Oracle® AI Database

Property Graph Visualization Developer's Guide and Reference

26.2
G53290-01
April 2026

Oracle AI Database Property Graph Visualization Developer's Guide and Reference, 26.2

G53290-01

Primary Author: Lavanya Jayapalan

Contributors: Melliyal Annamalai, Korbinian Schmid, Diego Ramirez, Jorge Barba

Contents

List of Figures

Preface

This documentation provides usage and reference information for the graph visualization library used in property graph visualization.

Audience

This document is intended for graph developers to build applications using Oracle Graph Visualization library. It is also applicable for graph users who visualize and analyze property graphs in applications that use the Graph Visualization library.

Related Documents

For more information, see these following document:

- *Oracle AI Database Graph Developer's Guide for Property Graph*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for This Guide

The following changes apply to the graph visualization library that is shipped with Oracle Graph Server and Client.

New Features in the Graph Visualization Library 26.2

- Added conditional search functionality in the graph visualization user interface.
  See savedSearches, savedSearchProvider, SavedSearchAction, and Saved Conditional Searches for more information.
- Added a setting to disable auto-generated legend entries.
  See `settings.defaultLegendEnabled` in settings for more information.
- Added ability to group edges based on conditions.
  See `ruleBasedGroups` in settings for more information.
- Added a toggle to show or hide reachability paths for the current visualization.
  See `reachabilityPaths` in featureFlags for more information.

Breaking Changes

`showLegend` is desupported. Instead, use `legendState`.

- Deprecated Features
- Desupported Features

Deprecated Features

The following section lists the deprecated features in the Graph Visualization library:

- Deprecated settings features:
  - The following `settings` properties are deprecated:
    - `smartGroups` is deprecated. Instead, use `ruleBasedGroups`.
    - `smartExpands` is deprecated. Instead, use `ruleBasedExpands`.
    - `selectedSmartGroup` is deprecated. Instead, use `selectedRuleBasedGroup`.
    - `selectedSmartExpand` is deprecated. Instead, use `selectedRuleBasedExpand`.
  - The `SmartGroup` and `SmartExpand` interfaces are deprecated. Instead, use `RuleBasedGroup` and `RuleBasedExpand` respectively.
  - The `SmartExplorer` and `SmartExplorerType` interfaces are deprecated. Instead, use `RuleBasedExplorer` and `RuleBasedExplorerType` respectively.
  - The `searchValueChanged` callback is deprecated. Instead, use the conditional search feature in the user interface.
  - The `label` property within `Style` is deprecated. Instead, use `caption` property.

Desupported Features

The following section lists the desupported features in the Graph Visualization library:

- Desupported settings features:
  - `showLegend` is desupported. Instead, use `legendState`.
  - `filters` is desupported. Instead, use `settings.ruleBasedStyles`.
  - `FilterCondition` is desupported. Instead, use `BasicCondition`.
  - `Filter` interface is desupported. Instead, use `RuleBasedStyleSetting`.
- `styles` property is desupported. Instead, use `settings.baseStyles`.
- `updateFilter` is desupported. Instead, use updateRuleBasedStyle.
- `*` operator desupported in `FilterOperator` type.

1

Introduction to Visualization in Oracle Graph

Oracle Graph enables you to visually explore, interact, and analyze property graphs using the graph visualization library.

- About Oracle Graph Visualization Library
  You can build your own custom property graph visualizations in your applications using the Graph Visualization library.
- Getting Started with Oracle Graph Visualization Library
  Oracle Graph Visualization library is released quarterly with Oracle Graph Server and Client Releases.

1.1 About Oracle Graph Visualization Library

You can build your own custom property graph visualizations in your applications using the Graph Visualization library.

The library is built using JavaScript and the Graph Visualization component (`@gvt/graphviz`) in the library supports:

- Custom vertex or edge styling based on its properties
- Interactive actions for graph exploration
- Tooltip with vertex and edge details
- Automatic legend
- Multiple graph layouts
- Icons libraries
- Schema View

The Graph Visualization library is used in the following software components:

- The Graph Explorer, included with Oracle Graph Server and Client releases.
- The APEX Graph Visualization plug-in, available in both on-premises and Cloud environments.
- The Graph Studio Application, which is supported on Oracle Autonomous AI Database Serverless.

1.2 Getting Started with Oracle Graph Visualization Library

Oracle Graph Visualization library is released quarterly with Oracle Graph Server and Client Releases.

Perform the following steps to get started with the Graph Visualization library.

1. Sign in to Oracle Software Delivery Cloud.
2. Enter *Oracle Graph Server and Client* in the search bar and select the required release.
3. Download the `V1048066-01` component which contains the Graph Visualization library.

4. Embed the downloaded `oracle-graph-visualization-library-25.1.0.zip` in your web application.

   See the demo application on GitHub for an example.

Related Topics

- Oracle Graph Server and Client Releases Documentation

2

Interactive Graph Visualization Features

Oracle Graph allows you to explore and interact with your graph data when visualizing property graphs.

The following describes a few selected features:

- Layouts
  The Graph Visualization library supports several graph layouts. Each layout has its own algorithm, which computes the placements of the vertices and edges, affecting the visual structure of the graph.
- Exploration Modes
  The Graph Visualization library supports three different modes for graph exploration.
- Graph Interaction Options
  The Graph Visualization library supports different types of graph interactions.

2.1 Layouts

The Graph Visualization library supports several graph layouts. Each layout has its own algorithm, which computes the placements of the vertices and edges, affecting the visual structure of the graph.

You can configure these layouts through the settings option as shown:

```
Settings:
{
  ...
  layout: <'circle', 'concentric', 'force', 'grid', 'hierarchical', 'preset',
'radial', 'random', 'geographical'>
  ...
}
```

In addition, you can create custom layouts by passing the layout specific options using the settings format as shown:

```
Settings:
{
  ...
  layout: {
    type: 'grid',
    spacing: 5
  }
  ...
}
```

The following describes the supported layouts in graph visualization.

