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
```

(continues on next page)

(continued from previous page)

```
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Prepare IoT data

Sample data provided by Databricks

We create tables for different plots:

- Raw table of device sensor reads
- Summarized table:
  - rounded latitude/longitude
  - summarize min/max/avg for battery_level, c02_level, humidity, timestamp

[ ]:
```
# Load the data from its source.
devices = spark.read \
  .format('json') \
  .load('/databricks-datasets/iot/iot_devices.json')

# Show the results.
print('type: ', str(type(devices)))
display(devices.take(10))
```

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql.functions import concat_ws, col, round

devices_with_rounded_locations = (
    devices
    .withColumn(
        'location_rounded1',
        concat_ws(
            '_',
            round(col('latitude'), 0).cast('integer'),
            round(col('longitude'), 0).cast('integer')))
    .withColumn(
        'location_rounded2',
        concat_ws(
            '_',
            round(col('latitude'), -1).cast('integer'),
            round(col('longitude'), -1).cast('integer')))
)

cols = ['battery_level', 'c02_level', 'humidity', 'timestamp']
id_cols = ['cca2', 'cca3', 'cn', 'device_name', 'ip', 'location_rounded1', 'location_
↪rounded2']
devices_summarized = (
    devices_with_rounded_locations.groupby('device_id').agg(
        *[F.min(col) for col in cols],
        *[F.max(col) for col in cols],
        *[F.avg(col) for col in cols],
```

(continues on next page)

(continued from previous page)

```
        *[F.first(col) for col in id_cols]
    )
)

# [(from1, to1), ...]
renames = (
    [('device_id', 'device_id')]
    + [(f'first({col})', f'{col}') for col in id_cols]
    + [(f'min({col})', f'{col}_min') for col in cols]
    + [(f'max({col})', f'{col}_max') for col in cols]
    + [(f'avg({col})', f'{col}_avg') for col in cols]
 )
devices_summarized = devices_summarized.select(list(
       map(lambda old,new:F.col(old).alias(new),*zip(*renames))
       ))

display(devices_summarized.take(10))
```

Notebook plot

- Simple: Graph connections between `device_name` and `cca3` (country code)
- Advanced: Graph multiple connections, like `ip -> device_name` and `locaation_rounded1 -> ip`

[ ]:
```
(
    graphistry
        .edges(devices.sample(fraction=0.1).toPandas(), 'device_name', 'cca3') \
        .settings(url_params={'strongGravity': 'true'}) \
        .plot()
)
```

[ ]:
```
hg = graphistry.hypergraph(
    devices_with_rounded_locations.sample(fraction=0.1).toPandas(),
    ['ip', 'device_name', 'location_rounded1', 'location_rounded2', 'cca3'],
    direct=True,
    opts={
        'EDGES': {
            'ip': ['device_name'],
            'location_rounded1': ['ip'],
            'location_rounded2': ['ip'],
            'cca3': ['location_rounded2']
        }
    })
g = hg['graph']
g = g.settings(url_params={'strongGravity': 'true'})  # this setting is great!

g.plot()
```

Dashboard plot

- Make a `graphistry` object as usual...
- ... Then disable the splash screen and optionally set custom dimensions

The visualization will now load without needing to interact in the dashboard (`view -> + New Dashboard`)

[ ]:
```
(
    g
        .settings(url_params={'splashAfter': 'false'})  # extends existing setting
        .plot(override_html_style="""
            border: 1px #DDD dotted;
            width: 50em; height: 50em;
        """)
)
```

Plot as a Shareable URL

[ ]:
```
url = g.plot(render=False)
url
```

[ ]:
```

```

10.8.6.8 Tutorial: Using Azure Data Explorer's Persistent Graphs with Kusto & Graphistry

This tutorial demonstrates integrating Azure Data Explorer's (ADX) Persistent Graphs with PyGraphistry, enabling easy GPU-accelerated graph visualization and analytics.

Why Integrate

Microsoft's ADX https://azure.microsoft.com/en-us/updates?id=495985 lets you define and reuse graph relationships directly with ADX. Native support brings reuse and speed.

PyGraphistry's GPU-accelerated visual analytics pipelinesthat make complex graph investigations more interactive, intuitive, and insightful. Teams typically use Graphistry from existing workflows in notebooks, dashboards, and custom web apps to quickly make insightful graph experiences.

Together, they simplify and accelerate full investigations into data already in Azure Data Explorer. Teams get to leverage their exsting investments into Kusto Query Language (KQL) and gain the ability to answer relationship-centric questions in domains like security, IT operations, user behavior, and supply chains, even at large scales.

For a genAI-native approach where analysts can work in natural language to talk to Kusto and generate Graphistry visualizations, you may also be interested in https://www.louie.ai.

Tutorial Outline

You'll learn to:

- Query Kusto and ADX graphs with PyGraphistry
- Create persistent graphs in Azure Data Explorer from a CSV
- Explore and visualize results as dataframes and Graphistry GPU graph visualizations
- Create graph pipelines with PyGraphistry

Let's begin!

Setup

Install pygraphistry and the Kusto python client

```
# Just Graphistry; bring your own Kusto install
pip install graphistry

# Bundled Kusto install
pip install graphistry[kusto]
```

Take it for a spin:

Connect to Kusto and Graphistry

Get a free https://hub.graphistry.com (external link) or run your own https://www.graphistry.com/get-started

To learn more about authentication methods for different Graphistry configurations, check out https://pygraphistry.readthedocs.io/en/latest/server/register.html

[1]:
```
import graphistry
from datetime import datetime
```

[ ]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

[ ]:
```
KUSTO_CONF = {
    "cluster": "https://<clustername>.<region>.kusto.windows.net",
    "database": "<YourDatabase>"
}


graphistry.configure_kusto(**KUSTO_CONF)
```

Ingest data into your Azure Data Explorer cluster.

Import the RedTeam50k dataset used in our https://github.com/graphistry/pygraphistry/blob/master/demos/ai/cyber/cyber-redteam-umap-demo.ipynb into your Azure Data Explorer cluster.

The dataset is a massaged version of the dataset publish by Alexander D. Kent.

Executing using graphistry

With your registered and configured pygraphistry object it is now easy to execute Kusto.

We load the redteam50k dataset into our cluster.

The "kql" function returns a list of dataframes.

[ ]:
```
graphistry.kql("""
.create-or-alter function graphistryRedTeam50k () {
    externaldata(index:long, event_time:long, src_domain:string, dst_domain:string, src_
↪computer:string, dst_computer:string, auth_type:string, logontype:string,␣
↪authentication_orientation:string, success_or_failure:string, RED:int, feats:string,␣
↪feats2:string)
    [
        h@"https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/data/
↪graphistry_redteam50k.csv"
    ]
    with(format="csv", ignoreFirstRecord=true)
    | extend event_time = datetime(2024-01-01) + event_time * 1s
}
""")
```

Grabbing a sample of data

[5]:
```
# Grabbing the first dataframe

df = graphistry.kql("graphistryRedTeam50k | take 100")

df.head(5)
```

```
Query returned 1 results shapes: [(100, 13)] in 0.374 sec
```

[5]:
```
      index               event_time    src_domain    dst_domain  \
0  30526246 2024-01-02 19:16:45+00:00   C7048$@DOM1   C7048$@DOM1
1   5928201 2024-01-01 10:28:10+00:00  C15034$@DOM1  C15034$@DOM1
2  21160461 2024-01-02 08:29:52+00:00    U2075@DOM1    U2075@DOM1
3   2182328 2024-01-01 06:06:59+00:00   C3547$@DOM1   C3547$@DOM1
4  28495743 2024-01-02 16:26:12+00:00    C567$@DOM1    C567$@DOM1

  src_computer dst_computer auth_type logontype authentication_orientation  \
0        C7048          TGT         ?         ?                        TGS
1       C15034         C467         ?         ?                        TGS
2         C529         C529         ?   Network                     LogOff
3         C457         C457         ?   Network                     LogOff
4         C574         C523  Kerberos   Network                      LogOn
```

(continues on next page)

(continued from previous page)

```
  success_or_failure  RED                       feats       feats2
0            Success    0               C7048 TGT ? ?    C7048 TGT
1            Success    0            C15034 C467 ? ?  C15034 C467
2            Success    0         C529 C529 ? Network    C529 C529
3            Success    0         C457 C457 ? Network    C457 C457
4            Success    0  C574 C523 Kerberos Network    C574 C523
```

Building the schema and persisting the graph

A graph model defines the specifications of a graph stored in your database metadata.

Schema definition: * Node and edge types with their properties * Data source mappings: Instructions for building the graph from tabular data * Labels: Both static (predefined) and dynamic (generated at runtime) labels for nodes and edges * Graph models contain the blueprint for creating graph snapshots, not the actual graph data.

Read more: https://learn.microsoft.com/en-us/kusto/management/graph/graph-persistent-overview?view=microsoft-fabric#graph-models

[ ]:
```
GRAPH_NAME = "graphistryRedTeamGraph"
graphistry.kql(f".create-or-alter graph_model {GRAPH_NAME}" + """```
{
    "Schema": {
        "Nodes": {
            "Computer": {"computerName": "string", "RED":"int"},
            "Domain": {"domainName": "string", "RED":"int"}
        },
        "Edges": {
            "AUTHENTICATES": {
                "event_time": "datetime",
                "src_computer": "string",
                "dst_computer": "string",
                "src_domain": "string",
                "dst_domain": "string",
                "auth_type": "string",
                "logontype": "string",
                "authentication_orientation": "string",
                "success_or_failure": "string",
                "RED": "int"
            }
        }
    },
    "Definition": {
        "Steps": [
            {
                "Kind": "AddNodes",
                "Query": "graphistryRedTeam50k | project computerName = src_computer,␣
↪RED, nodeType = 'Computer'",
                "NodeIdColumn": "computerName",
                "Labels": ["Computer"],
```

(continues on next page)

(continued from previous page)

```
                "LabelsColumn": "nodeType"
            },
            {
                "Kind": "AddNodes",
                "Query": "graphistryRedTeam50k | project computerName = dst_computer,␣
↪RED, nodeType = 'Computer'",
                "NodeIdColumn": "computerName",
                "Labels": ["Computer"],
                "LabelsColumn": "nodeType"
            },
            {
                "Kind": "AddNodes",
                "Query": "graphistryRedTeam50k | project domainName = src_domain,␣
↪nodeType = 'Domain',RED",
                "NodeIdColumn": "domainName",
                "Labels": ["Domain"],
                "LabelsColumn": "nodeType"
            },
            {
                "Kind": "AddNodes",
                "Query": "graphistryRedTeam50k | project domainName = dst_domain,␣
↪nodeType = 'Domain',RED",
                "NodeIdColumn": "domainName",
                "Labels": ["Domain"],
                "LabelsColumn": "nodeType"
            },
            {
                "Kind": "AddEdges",
                "Query": "graphistryRedTeam50k | project event_time, src_computer, dst_
↪computer, src_domain, dst_domain, auth_type, logontype, authentication_orientation,␣
↪success_or_failure, RED",
                "SourceColumn": "src_computer",
                "TargetColumn": "dst_computer",
                "Labels": ["AUTHENTICATES"]
            }
        ]
    }
}```
""")
```

Making the snapshot

A graph snapshot is the actual graph instance materialized from a graph model. It represents:

- A specific point-in-time view of the data as defined by the model
- The nodes, edges, and their properties in a queryable format
- A self-contained entity that persists until explicitly removed

Snapshots are the entities you query when working with persistent graphs. Read more: https://learn.microsoft.com/en-us/kusto/management/graph/graph-persistent-overview?view=microsoft-fabric#graph-snapshots

[7]:
```
# create snapshot name dynamically by adding current timestamp
timestamp = datetime.now().strftime("%m_%d_%Y_%H_%M_%S")

snapshot_name = "InitialSnap_" + timestamp  # append timestamp to always get a unique␣
↪snapshot name for each run
snapshot_name
```

[7]:
```
'InitialSnap_07_07_2025_21_37_03'
```

[ ]:
```
graph_snapshot_query = f".make graph_snapshot {snapshot_name} from {GRAPH_NAME}"

graphistry.kql(graph_snapshot_query)
```

10.8.6.9 Graph Visualization

Once your **data**, **persistent graph** and **snapshot** is created in your Azure Data Explorer cluster it is time to see the power of Graphistry's GPU-accelerated visual interface.

The kusto_graph function accepts two parameters. The name of the graph, and the name of your snapshot **(snap_name="name")**. If you don't provide a snapshot it will grab the latest snapshot.

The function returns a Graphistry plottable object.

You can inspect the nodes and edges, add customizations or .plot() it as is.

[9]:
```
g = graphistry.kusto_graph(GRAPH_NAME, snap_name=snapshot_name)
```

```
Query returned 2 results shapes: [(21984, 5), (50749, 12)] in 2.153 sec
```

Plotting your object

[10]:
```
g.plot()
```

[10]:
```
<IPython.core.display.HTML object>
```

Changing colors, icons and more

Our data consists of two datasets where one contains verified red team activity. In the dataset these are tagged with the value 1 in the column **RED**.

Let's make our red nodes pop out in our visualization. As our data is split into two different type of nodes **"Computer"** and **"Domain"** We also add some icons to make it easier to distinguish the different nodetypes we have.

Learn more here: https://pygraphistry.readthedocs.io/en/latest/notebooks/visualization.html

[11]:
```
g2 = g.encode_point_color(
    "RED",
    categorical_mapping={
        1: "red"
    },
    default_mapping='silver'
```

(continues on next page)

(continued from previous page)

```
)
g3 = g2.encode_point_icon(
    'nodeType',
    shape="circle",
    categorical_mapping={
        "Computer": "laptop",
        "Domain": "server"
    },
    default_mapping="question")

g3.plot()
```

[11]: <IPython.core.display.HTML object>

Next steps

- https://learn.microsoft.com/en-us/kusto/query/graph-semantics-overview?view=microsoft-fabric
- https://pygraphistry.readthedocs.io/en/latest/10min.html
- https://pygraphistry.readthedocs.io/en/latest/gfql/about.html
- https://louie.ai/

Data:

```
A. D. Kent, "Comprehensive, Multi-Source Cybersecurity Events,"
Los Alamos National Laboratory, http://dx.doi.org/10.17021/1179829, 2015.

@Misc{kent-2015-cyberdata1,
  author =     {Alexander D. Kent},
  title =      {{Comprehensive, Multi-Source Cyber-Security Events}},
  year =       {2015},
  howpublished = {Los Alamos National Laboratory},
  doi = {10.17021/1179829}
}
```

10.8.6.10 NodeXL <> Graphistry Converter

Now you can explore the results of your NodeXL data workflows with Graphistry GPU visuals!

- Takes a url to a NodeXL .xls file (you can upload to your Colab session as well) and creates a live Graphistry viz
- Creates a function `graphistry.nodexl('http://my_url/file.xls').plot()`, see calls to it at the bottom
- You can also specify the data source for more bindings, verbose to watch progress, and which Pandas-ready Excel engine: `graphistry.nodexl('http://my_url/file.xls', 'twitter', engine='xlrg', verbose=True)`
- Can upload an XLS file here too, see file upload menu on left (file will be `"./my_file.xls"`)
- Click graphistry logo to start a session or right-click to open in a new window and save its url
- Rerun cells top-to-bottom by hitting shift-enter

Installs, imports, & creds

First run (non-Graphistry distributions)

You may need to restart your notebook's Python runtime after

[ ]:
```
# ! pip install -q --user graphistry pandas
```

[ ]:
```
# Sometimes also need to install a new system Excel parser, and pass in as `graphistry.
↪nodexl(..., engine='openpyxl')
# pip install -q --user openpyxl
```

Imports & credentials

[ ]:
```
import pandas as pd
import graphistry
graphistry.__version__
```

[ ]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Sample use

[ ]:
```
#g = NodeXLGraphistry().xls(xls, 'twitter')
g = graphistry.nodexl('https://nodexlgraphgallery.org/Pages/Workbook.ashx?graphID=220232
↪')
```

[ ]:
```
print('%s nodes, %s edges' % (len(g._nodes), len(g._edges)))
g._nodes.sample(2)
```

[ ]:
```
g.plot()
```

Twitter Demos

Debate Warren

[ ]:
```
graphistry.nodexl('https://nodexlgraphgallery.org/Pages/Workbook.ashx?graphID=220055',
↪'twitter', verbose=True).plot()
```

CES Samsung

```
[ ]: graphistry.nodexl('https://nodexlgraphgallery.org/Pages/Workbook.ashx?graphID=219924',
     ↪'twitter', verbose=True).plot()
```

Larger Graph

```
[ ]: graphistry.nodexl('https://www.nodexlgraphgallery.org/Pages/Workbook.ashx?graphID=220124
     ↪', 'twitter', verbose=True).plot()
```

MediaWiki Demos

Demo 1

```
[ ]: graphistry.nodexl('https://nodexlgraphgallery.org/Pages/Workbook.ashx?graphID=203001',
     ↪'mediawiki').plot()
```

10.8.6.11 Splunk<> Graphistry

Graphistry brings modern visual analytics to event data in Splunk. The full platform is intended for enterprise teams, while this tutorials shares visibility techniques for researchers and hunters.

To use: * Read along, start the prebuilt visualizations by clicking on them * Plug in your Graphistry API Key & Splunk credentials to use for yourself

Further reading: * UI Guide: https://hub.graphistry.com/docs/ui/index/ * Python client tutorials & demos: https://github.com/graphistry/pygraphistry * Graphistry API Key: https://www.graphistry.com/api-request * DoD / VAST challenges: https://www.cs.umd.edu/hcil/varepository/benchmarks.php

0. Configure

```
[ ]:
#splunk
SPLUNK = {
    'host': 'MY.SPLUNK.com',
    'scheme': 'https',
    'port': 8089,
    'username': 'MY_SPLUNK_USER',
    'password': 'MY_SPLUNK_PWD'
}
```

1. Imports

[ ]:
```
import pandas as pd
```

Graphistry

[ ]:
```
!pip install graphistry

import graphistry
graphistry.__version__

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

```
Requirement already satisfied: graphistry in /usr/local/lib/python2.7/dist-packages (0.9.
↪56)
Requirement already satisfied: pandas>=0.17.0 in /usr/local/lib/python2.7/dist-packages␣
↪(from graphistry) (0.22.0)
Requirement already satisfied: numpy in /usr/local/lib/python2.7/dist-packages (from␣
↪graphistry) (1.14.6)
Requirement already satisfied: requests in /usr/local/lib/python2.7/dist-packages (from␣
↪graphistry) (2.18.4)
Requirement already satisfied: future>=0.15.0 in /usr/local/lib/python2.7/dist-packages␣
↪(from graphistry) (0.16.0)
Requirement already satisfied: protobuf>=2.6.0 in /usr/local/lib/python2.7/dist-packages␣
↪(from graphistry) (3.6.1)
Requirement already satisfied: python-dateutil in /usr/local/lib/python2.7/dist-packages␣
↪(from pandas>=0.17.0->graphistry) (2.5.3)
Requirement already satisfied: idna<2.7,>=2.5 in /usr/local/lib/python2.7/dist-packages␣
↪(from requests->graphistry) (2.6)
Requirement already satisfied: urllib3<1.23,>=1.21.1 in /usr/local/lib/python2.7/dist-
↪packages (from requests->graphistry) (1.22)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python2.7/dist-
↪packages (from requests->graphistry) (2018.8.24)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python2.7/dist-
↪packages (from requests->graphistry) (3.0.4)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python2.7/dist-packages (from␣
↪protobuf>=2.6.0->graphistry) (1.11.0)
Requirement already satisfied: setuptools in /usr/local/lib/python2.7/dist-packages␣
↪(from protobuf>=2.6.0->graphistry) (39.1.0)
```

```
u'0.9.56'
```

Splunk

[ ]:
```
# !pip install splunk-sdk

import splunklib
```

[ ]:
```
#Connect to Splunk. Replace settings with your own setup.
import splunklib.client as client
import splunklib.results as results

service = client.connect(**SPLUNK)
```

[ ]:
```
def extend(o, override):
  for k in override.keys():
    o[k] = override[k]
  return o

STEP = 10000;
def splunkToPandas(qry, overrides={}):
    kwargs_blockingsearch = extend({
        "count": 0,
        "earliest_time": "2010-01-24T07:20:38.000-05:00",
        "latest_time": "now",
        "search_mode": "normal",
        "exec_mode": "blocking"
    }, overrides)
    job = service.jobs.create(qry, **kwargs_blockingsearch)

    print "Search results:\n"
    resultCount = job["resultCount"]
    offset = 0;

    print 'results', resultCount
    out = None
    while (offset < int(resultCount)):
        print "fetching:", offset, '-', offset + STEP
        kwargs_paginate = extend(kwargs_blockingsearch,
                                 {"count": STEP,
                                  "offset": offset})

        # Get the search results and display them
        blocksearch_results = job.results(**kwargs_paginate)
        reader = results.ResultsReader(blocksearch_results)
        lst = [x for x in reader]
        df2 = pd.DataFrame(lst)
        out = df2 if type(out) == type(None) else pd.concat([out, df2], ignore_
↪index=True)
        offset += STEP
    return out
```

2. Get data

[ ]:
```
query = 'search index="vast" srcip=* destip=* | rename destip -> dest_ip, srcip -> src_
↪ip | fields dest_ip _time src_ip protocol | eval time=_time | fields - _* '
%time df = splunkToPandas(query, {"sample_ratio": 1000})

#df = splunkToPandasAll('search index="vast" | head 10')
#df = pd.concat([ splunkToPandas('search index="vast" | head 10'), splunkToPandas(
↪'search index="vast" | head 10') ], ignore_index=True)


print 'results', len(df)

df.sample(5)
```

```
Search results:

results 5035
fetching: 0 - 10000
CPU times: user 4.95 s, sys: 13.3 ms, total: 4.96 s
Wall time: 7.92 s
results 5035
```

	dest_ip	src_ip	protocol	time
4324	10.138.235.111	172.30.0.4	TCP	1505519752
2806	10.0.3.5	10.12.15.152	TCP	1505519767
2630	10.0.4.5	10.12.15.152	TCP	1505519769
20	10.0.4.7	10.6.6.7	TCP	1505519795
866	10.0.2.8	10.17.15.10	TCP	1505519787

3. Visualize!

A) Simple IP<>IP: 1326 nodes, 253K edges

[ ]:
```
graphistry.bind(source='src_ip', destination='dest_ip').edges(df).plot()
```

```
<IPython.core.display.HTML object>
```

B) IP<>IP + srcip<>protocol: 1328 nodes, 506K edges

[ ]:
```
def make_edges(df, src, dst):
  out = df.copy()
  out['src'] = df[src]
  out['dst'] = df[dst]
  return out



ip2ip = make_edges(df, 'src_ip', 'dest_ip')
srcip2protocol = make_edges(df, 'src_ip', 'protocol')
```

(continues on next page)

(continued from previous page)

```
combined = pd.concat([ip2ip, srcip2protocol], ignore_index=True)
combined.sample(6)
```

	dest_ip	src_ip	protocol	time	src	dst
6889	10.0.3.5	10.13.77.49	TCP	1505519777	10.13.77.49	TCP
3440	10.0.2.6	10.12.15.152	TCP	1505519761	10.12.15.152	10.0.2.6
6396	10.0.4.5	10.138.235.111	TCP	1505519782	10.138.235.111	TCP
1394	10.0.4.5	10.138.235.111	TCP	1505519782	10.138.235.111	10.0.4.5
5975	10.0.2.7	10.17.15.10	TCP	1505519786	10.17.15.10	TCP
8683	10.0.2.4	10.12.15.152	TCP	1505519759	10.12.15.152	TCP

[ ]:
```
graphistry.bind(source='src', destination='dst').edges(combined).plot()
```

```
<IPython.core.display.HTML object>
```

3. All<>All via Hypergraph: 254K nodes, 760K edges

[ ]:
```
hg = graphistry.hypergraph(df, entity_types=[ 'src_ip', 'dest_ip', 'protocol'] )
print hg.keys()
hg['graph'].plot()
```

```
('# links', 15105)
('# event entities', 5035)
('# attrib entities', 170)
['entities', 'nodes', 'edges', 'events', 'graph']
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```

Node Colors

[ ]:
```
nodes = pd.concat([
    df[['src_ip']].rename(columns={'src_ip': 'id'}).assign(orig_col='src_ip'),
    df[['dest_ip']].rename(columns={'dest_ip': 'id'}).assign(orig_col='dest_ip') ],
    ignore_index=True).drop_duplicates(['id'])

#see https://hub.graphistry.com/docs/api/api-color-palettes/
col2color = {
    "src_ip": 90005,
    "dest_ip": 46005
}

nodes_with_color = nodes.assign(color=nodes.apply(lambda row: col2color[ row['orig_col']␣
↪], axis=1))

nodes_with_color.sample(3)
```

```
              id orig_col  color
4383  172.30.0.3   src_ip  90005
9403   10.0.0.42  dest_ip  46005
4206  172.30.0.4   src_ip  90005
```

[ ]:
```
graphistry.bind(source='src_ip', destination='dest_ip').edges(df).nodes(nodes_with_color).
↪bind(node='id', point_color='color').plot()
```

```
<IPython.core.display.HTML object>
```

[ ]:

10.8.6.12 PyGraphistry <> Titan graph

In this notebook, we demonstrate how to create and modify a Titan graph in python, and then visualize the result using Graphistry's visual graph explorer.

We assume the gremlin server for our Titan graph is hosted locally on port 8182

- This notebook utilizes the python modules aiogremlin and asyncio.
- The GremlinClient class of aiogremlin communicates asynchronously with the gremlin server using websockets via asyncio coroutines.
- This implementation allows you to submit additional requests to the server before any responses are recieved, which is much faster than synchronous request / response cycles.
- For more information about these modules, please visit:
  - aiogremlin: http://aiogremlin.readthedocs.org/en/latest/index.html
  - asyncio: https://pypi.python.org/pypi/asyncio

[1]:
```
import asyncio
import aiogremlin
```

[2]:
```
# Create event loop and initialize gremlin client
loop = asyncio.get_event_loop()
client = aiogremlin.GremlinClient(url='ws://localhost:8182/', loop=loop)  # Default url
```

Functions for graph modification

[3]:
```
@asyncio.coroutine
def add_vertex_routine(name, label):
    yield from client.execute("graph.addVertex(label, l, 'name', n)", bindings={"l":
↪label, "n":name})

def add_vertex(name, label):
    loop.run_until_complete(add_vertex_routine(name, label))

@asyncio.coroutine
```

(continues on next page)

(continued from previous page)

```
def add_relationship_routine(who, relationship, whom):
    yield from client.execute("g.V().has('name', p1).next().addEdge(r, g.V().has('name',␣
↪p2).next())", bindings={"p1":who, "p2":whom, "r":relationship})

def add_relationship(who, relationship, whom):
    loop.run_until_complete(add_relationship_routine(who, relationship, whom))

@asyncio.coroutine
def remove_all_vertices_routine():
    resp = yield from client.submit("g.V()")
    results = []
    while True:
        msg = yield from resp.stream.read();
        if msg is None:
            break
        if msg.data is None:
            break
        for vertex in msg.data:
            yield from client.submit("g.V(" + str(vertex['id']) + ").next().remove()")

def remove_all_vertices():
    results = loop.run_until_complete(remove_all_vertices_routine())

@asyncio.coroutine
def remove_vertex_routine(name):
    return client.execute("g.V().has('name', n).next().remove()", bindings={"n":name})

def remove_vertex(name):
    return loop.run_until_complete(remove_vertex_routine(name));
```

Functions for translating a graph to node and edge lists:

```
- Currently, our API can only upload data from a pandas DataFrame, but we plan to␣
↪implement more flexible uploads in the future.
- For now, we can rely on the following functions to create the necessary DataFrames␣
↪from our graph.
```

[4]:
```
@asyncio.coroutine
def get_node_list_routine():
    resp = yield from client.submit("g.V().as('node')\
                          .label().as('type')\
                          .select('node').values('name').as('name')\
                          .select('name', 'type')")
    results = [];
    while True:
        msg = yield from resp.stream.read();
        if msg is None:
            break;
        if msg.data is None:
            break;
```

(continues on next page)

(continued from previous page)

```
        else:
            results.extend(msg.data)
    return results

def get_node_list():
    results =  loop.run_until_complete(get_node_list_routine())
    return results

@asyncio.coroutine
def get_edge_list_routine():
    resp = yield from client.submit("g.E().as('edge')\
                          .label().as('relationship')\
                          .select('edge').outV().values('name').as('source')\
                          .select('edge').inV().values('name').as('dest')\
                          .select('source', 'relationship', 'dest')")

    results = [];
    while True:
        msg = yield from resp.stream.read();
        if msg is None:
            break;
        if msg.data is None:
            break;
        else:
            results.extend(msg.data)
    return results
def get_edge_list():
    results =  loop.run_until_complete(get_edge_list_routine())
    return results
```

Let's start with an empty graph:

[5]:
```
remove_all_vertices()
```

And then populate it with the Graphistry team members and some of thier relationships:

[6]:
```
add_vertex("Paden", "Person")
add_vertex("Thibaud", "Person")
add_vertex("Leo", "Person")
add_vertex("Matt", "Person")
add_vertex("Brian", "Person")
add_vertex("Quinn", "Person")
add_vertex("Paul", "Person")
add_vertex("Lee", "Person")

add_vertex("San Francisco", "Place")
add_vertex("Oakland", "Place")
add_vertex("Berkeley", "Place")
```

(continues on next page)

(continued from previous page)

```
add_vertex("Turkey", "Thing")
add_vertex("Rocks", "Thing")
add_vertex("Motorcycles", "Thing")

add_relationship("Paden", "lives in", "Oakland")
add_relationship("Quinn", "lives in", "Oakland")
add_relationship("Thibaud", "lives in", "Berkeley")
add_relationship("Matt", "lives in", "Berkeley")
add_relationship("Leo", "lives in", "San Francisco")
add_relationship("Paul", "lives in", "San Francisco")
add_relationship("Brian", "lives in", "Oakland")

add_relationship("Paden", "eats", "Turkey")
add_relationship("Quinn", "cooks", "Turkey")
add_relationship("Thibaud", "climbs", "Rocks")
add_relationship("Matt", "climbs", "Rocks")
add_relationship("Brian", "rides", "Motorcycles")

add_vertex("Graphistry", "Work")

add_relationship("Paden", "works at", "Graphistry")
add_relationship("Thibaud", "works at", "Graphistry")
add_relationship("Matt", "co-founded", "Graphistry")
add_relationship("Leo", "co-founded", "Graphistry")
add_relationship("Paul", "works at", "Graphistry")
add_relationship("Quinn", "works at", "Graphistry")
add_relationship("Brian", "works at", "Graphistry")
```

Now, let's convert our graph database to a pandas DataFrame, so it can be uploaded into our tool:

[7]:
```
import pandas
```

[8]:
```
nodes = pandas.DataFrame(get_node_list())
edges = pandas.DataFrame(get_edge_list())
```

And color the nodes based on their "type" property:

[9]:
```
# Assign different color to each type in a round robin fashion.
# For more information and coloring options please visit: https://graphistry.github.io/
↪docs/legacy/api/0.9.2/api.html
unique_types = list(nodes['type'].unique())
nodes['color'] = nodes['type'].apply(lambda x: unique_types.index(x) % 11)
```

[10]:
```
nodes
```

[10]:

	name	type	color
0	Paden	Person	0

(continues on next page)

(continued from previous page)

```
1           Quinn  Person      0
2          Turkey   Thing      1
3         Thibaud  Person      0
4            Matt  Person      0
5             Leo  Person      0
6        Berkeley   Place      2
7           Rocks   Thing      1
8     Motorcycles   Thing      1
9      Graphistry    Work      3
10            Lee  Person      0
11        Oakland   Place      2
12          Brian  Person      0
13           Paul  Person      0
14  San Francisco   Place      2
```

[11]:
```
edges
```

[11]:
```
             dest relationship   source
0         Oakland     lives in    Paden
1          Turkey         eats    Paden
2      Graphistry     works at    Paden
3         Oakland     lives in    Quinn
4          Turkey        cooks    Quinn
5      Graphistry     works at    Quinn
6        Berkeley     lives in  Thibaud
7           Rocks       climbs  Thibaud
8      Graphistry     works at  Thibaud
9        Berkeley     lives in     Matt
10          Rocks       climbs     Matt
11     Graphistry   co-founded     Matt
12  San Francisco     lives in      Leo
13     Graphistry   co-founded      Leo
14        Oakland     lives in    Brian
15    Motorcycles        rides    Brian
16     Graphistry     works at    Brian
17  San Francisco     lives in     Paul
18     Graphistry     works at     Paul
```

Finally, let's vizualize the results!

[12]:
```
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

g = graphistry.bind(source="source", destination="dest", node='name', point_color='color
↪', edge_title='relationship')
g.plot(edges, nodes)
```

[12]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
```

10.8.6.13 Tutorial: Visualizing the Silk Road Blockchain with Graphistry and Neo4j

Investigating large datasets becomes easier by directly visualizing cypher (BOLT) query results with Graphistry. This tutorial walks through querying Neo4j, visualizing the results, and additional configurations and queries.

This analysis is based on a blockchain data extract the Graphistry team performed around court proceedings from when **Carl Force**, the key DEA agent in the Silk Road bust, was sentenced for embezzling money from **Ross Ulbricht** (Dread Pirate Roberts). We explore to how to recreate the analysis, and determine where Carl's money went after he performed the initial embezzling.

Instructions * Read along the various cells * Click the prebuilt visualizations to start them, and interact with them just like Google Maps * To try on your own, setup your own Neo4j instance & get a Graphistry API key, and run the data loading cells

Further reading

- UI Guide: https://hub.graphistry.com/docs/ui/index/
- Python client tutorials & demos: https://github.com/graphistry/pygraphistry
- Graphistry API Key: https://www.graphistry.com/api-request
- Neo4j-as-a-service: http://graphstory.com (external link)
- DEA incident: https://arstechnica.com/tech-policy/2016/08/stealing-bitcoins-with-badges-how-silk-roads-dirty-cops-got-caught/

Config

Install dependencies

- On first run of a non-Graphistry notebook server:
  1. Uncomment and run the first two lines
  2. Restart your Python kernel runtime from the top menue
- For advanced alternate installs, see subsequent commented lines

[ ]:
```
#!pip install --user pandas
#!pip install --user graphistry[bolt]

### ADVANCED:
### If you already have the neo4j python driver, you can leave out '[bolt]':
### !pip install --user graphistry
### If you already have graphistry but not neo4j, you can reuse your existing graphistry:
### !pip install --user neo4j
```

Import & test

[ ]:
```
import pandas as pd
import neo4j # just for testing
from neo4j import GraphDatabase # for data loader
import graphistry
print('neo4j', neo4j.__version__)
print('graphistry', graphistry.__version__)
```

Connect

- You may need to reconnect if your Neo4j connection closes
- Uncomment the below section for non-Graphistry notebook servers

[ ]:
```
NEO4J = {
    'uri': "bolt://my.site.COM:7687",
    'auth': ("neo4j", "myalphapwd1")
}

graphistry.register(bolt=NEO4J)

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Optional: Load tainted transactions into your own Neo4j DB

To populate your own Neo4j instance, set one or both of the top commands to True

[ ]:
```
DELETE_EXISTING_DATABASE=False
POPULATE_DATABASE=False

if DELETE_EXISTING_DATABASE:
    driver = GraphDatabase.driver(**NEO4J)
    with driver.session() as session:
        # split into 2 transancations case of memory limit errors
        print('Deleting existing transactions')
        tx = session.begin_transaction()
        tx.run("""MATCH (a:Account)-[r]->(b) DELETE r""")
        tx.commit()
        print('Deleting existing accounts')
        tx = session.begin_transaction()
        tx.run("""MATCH (a:Account) DELETE a""")
        tx.commit()
        print('Delete successful')

if POPULATE_DATABASE:
    edges = pd.read_csv('https://www.dropbox.com/s/q1daa707y99ind9/edges.csv?dl=1')
    edges = edges.rename(columns={'Amount $': "USD", 'Transaction ID': 'Transaction'})[[
↪'USD', 'Date', 'Source', 'Destination', 'Transaction']]
    id_len = len(edges['Source'][0].split('...')[0]) #truncate IDs (dirty data)
```

(continues on next page)

(continued from previous page)

```
    edges = edges.assign(
    Source=edges['Source'].apply(lambda id: id[:id_len]),
    Destination=edges['Destination'].apply(lambda id: id[:id_len]))
    ROSS_FULL='2a37b3bdca935152335c2097e5da367db24209cc'
    ROSS = ROSS_FULL[:32]
    CARL_FULL = 'b2233dd22ade4c9978ec1fd1fbb36eb7f9b4609e'
    CARL = CARL_FULL[:32]
    CARL_NICK = 'Carl Force (DEA)'
    ROSS_NICK = 'Ross Ulbricht (SilkRoad)'
    nodes = pd.read_csv('https://www.dropbox.com/s/nf796f1asow8tx7/nodes.csv?dl=1')
    nodes = nodes.rename(columns={'Balance $': 'USD', 'Balance (avg) $': 'USD_avg',
↪'Balance (max) $': 'USD_max', 'Tainted Coins': 'Tainted_Coins'})[['Account', 'USD',
↪'USD_avg', 'USD_max', 'Tainted_Coins']]
    nodes['Account'] = nodes['Account'].apply(lambda id: id[:id_len])
    nodes['Account'] = nodes['Account'].apply(lambda id: CARL_NICK if id == CARL else␣
↪ROSS_NICK if id == ROSS else id)
    driver = GraphDatabase.driver(**NEO4J)
    with driver.session() as session:
        tx = session.begin_transaction()
        print('Loading', len(nodes), 'accounts')
        for index, row in nodes.iterrows():
            if index % 2000 == 0:
                print('Committing', index - 2000, '...', index)
                tx.commit()
                tx = session.begin_transaction()
            tx.run("""
            CREATE (a:Account {
              Account: $Account,
              USD: $USD, USD_avg: $USD_avg, USD_max: $USD_max, Tainted_Coins: $Tainted_
↪Coins
            })
            RETURN id(a)
            """, **row)
            if index % 2000 == 0:
                print(index)
        print('Committing rest')
        tx.commit()
        tx = session.begin_transaction()
        print('Creating index on Account')
        tx.run("""  CREATE INDEX ON :Account(Account)  """)
        tx.commit()
    STATUS=1000
    BATCH=2000
    driver = GraphDatabase.driver(**NEO4J)

    with driver.session() as session:
        tx = session.begin_transaction()
        print('Loading', len(edges), 'transactions')
        for index, row in edges.iterrows():
            tx.run("""MATCH (a:Account),(b:Account)
                  WHERE a.Account = $Source AND b.Account = $Destination
                  CREATE (a)-[r:PAYMENT {
```

(continues on next page)

(continued from previous page)

```
                    Source: $Source, Destination: $Destination, USD: $USD, Date: $Date,␣
↪Transaction: $Transaction
                  }]->(b)
                  """, **row)
            if index % STATUS == 0:
                print(index)
            if index % BATCH == 0 and index > 0:
                print('sending batch out')
                tx.commit()
                print('... done')
                tx = session.begin_transaction()
        tx.commit()
```

Cypher Demos

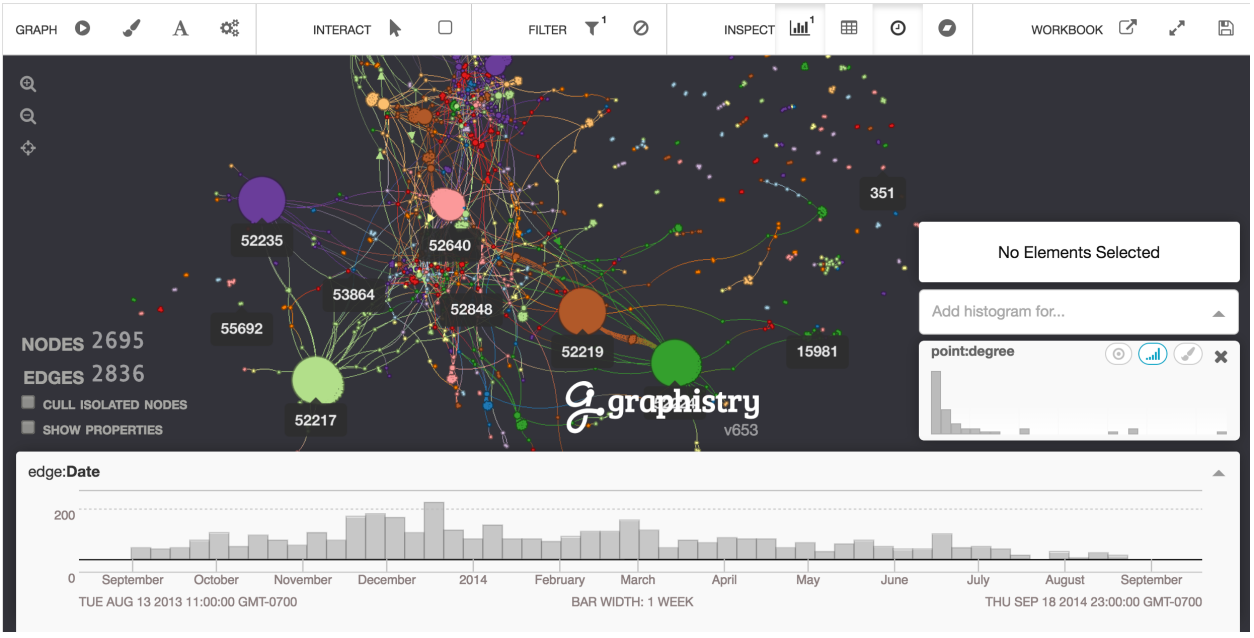
1a. Warmup: Visualize all $7K - $10K transactions

Try panning and zooming (same touchpad/mouse controls as Google Maps), and clicking on individual wallets and transactions.

[ ]:
```
g = graphistry.cypher("""
      MATCH (a)-[r:PAYMENT]->(b) WHERE r.USD > 7000 AND r.USD < 10000  RETURN a, r, b␣
↪ORDER BY r.USD DESC
  """)
```

[ ]:
```
g.plot()
```

Screenshot



1b. Cleanup: Configure node and edge titles to use amount fields

- **Static config**: We can preconfigure the visualization from directly within the notebook
- **Dynamic config**: Try dynamically improving the visualization on-the-fly within the tool by
  - Do `add histogram for...` on `edge:USD` and `point:USD_MAX`
  - Set edge/point coloring using them, and selecting a "Gradient (Spectral7 7)" blend, and toggling to reverse order (so cold to hot).
  - For `point:USD_MAX`, toggle it to controling point size, and in the `Scene settings`, increase the point size slider

[ ]:
```
g = g\
  .bind(point_title='Account')\
  .bind(edge_title='USD')

g.plot()
```

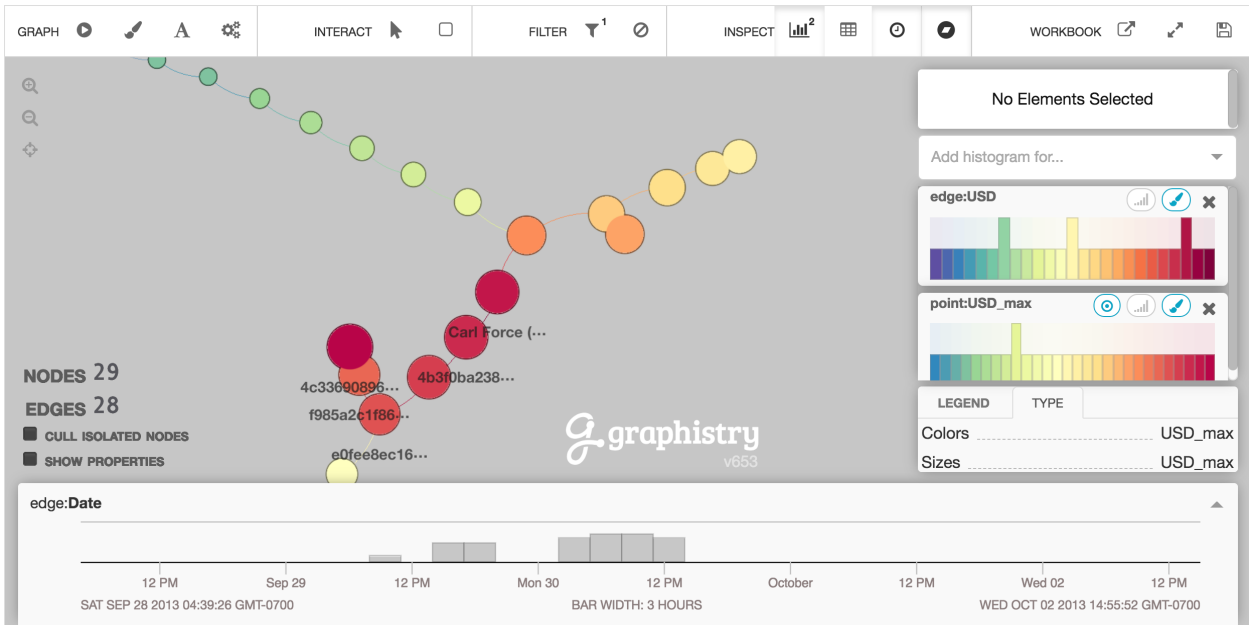
2. Look for all transactions 1-5 hops from embezzling DEA Agent Carl Force

2a. Downstream

Where did most of Carl's money go? * Try setting up filters on `edge:USD` to separate out small vs big money flows.

[ ]:
```
g.cypher("""
    match (a)-[r:PAYMENT*1..20]->(b)
    where a.Account = $root and ALL(transfer IN r WHERE transfer.USD > $min_amount and␣
↪transfer.USD < $max_amount )
    return a, r, b
  """,
  {'root': "Carl Force (DEA)",
   'min_amount': 999,
   'max_amount': 99999}).plot()
```

Screenshot:

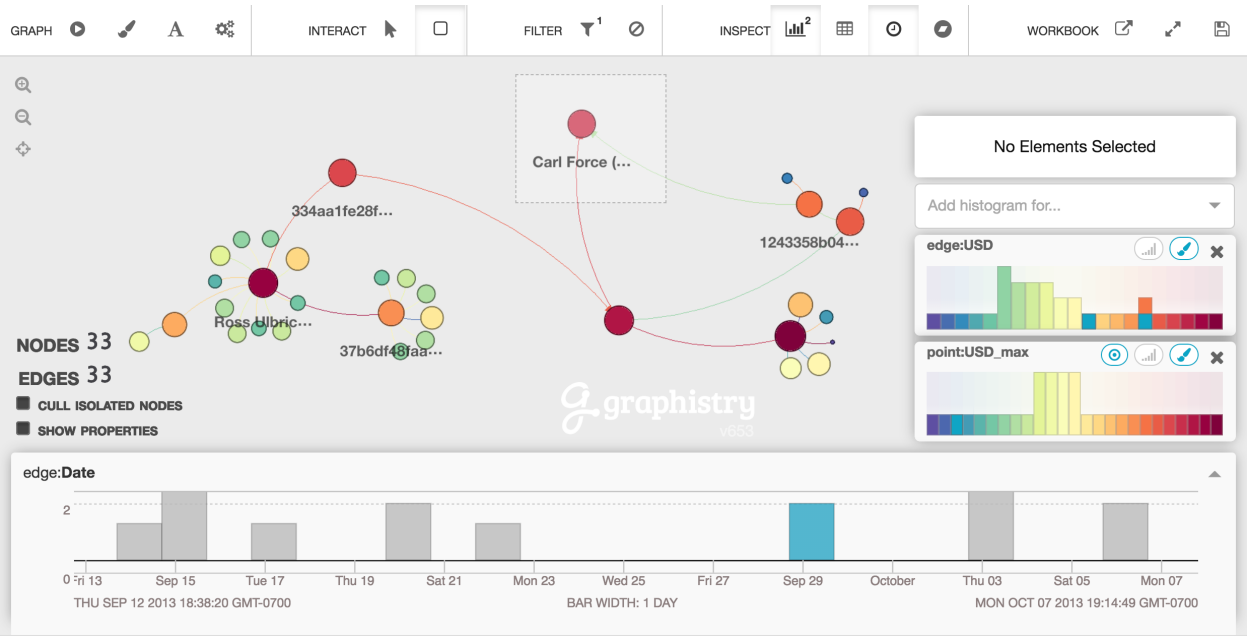


2b. Upstream

From where did Carl get most of his money?

[ ]:
```
g.cypher("""
      match (a)-[r:PAYMENT*1..10]->(b)
      where b.Account=$sink and ALL(transfer IN r WHERE transfer.USD > $min_amount and
↪transfer.USD < $max_amount )
      return r, a, b
    """,
    {'sink': "Carl Force (DEA)",
    'min_amount': 1999,
    'max_amount': 99999}).plot()
```

Screenshot:



3. Paths between Silk Road and Carl Force

```
[ ]: g.cypher("match (a)-[r:PAYMENT*1..10]->(b) where a.Account=$silk and b.Account=$dea␣
     ↪return r, a, b",
             {'dea': "Carl Force (DEA)", "silk": "Ross Ulbricht (SilkRoad)"}).plot()
```

Further Reading

- UI Guide: https://hub.graphistry.com/docs/ui/index/
- Python client tutorials & demos: https://github.com/graphistry/pygraphistry
- DEA incident: https://arstechnica.com/tech-policy/2016/08/stealing-bitcoins-with-badges-how-silk-roads-dirty-cops-got-caught/

10.8.6.14 Neo4j Twitter Trolls Tutorial

Goal: This notebook aims to show how to use PyGraphistry to visualize data from https://neo4j.com/developer/. We also show how to use https://neo4j.com/developer/graph-algorithms/ and use PyGraphistry to visualize the result of those algorithms.

Prerequesties: * You'll need a Graphistry API key, which you can request https://www.graphistry.com/api-request * Neo4j. We'll be using https://neo4j.com/sandbox-v2/ (free hosted Neo4j instances pre-populated with data) for this tutorial. Specifically the "Russian Twitter Trolls" sandbox. You can create a Neo4j Sandbox instance https://neo4j.com/sandbox-v2/ * Python requirements: * `neo4j-driver <https://github.com/neo4j/neo4j-python-driver>`___ - `pip install neo4j-driver` * `pygraphistry <https://github.com/graphistry/pygraphistry/>`___ - `pip install "graphistry[all]"`

Outline

- Connecting to Neo4j
  - using neo4j-driver Python client
  - query with Cypher
- Visualizing data in Graphistry from Neo4j
  - User-User mentions from Twitter data
- Graph algorithms
  - Enhancing our visualization with PageRank

[1]:
```
# import required dependencies
from neo4j.v1 import GraphDatabase, basic_auth
from pandas import DataFrame
import graphistry
```

[2]:
```
# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Connect To Neo4j

If you haven't already, create an instance of the Russian Twitter Trolls sandbox on https://neo4j.com/sandbox-v2/ We'll use the https://github.com/neo4j/neo4j-python-driver to fetch data from Neo4j. To do this we'll need to instantiate a `Driver` object, passing in the credentials for our Neo4j instance. If using Neo4j Sandbox you can find the credentials for your Neo4j instance in the "Details" tab. Specifically we need the IP address, bolt port, username, and password. Bolt is the binary protocol used by the Neo4j drivers so a typical database URL string takes the form `bolt://<IP_ADDRESS>:<BOLT_PORT>`

[3]:
```
# instantiate Neo4j driver instance
# be sure to replace the connection string and password with your own
driver = GraphDatabase.driver("bolt://34.201.165.36:34532", auth=basic_auth("neo4j",
↪"capitals-quality-loads"))
```

Once we've instantiated our Driver, we can use `Session` objects to execute queries against Neo4j. Here we'll use `session.run()` to execute a https://neo4j.com/developer/cypher-query-language/. Cypher is the query language for graphs that we use with Neo4j (you can think of Cypher as SQL for graphs).

[4]:
```
# neo4j-driver hello world
# execute a simple query to count the number of nodes in the database and print the␣
↪result
with driver.session() as session:
    results = session.run("MATCH (a) RETURN COUNT(a) AS num")
for record in results:
    print(record)
```

```
<Record num=281217>
```

If we inspect the datamodel in Neo4j we can see that we have inormation about Tweets and specifically Users mentioned in tweets.

Let's use Graphistry to visualize User-User Tweet mention interactions. We'll do this by querying Neo4j for all tweets that mention users.

Using Graphistry With Neo4j

Currently, PyGraphistry can work with data as a pandas DataFrame, NetworkX graph or IGraph graph object. In this section we'll show how to load data from Neo4j into PyGraphistry by converting results from the Python Neo4j driver into a pandas DataFrame.

Our goal is to visualize User-User Tweet mention interactions. We'll create two pandas DataFrames, one representing our nodes (Users) and a second representing the relationships in our graph (mentions).

Some users are known Troll accounts so we include a flag variable, `troll` to indicate when the user is a Troll. This will be used in our visualization to set the color of the known Troll accounts.

[7]:
```
# Create User DataFrame by querying Neo4j, converting the results into a pandas DataFrame
with driver.session() as session:
    results = session.run("""
    MATCH (u:User)
    WITH u.user_key AS screen_name, CASE WHEN "Troll" IN labels(u) THEN 5 ELSE 0 END AS␣
↪troll
    RETURN screen_name, troll""")
    users = DataFrame(results.data())
# show the first 5 rows of the DataFrame
users[:5]
```

[7]:

	screen_name	troll
0	robbydelaware	5
1	scottgohard	5
2	beckster319	5
3	skatewake1994	5
4	kadirovrussia	5

Next, we need some relationships to visualize. In this case we are interested in visualizing user interactions, specifically where users have mentioned users in Tweets.

[8]:
```
# Query for tweets mentioning a user and create a DataFrame adjacency list using screen_
↪name
# where u1 posted a tweet(s) that mentions u2
# num is the number of time u1 mentioned u2 in the dataset
with driver.session() as session:
    results = session.run("""
        MATCH (u1:User)-[:POSTED]->(:Tweet)-[:MENTIONS]->(u2:User)
        RETURN u1.user_key AS u1, u2.user_key AS u2, COUNT(*) AS num
    """)
    mentions  = DataFrame(results.data())
mentions[:5]
```

[8]:

	num	u1	u2
0	1	dorothiebell	dwstweets
1	1	happkendrahappy	nineworthies
2	2	aiden7757	theclobra
3	1	ameliebaldwin	dcclothesline
4	9	ameliebaldwin	jturnershow

Now we can visualize this mentions network using Graphistry. We'll specify the nodes and relationships for our graph. We'll also use the `troll` property to color the known Troll nodes red, setting them apart from other users in the graph.

[9]:
```
viz = graphistry.bind(source="u1", destination="u2", node="screen_name", point_color=
↪"troll").nodes(users).edges(mentions)
viz.plot()
```

[9]:
```
<IPython.core.display.HTML object>
```

After running the above Python cell you should see an interactive Graphistry visualization like this:

Known Troll user nodes are colored red, regular users colored blue. By default, the size of the nodes is proportional to the degree of the node (number of relationships). We'll see in the next section how we can use graph algorithms such as PageRank and visualize the results of those algorithms in Graphistry.

Graph Algorithms

The above visualization shows us User-User Tweet mention interactions from the data. What if we wanted to answer the question "Who is the most important user in this network?". One way to answer that would be to look at the degree, or number of relationships, of each node. By default, PyGraphistry uses degree to style the size of the node, allowing us to determine importance of nodes at a glance.

We can also use https://github.com/neo4j-contrib/neo4j-graph-algorithms such as PageRank to determine importance in the network. In this section we show how to https://neo4j.com/developer/graph-algorithms/ and use the results of these algorithms in our Graphistry visualization.

[10]:
```
# run PageRank on the projected mentions graph and update nodes by adding a pagerank␣
↪property score
with driver.session() as session:
    session.run("""
        CALL algo.pageRank("MATCH (t:User) RETURN id(t) AS id",
         "MATCH (u1:User)-[:POSTED]->(:Tweet)-[:MENTIONS]->(u2:User)
         RETURN id(u1) as source, id(u2) as target", {graph:'cypher', write: true})
     """)
```

Now that we've calculated PageRank for each User node we need to create a new pandas DataFrame for our user nodes by querying Neo4j:

[11]:
```
# create a new users DataFrame, now including PageRank score for each user
with driver.session() as session:
    results = session.run("""
    MATCH (u:User)
    WITH u.user_key AS screen_name, u.pagerank AS pagerank, CASE WHEN "Troll" IN␣
↪labels(u) THEN 5 ELSE 0 END AS troll
    RETURN screen_name, pagerank, troll""")
    users = DataFrame(results.data())
users[:5]
```

[11]:

	pagerank	screen_name	troll
0	0.150000	robbydelaware	5
1	0.151547	scottgohard	5
2	0.150000	beckster319	5
3	0.150000	skatewake1994	5
4	0.150000	kadirovrussia	5

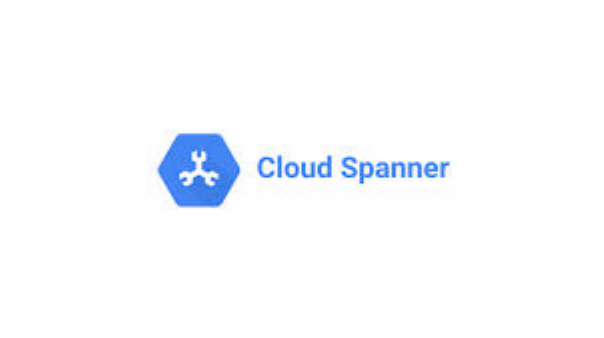
[12]:
```
# render the Graphistry visualization, binding node size to PageRank score
viz = graphistry.bind(source="u1", destination="u2", node="screen_name", point_size=
↪"pagerank", point_color="troll").nodes(users).edges(mentions)
viz.plot()
```

[12]:
```
<IPython.core.display.HTML object>
```

Now when we render the Graphistry visualization, node size is proprtional to the node's PageRank score. This results in a different set of nodes that are identified as most important.

By binding node size to the results of graph algorithms we are able to draw insight from the data at a glance and further explore the interactive visualization.

[ ]:



10.8.6.15 Demo Notebook - Graphistry and Google Spanner Graph

Graphistry is a cutting-edge platform for large-scale visual graph exploration and analysis. It enables users to intuitively investigate complex relationships, patterns, and anomalies across vast datasets through highly interactive, GPU-accelerated visualizations. Google Cloud Spanner, on the other hand, is a globally distributed, horizontally scalable, and strongly consistent database ideal for managing large, interconnected datasets.

This interactive guide demonstrates how to combine the power of Graphistry's visual graph analytics and AI with the robust data capabilities of Google Cloud Spanner Graph.