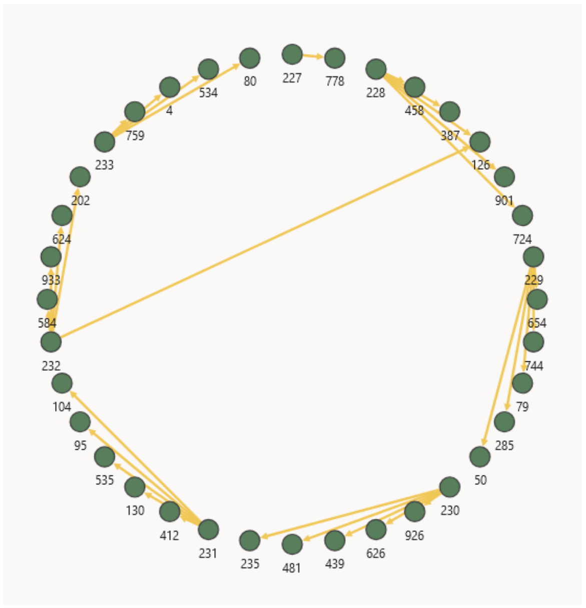
- Circle Layout
  The circle layout positions the graph vertices in a circle.

- Concentric Layout
  The concentric layout positions the graph vertices in concentric circles.
- Force Layout
  The force layout aims to create a visually appealing graph. It positions the graph vertices in the viewport so that all the edges are approximately equal in length and minimizes crossings between the edges.
- Geographical Layout
  The geographical layout allows you to overlay the graph on a map.
- Grid Layout
  The grid layout positions the graph vertices in a well-spaced grid.
- Hierarchical Layout
  The hierarchical layout organizes the graph using Directed Acyclic Graph (DAG) system. It is especially suitable for DAGs and trees.
- Radial Layout
  The radial layout displays the dependency chain of a graph by using an outwards expanding tree structure. It can be especially useful if the graph data has a hierarchical structure and contains many children for each parent vertex.
- Random Layout
  The random layout puts the graph vertices in random positions within the viewport.

2.1.1 Circle Layout

The circle layout positions the graph vertices in a circle.

Figure 2-1 Circle Layout

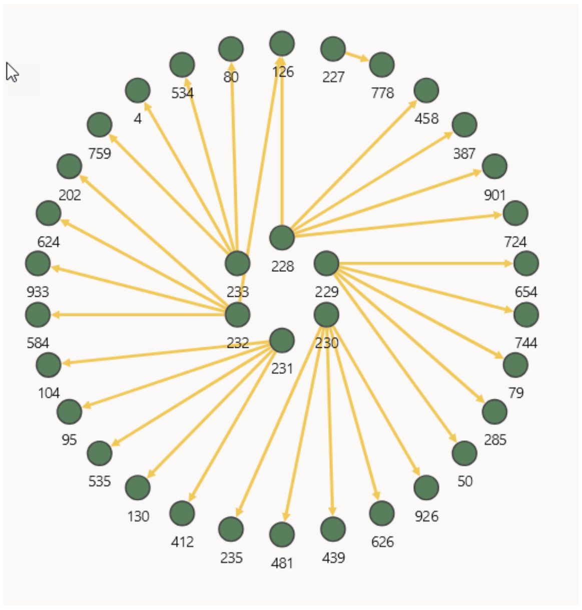


You can configure the `spacing` property to set the the radius of the circle.

2.1.2 Concentric Layout

The concentric layout positions the graph vertices in concentric circles.

Figure 2-2 Concentric Layout



You can configure the `spacing` property to set the minimum spacing between the vertices. It is basically used for adjusting the radius of the concentric circles.

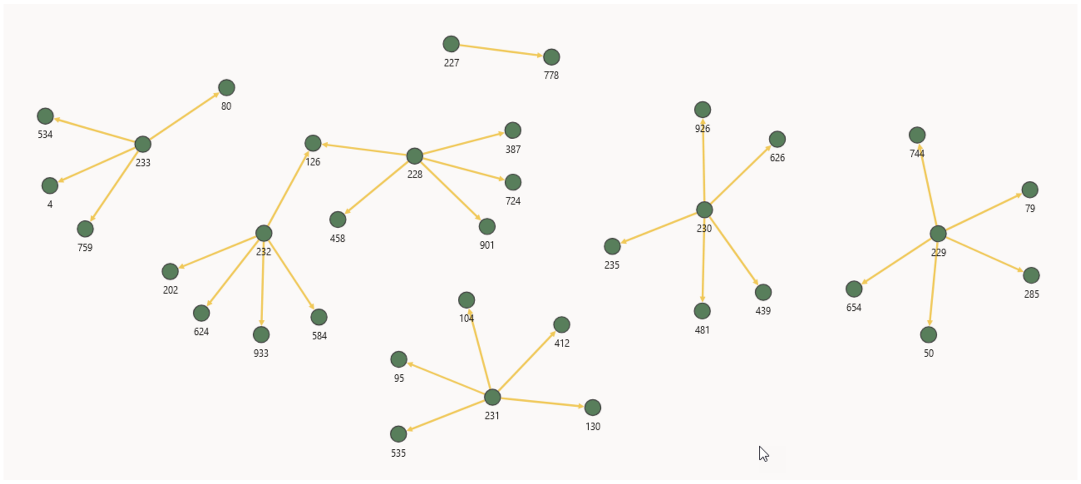
2.1.3 Force Layout

The force layout aims to create a visually appealing graph. It positions the graph vertices in the viewport so that all the edges are approximately equal in length and minimizes crossings between the edges.

The force layout can be used in one of the following modes:

- **Standard mode**: This is the default mode. In this mode, all vertices of the graph gravitate towards each other equally regardless of their label or property values.

Figure 2-3 Default Force Layout



Also, note that in densely populated graphs, the combination of vertex size and vertex count can generate strong repulsion forces in the force layout. This causes vertices to be pushed too far apart. You can correct this behavior by:

  - Reducing the number of vertices displayed per page.
  - Decreasing the size of vertices through styles.
  - Lowering the `spacing` property of the force layout.
  - Increasing the `velocityDecay` property of the force layout.

- **Cluster mode**: You can activate the cluster mode by setting `"clusterEnabled": true`. In this mode, vertices within the same cluster are attracted more strongly towards each other than those in different clusters, or no cluster. This is useful to visualize clusters or communities of vertices in the graph.
  You can define the following properties to configure the cluster layout. Note that you can use `clusterOptions` to specify the vertex property which defines the community or cluster membership of the vertices.

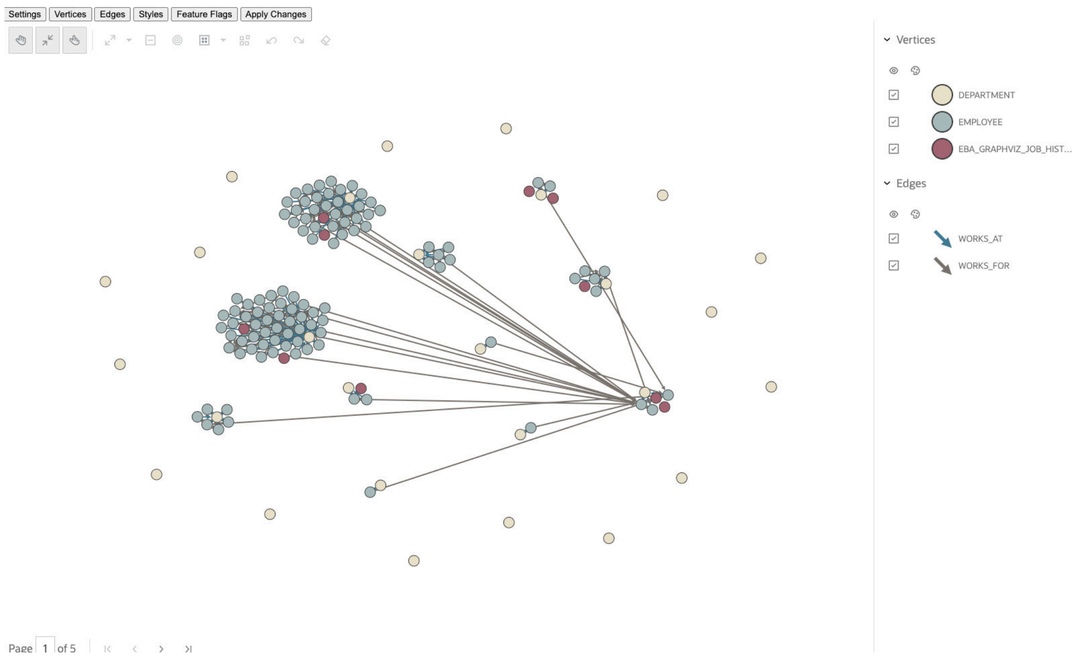
  - `edgeDistance`: Sets every edge to the specified length. This can affect the padding between the vertices.
  - `vertexCharge`: Influences the underlying forces (for example, to remain within the viewport, to push vertices away from each other). If `clusterEnabled` is `true`, then it influences the forces among clusters.
  - `velocityDecay`: Determines the speed of the simulation.
  - `spacing`: Determines the spacing between the vertices.
  - `clusterEnabled`: Determines if a cluster based layout is enabled.
  - `clusterOptions`: Related settings for cluster based layout only.
  - `clusterBy`: By default, the cluster layout uses the first element in `vertex.labels` to form the cluster. Alternatively, `clusterBy` can also be set to the property name of a vertex. In such a case, the clusters will be formed based on the property value.
  - `hideUnclusteredVertices`: Determines whether to display the vertices that do not belong to any cluster. Default is `false`.

  The following shows an example for cluster layout:

```
Settings:
{
  ...
  layout:
  {
    type: 'force',
    clusterEnabled: true,
    clusterOptions:
    {
      clusterBy: 'DEPARTMENT_ID',
      hideUnclusteredVertices: true
    }
  }
}
```

  The example aims to create clusters based on `DEPARTMENT_ID`. The corresponding visualization using cluster layout is as shown:

Figure 2-4 Cluster Layout

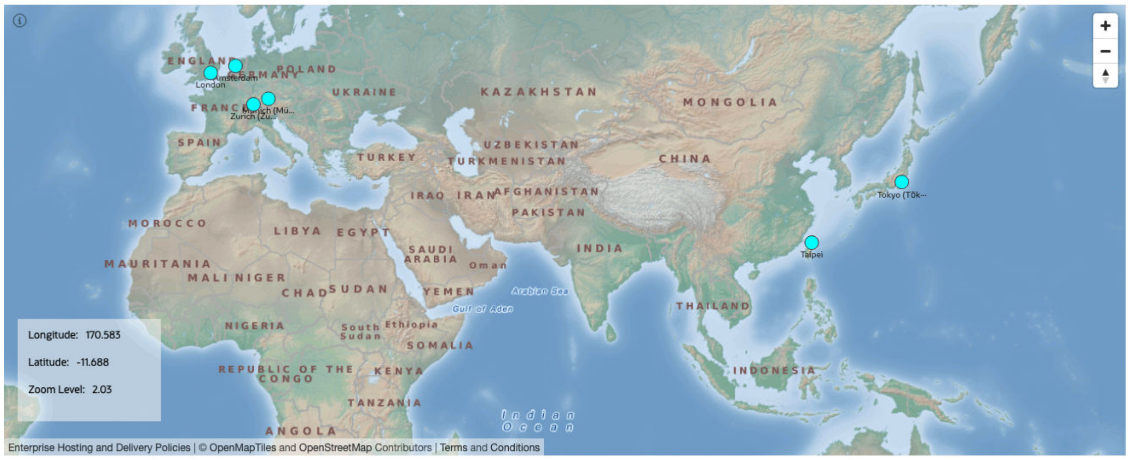


2.1.4 Geographical Layout

The geographical layout allows you to overlay the graph on a map.

However, this is provided that the latitude and longitude coordinates exist as graph properties on the graph's vertices.

Figure 2-5 Geographical Layout


London
Amsterdam
Munich (Mü...
Zurich (Zü...
Tokyo (Tōk...
Taipei
Longitude: 170.583
Latitude: -11.688
Zoom Level: 2.03
Enterprise Hosting and Delivery Policies | © OpenMapTiles and OpenStreetMap Contributors | Terms and Conditions

You can configure this layout using the following properties:

- `appId`: This accepts the `app id` that is used to fetch the maps from `http://maps.oracle.com/elocation`. If a value is not provided, then a generic `appId` will be used.
- `latitude`: The vertex property to use for determining the latitude of a vertex.