Together, these technologies empower you to:

- **Visualize Complex Graphs:** Easily explore relationships and uncover insights in your data through rich visual representations.
- **Handle Large Datasets:** Leverage Cloud Spanner's ability to manage vast amounts of interconnected information with strong consistency and scalability.
- **Perform Advanced Analytics:** Apply graph-based algorithms and clustering techniques to extract actionable insights from structured data.

This demo is designed for:

- **Data Scientists:** Interested in adding visual graph analytics to their toolkit.
- **Database Engineers:** Looking to integrate graph capabilities into their Cloud Spanner workflows.

- **Application Developers:** Prototyping applications built using Graphistry and Google Spanner.

This notebook showcases:

1. **Connecting to Cloud Spanner:** How to retrieve and preprocess data from Cloud Spanner for graph processing.
2. **Graph Visualization with Graphistry:** Turning raw data into meaningful visualizations to explore relationships and clusters.
3. **Real-World Use Cases:** Applying these tools to solve practical problems.

Prerequisites

- A Google Cloud account with access to Cloud Spanner.
- A Graphistry Enterprise Server or free-tier https://www.graphistry.com/get-started.
- Python environment with Graphistry and gcloud spanner support (see pip install below).
- This demo is based on https://codelabs.developers.google.com/codelabs/spanner-graph-getting-started#0.

Let's Get Started!

Dive in and see how the synergy of Graphistry and Google Cloud Spanner can transform your data exploration and analysis workflows.

[2]:
```
import os
import graphistry
graphistry.__version__
```

[2]:
```
'0.38.2+2.g6f1ae473.dirty'
```

Graphistry register and gcloud init

[3]:
```
# graphistry register

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options, see https://pygraphistry.readthedocs.io/en/latest/server/register.
↪html

import os

graphistry.register(api=3,
                    protocol  = "https",
                    server    = os.getenv("GRAPHISTRY_SERVER"),
                    username  = os.getenv("GRAPHISTRY_USERNAME"),
                    password  = os.getenv("GRAPHISTRY_PASSWORD"))
```

[3]:
```
<graphistry.pygraphistry.GraphistryClient at 0x16989a900>
```

Configure Spanner

[ ]:
```
SPANNER_INSTANCE_ID = os.getenv("SPANNER_INSTANCE_ID") or "finance-graph-instance"
SPANNER_DATABASE_ID = os.getenv("SPANNER_DATABASE_ID") or "finance-graph-db"

# Option 1: Project ID is required for interactive login
SPANNER_PROJECT_ID = os.getenv("SPANNER_PROJECT_ID") or "finance-graph-project"

# Option 2: use a service account key:
# SPANNER_SERVICE_ACCOUNT_JSON_PATH = os.getenv("SPANNER_SERVICE_ACCOUNT_JSON_PATH") or
↪"path/to/credentials.json"
```

[ ]:
```
# Set the google project id for interactive login

# !gcloud config set project {SPANNER_PROJECT_ID}
# %env GOOGLE_CLOUD_PROJECT={SPANNER_PROJECT_ID}

#!gcloud auth application-default login
```

[5]:
```
# Option 1: interactive login using gcloud auth application-default login (below)
graphistry.configure_spanner(
    project_id=SPANNER_PROJECT_ID,
    instance_id=SPANNER_INSTANCE_ID,
    database_id=SPANNER_DATABASE_ID
)


# Option 2: use a service account key:
# graphistry.configure_spanner(
#     instance_id=SPANNER_INSTANCE_ID,
#     database_id=SPANNER_DATABASE_ID,
#     credentials_file=SPANNER_SERVICE_ACCOUNT_JSON_PATH
# )



# optional setting to limit the number of records returned
LIMIT_CLAUSE = ""
# or use:
# LIMIT_CLAUSE = "limit 1000"
```

Example 1: GQL Path Query to Graphistry Visualization of all nodes and edges (LIMIT optional)

to extract the data from Spanner Graph as a graph with nodes and edges in a single object, a GQL path query is required.

The format of a path query is as follows, note the p= at the start of the MATCH clause, and the SAFE_TO_JSON(p) without these, the query will not produce the results needed to properly load a graphistry graph. LIMIT is optional, but for large graphs with millions of edges or more, it's best to filter either in the query or use LIMIT so as not to exhaust GPU memory.

```
GRAPH FinGraph
MATCH p = (a)-[b]->(c) where 1=1 LIMIT 10000 return SAFE_TO_JSON(p) as path
```

[6]:
```
query=f'''GRAPH FinGraph
MATCH p = (a)-[b]->(c) where 1=1 {LIMIT_CLAUSE} return SAFE_TO_JSON(p) as path'''

g = graphistry.spanner_gql(query)
```

[7]:
```
g.plot()
```

[7]:
```
<IPython.core.display.HTML object>
```

Example 1.1 - inspect contents of graphistry graph (nodes and edges):

[8]:
```
len(g._nodes), len(g._edges)
```

[8]:
```
(1312, 1700)
```

[9]:
```
g._nodes.head(3)
```

[9]:
```
     label                    identifier               create_time   id  \
0  Account  mUZpbkdyYXBoLkFjY291bnQAeJEC  2020-01-10T14:22:20.222Z    1
1     Loan  mUZpbkdyYXBoLkxvYW4AeJIBMA==  2021-04-12T02:01:54.223Z  152
4  Account  mUZpbkdyYXBoLkFjY291bnQAeJEK  2020-03-02T20:47:18.726Z    5

  is_blocked              type_   balance  interest_rate  loan_amount name  \
0      False  brokerage account       NaN            NaN          NaN  NaN
1        NaN                NaN  112149.8          0.035    2492216.8  NaN
4      False  brokerage account       NaN            NaN          NaN  NaN

      type
0  Account
1     Loan
4  Account
```

[10]:
```
g._edges.head(3)
```

[10]:
```
    label                                        identifier  \
0  Repays  mUZpbkdyYXBoLkFjY291bnRSZXBheUxvYW4AeJECkgEwmY...
1  Repays  mUZpbkdyYXBoLkFjY291bnRSZXBheUxvYW4AeJECkgEwmY...
2  Repays  mUZpbkdyYXBoLkFjY291bnRSZXBheUxvYW4AeJEKkSyZgA...

                        source                    destination   amount  \
0  mUZpbkdyYXBoLkFjY291bnQAeJEC  mUZpbkdyYXBoLkxvYW4AeJIBMA==  36633.6
1  mUZpbkdyYXBoLkFjY291bnQAeJEC  mUZpbkdyYXBoLkxvYW4AeJIBMA==  22836.6
2  mUZpbkdyYXBoLkFjY291bnQAeJEK      mUZpbkdyYXBoLkxvYW4AeJEs  15999.3

               create_time  id  loan_id  to_id  account_id    type
0  2022-03-08T05:27:06.265Z   1    152.0    NaN         NaN  Repays
1  2022-05-02T12:40:52.562Z   1    152.0    NaN         NaN  Repays
2  2022-03-14T21:28:25.972Z   5     22.0    NaN         NaN  Repays
```

Example 2: Quantified path traversal

(slightly modified from example to use a path query for visualization)

from: https://codelabs.developers.google.com/codelabs/spanner-graph-getting-started#6

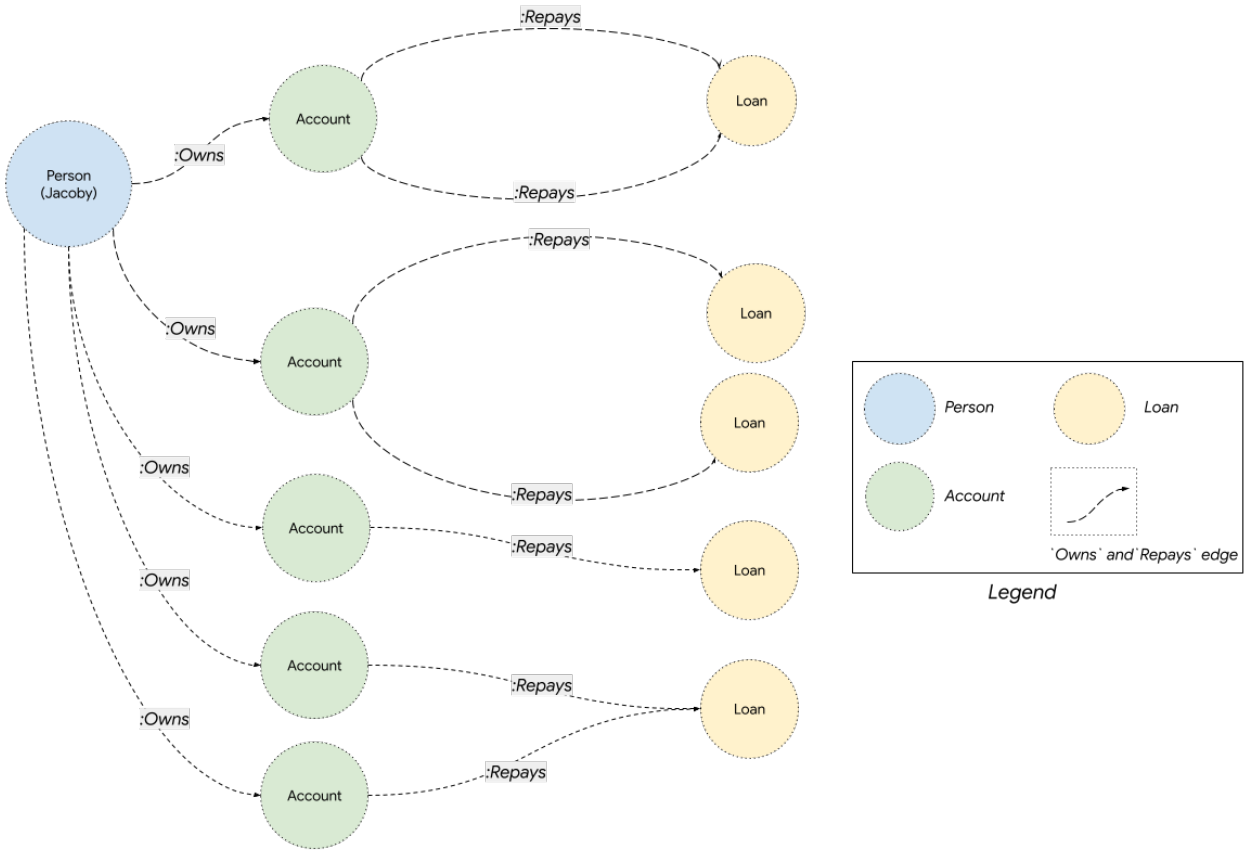
Query 2 - Quantified path traversal and return graph elements

The following query matches all account money transfers starting from a source account with id=75 within 3 to 6 hops, to reach a destination account with id=199. The {3,6} syntax is used to represent a quantified 3 to 6 hop path traversal between src_accnt and dst_accnt.

```
GRAPH FinGraph
MATCH
p = (src_accnt:Account {id:75})-[transfers:Transfers]->{3,6}
  (dst_accnt:Account {id:199})
RETURN SAFE_TO_JSON(p) as path
```

Visually, you can think of the quantified edge traversal like below: it starts from a src_account node, and fetches all possible account transfer paths between 3 to 6 hops, to reach dst_account.

The highlighted path at the bottom below, for example, is a 6-hop query.



source: https://codelabs.developers.google.com/static/codelabs/spanner-graph-getting-started/#6

[11]:
```
query2='''GRAPH FinGraph
MATCH
p = (src_accnt:Account {id:75})-[transfers:Transfers]->{3,6}
  (dst_accnt:Account {id:199}) where 1=1
RETURN SAFE_TO_JSON(p) as path
'''
```

[12]:
```
g2 = graphistry.spanner_gql(query2)
g2.plot()
```

[12]:
```
<IPython.core.display.HTML object>
```

[13]:
```
#  now run again and retrive all the paths
query2a='''GRAPH FinGraph
MATCH
p = (src_accnt:Account )-[transfers:Transfers]->{3,6}
  (dst_accnt:Account ) where 1=1
RETURN SAFE_TO_JSON(p) as path
'''
g2a = graphistry.spanner_gql(query2a)
g2a.plot()
```

[13]:
```
<IPython.core.display.HTML object>
```

Example 3: Spanner GQL Tabular Query to pandas dataframe (LIMIT optional)

This example shows a non-path query that returns tabular results, which are then convered to a dataframe for easy manipulation and inspection of the results.

`GRAPH FinGraph MATCH (p:Person)-[]-()->(l:Loan) RETURN p.id as ID, p.name AS Name, SUM(l.loan_amount) AS TotalBorrowed ORDER BY TotalBorrowed DESC LIMIT 10`

[14]:
```
query_top10='''GRAPH FinGraph
MATCH (p:Person)-[:Owns]-(:Account)->(l:Loan) WHERE 1=1
RETURN p.id as ID, p.name AS Name, SUM(l.loan_amount) AS TotalBorrowed
ORDER BY TotalBorrowed DESC
LIMIT 10'''
```

[15]:
```
Top10_Borrowers_df = graphistry.spanner_gql_to_df(query_top10)
```

[16]:
```
Top10_Borrowers_df.head(10)
```

[16]:

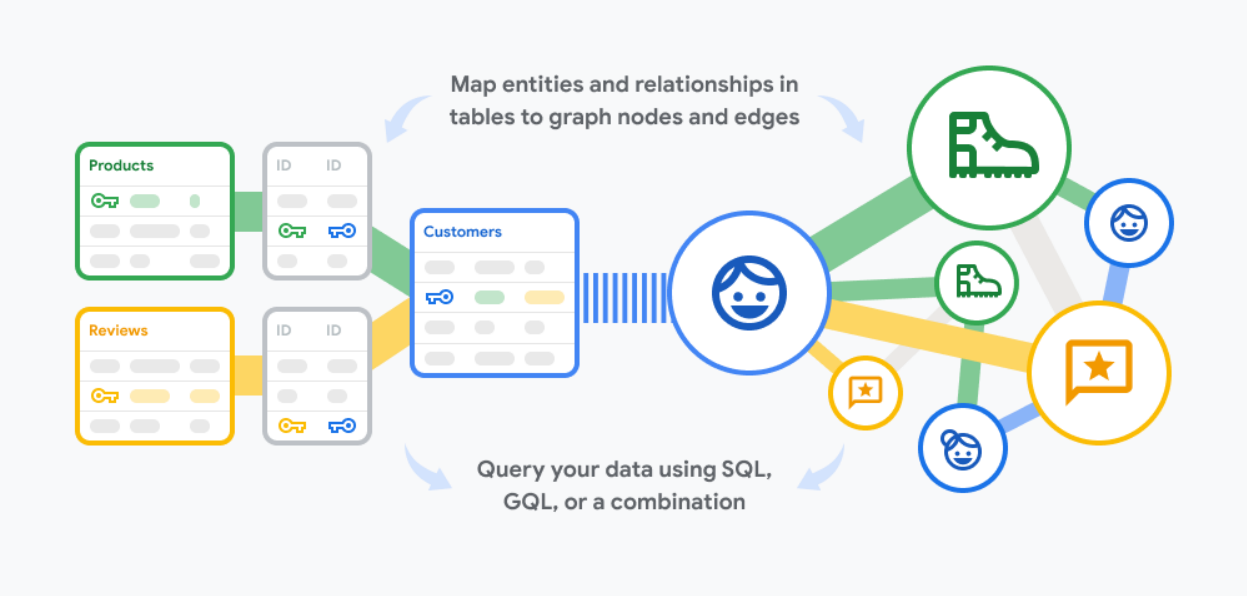
	ID	Name	TotalBorrowed
0	337	Tutmarc	15269003.6
1	484	Greiner	14098853.6
2	370	Morrisseau	13912146.2
3	113	Paakkonen	13022928.4
4	416	Greif	12713990.0
5	68	Cabon	12256398.8
6	66	Stinson	11462716.8
7	46	Riby	10772732.5
8	406	Jöhncke	10470230.8
9	169	Gubenko	10330528.5

Example 4: Spanner SQL Query to pandas dataframe

This example shows a SQL query to Spanner that returns tabular results, which are then convered to a dataframe for easy manipulation and inspection of the results.

Query:

```
SELECT * from Account
```



source: https://cloud.google.com/blog/products/databases/announcing-spanner-graph

[17]:
```
accounts_df =  graphistry.spanner_gql_to_df('SELECT * from Account')
```

[18]:
```
accounts_df.head(10)
```

[18]:

	id	create_time	is_blocked	type
0	1	2020-01-10 14:22:20.222000+00:00	False	brokerage account
1	2	2020-01-28 01:55:09.206000+00:00	False	prepaid card
2	3	2020-02-18 13:44:20.655000+00:00	False	brokerage account
3	4	2020-02-29 16:49:53.902000+00:00	False	debit card
4	5	2020-03-02 20:47:18.726000+00:00	False	brokerage account
5	6	2020-03-21 22:25:34.327000+00:00	False	custodial account
6	7	2020-04-14 00:53:48.932000+00:00	False	brokerage account
7	8	2020-04-15 03:08:15.427000+00:00	True	trust account
8	9	2020-04-20 13:20:25.717000+00:00	False	certificate of deposit
9	10	2020-04-26 00:12:17.773000+00:00	False	debit card

Example 5: Spanner SQL Query to inspect the database schema

This example shows a SQL query to Spanner that retrieves the tables, columns and types from the information schema in Spanner. This can be helpful for seeing what's available in the database or using this data as part of a workflow.

```
SELECT table_name, column_name, spanner_type FROM information_schema.columns
```

[19]:
```
columns_df =  graphistry.spanner_gql_to_df('SELECT table_name, column_name, spanner_type␣
↪FROM information_schema.columns')
columns_df.head(10)
```

[19]:

	table_name	column_name	spanner_type
0	Account	id	INT64
1	Account	create_time	TIMESTAMP
2	Account	is_blocked	BOOL
3	Account	type	STRING(MAX)
4	AccountAudits	id	INT64
5	AccountAudits	audit_timestamp	TIMESTAMP
6	AccountAudits	audit_details	STRING(MAX)
7	AccountRepayLoan	id	INT64
8	AccountRepayLoan	loan_id	INT64
9	AccountRepayLoan	amount	FLOAT64

[20]:
```
len(columns_df.table_name.unique())
```

[20]:
```
100
```

[21]:
```
query_tables='''
SELECT table_name, table_type
FROM information_schema.tables
WHERE table_catalog = ''
  AND table_schema = ''
  AND table_type IN ('BASE TABLE', 'VIEW');
'''

tables_df = graphistry.spanner_gql_to_df(query_tables)
tables_df
```

[21]:

	table_name	table_type
0	Account	BASE TABLE
1	AccountAudits	BASE TABLE
2	AccountRepayLoan	BASE TABLE
3	AccountTransferAccount	BASE TABLE
4	Loan	BASE TABLE
5	Person	BASE TABLE
6	PersonOwnAccount	BASE TABLE

Continue your Graph and AI journey...

Graphistry

Get started with PyGraphistry and explore AI-powered visual analytics:

- https://github.com/graphistry/pygraphistry/blob/master/demos/10-mins-to-pygraphistry.ipynb
- https://www.graphistry.com/get-started - Learn how to leverage Graphistry for your projects.
- https://louie.ai - AI-powered data insights and visualization.
- https://www.graphistry.com/support - Get support and insights from experts.

Google Spanner

Explore Google Spanner graph features, documentation, and use cases:

- https://cloud.google.com/python/docs/reference/spanner/latest - Official Python client library for Google Spanner.
- https://codelabs.developers.google.com/codelabs/spanner-graph-getting-started - Hands-on tutorial for getting started with Spanner graph features.
- https://cloud.google.com/spanner/docs/graph/overview - Learn about Spanner's graph processing capabilities.
- https://cloud.google.com/spanner/docs/graph/queries-overview - Understand how to query graph data in Spanner.
- https://cloud.google.com/spanner/docs/reference/standard-sql/graph-query-statements - Reference guide for writing graph queries in Spanner.
- https://cloud.google.com/spanner/docs/schema-and-data-model

10.8.6.16 SQL Example

Get & plot data from the first table in a SQL DB

- Uses SQL ODBC drivers prepacked with Graphistry
- Default: visualizes the schema migration table used within Graphistry
- Shows several viz modes + a convenience function for sql->interactive viz
- Try: Modify the indicated lines to change to visualize any other table

Further reading: - https://hub.graphistry.com/docs/ui/index/ - *CSV upload notebook app*

Setup

Graphistry

[ ]:
```
import graphistry

#pip install graphistry -q

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

SQL connection string

- Modify with your own db connection strig
- For heavier use and sharing, see sample for hiding creds from the notebook while still reusing them across sessions

[ ]:
```
user = "graphistry"
pwd = "password"
server = "postgres:5432"

##OPTIONAL: Mount in installation's ${PWD}/.notebooks/db_secrets.json and read in
#import json
#with open('/home/graphistry/notebooks/db_secrets.json') as json_file:
#    cfg = json.load(json_file)
#    user = cfg['user']
#    pwd = cfg['pwd']
#    server = cfg['server']
#
## .. The first time you run this notebook, save the secret cfg to the system's␣
↪persistent notebook folder:
#import json
#with open('/home/graphistry/notebooks/db_secrets.json', 'w') as outfile:
#    json.dump({
#        "user": "graphistry",
#        "pwd": "password",
#        "server": "postgres:5432"
#    }, outfile)
## Delete ^^^ after use

db_string = "postgres://" + user + ":" + pwd + "@" + server
### Take care not to save a print of the result
```

[ ]:
```
# OPTIONAL: Install ODBC drivers in other environments:
# ! apt-get update
# ! apt-get install -y g++ unixodbc unixodbc-dev
# ! conda install -c anaconda pyodbc=4.0.26 sqlalchemy=1.3.5
```

Connect to DB

[ ]:
```
import pandas as pd
import sqlalchemy as db
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, Float, String
from sqlalchemy.orm import sessionmaker
```

[ ]:
```
engine = create_engine(db_string)
Session = sessionmaker(bind=engine)
session = Session()
```

Inspect available tables

[ ]:
```
table_names = engine.table_names()
", ".join(table_names)
```

Optional: Modify to pick your own table!

[ ]:
```
if 'django_migrations' in table_names:
    table = 'django_migrations'
else:
    table = table_names[0]
```

Initialize viz: Get data

[ ]:
```
result = engine.execute("SELECT * FROM \"" + table + "\" LIMIT 1000")
df = pd.DataFrame(result.fetchall(), columns=result.keys())
print("table", table, '# rows', len(df))
df.sample(min(3, len(df)))
```

Plot

Several variants: 1. Treat each row & cell value as a node, and connect row<>cell values 2. Treat each cell value as a node, and connect all cell values together when they occur on the same row 3. Treat each cell value as an edge, and specify which columns to to connect values together on 4. Use explict node/edge tables

1. Treat each row & cell value as a node, and connect row<>cell values

[ ]:
```
graphistry.hypergraph(df)['graph'].plot()
```

2. Treat each cell value as a node, and connect all cell values together when they occur on the same row

[ ]:
```
graphistry.hypergraph(df, direct=True)['graph'].plot()
```

3. Treat each cell value as an edge, and specify which columns to to connect values together on

[ ]:
```
graphistry.hypergraph(df, direct=True,
    opts={
        'EDGES': {
            'id': ['name'],
            'applied': ['name'],
            'name': ['app']
        }
    })['graph'].plot()
```

4. Use explict node/edge tables

[ ]:
```
g = graphistry.bind(source='name', destination='app').edges(df.assign(name=df['name'].
↪apply(lambda x: 'id_' + x)))
g.plot()
```

[ ]:
```
# Add node bindings..
nodes_df = pd.concat([
    df[['name', 'id', 'applied']],
    df[['app']].drop_duplicates().assign(\
          id = df[['app']].drop_duplicates()['app'], \
        name = df[['app']].drop_duplicates()['app'])
], ignore_index=True, sort=False)

g = g.bind(node='id', point_title='name').nodes(nodes_df)

g.plot()
```

Convenience function

[ ]:
```
def explore(sql, *args, **kvargs):
    result = engine.execute(sql)
    df = pd.DataFrame(result.fetchall(), columns=result.keys())
    print('# rows', len(df))
    g = graphistry.hypergraph(df, *args, **kvargs)['graph']
    return g
```

Simple use

```
[ ]: explore("SELECT * FROM django_migrations LIMIT 1000").plot()
```

Pass in graphistry.hypergraph() options

```
[ ]: explore("SELECT * FROM django_migrations LIMIT 1000", direct=True).plot()
```

Get data back

```
[ ]: explore("SELECT * FROM django_migrations LIMIT 1000")._nodes
```

Further docs

- https://hub.graphistry.com/docs/ui/index/
- *CSV upload notebook app*

```
[ ]:
```

10.8.6.17 Tigergraph Bindings: Demo of IT Infra Analysis

Uses bindings built into PyGraphistry for Tigergraph:

- Configure DB connection
- Call dynamic endpoints for user-defined endpoints
- Call interpreted-mode query
- Visualize results

Import and connect

```
[12]: import graphistry

# !pip install graphistry -q

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

```
[13]: g = graphistry.tigergraph(
          protocol='http', server='www.acme.org',
          user='tigergraph', pwd='tigergraph',
          db='Storage', #optional
```

(continues on next page)

(continued from previous page)

```
    #web_port = 14240, api_port = 9000, verbose=True
)
```

Dynamic user-defined GSQL endpoints: Call, analyze, & plot

[14]:
```
g2 = g.gsql_endpoint(
    'StorageImpact', {'vertexType': 'Service', 'input': 61921, 'input.type': 'Pool'},
    #{'edges': '@@edgeList', 'nodes': '@@nodeList'}
)

print('# edges:', len(g2._edges))

g2.plot()
```

```
# edges: 241
```

[14]:
```
<IPython.core.display.HTML object>
```

On-the-fly GSQL interpreted queries: Call, analyze, & plot

[15]:
```
g3 = g.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 @@edgeList;
  }
    """,
  #{'edges': '@@edgeList', 'nodes': '@@nodeList'} # can skip by default
)

print('# edges:', len(g3._edges))

g3.plot()
```

```
# edges: 241
```

[15]:
```
<IPython.core.display.HTML object>
```

[ ]:
```
```

10.8.6.18 Tigergraph<>Graphistry Fraud Demo: Raw REST

Accesses Tigergraph's fraud demo directly via manual REST calls

[ ]:
```
#!pip install graphistry

import pandas as pd
import graphistry
import requests

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

TIGER = "http://MY_TIGER_SERVER:9000"
```

[ ]:
```
#curl -X GET "http://MY_TIGER_SERVER:9000/query/circleDetection?srcId=111"

# string -> dict
def query_raw(query_string):
    url = TIGER + "/query/" + query_string
    r = requests.get(url)
    return r.json()


def flatten (lst_of_lst):
    try:
        if type(lst_of_lst[0]) == list:
            return [item for sublist in lst_of_lst for item in sublist]
        else:
            return lst_of_lst
    except:
        print('fail', lst_of_lst)
        return lst_of_lst

#str * dict -> dict
def named_edge_to_record(name, edge):
    record = {k: edge[k] for k in edge.keys() if not (type(edge[k]) == dict) }
    record['type'] = name
    nested = [k for k in edge.keys() if type(edge[k]) == dict]
    if len(nested) == 1:
        for k in edge[nested[0]].keys():
            record[k] = edge[nested[0]][k]
    else:
```

(continues on next page)

(continued from previous page)

```
        for prefix in nested:
            for k in edge[nested[prefix]].keys():
                record[prefix + "_" + k] = edge[nested[prefix]][k]
    return record


def query(query_string):
    results = query_raw(query_string)['results']
    out = {}
    for o in results:
        for k in o.keys():
            if type(o[k]) == list:
                out[k] = flatten(o[k])
    out = flatten([[named_edge_to_record(k,v) for v in out[k]] for k in out.keys()])
    print('# results', len(out))
    return pd.DataFrame(out)


def plot_edges(edges):
    return graphistry.bind(source='from_id', destination='to_id').edges(edges).plot()
```

1. Fraud

1.a circleDetection

[ ]:
```
circle = query("circleDetection?srcId=10")
circle.sample(3)
```

[ ]:
```
plot_edges(circle)
```

1.b fraudConnectivity

[ ]:
```
connectivity = query("fraudConnectivity?inputUser=111&trustScore=0.1")
connectivity.sample(3)
```

[ ]:
```
plot_edges(connectivity)
```

Combined

[ ]:
```
circle['provenance'] = 'circle'
connectivity['provenance'] = 'connectivity'

plot_edges(pd.concat([circle, connectivity]))
```

Color by type

```
[ ]: edges = pd.concat([circle, connectivity])

     froms = edges.rename(columns={'from_id': 'id', 'from_type': 'node_type'})[['id', 'node_
     ↪type']]
     tos = edges.rename(columns={'to_id': 'id', 'to_type': 'node_type'})[['id', 'node_type']]
     nodes = pd.concat([froms, tos], ignore_index=True).drop_duplicates().dropna()
     nodes.sample(3)
```

```
[ ]: nodes['node_type'].unique()
```

```
[ ]: #https://hub.graphistry.com/docs/api/api-color-palettes/

     type2color = {
         'User': 0,
         'Transaction': 1,
         'Payment_Instrument': 2,
         'Device_Token': 3
     }

     nodes['color'] = nodes['node_type'].apply(lambda type_str: type2color[type_str])

     nodes.sample(3)
```

```
[ ]: graphistry.bind(source='from_id', destination='to_id', node='id', point_color='color').
     ↪edges(edges).nodes(nodes).plot()
```

10.8.6.19 Graphistry Tutorial: Notebooks + TigerGraph via raw REST calls

- Connect to Graphistry, TigerGraph
- Load data from TigerGraph into a Pandas Dataframes
- Plot in Graphistry as a Graph and Hypergraph
- Explore in Graphistry
- Advanced notebooks

Configuration

```
[ ]: TIGER_CONFIG = {
         'fqdn': 'http://MY_TIGER_SERVER:9000'
     }
```

Connect to Graphistry + Test

[ ]:
```
#!pip install graphistry
```

[ ]:
```
import pandas as pd
import requests
```

[ ]:
```
### COMMON ISSUES: wrong server, wrong key, wrong protocol, network notebook->graphistry␣
↪firewall permissions

import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

graphistry.__version__
```

[ ]:
```
### EXPECTED RESULT: Visualization of a curved triangle
### COMMON ISSUES: Blank box as HTTPS not configured on Graphistry server so browser␣
↪disallows iframe. Try plot(render=False)

g = graphistry\
  .edges(pd.DataFrame({'s': [0,1,2], 'd': [1,2,0], 'a': ['quick', 'brown', 'fox'] }))\
  .bind(source='s', destination='d')

g.plot() #g.plot(render=False)
```

Connect to TigerGraph and Test

[ ]:
```
### EXPECTED RESULT: {'GET /statistics': ...}
### COMMON ISSUES: returns '{}' (may need to run a few times); wrong fqdn; firewall␣
↪issues; ...
requests.get(TIGER_CONFIG['fqdn'] + '/statistics?seconds=60').json()
```

Query Tigergraph

[ ]:
```
# string -> dict
def query_raw(query_string):
    url = TIGER_CONFIG['fqdn'] + "/query/" + query_string
    r = requests.get(url)
    return r.json()


def flatten (lst_of_lst):
    try:
        if type(lst_of_lst[0]) == list:
```

(continues on next page)

(continued from previous page)

```
            return [item for sublist in lst_of_lst for item in sublist]
        else:
            return lst_of_lst
    except:
      print('fail', lst_of_lst)
      return lst_of_lst

#str * dict -> dict
def named_edge_to_record(name, edge):
    record = {k: edge[k] for k in edge.keys() if not (type(edge[k]) == dict) }
    record['type'] = name
    nested = [k for k in edge.keys() if type(edge[k]) == dict]
    if len(nested) == 1:
        for k in edge[nested[0]].keys():
            record[k] = edge[nested[0]][k]
    else:
        for prefix in nested:
            for k in edge[nested[prefix]].keys():
                record[prefix + "_" + k] = edge[nested[prefix]][k]
    return record


def query(query_string):
    results = query_raw(query_string)['results']
    out = {}
    for o in results:
        for k in o.keys():
            if type(o[k]) == list:
                out[k] = flatten(o[k])
    out = flatten([[named_edge_to_record(k,v) for v in out[k]] for k in out.keys()])
    print('# results', len(out))
    return pd.DataFrame(out)



def graph_edges(edges):
  return graphistry.bind(source='from_id', destination='to_id').edges(edges)
```

[ ]:
```
df = query("connection_mining?A=1&B=10&k=1000")
print('rows: ', len(df))
df.sample(3)
```

Visualize result of TigerGraph query

```
[ ]: ### EXPECTED RESULT: GRAPH VISUALIZATION
     ### COMMON ISSUES: try inspecting  query_raw('connection_mining?A=1&B=10&k=2')


     graph_edges(query("connection_mining?A=1&B=10&k=1000")).plot()
```

In-Tool UI Walkthrough

1. Clustering, Pan/Zoom, Data Table + Data Brush

Open https://hub.graphistry.com/docs/ui/index/ in a separate tab

1. **Toggle visual clustering**: Click to start, click to stop. (Edges invisible during clustering.)
2. **Pan/zoom**: Just like Google maps
3. **Autocenter** button when lost
4. Click node or edge to see details.
5. **Data Table** with Nodes, Edges, (Events) tabs
6. Use **Data brush** mode to click-drag to select region and filter data table

Challenge: What node has the most edges? What do its edges have in common?

2. Histograms and Using data for sizes & colors

- For `point:degree` histogram on bottom right, press each button and see what it does
- Set node size based on attribute. Then, `Scene settings` -> `Point size` slider.
- Make histogram *log scale* in case of an extreme distribution
- Pick any color. If UI doesn't update, try running clustering for one tick.
- `Add a histogram` for `point:_title`
- Try coloring via a `categorical` vs `gradient` : What is the difference?

3. Filtering

- Add histogram `edge:from_type`
- Click-drag the degree histogram to filter for multiple bins
- Open/close filter panel and toggle on/off the filter
- Toggle `cull isolated nodes` to remove noisey nodes with no edges left
- Click filter on histogram to remove
- You can manually create SQL WHERE clauses here. `filters` -> `edge:e_type` -> `edge:e_type ilike "%phone%"`
- `Toggle visual clustering` and then off when stablized

Challenge: How many distinct phone networks are there?

4. Data table

- Search points, e.g., 135 area code
- Export CSV (currently returns filtered as well)

Advanced Notebooks

Hypergraph

If you have a CSV and not a graph, hypergraphs are a quick way to analyze the data as a graph. They turn each entity into a node, and link them together if they are in the same row of the CSV. E.g., link together a phone and address. It does so indirectly – it creates a node for the row, and connects the row to each entity mentioned.

Challenge: What was the last tainted transaction, and the amount on it?

[ ]:
```
df = pd.read_csv('https://github.com/graphistry/pygraphistry/raw/master/demos/data/
↪transactions.csv')
df.sample(10)
```

[ ]:
```
hg = graphistry.hypergraph(df[:1000], entity_types=['Source', 'Destination',
↪'Transaction ID'])
print('Hypergraph parts', hg.keys())
hg['graph'].plot()
```

[ ]:
```
help(graphistry.hypergraph)
```

Adding Graphs

[ ]:
```
df1 = query("connection_mining?A=1&B=10&k=1000").assign(data_source='query1')
df2 = query("connection_mining?A=1&B=12&k=1000").assign(data_source='query2')
edges2 = pd.concat([df1, df2], ignore_index=True)

graph_edges(edges2).plot()
```

Custom Nodes and Attributes + Saving Sessions

[ ]:
```
conn = query("connection_mining?A=1&B=10&k=1000")

froms = conn.rename(columns={'from_id': 'id', 'from_type': 'node_type'})[['id', 'node_
↪type']]
tos = conn.rename(columns={'to_id': 'id', 'to_type': 'node_type'})[['id', 'node_type']]
nodes = pd.concat([froms, tos], ignore_index=True).drop_duplicates().dropna()
nodes.sample(3)
```

[ ]:
```
nodes['node_type'].unique()
```

[ ]:
```
#https://hub.graphistry.com/docs/api/api-color-palettes/

type2color = {
    'phone_call': 0,
    'citizen': 1,
    'bank_account': 2,
    'phone_number': 3,
    'bank_transfer_event': 4,
    'hotel_room_event': 5
}

nodes['color'] = nodes['node_type'].apply(lambda type_str: type2color[type_str])

nodes.sample(3)
```

[ ]:
```
g = graphistry.bind(source='from_id', destination='to_id').edges(conn)

#updating colors
g = g.bind(node='id', point_color='color').nodes(nodes)

#saving sessions
g = g.settings(url_params={'workbook': 'my_workbook1'})


g.plot()
```

[ ]:
```

```

10.8.7 Plugins - Compute & Layout

10.8.7.1 Gephi (GEXF)

GEXF (Graph Exchange XML Format) is a common interchange format used by Gephi and other tools. This notebook covers a small local sample and a medium-sized dataset with GEXF viz metadata.

[1]:
```
import os
from pathlib import Path
from urllib.request import urlretrieve


import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

[2]:
```
GRAPHISTRY_SERVER = os.environ.get("GRAPHISTRY_SERVER", "hub.graphistry.com")
GRAPHISTRY_PROTOCOL = os.environ.get("GRAPHISTRY_PROTOCOL", "https")
GRAPHISTRY_USERNAME = os.environ.get("GRAPHISTRY_USERNAME")
GRAPHISTRY_PASSWORD = os.environ.get("GRAPHISTRY_PASSWORD")

if not GRAPHISTRY_USERNAME or not GRAPHISTRY_PASSWORD:
    raise RuntimeError("Set GRAPHISTRY_USERNAME and GRAPHISTRY_PASSWORD to upload.")

graphistry.register(
    api=3,
    protocol=GRAPHISTRY_PROTOCOL,
    server=GRAPHISTRY_SERVER,
    username=GRAPHISTRY_USERNAME,
    password=GRAPHISTRY_PASSWORD,
)
```

[2]:
```
<graphistry.pygraphistry.GraphistryClient at 0x7a5cbe8579b0>
```

[3]:
```
gexf_path = Path("demos/demos_databases_apis/gexf/sample.gexf")
if not gexf_path.exists():
    gexf_path = Path("sample.gexf")
g = graphistry.gexf(str(gexf_path))

g._nodes.head()
```

[3]:
```
  node_id    label category viz_color  viz_opacity  viz_x  viz_y  viz_z  \
0     n10    Delta    typeA   #EFAD42          0.5   10.0   20.5    0.0
1     n11  Epsilon    typeB   #0A141E          1.0   -5.0    7.5    0.0

   viz_size viz_shape viz_shape_icon
0      2.50      disc         circle
1      1.25    square         square
```

GEXF viz attributes map to Graphistry bindings (color, size, position, opacity, icons). You can plot directly using the GEXF defaults:

[4]:
```
g.name("GEXF sample").plot()
```

[4]:
```
<IPython.core.display.HTML object>
```

Medium GEXF demo: SiS Words

This dataset includes GEXF viz encodings for node color, size, and position. The source uses a single color and size value, so the default plot looks uniform. Below we show the faithful default binding, how to drop GEXF colors/sizes while keeping layout, and then how to apply Graphistry encodings.

[5]:
```
DATA_URL = "https://raw.githubusercontent.com/medialab/medialab-network-dataset/master/
↪SiS%20Words.gexf"
DATA_DIR = Path("demos/demos_databases_apis/gexf/data")
if not DATA_DIR.exists():
```

(continues on next page)

(continued from previous page)

```
    DATA_DIR = Path("data")
GEXF_PATH = DATA_DIR / "sis_words.gexf"

DATA_DIR.mkdir(parents=True, exist_ok=True)
if not GEXF_PATH.exists():
    urlretrieve(DATA_URL, GEXF_PATH)

GEXF_PATH.exists()
```

[5]:
```
True
```

[6]:
```
g = graphistry.gexf(str(GEXF_PATH))
counts = {"nodes": len(g._nodes), "edges": len(g._edges)}
bindings = {
    "point_color": g._point_color,
    "point_size": g._point_size,
    "point_x": g._point_x,
    "point_y": g._point_y,
    "edge_color": g._edge_color,
    "play": g._url_params.get("play"),
}
counts, bindings
```

[6]:
```
({'nodes': 6704, 'edges': 71744},
 {'point_color': 'viz_color',
  'point_size': 'viz_size',
  'point_x': 'viz_x',
  'point_y': 'viz_y',
  'edge_color': None,
  'play': 0})
```

[7]:
```
g._nodes.head()
```

[7]:
```
  node_id                                label  \
0  w70401                 populations indigènes
1  w70416          impact des activités humaines
2  w70453             préservation de la qualité
3  w70455              préservation de la nature
4  w70454  préservation des ressources naturelles

                                          class  main  occurences  \
0  populations et amélioration des conditions de vie  True           3
1  réchauffement climatique et elavation du nivea...  True           2
2            développement durable et environnement  True           3
3   préservation de la nature et de la biodiversité  True           2
4            développement durable et environnement  True           4

   viz_size       viz_x       viz_y  viz_z viz_color
0      10.0  -649.47797 -996.466860    0.0   #999999
1      10.0   789.25270   10.201024    0.0   #999999
```

(continues on next page)

(continued from previous page)

```
2      10.0  1131.04210 -927.317500    0.0   #999999
3      10.0  1068.51270 -995.344000    0.0   #999999
4      10.0   982.29270 -890.796300    0.0   #999999
```

[8]:
```
g._edges.head()
```

[8]:
```
   source  target
0  w70401  w69745
1  w70401  w69741
2  w70401  w54632
3  w70401  w53692
4  w70401  w53637
```

[9]:
```
g.name("SiS Words (GEXF defaults)").plot()
```

[9]:
```
<IPython.core.display.HTML object>
```

Drop GEXF colors/sizes (keep layout)

Use `bind_node_viz` / `bind_edge_viz` to keep only the bindings you want. Here we keep position for layout, and drop color/size/opacity/icon bindings.

[10]:
```
g_layout_only = graphistry.gexf(
    str(GEXF_PATH),
    bind_node_viz=["position"],
    bind_edge_viz=[],
)
```

[11]:
```
g_layout_only.name("SiS Words (layout only)").plot()
```

[11]:
```
<IPython.core.display.HTML object>
```

Apply Graphistry encodings

After dropping bindings, use Graphistry encodings for color/size. Here we color by `class` using a categorical mapping for the most frequent classes (and a default for everything else), and size by `occurences`.

[12]:
```
required_cols = ["class", "occurences"]
missing_cols = [col for col in required_cols if col not in g_layout_only._nodes.columns]
assert not missing_cols, f"Missing expected node columns: {missing_cols}"

class_counts = g_layout_only._nodes["class"].value_counts()
top_classes = class_counts.head(8).index.tolist()
palette = ["#4C78A8", "#F58518", "#54A24B", "#E45756", "#72B7B2", "#EECA3B", "#B279A2", "
↪#FF9DA6"]
class_color_map = dict(zip(top_classes, palette))
```

(continues on next page)

(continued from previous page)

```
g_encoded = (
    g_layout_only
    .encode_point_color(
        "class",
        categorical_mapping=class_color_map,
        default_mapping="#D0D0D0",
        as_categorical=True,
    )
    .encode_point_size("occurences")
)

class_color_map
```

[12]:
```
{'génétique': '#4C78A8',
 'préservation de la nature et de la biodiversité': '#F58518',
 'commerce équitable': '#54A24B',
 'développement durable et environnement': '#E45756',
 'gaz à effet de serre et pollution de l air': '#72B7B2',
 'ma^ıtrise de l énergie': '#EECA3B',
 'connaissance et domaine scientifique': '#B279A2',
 'culture scientifique et technique et pédagogie des sciences': '#FF9DA6'}
```

[13]:
```
g_encoded.name("SiS Words (layout + encodings)").plot()
```

[13]:
```
<IPython.core.display.HTML object>
```

10.8.7.2 Gephi (GEXF) datasets

GEXF provides a small set of example datasets at https://gexf.net/datasets.html. This notebook downloads a few of them and renders with Graphistry.

[1]:
```
import os
from pathlib import Path
from urllib.request import Request, urlopen


import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

[2]:
```
GRAPHISTRY_SERVER = os.environ.get("GRAPHISTRY_SERVER", "hub.graphistry.com")
GRAPHISTRY_PROTOCOL = os.environ.get("GRAPHISTRY_PROTOCOL", "https")
GRAPHISTRY_USERNAME = os.environ.get("GRAPHISTRY_USERNAME")
GRAPHISTRY_PASSWORD = os.environ.get("GRAPHISTRY_PASSWORD")
```

(continues on next page)

(continued from previous page)

```
if not GRAPHISTRY_USERNAME or not GRAPHISTRY_PASSWORD:
    raise RuntimeError("Set GRAPHISTRY_USERNAME and GRAPHISTRY_PASSWORD to upload.")

graphistry.register(
    api=3,
    protocol=GRAPHISTRY_PROTOCOL,
    server=GRAPHISTRY_SERVER,
    username=GRAPHISTRY_USERNAME,
    password=GRAPHISTRY_PASSWORD,
)
```

[2]:
```
<graphistry.pygraphistry.GraphistryClient at 0x7bc24bf15d90>
```

We will download these datasets into a local `data/` folder:

- C. elegans
- Yeast
- EuroSiS web graph

[3]:
```
DATA_DIR = Path("demos/demos_databases_apis/gexf/data")
if not DATA_DIR.exists():
    DATA_DIR = Path("data")
DATA_DIR.mkdir(parents=True, exist_ok=True)

def download_gexf(url, path):
    req = Request(url, headers={"User-Agent": "Mozilla/5.0"})
    with urlopen(req) as response, open(path, "wb") as f:
        f.write(response.read())

DATASETS = [
    ("C. elegans", "https://gexf.net/data/celegans.gexf", DATA_DIR / "celegans.gexf"),
    ("Yeast", "https://gexf.net/data/yeast.gexf", DATA_DIR / "yeast.gexf"),
    ("EuroSiS", "https://gexf.net/data/WebAtlas_EuroSiS.gexf", DATA_DIR / "WebAtlas_
↪EuroSiS.gexf"),
]

for name, url, path in DATASETS:
    if not path.exists():
        download_gexf(url, path)

[path.exists() for _, _, path in DATASETS]
```

[3]:
```
[True, True, True]
```

C. elegans

[4]:
```
g_celegans = graphistry.gexf(str(DATA_DIR / "celegans.gexf"))
counts = {"nodes": len(g_celegans._nodes), "edges": len(g_celegans._edges)}
bindings = {
    "point_color": g_celegans._point_color,
    "point_size": g_celegans._point_size,
    "point_x": g_celegans._point_x,
    "point_y": g_celegans._point_y,
    "edge_color": g_celegans._edge_color,
    "play": g_celegans._url_params.get("play"),
}
counts, bindings
```

[4]:
```
({'nodes': 306, 'edges': 2345},
 {'point_color': None,
  'point_size': None,
  'point_x': None,
  'point_y': None,
  'edge_color': None,
  'play': None})
```

[5]:
```
g_celegans._nodes.head()
```

[5]:
```
  node_id label
0       0     1
1       1     2
2      10    11
3     100   101
4     101   102
```

[6]:
```
g_celegans.name("C. elegans (GEXF)").plot()
```

[6]:
```
<IPython.core.display.HTML object>
```

Yeast

[7]:
```
g_yeast = graphistry.gexf(str(DATA_DIR / "yeast.gexf"))
counts = {"nodes": len(g_yeast._nodes), "edges": len(g_yeast._edges)}
bindings = {
    "point_color": g_yeast._point_color,
    "point_size": g_yeast._point_size,
    "point_x": g_yeast._point_x,
    "point_y": g_yeast._point_y,
    "edge_color": g_yeast._edge_color,
    "play": g_yeast._url_params.get("play"),
}
counts, bindings
```

[7]:
```
({'nodes': 2361, 'edges': 7182},
 {'point_color': None,
  'point_size': None,
  'point_x': None,
  'point_y': None,
  'edge_color': None,
  'play': None})
```

[8]:
```
g_yeast._nodes.head()
```

[8]:
```
  node_id    label
0    4941  YBR236C
1    4942  YOR151C
2    4943  YML010W
3    4944  YNR016C
4    4945  YLR386W
```

[9]:
```
g_yeast.name("Yeast (GEXF)").plot()
```

[9]:
```
<IPython.core.display.HTML object>
```

EuroSiS web graph

[10]:
```
g_eurosis = graphistry.gexf(str(DATA_DIR / "WebAtlas_EuroSiS.gexf"))
counts = {"nodes": len(g_eurosis._nodes), "edges": len(g_eurosis._edges)}
bindings = {
    "point_color": g_eurosis._point_color,
    "point_size": g_eurosis._point_size,
    "point_x": g_eurosis._point_x,
    "point_y": g_eurosis._point_y,
    "edge_color": g_eurosis._edge_color,
    "play": g_eurosis._url_params.get("play"),
}
counts, bindings
```

[10]:
```
({'nodes': 1285, 'edges': 7524},
 {'point_color': None,
  'point_size': None,
  'point_x': None,
  'point_y': None,
  'edge_color': None,
  'play': None})
```

[11]:
```
g_eurosis._nodes.head()
```

[11]:
```
  node_id                                        label         country  \
0      10                      Astronomical Institute  Czech Republic
1    1002                CCSTI La Turbine Rh^one-Alpes          France
```

(continues on next page)

(continued from previous page)

```
2    1003              Laurea University of Applied Sciences         Finland
3    1004  European Association for Education Law and Policy   International
4    1006                           Les petits débrouillards         Belgium

  tag_gender tag_governance tag_health tag_info tag_internat tag_nano  \
0          0              0          0        0            0        0
1          0              0          0        0            0        0
2          0              0          0        0            0        0
3          0              0          0        0            0        0
4          0              0          0        0            0        0

  tag_people  ... tag_socioeco tag_space tag_transport tag_agri tag_biotech  \
0          0  ...            0         0             0        0           0
1          0  ...            0         0             0        0           0
2          0  ...            0         0             0        0           0
3          0  ...            0         0             0        0           0
4          0  ...            0         0             0        0           0

  tag_business tag_comm tag_energy tag_environment tag_food
0            0        1          0               0        0
1            0        1          0               0        0
2            0        0          0               0        0
3            0        0          0               0        0
4            0        1          0               0        0

[5 rows x 25 columns]
```

[12]:
```
g_eurosis.name("EuroSiS (GEXF)").plot()
```

[12]:
```
<IPython.core.display.HTML object>
```

10.8.7.3 Graphviz Layouts for Graphistry Visualization

The https://graphviz.org/ is popular for high-quality layout of small graphs, especially trees and directed acyclic graphs (DAGs).

This notebook shows how to use graphviz for **layout**, then visualize interactively with Graphistry's GPU-accelerated renderer.

For static image export (SVG, PNG for docs/reports), see the *static_rendering.ipynb* notebook.

The example below shows laying out and rendering company ownership data that is in a tree and benefits from graphviz's high-quality layout engine.

Setup

- graphviz: Install the graphviz engine and the pygraphviz bindings, see below (official https://pygraphviz.github.io/documentation/stable/install.html)
- Graphistry: Install PyGraphistry below, and https://www.graphistry.com/get-started or run your own server

Notes:

- You must install the graphviz engine, as well as its pygraphviz Python bindings and pygraphistry
- graphviz is most known for its `"dot"` layout engine, and it includes others as well
- graphviz is generally not recommended for layout of graphs over 10,000 nodes and edges

[1]:
```
#!apt-get install graphviz graphviz-dev

#!pip install -q graphistry[pygraphviz]
```

Imports

[2]:
```
import logging
try:
  import pygraphviz as pgv
except (ImportError, ModuleNotFoundError):
  logging.error("ImportError: Did you install pygraphviz and the supporting native␣
↪packages?")
  raise

import pandas as pd
import graphistry
from graphistry import Plottable
graphistry.register(api=3, username=FILL_ME_IN, password=FILL_ME_IN)

graphistry.__version__
```

[2]:
```
'0+unknown'
```

Sample graph: HSBC Beneficial ownership graph

Sample data from https://openownership.org/. Corporate ownership graphs often have deeply tree structure, and for bigger conglomerates with numerous subsidaries, officers, board officers, suppliers, and lenders, can greatly benefit from higher-quality tree layouts.

[3]:
```
companies_df = pd.DataFrame([{'label': 'Hsbc Finance (Netherlands)', 'n':
↪'1862294673469042014'},
 {'label': 'Hsbc Holdings Plc', 'n': '7622088245850069747'},
 {'label': 'Unknown person(s)', 'n': '7622088245850069747-unknown'},
 {'label': 'HSBC PROPERTY (UK) LIMITED', 'n': '16634236373777089526'},
 {'label': 'HSBC ALTERNATIVE INVESTMENTS LIMITED',
  'n': '18011320449780894329'},
 {'label': 'HSBC INVESTMENT COMPANY LIMITED', 'n': '9134577322728469115'},
```

(continues on next page)

(continued from previous page)

```
 {'label': 'HSBC IM PENSION TRUST LIMITED', 'n': '1446072728533515665'},
 {'label': 'MERCANTILE COMPANY LIMITED', 'n': '6904185395252167658'},
 {'label': 'Mp Payments Group Limited', 'n': '13630126251685975826'},
 {'label': 'MP PAYMENTS OPERATIONS LIMITED', 'n': '11514603667851101425'},
 {'label': 'MP PAYMENTS UK LIMITED', 'n': '13417892994160273884'},
 {'label': 'Hsbc Asia Pacific Holdings (Uk) Limited',
  'n': '2173486047275631423'},
 {'label': 'HSBC SECURITIES (JAPAN) LIMITED', 'n': '18045747820524565803'}])

ownership_df = pd.DataFrame([{'s': '7622088245850069747', 'd': '1862294673469042014'},
 {'s': '7622088245850069747-unknown', 'd': '7622088245850069747'},
 {'s': '1862294673469042014', 'd': '16634236373777089526'},
 {'s': '1862294673469042014', 'd': '18011320449780894329'},
 {'s': '1862294673469042014', 'd': '9134577322728469115'},
 {'s': '9134577322728469115', 'd': '1446072728533515665'},
 {'s': '9134577322728469115', 'd': '6904185395252167658'},
 {'s': '9134577322728469115', 'd': '13630126251685975826'},
 {'s': '13630126251685975826', 'd': '11514603667851101425'},
 {'s': '13630126251685975826', 'd': '13417892994160273884'},
 {'s': '9134577322728469115', 'd': '2173486047275631423'},
 {'s': '2173486047275631423', 'd': '18045747820524565803'},
 {'s': '9134577322728469115', 'd': '16634236373777089526'}])
```

[4]:
```
g = graphistry.edges(ownership_df, 's', 'd').nodes(companies_df, 'n').bind(point_title=
↪'label')
```

[5]:
```
g = g.nodes(g._nodes.assign(sz=1)).encode_point_size('sz')
```

Minimal tree layout and graphviz layout engines

Graphviz provides 15+ layout engines you can use. General guidance is to use for graphs up to 10,000 nodes and engines.

The "dot" layout engine is best known due to its beautiful hierarchical layouts for directed acycle graphs like trees.

[6]:
```
g2 = g.layout_graphviz('dot')
g2.plot()
```

[6]:
```
<IPython.core.display.HTML object>
```

[7]:
```
# Quick static render (see static_rendering.ipynb for full guide)
g.plot_static()
```

[7]:

Additional layout engines beyond "dot" are below. See also the https://graphviz.org/docs/layouts/. The same documentation, and the below section on global graph attributes, describe options you can pass in to different layout engines.

[8]:
```
from graphistry.plugins_types.graphviz_types import PROGS
PROGS
```

[8]:
```
['acyclic',
 'ccomps',
 'circo',
 'dot',
 'fdp',
 'gc',
 'gvcolor',
 'gvpr',
 'neato',
 'nop',
 'osage',
 'patchwork',
 'sccmap',
 'sfdp',
 'tred',
 'twopi',
 'unflatten']
```

[9]:
```
g2b = g.layout_graphviz('neato')
g2b.plot()
```

[9]:
```
<IPython.core.display.HTML object>
```

[10]:
```
from graphistry.plugins_types.graphviz_types import PROGS
```

Global attributes

You can set global attributes. Parameter `graph_attr <https://graphviz.org/docs/graph/>`__ generally refers to layout engine options, while `edge_attr <https://graphviz.org/docs/edges/>`__ and `node_attr <https://graphviz.org/docs/nodes/>`__ are generally for default colors, sizes, shapes, etc.

[11]:
```
g2b = g.layout_graphviz(
    'dot',
    graph_attr={'ratio': 10},
    edge_attr={},
    node_attr={}
)
g2b.plot()
```

[11]:
```
<IPython.core.display.HTML object>
```

[12]:
```

from graphistry.plugins_types.graphviz_types import GRAPH_ATTRS
GRAPH_ATTRS
```

[12]:
```
['_background',
 'bb',
 'beautify',
 'bgcolor',
 'center',
 'charset',
 'class',
```

(continues on next page)

(continued from previous page)

```
'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',
```

(continues on next page)

(continued from previous page)

```
 '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']
```

[13]:
```
from graphistry.plugins_types.graphviz_types import EDGE_ATTRS
EDGE_ATTRS
```

[13]:
```
['arrowhead',
 'arrowsize',
 'arrowtail',
```

(continues on next page)

(continued from previous page)

```
'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',
```

(continues on next page)

(continued from previous page)

```
 'taillabel',
 'tailport',
 'tailtarget',
 'tailtooltip',
 'tailURL',
 'target',
 'tooltip',
 'URL',
 'weight',
 'xlabel',
 'xlp']
```

[14]:
```
from graphistry.plugins_types.graphviz_types import NODE_ATTRS
NODE_ATTRS
```

[14]:
```
['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',
```

(continues on next page)

(continued from previous page)

```
 'showboxes',
 'sides',
 'skew',
 'sortv',
 'style',
 'target',
 'tooltip',
 'URL',
 'vertices',
 'width',
 'xlabel',
 'xlp',
 'z']
```

Per-entity styling attributes

You can add graphviz-specific columns to your node and edge dataframes that configure per-row render settings. These use the same names as in the above global attribute guidance, such as `color`, `shape`, and `label`.

Adding a column for an attribute will typically override the global attribute for that row.