- `longitude`: The vertex property to use for determining the longitude of a vertex.
- `mapType`: You can select the map type in map visualization or graph visualization settings. Alternatively, you can also provide your own sources and layers.
  The following map types are available:
  – *world_map_mb ("oracle-elocation")*
  – *osm_positron (default)*
  – *osm_bright*
  – *osm_darkmatter*
  – *custom_type*
    Note that the custom type has the following two additional fields:
    * `sources`: Provide your own sources to be used in the map in JSON format.
      Note: Due to security reasons, the `attribute` property is separate from visualization.
    * `layers`: Provide layers which you want to display on map in JSON elements array format. For example:

      ```
      [{
        "id": "elocation-tiles",
        "type": "raster",
        "source": "oracle-elocation"
      }]
      ```

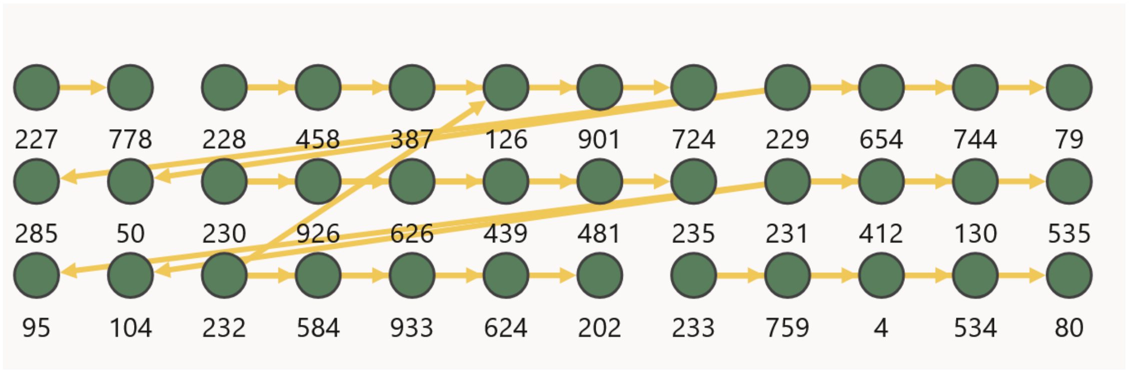
- `showInfo`: Displays an info box in the visualizer (see Figure 2-5) that shows the *Latitude* and *Longitude* of the mouse position and the *Zoom Level* of the map. Supported values are `true` or `false`.
- `showNavigation`: Shows the navigation controls towards the top right region of the map.
- `markers`: Displays location markers on the map. This parameter accepts an array of objects as shown in the following format:

  ```
  interface MapMarker {
    longitude: number;
    latitude: number;
    content?: string;
  }
  ```

2.1.5 Grid Layout

The grid layout positions the graph vertices in a well-spaced grid.

Figure 2-6 Grid Layout

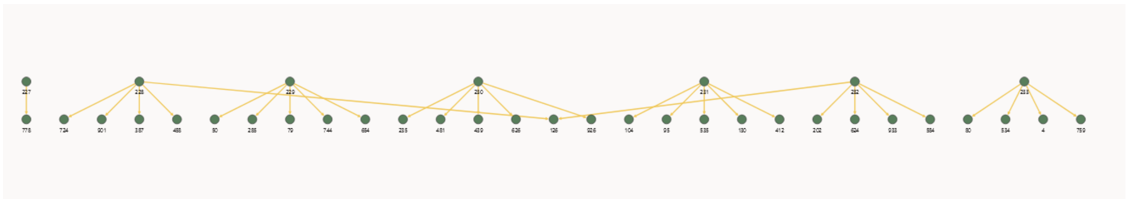


You can configure the `spacing` property to set the space between the elements in the grid.

2.1.6 Hierarchical Layout

The hierarchical layout organizes the graph using Directed Acyclic Graph (DAG) system. It is especially suitable for DAGs and trees.

Figure 2-7 Hierarchical



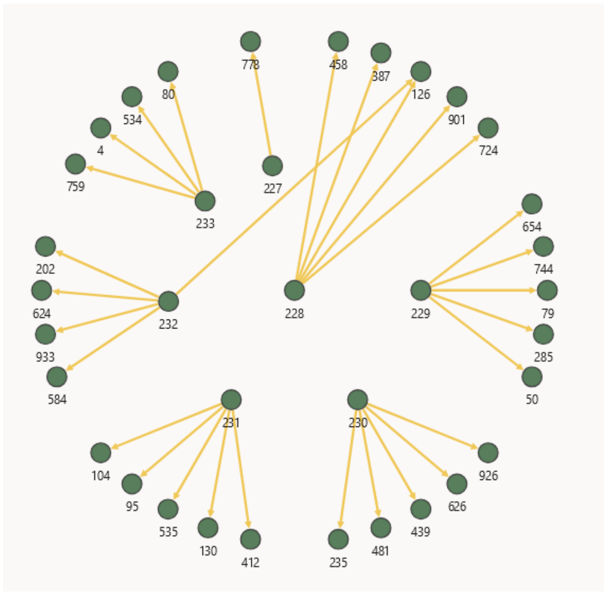
You can configure this layout using the following properties:

- `ranker`: Specifies the type of algorithm used to rank the vertices.
  Supported algorithms are:
  - *`network-simplex`*: The Network Simplex algorithm assigns ranks to each vertex in the input graph and iteratively improves the ranking to reduce the length of the edges.
  - *`tight-tree`*: The Tight Tree algorithm constructs a spanning tree with tight edges and adjust the ranks of the input vertex to achieve this. A tight edge is one that has a length that matches its `minlen` attribute.
  - *`longest-path`*: The Longest Path algorithm pushes the vertices to the lowest layer possible, leaving the bottom ranks wide and the edges longer than necessary.
- `rankDirection`: Controls the alignment of the ranked vertices. Supported values are: `UL` (upper left), `UR` (upper-right direction), `DL` (down-left direction), `DR` (down-right direction), `TB` (top-to-bottom direction), `BT` (bottom-to-top direction), `LR` (left-to-right direction), `RL` (right-to-left direction).
- `vertexSeparation`: Sets the horizontal separation between the vertices.
- `edgeSeparation`: Sets the horizontal separation between the edges.
- `rankSeparation`: Sets the separation between two ranks(levels) in the graph.

2.1.7 Radial Layout

The radial layout displays the dependency chain of a graph by using an outwards expanding tree structure. It can be especially useful if the graph data has a hierarchical structure and contains many children for each parent vertex.

Figure 2-8 Radial Layout

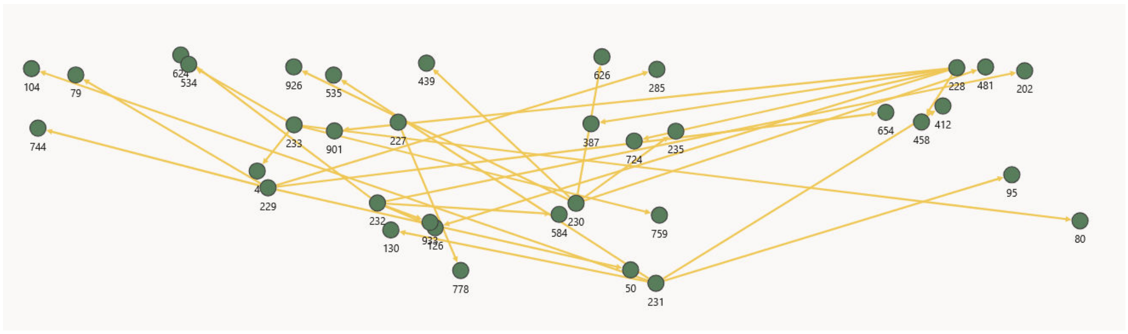


You can configure the `spacing` property to set the spacing between neighboring vertices if they share the same parent vertex. If set to zero, no spacing will be applied.

2.1.8 Random Layout

The random layout puts the graph vertices in random positions within the viewport.

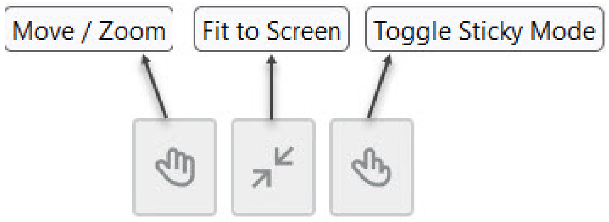
Figure 2-9 Random Layout



2.2 Exploration Modes

The Graph Visualization library supports three different modes for graph exploration.

Figure 2-10 Graph Exploration Modes



The supported modes are:

- **Move / Zoom**
  - If *Move / Zoom* mode is enabled, you can select and move multiple vertices and edges simultaneously within the visualization.
  - If *Move / Zoom* mode is disabled, you can freely explore the visualization by panning and zooming, focussing on specific groups of vertices and edges.
- **Fit to Screen**
  - If *Fit to Screen* mode is enabled, the visualization automatically adjusts to fit all vertices and edges within the available space, providing a complete view of the entire graph.
  - If *Fit to Screen* is disabled, the available space for visualization dynamically expands as required.
- **Toggle Sticky Mode**
  - If *Sticky* mode is enabled, the positions of vertices and edges in the graph are flexible. You can move the vertices and edges around, and their new positions will be retained until the mode is turned off.
  - If *Sticky* Mode is disabled, the positions of vertices and edges remain fixed. Even if you attempt to move the vertices around, they will snap back to their original positions once released.

Using a combination of modes

The following describes various scenarios for using a combination of exploration modes:

- **All modes switched off**
  This provides a full view of the graph, where the positions of the vertices are fixed. You can navigate around the visualization and zoom in or out as required.
- **Move / Zoom mode on, Fit to Screen and Sticky mode off**
  This allows you to select one or multiple vertices, and is useful for actions like grouping or expanding. However, the positions of the vertices remain fixed since *Sticky* mode is switched off.
- **Move / Zoom mode and Fit to Screen on, Sticky mode off**

This mode behaves similarly to the previous case. However, when an action like *expand* generates a new set of vertices, the component automatically adjusts its size to fit all new vertices and edges.

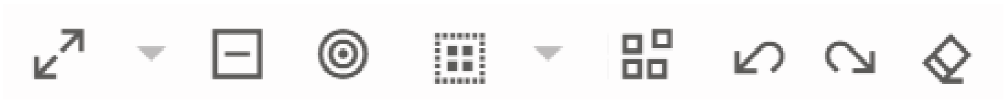
- **Move / Zoom mode and Fit to Screen off, Sticky mode on**
  You can navigate the visualization, zoom in and out, and reposition the vertices. These changes will be retained until the *Sticky* mode is turned off.
- **Move / Zoom mode and Sticky mode on, Fit to Screen off**
  You can select multiple vertices and move them around, with their positions retained. Vertices can also be moved outside the visible region since *Fit to Screen* mode is switched off.
- **All modes switched on**
  You can select multiple vertices and move them around, with their new positions retained. The visualization region will dynamically expand as needed to accommodate these changes.

2.3 Graph Interaction Options

The Graph Visualization library supports different types of graph interactions.

The following graph interaction options are supported (described in order from left to right):

Figure 2-11 Graph Interaction Options

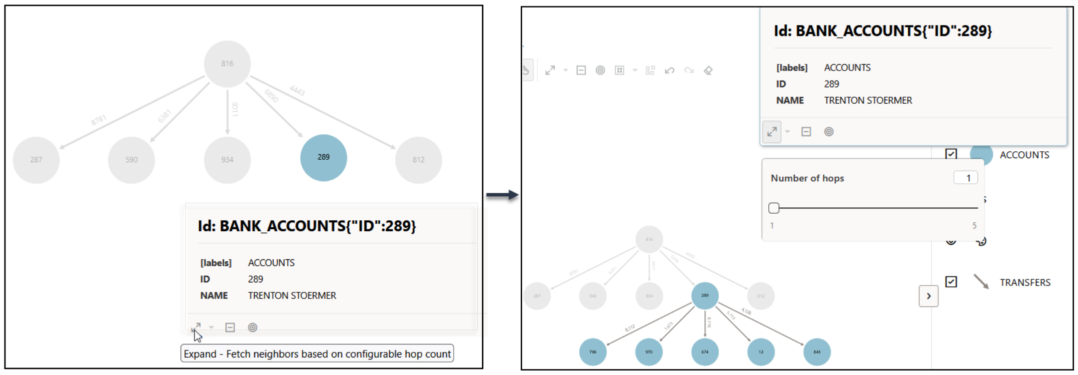


- **Expand**: You can expand one or more selected vertices to fetch n-hop neighbors.
  To expand a vertex, click on the specific vertex and click the **Expand** option in the toolbar. Alternatively, you can right-click on the vertex and click **Expand** in the context menu.