[15]:
```
g._nodes.apply(lambda row: row['n'], axis=1)
```

[15]:
```
0              1862294673469042014
1               762208824585006974 7
2     762208824585006974 7-unknown
3             16634236373777089526
4             18011320449780894329
5              9134577322728469115
6              1446072728533515665
7              6904185395252167658
8             13630126251685975826
9             11514603667851101425
10            13417892994160273884
11             2173486047275631423
12            18045747820524565803
dtype: object
```

[16]:
```
g._nodes.apply(lambda row: print('row', row['n']), 1)
```

```
row 1862294673469042014
row 762208824585006974 7
row 762208824585006974 7-unknown
row 16634236373777089526
row 18011320449780894329
row 9134577322728469115
row 1446072728533515665
row 6904185395252167658
row 13630126251685975826
row 11514603667851101425
row 13417892994160273884
```

(continues on next page)

(continued from previous page)

```
row 2173486047275631423
row 18045747820524565803
```

[16]:
```
0     None
1     None
2     None
3     None
4     None
5     None
6     None
7     None
8     None
9     None
10    None
11    None
12    None
dtype: object
```

[17]:
```
# Per-node styling with plot_static()

root_id = '7622088245850069747-unknown'

g2c = g.nodes(g._nodes.assign(
    label=g._nodes.apply(lambda row: "ROOT: Unknown person(s)" if row['n'] == root_id␣
↪else row['label'], axis=1),
    shape=g._nodes.n.apply(lambda n: "box" if n == root_id else None),
    color=g._nodes.n.apply(lambda n: "blue" if n == root_id else 'red')
)).edges(g._edges.assign(
    color=g._edges[g._source].apply(lambda n: 'blue' if n == root_id else None)
))

# Static render with per-node styling
g2c.plot_static(
    prog='dot',
    node_attr={'color': 'green'}  # default, overridden by per-node color column
)
```

[17]:

[18]:
```
g2d = g.layout_graphviz('circo')
g2d.plot()
```

[18]:
```
<IPython.core.display.HTML object>
```

[ ]:

See Also

- *static_rendering.ipynb* - Static image export (SVG, PNG) for docs, reports, CI pipelines
- https://graphviz.org/documentation/ - Full attribute reference
- Layout catalog - All available layout algorithms

10.8.7.4 HyperNetX + Graphistry = StrongStrongStrong

You can quickly explore HyperNetX graphs using Graphistry through the below sample class.

PNNL's https://github.com/pnnl/HyperNetX is a new Python library for manipulating hypergraphs: graphs where an edge may connect any number of nodes. The below helper class converts HyperNetX graphs into 2-edge graphs (node/edge property tables as Panda dataframes) in two different modes:

- `hypernetx_to_graphistry_bipartite(hnx_graph)`:
  - Turn every hyperedge and hypernode into nodes. They form a bipartite graph: whenever a hyperedge includes a hypernode, create an edge from the hyperedge's node to the hypernode's node.
  - ex: Hyperedge `0: ['a', 'b', 'c']` => edge `(0, 'a')`, edge `(0, 'b')`, edge `(0, 'c')`
- `hypernetx_to_graphistry_nodes(hnx_graph)`:
  - Turn every hypernode into a node, and whenever two hypernodes share the same hyperedge, create an edge between their corresponding nodes
  - To emphasize that edges are undirect, the library sets the edge curvature to 0 (straight)
  - ex: Hyperedge `0: ['a', 'b', 'c']` => edge `('a', 'b')`, edge `('a', 'c')`, edge `('b', 'c')`

Install

Dependencies already preinstalled in Graphistry Core distributions

[1]:
```
# ! pip install hypernetx -q
# ! pip install graphistry -q
```

Lib

[2]:
```
import pandas as pd

class HyperNetXG:

    def __init__(self, graphistry):
        self.graphistry = graphistry

    def normalize_id(self, id):
        t = type(id)
        if t == float or t == int:
            return '__id__' + str(id)
        return str(id)
```

(continues on next page)

(continued from previous page)

```
    def hypernetx_to_graphistry_bipartite(self, h):

        nodes_df = pd.concat(
            [pd.DataFrame({
                'node': [self.normalize_id(x) for x in list(H.nodes)],
                'type': 'hypernode'}),
             pd.DataFrame({
                 'node': [self.normalize_id(x) for x in H.edges],
                 'type': 'hyperedge'})],
            ignore_index=True,
            sort=False)

        edges_df = pd.concat(
            [ pd.DataFrame({'src': [], 'dst': []}) ] +
            [
                pd.DataFrame({
                    'src': self.normalize_id(k),
                    'dst': [self.normalize_id(x) for x in list(es)]
                })
                for k, es in H.incidence_dict.items()
            ], ignore_index=True, sort=False)

        return self.graphistry.bind(
              source='src',
              destination='dst',
              node='node').nodes(nodes_df).edges(edges_df)


    def __hyperedge_to_graph(self, k, es):
        lst = list(es)
        edges_df = pd.concat([
              pd.DataFrame({'src': [], 'dst': [], 'hyperedge': []})] + [
              pd.DataFrame({
                  'src': self.normalize_id(lst[i]),
                  'dst': [self.normalize_id(x) for x in lst[i+1:]],
                  'hyperedge': self.normalize_id(k)})
              for i in range(0, len(lst))
          ], ignore_index=True, sort=False)
        return edges_df

    def hypernetx_to_graphistry_nodes(self, h):
        hg = self.hypernetx_to_graphistry_bipartite(h)
        nodes_df = pd.DataFrame({
              'node': [self.normalize_id(x) for x in list(h.nodes)],
              'type': 'hypernode'})
        edges_df = pd.concat(
              [pd.DataFrame({'src': [], 'dst': [], 'hyperedge': []})] +
              [
                  self.__hyperedge_to_graph(k, es)
                  for (k, es) in h.incidence_dict.items()
```

(continues on next page)

(continued from previous page)

```
            ])
        return self.graphistry.bind(
            source='src',
            destination='dst',
            node='node').settings(url_params={'edgeCurvature': 0}).nodes(nodes_df).
↪edges(edges_df)
```

Demo

Init

[3]:
```
import hypernetx as hnx
import graphistry

# To specify Graphistry account & server, use:
# graphistry.register(api=3, username='...', password='...', protocol='https', server=
↪'hub.graphistry.com')
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html

scenes = [
    ('FN', 'TH'),
    ('TH', 'JV'),
    ('BM', 'FN', 'JA'),
    ('JV', 'JU', 'CH', 'BM'),
    ('JU', 'CH', 'BR', 'CN', 'CC', 'JV', 'BM'),
    ('TH', 'GP'),
    ('GP', 'MP'),
    ('MA', 'GP')
]

H = hnx.Hypergraph(dict(enumerate(scenes)))
hg = HyperNetXG(graphistry)
```

hypernetx_to_graphistry_bipartite

Flatten to Pandas Dataframes / Graphistry and inspect:

[4]:
```
g = hg.hypernetx_to_graphistry_bipartite(H)
g._nodes.sample(3)
```

[4]:
```
       node       type
18  __id__5  hyperedge
17  __id__4  hyperedge
15  __id__2  hyperedge
```

[5]:
```
g._edges.sample(3)
```

[5]:
```
        src dst
18  __id__5  TH
```

(continues on next page)

(continued from previous page)

```
20  __id__6  MP
10  __id__3  CH
```

[6]:
```
g.plot()
```

[6]:
```
<IPython.core.display.HTML object>
```

[7]:
```
hg.hypernetx_to_graphistry_bipartite(H.dual())._edges.sample(3)
```

[7]:
```
        src dst
16  __id__4  CN
17  __id__4  BR
14  __id__4  BM
```

[8]:
```
hg.hypernetx_to_graphistry_bipartite(H.dual()).plot()
```

[8]:
```
<IPython.core.display.HTML object>
```

hypernetx_to_graphistry_nodes

[9]:
```
g = hg.hypernetx_to_graphistry_nodes(H)
g._edges.sample(3)
```

[9]:
```
   src dst hyperedge
16  BM  CN   __id__4
0   JA  BM   __id__2
2   JU  CH   __id__3
```

[10]:
```
hg.hypernetx_to_graphistry_nodes(H).plot()
```

[10]:
```
<IPython.core.display.HTML object>
```

[11]:
```
hg.hypernetx_to_graphistry_nodes(H.dual()).plot()
```

[11]:
```
<IPython.core.display.HTML object>
```

10.8.7.5 NetworkX

NetworkX is a CPU graph manipulation library with a variety of algorithms and layouts. This notebook shows two PyGraphistry workflows: converting a NetworkX graph into a PyGraphistry graph, and enriching a PyGraphistry graph with the public `compute_networkx()` API.

Install the optional dependency with `pip install "pygraphistry[networkx]"`. Use `pygraphistry[networkx-scipy]` when you want NetworkX algorithms to use SciPy-backed implementations where NetworkX supports them.

[ ]:
```
import graphistry
import networkx as nx
import pandas as pd

# To specify a Graphistry account & server for plotting, use:
```

(continues on next page)

(continued from previous page)

```
# graphistry.register(api=3, username="...", password="...", protocol="https", server=
↪"hub.graphistry.com")
# For more options: https://pygraphistry.readthedocs.io/en/latest/server/register.html
```

Convert a NetworkX graph

`from_networkx()` preserves node and edge attributes as pandas dataframes. You can inspect or transform the data locally, then call `g.plot()` after registering with a Graphistry server.

[ ]:
```
G = nx.Graph()
G.add_nodes_from([
    (1, {"label": "one"}),
    (2, {"label": "two"}),
    (3, {"label": "three"}),
    (4, {"label": "four"}),
    (7, {"label": "seven"}),
    (8, {"label": "eight"}),
])
G.add_edges_from([
    (2, 3, {"kind": "path"}),
    (3, 4, {"kind": "path"}),
    (7, 8, {"kind": "pair"}),
])

g = graphistry.bind(source="src", destination="dst", node="nodeid").from_networkx(G)

assert isinstance(g._edges, pd.DataFrame)
assert isinstance(g._nodes, pd.DataFrame)

g._edges
```

Enrich nodes with NetworkX algorithms

`compute_networkx()` runs the same curated NetworkX algorithm subset exposed by GFQL local Cypher `CALL graphistry.nx.*`. Node algorithms append columns to `g._nodes` and preserve the graph bindings.

[ ]:
```
g_ranked = g.compute_networkx("degree_centrality", out_col="degree_score",␣
↪directed=False)

assert "degree_score" in g_ranked._nodes.columns
g_ranked._nodes.sort_values("degree_score", ascending=False)
```

Enrich edges or return a projected graph

Edge algorithms append columns to `g._edges`. Graph-returning algorithms such as `k_core` return a new PyGraphistry graph containing the projected NetworkX graph.

[ ]:
```
g_edges = g.compute_networkx("edge_betweenness_centrality", out_col="edge_bc",␣
↪directed=False)
assert "edge_bc" in g_edges._edges.columns

g_core = g.compute_networkx("k_core", params={"k": 1}, directed=False)
assert set(g_core._nodes["nodeid"]) == {2, 3, 4, 7, 8}

g_core._edges
```

Visualize the enriched graph

After registering with a Graphistry server, `plot()` uploads and embeds the interactive graph. This preview colors nodes with a continuous blue-to-red palette from low to high `degree_score`, making the NetworkX result visible in the rendered docs.

[5]:
```
degree_palette = ["#2c7bb6", "#abd9e9", "#ffffbf", "#fdae61", "#d7191c"]
g_colored = g_ranked.encode_point_color("degree_score", palette=degree_palette, as_
↪continuous=True)
g_colored.plot(render="ipython")
```

[5]:
```
<IPython.core.display.HTML object>
```

10.9 Cheatsheets

10.9.1 Install

You need to install the PyGraphistry Python client for local processing, and for server-accelerated visualization, connect it to a Graphistry GPU server of your choice:

10.9.1.1 Graphistry Server

Graphistry server account:

- Create a free https://www.graphistry.com/get-started for open data, or https://www.graphistry.com/get-started
- Later, https://github.com/graphistry/graphistry-cli your own private Docker instance (https://www.graphistry.com/demo-request)

10.9.1.2 PyGraphistry Client - overview

PyGraphistry Python client:

- `pip install --user graphistry` (Python 3.8+) or https://hub.graphistry.com/docs/api/
  - Use `pip install --user graphistry[all]` for optional dependencies such as Neo4j drivers
- To use from a notebook environment, run your own https://jupyter.org/ server (https://www.graphistry.com/get-started) or another such as https://colab.research.google.com (external link)
- See immediately following `configure` section for how to connect

10.9.1.3 PyGraphistry Client - pip install

`pip install` the client in your notebook or web app, and then connect to a https://www.graphistry.com/get-started or https://www.graphistry.com/get-started

```
# pip install --user graphistry              # minimal
# pip install --user graphistry[bolt,gremlin,nodexl,igraph,networkx]  # data plugins
# AI modules: Python 3.8+ with scikit-learn 1.0+:
# pip install --user graphistry[umap-learn]  # Lightweight: UMAP autoML (without text␣
↪support); scikit-learn 1.0+
# pip install --user graphistry[ai]          # Heavy: Full UMAP + GNN autoML, including␣
↪sentence transformers (1GB+)
```

10.9.2 Authenticate

```
import graphistry
graphistry.register(api=3, username='abc', password='xyz')  # Free: hub.graphistry.com
#graphistry.register(..., personal_key_id='pkey_id', personal_key_secret='pkey_secret
↪') # Key instead of username+password+org_name
#graphistry.register(..., is_sso_login=True)  # SSO instead of password
#graphistry.register(..., org_name='my-org') # Upload into an organization account vs␣
↪personal
#graphistry.register(..., protocol='https', server='my.site.ngo')  # Use with a self-
↪hosted server
# ... and if client (browser) URLs are different than python server<> graphistry␣
↪server uploads
#graphistry.register(..., client_protocol_hostname='https://public.acme.co')
```

10.9.2.1 Simple

Most users connect to a Graphistry GPU server account via:

- `graphistry.register(api=3, username='abc', password='xyz')`: personal hub.graphistry.com account
- `graphistry.register(api=3, username='abc', password='xyz', org_name='optional_org')`: team hub.graphistry.com account
- `graphistry.register(api=3, username='abc', password='xyz', org_name='optiona_org', protocol='http', server='my.private_server.org')`: private server

For more advanced configuration, read on for:

- Version: Use protocol `api=3`, which will soon become the default, or a legacy version
- JWT Tokens: Connect to a GPU server by providing a `username='abc'/password='xyz'`, or for advanced long-running service account software, a refresh loop using 1-hour-only JWT tokens
- Organizations: Optionally use `org_name` to set a specific organization
- Private servers: PyGraphistry defaults to using the free https://hub.graphistry.com (external link) public API
  - Connect to a https://www.graphistry.com/get-started and provide optional settings specific to it via `protocol`, `server`, and in some cases, `client_protocol_hostname`

Non-Python users may want to explore the underlying language-neutral https://hub.graphistry.com/docs/api/1/rest/auth/.

10.9.2.2 Advanced Login

- **For people:** Provide your account username/password:

```
import graphistry
graphistry.register(api=3, username='username', password='your password')
```

- **For service accounts**: Long-running services may prefer to use 1-hour JWT tokens:

```
import graphistry
graphistry.register(api=3, username='username', password='your password')
initial_one_hour_token = graphistry.api_token()
graphistry.register(api=3, token=initial_one_hour_token)

# must run every 59min
graphistry.refresh()
fresh_token = graphistry.api_token()
assert initial_one_hour_token != fresh_token
```

Refreshes exhaust their limit every day/month. An upcoming Personal Key feature enables non-expiring use.

Alternatively, you can rerun `graphistry.register(api=3, username='username', password='your password')`, which will also fetch a fresh token.

10.9.2.3 Advanced: Private servers - server uploads

Specify which Graphistry server to reach for Python uploads:

```
graphistry.register(protocol='https', server='hub.graphistry.com')
```

Private Graphistry notebook environments are preconfigured to fill in this data for you:

```
graphistry.register(protocol='http', server='nginx', client_protocol_hostname='')
```

Using `'http'`/`'nginx'` ensures uploads stay within the Docker network (vs. going more slowly through an outside network), and client protocol `''` ensures the browser URLs do not show `http://nginx/`, and instead use the server's name. (See immediately following **Switch client URL** section.)

10.9.2.4 Advanced: Private servers - switch client URL for browser views

In cases such as when the notebook server is the same as the Graphistry server, you may want your Python code to *upload* to a known local Graphistry address without going outside the network (e.g., `http://nginx` or `http://localhost`), but for web viewing, generate and embed URLs to a different public address (e.g., `https://graphistry.acme.ngo/`). In this case, explicitly set a client (browser) location different from `protocol` / `server`:

```
graphistry.register(
    ### fast local notebook<>graphistry upload
    protocol='http', server='nginx',

    ### shareable public URL for browsers
    client_protocol_hostname='https://graphistry.acme.ngo'
)
```

Prebuilt Graphistry servers are already setup to do this out-of-the-box.

10.9.3 Data loading

10.9.3.1 Explore any data as a graph

It is easy to turn arbitrary data into insightful graphs. PyGraphistry comes with many built-in connectors, and by supporting Python dataframes (Pandas, Arrow, RAPIDS), it's easy to bring standard Python data libraries. If the data comes as a table instead of a graph, PyGraphistry will help you extract and explore the relationships.

- https://pandas.pydata.org and https://www.rapids.ai dataframes from CSV, txt, JSON, parquet, arrow, ORC, and more

```
edges = pd.read_csv('facebook_combined.txt', sep=' ', names=['src', 'dst'])
graphistry.edges(edges, 'src', 'dst').plot()
```

```
edges = pd.read_parquet('my.parquet')
graphistry.edges(edges, 'src', 'dst').plot()
```

```
table_rows = pd.read_csv('honeypot.csv')
graphistry.hypergraph(table_rows, ['attackerIP', 'victimIP', 'victimPort', 'vulnName
↪'])['graph'].plot()
```

```
graphistry.hypergraph(table_rows, ['attackerIP', 'victimIP', 'victimPort', 'vulnName
↪'],
    direct=True,
    opts={'EDGES': {
      'attackerIP': ['victimIP', 'victimPort', 'vulnName'],
      'victimIP': ['victimPort', 'vulnName'],
      'victimPort': ['vulnName']
}})['graph'].plot()
```

- GFQL: Graph pattern queries on dataframes with friendly Cypher syntax, declarative semantics, and optional GPU acceleration (https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/hop_and_chain https://github.com/graphistry/pygraphistry/blob/master/demos/gfql/benchmark_hops_cpu_gpu.ipynb)

Run graph queries on dataframes with Cypher syntax through GFQL's columnar engine, without going to a database or Java:

```
from graphistry import n, e_undirected, is_in

g2 = g1.gfql([
  n({'user': 'Biden'}),
  e_undirected(),
  n(name='bridge'),
  e_undirected(),
  n({'user': is_in(['Trump', 'Obama'])})
])

print('# bridges', len(g2._nodes[g2._nodes.bridge]))
g2.plot()
```

Enable GFQL's optional automatic GPU acceleration for 43X+ speedups:

```
# Switch from Pandas CPU dataframes to RAPIDS GPU dataframes
import cudf
g2 = g1.edges(lambda g: cudf.DataFrame(g._edges))
# GFQL will automaticallly run on a GPU
g3 = g2.gfql([n(), e(hops=3), n()])
g3.plot()
```

- https://spark.apache.org//https://databricks.com/ (https://github.com/graphistry/pygraphistry/blob/master/demos/de notebook-dashboard.ipynb, https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databases_apis/da notebook-dashboard.dbc)

```
#optional but recommended
spark.conf.set("spark.sql.execution.arrow.enabled", "true")

edges_df = (
    spark.read.format('json').
      load('/databricks-datasets/iot/iot_devices.json')
      .sample(fraction=0.1)
)
g = graphistry.edges(edges_df, 'device_name', 'cn')

#notebook
displayHTML(g.plot())

#dashboard: pick size of choice
displayHTML(
  g.settings(url_params={'splashAfter': 'false'})
    .plot(override_html_style="""
      width: 50em;
      height: 50em;
    """)
)
```

- GPU https://www.rapids.ai cudf

```
edges = cudf.read_csv('facebook_combined.txt', sep=' ', names=['src', 'dst'])
graphistry.edges(edges, 'src', 'dst').plot()
```

- GPU https://www.rapids.ai cuML

```
g = graphistry.nodes(cudf.read_csv('rows.csv'))
g = graphistry.nodes(G)
g.umap(engine='cuml',metric='euclidean').plot()
```

- GPU https://www.rapids.ai cugraph (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_database

```
g = graphistry.from_cugraph(G)
g2 = g.compute_cugraph('pagerank')
g3 = g2.layout_cugraph('force_atlas2')
g3.plot()
G3 = g.to_cugraph()
```

- https://arrow.apache.org/

```
 edges = pa.Table.from_pandas(pd.read_csv('facebook_combined.txt', sep=' ', names=[
↪'src', 'dst']))
 graphistry.edges(edges, 'src', 'dst').plot()
```

The `graphistry.cypher(...)` examples below are the remote database Cypher path. For Cypher syntax through GFQL on a bound graph, use `g.gfql("MATCH ...")`. For remote GFQL execution, use `g.gfql_remote([...])`. See the *GFQL Cypher guide*.

- http://neo4j.com (external link) (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databases_ap

```
NEO4J_CREDS = {'uri': 'bolt://my.site.ngo:7687', 'auth': ('neo4j', 'mypwd')}
graphistry.register(bolt=NEO4J_CREDS)
graphistry.cypher("MATCH (n1)-[r1]->(n2) RETURN n1, r1, n2 LIMIT 1000").plot()
```

```
graphistry.cypher("CALL db.schema()").plot()
```

```
from neo4j import GraphDatabase, Driver
graphistry.register(bolt=GraphDatabase.driver(**NEO4J_CREDS))
g = graphistry.cypher("""
  MATCH (a)-[p:PAYMENT]->(b)
  WHERE p.USD > 7000 AND p.USD < 10000
  RETURN a, p, b
  LIMIT 100000""")
print(g._edges.columns)
g.plot()
```

- https://memgraph.com/ (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databases_apis/mem

```
from neo4j import GraphDatabase
MEMGRAPH = {
'uri': "bolt://localhost:7687",
'auth': (" ", " ")
}
graphistry.register(bolt=MEMGRAPH)
```

```
driver = GraphDatabase.driver(**MEMGRAPH)
with driver.session() as session:
session.run("""
  CREATE (per1:Person {id: 1, name: "Julie"})
  CREATE (fil2:File {id: 2, name: "welcome_to_memgraph.txt"})
  CREATE (per1)-[:HAS_ACCESS_TO]->(fil2) """)
g = graphistry.cypher("""
   MATCH (node1)-[connection]-(node2)
   RETURN node1, connection, node2;""")
g.plot()
```

- https://azure.microsoft.com/en-us/services/cosmos-db/

```
# pip install --user gremlinpython
# Options in help(graphistry.cosmos)
g = graphistry.cosmos(
    COSMOS_ACCOUNT='',
    COSMOS_DB='',
    COSMOS_CONTAINER='',
    COSMOS_PRIMARY_KEY=''
)
g2 = g.gremlin('g.E().sample(10000)').fetch_nodes()
g2.plot()
```

- https://aws.amazon.com/neptune/ (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databases_
https://aws.amazon.com/blogs/database/enabling-low-code-graph-data-apps-with-amazon-neptune-and-graphistry/)

```
# pip install --user gremlinpython==3.4.10
#   - Deploy tips: https://github.com/graphistry/graph-app-kit/blob/master/docs/
↪neptune.md
#   - Versioning tips: https://gist.github.com/lmeyerov/
↪459f6f0360abea787909c7c8c8f04cee
#   - Login options in help(graphistry.neptune)
g = graphistry.neptune(endpoint='wss://zzz:8182/gremlin')
g2 = g.gremlin('g.E().limit(100)').fetch_nodes()
g2.plot()
```

- https://tigergraph.com (external link) (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databas

```
g = graphistry.tigergraph(protocol='https', ...)
g2 = g.gsql("...", {'edges': '@@eList'})
g2.plot()
print('# edges', len(g2._edges))
```

```
g.endpoint('my_fn', {'arg': 'val'}, {'edges': '@@eList'}).plot()
```

- http://igraph.org

```
edges = pd.read_csv('facebook_combined.txt', sep=' ', names=['src', 'dst'])
g_a = graphistry.edges(edges, 'src', 'dst')
g_b = g_a.layout_igraph('sugiyama', directed=True)  # directed: for to_igraph
g_b.compute_igraph('pagerank', params={'damping': 0.85}).plot()  #params: for layout
```

(continues on next page)

(continued from previous page)

```
ig = igraph.read('facebook_combined.txt', format='edgelist', directed=False)
g = graphistry.from_igraph(ig)  # full conversion
g.plot()

ig2 = g.to_igraph()
ig2.vs['spinglass'] = ig2.community_spinglass(spins=3).membership
# selective column updates: preserve g._edges; merge 1 attribute from ig into g._
↪nodes
g2 = g.from_igraph(ig2, load_edges=False, node_attributes=[g._node, 'spinglass'])
```

- https://networkx.github.io (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databases_apis/ne

```
graph = networkx.read_edgelist('facebook_combined.txt')
g = graphistry.bind(source='src', destination='dst', node='nodeid').from_
↪networkx(graph)
g2 = g.compute_networkx('degree_centrality', out_col='degree_score', directed=False)
g2.plot()
```

- https://github.com/pnnl/HyperNetX (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databas

```
hg.hypernetx_to_graphistry_nodes(H).plot()
```

```
hg.hypernetx_to_graphistry_bipartite(H.dual()).plot()
```

- https://www.splunk.com (external link) (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_datab

```
df = splunkToPandas("index=netflow bytes > 100000 | head 100000", {})
graphistry.edges(df, 'src_ip', 'dest_ip').plot()
```

- https://www.nodexl.com (external link) (https://github.com/graphistry/pygraphistry/blob/master/demos/demos_datab

```
graphistry.nodexl('/my/file.xls').plot()
```

```
graphistry.nodexl('https://file.xls').plot()
```

```
graphistry.nodexl('https://file.xls', 'twitter').plot()
graphistry.nodexl('https://file.xls', verbose=True).plot()
graphistry.nodexl('https://file.xls', engine='xlsxwriter').plot()
graphistry.nodexl('https://file.xls')._nodes
```

10.9.4 Access control for sharing visualizations

Graphistry supports flexible sharing permissions that are similar to Google documents and Dropbox links

By default, visualizations are publicly viewable by anyone with the URL (that is unguessable & unlisted), and only editable by their owner.

- Private-only: You can globally default uploads to private:

```
graphistry.privacy()  # graphistry.privacy(mode='private')
```

- Organizations: You can login with an organization and share only within it

```
graphistry.register(api=3, username='...', password='...', org_name='my-org123')
graphistry.privacy(mode='organization')
```

- Invitees: You can share access to specify users, and optionally, even email them invites

```
VIEW = "10"
EDIT = "20"
graphistry.privacy(
  mode='private',
  invited_users=[
    {"email": "friend1@site1.com", "action": VIEW},
    {"email": "friend2@site2.com", "action": EDIT}
  ],
  notify=True)
```

- Per-visualization: You can choose different rules for global defaults vs. for specific visualizations

```
graphistry.privacy(invited_users=[...])
g = graphistry.hypergraph(pd.read_csv('...'))['graph']
g.privacy(notify=True).plot()
```

See additional examples in the https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_fe

10.9.5 Tutorial Start: Les Misérables

Let's visualize relationships between the characters in http://en.wikipedia.org/wiki/Les_Mis%C3%A9rables. For this example, we'll choose http://pandas.pydata.org to wrangle data and http://igraph.org to run a community detection algorithm. You can https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/simple/MarvelTutorial.ipynb the Jupyter notebook containing this example.

Our https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/data/lesmiserables.csv that looks like this:

source	target	value
Cravatte	Myriel	1
Valjean	Mme.Magloire	3
Valjean	Mlle.Baptistine	3

Source and *target* are character names, and the *value* column counts the number of time they meet. Parsing is a one-liner with Pandas:

```
import pandas
links = pandas.read_csv('./lesmiserables.csv')
```

10.9.5.1 Quick Visualization

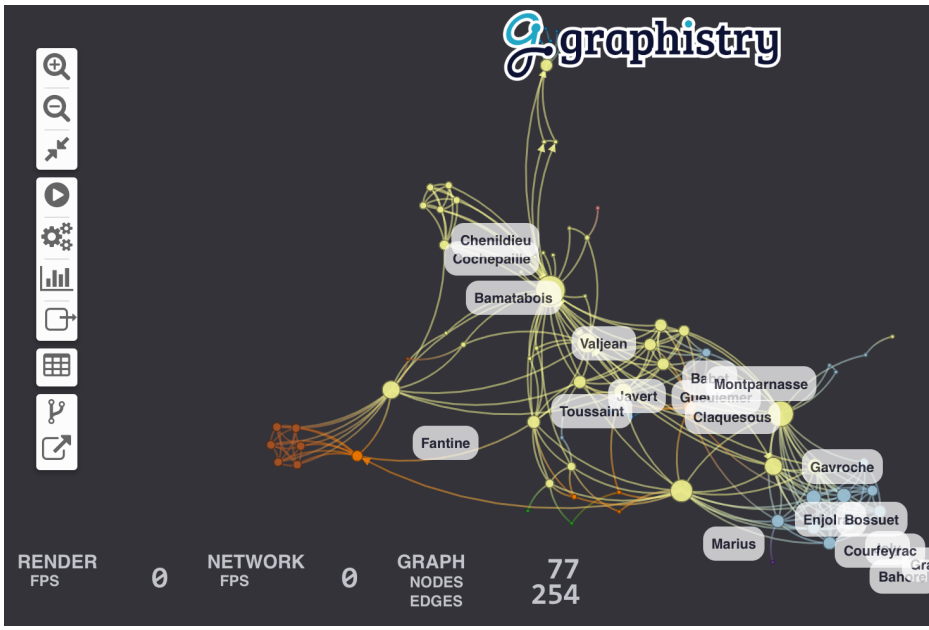
If you already have graph-like data, use this step. Otherwise, try the https://github.com/graphistry/pygraphistry/blob/master/demos/demos_by_use_case/logs/malware-hypergraph/Malware%20Hypergraph.ipynb for creating graphs from rows of data (logs, samples, records, ...).

PyGraphistry can plot graphs directly from Pandas data frames, Arrow tables, cuGraph GPU data frames, igraph graphs, or NetworkX graphs. Calling *plot* uploads the data to our visualization servers and return an URL to an embeddable webpage containing the visualization.

To define the graph, we `bind` *source* and *destination* to the columns indicating the start and end nodes of each edges:

```
import graphistry
graphistry.register(api=3, username='YOUR_ACCOUNT_HERE', password='YOUR_PASSWORD_HERE')

g = graphistry.bind(source="source", destination="target")
g.edges(links).plot()
```

You should see a beautiful graph like this one:



10.9.6 Visual encodings & settings

10.9.6.1 Adding Labels

Let's add labels to edges in order to show how many times each pair of characters met. We create a new column called *label* in edge table *links* that contains the text of the label and we bind *edge_label* to it.

```
links["label"] = links.value.map(lambda v: "#Meetings: %d" % v)
g = g.bind(edge_title="label")
g.edges(links).plot()
```

10.9.6.2 Controlling Node Title, Size, Color, and Position

Let's size nodes based on their http://en.wikipedia.org/wiki/PageRank score and color them using their https://en.wikipedia.org/wiki/Community_structure.

10.9.6.3 Warmup: igraph for computing statistics

http://igraph.org/python/ already has these algorithms implemented for us for small graphs. (See our cuGraph examples for big graphs.) If igraph is not already installed, fetch it with `pip install igraph`.

We start by converting our edge dateframe into an igraph. The plotter can do the conversion for us using the *source* and *destination* bindings. Then we compute two new node attributes (*pagerank* & *community*).

```
g = g.compute_igraph('pagerank', directed=True, params={'damping': 0.85}).compute_igraph(
↪'community_infomap')
```

The algorithm names `'pagerank'` and `'community_infomap'` correspond to method names of https://igraph.org/python/api/latest/igraph.Graph.html. Likewise, optional `params={...}` allow specifying additional parameters.

10.9.6.4 Bind node data to visual node attributes

We can then bind the node `community` and `pagerank` columns to visualization attributes:

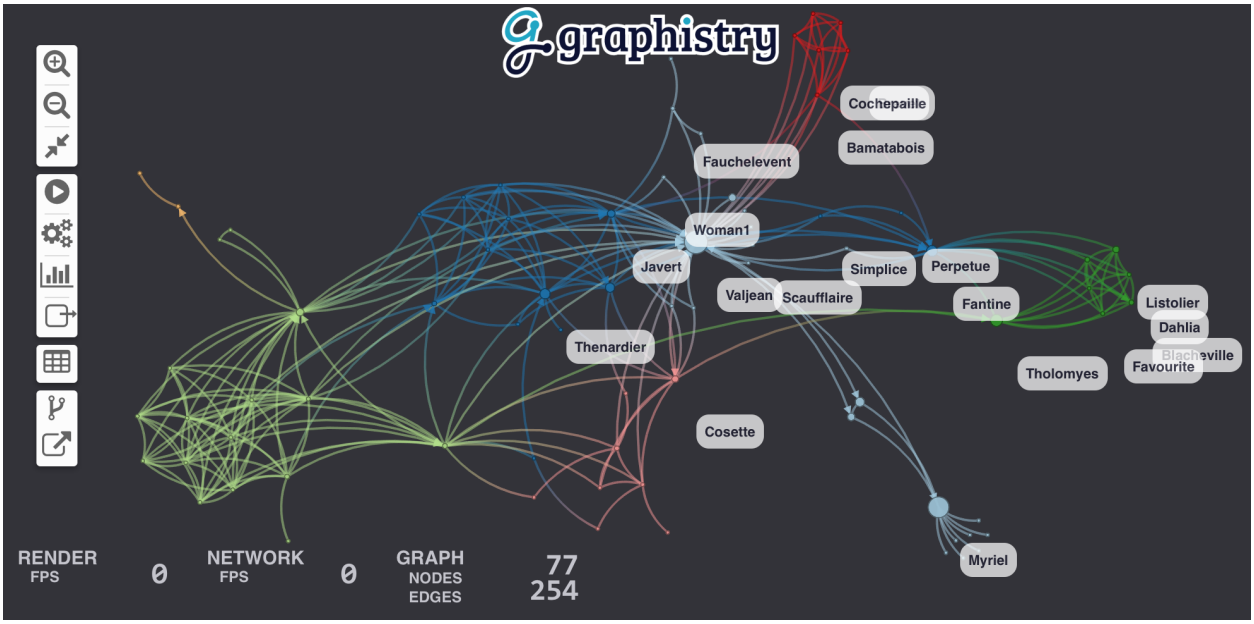
```
g.bind(point_color='community', point_size='pagerank').plot()
```

See the https://hub.graphistry.com/docs/api/2/rest/upload/colors/#extendedpalette2 for specifying color values by using built-in ColorBrewer palettes (`int32`) or custom RGB values (`int64`).

To control the position, we can add `.bind(point_x='colA', point_y='colB').settings(url_params={'play': 0})` (https://github.com/graphistry/pygraphistry/tree/master/demos/more_examples/gra and https://hub.graphistry.com/docs/api/1/rest/url/#urloptions]).

You may also want to bind `point_title`: `.bind(point_title='colA')`.

For more in-depth examples, check out the tutorials on https://github.com/graphistry/pygraphistry/tree/master/demos/more_ colors.ipynb and https://github.com/graphistry/pygraphistry/tree/master/demos/more_examples/graphistry_features/encodi sizes.ipynb.



10.9.6.5 Add edge colors and weights

By default, edges get colored as a gradient between their source/destination node colors. You can override this by setting `.bind(edge_color='colA')`, similar to how node colors function. (https://hub.graphistry.com/docs/api/2/rest/upload/colors/#extendedpalette2.)

Similarly, you can bind the edge weight, where higher weights cause nodes to cluster closer together: `.bind(edge_weight='colA')`. https://github.com/graphistry/pygraphistry/tree/master/demos/more_examples/graphistry_features/edge-weights.ipynb.

For more in-depth examples, check out the tutorials on https://github.com/graphistry/pygraphistry/tree/master/demos/more_e colors.ipynb and *weighted clustering*.

10.9.6.6 More advanced color and size controls

You may want more controls like using gradients or maping specific values:

```
g.encode_edge_color('int_col')  # int32 or int64
g.encode_edge_color('time_col', ["blue", "red"], as_continuous=True)
g.encode_edge_color('type', as_categorical=True,
  categorical_mapping={"cat": "red", "sheep": "blue"}, default_mapping='#CCC')
g.encode_edge_color('brand',
  categorical_mapping={'toyota': 'red', 'ford': 'blue'},
  default_mapping='#CCC')
g.encode_point_size('numeric_col')
g.encode_point_size('criticality',
  categorical_mapping={'critical': 200, 'ok': 100},
  default_mapping=50)
g.encode_point_color('int_col')  # int32 or int64
g.encode_point_color('time_col', ["blue", "red"], as_continuous=True)
g.encode_point_color('type', as_categorical=True,
  categorical_mapping={"cat": "red", "sheep": "blue"}, default_mapping='#CCC')
```

For subset-based coloring and conditional styling across multiple encodings, use Collections via `g.collections(...)` with GFQL AST helpers. See *Layout settings* and the *Collections tutorial notebook*.

For more in-depth examples, check out the tutorials on https://github.com/graphistry/pygraphistry/tree/master/demos/more_colors.ipynb.

10.9.6.7 Custom icons and badges

You can add a main icon and multiple peripherary badges to provide more visual information. Use column `type` for the icon type to appear visually in the legend. The glyph system supports text, icons, flags, and images, as well as multiple mapping and style controls.

Main icon

```
g.encode_point_icon(
  'some_column',
  shape="circle", #clip excess
  categorical_mapping={
      'macbook': 'laptop', #https://fontawesome.com/v4.7.0/icons/
      'Canada': 'flag-icon-ca', #ISO3611-Alpha-2: https://github.com/datasets/country-
↪codes/blob/master/data/country-codes.csv
      'embedded_smile': 'data:svg...',
      'external_logo': 'http://..../img.png'
  },
  default_mapping="question")
g.encode_point_icon(
  'another_column',
  continuous_binning=[
    [20, 'info'],
    [80, 'exclamation-circle'],
    [None, 'exclamation-triangle']
  ]
)
g.encode_point_icon(
  'another_column',
  as_text=True,
  categorical_mapping={
    'Canada': 'CA',
    'United States': 'US'
    }
)
```

For more in-depth examples, check out the tutorials on https://github.com/graphistry/pygraphistry/tree/master/demos/more_icons.ipynb.

Badges

```
# see icons examples for mappings and glyphs
g.encode_point_badge('another_column', 'TopRight', categorical_mapping=...)

g.encode_point_badge('another_column', 'TopRight', categorical_mapping=...,
  shape="circle",
  border={'width': 2, 'color': 'white', 'stroke': 'solid'},
  color={'mapping': {'categorical': {'fixed': {}, 'other': 'white'}}},
  bg={'color': {'mapping': {'continuous': {'bins': [], 'other': 'black'}}}})
```

For more in-depth examples, check out the tutorials on https://github.com/graphistry/pygraphistry/tree/master/demos/more_e
badges.ipynb.

Axes

For more automated use, see the section on radial layouts below.

Radial axes support three coloring types (`'external'`, `'internal'`, and `'space'`) and optional labels:

```
 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"}
])
```

Horizontal axis support optional labels and ranges:

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

Radial axis are generally used with radial positioning:

```
g2 = (g
  .nodes(
    g._nodes.assign(
      x = 1 + (g._nodes['ring']) * g._nodes['n'].apply(math.cos),
      y = 1 + (g._nodes['ring']) * g._nodes['n'].apply(math.sin)
  )).settings(url_params={'lockedR': 'true', 'play': 1000})
```

Horizontal axis are often used with pinned y and free x positions:

```
g2 = (g
  .nodes(
    g._nodes.assign(
```

(continues on next page)

(continued from previous page)

```
      y = 50 * g._nodes['level'])
  )).settings(url_params={'lockedY': 'true', 'play': 1000})
```

10.9.6.8 Theming

You can customize several style options to match your theme:

```
g.addStyle(bg={'color': 'red'})
g.addStyle(bg={
  'color': '#333',
  'gradient': {
    'kind': 'radial',
    'stops': [ ["rgba(255,255,255, 0.1)", "10%", "rgba(0,0,0,0)", "20%"] ]}})
g.addStyle(bg={'image': {'url': 'http://site.com/cool.png', 'blendMode': 'multiply'}})
g.addStyle(fg={'blendMode': 'color-burn'})
g.addStyle(page={'title': 'My site'})
g.addStyle(page={'favicon': 'http://site.com/favicon.ico'})
g.addStyle(logo={'url': 'http://www.site.com/transparent_logo.png'})
g.addStyle(logo={
  'url': 'http://www.site.com/transparent_logo.png',
  'dimensions': {'maxHeight': 200, 'maxWidth': 200},
  'style': {'opacity': 0.5}
})
```

10.9.6.9 Control render settings

```
g = graphistry.edges(pd.DataFrame({'s': ['a', 'b', 'b'], 'd': ['b', 'c', 'd']}))
g2 = g.scene_settings(
  #hide menus
  menu=False,
  info=False,
  #tweak graph
  show_arrows=False,
  point_size=1.0,  # 0.1-10.0 range, log scale (1.0 → "50" in UI)
  edge_curvature=0.0,  # 0.0-1.0 range
  edge_opacity=0.5,  # 0.0-1.0 range
  point_opacity=0.9  # 0.0-1.0 range
).plot()
```

With `pip install graphistry[igraph]`, you can also use https://igraph.org/python/doc/api/igraph.Graph.html#layout:

```
g.layout_igraph('sugiyama').plot()
g.layout_igraph('sugiyama', directed=True, params={}).plot()
```

10.9.7 Visualization Embedding

- **Embeddable:** Drop live views into your web dashboards and apps (and go further with https://hub.graphistry.com/docs):

```
iframe_url = g.plot(render=False)
print(f'<iframe src="{ iframe_url }"></iframe>')
```

10.9.8 Layout

10.9.8.1 Hierarchical layouts: Tree and radial

A hierachical view via horizontal or vertical trees, or radial. Graph data may also be presented using these layouts.

Tree

Tip: Also try `g.layout_graphviz("dot")` and `"circo"`

```
g = graphistry.edges(pd.DataFrame({'s': ['a', 'b', 'b'], 'd': ['b', 'c', 'd']}))

g2a = g.tree_layout()
g2b = g2.tree_layout(allow_cycles=False, remove_self_loops=False, vertical=False)
g2c = g2.tree_layout(ascending=False, level_align='center')
g2d = g2.tree_layout(level_sort_values_by=['type', 'degree'], level_sort_values_by_
↪ascending=False)

g3a = g2a.layout_settings(locked_r=True, play=1000)
g3b = g2a.layout_settings(locked_y=True, play=0)
g3c = g2a.layout_settings(locked_x=True)

g4 = g2.tree_layout().rotate(90)
```

To use with non-tree data, e.g., graphs with cycles, we recommend computing a tree such as via a minimum spanning tree, and then using that achieved layout with this algorithm. Alternatively, the radial layouts may more naturally support your graph.

Radial

A hierarchical view via radial rings that may be more space-efficient and aesthetic than the equivalent tree layout

Supports time-based, continuous, and categorical modes:

Time-based

Use when the value column defining the ring order is a time column. See https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/layout_time_ring.ipyn

```
g.time_ring_layout().plot()  # finds a time column and infers all settings

g.time_ring_layout(
  time_col='my_node_time_col',
  num_rings=20,
  time_start=np.datetime64('2014-01-22'),
  time_end=np.datetime64('2015-01-22'),
  time_unit= 'Y',  # s, m, h, D, W, M, Y, C
  min_r=100.0,  # smallest ring radius
  max_r=1000.0,  # biggest ring radius
  reverse=False,
  #format_axis: Optional[Callable[[List[Dict]], List[Dict]]] = None,
  #format_label: Optional[Callable[[np.datetime64, int, np.timedelta64], str]] = None,
  #play_ms: int = 2000,
  #engine='auto'  # 'auto', 'pandas', 'cudf'
).plot()
```

> **Tip:** When building GFQL JSON payloads, send `time_start` / `time_end` as ISO-8601 strings (for example, `"2014-01-22T00:00:00"`). The runtime converts them to `numpy.datetime64` before computing the layout, matching the behavior shown here with native `np.datetime64` inputs.

Continuous

Use when the value column defining the ring order is a continuous number, like distance or amount. See https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/layout_continuous_ri

```
g.ring_continuous_layout()  # find a numeric column and infers all settings

g.ring_continuous_layout(
  ring_col='my_numeric_col',
  #v_start=  # first ring at this value
  #v_end=  # last ring at this value
  #v_step=  # distance between rings in the value domain
  min_r=100.0,  # smallest ring radius
  max_r=1000.0, # biggest ring radius
  normalize_ring_col=True,  # remap [v_start,v_end] to [min_r,max_r]
  num_rings=20,
  ring_step=100,

  #Control axis labels and styles
  #axis: Optional[Union[Dict[float,str],List[str]]] = None,
  #format_axis: Optional[Callable[[List[Dict]], List[Dict]]] = None,
  #format_labels: Optional[Callable[[float, int, float], str]] = None,

  reverse=False,
  play_ms=0,
  #engine='auto',  # 'auto', 'pandas', 'cudf'
)
```

Categorical

Use when the value column defining the ring order is a categorical value, such as a name or ID. See https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/layout_categorical_ri

```
g.ring_categorical_layout('my_categorical_col')  # infers all settings

g.ring_categorical_layout(
  ring_col='my_numeric_col',
  order=['col1', 'my_col2'],
  drop_empty=True,  # remove unpopulated rings
  combine_unhandled=False,  # Put values not covered by order into one ring Other vs a␣
↪ring per unique value
  append_unhandled=True,  # Append vs prepend
  min_r=100.0,  # smallest ring radius
  max_r=1000.0, # biggest ring radius

  #Control axis labels and styles
  #axis: Optional[Dict[Any,str]] = None,
  #format_axis: Optional[Callable[[List[Dict]], List[Dict]]] = None,
  #format_labels: Optional[Callable[[Any, int, float], str]] = None,

  reverse=False,
  play_ms=0,
  #engine='auto',  # 'auto', 'pandas', 'cudf'
)
```

10.9.8.2 Modularity weighted

Weight edges by community membership to emphasize community structure. See https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/layout_modularity_w

```
g.modularity_weighted_layout().plot()
g.modularity_weighted_layout('my_community_col').plot()
g.modularity_weighted_layout(
  community_alg='louvain',
  engine='cudf',
  same_community_weight=2.0,
  cross_community_weight=0.3,
  edge_influence=2.0
).plot()
```

10.9.8.3 Group-in-a-box

https://ieeexplore.ieee.org/document/6113135 with igraph/pandas and cugraph/cudf implementations:

```
g.group_in_a_box_layout().plot()
g.group_in_a_box_layout(
  partition_alg='ecg',  # see igraph/cugraph algs
  #partition_key='some_col',  # use existing col
  #layout_alg='circle',  # see igraph/cugraph algs
```

(continues on next page)

(continued from previous page)

```
  #x, y, w, h
  #encode_colors=False,
  #colors=['#FFF', '#FF0', ...]
  engine='cudf'
).plot()
```

10.9.9 Compute

10.9.9.1 Transforms

The below methods let you quickly manipulate graphs directly and with dataframe methods: Search, pattern mine, transform, and more:

```
from graphistry import n, e_forward, e_reverse, e_undirected, is_in
g = (graphistry
  .edges(pd.DataFrame({
    's': ['a', 'b'],
    'd': ['b', 'c'],
    'k1': ['x', 'y']
  }))
  .nodes(pd.DataFrame({
    'n': ['a', 'b', 'c'],
    'k2': [0, 2, 4, 6]
  })
)

g2 = graphistry.hypergraph(g._edges, ['s', 'd', 'k1'])['graph']
g2.plot() # nodes are values from cols s, d, k1

(g
  .materialize_nodes()
  .get_degrees()
  .get_indegrees()
  .get_outdegrees()
  .pipe(lambda g2: g2.nodes(g2._nodes.assign(t=x))) # transform
  .filter_edges_by_dict({"k1": "x"})
  .filter_nodes_by_dict({"k2": 4})
  .prune_self_edges()
  .hop( # filter to subgraph
    #almost all optional
    direction='forward', # 'reverse', 'undirected'
    hops=2, # number (1..n hops, inclusive) or None if to_fixed_point
    to_fixed_point=False,
    # optional traversal range + labeling
    min_hops=1,  # inclusive lower bound (defaults to 1 unless hops==0)
    max_hops=3,  # inclusive upper bound; defaults to hops
    output_min_hops=None,  # optional output slice lower bound (post-filter; defaults␣
↪keep early hops)
    output_max_hops=None,  # optional output slice upper bound (post-filter; defaults to␣
↪max_hops)
```

(continues on next page)

(continued from previous page)

```
    label_node_hops='hop',  # write first hop step each node is reached (omit/None to␣
↪skip)
    label_edge_hops='edge_hop',  # hop step for each traversed edge
    label_seeds=True,  # also tag starting seeds as hop 0 when labeling

    #every edge source node must match these
    source_node_match={"k2": 0, "k3": is_in(['a', 'b', 3, 4])},
    source_node_query='k2 == 0',

    #every edge must match these
    edge_match={"k1": "x"},
    edge_query='k1 == "x"',

    #every edge destination node must match these
    destination_node_match={"k2": 2},
    destination_node_query='k2 == 2 or k2 == 4',
  )
  .gfql([ # filter to subgraph with GFQL pattern matching
    n(),
    n({'k2': 0, "m": 'ok'}), #specific values
    n({'type': is_in(["type1", "type2"])}), #multiple valid values
    n(query='k2 == 0 or k2 == 4'), #dataframe query
    n(name="start"), # add column 'start':bool
    e_forward({'k1': 'x'}, hops=1), # same API as hop()
    e_undirected(name='second_edge'),
    e_reverse(
      {'k1': 'x'}, # edge property match
      hops=2, # 1 to 2 hops
      #same API as hop()
      source_node_match={"k2": 2},
      source_node_query='k2 == 2 or k2 == 4',
      edge_match={"k1": "x"},
      edge_query='k1 == "x"',
      destination_node_match={"k2": 0},
      destination_node_query='k2 == 0')
  ])
  # replace as one node the node w/ given id + transitively connected nodes w/ col=attr
  .collapse(node='some_id', column='some_col', attribute='some val')
```

Both `hop()` and `gfql()` (GFQL) match dictionary expressions support dataframe series *predicates*. The above examples show `is_in([x, y, z, ...])`. Additional predicates include:

- categorical: is_in, duplicated
- temporal: is_month_start, is_month_end, is_quarter_start, is_quarter_end, is_year_start, is_year_end
- numeric: gt, lt, ge, le, eq, ne, between, isna, notna
- string: contains, startswith, endswith, match, isnumeric, isalpha, isdigit, islower, isupper, isspace, isalnum, isdecimal, istitle, isnull, notnull

Both `hop()` and `gfql()` will run on GPUs when passing in RAPIDS dataframes. Specify parameter `engine='cudf'` to be sure.

10.9.9.2 Table to graph

```
df = pd.read_csv('events.csv')
hg = graphistry.hypergraph(df, ['user', 'email', 'org'], direct=True)
g = hg['graph']  # g._edges: | src, dst, user, email, org, time, ... |
g.plot()
```

```
hg = graphistry.hypergraph(
  df,
  ['from_user', 'to_user', 'email', 'org'],
  direct=True,
  opts={

   # when direct=True, can define src -> [ dst1, dst2, ...] edges
  'EDGES': {
    'org': ['from_user'], # org->from_user
    'from_user': ['email', 'to_user'],  #from_user->email, from_user->to_user
  },

  'CATEGORIES': {
    # determine which columns share the same namespace for node generation:
    # - if user 'louie' is both a from_user and to_user, show as 1 node
    # - if a user & org are both named 'louie', they will appear as 2 different nodes
    'user': ['from_user', 'to_user']
  }
})
g = hg['graph']
g.plot()
```

10.9.9.3 Generate node table

```
g = graphistry.edges(pd.DataFrame({'s': ['a', 'b'], 'd': ['b', 'c']}))
g2 = g.materialize_nodes()
g2._nodes  # pd.DataFrame({'id': ['a', 'b', 'c']})
```

10.9.9.4 Compute degrees

```
g = graphistry.edges(pd.DataFrame({'s': ['a', 'b'], 'd': ['b', 'c']}))
g2 = g.get_degrees()
g2._nodes  # pd.DataFrame({
           #  'id': ['a', 'b', 'c'],
           #  'degree_in': [0, 1, 1],
           #  'degree_out': [1, 1, 0],
           #  'degree': [1, 1, 1]
           #})
```

See also `get_indegrees()` and `get_outdegrees()`

10.9.9.5 Graph pattern matching

PyGraphistry supports GFQL, its PyData-native variant of the popular Cypher graph query language, meaning you can do graph pattern matching directly from Pandas dataframes without installing a database or Java

See also https://github.com/graphistry/pygraphistry/tree/master/demos/more_examples/graphistry_features/hop_and_chai and the CPU/GPU https://github.com/graphistry/pygraphistry/tree/master/demos/gfql/benchmark_hops_cpu_gpu.ipynb

Traverse within a graph, or expand one graph against another

Simple node and edge filtering via `filter_edges_by_dict()` and `filter_nodes_by_dict()`:

```
g = graphistry.edges(pd.read_csv('data.csv'), 's', 'd')
g2 = g.materialize_nodes()

g3 = g.filter_edges_by_dict({"v": 1, "b": True})
g4 = g.filter_nodes_by_dict({"v2": 1, "b2": True})
```

Method `.hop()` enables slightly more complicated edge filters:

```

from graphistry import is_in, gt

# (a)-[{"v": 1, "type": "z"}]->(b) based on g
g2b = g2.hop(
  source_node_match={g2._node: "a"},
  edge_match={"v": 1, "type": "z"},
  destination_node_match={g2._node: "b"})
g2b = g2.hop(
  source_node_query='n == "a"',
  edge_query='v == 1 and type == "z"',
  destination_node_query='n == "b"')

# (a {x in [1,2] and y > 3})-[e]->(b) based on g
g2c = g2.hop(
  source_node_match={
    g2._node: "a",
    "x": is_in([1,2]),
    "y": gt(3)
  },
  destination_node_match={g2._node: "b"})
)

# (a or b)-[1 to 8 hops]->(anynode), based on graph g2
g3 = g2.hop(pd.DataFrame({g2._node: ['a', 'b']}), hops=8)

# Bounded hops with labels and sliced outputs
g4 = g2.hop(
  pd.DataFrame({g2._node: ['a']}),
  min_hops=2,
  max_hops=3,
  output_min_hops=2,
  output_max_hops=3,
  label_node_hops='hop',
```

(continues on next page)

(continued from previous page)

```
  label_edge_hops='edge_hop',
  label_seeds=True
)
g4._nodes[['node', 'hop']]

# (a or b)-[1 to 8 hops]->(anynode), based on graph g2
g3 = g2.hop(pd.DataFrame({g2._node: is_in(['a', 'b'])}), hops=8)

# (c)<-[any number of hops]-(any node), based on graph g3
# Note multihop matches check source/destination/edge match/query predicates
# against every encountered edge for it to be included
g4 = g3.hop(source_node_match={"node": "c"}, direction='reverse', to_fixed_point=True)

# (c)-[incoming or outgoing edge]-(any node),
# for c in g4 with expansions against nodes/edges in g2
g5 = g2.hop(pd.DataFrame({g4._node: g4[g4._node]}), hops=1, direction='undirected')

g5.plot()
```

Rich compound patterns are enabled via `.gfql()`:

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

g2.gfql([ n() ])
g2.gfql([ n({"x": 1, "y": True}) ]),
g2.gfql([ n(query='x == 1 and y == True') ]),
g2.gfql([ n({"z": is_in([1,2,4,'z'])}) ]), # multiple valid values
g2.gfql([ e_forward({"type": "x"}, hops=2) ]) # simple multi-hop
g3 = g2.gfql([
  n(name="start"),  # tag node matches
  e_forward(hops=3),
  e_forward(name="final_edge"), # tag edge matches
  n(name="end")
])
g2.gfql(n(), e_forward(), n(), e_reverse(), n())  # rich shapes
print('# end nodes: ', len(g3._nodes[ g3._nodes.end ]))
print('# end edges: ', len(g3._edges[ g3._edges.final_edge ]))
```

See table above for more predicates like `is_in()` and `gt()`

Queries can be serialized and deserialized, such as for saving and remote execution:

```
from graphistry.compute.chain import Chain

pattern = Chain([n(), e(), n()])
pattern_json = pattern.to_json()
pattern2 = Chain.from_json(pattern_json)
g.gfql(pattern2).plot()
```

Benefit from automatic GPU acceleration by passing in GPU dataframes:

```
import cudf
```

(continues on next page)

(continued from previous page)

```
g1 = graphistry.edges(cudf.read_csv('data.csv'), 's', 'd')
g2 = g1.gfql(..., engine='cudf')
```

The parameter `engine` is optional, defaulting to `'auto'`.

10.9.9.6 Pipelining

```
def capitalize(df, col):
  df2 = df.copy()
  df2[col] df[col].str.capitalize()
  return df2

g
  .cypher('MATCH (a)-[e]->(b) RETURN a, e, b')
  .nodes(lambda g: capitalize(g._nodes, 'nTitle'))
  .edges(capitalize, None, None, 'eTitle'),
  .pipe(lambda g: g.nodes(g._nodes.pipe(capitalize, 'nTitle')))
```

10.9.9.7 Removing nodes

```
g = graphistry.edges(pd.DataFrame({'s': ['a', 'b', 'c'], 'd': ['b', 'c', 'a']}))
g2 = g.drop_nodes(['c'])  # drops node c, edge c->a, edge b->c,
```

10.9.9.8 Keeping nodes

```
# keep nodes [a,b,c] and edges [(a,b),(b,c)]
g2 = g.keep_nodes(['a, b, c'])
g2 = g.keep_nodes(pd.Series(['a, b, c']))
g2 = g.keep_nodes(cudf.Series(['a, b, c']))
```

10.9.9.9 Collapsing adjacent nodes with specific k=v matches

One col/val pair:

```
g2 = g.collapse(
  node='root_node_id',  # rooted traversal beginning
  column='some_col',  # column to inspect
  attribute='some val' # value match to collapse on if hit
)
assert len(g2._nodes) <= len(g._nodes)
```

Collapse for all possible vals in a column, and assuming a stable root node id:

```
g3 = g
for v in g._nodes['some_col'].unique():
  g3 = g3.collapse(node='root_node_id', column='some_col', attribute=v)
```

10.9.10 Graph AI in a single line of code

Graph autoML features including:

10.9.10.1 Generate features from raw data

Automatically and intelligently transform text, numbers, booleans, and other formats to AI-ready representations:

- Featurization

```
g = graphistry.nodes(df).featurize(kind='nodes', X=['col_1', ..., 'col_n'], y=[
↪'label', ..., 'other_targets'], ...)

print('X', g._node_features)
print('y', g._node_target)
```

- Set `kind='edges'` to featurize edges:

```
g = graphistry.edges(df, src, dst).featurize(kind='edges', X=['col_1', ..., 'col_n'],
↪ y=['label', ..., 'other_targets'], ...)
```

- Use generated features with both Graphistry and external libraries:

```
# graphistry
g = g.umap()  # UMAP, GNNs, use features if already provided, otherwise will compute

# other pydata libraries
X = g._node_features  # g._get_feature('nodes') or g.get_matrix()
y = g._node_target  # g._get_target('nodes') or g.get_matrix(target=True)
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor().fit(X, y)  # assumes train/test split
new_df = pandas.read_csv(...)  # mini batch
X_new, _ = g.transform(new_df, None, kind='nodes', return_graph=False)
preds = model.predict(X_new)
```

- Encode model definitions and compare models against each other

```
# graphistry
from graphistry.features import search_model, topic_model, ngrams_model, ModelDict,
↪ default_featurize_parameters, default_umap_parameters

g = graphistry.nodes(df)
g2 = g.umap(X=[..], y=[..], **search_model)

# set custom encoding model with any feature/umap/dbscan kwargs
new_model = ModelDict(message='encoding new model parameters is easy', **default_
↪featurize_parameters)
new_model.update(dict(
                  y=[...],
                  kind='edges',
                  model_name='sbert/cool_transformer_model',
                  use_scaler_target='kbins',
                  n_bins=11,
```

(continues on next page)

(continued from previous page)

```
                    strategy='normal'))
print(new_model)

g3 = g.umap(X=[..], **new_model)
# compare g2 vs g3 or add to different pipelines
```

See `help(g.featurize)` for more options

10.9.10.2 https://umap-learn.readthedocs.io/en/latest/, https://docs.rapids.ai/api/cuml/stable/api.html?highlight=uma

- Reduce dimensionality by plotting a similarity graph from feature vectors:

```
 # automatic feature engineering, UMAP
 g = graphistry.nodes(df).umap()

 # plot the similarity graph without any explicit edge_dataframe passed in -- it␣
↪is created during UMAP.
 g.plot()
```

- Apply a trained model to new data:

```
 new_df = pd.read_csv(...)
 embeddings, X_new, _ = g.transform_umap(new_df, None, kind='nodes', return_
↪graph=False)
```

- Infer a new graph from new data using the old umap coordinates to run inference without having to train a new umap model.

```
 new_df = pd.read_csv(...)
 g2 = g.transform_umap(new_df, return_graph=True)   # return_graph=True is default
 g2.plot()  #

 # or if you want the new minibatch to cluster to closest points in previous fit:
 g3 = g.transform_umap(new_df, return_graph=True, merge_policy=True)
 g3.plot()  # useful to see how new data connects to old -- play with `sample` and␣
↪`n_neighbors` to control how much of old to include
```

- UMAP supports many options, such as supervised mode, working on a subset of columns, and passing arguments to underlying `featurize()` and UMAP implementations (see `help(g.umap)`):

```
 g.umap(kind='nodes', X=['col_1', ..., 'col_n'], y=['label', ..., 'other_targets'],
↪ ...)
```

- `umap(engine="...")` supports multiple implementations. It defaults to using the GPU-accelerated `engine="cuml"` when a GPU is available, resulting in orders-of-magnitude speedups, and falls back to CPU processing via `engine="umap_learn"`.:

```
 g.umap(engine='cuml')
```

You can also featurize edges and UMAP them as we did above.

UMAP support is rapidly evolving, please contact the team directly or on Slack for additional discussions

See `help(g.umap)` for more options

10.9.10.3 https://docs.dgl.ai/en/0.6.x/index.html

- Graphistry adds bindings and automation to working with popular GNN models, currently focusing on DGL/PyTorch:

```
g = (graphistry
    .nodes(ndf)
    .edges(edf, src, dst)
    .build_gnn(
        X_nodes=['col_1', ..., 'col_n'], #columns from nodes_dataframe
        y_nodes=['label', ..., 'other_targets'],
        X_edges=['col_1_edge', ..., 'col_n_edge'], #columns from edges_dataframe
        y_edges=['label_edge', ..., 'other_targets_edge'],
        ...)
)
G = g.DGL_graph

from [your_training_pipeline] import train, model
# Train
g = graphistry.nodes(df).build_gnn(y_nodes='target')
G = g.DGL_graph
train(G, model)
# predict on new data
X_new, _ = g.transform(new_df, None, kind='nodes' or 'edges', return_graph=False) #␣
↪no targets
predictions = model.predict(G_new, X_new)
```

Like `g.umap()`, GNN layers automate feature engineering (`.featurize()`)

See `help(g.build_gnn)` for options.

GNN support is rapidly evolving, please contact the team directly or on Slack for additional discussions

10.9.10.4 https://www.sbert.net/examples/applications/semantic-search/README.html

- Search textual data semantically and see the resulting graph:

```
  ndf = pd.read_csv(nodes.csv)
  edf = pd.read_csv(edges.csv)

  g = graphistry.nodes(ndf, 'node').edges(edf, 'src', 'dst')

  g2 = g.featurize(X = ['text_col_1', .., 'text_col_n'], kind='nodes',
                   min_words = 0,  # forces all named columns as textual ones
                   #encode text as paraphrase embeddings, supports any sbert model
                   model_name = "paraphrase-MiniLM-L6-v2")

  # or use convienence `ModelDict` to store parameters

  from graphistry.features import search_model
  g2 = g.featurize(X = ['text_col_1', .., 'text_col_n'], kind='nodes', **search_
↪model)

  # query using the power of transformers to find richly relevant results          ␣
```

(continues on next page)

(continued from previous page)

```
↪

  results_df, query_vector = g2.search('my natural language query', ...)

  print(results_df[['_distance', 'text_col', ..]])  #sorted by relevancy

  # or see graph of matching entities and original edges

  g2.search_graph('my natural language query', ...).plot()
```

- If edges are not given, `g.umap(..)` will supply them:

```
  ndf = pd.read_csv(nodes.csv)
  g = graphistry.nodes(ndf)
  g2 = g.umap(X = ['text_col_1', .., 'text_col_n'], min_words=0, ...)

  g2.search_graph('my natural language query', ...).plot()
```

See `help(g.search_graph)` for options

10.9.10.5 Knowledge Graph Embeddings

- Train a RGCN model and predict:

```
  edf = pd.read_csv(edges.csv)
  g = graphistry.edges(edf, src, dst)
  g2 = g.embed(relation='relationship_column_of_interest', **kwargs)

  # predict links over all nodes
  g3 = g2.predict_links_all(threshold=0.95)  # score high confidence predicted edges
  g3.plot()

  # predict over any set of entities and/or relations.
  # Set any `source`, `destination` or `relation` to `None` to predict over all of␣
↪them.
  # if all are None, it is better to use `g.predict_links_all` for speed.
  g4 = g2.predict_links(source=['entity_k'],
                  relation=['relationship_1', 'relationship_4', ..],
                  destination=['entity_l', 'entity_m', ..],
                  threshold=0.9,  # score threshold
                  return_dataframe=False)  # set to `True` to return dataframe, or␣
↪just access via `g4._edges`
```

- Detect Anamolous Behavior (example use cases such as Cyber, Fraud, etc)

```
  # Score anomolous edges by setting the flag `anomalous` to True and set␣
↪confidence threshold low
  g5 = g.predict_links_all(threshold=0.05, anomalous=True)  # score low confidence␣
↪predicted edges
  g5.plot()
```

(continues on next page)

(continued from previous page)

```
g6 = g.predict_links(source=['ip_address_1', 'user_id_3'],
                relation=['attempt_logon', 'phishing', ..],
                destination=['user_id_1', 'active_directory', ..],
                anomalous=True,
                threshold=0.05)
g6.plot()
```

- Train a RGCN model including auto-featurized node embeddings

```
edf = pd.read_csv(edges.csv)
ndf = pd.read_csv(nodes.csv)  # adding node dataframe

g = graphistry.edges(edf, src, dst).nodes(ndf, node_column)

# inherets all the featurization `kwargs` from `g.featurize`
g2 = g.embed(relation='relationship_column_of_interest', use_feat=True, **kwargs)
g2.predict_links_all(threshold=0.95).plot()
```

See `help(g.embed)`, `help(g.predict_links)` , or `help(g.predict_links_all)` for options

10.9.10.6 DBSCAN

- Enrich UMAP embeddings or featurization dataframe with GPU or CPU DBSCAN

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')

# cluster by UMAP embeddings
kind = 'nodes' | 'edges'
g2 = g.umap(kind=kind).dbscan(kind=kind)
print(g2._nodes['_dbscan']) | print(g2._edges['_dbscan'])

# dbscan in `umap` or `featurize` via flag
g2 = g.umap(dbscan=True, min_dist=0.2, min_samples=1)

# or 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, inhereted from `g.get_
↪matrix(cols)`
g2 = g.featurize().dbscan(cols=['ip_172', 'location', 'alert'], **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`

new_df = pd.read_csv(..)
```

(continues on next page)

(continued from previous page)

```
    # transform on new data according to fit dbscan model
    g3 = g2.transform_dbscan(new_df)
```

See `help(g.dbscan)` or `help(g.transform_dbscan)` for options

10.9.10.7 Quickly configurable

Set visual attributes through https://hub.graphistry.com/docs/api/2/rest/upload/#createdataset2 and set https://hub.graphistry.com/docs/api/1/rest/url/. Check out the tutorials on https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/encodings-colors.ipynb, https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/encodings-sizes.ipynb, https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/encodings-icons.ipynb, https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/encodings-badges.ipynb, https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/edge-weights.ipynb and https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_features/shar

```
  g
    .privacy(mode='private', invited_users=[{'email': 'friend1@site.ngo', 'action': '10'}
↪], notify=False)
    .edges(df, 'col_a', 'col_b')
    .edges(my_transform1(g._edges))
    .nodes(df, 'col_c')
    .nodes(my_transform2(g._nodes))
    .bind(source='col_a', destination='col_b', node='col_c')
    .bind(
      point_color='col_a',
      point_size='col_b',
      point_title='col_c',
      point_x='col_d',
      point_y='col_e')
    .bind(
      edge_color='col_m',
      edge_weight='col_n',
      edge_title='col_o')
    .encode_edge_color('timestamp', ["blue", "yellow", "red"], as_continuous=True)
    .encode_point_icon('device_type', categorical_mapping={'macbook': 'laptop', ...})
    .encode_point_badge('passport', 'TopRight', categorical_mapping={'Canada': 'flag-
↪icon-ca', ...})
    .encode_point_color('score', ['black', 'white'])
    .addStyle(bg={'color': 'red'}, fg={}, page={'title': 'My Graph'}, logo={})
    .settings(url_params={
      'play': 2000,
      'menu': True, 'info': True,
      'showArrows': True,
      'pointSize': 1.0,  # Node size (0.1-10.0, log scale: 0.2→"15", 1.0→"50", 5.0→"85
↪")
      'edgeCurvature': 0.5,  # Edge curvature (0.0-1.0)
      'edgeOpacity': 1.0,  # Edge transparency (0.0-1.0)
      'pointOpacity': 1.0,  # Node transparency (0.0-1.0)
      'lockedX': False, 'lockedY': False, 'lockedR': False,
      'linLog': False, 'strongGravity': False, 'dissuadeHubs': False,
```

(continues on next page)

(continued from previous page)

```
      'edgeInfluence': 1.0, 'precisionVsSpeed': 1.0, 'gravity': 1.0, 'scalingRatio': 1.0,
      'showLabels': True, 'showLabelOnHover': True,
      'showPointsOfInterest': True, 'showPointsOfInterestLabel': True,
↪'showLabelPropertiesOnHover': True,
      'pointsOfInterestMax': 5
    })
    .plot()
```

10.9.11 Plugins: Graph compute & layout

10.9.11.1 Use igraph (CPU) and cugraph (GPU) compute

Install the plugin of choice and then:

```
g2 =  g.compute_igraph('pagerank')
assert 'pagerank' in g2._nodes.columns

g3 = g.compute_cugraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

10.9.11.2 igraph

With `pip install graphistry[igraph]`, you can also use https://igraph.org/python/doc/api/igraph.Graph.html#layout:

```
g.layout_igraph('sugiyama').plot()
g.layout_igraph('sugiyama', directed=True, params={}).plot()
```

See list https://github.com/graphistry/pygraphistry/blob/master/graphistry/plugins/igraph.py#L365

10.9.11.3 graphviz

With graphviz installed, you can use its many layouts. See https://github.com/graphistry/pygraphistry/blob/master/demos/demos_databases_apis/graphviz/graphviz.ipynb

```
# 1. Engine: apt-get install graphviz graphviz-dev
# 2. Bindings: pip install -q graphistry[pygraphviz]

# graphviz dot layout with graphistry interactive render
g.layout_graphviz('dot').plot()

# save graphviz render to disk
g.layout_graphviz('dot', render_to_disk=True, path='./graph.png', format='render')

# custom attributes
assert 'color' in g._edges.columns and 'shape' in g._nodes.columns
g.layout_graphviz(
  'dot',
  graph_attrs={},
  node_attrs={'color': 'green'},
```

(continues on next page)

(continued from previous page)

```
  edge_attrs={}).plot()

help(g.layout_graphviz)
```

See layout algorithm list https://github.com/graphistry/pygraphistry/blob/master/graphistry/plugins_types/graphviz_types.p
The layout algorithms, and attributes at global and node/edge-level, are in the https://graphviz.org/docs/layouts/.

10.9.11.4 cuGraph

With https://www.rapids.ai install:

```
g.layout_cugraph('force_atlas2').plot()
help(g.layout_cugraph)
```

See list https://github.com/graphistry/pygraphistry/blob/master/graphistry/plugins/cugraph.py#L315

10.9.12 Resources

- Graphistry https://hub.graphistry.com/docs/ui/index/
- https://hub.graphistry.com/docs/api/:
  - https://hub.graphistry.com/docs/api/1/rest/url/#urloptions
  - https://hub.graphistry.com/docs/api/1/rest/auth/
  - https://hub.graphistry.com/docs/api/2/rest/upload/#createdataset2, including multiple file formats and settings
  - https://hub.graphistry.com/docs/api/2/rest/upload/colors/#extendedpalette2 and https://hub.graphistry.com/docs/api/api-color-palettes/ (ColorBrewer)
  - Bindings and colors, REST API, embedding URLs and URL parameters, dynamic JS API, and more
  - JavaScript and more!
- Python-specific
  - http://pygraphistry.readthedocs.org/en/latest/
  - Within a notebook, you can always run `help(graphistry)`, `help(graphistry.hypergraph)`, etc.
- https://github.com/graphistry/graph-app-kit dashboarding

10.10 Python API Reference

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

10.10.1 GraphistryClient

Build and visualize graphs, starting from various formats or database connections.

Register with our server to upload graphs for visualization.

10.10.1.1 GraphistryClient Class

Build graphs and authenticate for remote visualization.

Example:

```
import graphistry

# Register to the singleton GraphistryClient
graphistry.register(...)

# Create a :py:class:`graphistry.plotter.Plotter`
g = graphistry.bind(nodes=df_nodes, edges=df_edges)

g.plot()
```

Session Management: The GraphistryClient manages authentication and state through a ClientSession object. On plot(), the ArrowUploader manages user auth token refresh on that ClientSession.

Concurrency: Methods are the concurrency boundary. The global PyGraphistry instance uses shared session state, so it's not safe for concurrent use across threads. For concurrent or multi-tenant scenarios, create separate client instances using graphistry.client().

Multiple clients:

```
import graphistry
# Create independent client instances
alice_g = graphistry.client()
alice_g.register(api=3, username='alice', password='pw')

bob_g = graphistry.client()
bob_g.register(api=3, username='bob', password='pw')
```

Each client has its own isolated session (self.session) that tracks: - Authentication tokens and refresh state - Server configuration (hostname, protocol, API version) - Organization and privacy settings - SSO and personal key authentication state

See:*graphistry.plotter.Plotter*, for session management in plottables.

10.10.1.2 ClientSession Class

Holds all configuration and authentication state for a Graphistry client.

Session Isolation: Each GraphistryClient instance maintains its own ClientSession, providing isolation for concurrent and multi-tenant use cases. The session tracks:

- Authentication state (tokens, API keys, SSO state)
- Server configuration (hostname, protocol, API version)
- Organization and privacy settings

- Plugin configurations (Kusto, Spanner, etc.)

Thread Safety: A ClientSession instance should only be used within a single concurrency context (thread, async task, etc.). For multi-threaded applications, create separate client instances for each thread.

Token Management: Authentication tokens may be refreshed during plot() operations. The session maintains the current token state and handles refresh automatically.

10.10.2 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

10.10.2.1 Arrow Conversion Validation

`plot()`, `upload()`, and `to_arrow()` use `validate='autofix'` by default for pandas/cuDF to Arrow conversion. When Arrow rejects mixed-type object columns, autofix probes object columns, coerces only the failing columns to strings, and emits a warning naming those columns when `warn=True`. Use `validate='strict'` or `validate='strict-fast'` to raise instead.

For pandas inputs, autofix prefers pandas' nullable string dtype when available so missing values remain Arrow nulls after coercion. Older pandas versions fall back to standard Python string coercion.

When a plotter has an experimental `graphistry.schema.GraphSchema` bound via `g.bind(schema=schema)`, pass `schema_validate='strict'` to `plot()`, `upload()`, or `to_arrow()` to require declared columns and Arrow types at the boundary. Use `schema_validate='autofix'` to cast compatible columns to declared Arrow types after normal Arrow conversion. The default `schema_validate=False` preserves existing behavior.

10.10.2.2 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.

Implements the *graphistry.Plottable.Plottable* interface.

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.
- `graphistry.plugins.kusto.KustoMixin`: Integrates with Azure Kusto DB.
- `graphistry.plugins.spanner.SpannerMixin`: Integrates with Google Spanner DB.

Attributes:

All attributes are inherited from the mixins and base classes.

Session Binding:

A Plottable's state is tied to the client used to create it through two attributes:

- _pygraphistry: Reference to the *GraphistryClient* that created this plottable
- session: The *ClientSession* (self._pygraphistry.session)

See: *graphistry.pygraphistry.GraphistryClient* for more details.

This binding ensures that authentication state, server configuration, and other session-specific settings are preserved when plotting. The session reference is particularly important during plot() operations where token refresh may occur.

Concurrency:

Each plottable inherits the concurrency constraints of its parent client. A plottable should only be used within the same concurrency context as the client that created it.

To transfer a plottable between clients, use client.set_client_for(plottable).

10.10.2.3 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'})
```

`apply_encodings`(*react_encodings*, *validate='strict'*, *warn=True*)

Apply React-style declarative encoding payloads.

Supported keys: - `encodePointColor` / `encodeEdgeColor`: `[column, variation?, mapping_or_palette?]` - `encodePointSize` / `encodeEdgeSize` / `encodeEdgeWeight` /

`encodePointOpacity` / `encodeEdgeOpacity`: `[column, categorical_mapping?, default_mapping?]`

- `encodePointLabel` / `encodeEdgeLabel` / `encodePointTitle` / `encodeEdgeTitle`: `[column]`
- `encodePointIcons` / `encodeEdgeIcons`: `[column, categorical_mapping_or_bins?, default_mapping?]`
- `encodeAxis`: `rows` list accepted by *encode_axis()*

Parameters

- **react_encodings** (*ApplyEncodingsReactSettingsDict | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Plottable

base_url_client(*v=None*)

Parameters
v (*str | None*)

Return type
str

base_url_server(*v=None*)

Parameters
v (*str | None*)

Return type
str

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*, *point_longitude=None*, *point_latitude=None*, *dataset_id=None*, *url=None*, *nodes_file_id=None*, *edges_file_id=None*, *schema=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** (*Optional[str]*) – Attribute containing an edge's source ID
- **destination** (*Optional[str]*) – Attribute containing an edge's destination ID
- **node** (*Optional[str]*) – Attribute containing a node's ID
- **edge** (*Optional[str]*) – Attribute containing an edge's ID
- **edge_title** (*Optional[str]*) – Attribute overriding edge's minimized label text. By default, the edge source and destination is used.
- **edge_label** (*Optional[str]*) – Attribute overriding edge's expanded label text. By default, scrollable list of attribute/value mappings.

- **edge_color** (*Optional [str]*) – Attribute overriding edge's color. rgba (int64) or int32 palette index, see https://graphistry.github.io/docs/legacy/api/0.9.2/api.html#extendedpalette definitions for values. Based on Color Brewer.
- **edge_source_color** (*Optional [str]*) – Attribute overriding edge's source color if no edge_color, as an rgba int64 value.
- **edge_destination_color** (*Optional [str]*) – Attribute overriding edge's destination color if no edge_color, as an rgba int64 value.
- **edge_weight** (*Optional [str]*) – Attribute overriding edge weight. Default is 1. Advanced layout controls will relayout edges based on this value.
- **point_title** (*Optional [str]*) – Attribute overriding node's minimized label text. By default, the node ID is used.
- **point_label** (*Optional [str]*) – Attribute overriding node's expanded label text. By default, scrollable list of attribute/value mappings.
- **point_color** (*Optional [str]*) – Attribute overriding node's color.rgba (int64) or int32 palette index, see https://graphistry.github.io/docs/legacy/api/0.9.2/api.html#extendedpalette definitions for values. Based on Color Brewer.
- **point_size** (*Optional [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** (*Optional [str]*) – Attribute overriding node's initial x position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **point_y** (*Optional [str]*) – Attribute overriding node's initial y position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **point_longitude** (*Optional [str]*) – Attribute containing node's longitude coordinate for geographic visualization. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **point_latitude** (*Optional [str]*) – Attribute containing node's latitude coordinate for geographic visualization. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **dataset_id** (*Optional [str]*) – Remote dataset id
- **url** (*Optional [str]*) – Remote dataset URL
- **nodes_file_id** (*Optional [str]*) – Remote nodes file id
- **edges_file_id** (*Optional [str]*) – Remote edges file id
- **schema** (*Optional [Any]*) – Optional experimental public GFQL schema declaration from `graphistry.schema`.
- **edge_size** (*str | None*)
- **edge_opacity** (*str | None*)
- **edge_icon** (*str | None*)
- **point_weight** (*str | None*)
- **point_opacity** (*str | None*)
- **point_icon** (*str | None*)

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')

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 Union[List[ASTObject], Chain]

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

Return type
Plottable

chain_remote(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine=None*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*Literal['pandas', 'cudf'] | None*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
Plottable

chain_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine=None*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*Literal['pandas', 'cudf'] | None*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

`client_protocol_hostname`(*v=None*)

Get or set the client protocol and hostname, e.g., "https://hub.graphistry.com (external link)" .

By default, uses {protocol()}://{server()}. Typically used when public browser routes are different from backend server routes, e.g., enterprise WAF routes for browser use and internal firewalls routes for server use.

Note that sets are global as PyGraphistry._config entries, so be careful in multi-user environments.

Parameters
`v` (*str | None*)

Return type
str

`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

`collections`(*collections=None*, *show_collections=None*, *collections_global_node_color=None*, *collections_global_edge_color=None*, *validate='autofix'*, *warn=True*)

Set collections URL parameters. Additive over previous settings.

Parameters

- `collections` (*str | CollectionSet | CollectionIntersection | List[CollectionSet | CollectionIntersection] | None*) – List/dict of collections or JSON/URL-encoded JSON string (stored as URL-encoded JSON).
- `show_collections` (*bool | None*) – Toggle collections panel display.
- `collections_global_node_color` (*str | None*) – Hex color for non-collection nodes (leading # stripped).
- `collections_global_edge_color` (*str | None*) – Hex color for non-collection edges (leading # stripped).
- `validate` (*Literal['strict', 'strict-fast', 'autofix'] | bool*) – Validation mode. 'autofix' (default) drops invalid collections and color fields with warnings, 'strict' raises on issues.
- `warn` (*bool*) – Whether to emit warnings when validate='autofix'. validate=False forces warn=False.

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
- **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
```

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

Enrich or replace graph using igraph methods

igraph is a CPU-only library. cuDF DataFrames are automatically converted to pandas before calling igraph, and the result is converted back. For a GPU-native alternative, see *compute_cugraph()*.

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']})
```

(continues on next page)

(continued from previous page)

```
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
```

`compute_networkx`(*alg*, *out_col=None*, *params=None*, *directed=True*, *G=None*)

Run a supported NetworkX algorithm and return an enriched PyGraphistry graph.

Parameters

- **alg** (*str*) – Explicitly supported NetworkX algorithm name.
- **out_col** (*Optional [str]*) – Optional output column name for single-column node/edge algorithms.
- **params** (*Optional [Mapping [str, Any]]*) – Keyword parameters forwarded to the selected NetworkX algorithm.
- **directed** (*bool*) – Whether to build a directed NetworkX graph from the current graph.
- **G** (*Optional [Any]*) – Optional prebuilt NetworkX graph to run instead of converting `self`.
- **self** (Plottable)

Returns
Plotter

Return type
Plotter

Example: Degree centrality

```
g2 = g.compute_networkx("degree_centrality", out_col="degree_score")
```

Example: HITS

```
g2 = g.compute_networkx("hits")
assert "hubs" in g2._nodes.columns
assert "authorities" 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
- `self` (Plottable)

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()
```

embed(*relation*, *proto='DistMult'*, *embedding_dim=32*, *use_feat=False*, *X=None*, *epochs=2*, *batch_size=32*, *train_split=0.8*, *sample_size=1000*, *num_steps=50*, *lr=0.01*, *inplace=False*, *device='cpu'*, *evaluate=True*, **args*, ***kwargs*)

Parameters

- **relation** (*str*)
- **proto** (*str | Callable[[Any, Any, Any], Any] | None*)
- **embedding_dim** (*int*)
- **use_feat** (*bool*)
- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **epochs** (*int*)
- **batch_size** (*int*)
- **train_split** (*float | int*)
- **sample_size** (*int*)
- **num_steps** (*int*)
- **lr** (*float*)
- **inplace** (*bool | None*)
- **device** (*str | None*)
- **evaluate** (*bool*)

Return type

Plottable

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*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict [Any , Any] | None*)
- **continuous_binning** (*List [Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable [[Any , Any] , int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)
- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)
- **blend_mode** (*str | None*)

- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

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.)
- **continuous_binning** (*List[Any] | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

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

Parameters

- **column** (*str*)

- **categorical_mapping** (*Dict[Any, str] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_kepler(*kepler_encoding*)

Apply a complete Kepler.gl encoding to the plotter.

Accepts a *graphistry.kepler.KeplerEncoding* object or plain dictionary. Returns a new Plotter instance with the encoding applied (immutable pattern).

Parameters
kepler_encoding (*Union[Dict[str, Any], KeplerEncoding]*) – KeplerEncoding object or dict with structure: {'datasets': [...], 'layers': [...], 'options': {...}, 'config': {...}}

Returns
New Plotter instance with the Kepler encoding applied

Return type
Plottable

Example: Using KeplerEncoding container

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

kepler = (KeplerEncoding()
          .with_dataset(KeplerDataset(id="points", type="nodes"))
          .with_layer(KeplerLayer({
              "id": "point-layer",
              "type": "point",
              "config": {"dataId": "points", "columns": {"lat": "lat", "lng":
↪ "lng"}}
          })))
g = g.encode_kepler(kepler)
```

Example: Using plain dict

```
kepler_dict = {
    'datasets': [{'info': {'id': 'points'}, 'type': 'nodes'}],
    'layers': [{'id': 'point-layer', 'type': 'point', 'config': {'dataId':
↪'points'}}],
    'options': {'centerMap': True},
    'config': {'cullUnusedColumns': True}
}
g = g.encode_kepler(kepler_dict)
```

See *graphistry.kepler.KeplerEncoding* for complete documentation.

encode_kepler_config(*raw_dict: Dict[str, Any]*) → *Plottable*

encode_kepler_config(*raw_dict: None = None*, ***, *cull_unused_columns: bool | None = None*, *overlay_blending: Literal['normal', 'additive', 'subtractive'] | None = None*, *tile_style: Dict[str, Any] | None = None*) → *Plottable*

Apply Kepler.gl configuration settings to the plotter.

Accepts same parameters as *graphistry.kepler.KeplerConfig*. Returns a new Plotter instance with the config applied (immutable pattern).

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl config dictionary (if provided, all other params ignored)
- **cull_unused_columns** (*Optional[bool]*) – Remove columns not used by layers (default: True)
- **overlay_blending** (*Optional[Literal['normal', 'additive', 'subtractive']]*) – Blend mode - 'normal', 'additive', 'subtractive' (default: 'normal')
- **tile_style** (*Optional[Dict[str, Any]]*) – Base map tile style configuration

Returns
New Plotter instance with the config applied

Return type
Plottable

Example: Structured params

```
g = g.encode_kepler_config(
    cull_unused_columns=True,
    overlay_blending='additive'
)
```

Example: Native format

```
g = g.encode_kepler_config({
    "cullUnusedColumns": True,
    "overlayBlending": "additive"
})
```

See *graphistry.kepler.KeplerConfig* for complete parameter documentation.

encode_kepler_dataset(*raw_dict: Dict[str, Any]*) → *Plottable*

encode_kepler_dataset(*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['nodes'] = 'nodes'*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*) → *Plottable*

encode_kepler_dataset(*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['edges'] = 'edges'*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *map_node_coords: bool | None = None*, *map_node_coords_mapping: Dict[str, str] | None = None*, *computed_columns: Dict[str, Any] | None = None*) → *Plottable*

encode_kepler_dataset(*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['countries', 'zeroOrderAdminRegions'] = 'countries'*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *resolution: Literal[10, 50, 110] | None = None*, *boundary_lakes: bool | None = None*, *filter_countries_by_col: str | None = None*, *include_countries: List[str] | None = None*, *exclude_countries: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*) → *Plottable*

encode_kepler_dataset(*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['states', 'provinces', 'firstOrderAdminRegions'] = 'states'*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *boundary_lakes: bool | None = None*, *filter_states_by_col: str | None = None*, *include_states: List[str] | None = None*, *exclude_states: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*) → *Plottable*

Add a Kepler.gl dataset to the encoding.

Accepts same parameters as *graphistry.kepler.KeplerDataset*. Returns a new Plotter instance with the dataset appended (immutable pattern).

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl dataset dictionary (if provided, all other params ignored)
- **id** (*Optional[str]*) – Dataset identifier (auto-generated if None)
- **type** (*Optional[str]*) – Dataset type - 'nodes', 'edges', 'countries', 'states', etc.
- **label** (*Optional[str]*) – Display label (defaults to id)
- **include** (*Optional[List[str]]*) – Columns to include (whitelist)
- **exclude** (*Optional[List[str]]*) – Columns to exclude (blacklist)
- **computed_columns** (*Optional[Dict[str, Any]]*) – Computed/aggregated columns for data enrichment
- **map_node_coords** (*Optional[bool]*) – Auto-map source/target node coordinates to edges (edges only)
- **map_node_coords_mapping** (*Optional[Dict[str, str]]*) – Custom column names for mapped coordinates (edges only)
- **resolution** (*Optional[Literal[10, 50, 110]]*) – Map resolution - 10 (high), 50 (medium), 110 (low) (geographic datasets only)
- **boundary_lakes** (*Optional[bool]*) – Include lake boundaries (geographic datasets only)

Returns

New Plotter instance with the dataset added

Return type

Plottable

Example: Node dataset

```
g = g.encode_kepler_dataset(id="companies", type="nodes", label="Companies")
```

Example: Edge dataset with coordinate mapping

```
g = g.encode_kepler_dataset(
    id="relationships",
    type="edges",
    map_node_coords=True
)
```

Example: Countries dataset

```
g = g.encode_kepler_dataset(
    id="countries",
    type="countries",
    resolution=50
)
```

See *graphistry.kepler.KeplerDataset* for complete parameter documentation.

encode_kepler_layer(*raw_dict*)

Add a Kepler.gl layer to the encoding.

Accepts native Kepler.gl layer dictionary (same as *graphistry.kepler.KeplerLayer*). Returns a new Plotter instance with the layer appended (immutable pattern).

Parameters
raw_dict (*Dict[str, Any]*) – Native Kepler.gl layer dictionary with structure: {'id': ..., 'type': ..., 'config': {...}}

Returns
New Plotter instance with the layer added

Return type
Plottable

Example: Point layer

```
g = g.encode_kepler_layer({
    "id": "my-layer",
    "type": "point",
    "config": {
        "dataId": "my-dataset",
        "columns": {"lat": "latitude", "lng": "longitude"},
        "color": [255, 140, 0]
    }
})
```

Example: Arc layer

```
g = g.encode_kepler_layer({
    "id": "connections",
    "type": "arc",
    "config": {
        "dataId": "edges",
        "columns": {
            "lat0": "edgeSourceLatitude",
            "lng0": "edgeSourceLongitude",
            "lat1": "edgeTargetLatitude",
            "lng1": "edgeTargetLongitude"
        }
    }
})
```

See *graphistry.kepler.KeplerLayer* and *Kepler.gl Layer Format* for layer format details.

encode_kepler_options(*raw_dict: Dict[str, Any]*) → *Plottable*

encode_kepler_options(*raw_dict: None = None*, ***, *center_map: bool | None = None*, *read_only: bool | None = None*) → *Plottable*

Apply Kepler.gl visualization options to the plotter.

Accepts same parameters as *graphistry.kepler.KeplerOptions*. Returns a new Plotter instance with the options applied (immutable pattern).

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl options dictionary (if provided, all other params ignored)
- **center_map** (*Optional[bool]*) – Auto-center map on data (default: True)
- **read_only** (*Optional[bool]*) – Disable map interactions (default: False)

Returns
New Plotter instance with the options applied

Return type
Plottable

Example: Structured params

```
g = g.encode_kepler_options(center_map=True, read_only=False)
```

Example: Native format

```
g = g.encode_kepler_options({"centerMap": True, "readOnly": False})
```

See *graphistry.kepler.KeplerOptions* for complete parameter documentation.

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*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)
- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)
- **blend_mode** (*str | None*)

- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

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'
- **continuous_binning** (*List[Any] | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **shape** (*str | None*)

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

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)
```

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

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

fa2_layout(*fa2_params=None*, *circle_layout_params=None*, *singleton_layout=None*, *partition_key=None*, *engine='auto'*, *allow_cpu_fallback=False*)

Parameters

- **fa2_params** (*Dict[str, Any] | None*)
- **circle_layout_params** (*Dict[str, Any] | None*)
- **singleton_layout** (*Callable[[Plottable, Tuple[float, float, float, float] | Any], Plottable] | None*)
- **partition_key** (*str | None*)

- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **allow_cpu_fallback** (*bool*)

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

filter_weighted_edges(*scale=1.0*, *index_to_nodes_dict=None*, *inplace=False*, *kind='nodes'*)

Parameters

- **scale** (*float*)
- **index_to_nodes_dict** (*Dict | None*)
- **inplace** (*bool*)
- **kind** (*Literal['nodes', 'edges']*)

Return type
Plottable | None

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

Take input cugraph.Graph object and load in data and bindings (source, destination, edge_weight)

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

from_gexf(*source*, *name=None*, *description=None*, *bind_node_viz=None*, *bind_edge_viz=None*, *parse_engine='auto'*)

Load a GEXF file/URL/stream into a PyGraphistry plotter.

Parameters

- **self** (Plottable)
- **source** (*str | bytes | bytearray | IO[bytes] | IO[str]*)
- **name** (*str | None*)
- **description** (*str | None*)
- **bind_node_viz** (*Iterable[Literal['color', 'size', 'opacity', 'position', 'icon']] | None*)
- **bind_edge_viz** (*Iterable[Literal['color', 'size', 'opacity']] | None*)
- **parse_engine** (*Literal['auto', 'defused', 'stdlib']*)

Return type
Plottable

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** (*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
```

`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)
```

(continues on next page)

(continued from previous page)

```
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()
```

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

`gfql_remote`(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
Plottable

`gfql_remote_shape`(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)

- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

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

    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*, ***, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *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*, *include_zero_hop_seed=False*, *target_wave_front=None*, *engine='auto'*)

Parameters

- **nodes** (*DataFrame | None*)
- **hops** (*int | None*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)

- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)
- **label_seeds** (*bool*)
- **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*)
- **include_zero_hop_seed** (*bool*)
- **target_wave_front** (*DataFrame | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

Return type
Plottable

hypergraph(*raw_events: Any | None = None, *, entity_types: List[str] | None = None, opts: dict = {}, drop_na: bool = True, drop_edge_attrs: bool = False, verbose: bool = True, direct: bool = False, engine: EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto'] = 'auto', npartitions: int | None = None, chunksize: int | None = None, from_edges: bool = False, return_as: Literal['graph'] = 'graph'*) → *Plottable*

hypergraph(*raw_events: Any | None = None, *, entity_types: List[str] | None = None, opts: dict = {}, drop_na: bool = True, drop_edge_attrs: bool = False, verbose: bool = True, direct: bool = False, engine: EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto'] = 'auto', npartitions: int | None = None, chunksize: int | None = None, from_edges: bool = False, return_as: Literal['all']*) → HypergraphResult

hypergraph(*raw_events: Any | None = None, *, entity_types: List[str] | None = None, opts: dict = {}, drop_na: bool = True, drop_edge_attrs: bool = False, verbose: bool = True, direct: bool = False, engine: EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto'] = 'auto', npartitions: int | None = None, chunksize: int | None = None, from_edges: bool = False, return_as: Literal['entities', 'events', 'edges', 'nodes'] = 'graph'*) → Any

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

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

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()
```

Parameters
ig (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

`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)
- **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')
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*, *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

Graphviz is a CPU-only library. cuDF DataFrames are automatically converted to pandas before calling graphviz, and the result is converted back.

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, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Graphviz graph attributes, see https://graphviz.org/docs/graph/
- **node_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.NodeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Graphviz node attributes, see https://graphviz.org/docs/nodes/
- **edge_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.EdgeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – 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')
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.

igraph is a CPU-only library. cuDF DataFrames are automatically converted to pandas before calling igraph, and the result is converted back. For a GPU-native alternative, see *layout_cugraph()*.

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()
```

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='auto'*)

Parameters

- **reuse** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

Return type
Plottable

`name`(*name*)

Upload name

Parameters
name (*str*) – Upload name

`networkx2pandas`(*G*)

Parameters
G (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

`networkx_checkoverlap`(*g*)

Raise an error if the node attribute already exists in the graph

Parameters
g (*Any*)

Return type
None

`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
```

(continues on next page)

(continued from previous page)

```
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)
```

Parameters

- **edges** (*DataFrame*)
- **directed** (*bool*)

Return type

Any

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='auto'*, *skip_upload=False*, *as_files=False*, *memoize=True*, *erase_files_on_fail=True*, *extra_html=''*, *override_html_style=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

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** (*Optional[Union[bool, RenderModes]] = "auto"*) – Whether to render the visualization using the native environment (default “auto”, True), a URL (“url”, False, None), a PyGraphistry Plottable (“g”), thon object (“ipython”), interactive Databricks object (“databricks”), or open a local web browser (“browser”). If _render is set via .settings(), and set to None, use _render.
- **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.
- **erase_files_on_fail** (*bool*) – Removes uploaded files if an error is encountered during parse. Only applicable when upload as files enabled. 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** (*ValidationParam*) – Data validation mode. ‘autofix’ (default) auto-coerces mixed-type columns to string. ‘strict’ or ‘strict-fast’ raises ArrowConver-

sionError on mixed types. For backward compatibility: True maps to 'strict', False maps to 'autofix'.

- **warn** (*bool*) – Whether to emit warnings when auto-fixing data issues (only applies when validate='autofix'). validate=False forces warn=False. Default True.
- **schema_validate** (*SchemaValidationParam*) – Opt-in bound `GraphSchema` validation at the Arrow upload boundary. False disables schema enforcement. True/'strict' rejects missing or incompatible declared columns. 'autofix' casts compatible columns to declared Arrow types.

Return type
Any

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)
```

`plot_static`(*format='svg'*, *path=None*, *engine='graphviz-svg'*, *prog='dot'*, *args=None*, *reuse_layout=True*, *directed=True*, *strict=False*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *drop_unsanitary=False*, *max_nodes=None*, *max_edges=None*)

Render a static image of the current graph (e.g., for notebooks/docs).

Returns an IPython display object (SVG or Image) that auto-displays in notebooks. Use `.data` to access raw bytes for programmatic use.

Engines: - graphviz-svg (default) / graphviz-png: render image (optionally write to path) - graphviz: render to any Graphviz format (see Format), e.g., pdf - graphviz-dot: return DOT string (optionally write to path) - mermaid-code: return Mermaid DSL string (optionally write to path)

If point x/y encodings are bound (or columns named x/y exist), reuse them for rendering. Otherwise, Graphviz lays out the graph. When positions are reused, Graphviz is invoked with `neato -n2` to respect them.

Parameters

- **format** (*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']*) – Output format, e.g., 'svg', 'png', 'pdf' (graphviz engines)
- **path** (*str | None*) – Optional path to also write the image/text

- **engine** (*str*) – Rendering engine; supports graphviz, graphviz-svg/png, graphviz-dot, mermaid-code
- **prog** (*Literal['acyclic', 'ccomps', 'circo', 'dot', 'fdp', 'gc', 'gvcolor', 'gvpr', 'neato', 'nop', 'osage', 'patchwork', 'sccmap', 'sfdp', 'tred', 'twopi', 'unflatten']*) – Graphviz layout program when computing layout
- **args** (*str | None*) – Optional args passed to graphviz (e.g., '-n2' when reusing positions)
- **reuse_layout** (*bool*) – If True and positions are bound/available, reuse them; else layout
- **directed** (*bool*) – Graphviz directed flag
- **strict** (*bool*) – Graphviz strict flag
- **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'], str | int | float | bool] | None*) – Graphviz graph attributes
- **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'], str | int | float | bool] | None*) – Graphviz node attributes
- **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'], str | int | float | bool] | None) – Graphviz edge attributes

- **drop_unsanitary** (*bool*) – Reject unsanitary attributes
- **max_nodes** (*int | None*) – Optional cap on node count
- **max_edges** (*int | None*) – Optional cap on edge count

Returns
SVG or Image display object (use .data for bytes), or DOT/Mermaid string

Return type
bytes | str

Example: Basic usage

```
g.plot_static()  # Returns SVG, auto-displays in notebook
```

Example: Save to file

```
g.plot_static(path='graph.svg')  # Writes file AND returns SVG
```

Example: Get raw bytes

```
svg_bytes = g.plot_static().data
```

privacy(*mode=None*, *notify=None*, *invited_users=None*, *message=None*, *mode_action=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
- **mode_action** (*Optional[ModeAction]*) – Action to take when mode is changed, defaults to global privacy()

Return type
Plottable

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)
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')
```

(continues on next page)

(continued from previous page)

```
#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()
```

protocol(*v=None*)

Get or set the server protocol, e.g., "https"

Note that sets are global as PyGraphistry._config entries, so be careful in multi-user environments.

Parameters
v (*str | None*)

Return type
str

prune_self_edges()

Return type
Plottable

python_remote_g(*code*, *api_token=None*, *dataset_id=None*, *format='parquet'*, *output_type='all'*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape'] | ~typing.Literal['table', 'shape'] | ~typing.Literal['json'] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
Plottable

python_remote_json(*code*, *api_token=None*, *dataset_id=None*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
Any

python_remote_table(*code*, *api_token=None*, *dataset_id=None*, *format='parquet'*, *output_type='table'*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **output_type** (*Literal['table', 'shape'] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
DataFrame

render_graphviz(*prog='dot'*, *format='svg'*, *args=None*, *directed=True*, *strict=False*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *drop_unsanitary=False*, *max_nodes=None*, *max_edges=None*, *path=None*, *include_positions=False*)

Render a graph to an image via graphviz and return the rendered bytes.

This wraps `layout_graphviz_core()` to compute positions, then draws with pygraphviz. Optionally enforces caps to keep renders small/deterministic for docs/examples.

When `include_positions` is True and the plot has bound x/y values, the existing layout is preserved rather than recomputed by Graphviz.

Parameters

- **self** (Plottable) – Base graph
- **prog** (`graphistry.plugins_types.graphviz_types.Prog`) – Layout algorithm

- **format** (graphistry.plugins_types.graphviz_types.Format) – Render format
- **directed** (*bool*) – Whether the graph is directed
- **strict** (*bool*) – Whether to treat the graph as strict
- **graph_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.GraphAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Graph-level attributes
- **node_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.NodeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Node-level attributes
- **edge_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.EdgeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Edge-level attributes
- **drop_unsanitary** (*bool*) – Reject unsanitary attrs
- **max_nodes** (*Optional[int]*) – Optional cap on nodes for rendering
- **max_edges** (*Optional[int]*) – Optional cap on edges for rendering
- **path** (*Optional[str]*) – Optional path to also write the render
- **args** (*str | None*)
- **include_positions** (*bool*)

Returns
Rendered bytes (SVG/PNG/etc.)

Return type
bytes

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*)

search(*query*, *cols=None*, *thresh=5000*, *fuzzy=True*, *top_n=10*)

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*)

Parameters

- **query** (*str*)
- **scale** (*float*)
- **top_n** (*int*)
- **thresh** (*float*)
- **broader** (*bool*)
- **inplace** (*bool*)

Return type
Plottable

server(*v=None*)

Get or set the server basename, e.g., "hub.graphistry.com"

Note that sets are global as PyGraphistry._config entries, so be careful in multi-user environments.

Parameters
v (*str / None*)

Return type
str

session: ClientSession

settings(*height=None*, *url_params=None*, *render=None*, *validate='autofix'*, *warn=True*)

Specify iframe height and add URL parameter dictionary.

Collections URL params are normalized and URL-encoded at plot time; other params should already be URL-safe.

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
- **validate** (*ValidationParam*) – Validation mode for url_params. 'autofix' (default) drops invalid keys/types with warnings; 'strict' raises.
- **warn** (*bool*) – Whether to emit warnings in autofix mode.

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_arrow(*table=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*, *schema_table='edges'*)

Convert a DataFrame to Arrow format.

This method is useful for debugging Arrow conversion issues. If the DataFrame contains mixed-type columns that would cause Arrow conversion to fail, they will be automatically coerced to strings with a warning in autofix mode.

Parameters

- **validate** (*ValidationParam*) – Data validation mode. 'autofix' (default) auto-coerces mixed-type columns to string. 'strict' or 'strict-fast' raises ArrowConversionError on mixed types. For backward compatibility: True maps to 'strict', False maps to 'autofix'.
- **warn** (*bool*) – Whether to emit warnings when auto-fixing data issues (only applies when validate='autofix'). validate=False forces warn=False. Default True.
- **schema_validate** (*SchemaValidationParam*) – Opt-in bound `GraphSchema` validation. False disables schema enforcement. True/'strict' rejects missing or incompatible declared columns. 'autofix' casts compatible Arrow columns to declared types.
- **schema_table** (*str*) – Which bound schema table to validate against: 'edges' or 'nodes'.
- **table** (*Optional[pandas.DataFrame, cudf.DataFrame, pyarrow.Table]*) – DataFrame to convert. If None, converts the bound edges.

Returns
PyArrow Table, or None if table is None

Return type
Optional[pyarrow.Table]

Example: Debug Arrow conversion

```
import graphistry
import pandas as pd

df = pd.DataFrame({
    'src': [1, 2, 3],
    'dst': [2, 3, 1],
    'mixed': [b'bytes', 1.5, 'string']  # Mixed types
})
g = graphistry.edges(df, 'src', 'dst')

# Debug: see what Arrow conversion produces
arr = g.to_arrow(df, validate='autofix', warn=True)
print(arr.schema)

# Or convert bound edges
arr2 = g.to_arrow(validate='strict')
```

`to_cudf()`

Return type
Plottable

`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

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

`to_gexf(path=None, *, version='1.2draft', directed=True, include_viz=True, include_meta=True, meta_creator='graphistry', meta_description=None, meta_lastmodifieddate=None, node_attributes=None, edge_attributes=None)`

Export the current graph to a GEXF string (optionally writing to disk).

Parameters

- **self** (Plottable)

- **path** (*str | None*)
- **version** (*str*)
- **directed** (*bool*)
- **include_viz** (*bool*)
- **include_meta** (*bool*)
- **meta_creator** (*str | None*)
- **meta_description** (*str | None*)
- **meta_lastmodifieddate** (*str | None*)
- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)

Return type
str

to_igraph(*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.

igraph is a CPU-only library. cuDF DataFrames are automatically converted to pandas before being passed to igraph. For a GPU-native alternative, see *to_cugraph()*.

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)

Return type
Any

to_pandas()

Return type
Plottable

transform(*df*, *y=None*, *kind='nodes'*, *min_dist='auto'*, *n_neighbors=7*, *merge_policy=False*, *sample=None*, ***, *return_graph=True*, *scaled=True*, *verbose=False*)

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*)

Return type
Tuple[*DataFrame*, *DataFrame*] | Plottable

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={}*)

Parameters

- **df** (*DataFrame*)
- **y** (*DataFrame | None*)
- **kind** (*Literal['nodes', 'edges']*)
- **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*)

Parameters

- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **y** (*DataFrame | np.ndarray | List[str] | None*)
- **kind** (*Literal['nodes', 'edges']*)
- **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['auto', 'cuml', 'umap_learn']*)
- **feature_engine** (*str*)
- **inplace** (*bool*)
- **memoize** (*bool*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)
- **featurize_kwargs** (*Any*)

Return type
Plottable | None

umap_fit(*X*, *y=None*, *umap_fit_kwargs={}*)

Parameters

- **X** (*DataFrame*)
- **y** (*DataFrame | None*)
- **umap_fit_kwargs** (*Dict[str, Any]*)

Return type
Plottable

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

- **res** (Plottable)
- **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['auto', 'cuml', 'umap_learn']*)
- **suffix** (*str*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

Return type
Plottable

upload(*memoize=True*, *erase_files_on_fail=True*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Upload data to the Graphistry server and return as a Plottable. Headless-centric variant of plot().

Uses the currently bound schema structure and visual encodings. Optional parameters override the current bindings.

Upon successful upload, returned *Plottable* will have set the fields *dataset_id*, *url*, *edges_file_id*, and if applicable, *nodes_file_id*.

Parameters

- **memoize** (*bool*) – Tries to memoize pandas/cudf->arrow conversion, including skipping upload. Default true.
- **erase_files_on_fail** (*bool*) – Removes uploaded files if an error is encountered during parse. Only applicable when upload as files enabled. Default on.
- **validate** (*ValidationParam*) – Data validation mode. 'autofix' (default) auto-coerces mixed-type columns to string. 'strict' or 'strict-fast' raises ArrowConversionError on mixed types. For backward compatibility: True maps to 'strict', False maps to 'autofix'.
- **warn** (*bool*) – Whether to emit warnings when auto-fixing data issues (only applies when validate='autofix'). validate=False forces warn=False. Default True.
- **schema_validate** (*SchemaValidationParam*) – Opt-in bound `GraphSchema` validation at the Arrow upload boundary. False disables schema enforcement. True/'strict' rejects missing or incompatible declared columns. 'autofix' casts compatible columns to declared Arrow types.

Return type
Plottable

Example: Simple

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .bind(source='src', destination='dst')
    .edges(es)
```

(continues on next page)

(continued from previous page)

```
g2 = g1.upload()
print(f'dataset id: {g2._dataset_id}, url: {g2._url}')
```

property url: str | None

Get visualization URL if available.

Returns
Visualization URL if set by plot() or remote persistence

Return type
Optional[str]

validate_arrow_schema(*table='edges'*, ***, *validate='strict'*, *warn=True*)

Validate or coerce a bound table against the bound experimental `GraphSchema`.

`validate='strict'` rejects missing columns, Arrow type mismatches, and non-nullability violations. `validate='autofix'` casts compatible columns to declared Arrow types after normal Arrow conversion.