  By default, the expand action fetches 1-hop neighbors and also displays a hop count panel. You can control the hop count through the slider in the panel. The range for the hop count is between 1 and `settings.maxNumberOfHops` (default value is five). You can drag the slider or edit the hop count value directly in the text box as desired.

  The following figure illustrates expanding on vertex 289. The image on the right shows that the specified hop count value is one and therefore 1-hop neighbors are fetched when expanding on the vertex.

Figure 2-12 Expand Action



Id: BANK_ACCOUNTS{"ID":289}

[labels] ACCOUNTS
ID 289
NAME TRENTON STOERMER

Expand - Fetch neighbors based on configurable hop count

Id: BANK_ACCOUNTS{"ID":289}

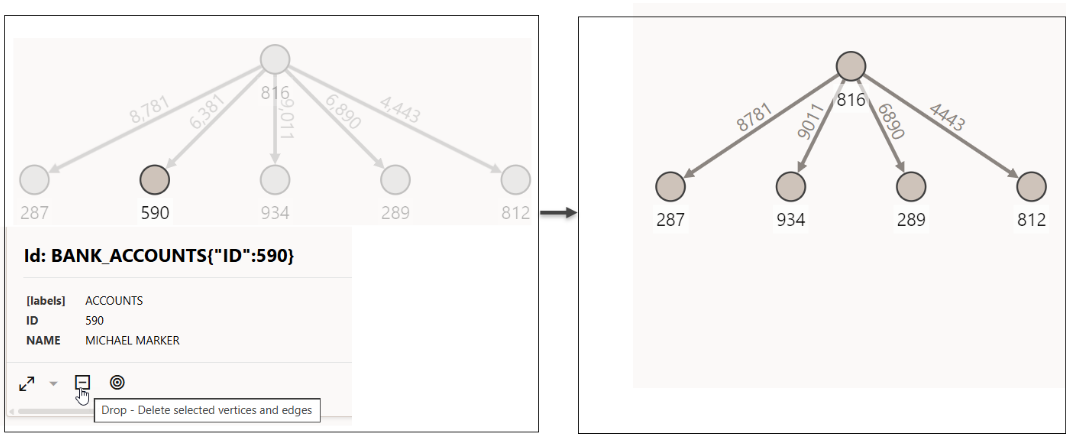
[labels] ACCOUNTS
ID 289
NAME TRENTON STOERMER

Number of hops 1

- **Drop Action**: You can remove one or more selected vertices from the visualization. To drop a vertex, click on the specific vertex and click the **Drop** option in the toolbar. Alternatively, you can right-click on the vertex and click **Drop** in the context menu.

  The following figure illustrates dropping of vertex `590`.

  Figure 2-13 Drop Action

  Id: BANK_ACCOUNTS{"ID":590}

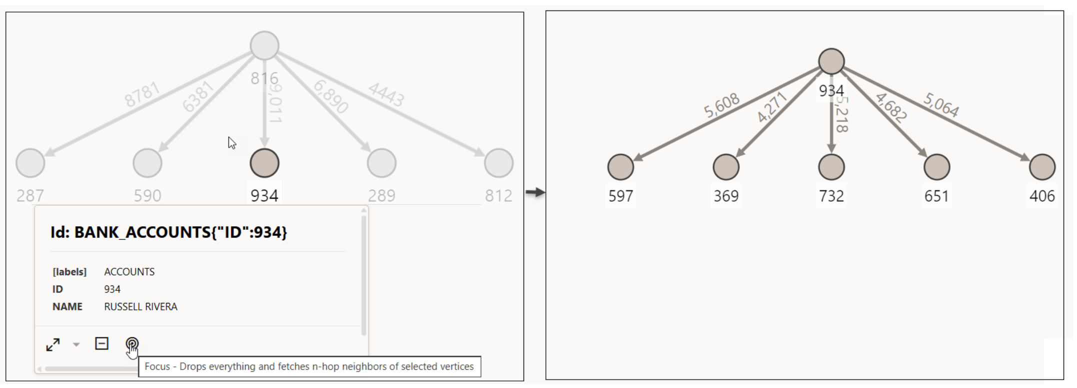
[labels]	ACCOUNTS
ID	590
NAME	MICHAEL MARKER

  Drop - Delete selected vertices and edges

- **Focus**: You can drop everything in your visualization and fetch n-hop neighbors of one or more selected vertices.
  To focus on a vertex, click on the specific vertex and click the **Focus** option in the toolbar. Alternatively, you can right-click on the vertex and click **Focus** in the context menu.

  The following figure illustrates focused visualization on vertex `934`.

  Figure 2-14 Focus Action

  Id: BANK_ACCOUNTS{"ID":934}

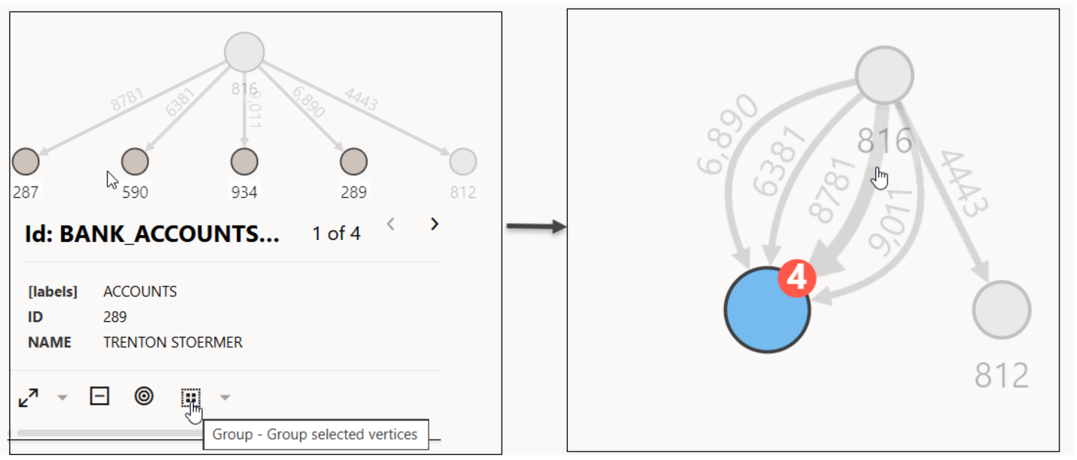
[labels]	ACCOUNTS
ID	934
NAME	RUSSELL RIVERA

  Focus - Drops everything and fetches n-hop neighbors of selected vertices

- **Group**: You can group selected vertices and collapse them into a single one.
  To group vertices, select multiple vertices and click the **Group** option in the toolbar. Alternatively, you can right-click on the selection and click **Group** in the context menu.