Parameters

- **table** (*str*)
- **validate** (*Literal['strict', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Table | None

graphistry.PlotterBase.**maybe_cudf**()

graphistry.PlotterBase.**maybe_dask_cudf**()

graphistry.PlotterBase.**maybe_dask_dataframe**()

graphistry.PlotterBase.**maybe_polars**()

graphistry.PlotterBase.**maybe_spark**()

10.10.2.4 Plottable Interface

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

Bases: Protocol

DGL_graph: Any | None

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

Parameters

- **fg** (*Dict[str, Any] | None*)
- **bg** (*Dict[str, Any] | None*)
- **page** (*Dict[str, Any] | None*)
- **logo** (*Dict[str, Any] | None*)

Return type
Plottable

`base_url_client`(*v=None*)

Parameters
v (*str | None*)

Return type
str

`base_url_server`(*v=None*)

Parameters
v (*str | None*)

Return type
str

`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*, *point_longitude=None*, *point_latitude=None*, *dataset_id=None*, *url=None*, *nodes_file_id=None*, *edges_file_id=None*, *schema=None*)

Parameters

- **source** (*str | None*)
- **destination** (*str | None*)
- **node** (*str | None*)
- **edge** (*str | None*)
- **edge_title** (*str | None*)
- **edge_label** (*str | None*)
- **edge_color** (*str | None*)
- **edge_weight** (*str | None*)
- **edge_size** (*str | None*)
- **edge_opacity** (*str | None*)
- **edge_icon** (*str | None*)
- **edge_source_color** (*str | None*)
- **edge_destination_color** (*str | None*)
- **point_title** (*str | None*)
- **point_label** (*str | None*)
- **point_color** (*str | None*)
- **point_weight** (*str | None*)
- **point_size** (*str | None*)
- **point_opacity** (*str | None*)
- **point_icon** (*str | None*)

- **point_x** (*str | None*)
- **point_y** (*str | None*)
- **point_longitude** (*str | None*)
- **point_latitude** (*str | None*)
- **dataset_id** (*str | None*)
- **url** (*str | None*)
- **nodes_file_id** (*str | None*)
- **edges_file_id** (*str | None*)
- **schema** (*Any | None*)

Return type
Plottable

chain(*ops*)

ops is Union[List[ASTObject], Chain]

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

Return type
Plottable

chain_remote(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine=None*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*Literal['pandas', 'cudf'] | None*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
Plottable

chain_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine=None*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*Literal['pandas', 'cudf'] | None*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

client_protocol_hostname(*v=None*)

Parameters
v (*str | None*)

Return type
str

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

collections(*collections=None*, *show_collections=None*, *collections_global_node_color=None*, *collections_global_edge_color=None*, *validate='autofix'*, *warn=True*)

Parameters

- **collections** (*str | CollectionSet | CollectionIntersection | List[CollectionSet | CollectionIntersection] | None*)

- **show_collections** (*bool | None*)
- **collections_global_node_color** (*str | None*)
- **collections_global_edge_color** (*str | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*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*)

Return type
Plottable

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

Parameters

- **alg** (*str*)
- **out_col** (*str | None*)
- **directed** (*bool | None*)
- **use_vids** (*bool*)
- **params** (*dict*)
- **stringify_rich_types** (*bool*)

Return type
Plottable

compute_networkx(*alg*, *out_col=None*, *params=None*, *directed=True*, *G=None*)

Parameters

- **alg** (*str*)
- **out_col** (*str | None*)
- **params** (*Dict[str, Any] | None*)
- **directed** (*bool*)
- **G** (*Any | None*)

Return type
Plottable

copy()

Return type
Plottable

description(*description*)

Parameters
description (*str*)

Return type
Plottable

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

Return type
Plottable

embed(*relation*, *proto='DistMult'*, *embedding_dim=32*, *use_feat=False*, *X=None*, *epochs=2*, *batch_size=32*, *train_split=0.8*, *sample_size=1000*, *num_steps=50*, *lr=0.01*, *inplace=False*, *device='cpu'*, *evaluate=True*, **args*, ***kwargs*)

Parameters

- **relation** (*str*)
- **proto** (*str | Callable[[Any, Any, Any], Any] | None*)
- **embedding_dim** (*int*)
- **use_feat** (*bool*)
- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **epochs** (*int*)
- **batch_size** (*int*)
- **train_split** (*float | int*)
- **sample_size** (*int*)
- **num_steps** (*int*)
- **lr** (*float*)

- **inplace** (*bool | None*)
- **device** (*str | None*)
- **evaluate** (*bool*)

Return type
Plottable

encode_axis(*rows=[]*)

Parameters
rows (*List[Dict]*)

Return type
Plottable

encode_edge_badge(*column*, *position='TopRight'*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *color=Ellipsis*, *bg=Ellipsis*, *fg=Ellipsis*, *for_current=False*, *for_default=True*, *as_text=Ellipsis*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)
- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_edge_color(*column*, *palette=Ellipsis*, *as_categorical=Ellipsis*, *as_continuous=Ellipsis*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)

- **palette** (*List[str] | None*)
- **as_categorical** (*bool | None*)
- **as_continuous** (*bool | None*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_edge_icon(*column*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*str | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)
- **as_text** (*bool*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_edge_label(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_opacity(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`encode_edge_size`(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`encode_edge_title`(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`encode_edge_weight`(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`encode_point_badge`(*column*, *position='TopRight'*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *color=Ellipsis*, *bg=Ellipsis*, *fg=Ellipsis*, *for_current=False*, *for_default=True*, *as_text=Ellipsis*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)
- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)

- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_point_color(*column*, *palette=Ellipsis*, *as_categorical=Ellipsis*, *as_continuous=Ellipsis*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **palette** (*List[str] | None*)
- **as_categorical** (*bool | None*)
- **as_continuous** (*bool | None*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_point_icon(*column*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*str | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)
- **as_text** (*bool*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

`encode_point_label`(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`encode_point_opacity`(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`encode_point_size`(*column*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

`encode_point_title`(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

`fa2_layout`(*fa2_params=None*, *circle_layout_params=None*, *singleton_layout=None*, *partition_key=None*, *engine='auto'*, *allow_cpu_fallback=False*)

Parameters

- **fa2_params** (*Dict[str, Any] | None*)
- **circle_layout_params** (*Dict[str, Any] | None*)
- **singleton_layout** (*Callable[[Plottable, Tuple[float, float, float, float] | Any], Plottable] | None*)
- **partition_key** (*str | None*)

- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **allow_cpu_fallback** (*bool*)

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

filter_weighted_edges(*scale=1.0*, *index_to_nodes_dict=None*, *inplace=False*, *kind='nodes'*)

Parameters

- **scale** (*float*)
- **index_to_nodes_dict** (*Dict | None*)
- **inplace** (*bool*)
- **kind** (*Literal['nodes', 'edges']*)

Return type
Plottable | None

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*)

Return type
Plottable

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

Parameters

- **ig** (*Any*)
- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)

- **load_nodes** (*bool*)
- **load_edges** (*bool*)
- **merge_if_existing** (*bool*)

Return type
Plottable

from_networkx(*G*)

Parameters
G (*Any*)

Return type
Plottable

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

gfql_remote(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
Plottable

gfql_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

graph(*ig*)

Parameters
ig (*Any*)

Return type
Plottable

hop(*nodes*, *hops=1*, ***, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *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*, *include_zero_hop_seed=False*, *target_wave_front=None*, *engine='auto'*)

Parameters

- nodes (*DataFrame | None*)
- hops (*int | None*)
- min_hops (*int | None*)
- max_hops (*int | None*)
- output_min_hops (*int | None*)
- output_max_hops (*int | None*)
- label_node_hops (*str | None*)
- label_edge_hops (*str | None*)
- label_seeds (*bool*)
- 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*)
- include_zero_hop_seed (*bool*)
- target_wave_front (*DataFrame | None*)
- engine (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

Return type
Plottable

hypergraph(*raw_events: Any | None = None*, ***, *entity_types: List[str] | None = None*, *opts: dict = {}*, *drop_na: bool = True*, *drop_edge_attrs: bool = False*, *verbose: bool = True*, *direct: bool = False*, *engine: EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto'] = 'auto'*, *npartitions: int | None = None*, *chunksize: int | None = None*, *from_edges: bool = False*, *return_as: Literal['graph'] = 'graph'*) → *Plottable*

hypergraph(*raw_events: Any | None = None*, ***, *entity_types: List[str] | None = None*, *opts: dict = {}*, *drop_na: bool = True*, *drop_edge_attrs: bool = False*, *verbose: bool = True*, *direct: bool = False*, *engine: EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto'] = 'auto'*, *npartitions: int | None = None*, *chunksize: int | None = None*, *from_edges: bool = False*, *return_as: Literal['all']*) → HypergraphResult

hypergraph(*raw_events: Any | None = None*, ***, *entity_types: List[str] | None = None*, *opts: dict = {}*, *drop_na: bool = True*, *drop_edge_attrs: bool = False*, *verbose: bool = True*, *direct: bool = False*, *engine: EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto'] = 'auto'*, *npartitions: int | None = None*, *chunksize: int | None = None*, *from_edges: bool = False*, *return_as: Literal['entities', 'events', 'edges', 'nodes']*) → Any

igraph2pandas(*ig*)

Parameters
ig (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

infer_labels()

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*)

Return type
Plottable

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*)

Parameters

- **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*)

- **skip_styling** (*bool*)
- **render_to_disk** (*bool*)
- **path** (*str | None*)
- **format** (*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'] | None*)

Return type
Plottable

layout_igraph(*layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Parameters

- **layout** (*str*)
- **directed** (*bool | None*)
- **use_vids** (*bool*)
- **bind_position** (*bool*)
- **x_out_col** (*str*)
- **y_out_col** (*str*)
- **play** (*int | None*)
- **params** (*dict*)

Return type
Plottable

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*)

Return type
Plottable

materialize_nodes(*reuse=True*, *engine='auto'*)

Parameters

- **reuse** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

Return type
Plottable

name(*name*)

Parameters
name (*str*)

Return type
Plottable

networkx2pandas(*G*)

Parameters
G (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

networkx_checkoverlap(*g*)

Parameters
g (*Any*)

Return type
None

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

Parameters

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

Return type
Plottable

pandas2igraph(*edges*, *directed=True*)

Parameters

- **edges** (*DataFrame*)

- **directed** (*bool*)

Return type
Any

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

Parameters

- **graph_transform** (*Callable*)
- **args** (*Any*)
- **kwargs** (*Any*)

Return type
Plottable

plot(*graph=None*, *nodes=None*, *name=None*, *description=None*, *render='auto'*, *skip_upload=False*, *as_files=False*, *memoize=True*, *erase_files_on_fail=True*, *extra_html=''*, *override_html_style=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Parameters

- **graph** (*Any | None*)
- **nodes** (*Any | None*)
- **name** (*str | None*)
- **description** (*str | None*)
- **render** (*bool | Literal['auto'] | ~typing.Literal['g', 'url', 'ipython', 'databricks', 'browser'] | None*)
- **skip_upload** (*bool*)
- **as_files** (*bool*)
- **memoize** (*bool*)
- **erase_files_on_fail** (*bool*)
- **extra_html** (*str*)
- **override_html_style** (*str | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)

Return type
Any

privacy(*mode=None*, *notify=None*, *invited_users=None*, *message=None*, *mode_action=None*)

Parameters

- **mode** (*Literal['private', 'organization', 'public'] | None*)
- **notify** (*bool | None*)
- **invited_users** (*List[str] | None*)
- **message** (*str | None*)
- **mode_action** (*Literal['10', '20'] | None*)

Return type
Plottable

`protocol`(*v=None*)

Parameters
v (*str | None*)

Return type
str

`prune_self_edges`()

Return type
Plottable

`python_remote_g`(*code*, *api_token=None*, *dataset_id=None*, *format='parquet'*, *output_type='all'*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape'] | ~typing.Literal['table', 'shape'] | ~typing.Literal['json'] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
Plottable

`python_remote_json`(*code*, *api_token=None*, *dataset_id=None*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
Any

python_remote_table(*code*, *api_token=None*, *dataset_id=None*, *format='parquet'*, *output_type='table'*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **output_type** (*Literal['table', 'shape'] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
DataFrame

reset_caches()

Return type
None

scene_settings(*menu=None*, *info=None*, *show_arrows=None*, *point_size=None*, *edge_curvature=None*, *edge_opacity=None*, *point_opacity=None*)

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*)

Return type
Plottable

search(*query*, *cols=None*, *thresh=5000*, *fuzzy=True*, *top_n=10*)

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*)

Parameters

- **query** (*str*)
- **scale** (*float*)
- **top_n** (*int*)
- **thresh** (*float*)
- **broader** (*bool*)
- **inplace** (*bool*)

Return type
Plottable

`server`(*v=None*)

Parameters
v (*str | None*)

Return type
str

`session: ClientSession`

`settings`(*height=None*, *url_params=None*, *render=None*, *validate='autofix'*, *warn=True*)

Parameters

- **height** (*int | None*)
- **url_params** (*Dict[str, None | str | int | float | bool | List[SettingsValue] | Dict[str, SettingsValue]] | None*)
- **render** (*bool | Literal['auto'] | ~typing.Literal['g', 'url', 'ipython', 'databricks', 'browser'] | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Plottable

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

Parameters

- **fg** (*Dict[str, Any] | None*)
- **bg** (*Dict[str, Any] | None*)
- **page** (*Dict[str, Any] | None*)
- **logo** (*Dict[str, Any] | None*)

Return type
Plottable

`to_arrow`(*table=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*, *schema_table='edges'*)

Parameters

- **table** (*Any | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)
- **schema_table** (*str*)

Return type
Any | None

to_cudf()

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*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *use_vids=False*)

Parameters

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

Return type
Any

to_pandas()

Return type
Plottable

transform(*df: DataFrame*, *y: DataFrame | None = None*, *kind: str = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[True] = True*, *scaled: bool = True*, *verbose: bool = False*) → *Plottable*

transform(*df: DataFrame*, *y: DataFrame | None = None*, *kind: str = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[False]*, *scaled: bool = True*, *verbose: bool = False*) → Tuple[DataFrame, DataFrame]

transform_umap(*df: DataFrame*, *y: DataFrame | None = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[True] = True*, *fit_umap_embedding: bool = True*, *umap_transform_kwargs: Dict[str, Any] = {}*) → *Plottable*

transform_umap(*df: DataFrame*, *y: DataFrame | None = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[False]*, *fit_umap_embedding: bool = True*, *umap_transform_kwargs: Dict[str, Any] = {}*) → Tuple[DataFrame, DataFrame, DataFrame]

umap(*X: XSymbolic = None*, *y: YSymbolic = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *scale: float = 1.0*, *n_neighbors: int = 12*, *min_dist: float = 0.1*, *spread: float = 0.5*, *local_connectivity: int = 1*, *repulsion_strength: float = 1*, *negative_sample_rate: int = 5*, *n_components: int = 2*, *metric: str = 'euclidean'*, *suffix: str = ''*, *play: int | None = 0*, *encode_position: bool = True*, *encode_weight: bool = True*, *dbscan: bool = False*, *engine: Literal['auto', 'cuml', 'umap_learn'] = 'auto'*, *feature_engine: str = 'auto'*, *inplace: Literal[False] = False*, *memoize: bool = True*, *umap_kwargs: Dict[str, Any] = {}*, *umap_fit_kwargs: Dict[str, Any] = {}*, *umap_transform_kwargs: Dict[str, Any] = {}*, ***featurize_kwargs: Any*) → *Plottable*

umap(*X: XSymbolic = None*, *y: YSymbolic = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *scale: float = 1.0*, *n_neighbors: int = 12*, *min_dist: float = 0.1*, *spread: float = 0.5*, *local_connectivity: int = 1*, *repulsion_strength: float = 1*, *negative_sample_rate: int = 5*, *n_components: int = 2*, *metric: str = 'euclidean'*, *suffix: str = ''*, *play: int | None = 0*, *encode_position: bool = True*, *encode_weight: bool = True*, *dbscan: bool = False*, *engine: Literal['auto', 'cuml', 'umap_learn'] = 'auto'*, *feature_engine: str = 'auto'*, ***, *inplace: Literal[True]*, *memoize: bool = True*, *umap_kwargs: Dict[str, Any] = {}*, *umap_fit_kwargs: Dict[str, Any] = {}*, *umap_transform_kwargs: Dict[str, Any] = {}*, ***featurize_kwargs: Any*) → None

umap_fit(*X*, *y=None*, *umap_fit_kwargs={}*)

Parameters

- **X** (*DataFrame*)
- **y** (*DataFrame | None*)
- **umap_fit_kwargs** (*Dict[str, Any]*)

Return type
Plottable

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

- **res** (Plottable)
- **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['auto', 'cuml', 'umap_learn']*)
- **suffix** (*str*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

Return type
Plottable

upload(*memoize=True*, *erase_files_on_fail=True*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Parameters

- **memoize** (*bool*)
- **erase_files_on_fail** (*bool*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)

Return type
Plottable

property url: str | None

validate_arrow_schema(*table='edges'*, ***, *validate='strict'*, *warn=True*)

Parameters

- **table** (*str*)
- **validate** (*Literal['strict', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Any | None

10.10.3 GFQL API Reference

10.10.3.1 AST Objects

ASTSerializable

class graphistry.compute.ASTSerializable.ASTSerializable

Bases: ABC

Internal, not intended for use outside of this module. Class name becomes o['type'], and all non reserved_fields become JSON-typed key

classmethod from_json(*d*, *validate=True*)

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- **d** (*Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **validate** (*bool*)

Return type
ASTSerializable

reserved_fields = ['type']

to_json(*validate=True*)

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, None | bool | str | float | int | *List*[*Any*] | *Dict*[str, *Any*]]

validate(*collect_all=False*)

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
collect_all (*bool*)

Return type
List[GFQLValidationError] | None

ASTObject

class graphistry.compute.ast.**ASTObject**(*name=None*)

Bases: ASTSerializable

Internal, not intended for use outside of this module. These are operator-level expressions used as g.chain(List<ASTObject>)

Parameters
name (*str | None*)

abstract execute(*g*, *prev_node_wavefront*, *target_wave_front*, *engine*)

Parameters

- **g** (Plottable)
- **prev_node_wavefront** (*Any | None*)
- **target_wave_front** (*Any | None*)
- **engine** (*Engine*)

Return type
Plottable

classmethod from_json(*d*, *validate=True*)

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- **d** (*Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **validate** (*bool*)

Return type
ASTSerializable

reserved_fields = ['type']

abstract reverse()

Return type
ASTObject

`to_json`(*validate=True*)

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, None | bool | str | float | int | *List*[*Any*] | *Dict*[str, *Any*]]

`validate`(*collect_all=False*)

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

ASTEdge

`class graphistry.compute.ast.ASTEdge`(*direction='forward'*, *edge_match=None*, *hops=1*, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *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*, *prune_to_endpoints=False*, *include_zero_hop_seed=False*)

Bases: `ASTObject`

Internal, not intended for use outside of this module.

Parameters

- `direction` (*Literal['forward', 'reverse', 'undirected'] | None*)
- `edge_match` (*dict | None*)
- `hops` (*int | None*)
- `min_hops` (*int | None*)
- `max_hops` (*int | None*)
- `output_min_hops` (*int | None*)
- `output_max_hops` (*int | None*)
- `label_node_hops` (*str | None*)

- **label_edge_hops** (*str | None*)
- **label_seeds** (*bool*)
- **to_fixed_point** (*bool*)
- **source_node_match** (*dict | None*)
- **destination_node_match** (*dict | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **name** (*str | None*)
- **prune_to_endpoints** (*bool*)
- **include_zero_hop_seed** (*bool*)

execute(*g*, *prev_node_wavefront*, *target_wave_front*, *engine*)

Parameters

- **g** (Plottable)
- **prev_node_wavefront** (*Any | None*)
- **target_wave_front** (*Any | None*)
- **engine** (*Engine*)

Return type
Plottable

classmethod from_json(*d*, *validate=True*)

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- **d** (*Dict[str, Any]*)
- **validate** (*bool*)

Return type
ASTEdge

is_simple_single_hop()

Check if edge is single-hop without hop labels (safe to skip backward hop call).

Return type
bool

reserved_fields = ['type']

`reverse()`

Return type
ASTEdge

`to_json(validate=True)`

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, *Any*]

`validate(collect_all=False)`

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

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

`classmethod from_json(d, validate=True)`

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- `d` (*Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- `validate` (*bool*)

Return type
ASTSerializable

`reserved_fields = ['type']`

`to_json`(*validate=True*)

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, None | bool | str | float | int | *List*[*Any*] | *Dict*[str, *Any*]]

`validate`(*collect_all=False*)

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

10.10.3.2 GFQL Chain Matcher

Chain enables combining multiple matchers into a single matcher, e.g., for mining paths and subgraphs.

Same-path constraints are expressed via *where* on a Chain; see *GFQL WHERE (Same-Path Constraints)*.

`class graphistry.compute.chain.Chain`(*chain*, *where=None*, *validate=True*)

Bases: `ASTSerializable`

Parameters

- `chain` (*List[ASTObject]*)
- `where` (*Sequence[WhereComparison] | None*)
- `validate` (*bool*)

`classmethod from_json`(*d*, *validate=True*)

Convert a JSON AST into a list of ASTObjects

Parameters

- `d` (*Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- `validate` (*bool*)

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*[*Any*] | *Dict*[str, *Any*]]

`validate`(*collect_all=False*)

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

`validate_schema`(*g*, *collect_all=False*)

Validate this chain against a graph's schema without executing.

Args:
g: Graph to validate against collect_all: If True, collect all errors. If False, raise on first.

Returns:
If collect_all=True: List of errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLSchemaError: If collect_all=False and validation fails

Parameters

- `g` (Plottable)
- `collect_all` (*bool*)

Return type
List[GFQLSchemaError] | None

`graphistry.compute.chain.chain`(*self*, *ops*, *engine=EngineAbstract.AUTO*, *validate_schema=True*, *policy=None*, *context=None*, *start_nodes=None*)

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
- **validate_schema** (*bool*) – Whether to validate the chain against the graph schema before executing
- **policy** – Optional policy dict for hooks
- **context** – Optional ExecutionContext for tracking execution state
- **start_nodes** (*Any | None*) – Optional node wavefront for the first traversal step
- **self** (Plottable)
- **engine** (*EngineAbstract | str*)

Returns
Plotter

Return type
Plotter

graphistry.compute.chain.**combine_steps**(*g*, *kind*, *steps*, *engine*, *label_steps=None*)

Collect nodes and edges, taking care to deduplicate and tag any names

Parameters

- **g** (Plottable)
- **kind** (*str*)
- **steps** (*List[Tuple[ASTObject, Plottable]]*)
- **engine** (*Engine*)
- **label_steps** (*List[Tuple[ASTObject, Plottable]] | None*)

Return type
Any

10.10.3.3 GFQL Cypher Syntax API Reference

This page documents the Python helper APIs behind PyGraphistry's Cypher-syntax support in GFQL.

- **Cypher** is a graph query language popularized by Neo4j and related tools.
- **GFQL** is PyGraphistry's dataframe-native graph query language: the first fully vectorized graph query implementation with an open-source GPU runtime.
- PyGraphistry supports a read-only Cypher surface on bound graphs that can be parsed, validated, compiled, and executed through GFQL's columnar engine.

Use this page when you want to:

- run a supported Cypher query through `g.gfql("MATCH ...")` on a bound graph
- preflight a query with `parse_cypher()` or `compile_cypher()`
- translate a supported query into a GFQL `Chain` programmatically

This page is an API reference, not the main tutorial. It covers Cypher syntax through `g.gfql("MATCH ...")` on a bound graph, which is the on-ramp for Cypher users who want familiar graph-pattern syntax without giving up GFQL's fully vectorized dataframe/GPU execution model. For **remote GFQL** execution on Graphistry infrastructure, use `g.gfql_remote([...])`. For **remote database Cypher** over Bolt/Neo4j-style backends, use `g.cypher(...)` or `graphistry.cypher(...)`.

See also:

- *Cypher Syntax In GFQL* for the user-facing guide, supported syntax forms, and current boundaries
- *GFQL Remote Mode* for remote GFQL execution
- *GFQL: The Dataframe-Native Graph Query Language* or *GFQL Quick Reference* if you are new to GFQL itself
- *Cypher to GFQL Python & Wire Protocol Mapping* for translation-oriented guidance

Start Here: Cypher Syntax Through `g.gfql()`

If you only want to run a supported Cypher query on a bound graph, start with `g.gfql(...)`. The method always returns a `Plottable`, but the result shape depends on what you ask for:

- native GFQL chains preserve graph state in `_nodes` and `_edges`
- Cypher `RETURN` projections surface tabular rows in the returned `_nodes` dataframe

For the broader graph-state vs row-state model, see *GFQL Quick Reference*.

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

# Graph/subgraph result: native GFQL chains stay in graph state.
g2 = g1.gfql([n({"type": "Person"}), e_forward(), n()])

# Row/table result: Cypher RETURN projections surface rows in _nodes.
df = g1.gfql(
    "MATCH (p:Person) RETURN p.name AS name ORDER BY name DESC LIMIT $top_n",
    params={"top_n": 5},
)._nodes
```

When the query argument is a string, the `language` selector defaults to `"cypher"`. Top-level `params=...` is currently only supported for string query compilation; regular GFQL AST / Chain inputs use normal Python values in the AST itself.

Helper Functions

Import the helpers from `graphistry.compute.gfql.cypher`:

```
from graphistry.compute.gfql.cypher import (
    parse_cypher,
    compile_cypher,
    cypher_to_gfql,
    gfql_from_cypher,
)
```

`parse_cypher(query)`

- Parses supported Cypher text into the typed AST used by the GFQL Cypher compiler.
- Returns `CypherQuery` or `CypherUnionQuery`.

`compile_cypher(query, params=None)`

- **Deprecated** — scheduled for removal in a future release (tracked in https://github.com/graphistry/pygraphistry/issues/1169).
- Parses and lowers a supported Cypher query into the compiled program used by `g.gfql("MATCH ...")` execution.
- Returns compiler-internal shapes (`CompiledCypherQuery` / `CompiledCypherUnionQuery` / `CompiledCypherGraphQuery`) that are also deprecated internals scheduled for full removal.
- Prefer `g.gfql("...", language="cypher")` for execution and `cypher_to_gfql(...)` / `gfql_from_cypher(...)` for single-chain translation.

`cypher_to_gfql(query, params=None)`

- Compiles a supported Cypher query into a single GFQL `Chain`.
- Use this when you want the translated GFQL chain object instead of immediate execution.
- Queries that require `UNION` or a row-returning `CALL` flow intentionally raise `GFQLValidationError` here; execute those directly through `g.gfql("...", language="cypher")` instead.

`gfql_from_cypher(query, params=None)`

- Alias for `cypher_to_gfql(...)` for callers that prefer GFQL-first naming.

10.10.3.4 GFQL Edge Matchers

e_forward

`graphistry.compute.ast.e_forward = <class 'graphistry.compute.ast.ASTEdgeForward'>`

Internal, not intended for use outside of this module.

Parameters

- **edge_match** (*dict | None*)
- **hops** (*int | None*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)
- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)

- **label_seeds** (*bool*)
- **source_node_match** (*dict | None*)
- **destination_node_match** (*dict | None*)
- **to_fixed_point** (*bool*)
- **name** (*str | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **prune_to_endpoints** (*bool*)
- **include_zero_hop_seed** (*bool*)

e_forward

Primary alias for the `graphistry.compute.ast.ASTEdgeForward` class.

> **Note**
>
> While *e_forward* is the preferred alias for this class, the methods and attributes are defined under `graphistry.compute.ast.ASTEdgeForward`.

class graphistry.compute.ast.**ASTEdgeForward**(*edge_match=None*, *hops=1*, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *source_node_match=None*, *destination_node_match=None*, *to_fixed_point=False*, *name=None*, *source_node_query=None*, *destination_node_query=None*, *edge_query=None*, *prune_to_endpoints=False*, *include_zero_hop_seed=False*)

Bases: `ASTEdge`

Internal, not intended for use outside of this module.

Parameters

- **edge_match** (*dict | None*)
- **hops** (*int | None*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)
- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)
- **label_seeds** (*bool*)
- **source_node_match** (*dict | None*)

- **destination_node_match** (*dict | None*)
- **to_fixed_point** (*bool*)
- **name** (*str | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **prune_to_endpoints** (*bool*)
- **include_zero_hop_seed** (*bool*)

direction: Literal['forward', 'reverse', 'undirected']

execute(*g*, *prev_node_wavefront*, *target_wave_front*, *engine*)

Parameters

- **g** (Plottable)
- **prev_node_wavefront** (*Any | None*)
- **target_wave_front** (*Any | None*)
- **engine** (*Engine*)

Return type
Plottable

classmethod from_json(*d*, *validate=True*)

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- **d** (*Dict[str, Any]*)
- **validate** (*bool*)

Return type
ASTEdge

is_simple_single_hop()

Check if edge is single-hop without hop labels (safe to skip backward hop call).

Return type
bool

reserved_fields = ['type']

`reverse()`

Return type
ASTEdge

`to_json(validate=True)`

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, *Any*]

`validate(collect_all=False)`

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

e_reverse

`graphistry.compute.ast.e_reverse = <class 'graphistry.compute.ast.ASTEdgeReverse'>`

Internal, not intended for use outside of this module.

Parameters

- `edge_match` (*dict | None*)
- `hops` (*int | None*)
- `min_hops` (*int | None*)
- `max_hops` (*int | None*)
- `output_min_hops` (*int | None*)
- `output_max_hops` (*int | None*)
- `label_node_hops` (*str | None*)
- `label_edge_hops` (*str | None*)
- `label_seeds` (*bool*)
- `source_node_match` (*dict | None*)
- `destination_node_match` (*dict | None*)
- `to_fixed_point` (*bool*)

- **name** (*str | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **prune_to_endpoints** (*bool*)
- **include_zero_hop_seed** (*bool*)

e_reverse

Primary alias for the `graphistry.compute.ast.ASTEdgeReverse` class.

> **Note**
>
> While *e_reverse* is the preferred alias for this class, the methods and attributes are defined under `graphistry.compute.ast.ASTEdgeReverse`.

`class graphistry.compute.ast.ASTEdgeReverse`(*edge_match=None*, *hops=1*, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *source_node_match=None*, *destination_node_match=None*, *to_fixed_point=False*, *name=None*, *source_node_query=None*, *destination_node_query=None*, *edge_query=None*, *prune_to_endpoints=False*, *include_zero_hop_seed=False*)

Bases: `ASTEdge`

Internal, not intended for use outside of this module.

Parameters

- **edge_match** (*dict | None*)
- **hops** (*int | None*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)
- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)
- **label_seeds** (*bool*)
- **source_node_match** (*dict | None*)
- **destination_node_match** (*dict | None*)
- **to_fixed_point** (*bool*)
- **name** (*str | None*)
- **source_node_query** (*str | None*)

- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **prune_to_endpoints** (*bool*)
- **include_zero_hop_seed** (*bool*)

direction: Literal['forward', 'reverse', 'undirected']

execute(*g*, *prev_node_wavefront*, *target_wave_front*, *engine*)

Parameters

- **g** (Plottable)
- **prev_node_wavefront** (*Any | None*)
- **target_wave_front** (*Any | None*)
- **engine** (*Engine*)

Return type
Plottable

classmethod from_json(*d*, *validate=True*)

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- **d** (*Dict[str, Any]*)
- **validate** (*bool*)

Return type
ASTEdge

is_simple_single_hop()

Check if edge is single-hop without hop labels (safe to skip backward hop call).

Return type
bool

reserved_fields = ['type']

reverse()

Return type
ASTEdge

to_json(*validate=True*)

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, *Any*]

`validate`(*collect_all=False*)

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

e

`graphistry.compute.ast.e = <class 'graphistry.compute.ast.ASTEdgeUndirected'>`

Internal, not intended for use outside of this module.

Parameters

- `edge_match` (*dict | None*)
- `hops` (*int | None*)
- `min_hops` (*int | None*)
- `max_hops` (*int | None*)
- `output_min_hops` (*int | None*)
- `output_max_hops` (*int | None*)
- `label_node_hops` (*str | None*)
- `label_edge_hops` (*str | None*)
- `label_seeds` (*bool*)
- `source_node_match` (*dict | None*)
- `destination_node_match` (*dict | None*)
- `to_fixed_point` (*bool*)
- `name` (*str | None*)
- `source_node_query` (*str | None*)
- `destination_node_query` (*str | None*)
- `edge_query` (*str | None*)
- `prune_to_endpoints` (*bool*)

- **include_zero_hop_seed** (*bool*)

e

Primary alias for the `graphistry.compute.ast.ASTEdgeUndirected` class.

> **Note**
>
> While *e* is the preferred alias for this class, the methods and attributes are defined under `graphistry.compute.ast.ASTEdgeUndirected`.

e_undirected

Secondary alias for the `graphistry.compute.ast.ASTEdgeUndirected` class.

class graphistry.compute.ast.**ASTEdgeUndirected**(*edge_match=None*, *hops=1*, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *source_node_match=None*, *destination_node_match=None*, *to_fixed_point=False*, *name=None*, *source_node_query=None*, *destination_node_query=None*, *edge_query=None*, *prune_to_endpoints=False*, *include_zero_hop_seed=False*)

Bases: `ASTEdge`

Internal, not intended for use outside of this module.

Parameters

- **edge_match** (*dict | None*)
- **hops** (*int | None*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)
- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)
- **label_seeds** (*bool*)
- **source_node_match** (*dict | None*)
- **destination_node_match** (*dict | None*)
- **to_fixed_point** (*bool*)
- **name** (*str | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **prune_to_endpoints** (*bool*)

- **include_zero_hop_seed** (*bool*)

direction: Literal['forward', 'reverse', 'undirected']

execute(*g*, *prev_node_wavefront*, *target_wave_front*, *engine*)

Parameters

- **g** (Plottable)
- **prev_node_wavefront** (*Any | None*)
- **target_wave_front** (*Any | None*)
- **engine** (*Engine*)

Return type
Plottable

classmethod from_json(*d*, *validate=True*)

Given c.to_json(), hydrate back c

Args:
d: Dictionary from to_json() validate: If True (default), validate after parsing

Returns:
Hydrated AST object

Raises:
GFQLValidationError: If validate=True and validation fails

Parameters

- **d** (*Dict[str, Any]*)
- **validate** (*bool*)

Return type
ASTEdge

is_simple_single_hop()

Check if edge is single-hop without hop labels (safe to skip backward hop call).

Return type
bool

reserved_fields = ['type']

reverse()

Return type
ASTEdge

to_json(*validate=True*)

Returns JSON-compatible dictionry {"type": "ClassName", "arg1": val1, ...} Emits all non-reserved instance fields

Return type
Dict[str, *Any*]

`validate`(*collect_all=False*)

Validate this AST node.

Args:

collect_all: If True, collect all errors instead of raising on first.
If False (default), raise on first error.

Returns:
If collect_all=True: List of validation errors (empty if valid) If collect_all=False: None if valid

Raises:
GFQLValidationError: If collect_all=False and validation fails

Parameters
`collect_all` (*bool*)

Return type
List[GFQLValidationError] | None

10.10.3.5 GFQL Hop Matcher

Hop is the core primitive behind a single matcher step in chain.

Calling hop directly has performance benefits over calling chain so may be helpful for larger graphs.

For cross-step constraints, use *g.gfql([...], where=[...])* (or the explicit *Chain(..., where=[...])* form); see *GFQL WHERE (Same-Path Constraints)*. Graph hop/traversal operations for PyGraphistry.

NOTE: Excluded from pyre (.pyre_configuration) - hop() complexity causes hang. Use mypy.

`graphistry.compute.hop.hop`(*self*, *nodes=None*, *hops=1*, ***, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *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*, *include_zero_hop_seed=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). Shorthand for max_hops. min_hops/max_hops: inclusive traversal bounds; defaults preserve legacy behavior (min=1 unless max=0; max defaults to hops). output_min_hops/output_max_hops: optional output slice applied after traversal; defaults keep all traversed hops up to max_hops. Useful for showing a subrange (e.g., min/max = 2..4 but display only hops 3..4). label_node_hops/label_edge_hops: optional column names for hop numbers (omit or None to skip). Nodes record the first retained hop step they are reached (1 = first expansion); when min_hops prunes shorter branches, labels reflect the shortest retained path. Edges record the hop step that traversed them. label_seeds: when True and labeling, also write hop 0 for seed nodes in the node label column. 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 include_zero_hop_seed: internal Cypher opt-in for exact zero-hop path semantics Note: chain() reverse passes set return_as_wave_front=True and use target_wave_front to constrain reachability. 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*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)
- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)
- **label_seeds** (*bool*)
- **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*)
- **include_zero_hop_seed** (*bool*)
- **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

10.10.3.6 GFQL Node Matchers

n

n

Primary alias for the `graphistry.compute.ast.ASTNode` class.

> **Note**
>
> While *n* is the preferred alias for this class, the methods and attributes are defined under `graphistry.compute.ast.ASTNode`.

Internal, not intended for use outside of this module.

10.10.3.7 GFQL Attribute Matchers

For cross-step comparisons, use *g.gfql([...], where=[...])* (or the explicit *Chain(..., where=[...])* form); see *GFQL WHERE (Same-Path Constraints)*.

Categorical

`class graphistry.compute.predicates.categorical.Duplicated(keep='first')`

Bases: `ASTPredicate`

Parameters
keep (*Literal['first', 'last', False]*)

`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[int | float | str | number | None | Timestamp | datetime | date | time | DateTimeWire | DateWire | TimeWire | TemporalValue]*)

`to_json`(*validate=True*)

Override to handle temporal values in options

Return type
dict

`graphistry.compute.predicates.is_in.is_in`(*options*)

Parameters
options (*List[int | float | str | number | None | Timestamp | datetime | date | time | DateTimeWire | DateWire | TimeWire | TemporalValue]*)

Return type
IsIn

Numeric

`class graphistry.compute.predicates.numeric.Between`(*lower*, *upper*, *inclusive=True*)

Bases: `ASTPredicate`

Parameters

- **lower** (*float*)
- **upper** (*float*)
- **inclusive** (*bool*)

`class graphistry.compute.predicates.numeric.EQ`(*val*)

Bases: `NumericASTPredicate`

Parameters
val (*int | float*)

`static op`(*a*, *b*, */*)

Same as a == b.

`class graphistry.compute.predicates.numeric.GE`(*val*)

Bases: `NumericASTPredicate`

Parameters
val (*int | float*)

`static op`(*a*, *b*, */*)

Same as a >= b.

`class graphistry.compute.predicates.numeric.GT`(*val*)

Bases: `NumericASTPredicate`

Parameters
val (*int | float*)

`static op`(*a*, *b*, */*)

Same as a > b.

`class graphistry.compute.predicates.numeric.IsNA`

Bases: `ASTPredicate`

class graphistry.compute.predicates.numeric.**LE**(*val*)

Bases: NumericASTPredicate

Parameters
val (*int* / *float*)

static **op**(*a*, *b*, */*)

Same as a <= b.

class graphistry.compute.predicates.numeric.**LT**(*val*)

Bases: NumericASTPredicate

Parameters
val (*int* / *float*)

static **op**(*a*, *b*, */*)

Same as a < b.

class graphistry.compute.predicates.numeric.**NE**(*val*)

Bases: NumericASTPredicate

Parameters
val (*int* / *float*)

static **op**(*a*, *b*, */*)

Same as a != b.

class graphistry.compute.predicates.numeric.**NotNA**

Bases: ASTPredicate

class graphistry.compute.predicates.numeric.**NumericASTPredicate**(*val*)

Bases: ASTPredicate

Parameters
val (*int* / *float*)

op: ClassVar[Any]

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*)

class graphistry.compute.predicates.str.**Endswith**(*pat*, *case=True*, *na=None*)

Bases: _BoundaryStringPredicate

Parameters

- **pat** (*str | tuple*)
- **case** (*bool*)
- **na** (*bool | None*)

class graphistry.compute.predicates.str.**Fullmatch**(*pat*, *case=True*, *flags=0*, *na=None*)

Bases: _RegexStringPredicate

Parameters

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)

class graphistry.compute.predicates.str.**IsAlnum**

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.**IsAlpha**

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.IsDecimal

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.IsDigit

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.IsLower

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.IsNull

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.IsNumeric

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.IsSpace

Bases: _CallablePredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.**IsTitle**

Bases: _CallablePredicate

static **predicate**(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.**IsUpper**

Bases: _CallablePredicate

static **predicate**(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.**Match**(*pat*, *case=True*, *flags=0*, *na=None*)

Bases: _RegexStringPredicate

Parameters

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)

class graphistry.compute.predicates.str.**NeverMatch**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**NotNull**

Bases: _CallablePredicate

static **predicate**(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.str.**Startswith**(*pat*, *case=True*, *na=None*)

Bases: _BoundaryStringPredicate

Parameters

- **pat** (*str | tuple*)
- **case** (*bool*)
- **na** (*bool | 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*, *case=True*, *na=None*)

Return whether a given pattern or tuple of patterns is at the end of a string.

Parameters

- **pat** (*str | tuple*) – Pattern (str) or tuple of patterns to match at end of string. When tuple, returns True if the string ends with ANY pattern (OR logic).
- **case** (*bool*) – If True, case-sensitive matching (default: True).
- **na** (*bool | None*) – Fill value for missing values (default: None).

Returns
Endswith predicate.

Return type
Endswith

Examples

```
>>> n({"email": endswith(".com")})
>>> n({"email": endswith(".COM", case=False)})
>>> n({"filename": endswith((".txt", ".csv"))})
>>> n({"filename": endswith((".TXT", ".CSV"), case=False)})
```

graphistry.compute.predicates.str.**fullmatch**(*pat*, *case=True*, *flags=0*, *na=None*)

Return whether a given pattern matches the entire string

Unlike match() which matches from the start, fullmatch() requires the pattern to match the entire string. This is useful for exact validation of formats like emails, phone numbers, or IDs.

Args:
pat: Regular expression pattern to match against entire string case: If True, case-sensitive matching (default: True) flags: Regex flags (e.g., re.IGNORECASE, re.MULTILINE) na: Fill value for missing values (default: None)

Returns:
Fullmatch predicate

Examples:

```
>>> # Exact digit match
>>> n({"code": fullmatch(r"\d{3}")})  # Matches "123" but not "123abc"
>>>
>>> # Case-insensitive email validation
>>> n({"email": fullmatch(r"[a-z]+@[a-z]+\.com", case=False)})
>>>
>>> # With regex flags
>>> import re
>>> n({"id": fullmatch(r"[A-Z]{3}-\d{4}", flags=re.IGNORECASE)})
```

Parameters

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)

Return type
Fullmatch

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.**never_match**()

Return type
NeverMatch

graphistry.compute.predicates.str.**notnull**()

Return whether a given string is not null

Return type
NotNull

graphistry.compute.predicates.str.**startswith**(*pat*, *case=True*, *na=None*)

Return whether a given pattern or tuple of patterns is at the start of a string.

Parameters

- **pat** (*str / tuple*) – Pattern (str) or tuple of patterns to match at start of string. When tuple, returns True if the string starts with ANY pattern (OR logic).
- **case** (*bool*) – If True, case-sensitive matching (default: True).
- **na** (*bool / None*) – Fill value for missing values (default: None).

Returns
Startswith predicate.

Return type
Startswith

Examples

```
>>> n({"name": startswith("John")})
>>> n({"name": startswith("john", case=False)})
>>> n({"filename": startswith(("test_", "demo_"))})
>>> n({"filename": startswith(("TEST", "DEMO"), case=False)})
```

Temporal

class graphistry.compute.predicates.temporal.**IsLeapYear**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.temporal.**IsMonthEnd**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.temporal.**IsMonthStart**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.temporal.**IsQuarterEnd**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.temporal.**IsQuarterStart**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.temporal.**IsYearEnd**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

class graphistry.compute.predicates.temporal.**IsYearStart**

Bases: _DatetimePropertyPredicate

static predicate(*s*)

Parameters
s (*Any*)

Return type
Any

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

Temporal Values

class graphistry.compute.ast_temporal.**DateTimeValue**(*value*, *timezone='UTC'*)

Bases: `TemporalValue`

Tagged datetime value with timezone support

Parameters

- **value** (*str*)
- **timezone** (*str*)

as_pandas_value()

Convert to pandas-compatible value for comparison

Return type
Timestamp

classmethod from_datetime(*dt*)

Create from Python datetime

Parameters
dt (*datetime*)

Return type
DateTimeValue

classmethod from_pandas_timestamp(*ts*)

Create from pandas Timestamp

Parameters
ts (*Timestamp*)

Return type
DateTimeValue

to_json()

Return dict for tagged temporal value

Return type
DateTimeWire

class graphistry.compute.ast_temporal.DateValue(*value*)

Bases: _ScalarTemporalValue

Tagged date value

Parameters
value (*str*)

as_pandas_value()

Convert to pandas-compatible value for comparison

Return type
Timestamp

classmethod from_date(*d*)

Create from Python date

Parameters
d (*date*)

Return type
DateValue

class graphistry.compute.ast_temporal.TemporalValue

Bases: ABC

Base class for temporal values with tagging support

abstract as_pandas_value()

Convert to pandas-compatible value for comparison

Return type
Any

abstract to_json()

Serialize to JSON-compatible dictionary

Return type
DateTimeWire | *DateWire* | *TimeWire*

class graphistry.compute.ast_temporal.TimeValue(*value*)

Bases: _ScalarTemporalValue

Tagged time value

Parameters
value (*str*)

as_pandas_value()

Convert to pandas-compatible value for comparison

Return type
time

classmethod from_time(*t*)

Create from Python time

Parameters
t (*time*)

Return type
TimeValue

graphistry.compute.ast_temporal.temporal_value_from_json(*d*)

Factory function to create temporal value from JSON dict

Parameters
d (*Dict[str, Any]*)

Return type
TemporalValue

10.10.3.8 GFQL Schema

Experimental public declarative graph schema model for GFQL validation.

The import path is public, but this schema surface is still experimental while inference, coercion, remote transport, and planner use are developed.

class graphistry.schema.**EdgeTopology**(*relationship_type*, *source_labels*, *destination_labels*)

Bases: object

Experimental source/destination-label contract for a relationship type.

Parameters

- **relationship_type** (*str*)
- **source_labels** (*FrozenSet[str]*)
- **destination_labels** (*FrozenSet[str]*)

as_metadata()

Return type
Dict[str, object]

destination_labels: FrozenSet[str]

classmethod from_metadata(*value*)

Import topology from the metadata shape emitted by as_metadata().

Parameters
value (*Mapping[str, object]*)

Return type
EdgeTopology

relationship_type: str

source_labels: FrozenSet[str]

class graphistry.schema.**EdgeType**(*name*, *source*, *destination*, *properties=None*)

Bases: object

Experimental declarative edge contract with topology constraints.

Parameters

- **name** (*str*)
- **source** (*FrozenSet[str]*)
- **destination** (*FrozenSet[str]*)

- **properties** (*Mapping[str, NodeRef | EdgeRef | ScalarType | PathType | ListType]*)

property columns: FrozenSet[str]

Columns admitted for this edge type, including relationship label.

destination: FrozenSet[str]

classmethod from_arrow(*name*, *source*, *destination*, *schema*, ***, *include_type_label=True*, *coercion='widen'*)

Import an edge contract from a `pyarrow.Schema` declaration.

Parameters

- **name** (*str*)
- **source** (NodeType | *str* | *Iterable[str]*)
- **destination** (NodeType | *str* | *Iterable[str]*)
- **schema** (*Any*)
- **include_type_label** (*bool*)
- **coercion** (*Literal['strict', 'widen']*)

Return type
EdgeType

name: str

properties: Mapping[str, NodeRef | EdgeRef | ScalarType | PathType | ListType]

source: FrozenSet[str]

to_arrow(***, *include_type_label=True*, *coercion='widen'*)

Export this edge contract as a `pyarrow.Schema`.

Parameters

- **include_type_label** (*bool*)
- **coercion** (*Literal['strict', 'widen']*)

Return type
Any

to_row_schema(***, *include_type_label=True*)

Export this edge contract as GFQL's Arrow-bridge row schema.

Parameters
include_type_label (*bool*)

Return type
RowSchema

property topology: *EdgeTopology*

class graphistry.schema.GraphSchema(*node_types=()*, *edge_types=()*, ***, *strict=True*, *node_id_column=None*, *edge_source_column=None*, *edge_destination_column=None*)

Bases: `object`

Experimental public graph schema contract for GFQL validation.

Parameters

- **node_types** (*Tuple[NodeType, ...]*)
- **edge_types** (*Tuple[EdgeType, ...]*)
- **strict** (*bool*)
- **node_id_column** (*str | None*)
- **edge_source_column** (*str | None*)
- **edge_destination_column** (*str | None*)

edge_arrow(***, *include_type_labels=True*, *coercion='widen'*)

Export the union of edge declarations as a `pyarrow.Schema`.

Parameters

- **include_type_labels** (*bool*)
- **coercion** (*Literal['strict', 'widen']*)

Return type

Any

property edge_columns: FrozenSet[str]

property edge_columns_by_type: Dict[str, Tuple[str, ...]]

edge_destination_column: str | None = None

edge_source_column: str | None = None

edge_types: Tuple[*EdgeType*, ...]

classmethod from_arrow(*declaration=None*, ***, *node_types=None*, *edge_types=None*, *strict=True*, *node_id_column=None*, *edge_source_column=None*, *edge_destination_column=None*, *coercion='widen'*)

Import graph schema declarations from Arrow schemas.

This is not inference: callers provide node/edge names and edge topology, either directly or via the payload emitted by `to_arrow()`.

Parameters

- **declaration** (*Mapping[str, Any] | None*)
- **node_types** (*Mapping[str, Any] | None*)
- **edge_types** (*Mapping[str, Any] | None*)
- **strict** (*bool*)
- **node_id_column** (*str | None*)
- **edge_source_column** (*str | None*)
- **edge_destination_column** (*str | None*)
- **coercion** (*Literal['strict', 'widen']*)

Return type

GraphSchema

`node_arrow`(*, *include_labels=True*, *coercion='widen'*)

Export the union of node declarations as a `pyarrow.Schema`.

Parameters

- `include_labels` (*bool*)
- `coercion` (*Literal['strict', 'widen']*)

Return type
Any

`property node_columns: FrozenSet[str]`

`property node_columns_by_label: Dict[str, Tuple[str, ...]]`

`node_id_column: str | None = None`

`node_types: Tuple[NodeType, ...]`

`strict: bool = True`

`to_arrow`(*, *include_labels=True*, *include_type_labels=True*, *coercion='widen'*)

Export this graph schema as Arrow declarations.

The payload is intentionally declaration-shaped rather than inferred: per-node and per-edge entries preserve public type names and topology, while `nodes` and `edges` expose table-level merged schemas for dataframe boundary validation.

Parameters

- `include_labels` (*bool*)
- `include_type_labels` (*bool*)
- `coercion` (*Literal['strict', 'widen']*)

Return type
Dict[str, *Any*]

`to_catalog`(*, *node_id_column=None*, *edge_source_column=None*, *edge_destination_column=None*, *strict=None*)

Adapt this public schema into the internal GFQL schema catalog.

Parameters

- `node_id_column` (*str | None*)
- `edge_source_column` (*str | None*)
- `edge_destination_column` (*str | None*)
- `strict` (*bool | None*)

Return type
GraphSchemaCatalog

`class graphistry.schema.NodeType`(*name*, *properties=None*, *labels=None*)

Bases: `object`

Experimental declarative node contract for GFQL schema validation.

`name` is the stable user-facing type name. `labels` defaults to `(name,)` and maps to GFQL's existing `label__<Label>` convention.

Parameters

- **name** (*str*)
- **properties** (*Mapping[str, NodeRef | EdgeRef | ScalarType | PathType | ListType]*)
- **labels** (*FrozenSet[str]*)

property columns: FrozenSet[str]

Columns admitted for this node type, including label columns.

classmethod from_arrow(*name*, *schema*, ***, *labels=None*, *include_labels=True*, *coercion='widen'*)

Import a node contract from a pyarrow.Schema declaration.

Parameters

- **name** (*str*)
- **schema** (*Any*)
- **labels** (*Iterable[str] | None*)
- **include_labels** (*bool*)
- **coercion** (*Literal['strict', 'widen']*)

Return type

NodeType

labels: FrozenSet[str]

name: str

properties: Mapping[str, NodeRef | EdgeRef | ScalarType | PathType | ListType]

to_arrow(***, *include_labels=True*, *coercion='widen'*)

Export this node contract as a pyarrow.Schema.

Parameters

- **include_labels** (*bool*)
- **coercion** (*Literal['strict', 'widen']*)

Return type

Any

to_row_schema(***, *include_labels=True*)

Export this node contract as GFQL's Arrow-bridge row schema.

Parameters

include_labels (*bool*)

Return type

RowSchema

10.10.4 Compute API Reference

10.10.4.1 ComputeMixin module

class graphistry.compute.ComputeMixin.**ComputeMixin**(*a, **kw)

Bases: *Plottable*

chain(*args, **kwargs)

Deprecated since version 2.XX.X: Use `gfql()` instead for a unified API that supports both chains and DAGs.

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
- **validate_schema** – Whether to validate the chain against the graph schema before executing
- **policy** – Optional policy dict for hooks
- **context** – Optional ExecutionContext for tracking execution state
- **start_nodes** – Optional node wavefront for the first traversal step

Returns
Plotter

Return type
Plotter

chain_remote(*args, **kwargs)

Deprecated since version 2.XX.X: Use `gfql_remote()` instead for a unified API that supports both chains and DAGs.

Remotely run GFQL chain query on a remote dataset.

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

param chain
GFQL query as a Python object, serialized GFQL JSON, or Cypher string

type chain
Union[Chain, List[ASTObject], Dict[str, JSONVal], ASTLet, str]

param api_token
Optional JWT token. If not provided, refreshes JWT and uses that.

type api_token
Optional[str]

param dataset_id
Optional dataset_id. If not provided, will fallback to self._dataset_id. If not provided, will upload current data, store that dataset_id, and run GFQL against that.

type dataset_id
Optional[str]

param output_type
Whether to return nodes and edges ("all", default), Plottable with just nodes ("nodes"), or Plottable with just edges ("edges"). For just a dataframe of the resultant graph shape (output_type="shape"), use instead chain_remote_shape().

type output_type
OutputType

param format
What format to fetch results. We recommend a columnar format such as parquet, which it defaults to when output_type is not shape.

type format
Optional[FormatType]

param df_export_args
When server parses data, any additional parameters to pass in.

type df_export_args
Optional[Dict, str, Any]]

param node_col_subset
When server returns nodes, what property subset to return. Defaults to all.

type node_col_subset
Optional[List[str]]

param edge_col_subset
When server returns edges, what property subset to return. Defaults to all.

type edge_col_subset
Optional[List[str]]

param engine
Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.

type engine
EngineAbstractType

param validate
Whether to locally test code, and if uploading data, the data. Default true.

type validate
bool

param persist
Whether to persist dataset on server and return dataset_id for immediate URL generation. Default false.

type persist
bool

Example: Explicitly upload graph and return subgraph where nodes have at least one edge

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst').upload()
assert g1._dataset_id, "Graph should have uploaded"

g2 = g1.chain_remote([n(), e(), n()])
print(f'dataset id: {g2._dataset_id}, # nodes: {len(g2._nodes)}')
```

Example: Return subgraph where nodes have at least one edge, with implicit upload

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst')
g2 = g1.chain_remote([n(), e(), n()])
print(f'dataset id: {g2._dataset_id}, # nodes: {len(g2._nodes)}')
```

Example: Return subgraph where nodes have at least one edge, with implicit upload, and force GPU mode

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst')
g2 = g1.chain_remote([n(), e(), n()], engine='cudf')
print(f'dataset id: {g2._dataset_id}, # nodes: {len(g2._nodes)}')
```

Return type
Plottable

`chain_remote_shape`(**args*, ***kwargs*)

Deprecated since version 2.XX.X: Use `gfql_remote_shape()` instead for a unified API that supports both chains and DAGs.

Like chain_remote(), except instead of returning a Plottable, returns a pd.DataFrame of the shape of the resulting graph.

Useful as a fast success indicator that avoids the need to return a full graph when a match finds hits, return just the metadata.

Example: Upload graph and compute number of nodes with at least one edge

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst').upload()
assert g1._dataset_id, "Graph should have uploaded"

shape_df = g1.chain_remote_shape([n(), e(), n()])
print(shape_df)
```

Example: Compute number of nodes with at least one edge, with implicit upload, and force GPU mode

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst')

shape_df = g1.chain_remote_shape([n(), e(), n()], engine='cudf')
print(shape_df)
```

Return type
DataFrame

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
```

(continues on next page)

(continued from previous page)

```
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

gfql(**args*, ***kwargs*)

Execute a GFQL query - either a chain or a DAG

Unified entrypoint that automatically detects query type and dispatches to the appropriate execution engine.

Parameters

- **query** – GFQL query - ASTObject, List[ASTObject], Chain, ASTLet, dict, or supported query string

- **engine** – Execution engine (auto, pandas, cudf)
- **output** – For DAGs, name of binding to return (default: last executed)
- **policy** – Optional policy hooks for external control (preload, postload, precall, postcall phases)
- **where** – Optional same-path constraints for list/Chain queries
- **language** – Optional string-query language selector. Defaults to `"cypher"` when `query` is a string.
- **params** – Optional parameter dictionary for string-query compilation
- **validate** – When `True`, run local preflight validation before execution via `g.gfql_validate(...)`.
- **shortest_path_backend** – Backend for shortestPath execution: `"auto"` (default), `"igraph"` (require igraph, raise if missing), `"cugraph"` (require cugraph, raise if missing), or `"bfs"` (always use DataFrame BFS). `"auto"` tries cugraph on CUDF engine, igraph on pandas, falls back to BFS silently.

Returns
Resulting Plottable

Return type
Plottable

gfql_remote(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*, *params=None*, *output=None*)

Run GFQL query remotely.

This is the remote execution version of `gfql()`. It supports chains, Let/DAG patterns, and Cypher strings.

The query is compiled locally and sent to the server as wire-protocol JSON. A `gfql_query` field carries the full typed envelope (including WHERE clauses); `gfql_operations` carries a flat array for backward compatibility with older servers.

Parameters

- **chain** (*Chain | List[ASTObject] | ASTLet | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]] | str*) – GFQL query — Chain, List[ASTObject], ASTLet, Dict, or Cypher string (compiled locally before sending).
- **params** (*Dict[str, Any] | None*) – Optional parameter dict for Cypher string queries (e.g., `params={"val": 10}` for `$val` references).
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)

- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)
- **output** (*str | None*)

Return type
Plottable

Example:

```
# Chain (existing)
g.gfql_remote([n(), e(), n()])

# Cypher string with params
g.gfql_remote(
    "MATCH (n) WHERE n.score > $cutoff RETURN n",
    params={"cutoff": 10},
)

# GRAPH constructor
g.gfql_remote("GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 5 }")
```

See `chain_remote()` for additional parameter documentation.

gfql_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

Get shape metadata for remote GFQL query execution.

This is the remote shape version of `gfql()`. Returns metadata about the resulting graph without downloading the full data.

See `chain_remote_shape()` for detailed documentation (chain_remote_shape is deprecated).

Parameters

- **chain** (*Chain | List[ASTObject] | ASTLet | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]] | str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

gfql_validate(**args*, ***kwargs*)

Validate a GFQL/Cypher query without executing it.

Raises structured GFQL exceptions on validation failures and never dispatches query execution operators.

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). Shorthand for max_hops. min_hops/max_hops: inclusive traversal bounds; defaults preserve legacy behavior (min=1 unless max=0; max defaults to hops). output_min_hops/output_max_hops: optional output slice applied after traversal; defaults keep all traversed hops up to max_hops. Useful for showing a subrange (e.g., min/max = 2..4 but display only hops 3..4). label_node_hops/label_edge_hops: optional column names for hop numbers (omit or None to skip). Nodes record the first retained hop step they are reached (1 = first expansion); when min_hops prunes shorter branches, labels reflect the shortest retained path. Edges record the hop step that traversed them. label_seeds: when True and labeling, also write hop 0 for seed nodes in the node label column. 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 include_zero_hop_seed: internal Cypher opt-in for exact zero-hop path semantics Note: chain() reverse passes set return_as_wave_front=True and use target_wave_front to constrain reachability. 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()

python_remote_g(**args*, ***kwargs*)

Remotely run Python code on a remote dataset that returns a Plottable

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

Parameters

- **code** (*Union[str, Callable[..., object]]*) – Python code that includes a top-level function *def task(g: Plottable) -> Union[str, Dict]*.
- **api_token** (*Optional[str]*) – Optional JWT token. If not provided, refreshes JWT and uses that.
- **dataset_id** (*Optional[str]*) – Optional dataset_id. If not provided, will fallback to self._dataset_id. If not defined, will upload current data, store that dataset_id, and run code against that.
- **format** (*Optional[FormatType]*) – What format to fetch results. Defaults to 'parquet'.
- **output_type** (*Optional[OutputTypeGraph]*) – What shape of output to fetch. Defaults to 'all'. Options include 'nodes', 'edges', 'all' (both). For other variants, see python_remote_shape and python_remote_json.
- **engine** (*EngineAbstractType*) – Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.
- **run_label** (*Optional[str]*) – Optional label for the run for serverside job tracking.
- **validate** (*bool*) – Whether to locally test code, and if uploading data, the data. Default true.

Return type
Any

Example: Upload data and count the results

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .edges(es, source='src', destination='dst')
    .upload()
assert g1._dataset_id is not None, "Successfully uploaded"
g2 = g1.python_remote_g(
    code='''
        from typing import Any, Dict
        from graphistry import Plottable
```

(continues on next page)

(continued from previous page)

```
        def task(g: Plottable) -> Dict[str, Any]:
            return g
    ''',
    engine='cudf')
num_edges = len(g2._edges)
print(f'num_edges: {num_edges}')
```

python_remote_json(**args*, ***kwargs*)

Remotely run Python code on a remote dataset that returns json

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

Parameters

- **code** (*Union[str, Callable[..., object]]*) – Python code that includes a top-level function *def task(g: Plottable) -> Union[str, Dict]*.
- **api_token** (*Optional[str]*) – Optional JWT token. If not provided, refreshes JWT and uses that.
- **dataset_id** (*Optional[str]*) – Optional dataset_id. If not provided, will fallback to self._dataset_id. If not defined, will upload current data, store that dataset_id, and run code against that.
- **engine** (*EngineAbstractType*) – Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.
- **run_label** (*Optional[str]*) – Optional label for the run for serverside job tracking.
- **validate** (*bool*) – Whether to locally test code, and if uploading data, the data. Default true.

Return type

Any

Example: Upload data and count the results

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .edges(es, source='src', destination='dst')
    .upload()
assert g1._dataset_id is not None, "Successfully uploaded"
obj = g1.python_remote_json(
    code='''
        from typing import Any, Dict
        from graphistry import Plottable

        def task(g: Plottable) -> Dict[str, Any]:
            return {'num_edges': len(g._edges)}
    ''',
```

(continues on next page)

(continued from previous page)

```
    engine='cudf')
num_edges = obj['num_edges']
print(f'num_edges: {num_edges}')
```

python_remote_table(**args*, ***kwargs*)

Remotely run Python code on a remote dataset that returns a table

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

Parameters

- **code** (*Union[str, Callable[..., object]]*) – Python code that includes a top-level function *def task(g: Plottable) -> Union[str, Dict]*.
- **api_token** (*Optional[str]*) – Optional JWT token. If not provided, refreshes JWT and uses that.
- **dataset_id** (*Optional[str]*) – Optional dataset_id. If not provided, will fallback to self._dataset_id. If not defined, will upload current data, store that dataset_id, and run code against that.
- **format** (*Optional[FormatType]*) – What format to fetch results. Defaults to 'parquet'.
- **output_type** (*Optional[OutputTypeGraph]*) – What shape of output to fetch. Defaults to 'table'. Options include 'table', 'nodes', and 'edges'.
- **engine** (*EngineAbstractType*) – Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.
- **run_label** (*Optional[str]*) – Optional label for the run for serverside job tracking.
- **validate** (*bool*) – Whether to locally test code, and if uploading data, the data. Default true.

Return type

Any

Example: Upload data and count the results

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .edges(es, source='src', destination='dst')
    .upload()
assert g1._dataset_id is not None, "Successfully uploaded"
edges_df = g1.python_remote_table(
    code='''
        from typing import Any, Dict
        from graphistry import Plottable

        def task(g: Plottable) -> Dict[str, Any]:
            return g._edges
```

(continues on next page)

(continued from previous page)

```
    ''',
    engine='cudf')
num_edges = len(edges_df)
print(f'num_edges: {num_edges}')
```

`to_cudf()`

Convert to GPU mode by converting any defined nodes and edges to cudf dataframes

When nodes or edges are already cudf dataframes, they are left as is

Parameters
g (Plottable) – Graphistry object

Returns
Graphistry object

Return type
Plottable

`to_pandas()`

Convert nodes and edges to pandas DataFrames.

Supports all input types: cuDF, Arrow, Polars, Spark, dask, and pandas (identity).

Return type
Plottable

10.10.4.2 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)

Generates unique internal column names to avoid conflicts with user data. Stores the generated names as attributes on the graph object: - g._collapse_node_col - g._collapse_src_col - g._collapse_dst_col

Parameters
g (Plottable) – graphistry instance

Returns
graphistry instance

`graphistry.compute.collapse.check_has_set(ndf, parent, child, collapse_node_col)`

Parameters
collapse_node_col (*str*)

`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*, *collapse_node_col*)

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
- **collapse_node_col** (*str*) – the collapse node column name

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*, *collapse_node_col*)

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*
- **collapse_node_col** (*str*) – the collapse node column name

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*, *collapse_node_col*)

checks if node is in collapse_node in COLLAPSE column of nodes DataFrame

Parameters

- **ndf** (*DataFrame*) – nodes DataFrame
- **node** (*str / int*) – node to find
- **collapse_node_col** (*str*) – the collapse node column name

Returns
bool

Return type
bool

graphistry.compute.collapse.**melt**(*ndf*, *node*, *collapse_node_col*)

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 :param collapse_node_col: the collapse node column name :returns new_parent_name of super node

Parameters

- **ndf** (*DataFrame*)
- **node** (*str / int*)
- **collapse_node_col** (*str*)

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

10.10.4.3 Conditional

class graphistry.compute.conditional.**ConditionalMixin**(**a*, ***kw*)

Bases: *Plottable*

DGL_graph: Any | None

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

Parameters

- **fg** (*Dict[str, Any] | None*)
- **bg** (*Dict[str, Any] | None*)
- **page** (*Dict[str, Any] | None*)
- **logo** (*Dict[str, Any] | None*)

Return type
Plottable

`base_url_client`(*v=None*)

Parameters
`v` (*str | None*)

Return type
str

`base_url_server`(*v=None*)

Parameters
`v` (*str | None*)

Return type
str

`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*, *point_longitude=None*, *point_latitude=None*, *dataset_id=None*, *url=None*, *nodes_file_id=None*, *edges_file_id=None*, *schema=None*)

Parameters

- `source` (*str | None*)
- `destination` (*str | None*)
- `node` (*str | None*)
- `edge` (*str | None*)
- `edge_title` (*str | None*)
- `edge_label` (*str | None*)
- `edge_color` (*str | None*)
- `edge_weight` (*str | None*)
- `edge_size` (*str | None*)
- `edge_opacity` (*str | None*)
- `edge_icon` (*str | None*)
- `edge_source_color` (*str | None*)
- `edge_destination_color` (*str | None*)
- `point_title` (*str | None*)
- `point_label` (*str | None*)
- `point_color` (*str | None*)
- `point_weight` (*str | None*)
- `point_size` (*str | None*)
- `point_opacity` (*str | None*)
- `point_icon` (*str | None*)
- `point_x` (*str | None*)
- `point_y` (*str | None*)

- **point_longitude** (*str | None*)
- **point_latitude** (*str | None*)
- **dataset_id** (*str | None*)
- **url** (*str | None*)
- **nodes_file_id** (*str | None*)
- **edges_file_id** (*str | None*)
- **schema** (*Any | None*)

Return type
Plottable

chain(*ops*)

ops is Union[List[ASTObject], Chain]

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

Return type
Plottable

chain_remote(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine=None*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*Literal['pandas', 'cudf'] | None*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
Plottable

chain_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine=None*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*Literal['pandas', 'cudf'] | None*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

client_protocol_hostname(*v=None*)

Parameters
v (*str | None*)

Return type
str

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

collections(*collections=None*, *show_collections=None*, *collections_global_node_color=None*, *collections_global_edge_color=None*, *validate='autofix'*, *warn=True*)

Parameters

- **collections** (*str | CollectionSet | CollectionIntersection | List[CollectionSet | CollectionIntersection] | None*)
- **show_collections** (*bool | None*)
- **collections_global_node_color** (*str | None*)
- **collections_global_edge_color** (*str | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)

- **warn** (*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*)

Return type
Plottable

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

Parameters

- **alg** (*str*)
- **out_col** (*str | None*)
- **directed** (*bool | None*)
- **use_vids** (*bool*)
- **params** (*dict*)
- **stringify_rich_types** (*bool*)

Return type
Plottable

compute_networkx(*alg*, *out_col=None*, *params=None*, *directed=True*, *G=None*)

Parameters

- **alg** (*str*)
- **out_col** (*str | None*)
- **params** (*Dict[str, Any] | None*)
- **directed** (*bool*)
- **G** (*Any | None*)

Return type
Plottable

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

`copy`()

Return type

Plottable

`description`(*description*)

Parameters

description (*str*)

Return type

Plottable

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

Return type

Plottable

embed(*relation*, *proto='DistMult'*, *embedding_dim=32*, *use_feat=False*, *X=None*, *epochs=2*, *batch_size=32*, *train_split=0.8*, *sample_size=1000*, *num_steps=50*, *lr=0.01*, *inplace=False*, *device='cpu'*, *evaluate=True*, **args*, ***kwargs*)

Parameters

- **relation** (*str*)
- **proto** (*str | Callable[[Any, Any], Any] | None*)
- **embedding_dim** (*int*)
- **use_feat** (*bool*)
- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **epochs** (*int*)
- **batch_size** (*int*)
- **train_split** (*float | int*)
- **sample_size** (*int*)
- **num_steps** (*int*)
- **lr** (*float*)
- **inplace** (*bool | None*)
- **device** (*str | None*)
- **evaluate** (*bool*)

Return type
Plottable

encode_axis(*rows=[]*)

Parameters
rows (*List[Dict]*)

Return type
Plottable

encode_edge_badge(*column*, *position='TopRight'*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *color=Ellipsis*, *bg=Ellipsis*, *fg=Ellipsis*, *for_current=False*, *for_default=True*, *as_text=Ellipsis*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)

- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_edge_color(*column*, *palette=Ellipsis*, *as_categorical=Ellipsis*, *as_continuous=Ellipsis*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **palette** (*List[str] | None*)
- **as_categorical** (*bool | None*)
- **as_continuous** (*bool | None*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_edge_icon(*column*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*str | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)
- **as_text** (*bool*)
- **blend_mode** (*str | None*)

- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_edge_label(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_opacity(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_size(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_title(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_weight(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_point_badge(*column*, *position='TopRight'*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *color=Ellipsis*, *bg=Ellipsis*, *fg=Ellipsis*, *for_current=False*, *for_default=True*, *as_text=Ellipsis*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)
- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_point_color(*column*, *palette=Ellipsis*, *as_categorical=Ellipsis*, *as_continuous=Ellipsis*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **palette** (*List[str] | None*)
- **as_categorical** (*bool | None*)
- **as_continuous** (*bool | None*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_point_icon(*column*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*str | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)
- **as_text** (*bool*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_point_label(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_point_opacity(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_point_size(*column*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)

- **for_current** (*bool*)

Return type
Plottable

encode_point_title(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

fa2_layout(*fa2_params=None*, *circle_layout_params=None*, *singleton_layout=None*, *partition_key=None*, *engine='auto'*, *allow_cpu_fallback=False*)

Parameters

- **fa2_params** (*Dict[str, Any] | None*)
- **circle_layout_params** (*Dict[str, Any] | None*)
- **singleton_layout** (*Callable[[Plottable, Tuple[float, float, float, float]] | Any], Plottable] | None*)
- **partition_key** (*str | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **allow_cpu_fallback** (*bool*)

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

filter_weighted_edges(*scale=1.0*, *index_to_nodes_dict=None*, *inplace=False*, *kind='nodes'*)

Parameters

- **scale** (*float*)
- **index_to_nodes_dict** (*Dict | None*)
- **inplace** (*bool*)
- **kind** (*Literal['nodes', 'edges']*)

Return type
Plottable | None

`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*)

Return type
Plottable

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

Parameters

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

Return type
Plottable

`from_networkx`(*G*)

Parameters
G (*Any*)

Return type
Plottable

`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

`gfql_remote`(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- `self` (Plottable)
- `chain` (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- `api_token` (*str | None*)
- `dataset_id` (*str | None*)
- `output_type` (*Literal['all', 'nodes', 'edges', 'shape']*)
- `format` (*Literal['json', 'csv', 'parquet'] | None*)
- `df_export_args` (*Dict[str, Any] | None*)
- `node_col_subset` (*List[str] | None*)
- `edge_col_subset` (*List[str] | None*)
- `engine` (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- `validate` (*bool*)
- `persist` (*bool*)

Return type
Plottable

gfql_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

chain is Union[List[ASTObject], Chain]

Parameters

- **self** (Plottable)
- **chain** (*Any | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]]*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

graph(*ig*)

Parameters
ig (*Any*)

Return type
Plottable

hop(*nodes*, *hops=1*, ***, *min_hops=None*, *max_hops=None*, *output_min_hops=None*, *output_max_hops=None*, *label_node_hops=None*, *label_edge_hops=None*, *label_seeds=False*, *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*, *include_zero_hop_seed=False*, *target_wave_front=None*, *engine='auto'*)

Parameters

- **nodes** (*DataFrame | None*)
- **hops** (*int | None*)
- **min_hops** (*int | None*)
- **max_hops** (*int | None*)
- **output_min_hops** (*int | None*)
- **output_max_hops** (*int | None*)
- **label_node_hops** (*str | None*)
- **label_edge_hops** (*str | None*)

- **label_seeds** (*bool*)
- **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*)
- **include_zero_hop_seed** (*bool*)
- **target_wave_front** (*DataFrame | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

Return type
Plottable

hypergraph(*raw_events=None*, ***, *entity_types=None*, *opts={}*, *drop_na=True*, *drop_edge_attrs=False*, *verbose=True*, *direct=False*, *engine='auto'*, *npartitions=None*, *chunksize=None*, *from_edges=False*, *return_as='graph'*)

Parameters

- **raw_events** (*Any | None*)
- **entity_types** (*List[str] | None*)
- **opts** (*dict*)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)
- **verbose** (*bool*)
- **direct** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **from_edges** (*bool*)
- **return_as** (*Literal['graph', 'all', 'entities', 'events', 'edges', 'nodes']*)

Return type
Plottable | *HypergraphResult* | *Any*

`igraph2pandas`(*ig*)

Parameters
`ig` (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

`infer_labels`()

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*)

Return type
Plottable

`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*)

Parameters

- `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*)
- **skip_styling** (*bool*)
- **render_to_disk** (*bool*)
- **path** (*str | None*)
- **format** (*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'] | None*)

Return type

Plottable

layout_igraph(*layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Parameters

- **layout** (*str*)
- **directed** (*bool* / *None*)
- **use_vids** (*bool*)
- **bind_position** (*bool*)
- **x_out_col** (*str*)
- **y_out_col** (*str*)
- **play** (*int* / *None*)
- **params** (*dict*)

Return type
Plottable

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*)

Return type
Plottable

`materialize_nodes`(*reuse=True*, *engine='auto'*)

Parameters

- **reuse** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

Return type
Plottable

`name`(*name*)

Parameters
name (*str*)

Return type
Plottable

`networkx2pandas`(*G*)

Parameters
G (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

`networkx_checkoverlap`(*g*)

Parameters
g (*Any*)

Return type
None

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

Parameters

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

Return type
Plottable

`pandas2igraph`(*edges*, *directed=True*)

Parameters

- **edges** (*DataFrame*)
- **directed** (*bool*)

Return type
Any

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

Parameters

- **graph_transform** (*Callable*)
- **args** (*Any*)
- **kwargs** (*Any*)

Return type
Plottable

plot(*graph=None*, *nodes=None*, *name=None*, *description=None*, *render='auto'*, *skip_upload=False*, *as_files=False*, *memoize=True*, *erase_files_on_fail=True*, *extra_html=''*, *override_html_style=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Parameters

- **graph** (*Any | None*)
- **nodes** (*Any | None*)
- **name** (*str | None*)
- **description** (*str | None*)
- **render** (*bool | Literal['auto'] | ~typing.Literal['g', 'url', 'ipython', 'databricks', 'browser'] | None*)
- **skip_upload** (*bool*)
- **as_files** (*bool*)
- **memoize** (*bool*)
- **erase_files_on_fail** (*bool*)
- **extra_html** (*str*)
- **override_html_style** (*str | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)

Return type
Any

privacy(*mode=None*, *notify=None*, *invited_users=None*, *message=None*, *mode_action=None*)

Parameters

- **mode** (*Literal['private', 'organization', 'public'] | None*)
- **notify** (*bool | None*)
- **invited_users** (*List[str] | None*)
- **message** (*str | None*)
- **mode_action** (*Literal['10', '20'] | None*)

Return type
Plottable

`protocol`(*v=None*)

Parameters
`v` (*str | None*)

Return type
str

`prune_self_edges`()

Return type
Plottable

`python_remote_g`(*code*, *api_token=None*, *dataset_id=None*, *format='parquet'*, *output_type='all'*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- `self` (Plottable)
- `code` (*str*)
- `api_token` (*str | None*)
- `dataset_id` (*str | None*)
- `format` (*Literal['json', 'csv', 'parquet'] | None*)
- `output_type` (*Literal['all', 'nodes', 'edges', 'shape'] | ~typing.Literal['table', 'shape'] | ~typing.Literal['json'] | None*)
- `engine` (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- `run_label` (*str | None*)
- `validate` (*bool*)

Return type
Plottable

`python_remote_json`(*code*, *api_token=None*, *dataset_id=None*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- `self` (Plottable)
- `code` (*str*)
- `api_token` (*str | None*)
- `dataset_id` (*str | None*)
- `engine` (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- `run_label` (*str | None*)
- `validate` (*bool*)

Return type
Any

python_remote_table(*code*, *api_token=None*, *dataset_id=None*, *format='parquet'*, *output_type='table'*, *engine='auto'*, *run_label=None*, *validate=True*)

Parameters

- **self** (Plottable)
- **code** (*str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **output_type** (*Literal['table', 'shape'] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **run_label** (*str | None*)
- **validate** (*bool*)

Return type
DataFrame

reset_caches()

Return type
None

scene_settings(*menu=None*, *info=None*, *show_arrows=None*, *point_size=None*, *edge_curvature=None*, *edge_opacity=None*, *point_opacity=None*)

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*)

Return type
Plottable

search(*query*, *cols=None*, *thresh=5000*, *fuzzy=True*, *top_n=10*)

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*)

Parameters

- **query** (*str*)
- **scale** (*float*)
- **top_n** (*int*)
- **thresh** (*float*)
- **broader** (*bool*)
- **inplace** (*bool*)

Return type
Plottable

server(*v=None*)

Parameters
v (*str | None*)

Return type
str

session: ClientSession

settings(*height=None*, *url_params=None*, *render=None*, *validate='autofix'*, *warn=True*)

Parameters

- **height** (*int | None*)
- **url_params** (*Dict[str, None | str | int | float | bool | List[SettingsValue] | Dict[str, SettingsValue]] | None*)
- **render** (*bool | Literal['auto'] | ~typing.Literal['g', 'url', 'ipython', 'databricks', 'browser'] | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Plottable

style(*fg=None*, *bg=None*, *page=None*, *logo=None*)

Parameters

- **fg** (*Dict[str, Any] | None*)
- **bg** (*Dict[str, Any] | None*)
- **page** (*Dict[str, Any] | None*)
- **logo** (*Dict[str, Any] | None*)

Return type
Plottable

to_arrow(*table=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*, *schema_table='edges'*)

Parameters

- **table** (*Any | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)
- **schema_table** (*str*)

Return type
Any | None

`to_cudf()`

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, include_nodes=True, node_attributes=None, edge_attributes=None, use_vids=False)`

Parameters

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

Return type
Any

`to_pandas()`

Return type
Plottable

`transform(df, y=None, kind='nodes', min_dist='auto', n_neighbors=7, merge_policy=False, sample=None, *, return_graph=True, scaled=True, verbose=False)`

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*)

Return type
Tuple[*DataFrame*, *DataFrame*] | Plottable

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={}*)

Parameters

- **df** (*DataFrame*)
- **y** (*DataFrame | None*)
- **kind** (*Literal['nodes', 'edges']*)
- **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*)

Parameters

- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **y** (*DataFrame | np.ndarray | List[str] | None*)
- **kind** (*Literal['nodes', 'edges']*)
- **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['auto', 'cuml', 'umap_learn']*)
- **feature_engine** (*str*)
- **inplace** (*bool*)
- **memoize** (*bool*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)
- **featurize_kwargs** (*Any*)

Return type
Plottable | None

umap_fit(*X*, *y=None*, *umap_fit_kwargs={}*)

Parameters

- **X** (*DataFrame*)
- **y** (*DataFrame | None*)
- **umap_fit_kwargs** (*Dict[str, Any]*)

Return type
Plottable

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

- **res** (Plottable)
- **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['auto', 'cuml', 'umap_learn']*)
- **suffix** (*str*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

Return type
Plottable

upload(*memoize=True*, *erase_files_on_fail=True*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Parameters

- **memoize** (*bool*)
- **erase_files_on_fail** (*bool*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)

Return type
Plottable

property url: str | None

validate_arrow_schema(*table='edges'*, ***, *validate='strict'*, *warn=True*)

Parameters

- **table** (*str*)
- **validate** (*Literal['strict', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Any | None

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*)

10.10.4.4 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=None*, *engine=EngineAbstract.AUTO*)

filter edges to those that match all values in filter_dict

Parameters

- **self** (Plottable)
- **filter_dict** (*dict | None*)
- **engine** (*EngineAbstract | str*)

Return type
Plottable

graphistry.compute.filter_by_dict.**filter_nodes_by_dict**(*self*, *filter_dict=None*, *engine=EngineAbstract.AUTO*)

filter nodes to those that match all values in filter_dict

Parameters

- **self** (Plottable)
- **filter_dict** (*dict | None*)
- **engine** (*EngineAbstract | str*)

Return type
Plottable

graphistry.compute.filter_by_dict.**resolve_filter_column**(*df*, *col*, *val*)

Parameters

- **df** (*Any*)
- **col** (*str*)
- **val** (*Any*)

Return type
Tuple[str, *Any*]

10.10.5 Hypergraphs

Hypergraphs are graphs where edges may connect more than two nodes, such as an event involving multiple entities.

Graphistry encodes hypergraphs as regular graphs of two forms. One is a bipartite graph between hypernodes and regular nodes connected by hyperedges. The other is regular nodes connected by hyperedges. In both cases, each hyperedge is encoded by multiple regular src/dst edges.

10.10.5.1 Hypergraph

graphistry.PlotterBase.PlotterBase.**hypergraph = <function PlotterBase.hypergraph>**

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*)
- **from_edges** (*bool*)
- **return_as** (*Literal['graph', 'all', 'entities', 'events', 'edges', 'nodes']*)

Return type
Plottable | *HypergraphResult* | *Any*

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

- **raw_events** (*Any | None*)
- **entity_types** (*List[str] | None*)
- **opts** (*dict*)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)

- **verbose** (*bool*)
- **direct** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **from_edges** (*bool*)
- **return_as** (*Literal['graph', 'all', 'entities', 'events', 'edges', 'nodes']*)

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']})
```

(continues on next page)

(continued from previous page)

```
h = graphistry.hypergraph(users_gdf, engine='cudf')
g = h['graph'].plot()
```

hypergraph

Primary alias for function *graphistry.hyper_dask.hypergraph()*.

class graphistry.hyper_dask.**HyperBindings**(*TITLE='nodeTitle'*, *DELIM='::'*, *NODEID='nodeID'*, *ATTRIBID='attribID'*, *EVENTID='EventID'*, *EVENTTYPE='event'*, *SOURCE='src'*, *DESTINATION='dst'*, *CATEGORY='category'*, *NODETYPE='type'*, *EDGETYPE='edgeType'*, *NULLVAL='null'*, *SKIP=None*, *CATEGORIES={}*, *EDGES=None*)

Bases: `object`

Parameters

- **TITLE** (*str*)
- **DELIM** (*str*)
- **NODEID** (*str*)
- **ATTRIBID** (*str*)
- **EVENTID** (*str*)
- **EVENTTYPE** (*str*)
- **SOURCE** (*str*)
- **DESTINATION** (*str*)
- **CATEGORY** (*str*)
- **NODETYPE** (*str*)
- **EDGETYPE** (*str*)
- **NULLVAL** (*str*)
- **SKIP** (*List[str] | None*)
- **CATEGORIES** (*Dict[str, List[str]]*)
- **EDGES** (*Dict[str, List[str]] | None*)

class graphistry.hyper_dask.**Hypergraph**(*g*, *defs*, *entities*, *event_entities*, *edges*, *source*, *destination*, *engine=Engine.PANDAS*, *debug=False*)

Bases: `object`

Parameters

- **entities** (*Any*)
- **event_entities** (*Any*)
- **edges** (*Any*)
- **source** (*str*)
- **destination** (*str*)
- **engine** (*Engine*)

- **debug** (*bool*)

graphistry.hyper_dask.**clean_events**(*events*, *defs*, *engine*, *npartitions=None*, *chunksize=None*, *dropna=False*, *debug=False*)

Copy with reset index and in the target engine format

Parameters

- **events** (*Any*)
- **defs** (HyperBindings)
- **engine** (*Engine*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **dropna** (*bool*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**coerce_col_safe**(*s*, *to_dtype*)

graphistry.hyper_dask.**col2cat**(*cat_lookup*, *col*)

Parameters

- **cat_lookup** (*Dict[str, str]*)
- **col** (*str*)

graphistry.hyper_dask.**concat**(*dfs*, *engine*, *debug=False*)

Parameters

- **dfs** (*List[Any]*)
- **engine** (*Engine*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**df_coercion**(*df*, *engine*, *npartitions=None*, *chunksize=None*, *debug=False*)

Go from df to engine of choice

Supported coercions:
pd <- pd cudf <- pd, cudf ddf <- pd, ddf dgdf <- pd, cudf, dgdf

Parameters

- **df** (*Any*)
- **engine** (*Engine*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**direct_edgelist_shape**(*entity_types*, *defs*)

Edges take format {src_col: [dest_col1, dest_col2], ...} If None, create connect all to all, leaving up to algorithm in which direction

Parameters

- **entity_types** (*List[str]*)
- **defs** (HyperBindings)

Return type
Dict[str, *List*[str]]

graphistry.hyper_dask.**format_direct_edges**(*engine*, *events*, *entity_types*, *defs*, *edge_shape*, *drop_na*, *drop_edge_attrs*, *debug=False*)

Parameters

- **engine** (*Engine*)
- **events** (*Any*)
- **defs** (HyperBindings)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**format_entities**(*events*, *entity_types*, *defs*, *direct*, *drop_na*, *engine*, *npartitions*, *chunksize*, *debug=False*)

Parameters

- **events** (*Any*)
- **entity_types** (*List[str]*)
- **defs** (HyperBindings)
- **direct** (*bool*)
- **drop_na** (*bool*)
- **engine** (*Engine*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**format_entities_from_col**(*defs*, *cat_lookup*, *drop_na*, *engine*, *col_name*, *df_with_col*, *meta*, *debug*)

For unique v in column col, create [{col: str(v), title: str(v), nodetype: col, nodeid: <*cat*><*delim*><*v*>}]

- respect drop_na
- respect colname overrides
- receive+return pd.DataFrame / cudf.DataFrame depending on engine

Parameters

- **defs** (HyperBindings)
- **cat_lookup** (*Dict[str, str]*)
- **drop_na** (*bool*)
- **engine** (*Engine*)
- **col_name** (*str*)
- **df_with_col** (*Any*)
- **meta** (*DataFrame*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**format_hyperedges**(*engine*, *events*, *entity_types*, *defs*, *drop_na*, *drop_edge_attrs*, *debug=False*)

Parameters

- **engine** (*Engine*)
- **events** (*Any*)
- **entity_types** (*List[str]*)
- **defs** (HyperBindings)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)
- **debug** (*bool*)

Return type
Any

graphistry.hyper_dask.**format_hypernodes**(*events*, *defs*, *drop_na*)

graphistry.hyper_dask.**get_df_cons**(*engine*)

Parameters
engine (*Engine*)

graphistry.hyper_dask.**get_series_cons**(*engine*, *dtype='int32'*)

Parameters
engine (*Engine*)

graphistry.hyper_dask.**hyperbinding**(*g*, *defs*, *entities*, *event_entities*, *edges*, *source*, *destination*)

graphistry.hyper_dask.**hypergraph**(*g*, *raw_events=None*, *entity_types=None*, *opts={}*, *drop_na=True*, *drop_edge_attrs=False*, *verbose=True*, *direct=False*, *engine='auto'*, *npartitions=None*, *chunksize=None*, *from_edges=False*, *return_as='graph'*, *debug=False*)

Internal details:

- IDs currently strings: *${namespace(col)}${delim}${str(val)}*
- debug: sprinkle persist() to catch bugs earlier

Parameters

- **raw_events** (*Any | None*)
- **entity_types** (*List[str] | None*)
- **opts** (*dict*)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)
- **verbose** (*bool*)
- **direct** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **from_edges** (*bool*)
- **return_as** (*Literal['graph', 'all', 'entities', 'events', 'edges', 'nodes']*)
- **debug** (*bool*)

Return type
Hypergraph

graphistry.hyper_dask.**make_reverse_lookup**(*categories*)

graphistry.hyper_dask.**mt_df**(*engine*)

Parameters
engine (*Engine*)

graphistry.hyper_dask.**mt_nodes**(*defs*, *events*, *entity_types*, *direct*, *engine*)

Parameters

- **defs** (HyperBindings)
- **events** (*Any*)
- **entity_types** (*List[str]*)
- **direct** (*bool*)
- **engine** (*Engine*)

Return type
DataFrame

graphistry.hyper_dask.**mt_series**(*engine*, *dtype='int32'*)

Create empty Series for given engine with proper dtype handling

Parameters
engine (*Engine*)

graphistry.hyper_dask.**screen_entities**(*events*, *entity_types*, *defs*)

List entity columns: Unskipped user-specified entities when provided, else unskipped cols

Parameters

- **events** (*Any*)
- **entity_types** (*List[str] | None*)
- **defs** (HyperBindings)

Return type
List[str]

graphistry.hyper_dask.**series_cons**(*engine*, *arr*, *dtype='int32'*, *npartitions=None*, *chunksize=None*)

Parameters

- **engine** (*Engine*)
- **arr** (*List*)

graphistry.hyper_dask.**shallow_copy**(*df*, *engine*, *debug=False*)

Parameters

- **df** (*Any*)
- **engine** (*Engine*)
- **debug** (*bool*)

Return type
Any

10.10.6 AI

graphistry['ai'] provides a set of utilities for AI and machine learning workflows on graphs, with optional GPU support

10.10.6.1 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*, *kind='nodes'*)

Bases: `object`

Parameters

- `df` (*DataFrame*)
- `y` (*DataFrame*)

`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.

Parameters

- **df** (*DataFrame*)
- **ydf** (*DataFrame | None*)

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**(**a*, ***kw*)

Bases: ComputeMixin

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')
```

DGL_graph: Any | None

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

Parameters

- **fg** (*Dict[str, Any] | None*)
- **bg** (*Dict[str, Any] | None*)
- **page** (*Dict[str, Any] | None*)
- **logo** (*Dict[str, Any] | None*)

Return type

Plottable

`base_url_client`(*v=None*)

Parameters
`v` (*str / None*)

Return type
str

`base_url_server`(*v=None*)

Parameters
`v` (*str / None*)

Return type
str

`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*, *point_longitude=None*, *point_latitude=None*, *dataset_id=None*, *url=None*, *nodes_file_id=None*, *edges_file_id=None*, *schema=None*)

Parameters

- `source` (*str / None*)
- `destination` (*str / None*)
- `node` (*str / None*)
- `edge` (*str / None*)
- `edge_title` (*str / None*)
- `edge_label` (*str / None*)
- `edge_color` (*str / None*)
- `edge_weight` (*str / None*)
- `edge_size` (*str / None*)
- `edge_opacity` (*str / None*)
- `edge_icon` (*str / None*)
- `edge_source_color` (*str / None*)
- `edge_destination_color` (*str / None*)
- `point_title` (*str / None*)
- `point_label` (*str / None*)
- `point_color` (*str / None*)
- `point_weight` (*str / None*)
- `point_size` (*str / None*)
- `point_opacity` (*str / None*)
- `point_icon` (*str / None*)
- `point_x` (*str / None*)
- `point_y` (*str / None*)

- **point_longitude** (*str | None*)
- **point_latitude** (*str | None*)
- **dataset_id** (*str | None*)
- **url** (*str | None*)
- **nodes_file_id** (*str | None*)
- **edges_file_id** (*str | None*)
- **schema** (*Any | None*)

Return type
Plottable

chain(**args*, ***kwargs*)

Deprecated since version 2.XX.X: Use *gfql()* instead for a unified API that supports both chains and DAGs.

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
- **validate_schema** – Whether to validate the chain against the graph schema before executing
- **policy** – Optional policy dict for hooks
- **context** – Optional ExecutionContext for tracking execution state
- **start_nodes** – Optional node wavefront for the first traversal step

Returns
Plotter

Return type
Plotter

chain_remote(**args*, ***kwargs*)

Deprecated since version 2.XX.X: Use *gfql_remote()* instead for a unified API that supports both chains and DAGs.

Remotely run GFQL chain query on a remote dataset.

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

param chain
GFQL query as a Python object, serialized GFQL JSON, or Cypher string

type chain
Union[Chain, List[ASTObject], Dict[str, JSONVal], ASTLet, str]

param api_token
Optional JWT token. If not provided, refreshes JWT and uses that.

type api_token
Optional[str]

param dataset_id
Optional dataset_id. If not provided, will fallback to self._dataset_id. If not provided, will upload current data, store that dataset_id, and run GFQL against that.

type dataset_id
Optional[str]

param output_type
Whether to return nodes and edges ("all", default), Plottable with just nodes ("nodes"), or Plottable with just edges ("edges"). For just a dataframe of the resultant graph shape (output_type="shape"), use instead chain_remote_shape().

type output_type
OutputType

param format
What format to fetch results. We recommend a columnar format such as parquet, which it defaults to when output_type is not shape.

type format
Optional[FormatType]

param df_export_args
When server parses data, any additional parameters to pass in.

type df_export_args
Optional[Dict, str, Any]]

param node_col_subset
When server returns nodes, what property subset to return. Defaults to all.

type node_col_subset
Optional[List[str]]

param edge_col_subset
When server returns edges, what property subset to return. Defaults to all.

type edge_col_subset
Optional[List[str]]

param engine
Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.

type engine
EngineAbstractType

param validate
Whether to locally test code, and if uploading data, the data. Default true.

type validate
bool

param persist
Whether to persist dataset on server and return dataset_id for immediate URL generation. Default false.

type persist
bool

Example: Explicitly upload graph and return subgraph where nodes have at least one edge

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst').upload()
assert g1._dataset_id, "Graph should have uploaded"

g2 = g1.chain_remote([n(), e(), n()])
print(f'dataset id: {g2._dataset_id}, # nodes: {len(g2._nodes)}')
```

Example: Return subgraph where nodes have at least one edge, with implicit upload

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst')
g2 = g1.chain_remote([n(), e(), n()])
print(f'dataset id: {g2._dataset_id}, # nodes: {len(g2._nodes)}')
```

Example: Return subgraph where nodes have at least one edge, with implicit upload, and force GPU mode

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst')
g2 = g1.chain_remote([n(), e(), n()], engine='cudf')
print(f'dataset id: {g2._dataset_id}, # nodes: {len(g2._nodes)}')
```

Return type
Plottable

`chain_remote_shape`(**args*, ***kwargs*)

Deprecated since version 2.XX.X: Use *`gfql_remote_shape()`* instead for a unified API that supports both chains and DAGs.

Like chain_remote(), except instead of returning a Plottable, returns a pd.DataFrame of the shape of the resulting graph.

Useful as a fast success indicator that avoids the need to return a full graph when a match finds hits, return just the metadata.

Example: Upload graph and compute number of nodes with at least one edge

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

(continues on next page)

(continued from previous page)

```
g1 = graphistry.edges(es, 'src', 'dst').upload()
assert g1._dataset_id, "Graph should have uploaded"

shape_df = g1.chain_remote_shape([n(), e(), n()])
print(shape_df)
```

Example: Compute number of nodes with at least one edge, with implicit upload, and force GPU mode

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry.edges(es, 'src', 'dst')

shape_df = g1.chain_remote_shape([n(), e(), n()], engine='cudf')
print(shape_df)
```

Return type
DataFrame

`client_protocol_hostname`(*v=None*)

Parameters
v (*str | None*)

Return type
str

`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

`collections`(*collections=None*, *show_collections=None*, *collections_global_node_color=None*, *collections_global_edge_color=None*, *validate='autofix'*, *warn=True*)

Parameters

- **collections** (*str | CollectionSet | CollectionIntersection | List[CollectionSet | CollectionIntersection] | None*)

- **show_collections** (*bool | None*)
- **collections_global_node_color** (*str | None*)
- **collections_global_edge_color** (*str | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*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*)

Return type
Plottable

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

Parameters

- **alg** (*str*)
- **out_col** (*str | None*)
- **directed** (*bool | None*)
- **use_vids** (*bool*)
- **params** (*dict*)
- **stringify_rich_types** (*bool*)

Return type
Plottable

compute_networkx(*alg*, *out_col=None*, *params=None*, *directed=True*, *G=None*)

Parameters

- **alg** (*str*)
- **out_col** (*str | None*)
- **params** (*Dict[str, Any] | None*)
- **directed** (*bool*)
- **G** (*Any | None*)

Return type
Plottable

`copy()`

Return type
Plottable

`description(description)`

Parameters
description (*str*)

Return type
Plottable

`drop_nodes(nodes)`

return g with any nodes/edges involving the node id series removed

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

Parameters

- **edges** (*Callable | Any*)
- **source** (*str | None*)
- **destination** (*str | None*)
- **edge** (*str | None*)
- **args** (*Any*)
- **kwargs** (*Any*)

Return type
Plottable

`embed(relation, proto='DistMult', embedding_dim=32, use_feat=False, X=None, epochs=2, batch_size=32, train_split=0.8, sample_size=1000, num_steps=50, lr=0.01, inplace=False, device='cpu', evaluate=True, *args, **kwargs)`

Parameters

- **relation** (*str*)
- **proto** (*str | Callable[[Any, Any, Any], Any] | None*)
- **embedding_dim** (*int*)
- **use_feat** (*bool*)
- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **epochs** (*int*)
- **batch_size** (*int*)
- **train_split** (*float | int*)
- **sample_size** (*int*)
- **num_steps** (*int*)
- **lr** (*float*)
- **inplace** (*bool | None*)
- **device** (*str | None*)
- **evaluate** (*bool*)

Return type
Plottable

`encode_axis`(*rows=[]*)

Parameters
`rows` (*List [Dict]*)

Return type
Plottable

`encode_edge_badge`(*column*, *position='TopRight'*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *color=Ellipsis*, *bg=Ellipsis*, *fg=Ellipsis*, *for_current=False*, *for_default=True*, *as_text=Ellipsis*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- `column` (*str*)
- `position` (*str*)
- `categorical_mapping` (*Dict [Any , Any] | None*)
- `continuous_binning` (*List [Any] | None*)
- `default_mapping` (*Any | None*)
- `comparator` (*Callable [[Any , Any] , int] | None*)
- `color` (*str | None*)
- `bg` (*str | None*)
- `fg` (*str | None*)
- `for_current` (*bool*)
- `for_default` (*bool*)
- `as_text` (*bool | None*)
- `blend_mode` (*str | None*)
- `style` (*Dict [str , Any] | None*)
- `border` (*Dict [str , Any] | None*)
- `shape` (*str | None*)

Return type
Plottable

`encode_edge_color`(*column*, *palette=Ellipsis*, *as_categorical=Ellipsis*, *as_continuous=Ellipsis*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- `column` (*str*)
- `palette` (*List [str] | None*)
- `as_categorical` (*bool | None*)
- `as_continuous` (*bool | None*)

- **categorical_mapping** (*Dict[Any, Any] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_edge_icon(*column*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*str | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)
- **as_text** (*bool*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_edge_label(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_opacity(**args*, ***kwargs*)

Parameters

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

Return type
Plottable

encode_edge_size(*args, **kwargs)

Parameters

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

Return type
Plottable

encode_edge_title(*args, **kwargs)

Parameters

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

Return type
Plottable

encode_edge_weight(*args, **kwargs)

Parameters

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

Return type
Plottable

encode_point_badge(*column*, *position='TopRight'*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *color=Ellipsis*, *bg=Ellipsis*, *fg=Ellipsis*, *for_current=False*, *for_default=True*, *as_text=Ellipsis*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **position** (*str*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*Any | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **color** (*str | None*)
- **bg** (*str | None*)
- **fg** (*str | None*)
- **for_current** (*bool*)
- **for_default** (*bool*)
- **as_text** (*bool | None*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)

- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_point_color(*column*, *palette=Ellipsis*, *as_categorical=Ellipsis*, *as_continuous=Ellipsis*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **palette** (*List[str] | None*)
- **as_categorical** (*bool | None*)
- **as_continuous** (*bool | None*)
- **categorical_mapping** (*Dict[Any, Any] | None*)
- **default_mapping** (*str | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_point_icon(*column*, *categorical_mapping=Ellipsis*, *continuous_binning=Ellipsis*, *default_mapping=Ellipsis*, *comparator=Ellipsis*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=Ellipsis*, *style=Ellipsis*, *border=Ellipsis*, *shape=Ellipsis*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, str] | None*)
- **continuous_binning** (*List[Any] | None*)
- **default_mapping** (*str | None*)
- **comparator** (*Callable[[Any, Any], int] | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)
- **as_text** (*bool*)
- **blend_mode** (*str | None*)
- **style** (*Dict[str, Any] | None*)
- **border** (*Dict[str, Any] | None*)
- **shape** (*str | None*)

Return type
Plottable

encode_point_label(*args, **kwargs)

Parameters

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

Return type
Plottable

encode_point_opacity(*args, **kwargs)

Parameters

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

Return type
Plottable

encode_point_size(*column*, *categorical_mapping=Ellipsis*, *default_mapping=Ellipsis*, *for_default=True*, *for_current=False*)

Parameters

- **column** (*str*)
- **categorical_mapping** (*Dict[Any, int | float] | None*)
- **default_mapping** (*int | float | None*)
- **for_default** (*bool*)
- **for_current** (*bool*)

Return type
Plottable

encode_point_title(*args, **kwargs)

Parameters

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

Return type
Plottable

fa2_layout(*fa2_params=None*, *circle_layout_params=None*, *singleton_layout=None*, *partition_key=None*, *engine='auto'*, *allow_cpu_fallback=False*)

Parameters

- **fa2_params** (*Dict[str, Any] | None*)
- **circle_layout_params** (*Dict[str, Any] | None*)
- **singleton_layout** (*Callable[[Plottable, Tuple[float, float, float, float] | Any], Plottable] | None*)
- **partition_key** (*str | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

- **allow_cpu_fallback** (*bool*)

Return type
Plottable

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*) – skrub 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', 'skrub', 'torch', 'dirty_cat', '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', 'skrub', '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', 'skrub', 'torch']*)
- **memoize** (*bool*)
- **verbose** (*bool*)

Return type
Tuple[*DataFrame*, *DataFrame*, object]

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

`filter_weighted_edges`(*scale=1.0*, *index_to_nodes_dict=None*, *inplace=False*, *kind='nodes'*)

Parameters

- **scale** (*float*)
- **index_to_nodes_dict** (*Dict | None*)
- **inplace** (*bool*)
- **kind** (*Literal['nodes', 'edges']*)

Return type
Plottable | None

`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*)

Return type
Plottable

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

Parameters

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

Return type
Plottable

`from_networkx`(*G*)

Parameters
G (*Any*)

Return type
Plottable

`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_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** (*Literal['nodes', 'edges']*)
- **target** (*bool*)

Return type
DataFrame

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

`gfql`(**args*, ***kwargs*)

Execute a GFQL query - either a chain or a DAG

Unified entrypoint that automatically detects query type and dispatches to the appropriate execution engine.

Parameters

- **query** – GFQL query - ASTObject, List[ASTObject], Chain, ASTLet, dict, or supported query string
- **engine** – Execution engine (auto, pandas, cudf)
- **output** – For DAGs, name of binding to return (default: last executed)
- **policy** – Optional policy hooks for external control (preload, postload, precall, postcall phases)
- **where** – Optional same-path constraints for list/Chain queries
- **language** – Optional string-query language selector. Defaults to `"cypher"` when `query` is a string.
- **params** – Optional parameter dictionary for string-query compilation
- **validate** – When `True`, run local preflight validation before execution via `g.gfql_validate(...)`.
- **shortest_path_backend** – Backend for shortestPath execution: `"auto"` (default), `"igraph"` (require igraph, raise if missing), `"cugraph"` (require cugraph, raise if missing), or `"bfs"` (always use DataFrame BFS). `"auto"` tries cugraph on CUDF engine, igraph on pandas, falls back to BFS silently.

Returns
Resulting Plottable

Return type
Plottable

`gfql_remote`(*chain*, *api_token=None*, *dataset_id=None*, *output_type='all'*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*, *params=None*, *output=None*)

Run GFQL query remotely.

This is the remote execution version of *gfql()*. It supports chains, Let/DAG patterns, and Cypher strings.

The query is compiled locally and sent to the server as wire-protocol JSON. A `gfql_query` field carries the full typed envelope (including WHERE clauses); `gfql_operations` carries a flat array for backward compatibility with older servers.

Parameters

- **chain** (*Chain | List[ASTObject] | ASTLet | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]] | str*) – GFQL query — Chain, List[ASTObject], ASTLet, Dict, or Cypher string (compiled locally before sending).
- **params** (*Dict[str, Any] | None*) – Optional parameter dict for Cypher string queries (e.g., `params={"val": 10}` for `$val` references).
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)

- **output_type** (*Literal['all', 'nodes', 'edges', 'shape']*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)
- **output** (*str | None*)

Return type
Plottable

Example:

```
# Chain (existing)
g.gfql_remote([n(), e(), n()])

# Cypher string with params
g.gfql_remote(
    "MATCH (n) WHERE n.score > $cutoff RETURN n",
    params={"cutoff": 10},
)

# GRAPH constructor
g.gfql_remote("GRAPH { MATCH (a)-[r]->(b) WHERE a.score > 5 }")
```

See *chain_remote()* for additional parameter documentation.

gfql_remote_shape(*chain*, *api_token=None*, *dataset_id=None*, *format=None*, *df_export_args=None*, *node_col_subset=None*, *edge_col_subset=None*, *engine='auto'*, *validate=True*, *persist=False*)

Get shape metadata for remote GFQL query execution.

This is the remote shape version of *gfql()*. Returns metadata about the resulting graph without downloading the full data.

See *chain_remote_shape()* for detailed documentation (chain_remote_shape is deprecated).

Parameters

- **chain** (*Chain | List[ASTObject] | ASTLet | Dict[str, None | bool | str | float | int | List[Any] | Dict[str, Any]] | str*)
- **api_token** (*str | None*)
- **dataset_id** (*str | None*)
- **format** (*Literal['json', 'csv', 'parquet'] | None*)
- **df_export_args** (*Dict[str, Any] | None*)
- **node_col_subset** (*List[str] | None*)
- **edge_col_subset** (*List[str] | None*)

- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **validate** (*bool*)
- **persist** (*bool*)

Return type
DataFrame

gfql_validate(**args*, ***kwargs*)

Validate a GFQL/Cypher query without executing it.

Raises structured GFQL exceptions on validation failures and never dispatches query execution operators.

graph(*ig*)

Parameters
ig (*Any*)

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). Shorthand for max_hops. min_hops/max_hops: inclusive traversal bounds; defaults preserve legacy behavior (min=1 unless max=0; max defaults to hops). output_min_hops/output_max_hops: optional output slice applied after traversal; defaults keep all traversed hops up to max_hops. Useful for showing a subrange (e.g., min/max = 2..4 but display only hops 3..4). label_node_hops/label_edge_hops: optional column names for hop numbers (omit or None to skip). Nodes record the first retained hop step they are reached (1 = first expansion); when min_hops prunes shorter branches, labels reflect the shortest retained path. Edges record the hop step that traversed them. label_seeds: when True and labeling, also write hop 0 for seed nodes in the node label column. 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 include_zero_hop_seed: internal Cypher opt-in for exact zero-hop path semantics Note: chain() reverse passes set return_as_wave_front=True and use target_wave_front to constrain reachability. target_wave_front: Only consider these nodes + self._nodes for reachability engine: 'auto', 'pandas', 'cudf' (GPU)

hypergraph(*raw_events=None*, ***, *entity_types=None*, *opts={}*, *drop_na=True*, *drop_edge_attrs=False*, *verbose=True*, *direct=False*, *engine='auto'*, *npartitions=None*, *chunksize=None*, *from_edges=False*, *return_as='graph'*)

Parameters

- **raw_events** (*Any | None*)

- **entity_types** (*List[str] | None*)
- **opts** (*dict*)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)
- **verbose** (*bool*)
- **direct** (*bool*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)
- **npartitions** (*int | None*)
- **chunksize** (*int | None*)
- **from_edges** (*bool*)
- **return_as** (*Literal['graph', 'all', 'entities', 'events', 'edges', 'nodes']*)

Return type
Plottable | *HypergraphResult* | *Any*

igraph2pandas(*ig*)

Parameters
ig (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

infer_labels()

Return type
Plottable

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

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*)

Return type
Plottable

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*)

Parameters

- **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)

- **skip_styling** (*bool*)
- **render_to_disk** (*bool*)
- **path** (*str | None*)
- **format** (*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'] | None*)

Return type
Plottable

layout_igraph(*layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Parameters

- **layout** (*str*)
- **directed** (*bool | None*)
- **use_vids** (*bool*)
- **bind_position** (*bool*)
- **x_out_col** (*str*)
- **y_out_col** (*str*)
- **play** (*int | None*)
- **params** (*dict*)

Return type
Plottable

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*)

Return type
Plottable

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

name(*name*)

Parameters
name (*str*)

Return type
Plottable

networkx2pandas(*G*)

Parameters
G (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

networkx_checkoverlap(*g*)

Parameters
g (*Any*)

Return type
None

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

Parameters

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

Return type
Plottable

pandas2igraph(*edges*, *directed=True*)

Parameters

- edges (*DataFrame*)
- directed (*bool*)

Return type
Any

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

Parameters

- graph_transform (*Callable*)
- args (*Any*)
- kwargs (*Any*)

Return type
Plottable

plot(*graph=None*, *nodes=None*, *name=None*, *description=None*, *render='auto'*, *skip_upload=False*, *as_files=False*, *memoize=True*, *erase_files_on_fail=True*, *extra_html=''*, *override_html_style=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Parameters

- graph (*Any | None*)
- nodes (*Any | None*)
- name (*str | None*)
- description (*str | None*)
- render (*bool | Literal['auto'] | ~typing.Literal['g', 'url', 'ipython', 'databricks', 'browser'] | None*)
- skip_upload (*bool*)
- as_files (*bool*)

- **memoize** (*bool*)
- **erase_files_on_fail** (*bool*)
- **extra_html** (*str*)
- **override_html_style** (*str | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)

Return type
Any

privacy(*mode=None*, *notify=None*, *invited_users=None*, *message=None*, *mode_action=None*)

Parameters

- **mode** (*Literal['private', 'organization', 'public'] | None*)
- **notify** (*bool | None*)
- **invited_users** (*List[str] | None*)
- **message** (*str | None*)
- **mode_action** (*Literal['10', '20'] | None*)

Return type
Plottable

protocol(*v=None*)

Parameters
v (*str | None*)

Return type
str

prune_self_edges()

python_remote_g(**args*, ***kwargs*)

Remotely run Python code on a remote dataset that returns a Plottable

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

Parameters

- **code** (*Union[str, Callable[..., object]]*) – Python code that includes a top-level function *def task(g: Plottable) -> Union[str, Dict]*.
- **api_token** (*Optional[str]*) – Optional JWT token. If not provided, refreshes JWT and uses that.
- **dataset_id** (*Optional[str]*) – Optional dataset_id. If not provided, will fallback to self._dataset_id. If not defined, will upload current data, store that dataset_id, and run code against that.
- **format** (*Optional[FormatType]*) – What format to fetch results. Defaults to 'parquet'.

- **output_type** (*Optional[OutputTypeGraph]*) – What shape of output to fetch. Defaults to 'all'. Options include 'nodes', 'edges', 'all' (both). For other variants, see python_remote_shape and python_remote_json.
- **engine** (*EngineAbstractType*) – Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.
- **run_label** (*Optional[str]*) – Optional label for the run for serverside job tracking.
- **validate** (*bool*) – Whether to locally test code, and if uploading data, the data. Default true.

Return type
Any

Example: Upload data and count the results

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .edges(es, source='src', destination='dst')
    .upload()
assert g1._dataset_id is not None, "Successfully uploaded"
g2 = g1.python_remote_g(
    code='''
        from typing import Any, Dict
        from graphistry import Plottable

        def task(g: Plottable) -> Dict[str, Any]:
            return g
    ''',
    engine='cudf')
num_edges = len(g2._edges)
print(f'num_edges: {num_edges}')
```

python_remote_json(**args*, ***kwargs*)

Remotely run Python code on a remote dataset that returns json

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

Parameters

- **code** (*Union[str, Callable[..., object]]*) – Python code that includes a top-level function *def task(g: Plottable) -> Union[str, Dict]*.
- **api_token** (*Optional[str]*) – Optional JWT token. If not provided, refreshes JWT and uses that.
- **dataset_id** (*Optional[str]*) – Optional dataset_id. If not provided, will fallback to self._dataset_id. If not defined, will upload current data, store that dataset_id, and run code against that.
- **engine** (*EngineAbstractType*) – Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or

'cudf'.

- **run_label** (*Optional [str]*) – Optional label for the run for serverside job tracking.
- **validate** (*bool*) – Whether to locally test code, and if uploading data, the data. Default true.

Return type
Any

Example: Upload data and count the results

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .edges(es, source='src', destination='dst')
    .upload()
assert g1._dataset_id is not None, "Successfully uploaded"
obj = g1.python_remote_json(
    code='''
        from typing import Any, Dict
        from graphistry import Plottable

        def task(g: Plottable) -> Dict[str, Any]:
            return {'num_edges': len(g._edges)}
    ''',
    engine='cudf')
num_edges = obj['num_edges']
print(f'num_edges: {num_edges}')
```

python_remote_table(**args*, ***kwargs*)

Remotely run Python code on a remote dataset that returns a table

Uses the latest bound *_dataset_id*, and uploads current dataset if not already bound. Note that rebinding calls of *edges()* and *nodes()* reset the *_dataset_id* binding.

Parameters

- **code** (*Union [str, Callable [..., object]]*) – Python code that includes a top-level function *def task(g: Plottable) -> Union[str, Dict]*.
- **api_token** (*Optional [str]*) – Optional JWT token. If not provided, refreshes JWT and uses that.
- **dataset_id** (*Optional [str]*) – Optional dataset_id. If not provided, will fallback to self._dataset_id. If not defined, will upload current data, store that dataset_id, and run code against that.
- **format** (*Optional [FormatType]*) – What format to fetch results. Defaults to 'parquet'.
- **output_type** (*Optional [OutputTypeGraph]*) – What shape of output to fetch. Defaults to 'table'. Options include 'table', 'nodes', and 'edges'.
- **engine** (*EngineAbstractType*) – Override which run mode GFQL uses. Defaults to 'auto' which auto-detects based on DataFrame type. Also accepts 'pandas' or 'cudf'.

- **run_label** (*Optional[str]*) – Optional label for the run for serverside job tracking.
- **validate** (*bool*) – Whether to locally test code, and if uploading data, the data. Default true.

Return type
Any

Example: Upload data and count the results

```
import graphistry
from graphistry import n, e
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g1 = graphistry
    .edges(es, source='src', destination='dst')
    .upload()
assert g1._dataset_id is not None, "Successfully uploaded"
edges_df = g1.python_remote_table(
    code='''
        from typing import Any, Dict
        from graphistry import Plottable

        def task(g: Plottable) -> Dict[str, Any]:
            return g._edges
    ''',
    engine='cudf')
num_edges = len(edges_df)
print(f'num_edges: {num_edges}')
```

reset_caches()

Return type
None

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=
```

(continues on next page)