  The following figure illustrates grouping of vertices `287`, `590`, `934`, and `289`.

Figure 2-15   Group Action

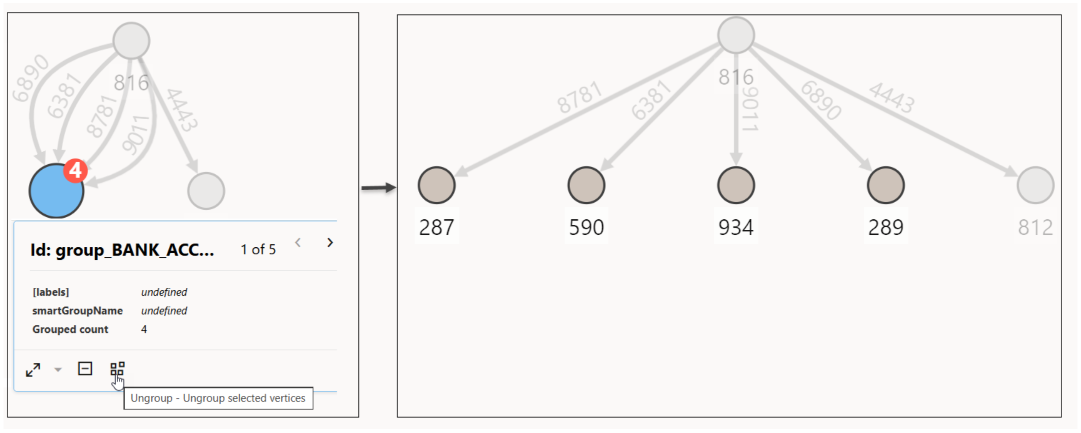


[labels]	ACCOUNTS
ID	289
NAME	TRENTON STOERMER

- **Ungroup**: You can ungroup a group of selected vertices.
  To ungroup, select the group and click the ungroup option in the toolbar. Alternatively, you can right-click on the group and click **ungroup** in the context menu.

  The following figure illustrates ungrouping of vertices `287`, `590`, `934`, and `289`.

Figure 2-16   Ungroup Action



[labels]	*undefined*
smartGroupName	*undefined*
Grouped count	4

- **Undo**: To undo the last action in your visualization.
- **Redo**: To redo the last action in your visualization.
- **Reset**: To reset the visualization to its original state.

3

Graph Visualization Library Reference

This section provides the JavaScript API reference documentation for the Graph Visualization library.

Learn about the different properties and events supported by the Graph Visualization library.

- Properties
- Events
- Methods
- Schema View
  The Graph Visualization library allows you to visualize a graph's schema in the form of a property graph.

3.1 Properties

The graph visualization component contains the following properties:

- types
  This section describes the custom types supported in the Graph Visualization library.
- data
  This section describes the interfaces that support the initial graph data in a visualization.
- settings
  This section describes the settings to configure the graph layout, page size, theme, legends, animation, and so on.
- schema
  This section describes the properties to visualize the graph schema.
- schemaSettings
  You can configure specific settings for the schema visualization.
- featureFlags
  This section describes the hierarchical flags to hide specified features or group of features.
- fetchMore
  This section describes the callback to retrieve a given page of graph data.
- expand
  This section describes the callback to retrieve n-hops neighbors of specified vertices.
- eventHandlers
  This section describes the callbacks to handle events triggered by the graph entities (vertices or edges).
- persist
  This section describes the callback to save any graph modification to a datasource.
- fetchActions
  This section describes the callback to retrieve actions from a data source and apply them during the initial loading of the graph.

- search
  This section describes the callback to retrieve a list of vertices and edges that matches a search.
- updateEvolution
  This section describes the callback to enable or disable the evolution feature.
- updateSelectedOption
  This section describes the callback to update the selected option for rule-based expand or rule-based group.
- updateSearchValue
  This section describes the callback to update the value used for live search.
- updateGraphData
  This section describes the callback to handle events when the graph data is updated.
- updateRuleBasedStyle
  This section describes the callback to perform custom handling when visibility or styling toggle, or reordering is performed in the legend area.
- editRuleBasedStyle
  This section describes the callback to handle the editing of rule based styles using an external user interface.
- onManyClick
  This section describes the callback to retrieve vertex and edge counts by executing the provided graph query and vertex/edge ID column names.
- savedSearches
  This section provides the list of saved conditional search configurations. When provided, these configurations will be available in the user interface for conditional search.
- savedSearchProvider
  This section describes the provider callbacks for listing, saving, or deleting saved conditional searches.

3.1.1 types

This section describes the custom types supported in the Graph Visualization library.

```
type Optional<T> = T | undefined;

type Nullable<T> = T | null;

type TypedMap<T> = Record<string, T>;

type NonEmptyArray<T> = [T, ...T[]];

type VertexSearchResult = Record<Id, Vertex>;

type EdgesSearchResult = Record<Id, Edge>;

type DefaultProps = Record<Id, string | number>;

type EntityProperties = TypedMap<Optional<string | number | boolean |
TypedMap>>;

type TriangleDirection = 'up' | 'down' | 'left' | 'right';
```

```
interface CircleVertexShape {
  type: 'circle';
}

interface EllipseVertexShape {
  type: 'ellipse';
  widthHeightRatio?: number;
}

interface SquareVertexShape {
  type: 'square';
}

interface RectangleVertexShape {
  type: 'rectangle';
  widthHeightRatio?: number;
}

interface RoundedRectangleVertexShape {
  type: 'roundedRectangle';
  widthHeightRatio?: number;
  cornerRadiusRatio?: number;
}

interface TriangleVertexShape {
  type: 'triangle';
  direction?: TriangleDirection;
}

interface DiamondVertexShape {
  type: 'diamond';
}

interface PentagonVertexShape {
  type: 'pentagon';
}

interface HexagonVertexShape {
  type: 'hexagon';
}

interface OctagonVertexShape {
  type: 'octagon';
}

type VertexShape =
  | CircleVertexShape
  | EllipseVertexShape
  | SquareVertexShape
  | RectangleVertexShape
  | RoundedRectangleVertexShape
  | TriangleVertexShape
  | DiamondVertexShape
  | PentagonVertexShape
  | HexagonVertexShape
  | OctagonVertexShape;
```

```
type ShapeType = VertexShape['type'];

type OnManyClick = (
  graphQuery: string,
  vertexIdColumnNames: string[],
  edgeIdColumnNames: string[]
) => Promisable<{ vertexCount: number; edgeCount: number }>;
```