(continued from previous page)

```
↪'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*)

scene_settings(*menu=None*, *info=None*, *show_arrows=None*, *point_size=None*, *edge_curvature=None*, *edge_opacity=None*, *point_opacity=None*)

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*)

Return type
Plottable

search(*query*, *cols=None*, *thresh=5000*, *fuzzy=True*, *top_n=10*)

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*)

Parameters

- **query** (*str*)
- **scale** (*float*)
- **top_n** (*int*)
- **thresh** (*float*)
- **broader** (*bool*)

- **inplace** (*bool*)

Return type
Plottable

server(*v=None*)

Parameters
v (*str | None*)

Return type
str

session: ClientSession

settings(*height=None*, *url_params=None*, *render=None*, *validate='autofix'*, *warn=True*)

Parameters

- **height** (*int | None*)
- **url_params** (*Dict[str, None | str | int | float | bool | List[SettingsValue] | Dict[str, SettingsValue]] | None*)
- **render** (*bool | Literal['auto'] | ~typing.Literal['g', 'url', 'ipython', 'databricks', 'browser'] | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Plottable

style(*fg=None*, *bg=None*, *page=None*, *logo=None*)

Parameters

- **fg** (*Dict[str, Any] | None*)
- **bg** (*Dict[str, Any] | None*)
- **page** (*Dict[str, Any] | None*)
- **logo** (*Dict[str, Any] | None*)

Return type
Plottable

to_arrow(*table=None*, *validate='autofix'*, *warn=True*, *schema_validate=False*, *schema_table='edges'*)

Parameters

- **table** (*Any | None*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **schema_validate** (*Literal['strict', 'autofix'] | bool*)
- **schema_table** (*str*)

Return type
Any | None

`to_cudf()`

Convert to GPU mode by converting any defined nodes and edges to cudf dataframes

When nodes or edges are already cudf dataframes, they are left as is

Parameters
`g` (`Plottable`) – Graphistry object

Returns
Graphistry object

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, include_nodes=True, node_attributes=None, edge_attributes=None, use_vids=False)`

Parameters

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

Return type
Any

`to_pandas()`

Convert nodes and edges to pandas DataFrames.

Supports all input types: cuDF, Arrow, Polars, Spark, dask, and pandas (identity).

Return type
Plottable

`transform(df: DataFrame, y: DataFrame | None = None, kind: str = 'nodes', min_dist: str | float | int = 'auto', n_neighbors: int = 7, merge_policy: bool = False, sample: int | None = None, *, return_graph: Literal[True] = True, scaled: bool = True, verbose: bool = False) → Plottable`

`transform`(*df: DataFrame*, *y: DataFrame | None = None*, *kind: str = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[False]*, *scaled: bool = True*, *verbose: bool = False*) → Tuple[DataFrame, DataFrame]

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

`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={}*)

Parameters

- **df** (*DataFrame*)
- **y** (*DataFrame | None*)
- **kind** (*Literal['nodes', 'edges']*)
- **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*)

Parameters

- **X** (*DataFrame | np.ndarray | List[str] | None*)
- **y** (*DataFrame | np.ndarray | List[str] | None*)
- **kind** (*Literal['nodes', 'edges']*)
- **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['auto', 'cuml', 'umap_learn']*)
- **feature_engine** (*str*)
- **inplace** (*bool*)
- **memoize** (*bool*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

- **featurize_kwargs** (*Any*)

Return type
Plottable | None

umap_fit(*X*, *y=None*, *umap_fit_kwargs={}*)

Parameters

- **X** (*DataFrame*)
- **y** (*DataFrame | None*)
- **umap_fit_kwargs** (*Dict[str, Any]*)

Return type
Plottable

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

- **res** (Plottable)
- **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['auto', 'cuml', 'umap_learn']*)
- **suffix** (*str*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

Return type
Plottable

upload(*memoize=True*, *erase_files_on_fail=True*, *validate='autofix'*, *warn=True*, *schema_validate=False*)

Parameters

- **memoize** (*bool*)
- **erase_files_on_fail** (*bool*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)

- **schema_validate** (*Literal['strict', 'autofix'] | bool*)

Return type
Plottable

property url: str | None

validate_arrow_schema(*table='edges'*, ***, *validate='strict'*, *warn=True*)

Parameters

- **table** (*str*)
- **validate** (*Literal['strict', 'autofix'] | bool*)
- **warn** (*bool*)

Return type
Any | None

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.drop_duplicates_with_warning(*df*)

Parameters
df (*DataFrame*)

Return type
DataFrame

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.**normalize_X_y**(*X*, *y*, *feature_names_in=None*, *target_names_in=None*)

Prepare for most finnicky featurizers: drop duplicates, and remove targets from data

Warns on fixed violations

Parameters

- **X** (*DataFrame*)
- **y** (*DataFrame*)
- **feature_names_in** (*Index | None*)
- **target_names_in** (*Index | None*)

Return type
Tuple[*DataFrame*, *DataFrame*]

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'*)

skrub 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 TableVectorizer
- **n_topics_target** (*int*)
- **categories** (*str | None*)
- **multilabel** (*bool*)
- **feature_engine** (*Literal['none', 'pandas', 'skrub', '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', 'skrub', '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 skrub

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', 'skrub', '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

Return type
DataFrame

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', 'skrub', 'torch', 'dirty_cat', 'auto']*)

Return type
Literal['none', 'pandas', 'skrub', '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', 'skrub', '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', 'skrub', '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*, *feature_names_in*, *target_names_in*)

Parameters

- **df** (*DataFrame*)
- **ydf** (*DataFrame | None*)
- **res** (*List*)
- **kind** (*str*)
- **feature_names_in** (*Index*)
- **target_names_in** (*Index*)

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*)

graphistry.feature_utils.**FeatureEngine**
alias of Literal['none', 'pandas', 'skrub', 'torch', 'dirty_cat', 'auto']

graphistry.feature_utils.**FeatureEngineConcrete**
alias of Literal['none', 'pandas', 'skrub', 'torch']

10.10.6.2 UMAP

class graphistry.umap_utils.**UMAPMixin**(**a*, ***kw*)
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: DataFrame*, *y: DataFrame | None = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[True] = True*, *fit_umap_embedding: bool = True*, *umap_transform_kwargs: Dict[str, Any] = {}*) → *Plottable*

transform_umap(*df: DataFrame*, *y: DataFrame | None = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *min_dist: str | float | int = 'auto'*, *n_neighbors: int = 7*, *merge_policy: bool = False*, *sample: int | None = None*, ***, *return_graph: Literal[False]*, *fit_umap_embedding: bool = True*, *umap_transform_kwargs: Dict[str, Any] = {}*) → Tuple[DataFrame, DataFrame, DataFrame]

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

umap(*X: DataFrame | ndarray | List[str] | None = None*, *y: DataFrame | ndarray | List[str] | None = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *scale: float = 1.0*, *n_neighbors: int = 12*, *min_dist: float = 0.1*, *spread: float = 0.5*, *local_connectivity: int = 1*, *repulsion_strength: float = 1*, *negative_sample_rate: int = 5*, *n_components: int = 2*, *metric: str = 'euclidean'*, *suffix: str = ''*, *play: int | None = 0*, *encode_position: bool = True*, *encode_weight: bool = True*, *dbscan: bool = False*, *engine: Literal['cuml', 'umap_learn', 'auto'] = 'auto'*, *feature_engine: str = 'auto'*, *inplace: Literal[False] = False*, *memoize: bool = True*, *umap_kwargs: Dict[str, Any] = {}*, *umap_fit_kwargs: Dict[str, Any] = {}*, *umap_transform_kwargs: Dict[str, Any] = {}*, ***featurize_kwargs*) → *Plottable*

umap(*X: DataFrame | ndarray | List[str] | None = None*, *y: DataFrame | ndarray | List[str] | None = None*, *kind: Literal['nodes', 'edges'] = 'nodes'*, *scale: float = 1.0*, *n_neighbors: int = 12*, *min_dist: float = 0.1*, *spread: float = 0.5*, *local_connectivity: int = 1*, *repulsion_strength: float = 1*, *negative_sample_rate: int = 5*, *n_components: int = 2*, *metric: str = 'euclidean'*, *suffix: str = ''*, *play: int | None = 0*, *encode_position: bool = True*, *encode_weight: bool = True*, *dbscan: bool = False*, *engine: Literal['cuml', 'umap_learn', 'auto'] = 'auto'*, *feature_engine: str = 'auto'*, ***, *inplace: Literal[True]*, *memoize: bool = True*, *umap_kwargs: Dict[str, Any] = {}*, *umap_fit_kwargs: Dict[str, Any] = {}*, *umap_transform_kwargs: Dict[str, Any] = {}*, ***featurize_kwargs*) → None

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']}))
>>> 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", "skrub", "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

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

- **res** (Plottable)
- **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_umap_gpu_dataframes**(*X*, *y*, *engine*)

Parameters

- **X** (*DataFrame*)
- **y** (*DataFrame | None*)
- **engine** (*Literal['cuml', 'umap_learn']*)

Return type
Tuple[*DataFrame*, *DataFrame* | None]

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*)

Return type
Plottable | None

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/latest/graphistry/c*

Parameters
engine (*Literal['cuml', 'umap_learn']*)

graphistry.umap_utils.**umap_model_to_engine**(*v*)

Parameters
v (*Any*)

Return type
Literal['cuml', 'umap_learn'] | None

10.10.6.3 Semantic Search

class graphistry.text_utils.**SearchToGraphMixin**(**a*, ***kw*)

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*)

Return type
Tuple[*DataFrame*, *ndarray*[tuple[*Any*, ...], *dtype*[*float32*]] | *ndarray*[tuple[*Any*, ...], *dtype*[*float64*]] | None]

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*)

Return type
Plottable

10.10.6.4 DBSCAN

class graphistry.compute.cluster.**ClusterMixin**(*a, **kw)

Bases: object

dbscan(*min_dist=0.2*, *min_samples=1*, *cols=None*, *kind='nodes'*, *fit_umap_embedding=True*, *target=False*, *verbose=False*, *engine_dbscan='auto'*, *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.

Saves model as g._dbscan_nodes or g._dbscan_edges

Examples:

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')

# cluster by UMAP embeddings
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** (*Literal['nodes', 'edges']*)
- **fit_umap_embedding** (*bool*)
- **target** (*bool*)
- **verbose** (*bool*)
- **engine_dbscan** (*Literal['cuml', 'sklearn', 'auto']*)

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_inplace**(*res*, *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.

Sets:

- *res._dbscan_edges* or *res._dbscan_nodes* to the DBSCAN model
- *res._edges* or *res._nodes* gains column *_dbscan*

Args:

res
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)

target
whether to use the target dataframe or features dataframe (typically False, for features)

Parameters

- **res** (Plottable)
- **dbscan** (*Any*)
- **kind** (*Literal['nodes', 'edges']*)
- **cols** (*List | str | None*)
- **use_umap_embedding** (*bool*)
- **target** (*bool*)
- **verbose** (*bool*)

Return type
None

graphistry.compute.cluster.**dbscan_predict_cuml**(*X*, *model*)

Parameters

- **X** (*Any*)
- **model** (*Any*)

Return type
Any

graphistry.compute.cluster.**dbscan_predict_sklearn**(*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*)

Return type
ndarray

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

- **g** (Plottable)
- **kind** (*Literal['nodes', 'edges']*)
- **cols** (*List | str | None*)

Return type
Any

graphistry.compute.cluster.**make_safe_gpu_dataframes**(*X*, *y*, *engine*)

Coerce a dataframe to pd vs cudf based on engine

Parameters

- **X** (*Any | None*)
- **y** (*Any | None*)
- **engine** (*Engine*)

Return type
Tuple[*Any* | None, *Any* | None]

graphistry.compute.cluster.**resolve_dbscan_engine**(*engine*, *g_or_df=None*)

Resolves the engine to use for DBSCAN clustering

If 'auto', decide by checking if cuml or sklearn is installed, and if provided, natural type of the dataset. GPU is used if both a GPU dataset and GPU library is installed. Otherwise, CPU library.

Parameters

- **engine** (*Literal['cuml', 'sklearn', 'auto']*)
- **g_or_df** (*Any | None*)

Return type
Literal['cuml', 'sklearn']

10.10.6.5 RGCN

class graphistry.networks.**DotProductPredictor**

Bases: object

forward(*graph*, *h*)

class graphistry.networks.**GCN**(*in_feats*, *h_feats*, *num_classes*)

Bases: object

forward(*g*, *in_feat*)

class graphistry.networks.**HeteroClassifier**(*in_dim*, *hidden_dim*, *n_classes*, *rel_names*)

Bases: object

forward(*g*)

class graphistry.networks.**HeteroEmbed**(*num_nodes*, *num_rels*, *d*, *proto*, *node_features=None*, *device='cpu'*, *reg=0.01*)

Bases: object

Parameters

- **num_nodes** (*int*)
- **num_rels** (*int*)
- **d** (*int*)

loss(*node_embedding*, *triplets*, *labels*)

score(*node_embedding*, *triplets*)

class graphistry.networks.**LinkPredModel**(*in_features*, *hidden_features*, *out_features*)

Bases: object

forward(*g*, *x*)

class graphistry.networks.**LinkPredModelMultiOutput**(*in_features*, *hidden_features*, *out_features*, *out_classes*)

Bases: object

`embed`(*g*, *x*)

`forward`(*g*, *x*)

`class graphistry.networks.MLPPredictor`(*in_features*, *out_classes*)

Bases: `object`

One can also write a prediction function that predicts a vector for each edge with an MLP. Such vector can be used in further downstream tasks, e.g. as logits of a categorical distribution.

`apply_edges`(*edges*)

`forward`(*graph*, *h*)

`class graphistry.networks.RGCN`(*in_feats*, *hid_feats*, *out_feats*, *rel_names*)

Bases: `object`

Heterograph where we gather message from neighbors along all edge types. You can use the module dgl.nn.pytorch.HeteroGraphConv (also available in MXNet and Tensorflow) to perform message passing on all edge types, then combining different graph convolution modules for each edge type.

:returns
torch model with forward pass methods useful for fitting model in standard way

`forward`(*graph*, *inputs*)

`class graphistry.networks.RGCNEmbed`(*d*, *num_nodes*, *num_rels*, *hidden=None*, *device='cpu'*)

Bases: `object`

`forward`(*g*, *node_features=None*)

`class graphistry.networks.SAGE`(*in_feats*, *hid_feats*, *out_feats*)

Bases: `object`

`forward`(*graph*, *inputs*)

`graphistry.networks.train_link_pred`(*model*, *G*, *epochs=100*, *use_cross_entropy_loss=False*)

10.10.6.6 HeterographEmbedModuleMixin

`class graphistry.embed_utils.EmbedDistScore`

Bases: `object`

`static DistMult`(*h*, *r*, *t*)

Parameters

- **h** (*Any*)
- **r** (*Any*)
- **t** (*Any*)

Return type
Any

`static RotatE(h, r, t)`

Parameters

- **h** (*Any*)
- **r** (*Any*)
- **t** (*Any*)

Return type
Any

`static TransE(h, r, t)`

Parameters

- **h** (*Any*)
- **r** (*Any*)
- **t** (*Any*)

Return type
Any

`class graphistry.embed_utils.HeterographEmbedModuleMixin(*args, **kwargs)`

Bases: `ComputeMixin`

`embed(relation, proto='DistMult', embedding_dim=32, use_feat=False, X=None, epochs=2, batch_size=32, train_split=0.8, sample_size=1000, num_steps=50, lr=0.01, inplace=False, device='cpu', evaluate=True, *args, **kwargs)`

Embed a graph using a relational graph convolutional network (RGCN), and return a new graphistry graph with the embeddings as node attributes.

Parameters

relation
[str] column to use as relation between nodes

proto
[ProtoSymbolic] metric to use, ['TransE', 'RotateE', 'DistMult'] or provide your own. Defaults to 'DistMult'.

embedding_dim
[int] relation embedding dimension. defaults to 32

use_feat
[bool] wether to featurize nodes, if False will produce random embeddings and shape them during training. Defaults to True

X
[XSymbolic] Which columns in the nodes dataframe to featurize. Inherets args from graphistry.featurize(). Defaults to None.

epochs
[int] Number of training epochs. Defaults to 2

batch_size
[int] batch_size. Defaults to 32

train_split
[Union[float, int]] train percentage, between 0, 1. Defaults to 0.8.

sample_size
[int] sample size. Defaults to 1000

num_steps
[int] num_steps. Defaults to 50

lr
[float] learning rate. Defaults to 0.002

inplace
[Optional[bool]] inplace

device
[Optional[str]] accelarator. Defaults to "cpu"

evaluate
[bool] Whether to evaluate. Defaults to False.

Returns

self : graphistry instance

Parameters

- **relation** (*str*)
- **proto** (*str | Callable[[Any, Any, Any], Any] | None*)
- **embedding_dim** (*int*)
- **use_feat** (*bool*)
- **X** (*DataFrame | ndarray | List[str] | None*)
- **epochs** (*int*)
- **batch_size** (*int*)
- **train_split** (*float | int*)
- **sample_size** (*int*)
- **num_steps** (*int*)
- **lr** (*float*)
- **inplace** (*bool | None*)
- **device** (*str | None*)
- **evaluate** (*bool*)

Return type
Plottable

predict_links(*source=None*, *relation=None*, *destination=None*, *threshold=0.5*, *anomalous=False*, *retain_old_edges=False*, *return_dataframe=False*)

predict_links over all the combinations of given source, relation, destinations.

Parameters

source: list
Targeted source nodes. Defaults to None(all).

relation: list
Targeted relations. Defaults to None(all).

destination: list
Targeted destinations. Defaults to None(all).

threshold
[Optional[float]] Probability threshold. Defaults to 0.5

retain_old_edges
[Optional[bool]] will include old edges in predicted graph. Defaults to False.

return_dataframe
[Optional[bool]] will return a dataframe instead of a graphistry instance. Defaults to False.

anomalous
[Optional[False]] will return the edges < threshold or low confidence edges(anomaly).

Returns

Graphistry Instance
containing the corresponding source, relation, destination and score column where score >= threshold if anamalous if False else score <= threshold, or a dataframe

Parameters

- `source` (*list* / *None*)
- `relation` (*list* / *None*)
- `destination` (*list* / *None*)
- `threshold` (*float* / *None*)
- `anomalous` (*bool* / *None*)
- `retain_old_edges` (*bool* / *None*)
- `return_dataframe` (*bool* / *None*)

Return type
Plottable

`predict_links_all`(*threshold=0.5*, *anomalous=False*, *retain_old_edges=False*, *return_dataframe=False*)

predict_links over entire graph given a threshold

Parameters

threshold
[Optional[float]] Probability threshold. Defaults to 0.5

anomalous
[Optional[False]] will return the edges < threshold or low confidence edges(anomaly).

retain_old_edges
[Optional[bool]] will include old edges in predicted graph. Defaults to False.

return_dataframe: Optional[bool]
will return a dataframe instead of a graphistry instance. Defaults to False.

Returns

Plottable
graphistry graph instance containing all predicted/anomalous links or dataframe

Parameters

- **threshold** (*float | None*)
- **anomalous** (*bool | None*)
- **retain_old_edges** (*bool | None*)
- **return_dataframe** (*bool | None*)

Return type
Plottable

`class graphistry.embed_utils.SubgraphIterator(g, sample_size=3000, num_steps=1000)`

Bases: `object`

Parameters

- **sample_size** (*int*)
- **num_steps** (*int*)

`graphistry.embed_utils.check_cudf()`

`graphistry.embed_utils.log(msg)`

Parameters
msg (*str*)

Return type
None

10.10.7 Kepler API Reference

Geographic visualization with Kepler.gl integration.

The Kepler module provides classes and methods for creating interactive map-based visualizations using Kepler.gl. This includes configuration for datasets, layers, and complete geographic visualizations.

Note

Graphistry automatically performs geographic visualization when `point_longitude` and `point_latitude` are bound. The Kepler API provides additional control for customizing datasets, layers, and map styling.

10.10.7.1 Quick Links

- *Maps & Geographic Visualization*: Maps & geographic visualization guide
- *Mercator Layout*: Mercator projection layout (optional)
- https://docs.kepler.gl/

10.10.7.2 API Components

Configuration Classes

KeplerDataset

Configuration class for Kepler.gl datasets.

`class graphistry.kepler.KeplerDataset(`*raw_dict: Dict[str, Any]*`)`

`class graphistry.kepler.KeplerDataset(`*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['nodes']*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*`)`

`class graphistry.kepler.KeplerDataset(`*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['edges']*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*, *map_node_coords: bool | None = None*, *map_node_coords_mapping: Dict[str, str] | None = None*`)`

`class graphistry.kepler.KeplerDataset(`*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['countries', 'zeroOrderAdminRegions']*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*, *resolution: Literal[10, 50, 110] | None = None*, *boundary_lakes: bool | None = None*, *filter_countries_by_col: str | None = None*, *include_countries: List[str] | None = None*, *exclude_countries: List[str] | None = None*`)`

class graphistry.kepler.KeplerDataset(*raw_dict: None = None*, ***, *id: str | None = None*, *type: Literal['states', 'provinces', 'firstOrderAdminRegions']*, *label: str | None = None*, *include: List[str] | None = None*, *exclude: List[str] | None = None*, *computed_columns: Dict[str, Any] | None = None*, *boundary_lakes: bool | None = None*, *filter_countries_by_col: str | None = None*, *include_countries: List[str] | None = None*, *exclude_countries: List[str] | None = None*, *filter_1st_order_regions_by_col: str | None = None*, *include_1st_order_regions: List[str] | None = None*, *exclude_1st_order_regions: List[str] | None = None*)

Bases: `object`

Configure a Kepler.gl dataset for visualization.

Creates a dataset configuration that makes Graphistry data (nodes/edges) or geographic data (countries/states) available to Kepler.gl for visualization.

Common parameters (all dataset types):

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl dataset dictionary (if provided, all other params ignored)
- **id** (*Optional[str]*) – Dataset identifier (auto-generated if None)
- **type** (*Optional[str]*) – Dataset type - 'nodes', 'edges', 'countries', 'states', etc.
- **label** (*Optional[str]*) – Display label (defaults to id)
- **include** (*Optional[List[str]]*) – Columns to include (whitelist)
- **exclude** (*Optional[List[str]]*) – Columns to exclude (blacklist)
- **computed_columns** (*Optional[Dict[str, Any]]*) – Computed/aggregated columns for data enrichment
- **kwargs** (*Any*)

For nodes type:

No additional parameters beyond common ones.

For edges type:

Parameters

- **map_node_coords** (*Optional[bool]*) – Auto-map source/target node coordinates to edges (adds columns: edgeSourceLatitude, edgeSourceLongitude, edgeTargetLatitude, edgeTargetLongitude)
- **map_node_coords_mapping** (*Optional[Dict[str, str]]*) – Custom column names for mapped coordinates. Dict mapping default names to custom names, e.g., {"edgeSourceLongitude": "src_lng", "edgeSourceLatitude": "src_lat", "edgeTargetLongitude": "dst_lng", "edgeTargetLatitude": "dst_lat"}
- **raw_dict** (*Dict[str, Any] | None*)
- **id** (*str | None*)
- **type** (*str | None*)
- **label** (*str | None*)

- **kwargs** (*Any*)

For countries/zeroOrderAdminRegions type:

Parameters

- **resolution** (*Optional[Literal[10, 50, 110]]*) – Map resolution (10=high, 50=medium, 110=low)
- **boundary_lakes** (*Optional[bool]*) – Include lake boundaries (default: True)
- **filter_countries_by_col** (*Optional[str]*) – Column to filter countries
- **include_countries** (*Optional[List[str]]*) – Countries to include
- **exclude_countries** (*Optional[List[str]]*) – Countries to exclude
- **raw_dict** (*Dict[str, Any] | None*)
- **id** (*str | None*)
- **type** (*str | None*)
- **label** (*str | None*)
- **kwargs** (*Any*)

For states/provinces/firstOrderAdminRegions type:

Parameters

- **boundary_lakes** (*Optional[bool]*) – Include lake boundaries (default: True)
- **filter_countries_by_col** (*Optional[str]*) – Column to filter countries
- **include_countries** (*Optional[List[str]]*) – Countries to include
- **exclude_countries** (*Optional[List[str]]*) – Countries to exclude
- **filter_1st_order_regions_by_col** (*Optional[str]*) – Column to filter regions
- **include_1st_order_regions** (*Optional[List[str]]*) – Regions to include
- **exclude_1st_order_regions** (*Optional[List[str]]*) – Regions to exclude
- **raw_dict** (*Dict[str, Any] | None*)
- **id** (*str | None*)
- **type** (*str | None*)
- **label** (*str | None*)
- **kwargs** (*Any*)

Example: Node dataset

```
from graphistry import KeplerDataset

# Basic node dataset
ds = KeplerDataset(id="companies", type="nodes", label="Companies")

# With column filtering
ds = KeplerDataset(
    type="nodes",
    include=["name", "latitude", "longitude", "revenue"]
)
```

Example: Edge dataset with coordinate mapping

```
# Auto-map source/target node coordinates to edges
ds = KeplerDataset(
    type="edges",
    map_node_coords=True
)
```

Example: Countries with computed columns

```
# High-resolution countries with aggregated metrics
ds = KeplerDataset(
    type="countries",
    resolution=10,
    computed_columns={
        "avg_revenue": {
            "type": "aggregate",
            "computeFromDataset": "companies",
            "sourceKey": "country",
            "targetKey": "name",
            "aggregate": "mean",
            "aggregateCol": "revenue"
        }
    }
)
```

Example: Using raw_dict

```
# Pass through native Kepler.gl dataset dict
ds = KeplerDataset({
    "info": {"id": "my-dataset", "label": "My Data"},
    "data": {...}
})
```

`id: str | None`

`label: str | None`

`to_dict()`

Serialize to dictionary format for Kepler.gl.

Return type
Dict[str, *Any*]

`type: str | None`

Note

For the native Kepler.gl dataset format when using `raw_dict`, see *Kepler.gl Dataset Format*.

Computed Columns

computed_columns (dict, optional)

Define computed columns for data enrichment. Each computed column is added as a new column to the current dataset (the dataset where `computed_columns` is defined). The key in the dictionary becomes the new column name.

Structure:

```
{
    "new_column_name": {          # The key becomes the new column name in THIS␣
↪dataset
        "type": "aggregate",              # Aggregation type
        "computeFromDataset": "source_dataset_id",
        "sourceKey": "join_column",       # Column in source dataset
        "targetKey": "join_column",       # Column in target (this) dataset
        "aggregate": "mean",              # Aggregation function: mean, sum, min,␣
↪max, count
        "aggregateCol": "value_column",   # Column to aggregate
        "normalizer": "mean",             # Optional: normalize by another␣
↪aggregation
        "normalizerCol": "divisor_col",   # Optional: column for normalization
        "bins": [0, 1, 2, 5, 10],        # Optional: bin continuous values
        "right": False,                   # Optional: bin right-inclusivity
        "includeLowest": True             # Optional: include lowest bin edge
    }
}
```

Example: A countries dataset can create `avg_revenue` by aggregating company revenue via country name.

Computed Column Fields:

- `type` (str): Currently supports “aggregate”
- `computeFromDataset` (str): ID of the dataset to aggregate from (the source)
- `sourceKey` (str): Join column in the source dataset
- `targetKey` (str): Join column in the target dataset (this dataset)
- `aggregate` (str): Aggregation function name as string. Common options: “mean”, “sum”, “min”, “max”, “count”, “std”, “var”, “median”, “first”, “last”, “prod”, “nunique”. See https://docs.rapids.ai/api/cudf/stable/user_guide/api_docs/groupby.html for full list.
- `aggregateCol` (str): Column name to aggregate from the source dataset
- `normalizer` (str, optional): Secondary aggregation function for normalization (e.g., divide mean by mean). Uses same aggregation function names as `aggregate`.
- `normalizerCol` (str, optional): Column for normalization denominator
- `bins` (List[float], optional): Bin edges for discretizing continuous values
- `right` (bool, optional): Whether bins are right-inclusivity
- `includeLowest` (bool, optional): Whether to include the lowest bin edge

Example

```
# Aggregate data from another dataset
countries_with_stats = KeplerDataset(
    id="countries-stats",
    type="countries",
    resolution=110,
    computed_columns={
        "avg_revenue": {
            "type": "aggregate",
            "computeFromDataset": "companies",
            "sourceKey": "country",
            "targetKey": "name",
            "aggregate": "mean",
            "aggregateCol": "revenue"
        }
    }
)
```

See Also

- *Kepler.gl Dataset Format*: Native Kepler.gl dataset format reference
- *KeplerLayer*: Layer configuration
- *KeplerEncoding*: Complete Kepler configuration
- *Maps & Geographic Visualization*: User guide with examples

KeplerLayer

Configuration class for Kepler.gl visualization layers.

class graphistry.kepler.**KeplerLayer**(*raw_dict*)

Bases: `object`

Configure a Kepler.gl visualization layer.

Creates a layer configuration using native Kepler.gl format. Layers define how datasets are visualized on the map (points, arcs, hexbins, etc.). This class accepts only raw dictionary format for now.

Parameters
raw_dict (*Dict[str, Any]*) – Native Kepler.gl layer configuration dictionary

Example: Point layer

```
from graphistry import KeplerLayer

layer = KeplerLayer({
    "id": "cities",
    "type": "point",
    "config": {
        "dataId": "companies",
```

(continues on next page)

(continued from previous page)

```
        "label": "Company Locations",
        "columns": {"lat": "latitude", "lng": "longitude"},
        "color": [255, 140, 0],
        "visConfig": {"radius": 10, "opacity": 0.8}
    }
})
```

Example: Arc layer for connections

```
layer = KeplerLayer({
    "id": "connections",
    "type": "arc",
    "config": {
        "dataId": "relationships",
        "columns": {
            "lat0": "edgeSourceLatitude",
            "lng0": "edgeSourceLongitude",
            "lat1": "edgeTargetLatitude",
            "lng1": "edgeTargetLongitude"
        },
        "color": [0, 200, 255],
        "visConfig": {"opacity": 0.3, "thickness": 2}
    }
})
```

Example: Hexagon aggregation

```
layer = KeplerLayer({
    "id": "density",
    "type": "hexagon",
    "config": {
        "dataId": "events",
        "columns": {"lat": "latitude", "lng": "longitude"},
        "visConfig": {
            "worldUnitSize": 1,
            "elevationScale": 5,
            "enable3d": True
        }
    }
})
```

`to_dict()`

Serialize to dictionary format for Kepler.gl.

Return type
Dict[str, *Any*]

Note

`KeplerLayer` accepts only native Kepler.gl layer dictionaries via the `raw_dict` parameter. For the complete layer format reference including all layer types and configuration options, see *Kepler.gl Layer Format*.

See Also

- *Kepler.gl Layer Format*: Native Kepler.gl layer format reference
- *KeplerDataset*: Dataset configuration
- *KeplerEncoding*: Complete Kepler configuration
- *Maps & Geographic Visualization*: User guide with examples

KeplerOptions

Configuration class for Kepler.gl visualization options.

class graphistry.kepler.**KeplerOptions**(*raw_dict: Dict[str, Any]*)

class graphistry.kepler.**KeplerOptions**(*raw_dict: None = None*, ***, *center_map: bool | None = None*, *read_only: bool | None = None*)

Bases: `object`

Configure Kepler.gl visualization options.

Controls map behavior and interaction settings such as auto-centering and read-only mode.

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl options dictionary (if provided, kwargs ignored)
- **center_map** (*Optional[bool]*) – Auto-center map on data (default: True)
- **read_only** (*Optional[bool]*) – Disable map interactions (default: False)
- **kwargs** (*Any*)

Example: Structured parameters

```
from graphistry import KeplerOptions

# Auto-center with interactions enabled
opts = KeplerOptions(center_map=True, read_only=False)

# Read-only mode for presentations
opts = KeplerOptions(read_only=True)
```

Example: Using raw_dict

```
# Pass native format
opts = KeplerOptions({"centerMap": True, "readOnly": False})
```

to_dict()

Serialize to dictionary format for Kepler.gl.

Return type
Dict[str, *Any*]

See Also

- *KeplerConfig*: Configuration settings
- *KeplerEncoding*: Complete Kepler configuration
- *Maps & Geographic Visualization*: User guide with examples
- https://docs.kepler.gl/docs/api-reference/advanced-usages/custom-map-style

KeplerConfig

Configuration class for Kepler.gl map and rendering settings.

class graphistry.kepler.**KeplerConfig**(*raw_dict: Dict[str, Any]*)

class graphistry.kepler.**KeplerConfig**(*raw_dict: None = None*, ***, *cull_unused_columns: bool | None = None*, *overlay_blending: Literal['normal', 'additive', 'subtractive'] | None = None*, *tile_style: Dict[str, Any] | None = None*, *auto_graph_renderer_switching: bool | None = None*)

Bases: `object`

Configure Kepler.gl map and rendering settings.

Controls map appearance, data optimization, and layer blending behavior.

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl config dictionary (if provided, kwargs ignored)
- **cull_unused_columns** (*Optional[bool]*) – Remove unused columns from datasets (default: True)
- **overlay_blending** (*Optional[Literal['normal', 'additive', 'subtractive']]*) – Blend mode - 'normal', 'additive', 'subtractive' (default: 'normal')
- **tile_style** (*Optional[Dict[str, Any]]*) – Base map tile style configuration
- **auto_graph_renderer_switching** (*Optional[bool]*) – Enable automatic graph renderer switching, which allows Graphistry to hide Kepler node and edge layers depending on the mode (default: True)
- **kwargs** (*Any*)

Example: Structured parameters

```
from graphistry import KeplerConfig

# Optimize data transfer
cfg = KeplerConfig(cull_unused_columns=True)

# Additive blending for heatmaps
cfg = KeplerConfig(overlay_blending='additive')

# Custom dark base map
cfg = KeplerConfig(
    tile_style={
```

(continues on next page)

(continued from previous page)

```
        "id": "dark",
        "label": "Dark Mode",
        "url": "mapbox://styles/mapbox/dark-v10"
    }
)
```

Example: Using raw_dict

```
# Pass native format
cfg = KeplerConfig({
    "cullUnusedColumns": True,
    "overlayBlending": "additive"
})
```

`to_dict()`

Serialize to dictionary format for Kepler.gl.

Return type
Dict[str, *Any*]

See Also

- *KeplerOptions*: Visualization options
- *KeplerEncoding*: Complete Kepler configuration
- *Maps & Geographic Visualization*: User guide with examples
- https://docs.kepler.gl/docs/api-reference/advanced-usages/using-updaters

KeplerEncoding

Immutable container for complete Kepler.gl configuration.

`class graphistry.kepler.KeplerEncoding(`*datasets=None*, *layers=None*, *options=None*, *config=None*`)`

Bases: `object`

Immutable container for complete Kepler.gl configuration.

Combines datasets, layers, options, and config into a complete Kepler visualization configuration. Follows an immutable builder pattern where methods return new instances rather than modifying in place.

Parameters

- **datasets** (*Optional [List [KeplerDataset]]*) – List of dataset configurations
- **layers** (*Optional [List [KeplerLayer]]*) – List of layer configurations
- **options** (*Optional [KeplerOptions]*) – Visualization options
- **config** (*Optional [KeplerConfig]*) – Map configuration settings

Example: Building configuration with method chaining

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

encoding = (
    KeplerEncoding()
    .with_dataset(KeplerDataset(id="companies", type="nodes"))
    .with_dataset(KeplerDataset(id="relationships", type="edges"))
    .with_layer(KeplerLayer({
        "id": "points",
        "type": "point",
        "config": {
            "dataId": "companies",
            "columns": {"lat": "latitude", "lng": "longitude"}
        }
    }))
    .with_options(center_map=True, read_only=False)
    .with_config(cull_unused_columns=True)
)
```

Example: Using with Plotter

```
import graphistry

g = graphistry.nodes(df, 'id')
g = g.encode_kepler(encoding)
g.plot()
```

`to_dict()`

Serialize to dictionary format for Kepler.gl.

Returns
Dictionary with datasets, layers, options, and config

Return type
Dict[str, Any]

`with_config`(*config:* KeplerConfig) → *KeplerEncoding*

`with_config`(*config: None = None, *, cull_unused_columns: bool | None = None, overlay_blending: Literal['normal', 'additive', 'subtractive'] | None = None, tile_style: Dict[str, Any] | None = None, auto_graph_renderer_switching: bool | None = None*) → *KeplerEncoding*

Return a new KeplerEncoding with updated config.

Parameters

- **config** (*Optional [KeplerConfig]*) – KeplerConfig object to replace current config
- **cull_unused_columns** (*Optional [bool]*) – Remove columns not used by layers
- **overlay_blending** (*Optional [Literal ['normal', 'additive', 'subtractive']]*) – Blend mode - 'normal', 'additive', 'subtractive'
- **tile_style** (*Optional [Dict [str, Any]]*) – Base map tile style configuration
- **auto_graph_renderer_switching** (*Optional [bool]*) – Enable automatic graph renderer switching

Returns
New KeplerEncoding instance with updated config

Return type
KeplerEncoding

`with_dataset`(*dataset*)

Return a new KeplerEncoding with the dataset appended.

Parameters
dataset (KeplerDataset) – KeplerDataset to append

Returns
New KeplerEncoding instance with the dataset added

Return type
KeplerEncoding

`with_layer`(*layer*)

Return a new KeplerEncoding with the layer appended.

Parameters
layer (KeplerLayer) – KeplerLayer to append

Returns
New KeplerEncoding instance with the layer added

Return type
KeplerEncoding

`with_options`(*options:* KeplerOptions) → *KeplerEncoding*

`with_options`(*options: None = None*, ***, *center_map: bool | None = None*, *read_only: bool | None = None*) → *KeplerEncoding*

Return a new KeplerEncoding with updated options.

Parameters

- **options** (*Optional[KeplerOptions]*) – KeplerOptions object to replace current options
- **center_map** (*Optional[bool]*) – Auto-center map on data
- **read_only** (*Optional[bool]*) – Disable map interactions

Returns
New KeplerEncoding instance with updated options

Return type
KeplerEncoding

Examples

Basic Configuration

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

# Start with empty encoding
encoding = KeplerEncoding()
```

(continues on next page)

(continued from previous page)

```
# Add dataset
encoding = encoding.with_dataset(
    KeplerDataset(id="nodes", type="nodes", label="Companies")
)

# Add layer
encoding = encoding.with_layer(
    KeplerLayer({
        "id": "point-layer",
        "type": "point",
        "config": {
            "dataId": "nodes",
            "columns": {"lat": "latitude", "lng": "longitude"}
        }
    })
)

# Configure options
encoding = encoding.with_options(center_map=True, read_only=False)

# Apply to graph
g = g.encode_kepler(encoding)
```

Chained Builder Pattern

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

# Chain multiple operations
encoding = (
    KeplerEncoding()
    .with_dataset(KeplerDataset(id="companies", type="nodes"))
    .with_dataset(KeplerDataset(id="relationships", type="edges"))
    .with_layer(KeplerLayer({
        "id": "nodes",
        "type": "point",
        "config": {"dataId": "companies", "columns": {"lat": "latitude", "lng":
↪"longitude"}}
    }))
    .with_layer(KeplerLayer({
        "id": "edges",
        "type": "arc",
        "config": {
            "dataId": "relationships",
            "columns": {
                "lat0": "edgeSourceLatitude",
                "lng0": "edgeSourceLongitude",
                "lat1": "edgeTargetLatitude",
                "lng1": "edgeTargetLongitude"
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }))
    .with_options(center_map=True)
    .with_config(cull_unused_columns=False)
)

g = g.encode_kepler(encoding)
```

Constructor Pattern

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

# Pass all components to constructor
encoding = KeplerEncoding(
    datasets=[
        KeplerDataset(id="nodes", type="nodes", label="Companies"),
        KeplerDataset(id="edges", type="edges", label="Relationships")
    ],
    layers=[
        KeplerLayer({"id": "point-layer", "type": "point", "config": {...}}),
        KeplerLayer({"id": "arc-layer", "type": "arc", "config": {...}})
    ],
    options=KeplerOptions(center_map=True, read_only=False),
    config=KeplerConfig(cull_unused_columns=False)
)

g = g.encode_kepler(encoding)
```

Multi-Layer Visualization

```
# Create complex visualization with multiple layers
encoding = (
    KeplerEncoding()

    # Datasets
    .with_dataset(KeplerDataset(id="companies", type="nodes"))
    .with_dataset(KeplerDataset(id="relationships", type="edges", map_node_coords=True))
    .with_dataset(KeplerDataset(
        id="countries",
        type="countries",
        resolution=50,
        computed_columns={
            "avg_revenue": {
                "type": "aggregate",
                "computeFromDataset": "companies",
                "sourceKey": "country",
                "targetKey": "name",
                "aggregate": "mean",
```

(continues on next page)

(continued from previous page)

```
                "aggregateCol": "revenue"
            }
        }
    ))

    # Layers
    .with_layer(KeplerLayer({
        "id": "companies-points",
        "type": "point",
        "config": {
            "dataId": "companies",
            "label": "Company Locations",
            "columns": {"lat": "latitude", "lng": "longitude"},
            "isVisible": True,
            "color": [255, 140, 0]
        }
    }))
    .with_layer(KeplerLayer({
        "id": "relationships-arcs",
        "type": "arc",
        "config": {
            "dataId": "relationships",
            "label": "Business Relationships",
            "columns": {
                "lat0": "edgeSourceLatitude",
                "lng0": "edgeSourceLongitude",
                "lat1": "edgeTargetLatitude",
                "lng1": "edgeTargetLongitude"
            },
            "isVisible": False,
            "color": [100, 200, 200],
            "visConfig": {"opacity": 0.2}
        }
    }))
    .with_layer(KeplerLayer({
        "id": "countries-geojson",
        "type": "geojson",
        "config": {
            "dataId": "countries",
            "label": "Countries by Avg Revenue",
            "columns": {"geojson": "_geometry"},
            "isVisible": True
        },
        "visualChannels": {
            "colorField": {"name": "avg_revenue", "type": "real"},
            "colorScale": "quantile"
        }
    }))

    # Options and config
    .with_options(center_map=True, read_only=False)
    .with_config(cull_unused_columns=False, overlay_blending="additive")
```

(continues on next page)

(continued from previous page)

```
)

g = g.encode_kepler(encoding)
```

Serialization

```
# Convert to dictionary for inspection or manual editing
encoding_dict = encoding.to_dict()

# Result structure:
# {
#     'datasets': [...],
#     'layers': [...],
#     'options': {...},
#     'config': {...}
# }
```

See Also

- *KeplerDataset*: Dataset configuration
- *KeplerLayer*: Layer configuration
- *Maps & Geographic Visualization*: User guide with examples
- *Plotter Methods*: Plotter methods (encode_kepler, encode_kepler_dataset, encode_kepler_layer)

Plotter Methods

Plotter Methods

The following methods are provided for applying Kepler.gl configurations to Graphistry graphs.

`graphistry.PlotterBase.PlotterBase.encode_kepler`(*self*, *kepler_encoding*)

Apply a complete Kepler.gl encoding to the plotter.

Accepts a *graphistry.kepler.KeplerEncoding* object or plain dictionary. Returns a new Plotter instance with the encoding applied (immutable pattern).

Parameters

kepler_encoding (*Union[Dict[str, Any],* KeplerEncoding]) – KeplerEncoding object or dict with structure: {'datasets': [...], 'layers': [...], 'options': {...}, 'config': {...}}

Returns

New Plotter instance with the Kepler encoding applied

Return type

Plottable

Example: Using KeplerEncoding container

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

kepler = (KeplerEncoding()
          .with_dataset(KeplerDataset(id="points", type="nodes"))
          .with_layer(KeplerLayer({
              "id": "point-layer",
              "type": "point",
              "config": {"dataId": "points", "columns": {"lat": "lat", "lng":
↪"lng"}}
          })))
g = g.encode_kepler(kepler)
```

Example: Using plain dict

```
kepler_dict = {
    'datasets': [{'info': {'id': 'points'}, 'type': 'nodes'}],
    'layers': [{'id': 'point-layer', 'type': 'point', 'config': {'dataId':
↪'points'}}],
    'options': {'centerMap': True},
    'config': {'cullUnusedColumns': True}
}
g = g.encode_kepler(kepler_dict)
```

See *graphistry.kepler.KeplerEncoding* for complete documentation.

graphistry.PlotterBase.PlotterBase.**encode_kepler_dataset**(*self*, **args*, ***kwargs*)

Add a Kepler.gl dataset to the encoding.

Accepts same parameters as *graphistry.kepler.KeplerDataset*. Returns a new Plotter instance with the dataset appended (immutable pattern).

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl dataset dictionary (if provided, all other params ignored)
- **id** (*Optional[str]*) – Dataset identifier (auto-generated if None)
- **type** (*Optional[str]*) – Dataset type - 'nodes', 'edges', 'countries', 'states', etc.
- **label** (*Optional[str]*) – Display label (defaults to id)
- **include** (*Optional[List[str]]*) – Columns to include (whitelist)
- **exclude** (*Optional[List[str]]*) – Columns to exclude (blacklist)
- **computed_columns** (*Optional[Dict[str, Any]]*) – Computed/aggregated columns for data enrichment
- **map_node_coords** (*Optional[bool]*) – Auto-map source/target node coordinates to edges (edges only)
- **map_node_coords_mapping** (*Optional[Dict[str, str]]*) – Custom column names for mapped coordinates (edges only)
- **resolution** (*Optional[Literal[10, 50, 110]]*) – Map resolution - 10 (high), 50 (medium), 110 (low) (geographic datasets only)
- **boundary_lakes** (*Optional[bool]*) – Include lake boundaries (geographic datasets only)

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

Returns
New Plotter instance with the dataset added

Return type
Plottable

Example: Node dataset

```
g = g.encode_kepler_dataset(id="companies", type="nodes", label="Companies")
```

Example: Edge dataset with coordinate mapping

```
g = g.encode_kepler_dataset(
    id="relationships",
    type="edges",
    map_node_coords=True
)
```

Example: Countries dataset

```
g = g.encode_kepler_dataset(
    id="countries",
    type="countries",
    resolution=50
)
```

See *graphistry.kepler.KeplerDataset* for complete parameter documentation.

graphistry.PlotterBase.PlotterBase.**encode_kepler_layer**(*self*, *raw_dict*)

Add a Kepler.gl layer to the encoding.

Accepts native Kepler.gl layer dictionary (same as *graphistry.kepler.KeplerLayer*). Returns a new Plotter instance with the layer appended (immutable pattern).

Parameters
raw_dict (*Dict[str, Any]*) – Native Kepler.gl layer dictionary with structure: {'id': ..., 'type': ..., 'config': {...}}

Returns
New Plotter instance with the layer added

Return type
Plottable

Example: Point layer

```
g = g.encode_kepler_layer({
    "id": "my-layer",
    "type": "point",
    "config": {
        "dataId": "my-dataset",
        "columns": {"lat": "latitude", "lng": "longitude"},
        "color": [255, 140, 0]
```

(continues on next page)

(continued from previous page)

```
    }
})
```

Example: Arc layer

```
g = g.encode_kepler_layer({
    "id": "connections",
    "type": "arc",
    "config": {
        "dataId": "edges",
        "columns": {
            "lat0": "edgeSourceLatitude",
            "lng0": "edgeSourceLongitude",
            "lat1": "edgeTargetLatitude",
            "lng1": "edgeTargetLongitude"
        }
    }
})
```

See *graphistry.kepler.KeplerLayer* and *Kepler.gl Layer Format* for layer format details.

graphistry.PlotterBase.PlotterBase.**encode_kepler_options**(*self*, **args*, ***kwargs*)

Apply Kepler.gl visualization options to the plotter.

Accepts same parameters as *graphistry.kepler.KeplerOptions*. Returns a new Plotter instance with the options applied (immutable pattern).

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl options dictionary (if provided, all other params ignored)
- **center_map** (*Optional[bool]*) – Auto-center map on data (default: True)
- **read_only** (*Optional[bool]*) – Disable map interactions (default: False)
- **args** (*Any*)
- **kwargs** (*Any*)

Returns

New Plotter instance with the options applied

Return type

Plottable

Example: Structured params

```
g = g.encode_kepler_options(center_map=True, read_only=False)
```

Example: Native format

```
g = g.encode_kepler_options({"centerMap": True, "readOnly": False})
```

See *graphistry.kepler.KeplerOptions* for complete parameter documentation.

graphistry.PlotterBase.PlotterBase.**encode_kepler_config**(*self*, **args*, ***kwargs*)

Apply Kepler.gl configuration settings to the plotter.

Accepts same parameters as *graphistry.kepler.KeplerConfig*. Returns a new Plotter instance with the config applied (immutable pattern).

Parameters

- **raw_dict** (*Optional[Dict[str, Any]]*) – Native Kepler.gl config dictionary (if provided, all other params ignored)
- **cull_unused_columns** (*Optional[bool]*) – Remove columns not used by layers (default: True)
- **overlay_blending** (*Optional[Literal['normal', 'additive', 'subtractive']]*) – Blend mode - 'normal', 'additive', 'subtractive' (default: 'normal')
- **tile_style** (*Optional[Dict[str, Any]]*) – Base map tile style configuration
- **args** (*Any*)
- **kwargs** (*Any*)

Returns

New Plotter instance with the config applied

Return type

Plottable

Example: Structured params

```
g = g.encode_kepler_config(
    cull_unused_columns=True,
    overlay_blending='additive'
)
```

Example: Native format

```
g = g.encode_kepler_config({
    "cullUnusedColumns": True,
    "overlayBlending": "additive"
})
```

See *graphistry.kepler.KeplerConfig* for complete parameter documentation.

See Also

- *KeplerDataset*: KeplerDataset configuration
- *KeplerLayer*: KeplerLayer configuration
- *KeplerOptions*: KeplerOptions configuration
- *KeplerConfig*: KeplerConfig configuration
- *KeplerEncoding*: KeplerEncoding configuration
- *Maps & Geographic Visualization*: User guide with examples

Native Formats

Kepler.gl Dataset Format

Native Kepler.gl dataset configuration format reference.

When using `KeplerDataset(raw_dict={...})`, the dictionary should follow the native Kepler.gl dataset structure documented here.

Dataset Structure

A Kepler.gl dataset configuration has the following structure:

```
{
    "version": "v1",
    "data": {
        "id": "dataset-id",          # Dataset identifier
        "label": "Dataset Label",    # Display label
        "color": [255, 0, 0],        # Optional RGB color
        "allData": [...],            # Array of data rows (optional)
        "fields": [...]              # Array of field definitions (optional)
    }
}
```

Fields Definition

Each field in the `fields` array describes a column:

```
{
    "name": "column_name",        # Column name
    "type": "string",             # Data type: string, integer, real, boolean, timestamp,
↪ geometry
    "format": "",                 # Optional format string
    "analyzerType": "STRING"      # Analyzer type: STRING, INT, FLOAT, BOOLEAN, DATE,␣
↪GEOMETRY
}
```

Common Field Types

- `string` / `STRING`: Text data
- `integer` / `INT`: Integer numbers
- `real` / `FLOAT`: Floating point numbers
- `boolean` / `BOOLEAN`: True/false values
- `timestamp` / `DATE`: Temporal data
- `geometry` / `GEOMETRY`: Spatial data (GeoJSON)

Examples

Basic Dataset

```
from graphistry import KeplerDataset

dataset = KeplerDataset({
    "version": "v1",
    "data": {
        "id": "cities",
        "label": "City Locations",
        "color": [255, 140, 0]
    }
})
```

Dataset with Fields

```
dataset = KeplerDataset({
    "version": "v1",
    "data": {
        "id": "points",
        "label": "Points of Interest",
        "fields": [
            {"name": "name", "type": "string", "analyzerType": "STRING"},
            {"name": "latitude", "type": "real", "analyzerType": "FLOAT"},
            {"name": "longitude", "type": "real", "analyzerType": "FLOAT"},
            {"name": "count", "type": "integer", "analyzerType": "INT"},
            {"name": "timestamp", "type": "timestamp", "analyzerType": "DATE"}
        ]
    }
})
```

See Also

- *KeplerDataset*: KeplerDataset class API
- *Kepler.gl Layer Format*: Kepler.gl layer format
- https://docs.kepler.gl/docs/api-reference/actions/actions#adddatatomap

Kepler.gl Layer Format

Native Kepler.gl layer configuration format reference.

When using `KeplerLayer(raw_dict={...})`, the dictionary should follow the native Kepler.gl layer structure documented here.

Layer Structure

A Kepler.gl layer configuration has the following structure:

```
{
    "id": "layer-id",               # Layer identifier
    "type": "point",                # Layer type (see below)
    "config": {
        "dataId": "dataset-id",     # Dataset this layer references
        "label": "Layer Label",     # Display label
        "color": [255, 0, 0],       # RGB color array
        "columns": {...},           # Column mappings (type-specific)
        "isVisible": True,          # Visibility toggle
        "visConfig": {...}          # Visual configuration (type-specific)
    },
    "visualChannels": {...}         # Optional: data-driven styling
}
```

Layer Types

Point Layer

Displays individual points on the map.

```
{
    "type": "point",
    "config": {
        "columns": {
            "lat": "latitude",      # Latitude column
            "lng": "longitude"      # Longitude column
        },
        "visConfig": {
            "radius": 10,           # Point radius
            "opacity": 0.8,         # Opacity (0-1)
            "outline": False,       # Show outline
            "thickness": 2          # Outline thickness
        }
    }
}
```

Arc Layer

Displays curved lines connecting two points (great circle arcs).

```
{
    "type": "arc",
    "config": {
        "columns": {
            "lat0": "src_lat",      # Source latitude
            "lng0": "src_lon",      # Source longitude
```

(continues on next page)

(continued from previous page)

```
            "lat1": "dst_lat",      # Target latitude
            "lng1": "dst_lon"       # Target longitude
        },
        "visConfig": {
            "opacity": 0.3,         # Opacity (0-1)
            "thickness": 2          # Arc thickness
        }
    }
}
```

Line Layer

Displays straight lines between points.

```
{
    "type": "line",
    "config": {
        "columns": {
            "lat0": "src_lat",
            "lng0": "src_lon",
            "lat1": "dst_lat",
            "lng1": "dst_lon"
        },
        "visConfig": {
            "opacity": 0.8,
            "thickness": 2
        }
    }
}
```

Hexagon Layer

Aggregates points into hexagonal bins.

```
{
    "type": "hexagon",
    "config": {
        "columns": {
            "lat": "latitude",
            "lng": "longitude"
        },
        "visConfig": {
            "opacity": 0.8,
            "worldUnitSize": 1,      # Hexagon size in km
            "resolution": 8,         # H3 resolution (0-15)
            "coverage": 1,           # Coverage ratio (0-1)
            "enable3d": True,        # Enable 3D extrusion
            "elevationScale": 5,     # Height multiplier
            "colorRange": {...}      # Color palette
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

Grid Layer

Aggregates points into square grid cells.

```
{
    "type": "grid",
    "config": {
        "columns": {
            "lat": "latitude",
            "lng": "longitude"
        },
        "visConfig": {
            "opacity": 0.8,
            "worldUnitSize": 1,      # Cell size in km
            "colorRange": {...},
            "coverage": 1,
            "enable3d": True,
            "elevationScale": 5
        }
    }
}
```

GeoJSON Layer

Displays polygon/multipolygon geometries.

```
{
    "type": "geojson",
    "config": {
        "columns": {
            "geojson": "geometry"    # GeoJSON geometry column
        },
        "visConfig": {
            "opacity": 0.8,
            "filled": True,          # Fill polygons
            "stroked": True,         # Show outline
            "thickness": 1,          # Outline thickness
            "strokeColor": [255, 255, 255],
            "colorRange": {...},
            "radius": 10,            # For point features
            "elevationScale": 1,     # Height multiplier
            "enable3d": False
        }
    }
}
```

Heatmap Layer

Displays data as a continuous heat distribution.

```
{
    "type": "heatmap",
    "config": {
        "columns": {
            "lat": "latitude",
            "lng": "longitude"
        },
        "visConfig": {
            "opacity": 0.8,
            "radius": 20,            # Influence radius in pixels
            "intensity": 1,          # Heat intensity
            "threshold": 0.05,       # Visibility threshold
            "colorRange": {...}
        }
    }
}
```

Cluster Layer

Clusters nearby points together.

```
{
    "type": "cluster",
    "config": {
        "columns": {
            "lat": "latitude",
            "lng": "longitude"
        },
        "visConfig": {
            "opacity": 0.8,
            "clusterRadius": 40,     # Cluster radius in pixels
            "colorRange": {...}
        }
    }
}
```

Icon Layer

Displays custom icons at point locations.

```
{
    "type": "icon",
    "config": {
        "columns": {
            "lat": "latitude",
            "lng": "longitude",
```

(continues on next page)

(continued from previous page)

```
            "icon": "icon_name"      # Optional icon name column
        },
        "visConfig": {
            "radius": 10,            # Icon size
            "opacity": 0.8
        }
    }
}
```

Trip Layer

Displays animated paths/trips over time.

```
{
    "type": "trip",
    "config": {
        "columns": {
            "geojson": "path"        # GeoJSON LineString
        },
        "visConfig": {
            "opacity": 0.8,
            "thickness": 2,
            "trailLength": 180       # Trail length in seconds
        }
    }
}
```

Visual Channels

Visual channels enable data-driven styling:

```
{
    "visualChannels": {
        "colorField": {
            "name": "column_name",
            "type": "real"
        },
        "colorScale": "quantile",    # quantile, quantize, ordinal
        "sizeField": {
            "name": "size_column",
            "type": "real"
        },
        "sizeScale": "linear"        # linear, sqrt, log
    }
}
```

Complete Example

```
from graphistry import KeplerLayer

layer = KeplerLayer({
    "id": "city-points",
    "type": "point",
    "config": {
        "dataId": "cities",
        "label": "City Locations",
        "color": [255, 140, 0],
        "columns": {
            "lat": "latitude",
            "lng": "longitude"
        },
        "isVisible": True,
        "visConfig": {
            "radius": 10,
            "opacity": 0.8,
            "outline": True,
            "thickness": 2
        }
    },
    "visualChannels": {
        "colorField": {
            "name": "population",
            "type": "real"
        },
        "colorScale": "quantile",
        "sizeField": {
            "name": "gdp",
            "type": "real"
        },
        "sizeScale": "sqrt"
    }
})
```

See Also

- *KeplerLayer*: KeplerLayer class API
- *Kepler.gl Dataset Format*: Kepler.gl dataset format
- https://docs.kepler.gl/docs/user-guides/b-kepler-gl-workflow/a-add-data-to-the-map
- https://docs.kepler.gl/docs/api-reference/layers

10.10.7.3 Overview

Geographic Bindings

Bind geographic coordinate columns to enable automatic Kepler rendering:

```
g = g.bind(
    point_longitude='longitude_column',
    point_latitude='latitude_column'
)
g.plot()  # Automatically renders as geographic visualization
```

Configuration Classes

- *KeplerDataset*: Configure datasets (nodes, edges, countries, states)
- *KeplerLayer*: Configure visualization layers (point, arc, hexagon, etc.)
- *KeplerOptions*: Configure visualization options (map centering, interactions)
- *KeplerConfig*: Configure map settings (blending, tile style)
- *KeplerEncoding*: Complete Kepler configuration container

Plotter Methods

- *encode_kepler()*: Apply complete Kepler configuration
- *encode_kepler_dataset()*: Add a dataset
- *encode_kepler_layer()*: Add a visualization layer
- *encode_kepler_options()*: Apply visualization options
- *encode_kepler_config()*: Apply map configuration

Layout Methods

- *mercator_layout()*: Convert lat/lon to Mercator projection (optional)

10.10.7.4 Basic Usage Example

```
import graphistry
import pandas as pd

# Prepare data
nodes_df = pd.DataFrame({
    'id': ['NYC', 'LA', 'Chicago'],
    'lat': [40.7128, 34.0522, 41.8781],
    'lon': [-74.0060, -118.2437, -87.6298]
})
```

(continues on next page)

(continued from previous page)

```
# Automatic geographic visualization
g = graphistry.nodes(nodes_df, 'id')
g = g.bind(point_longitude='lon', point_latitude='lat')
g.plot()
```

10.10.7.5 Advanced Configuration Example

```
from graphistry import KeplerEncoding, KeplerDataset, KeplerLayer

# Build custom configuration
encoding = (
    KeplerEncoding()
    .with_dataset(KeplerDataset(id="cities", type="nodes", label="Cities"))
    .with_dataset(KeplerDataset(id="routes", type="edges", label="Routes"))
    .with_layer(KeplerLayer(
        id="city-points",
        type="point",
        config={
            "dataId": "cities",
            "columns": {"lat": "latitude", "lng": "longitude"},
            "color": [255, 140, 0],
            "visConfig": {"radius": 10, "opacity": 0.8}
        }
    ))
    .with_layer(KeplerLayer(
        id="route-arcs",
        type="arc",
        config={
            "dataId": "routes",
            "columns": {
                "lat0": "edgeSourceLatitude",
                "lng0": "edgeSourceLongitude",
                "lat1": "edgeTargetLatitude",
                "lng1": "edgeTargetLongitude"
            },
            "color": [0, 200, 255],
            "visConfig": {"opacity": 0.3}
        }
    ))
    .with_options(centerMap=True)
)

# Apply to graph
g = g.encode_kepler(encoding)
g.plot()
```

10.10.7.6 See Also

- *Maps & Geographic Visualization*: Maps & geographic visualization guide
- *Mercator Layout*: Mercator projection layout
- *PyGraphistry Layout Catalog*: Layout catalog with all available layouts

10.10.8 Utilities

10.10.8.1 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*, *nodes_file_id=None*, *edges_file_id=None*, *metadata=None*, *certificate_validation=True*, *org_name=None*, *client_session=None*)

Bases: `object`

Parameters

- **server_base_path** (*str*)
- **view_base_path** (*str*)
- **name** (*str | None*)
- **description** (*str | None*)
- **edges** (*Table | None*)
- **nodes** (*Table | None*)
- **node_encodings** (*Dict[str, Any] | None*)
- **edge_encodings** (*Dict[str, Any] | None*)
- **token** (*str | None*)
- **dataset_id** (*str | None*)
- **nodes_file_id** (*str | None*)
- **edges_file_id** (*str | None*)
- **metadata** (*Dict[str, Any] | None*)
- **certificate_validation** (*bool*)
- **org_name** (*str | None*)
- **client_session** (*ClientSession | 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)
- session
- hard-coded

Parameters

- **mode** (*Literal['private', 'organization', 'public'] | None*)
- **notify** (*bool | None*)
- **invited_users** (*List[str] | None*)
- **mode_action** (*Literal['10', '20'] | None*)
- **message** (*str | None*)

`property certificate_validation`

`create_dataset`(*json*, *validate='autofix'*, *warn=True*)

Create dataset with optional encoding validation.

Args:

json: Dataset JSON payload validate: 'autofix' (continue), 'strict'/'strict-fast' (raise); True maps to 'strict', False maps to 'autofix' warn: If True and validate='autofix', emit warnings on validation errors. validate=False forces warn=False.

Parameters

- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)

`property dataset_id: str`

`property description: str`

`property edge_encodings`

`property edges: Table | None`

`property edges_file_id: str`

`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_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 nodes_file_id: str`

`property org_name: str | None`

`pkey_login(personal_key_id, personal_key_secret, org_name=None)`

Parameters

- **personal_key_id** (*str*)
- **personal_key_secret** (*str*)
- **org_name** (*str | None*)

Return type
ArrowUploader

`post(as_files=True, memoize=True, validate='autofix', warn=True, erase_files_on_fail=True)`

Upload data to Graphistry server.

Note: likely want to pair with self.maybe_post_share_link(g)

Args:
as_files: Upload as separate files (default True) memoize: Memoize conversions (default True) validate: 'autofix' (continue), 'strict'/'strict-fast' (raise); True maps to 'strict', False maps to 'autofix' warn: If True and validate='autofix', emit warnings on validation errors. validate=False forces warn=False. erase_files_on_fail: Clean up files on failure (default True)

Parameters

- **as_files** (*bool*)
- **memoize** (*bool*)
- **validate** (*Literal['strict', 'strict-fast', 'autofix'] | bool*)
- **warn** (*bool*)
- **erase_files_on_fail** (*bool*)

Return type
ArrowUploader

`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

Parameters

- **org_name** (*str | None*)
- **idp_name** (*str | None*)

Return type
ArrowUploader

property token: str

verify(*token=None*)

Return type
bool

property view_base_path: str

10.10.8.2 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 a new file (unless *file_id* supplied) and upload *arr*.

If *memoize* is True (default):

- Returns a cached *file_id* when there is a hash match for the table, in the file_id cache.

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`

10.10.8.3 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*)

10.10.8.4 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

DGL Utils

class graphistry.dgl_utils.**DGLGraphMixin**(**args*, ***kwargs*)

Bases: *FeatureMixin*

Automagic DGL models from Graphistry Instances.

build_gnn(*X_nodes=None*, *X_edges=None*, *y_nodes=None*, *y_edges=None*, *weight_column=None*, *reuse_if_existing=True*, *featurize_edges=True*, *use_node_scaler=None*, *use_node_scaler_target=None*, *use_edge_scaler=None*, *use_edge_scaler_target=None*, *train_split=0.8*, *device='cpu'*, *inplace=False*, **args*, ***kwargs*)

Builds GNN model using (DGL)[https://www.dgl.ai/]

Will auto-featurize, and if no explicit edges are found, automatically UMAP to produce implicit edges.

param X_nodes
Which node dataframe columns to featurize. If None, will use all columns. If passing in explicit dataframe, will set them as attributes.

param X_edges
Which edge dataframe columns to featurize. If None, will use all columns. If passing in explicit dataframe, will set them as attributes.

param y_nodes
Optional target column from nodes dataframe.

param y_edges
Optional target column from edges dataframe

param weight_column
Optional Weight column if explicit edges table exists with said weights. Otherwise, weight_column is inhereted by UMAP.

param train_split
Randomly assigns a train and test mask according to the split value, default 80%.

param use_node_scaler
selects which scaling to use on featurized nodes dataframe. Default None

param use_edge_scaler
selects which scaling to use on featurized edges dataframe. Default None

param device
device to run model, default *cpu*, with *gpu* the other choice. Can be handled in outer scope.

param inplace
default, False, whether to return Graphistry instance in place or not.

Parameters

- `X_nodes` (*List [str] / str / DataFrame / None*)
- `X_edges` (*List [str] / str / DataFrame / None*)
- `y_nodes` (*List [str] / str / DataFrame / None*)
- `y_edges` (*List [str] / str / DataFrame / None*)
- `weight_column` (*str / None*)
- `reuse_if_existing` (*bool*)
- `featurize_edges` (*bool*)
- `use_node_scaler` (*str / None*)
- `use_node_scaler_target` (*str / None*)
- `use_edge_scaler` (*str / None*)
- `use_edge_scaler_target` (*str / None*)
- `train_split` (*float*)
- `device` (*str*)

- **inplace** (*bool*)

convert_kwargs(**args*, ***kwargs*)

dgl_lazy_init(*train_split=0.8*, *device='cpu'*)

Initialize DGL graph lazily :return:

Parameters

- **train_split** (*float*)
- **device** (*str*)

graphistry.dgl_utils.**convert_to_torch**(*X_enc*, *y_enc*)

Convert X and y to torch tensors compatible with DGL ndata/edata.

Parameters

- **X_enc** (*DataFrame*) – DataFrame matrix of values for model matrix
- **y_enc** (*DataFrame | None*) – DataFrame matrix of values for target

Returns

Dictionary of torch-encoded arrays

graphistry.dgl_utils.**get_available_devices**()

Get IDs of all available GPUs.

Returns:

device (torch.device): Main device (GPU 0 or CPU). gpu_ids (list): List of IDs of all GPUs that are available.

graphistry.dgl_utils.**get_torch_train_test_mask**(*n*, *ratio=0.8*)

Generate random train/test torch boolean masks.

Parameters

- **n** (*int*) – Size of mask
- **ratio** (*float*) – Train/test split ratio (fraction of True entries)

Returns

Tuple of (train_mask, test_mask)

graphistry.dgl_utils.**pandas_to_dgl_graph**(*df*, *src*, *dst*, *weight_col=None*, *device='cpu'*)

Convert an edge DataFrame to a DGL graph plus adjacency matrix.

Example:

```
g, sp_mat, ordered_nodes_dict = pandas_to_sparse_adjacency(df, 'to_node', 'from_node
↪')
```

Parameters

- **df** (*DataFrame*) – DataFrame with source/destination (and optional weight) columns
- **src** (*str*) – Source column name for the COO matrix
- **dst** (*str*) – Destination column name for the COO matrix
- **weight_col** (*str | None*) – Optional weight column when constructing the COO matrix
- **device** (*str*) – Whether to put the DGL graph on CPU or GPU

Returns
Tuple of (DGL graph, sparse adjacency matrix, node index mapping)

Return type
Tuple[dgl.DGLGraph, scipy.sparse.coo_matrix, *Dict*]

`graphistry.dgl_utils.pandas_to_sparse_adjacency`(*df*, *src*, *dst*, *weight_col*)

Build a COO sparse adjacency matrix from an edge DataFrame.

Parameters

- **df** – Edge DataFrame
- **src** – Source column
- **dst** – Destination column
- **weight_col** – Optional weight column

Returns
Tuple of (COO sparse matrix, node index mapping)

`graphistry.dgl_utils.reindex_edgelist`(*df*, *src*, *dst*)

Relabel edges so DGL gets contiguous integer node IDs.

Example:

```
df, ordered_nodes_dict = reindex_edgelist(df, 'to_node', 'from_node')
```

Parameters

- **df** – Edge DataFrame
- **src** – Source column name
- **dst** – Destination column name

Returns
Tuple of (reindexed DataFrame, ordered node mapping)

Modules

Plugins

Plugin Types

Graphistry Validate Module

This module contains functions related to the validation of node and edge encodings.

10.10.9 Layouts

Native layout engines within Graphistry.

We recommend using the various plugins for additional layouts, such as for tree and hierarchical data diagramming.

10.10.9.1 Circle Layout

`graphistry.layout.circle.circle_layout`(*self*, *bounding_box=None*, *ring_spacing=None*, *point_spacing=None*, *partition_by=None*, *sort_by=None*, *ascending=True*, *na_position='last'*, *ignore_index=True*, *engine=EngineAbstract.AUTO*)

Arranges nodes in a circular layout

If partition_by and and bounding_box df are provided, do as multiple circles

Each circle is sorted, by default by degree

The ring radius is set to circumscribe the bounding box of the nodes

Parameters

param self
Plottable

type self
Plottable

param bounding_box
The bounding box for the circular layout, in the format (cx, cy, width, height), or a partition-keyed dataframe of the same. If not provided, the bounding box is determined based on the nodes' positions.

type bounding_box
Optional[Tuple[float, float, float, float] | df[[partition_key, cx, cy, w, h]]]

param ring_spacing
The spacing between successive rings. Defaults to 1.0 if not provided.

type ring_spacing
Optional[float]

param point_spacing
The distance between nodes within a ring, along the circumference. Defaults to ring_spacing * 0.1 if not provided.

type point_spacing
Optional[float]

param partition_by
Column name or list of column names to sort nodes by. Defaults to None, in which case no sorting is applied.

type partition_by
Optional[Union[str, List[str]]]

param sort_by
Column name or list of column names to sort nodes by. Defaults to None, in which case sorting is by degree, in-degree, outdegree.

type sort_by
Optional[Union[str, List[str]]]

param ascending
Whether to sort ascending or descending.

type ascending
Union[bool, List[bool]]

param na_position
Where to position NaNs in the sorting order. Defaults to 'last'.

type na_position
str

param ignore_index
Whether to ignore the index when sorting. Defaults to True.

type ignore_index
bool

param engine
The engine to use for computations (either 'pandas' or 'cudf'). Defaults to EngineAbstract.AUTO.

type engine
Union[EngineAbstract, str]

Returns

Plottable
A graph object with nodes arranged in a circular layout.

Parameters

- `self` (Plottable)
- `bounding_box` (*Tuple[float, float, float, float] | Any | None*)
- `ring_spacing` (*float | None*)
- `point_spacing` (*float | None*)
- `partition_by` (*str | List[str] | None*)
- `sort_by` (*str | List[str] | None*)
- `ascending` (*bool | List[bool]*)
- `na_position` (*str*)
- `ignore_index` (*bool*)
- `engine` (*EngineAbstract | str*)

Return type
Plottable

graphistry.layout.circle.**print_gpu_memory_usage**(*prefix=''*)

10.10.9.2 ForceAtlas2 Layout

graphistry.layout.fa2.**compute_bounding_boxes**(*self*, *partition_key*, *engine*)

Returns the bounding boxes for each partition based on the center coordinates and dimensions of the nodes. DF keys: - cx: Center x-coordinate - cy: Center y-coordinate - w: Width - h: Height - partition_key: Partition key

Parameters

- **self** (Plottable)
- **partition_key** (*str*)
- **engine** (*Engine*)

Return type
Any

graphistry.layout.fa2.**fa2_layout**(*self*, *fa2_params=None*, *circle_layout_params=None*, *singleton_layout=None*, *partition_key=None*, *engine=EngineAbstract.AUTO*, *allow_cpu_fallback=False*)

Applies FA2 layout for connected nodes and circle layout for singleton (edgeless) nodes

Allows optional parameterization of the circle layout, e.g., sort keys

Parameters

- **g** (graphistry.Plottable.Plottable) – The graph object with nodes and edges, in a format compatible with Graphistry's Plottable object.
- **fa2_params** (*Optional[Dict[str, Any]]*) – Optional parameters for customizing the Force-Atlas 2 (FA2) layout, passed through to *fa2_layout*.
- **circle_layout_params** (*Optional[Dict[str, Any]]*) – Optional parameters for customizing the circle layout, passed through to *general_circle_layout*. Can include: - *by*: Column name(s) for sorting nodes (default: 'degree'). - *ascending*: Boolean(s) to control sorting order. - *ring_spacing*: Spacing between rings in the circle layout. - *point_spacing*: Spacing between points in each ring.
- **singleton_layout** (*Optional[Callable[[Plottable, Tuple[float, float, float, float] | Any], Plottable]]*) – Optional custom layout function for singleton nodes (default: circle_layout).
- **partition_key** (*Optional[str]*) – The key for partitioning nodes (used for picking bounding box type). Default is None.
- **allow_cpu_fallback** (*bool*) – When True, allow a CPU (igraph Fruchterman-Reingold) fallback instead of raising. Only intended for internal callers that explicitly expect the approximation.
- **self** (Plottable)
- **engine** (*EngineAbstract | str*)

Returns
A graph object with FA2 and circle layouts applied.

Return type
graphistry.Plottable.Plottable

10.10.9.3 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.

This layout algorithm organizes nodes into rectangular bounding boxes based on a partitioning algorithm. It supports various layout algorithms within each partition and optional color encoding based on the partition.

Supports passing in a custom per-partition layout algorithm handler.

Parameters

- **partition_alg** (*Optional[str]*) – (optional) The algorithm to use for partitioning the graph nodes. Examples include 'community' or 'louvain'.
- **partition_params** (*Optional[Dict[str, Any]]*) – (optional) Parameters for the partition algorithm, passed as a dictionary.
- **layout_alg** (*Optional[Union[str, Callable[[*Plottable*], Plottable]]]*) – (optional) The layout algorithm to arrange nodes within each partition.
  - In GPU mode, defaults to `graphistry.layout.fa2.fa2_layout()` for individual partitions.
  - CPU mode defaults to `graphistry.plugins.igraph.layout_igraph()` with layout *"fr"*.
  - Can be a string referring to an igraph algorithm (CPU), cugraph algorithm (GPU), or a callable function.
- **layout_params** (*Optional[Dict[str, Any]]*) – (optional) Parameters for the layout algorithm.
- **x** (*float*) – (optional) The x-coordinate for the top-left corner of the layout. Default is 0.
- **y** (*float*) – (optional) The y-coordinate for the top-left corner of the layout. Default is 0.
- **w** (*Optional[float]*) – (optional) The width of the layout. If None, it will be automatically determined based on the number of partitions.
- **h** (*Optional[float]*) – (optional) The height of the layout. If None, it will be automatically determined based on the number of partitions.
- **encode_colors** (*bool*) – (optional) Whether to apply color encoding to nodes based on partitions. Default is True.
- **colors** (*Optional[List[str]]*) – (optional) List of colors to use for the partitions. If None, default colors will be applied.
- **partition_key** (*Optional[str]*) – (optional) The key for partitioning nodes. If not provided, defaults to a relevant partitioning key for the algorithm.
- **engine** (*Union[graphistry.Engine.EngineAbstract, Literal["auto"]]*) – (optional) The execution engine for the layout, either "auto" (default), "cpu", or "gpu".

- **self** (Plottable)

Returns
A graph object with nodes arranged in a group-in-a-box layout.

Return type
graphistry.Plottable.Plottable

Example 1: Basic GPU Group-in-a-Box Layout Using ECG Community Detection

```
g_final = g.group_in_a_box_layout(partition_alg='ecg')
```

Example 2: Group-in-a-Box on a precomputed partition key

```
g_partitioned = g.compute_cugraph('ecg')
g_final = g_partitioned.group_in_a_box_layout(partition_key='ecg')
```

Example 3: Custom Group-in-a-Box Layout with FA2 for Layout and Color Encoding

```
g_final = g.group_in_a_box_layout(
    partition_alg='louvain',
    partition_params={'resolution': 1.0},
    layout_alg=lambda g: fa2_with_circle_singletons(g),
    encode_colors=True,
    colors=['#ff0000', '#00ff00', '#0000ff']
)
```

Example 4: Advanced Usage with Custom Bounding Box and GPU Execution

```
g_final = g.group_in_a_box_layout(
    partition_alg='louvain',
    layout_alg='force_atlas2',
    x=100, y=100, w=500, h=500,  # Custom bounding box
    engine='gpu'  # Use GPU for faster layout
)
```

graphistry.layout.gib.gib.**resolve_partition_key**(*g*, *partition_key=None*, *partition_alg=None*)

Parameters
partition_alg (*str | None*)

10.10.9.4 Mercator Layout

Geographic layout using Mercator projection for latitude/longitude coordinates.

> **Note**
>
> This layout is **optional** for Graphistry visualization. Graphistry automatically performs server-side geographic layout when latitude/longitude bindings are detected. Use `mercator_layout()` only when you need projected coordinates locally for analysis or exporting to other tools.

Mercator layout: Convert latitude/longitude coordinates to Mercator projection.

This module provides the mercator_layout method for Plotter objects.

graphistry.layout.mercator.mercator_layout(*self*, *scale_for_graphistry=True*)

Convert latitude/longitude coordinates to Mercator projection.

Uses point_latitude and point_longitude bindings if set, otherwise defaults to 'latitude' and 'longitude' columns. Uses existing point_x and point_y bindings if available, otherwise defaults to 'x' and 'y'. Uses GPU acceleration via CuPy if available, falls back to CPU/pandas otherwise.

Note: Graphistry automatically performs server-side geographic layout when latitude/longitude columns are detected. Use mercator_layout() when you need projected coordinates locally for analysis or need to export coordinates for use with other tools.

Parameters

- **scale_for_graphistry** (*bool*) – If True (default), use scaled Earth radius (~637) for manageable coordinate values in Graphistry visualizations. If False, use standard Earth radius (~6,378,137 meters) for accurate geographic coordinates.
- **self** (Plottable)

Returns

Plottable with Mercator projection coordinates

Return type

Plottable

Example

```
import graphistry
import pandas as pd

# Using default column names 'latitude' and 'longitude'
nodes_df = pd.DataFrame({
    'id': ['NYC', 'LA', 'London'],
    'latitude': [40.7128, 34.0522, 51.5074],
    'longitude': [-74.0060, -118.2437, -0.1278]
})
g = graphistry.nodes(nodes_df, 'id').mercator_layout()

# Or with custom column names
nodes_df2 = pd.DataFrame({
    'id': ['NYC', 'LA', 'London'],
    'lat': [40.7128, 34.0522, 51.5074],
    'lon': [-74.0060, -118.2437, -0.1278]
})
g2 = (graphistry
      .nodes(nodes_df2, 'id')
      .bind(point_latitude='lat', point_longitude='lon')
      .mercator_layout())
```

See Also

- *Maps & Geographic Visualization*: Geographic visualization guide
- *Kepler API Reference*: Kepler.gl integration API

10.10.9.5 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()
```

10.10.9.6 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
EngineAbstractType, 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 | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

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
EngineAbstractType, 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 | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

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.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*, *format_labels=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[Union[str, numpy.datetime64]] First ring and axis label

Time_end
Optional[Union[str, 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

Format_labels
Optional[Callable[[numpy.datetime64, int, numpy.timedelta64], str]] Deprecated alias for `format_label`

Play_ms
int initial layout time in milliseconds, default 2000

Engine
EngineAbstractType, 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** (*str | datetime64 | None*)
- **time_end** (*str | 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*)
- **format_labels** (*Callable[[datetime64, int, timedelta64], str] | None*)
- **play_ms** (*int*)
- **engine** (*EngineAbstract | Literal['pandas', 'cudf', 'dask', 'dask_cudf', 'auto']*)

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.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

10.10.9.7 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

10.10.9.8 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

10.10.9.9 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

10.10.9.10 LayoutsMixin

Base class for protocol classes.

Protocol classes are defined as:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

Such classes are primarily used with static type checkers that recognize structural subtyping (static duck-typing).

For example:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()
```

(continues on next page)

(continued from previous page)

```
func(C())  # Passes static type check
```

See PEP 544 for details. Protocol classes decorated with @typing.runtime_checkable act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures. Protocol classes can be generic, they are defined as:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

10.10.10 Plugins

10.10.10.1 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.**compute_cugraph_core**(*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*)

Take input cugraph.Graph object and load in data and bindings (source, destination, edge_weight)

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')
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.**layout_cugraph_core**(*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')
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']

Gephi (GEXF)

GEXF (Graph Exchange XML Format) is commonly used by Gephi and other tools for graph interchange. Graphistry supports GEXF 1.1draft, 1.2draft, and 1.3 for import/export with no extra dependencies.

Use `graphistry.gexf()` (or *graphistry.PlotterBase.PlotterBase.from_gexf()*) to load a GEXF file, URL, or stream into a PyGraphistry plotter. By default, any available GEXF viz fields are bound; if the file has no viz data, Graphistry defaults apply.

When to use

- You have a Gephi (or other tool) export and want to preserve its layout/viz.
- You need a lightweight interchange format for graph attributes or layouts.

Example

```
import graphistry

# Preserve GEXF's layout, size, colors, shape icons
g = graphistry.gexf("my_graph.gexf")
g.plot()

# Keep layout while dropping GEXF colors/sizes
g_layout_only = graphistry.gexf(
```

(continues on next page)

(continued from previous page)

```
    "my_graph.gexf",
    bind_node_viz=["position"],
    bind_edge_viz=[],
)
g_layout_only.plot()
```

For notebook walkthroughs (small + medium GEXF) and dataset samples, see *Plugins - Compute & Layout*.

Export

You can export a graph to GEXF using `graphistry.to_gexf()` or *graphistry.PlotterBase.PlotterBase.to_gexf()*:

```
xml_str = g.to_gexf()
g.to_gexf("out.gexf")
```

Viz attribute mapping

The loader maps standard GEXF viz attributes into Graphistry bindings:

- `label` (node) → `point_title`
- `label` (edge) → `edge_title`
- `viz:color` → `point_color` / `edge_color` (hex color strings)
- `viz:size` → `point_size`
- `viz:position` → `point_x` / `point_y` (and auto-sets `play=0`)
- `viz:thickness` → `edge_size`
- `viz:color` alpha → `point_opacity` / `edge_opacity`
- `viz:shape` (nodes) → `point_icon` (FA4 icon names without the `fa-` prefix: disc→circle, square→square, triangle→caret-up, diamond→diamond; image uses the `uri` when available)
- edge `weight` → `edge_weight`

Viz binding controls

Use `bind_node_viz` / `bind_edge_viz` as allowlists to restrict which GEXF viz fields are bound to Graphistry encodings (unlisted fields still load as columns):

- node fields: `color`, `size`, `opacity`, `position`, `icon`
- edge fields: `color`, `size`, `opacity`

`None` (default) binds all supported fields present in the file. Passing an empty list disables all viz bindings for that element type. `position` binds `point_x`/`point_y` and sets `play=0` to respect precomputed layouts.

Examples

```
# Bind all available GEXF viz fields (default)
g = graphistry.gexf("my_graph.gexf")

# Bind no GEXF viz fields (use Graphistry defaults)
g = graphistry.gexf("my_graph.gexf", bind_node_viz=[], bind_edge_viz=[])

# Bind only layout positions for nodes, drop edge viz
g = graphistry.gexf("my_graph.gexf", bind_node_viz=["position"], bind_edge_viz=[])
```

After loading, you can apply Graphistry's declarative encodings (for example, `encode_point_color` or `encode_point_size`) to override GEXF defaults.

Validation

The loader raises `ValueError` for common errors such as missing nodes, missing node IDs, or edges that reference unknown nodes. These checks run inside the GEXF loader (XML well-formedness + basic structural checks), not PyGraphistry's broader graph validation or full GEXF schema validation. For untrusted inputs, install `defusedxml`; it will be used automatically for safer XML parsing. Use `parse_engine="stdlib"` or `parse_engine="defused"` to override the parser (useful in tests). GEXF import/export helpers.

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.

For static outputs in notebooks or docs, you can either call `graphistry.Plottable.plot_static()` (preferred, auto-reuses x/y when present) or `graphistry.plugins.graphviz.render_graphviz()` for lower-level control.

Auto-display: When called in a Jupyter notebook, `plot_static` automatically displays the rendered output inline. It returns an SVG or Image object (use `.data` for raw bytes).

```
# Simplest form - auto-displays in notebook
g.plot_static()

# Save to file while also displaying
svg_obj = g.plot_static(path='graph.svg')  # saves file AND returns SVG object
```

`plot_static` engines:

- `graphviz-svg` / `graphviz-png` (default render image, optional path)
- `graphviz` (render to any Graphviz format, e.g., pdf)
- `graphviz-dot` (DOT text, optional path)
- `mermaid-code` (Mermaid DSL text, optional path)

Styling: Use `graph_attr`, `node_attr`, and `edge_attr` dictionaries:

```
g.plot_static(
    graph_attr={'rankdir': 'LR', 'bgcolor': 'white'},
    node_attr={'style': 'filled', 'fillcolor': 'lightblue'},
    edge_attr={'color': 'gray'}
)
```

`plot_static()` works with any layout source—it will use existing x/y positions if available (`reuse_layout=True`), or compute layout via graphviz if not.

See the *static rendering tutorial* for complete examples.

`graphistry.plugins.graphviz.g_to_pgv`(*g*, *directed=True*, *strict=False*, *drop_unsanitary=False*, *include_positions=False*)

Parameters

- **g** (Plottable)

- **directed** (*bool*)
- **strict** (*bool*)
- **drop_unsanitary** (*bool*)
- **include_positions** (*bool*)

Return type
AGraph

graphistry.plugins.graphviz.**g_with_pgv_layout**(*g*, *graph*)

Parameters

- **g** (Plottable)
- **graph** (*AGraph*)

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

Graphviz is a CPU-only library. cuDF DataFrames are automatically converted to pandas before calling graphviz, and the result is converted back.

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, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Graphviz graph attributes, see https://graphviz.org/docs/graph/
- **node_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.NodeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Graphviz node attributes, see https://graphviz.org/docs/nodes/
- **edge_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.EdgeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – 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',
```

(continues on next page)

(continued from previous page)

```
    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*, *include_positions=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'], str | int | float | bool] | 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'], str | int | float | bool] | 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'], str | int | float | bool] | None*)
- **drop_unsanitary** (*bool*)
- **include_positions** (*bool*)

Return type
AGraph

graphistry.plugins.graphviz.**pgv_styling**(*g*)

Parameters
g (Plottable)

Return type
Plottable

graphistry.plugins.graphviz.**render_graphviz**(*self*, *prog='dot'*, *format='svg'*, *args=None*, *directed=True*, *strict=False*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *drop_unsanitary=False*, *max_nodes=None*, *max_edges=None*, *path=None*, *include_positions=False*)

Render a graph to an image via graphviz and return the rendered bytes.

This wraps `layout_graphviz_core()` to compute positions, then draws with pygraphviz. Optionally enforces caps to keep renders small/deterministic for docs/examples.

When `include_positions` is True and the plot has bound x/y values, the existing layout is preserved rather than recomputed by Graphviz.

Parameters

- **self** (Plottable) – Base graph
- **prog** (graphistry.plugins_types.graphviz_types.Prog) – Layout algorithm
- **format** (graphistry.plugins_types.graphviz_types.Format) – Render format
- **directed** (*bool*) – Whether the graph is directed
- **strict** (*bool*) – Whether to treat the graph as strict
- **graph_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.GraphAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Graph-level attributes
- **node_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.NodeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Node-level attributes
- **edge_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.EdgeAttr, graphistry.plugins_types.graphviz_types.GraphvizAttrValue]]) – Edge-level attributes
- **drop_unsanitary** (*bool*) – Reject unsanitary attrs
- **max_nodes** (*Optional[int]*) – Optional cap on nodes for rendering
- **max_edges** (*Optional[int]*) – Optional cap on edges for rendering
- **path** (*Optional[str]*) – Optional path to also write the render
- **args** (*str | None*)
- **include_positions** (*bool*)

Returns
Rendered bytes (SVG/PNG/etc.)

Return type
bytes

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.**GraphvizAttrValue**
alias of `str` | `int` | `float` | `bool`

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

igraph is a CPU-only library. cuDF DataFrames are automatically converted to pandas before calling igraph, and the result is converted back. For a GPU-native alternative, see `compute_cugraph()`.

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.**ensure_pandas**(*df*)

Convert to pandas if not already (e.g. cuDF). No-op for pandas.

Delegates to `graphistry.compute.engine_coercion.ensure_pandas()`. Defined here to avoid a circular import at module load time.

Parameters
`df` (*Any*)

Return type
DataFrame

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.

igraph is a CPU-only library. cuDF DataFrames are automatically converted to pandas before calling igraph, and the result is converted back. For a GPU-native alternative, see `layout_cugraph()`.

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.**restore_engine**(*g*, *original_nodes*, *original_edges*)

Convert result DataFrames back to the original engine if needed.

Delegates to `graphistry.compute.engine_coercion.restore_engine()`. Defined here to avoid a circular import at module load time.

Parameters

- **g** (Plottable)
- **original_nodes** (*Any*)
- **original_edges** (*Any*)

Return type
Plottable

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.

igraph is a CPU-only library. cuDF DataFrames are automatically converted to pandas before being passed to igraph. For a GPU-native alternative, see `to_cugraph()`.

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)

Return type
Any

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

The following methods are provided for converting and managing NetworkX graph data within PyGraphistry. `g.compute_networkx(...)` exposes the same curated NetworkX algorithm subset as GFQL local Cypher `CALL graphistry.nx.*`: `pagerank`, `betweenness_centrality`, `degree_centrality`, `closeness_centrality`, `eigenvector_centrality`, `katz_centrality`, `connected_components`, `strongly_connected_components`, `core_number`, `hits`, `edge_betweenness_centrality`, and `k_core`.

Install with `pygraphistry[networkx]` or `pygraphistry[networkx-scipy]`. For an executable notebook walkthrough, see *NetworkX*.

Node algorithm example:

```
g2 = g.compute_networkx("degree_centrality", out_col="degree_score")
assert "degree_score" in g2._nodes.columns
```

Edge algorithm example:

```
g2 = g.compute_networkx("edge_betweenness_centrality", out_col="edge_bc", directed=False)
assert "edge_bc" in g2._edges.columns
```

Graph-returning algorithm example:

```
g2 = g.compute_networkx("k_core", params={"k": 2}, directed=False)
```

Result shape:

- Node algorithms append one or more columns to `g._nodes`.
- Edge algorithms append one column to `g._edges`.
- `k_core` returns a projected PyGraphistry graph.
- `hits` writes `hubs` and `authorities` and does not accept `out_col`.
- `connected_components` uses weak components when `directed=True` and connected components when `directed=False`.

graphistry.PlotterBase.PlotterBase.**compute_networkx**(*self*, *alg*, *out_col=None*, *params=None*, *directed=True*, *G=None*)

Run a supported NetworkX algorithm and return an enriched PyGraphistry graph.

Parameters

- **alg** (*str*) – Explicitly supported NetworkX algorithm name.
- **out_col** (*Optional[str]*) – Optional output column name for single-column node/edge algorithms.
- **params** (*Optional[Mapping[str, Any]]*) – Keyword parameters forwarded to the selected NetworkX algorithm.
- **directed** (*bool*) – Whether to build a directed NetworkX graph from the current graph.

- **G** (*Optional [Any]*) – Optional prebuilt NetworkX graph to run instead of converting `self`.
- **self** (Plottable)

Returns
Plotter

Return type
Plotter

Example: Degree centrality

```
g2 = g.compute_networkx("degree_centrality", out_col="degree_score")
```

Example: HITS

```
g2 = g.compute_networkx("hits")
assert "hubs" in g2._nodes.columns
assert "authorities" in g2._nodes.columns
```

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]
])
```

(continues on next page)

(continued from previous page)

```
# 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*)

Parameters
G (*Any*)

Return type
Tuple[*DataFrame*, *DataFrame*]

graphistry.PlotterBase.PlotterBase.**networkx_checkoverlap**(*self*, *g*)

Raise an error if the node attribute already exists in the graph

Parameters
g (*Any*)

Return type
None

10.10.10.2 Data Providers

PyGraphistry supports a variety of data providers natively. Many systems not listed here are likely also supported through PyGraphistry's native support for pydata ecosystem accelerated glue layers such as pandas, Apache Arrow, and cudf.

Azure Cosmos DB for Apache Gremlin

Azure Cosmos DB supports Gremlin graph queries

class graphistry.gremlin.**CosmosMixin**(**a*, ***kw*)

Bases: *GremlinMixin*

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: *Plottable*

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.

Return type
Plottable

`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 return the result as a PyGraphistry graph object.

This method allows you to execute Gremlin queries, either as a single string or an iterable of strings, and retrieve the results in a format that PyGraphistry can process and visualize. To support databases like CosmosDB, the queries are sent as strings.

Parameters
queries (*Union[str, Iterable[str]]*) – One or more Gremlin queries to execute. Can be a single query (as a string) or multiple queries (as a list of strings).

Returns
A PyGraphistry *Plottable* graph object containing the query results.

Return type
Plottable

Example: Execute a Gremlin Query and Plot

```
import graphistry
from gremlin_python.driver.client import Client
from gremlin_python.driver.serializer import GraphSONSerializersV2d0

# Create a Gremlin client for CosmosDB
my_gremlin_client = Client(
    f'wss://MY_ACCOUNT.gremlin.cosmosdb.azure.com:443/',
    'g',
    username=f"/dbs/MY_DB/colls/{self.COSMOS_CONTAINER}",
    password='MY_COSMOS_PRIMARY_KEY',
    message_serializer=GraphSONSerializersV2d0()
)

# Run a Gremlin query and visualize the result
```

(continues on next page)

(continued from previous page)

```
graphistry                      .gremlin_client(my_gremlin_client)
↪           .gremlin('g.V().hasLabel("person").limit(5)')
↪   .fetch_nodes()                      .plot()
```

In this example, the Gremlin query selects the first 5 vertices with the label "person" and plots the result in PyGraphistry.

Example: Running Multiple Gremlin Queries

```
queries = [
    'g.V().hasLabel("person").limit(5)',
    'g.E().limit(10)'
]

graphistry                      .gremlin_client(my_gremlin_client)
↪           .gremlin(queries)                    .fetch_nodes()
↪          .plot()
```

This example demonstrates how to run multiple Gremlin queries, fetch their results, and visualize them.

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
```

(continues on next page)

(continued from previous page)

```
)

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

Kusto Plugin

class graphistry.plugins.kusto.KustoMixin(**args*, ***kwargs*)

Bases: *Plottable*

KustoMixin is a Graphistry Mixin that allows you to plot data from Kusto.

configure_kusto(*cluster*, *database='NetDefaultDB'*, *client_id=None*, *client_secret=None*, *tenant_id=None*)

Configure Azure Data Explorer (Kusto) connection settings.

Sets up the connection parameters for accessing a Kusto cluster. Authentication can be done via service principal (client_id, client_secret, tenant_id) or managed identity (omit authentication parameters).

Parameters

- **cluster** (*str*) – Kusto cluster URL (e.g., 'https://mycluster.westus2.kusto.windows.net')
- **database** (*str*) – Database name (defaults to 'NetDefaultDB')
- **client_id** (*Optional[str]*) – Azure AD application (client) ID for service principal auth
- **client_secret** (*Optional[str]*) – Azure AD application secret for service principal auth
- **tenant_id** (*Optional[str]*) – Azure AD tenant ID for service principal auth

Returns
Self for method chaining

Return type
Plottable

Example: Service principal authentication

```
import graphistry
g = graphistry.configure_kusto(
    cluster="https://mycluster.westus2.kusto.windows.net",
    database="SecurityDatabase",
    client_id="your-client-id",
    client_secret="your-client-secret",
    tenant_id="your-tenant-id"
)
```

Example: Managed identity authentication

```
import graphistry
g = graphistry.configure_kusto(
    cluster="https://mycluster.westus2.kusto.windows.net",
    database="SecurityDatabase"
    # No auth params - uses managed identity
)
```

`kql`(*query: str*, ***, *unwrap_nested: bool | None = None*, *single_table: Literal[True] = True*) → List[DataFrame]

`kql`(*query: str*, ***, *unwrap_nested: bool | None = None*, *single_table: Literal[False]*) → DataFrame

`kql`(*query: str*, ***, *unwrap_nested: bool | None = None*, *single_table: bool = True*) → DataFrame | List[DataFrame]

Execute KQL query and return result tables as DataFrames.

Submits a Kusto Query Language (KQL) query to Azure Data Explorer and returns the results. By default, expects a single table result and returns it as a DataFrame. If multiple tables are returned, only the first is returned with a warning. Set single_table=False to always get a list of all result tables.

Parameters

- **query** (*str*) – KQL query string to execute
- **unwrap_nested** (*Optional[bool]*) – Strategy for handling nested/dynamic columns
- **single_table** (*bool*) – If True, return single DataFrame (first table if multiple); if False, return list

Returns
Single DataFrame if single_table=True, else list of DataFrames

Return type
Union[pd.DataFrame, List[pd.DataFrame]]

unwrap_nested semantics:

- **True**: Always attempt to unwrap nested columns; raise on failure
- **None**: Use heuristic - unwrap if the first result looks nested
- **False**: Never attempt to unwrap nested columns

Example: Basic security query (single table mode)

```
import graphistry
g = graphistry.configure_kusto(...)

query = '''
SecurityEvent
| where TimeGenerated > ago(1d)
| where EventID == 4624  // Successful logon
| project TimeGenerated, Account, Computer, IpAddress
| take 1000
'''

# Single table mode returns DataFrame directly (default)
df = g.kql(query)
print(f"Found {len(df)} logon events")
```

Example: Get all tables as list

```
# Always get a list of all tables
dfs = g.kql(query, single_table=False)
df = dfs[0]
```

Example: Multi-table query

```
query = '''
SecurityEvent | take 10;
Heartbeat | take 5
'''

# With single_table=True (default), returns first table with warning
df = g.kql(query)  # Returns SecurityEvent data, warns about multiple tables

# With single_table=False, returns all tables
frames = g.kql(query, single_table=False)
security_df = frames[0]
heartbeat_df = frames[1]
```

property kusto_client: Any

kusto_close()

Close the active Kusto client connection.

Properly closes the underlying Kusto client connection to free resources. This should be called when you're done using the Kusto connection.

Example

```
import graphistry
g = graphistry.configure_kusto(...)
# ... perform queries ...
g.kusto_close()  # Clean up connection
```

Return type
None

kusto_from_client(*client*, *database='NetDefaultDB'*)

Configure Kusto using an existing client connection.

Use this method when you already have a configured Kusto client connection and want to reuse it with Graphistry.

Parameters

- **client** (*azure.kusto.data.KustoClient*) – Pre-configured Kusto client
- **database** (*str*) – Database name to query against

Returns

Self for method chaining

Return type

Plottable

Example

```
from azure.kusto.data import KustoClient, KustoConnectionStringBuilder
import graphistry

# Create Kusto client
kcsb = KustoConnectionStringBuilder.with_aad_device_authentication(
    "https://mycluster.kusto.windows.net"
)
kusto_client = KustoClient(kcsb)

# Use with Graphistry
g = graphistry.kusto_from_client(kusto_client, "MyDatabase")
```

kusto_graph(*graph_name*, *snap_name=None*)

Fetch a Kusto graph entity as a Graphistry visualization object.

Retrieves a named graph entity (and optional snapshot) from Kusto using the graph() operator and graph-to-table transformation. The result is automatically bound as nodes and edges for visualization.

Parameters

- **graph_name** (*str*) – Name of the Kusto graph entity to fetch
- **snap_name** (*Optional[str]*) – Optional snapshot/version identifier

Returns

Plottable object ready for visualization or further transforms

Return type

Plottable

Example: Basic graph visualization

```
import graphistry
g = graphistry.configure_kusto(...)

# Fetch and visualize a named graph
graph_viz = g.kusto_graph("NetworkTopology")
graph_viz.plot()
```

Example: Specific snapshot

```
# Fetch a specific snapshot of the graph
graph_viz = g.kusto_graph("NetworkTopology", "2023-12-01")
graph_viz.plot()
```

kusto_health_check()

Perform a health check on the Kusto connection.

Executes a simple query (.show tables) to verify that the connection to the Kusto cluster is working properly.

Raises
RuntimeError – If the connection test fails

Return type
None

Example

```
import graphistry
g = graphistry.configure_kusto(...)
g.kusto_health_check()  # Verify connection works
```

graphistry.plugins.kusto.**init_kusto_client**(*cfg*)

Parameters
cfg (*KustoConfig*)

Return type
Any

Amazon Neptune

Amazon Neptune is a managed graph database by Amazon. It supports OpenCypher, RDF, Gremlin, and various analytical capabilities.

class graphistry.gremlin.**NeptuneMixin**(**a*, ***kw*)

Bases: *GremlinMixin*

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-
```

(continues on next page)

(continued from previous page)

```
↪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*)

Spanner Plugin

class graphistry.plugins.spanner.SpannerMixin(*args, **kwargs)

Bases: *Plottable*

SpannerMixin is a Graphistry Mixin that allows you to plot data from Spanner.

static add_type_from_label_to_df(*df*)

Add `type` column from `label` for Graphistry type handling.

Creates a `type` column from the `label` column for proper visualization in Graphistry. If a `type` column already exists, it is renamed to `type_` before creating the new `type` column.

Parameters
`df` (*pd.DataFrame*) – DataFrame containing node or edge data with `label` column

Returns
Modified DataFrame with the updated `type` column

Return type
pd.DataFrame

configure_spanner(*instance_id*, *database_id*, *project_id=None*, *credentials_file=None*)

Configure Google Cloud Spanner connection settings.

Sets up the connection parameters for accessing a Spanner database instance. Either project_id or credentials_file must be provided for authentication.

Parameters

- `instance_id` (*str*) – The Spanner instance identifier
- `database_id` (*str*) – The Spanner database identifier
- `project_id` (*Optional [str]*) – Google Cloud project ID (optional if using credentials_file)
- `credentials_file` (*Optional [str]*) – Path to service account credentials JSON file

Returns
Self for method chaining

Return type
Plottable

Raises
`ValueError` – If neither credentials_file nor project_id is provided

Example: Using project ID

```
import graphistry
g = graphistry.configure_spanner(
    project_id="my-project",
    instance_id="my-instance",
    database_id="my-database"
)
```

Example: Using service account credentials

```
import graphistry
g = graphistry.configure_spanner(
    instance_id="my-instance",
    database_id="my-database",
    credentials_file="/path/to/credentials.json"
)
```

static convert_spanner_json(*data*)

Convert Spanner JSON query results to structured graph data.

Transforms raw Spanner JSON query results into a standardized format with separate nodes and edges arrays for graph processing.

Parameters
data (*List[Any]*) – Raw JSON data from Spanner query results

Returns
Structured graph data with 'nodes' and 'edges' arrays

Return type
List[Dict[str, Any]]

static get_edges_df(*json_data*)

Convert Spanner JSON edges into a pandas DataFrame.

Extracts edge data from structured JSON results and creates a DataFrame with columns for label, identifier, source, destination, and all edge properties.

Parameters
json_data (*list*) – Structured JSON data containing graph edges

Returns
DataFrame containing edge data with properties as columns

Return type
pd.DataFrame

static get_nodes_df(*json_data*)

Convert Spanner JSON nodes into a pandas DataFrame.

Extracts node data from structured JSON results and creates a DataFrame with columns for label, identifier, and all node properties.

Parameters
json_data (*list*) – Structured JSON data containing graph nodes

Returns
DataFrame containing node data with properties as columns

Return type
pd.DataFrame

property spanner_client: Any

spanner_close()

Close the active Spanner database connection.

Properly closes the underlying Spanner client connection to free resources. This should be called when you're done using the Spanner connection.

Example

```
import graphistry
g = graphistry.configure_spanner(...)
# ... perform queries ...
g.spanner_close()  # Clean up connection
```

Return type
None

property spanner_config: SpannerConfig

spanner_from_client(*client*)

Configure Spanner using an existing client connection.

Use this method when you already have a configured Spanner client connection and want to reuse it with Graphistry.

Parameters
client (*google.cloud.spanner_dbapi.connection.Connection*) – Pre-configured Spanner database connection

Returns
Self for method chaining

Return type
Plottable

Example

```
from google.cloud import spanner
import graphistry

# Create Spanner client
spanner_client = spanner.Client(project="my-project")
instance = spanner_client.instance("my-instance")
database = instance.database("my-database")

# Use with Graphistry
g = graphistry.spanner_from_client(database)
```

spanner_gql(*query*)

Execute GQL path query and return graph visualization.

Executes a Graph Query Language (GQL) path query on the configured Spanner database and returns a Plottable object ready for visualization. The query must return path data using SAFE_TO_JSON(p) format.

Parameters
query (*str*) – GQL path query string with SAFE_TO_JSON(path) format

Returns
Plottable object with nodes and edges populated from query results

Return type
Plottable

Example: Basic path query

```
import graphistry
graphistry.configure_spanner(
    project_id="my-project",
    instance_id="my-instance",
    database_id="my-database"
)

query = '''
GRAPH FinGraph
MATCH p = (a:Account)-[t:Transfers]->(b:Account)
LIMIT 10000
RETURN SAFE_TO_JSON(p) as path
'''

g = graphistry.spanner_gql(query)
g.plot()
```

spanner_gql_to_df(*query*)

Execute GQL/SQL query and return results as DataFrame.

Executes a Graph Query Language (GQL) or SQL query on the configured Spanner database and returns the results as a pandas DataFrame. This method is suitable for tabular queries that don't require graph visualization.

Parameters
query (*str*) – GQL or SQL query string

Returns
DataFrame containing query results with column names

Return type
pd.DataFrame

Example: Aggregation query

```
import graphistry
graphistry.configure_spanner(
    project_id="my-project",
    instance_id="my-instance",
    database_id="my-database"
)

query = '''
GRAPH FinGraph
MATCH (p:Person)-[:Owns]-(:Account)->(l:Loan)
RETURN p.id as PersonID, p.name AS Name,
       SUM(l.loan_amount) AS TotalBorrowed
ORDER BY TotalBorrowed DESC
LIMIT 10
'''

df = graphistry.spanner_gql_to_df(query)
print(df.head())
```

Example: SQL query

```
query = "SELECT * FROM Account WHERE type = 'checking' LIMIT 1000"
df = graphistry.spanner_gql_to_df(query)
```

graphistry.plugins.spanner.**init_spanner_client**(*cfg*)

Lazily establish a DB-API connection using the parameters in *session.config*.

Parameters
cfg (*SpannerConfig*)

Return type
Any

10.10.11 Schema Artifacts

PyGraphistry ships structural JSON Schema artifacts for downstream tooling and LLM contract generation:

- `schemas/encodings.schema.json`
- `schemas/react-settings.schema.json`
- `schemas/url-params.schema.json`

Generate or check them with:

```
python -m graphistry.devschemas.export
python -m graphistry.devschemas.export --check
```

These schemas are structural contracts. Runtime semantic validation remains owned by the existing PyGraphistry validators.

10.11 Join the Community

The Graphistry team is active in a few places, so come join us:

- https://www.graphistry.com/blog
- https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g
- https://github.com/graphistry/pygraphistry
- https://twitter.com/graphistry
- https://www.linkedin.com/company/graphistry

10.12 Support

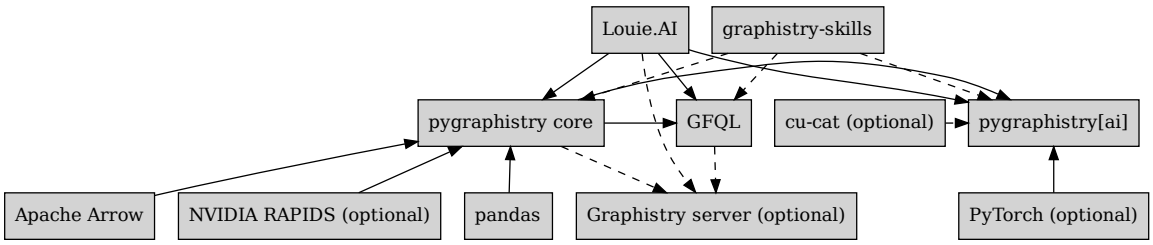
Stuck or thinking about a new project? Let's chat!

- https://www.graphistry.com/get-started
- https://www.graphistry.com/blog
- https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g

- https://graphistry.zendesk.com/hc/en-us
- https://github.com/graphistry/pygraphistry
- https://twitter.com/graphistry
- https://www.linkedin.com/company/graphistry

10.13 Graphistry Ecosystem and Louie.AI

The Graphistry community of projects, open source, and partners has grown over the years:



Legend: Solid arrows show "can drive/use" relationships, not dataflow. Dashed arrows show optional server usage. PyGraphistry, GFQL, and Louie.AI can run on a Graphistry server to generate visualizations for embedding.

10.13.1 Graphistry Core

- https://hub.graphistry.com/docs/api/
- https://github.com/graphistry/graphistry-js: Node, React, and vanilla JS
- https://github.com/graphistry/graph-app-kit: Python dashboarding with Graphistry and Streamlit

10.13.2 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.

10.13.3 Generative AI: Louie.AI and graphistry-skills

Louie.AI

https://www.louie.ai/ is the genAI-native experience for Graphistry and your favorite databases:

- 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 https://www.louie.ai/ for more information and early access.

graphistry-skills

https://github.com/graphistry/graphistry-skills provides skills for LLM coding assistants (Claude Code, Cursor, Codex) to generate better PyGraphistry code (~90% vs ~50% success rate):

```
npx skills add graphistry/graphistry-skills
```

Skills provide context-aware guidance for graph ETL, visualization, GFQL queries, and AI workflows.

10.13.4 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 https://github.com/graphistry/cu-cat

10.13.5 Community

Graphistry works with a variety of partners and projects, some of which include:

GPU dataframes:

- https://rapids.ai/: cuDF, cuGraph, cuML
- https://arrow.apache.org/: Python, JS, and more

Graph

- https://neo4j.com/
- https://www.trovares.com/
- https://www.tigergraph.com/
- https://www.arangodb.com/
- https://janusgraph.org/
- https://memgraph.com/

Log databases and SIEMs:

- https://www.elastic.co/
- https://docs.microsoft.com/en-us/azure/data-explorer/
- https://github.com/microsoft/msticpy
- https://opensearch.org/
- https://www.splunk.com/

Graph analytics:

- https://igraph.org/python/

Python data science ecosystem:

- https://streamlit.io/

- https://jupyter.org/
- https://pandas.pydata.org/
- https://www.dask.org/

10.14 Architecture

This document should help get started with modifying code. See also *develop.md* for developer commands and *CONTRIBUTING.md* for community guidelines.

10.14.1 Client/Server Wrangling Tool

PyGraphistry is Python client library primarily for:

- Loading in PyData, or converting non-PyData into PyData
- Setting declarative bindings for graph shaping and visual encodings
- Turning that into a live, embeddable Graphistry visualization

It is also increasingly used for intermediate compute and deeper access to Graphistry APIs. However, these should not get in the way of the primary use case.

It is currently heavily using Pandas, and slowly porting to Arrow-based and RAPIDS.ai-based internals.

10.14.2 Functional

Most user interaction is functional, where every `.bind()` will create a new Graphistry object that is a clone of the one being chained. Thus most calls have a cheap `.copy()` over shallow immutable bindings.

Account-related settings are generally global with local functional overrides.

10.14.3 DetaFrames: Lazy vs. Eager

At the `plot()` call, lazily bound data is materialized into the format needed for upload processing, e.g., API=3 will require pandas dataframes to be transformed into Arrow.

Database connectors should convert to Pandas, or better, Arrow, upon load. This enables immediate analysis using `._nodes` and `._edges`.

10.14.4 Plugins

New code is increasingly put into separate files:

- Graphistry APIs: Arrow conveniences classes to match Graphistry server APIs. Each entity type generally has different types. These are being written to allow standalone use.
- Per-Connector: These should be separate files and standalone. For convenience, their `connect()` and `query()` methods can be added to the global namespace. Ex: `g.bolt(...).cypher(...).plot()`.
- Methods are intended for Notebook-based inspection. For example, calling `g.cypher` should guide users with the available arguments and examples of use.

10.15 Contribute

10.15.1 Code of conduct

We follow https://www.apache.org/foundation/policies/conduct.html. Be open, empathetic, collaborative, inquisitive, carefully-worded, and concise. We do not support harrassment, non-welcoming behavior, and other behaviors detrimental to our community.

10.15.2 GitHub preferred

Developer communications should primarily live in GitHub issues and PRs, as this best helps with asynchronous communications and future reference

10.15.3 Chat!

When in doubt, https://join.slack.com/t/graphistry-community/shared_invite/zt-53ik36w2-fpP0Ibjbk7IJuVFIRSnr6g

10.15.4 Report bugs and propose features

There are more ways to contribute than code. Filing bugs, including for usability, or coming up with and voting on big feature win ideas, are great as well!

Please use GitHub issues to report and discuss topics and discuss them. Search for open/closed ones first.

When filing a bug, pleasse provide a fully reproducible code snippet. We should be able to copy-paste it into a Jupyter notebook and reproduce the problem.

10.15.5 Improve docs and share examples

Docs can always be better:

- Feel free to expand on the core docs
- Styling can always use care
- Examples of use cases, integrations, and techniques can always help others

10.15.6 PRs welcome

We are happy to accept PRs!

- Check for `good first issue` and `help wanted` tags for ideas on where to start
- If you have something specific you'd like to add, we are happy to provide guidance via GitHub on where to start and what to add to land it
- Data integrations, convenience methods, fixes, and more are all welcome
- When in doubt, ask on an Issue / PR / Slack!

10.15.6.1 Git conventions

Commits should be atomic. Every commit – or squashed PR – should be a self-contained addition/removal so we can cherrypick them as needed.

For example, if you fix some bug, refactor some code, and update dependencies while there... split that into three commits: `fix()`, `refactor()`, and `garden()`.

We use https://www.conventionalcommits.org/en/v1.0.0/.

Messages should look like:

```
fix(some feature name): verb action taken
```

The commit types are `fix()`, `feat()`, `infra()`, `garden()` / `refactor()`, `docs()`, `security()`.

Descriptive PRs and CHANGELOG.md

- Every PR should detail the additions/removals/fixes and breaking changes
- When adding features, try to add examples in the PR
- Manually update CHANGELOG.md as part of the PR

Automation

- PRs must pass CI, including style checks
- Maintainers are responsible for publishing

10.16 Development Setup

See also *CONTRIBUTING.md* and *ARCHITECTURE.md*

Development is setup for local native and containerized Python coding & testing, and with automatic GitHub Actions for CI + CD. The server tests are like the local ones, except against a wider test matrix of environments.

10.16.1 LFS

We are starting to use git lfs for data:

```
# install git lfs: os-specific commands below
git lfs install
git lfs checkout
```

10.16.1.1 git lfs: ubuntu

```
curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo␣
↪bash
sudo apt-get install git-lfs
```

10.16.2 Docker

10.16.2.1 Install

```
cd docker && docker compose build && docker compose up -d
```

For just CPU tests, you can focus on `test-cpu` and use the run instructions below:

```
cd docker && docker compose build test-cpu
```

10.16.2.2 Run local tests without rebuild

Containerized call to `pytest` for CPU + GPU modes:

```
cd docker

# cpu - pandas
./test-cpu-local.sh

# cpu - fast & targeted
WITH_LINT=0 WITH_TYPECHECK=0 WITH_BUILD=0 ./test-cpu-local.sh graphistry/tests/test_
↪hyper_dask.py::TestHypergraphPandas::test_hyper_to_pa_mixed2

# gpu - pandas, cudf, dask, dask_cudf; test only one file
./test-gpu-local.sh graphistry/tests/test_hyper_dask.py
```

Connector tests (currently neo4j-only): `cd docker && WITH_NEO4J=1 ./test-cpu-local.sh` (optional `WITH_SUDO=" "`)

- Will start a local neo4j (docker) then enable+run tests against it

Remote Graphistry integration tests are opt-in because they require a live server and credentials:

```
TEST_REMOTE_INTEGRATION=1 \
GRAPHISTRY_API_TOKEN=<jwt> \
python -m pytest graphistry/tests/compute/test_chain_let_remote_integration.py
```

Use `GRAPHISTRY_USERNAME`/`GRAPHISTRY_PASSWORD` instead of `GRAPHISTRY_API_TOKEN` when token auth is not available. For service-account style authentication in application code, prefer `personal_key_id` + `personal_key_secret`. Optional env vars: `GRAPHISTRY_SERVER` and `GRAPHISTRY_TEST_DATASET_ID`.

10.16.3 Docs

Automatically build via ReadTheDocs from inline definitions.

To manually build, see `docs/`.

10.16.4 Ignore files

You may need to add ignore rules:

- ruff: pyproject.toml (or bin/lint.sh)
- mypi: mypi.ini
- sphinx: docs/source/conf.py

10.16.5 Remote

Some databases like Neptune can be easier via cloud editing, especially within Jupyter:

```
git clone https://github.com/graphistry/pygraphistry.git
git checkout origin/my_branch
pip install --user -e .
git diff
```

and

```
import logging
logging.basicConfig(level=logging.DEBUG)

import graphistry
graphistry.__version__
```

10.16.6 CI

GitHub Actions: See `.github/workflows`

CI runs on every PR and updates them

10.16.6.1 Cypher Surface Growth Guard

CI includes `cypher-frontend-surface-guard`, which enforces bounded growth for:

- `graphistry/compute/gfql/cypher/lowering.py` total line count
- `CompiledCypherQuery`, `CompiledGraphBinding`, `CompiledCypherGraphQuery` dataclass field/property counts

Guard implementation + baseline:

- Script: `bin/ci_cypher_surface_guard.py`
- Baseline: `bin/ci_cypher_surface_guard_baseline.json`

If growth is intentional, regenerate baseline in your branch and include explicit PR rationale:

```
python bin/ci_cypher_surface_guard.py --write-baseline
```

Then commit both code changes and baseline update together.

10.16.6.2 GPU CI

GPU CI can be manually triggered by core dev team members:

1. Push intended changes to protected branches `gpu-public` or `master`
2. Manually trigger action https://github.com/graphistry/pygraphistry/actions/workflows/ci-gpu.yml on one of the above branches

GPU tests can also be run locally via `./docker/test-gpu-local.sh` .

10.16.7 Debugging Tips

- Use the unit tests
- use the `logging` module per-file

10.16.8 Publish: Merge, Tag, & Upload

1. Update CHANGELOG.md in your PR branch
   - Convert `## [Development]` section to `## [X.Y.Z - YYYY-MM-DD]`
   - Document all changes following https://keepachangelog.com/ format
   - Commit and push to PR branch
2. Merge the PR to master (via GitHub UI or `gh pr merge`)
3. Switch to master and pull the merged changes

```
git checkout master
git pull --ff-only origin master
git status --short  # should be empty before tagging
```

4. Tag the repository with the new version number (semantic versioning *X.Y.Z*)

```
git tag X.Y.Z
git push origin refs/tags/X.Y.Z
```

5. Confirm the https://github.com/graphistry/pygraphistry/actions?query=workflow%3A%22Publish+Python+%F0%9F%
   Github Action published to https://pypi.org/project/graphistry/
   - Auto-triggers on tag push
   - **Expected gate**: on tag-triggered releases, the final `Publish distribution to PyPI` job can pause in `waiting` until a maintainer approves `Review deployments` for environment `pypi-release`.
   - If the run is waiting, open the run page and approve `Review deployments`, then wait for the PyPI job to complete.
   - If manually triggering (`workflow_dispatch`), choose `release_mode`:
     - `evidence`: build + SBOM + provenance + evidence artifacts only (no publish)
     - `test`: includes TestPyPI publish, skips PyPI (uses synthetic runner-local version `0.0.dev<run_id>` to avoid local-version upload rejection)
     - `release`: TestPyPI + PyPI publish (restricted to `master`, with `pypi-release` approval)
   - Do not rerun publish for a version that is already on PyPI (duplicate-file uploads are rejected)

- Verify version appears on PyPI: `curl -s https://pypi.org/pypi/graphistry/json | jq -r '.info.version'`
- Verify release evidence artifacts from the workflow run:
  - built distributions (`dist/*.whl`, `dist/*.tar.gz`)
  - SBOM (`evidence/sbom-cyclonedx.json`)
  - GitHub build provenance attestation for built distributions (`dist/*.whl`, `dist/*.tar.gz`)
- Keep the PyPI Trusted Publisher binding aligned with this workflow:
  - repository: `graphistry/pygraphistry`
  - workflow file: `.github/workflows/publish-pypi.yml`
  - environment: `pypi-release`
  - refs: tag pushes and `workflow_dispatch` on `master` only
- This workflow publishes with attestations enabled for both TestPyPI and PyPI.

6. Toggle version as active at https://readthedocs.org/projects/pygraphistry/versions/
7. Create GitHub Release with detailed release notes

```
gh release create X.Y.Z --title "vX.Y.Z - Brief Title" --notes "Release notes in␣
↪markdown..."
```

Or create via GitHub UI: https://github.com/graphistry/pygraphistry/releases/new?tag=X.Y.Z

Release notes should include:

- Critical fixes and breaking changes (if any)
- Major features from current and recent versions
- Links to full CHANGELOG and installation instructions
- Highlight important API changes, new capabilities, and use cases

10.16.9 CI Dependency Lockfiles

CI uses per-Python-version hashed lockfiles for supply chain security:

- **Generation**: A `generate-lockfiles` CI job runs `bin/generate-lockfiles.sh` to produce lockfiles for all profile × Python version combos. Most are uploaded as artifacts, not committed.
- **ReadTheDocs lockfile**: `requirements/rtd-py3.12.lock` is committed because `.readthedocs.yml` consumes it directly. Update it when changing RTD's Python version, docs/pygraphviz extras, `setup.py` dependency constraints that affect docs, or RTD install steps:

```
PROFILES=rtd VERSIONS=3.12 ./bin/generate-lockfiles.sh
```

CI's `check-rtd-lockfile` job regenerates only the RTD profile using the committed lockfile's `--exclude-newer` timestamp and fails if `requirements/rtd-py3.12.lock` is out of sync. To fix a red `check-rtd-lockfile`, rerun the command above and commit the resulting lockfile.

- **Spark lockfile**: `requirements/spark-py3.14.lock` is committed because the `test-spark` job installs a small Spark-specific smoke-test environment without the broader test extras. Update `requirements/spark-py3.14.in` when changing the direct Spark smoke dependencies, then regenerate and commit the lockfile:

```
PROFILES=spark VERSIONS=3.14 ./bin/generate-lockfiles.sh
```

CI's `check-spark-lockfile` job uses the committed lockfile's `--exclude-newer` timestamp and fails if `requirements/spark-py3.14.lock` is out of sync.

- **6-day cooldown**: `--exclude-newer` ensures no package published in the last 6 days is included, mitigating 0-day supply chain attacks. `UV_EXCLUDE_NEWER` is also set globally as belt-and-suspenders.
- **Hash verification**: `--require-hashes` on install ensures tamper-proof installs (except AI/umap profiles where torch conflicts prevent it).
- **Adding a dependency**: After modifying most `setup.py` extras, CI automatically regenerates artifact lockfiles. If the change affects ReadTheDocs docs dependencies, also update and commit `requirements/rtd-py3.12.lock`.
- **Emergency override**: Set `COOLDOWN_DAYS=0` in `bin/generate-lockfiles.sh` to disable the 6-day cooldown for urgent patches.
- genindex
- modindex
- search