3.1.2 data

This section describes the interfaces that support the initial graph data in a visualization.

```
interface TypedArrayMap<TValue = any> {
  [key: string]: TValue;
}

interface Fetchable {
  // Number of results used for tooltip pagination
  numResults?: number;
  // Flag indicating whether this is the last result set.
  isLastResultSet?: boolean;
}

interface Graph extends Fetchable {
  // Graph vertices
  vertices: Vertex[];
  // Graph edges
  edges: Edge[];
  // Vertex ID column names
  vertexIdColumnNames?: string[];
  // Edge ID column names
  edgeIdColumnNames?: string[];
}

declare type Id = string | number;

interface Classable {
  // Entity classes used for styling
  classes?: string[];
}

interface Entity extends Classable {
  // Entity id
  id: Id;
  // Arbitrary entity properties
  properties?: EntityProperties;
  // Inline style
  style?: Style;
  // Labels associate with entity
  labels?: string[];
}

interface Vertex extends Entity {}
interface Edge extends Entity {
  // Source vertex id
  source: Id;
  // Target vertex id
  target: Id;
 // Indicates an artificial edge that represents an indirect path between 
visible vertices.
  isReachability?: boolean;
  // Lists hidden vertex ids that lie on the underlying indirect path.
  via?: Id[];
  // Id of the original edge that supplied the base styling for this indirect 
edge.
  seedEdgeId?: Id;
}

interface EdgeWithLayout extends Edge {
  _source: Id;
  _target: Id;
}

interface ReachabilityPathDescriptor {
  source: Id;
  target: Id;
  via: Id[];
  seedEdgeId: Id;
  hiddenVertexCount: number;
  hiddenVertices: Id[];
  hiddenEdgeCount: number;
  hiddenEdges: Id[];
}

interface QueueEntry {
  seedEdgeId: Id;
  current: Id;
  via: Id[];
  visited: Set<Id>;
  depth: number;
  hiddenPathEdges: Id[];
}
```

3.1.3 settings

This section describes the settings to configure the graph layout, page size, theme, legends, animation, and so on.

```
interface SearchResult {
  vertices?: VertexSearchResult;
  edges?: EdgesSearchResult;
  defaultProps?: DefaultProps;
}

type Theme = 'light' | 'dark';

type EdgeMarker = 'arrow' | 'none';
type SizeMode = 'compact' | 'normal';

type ExpandedState = 'expanded' | 'collapsed';

type DefaultSettings = {
  // Specifies the default state of the 'Select - Move/Zoom' toggle button in
the toolbar. True activates 'Select' mode and false switches to 'Move/Zoom'
mode.
  interactionActive: Optional<Boolean>;
  // Specifies the default state of the 'Fit to Screen' toggle button in the
toolbar. True activates the button and false deactivates it.
  fitToScreenActive: Optional<Boolean>;
  // Specifies the default state of the 'Sticky mode' toggle button in the
toolbar. True activates the button and false deactivates it.
  stickyActive: Optional<Boolean>;
  // Specifies the default state of the 'Evolution' toggle button in the
toolbar. True activates the button and false deactivates it.
  evolutionActive: Optional<Boolean>;
  // Specifies the default state of the 'Reachability Paths' toggle button in
the toolbar. True activates the button and false deactivates it.
  reachabilityPathsActive: Optional<Boolean>;
};

// When the mode is 'hideWhenAnyUnchecked', graph element (vertex/edge) will
be hidden when one of the legend items affecting it has its visibility turned
off.
// When the mode is 'hideWhenAllUnchecked', graph element (vertex/edge) will
be hidden only when all the legend items affecting it has its visibility
turned off.
type VisibilityToggleMode = 'hideWhenAnyUnchecked' | 'hideWhenAllUnchecked';

interface Settings {
  // Number of elements to display on first load (default 100)
  displaySizeOnLoad?: number;
  // Size of pagination page (default 100).
  // Note: pageSize is deprecated in Graph Server and Client Release 25.3.
Instead, use displaySizeOnLoad.
  pageSize?: number;
  // Whether to group edges with the same source and target (default false).
  groupEdges: boolean;
  // Marks if numbers should be formatted, locale aware (Default true).
  formatNumbers: boolean;
  // Layout type or LayoutSettings (default force).
  layout: LayoutType | Partial<LayoutSettings>;
  // Network Evolution configuration.
  evolution: NestedPartial<Shortcuts<Evolution>>;
  // Rule-based group definitions for vertex and edge groups.
  ruleBasedGroups: RuleBasedGroup[];
  // Rule-based expand definitions for vertices.
  ruleBasedExpands: RuleBasedExpand[];
  // @Deprecated since version 26.2, use ruleBasedGroups instead.
  smartGroups: SmartGroup[];
  // @Deprecated since version 26.2, use ruleBasedExpands instead.
  smartExpands: SmartExpand[];
  // Stores the selected rule-based group id (vertex or edge).
  selectedRuleBasedGroup: Nullable<number>;
  // Stores the selected rule-based expand id.
  selectedRuleBasedExpand: Nullable<number>;
  // @Deprecated since version 26.2, use selectedRuleBasedGroup instead.
  selectedSmartGroup: Nullable<number>;
  // @Deprecated since version 26.2, use selectedRuleBasedExpand instead.
  selectedSmartExpand: Nullable<number>;
  // Enables live search feature.
  searchEnabled: boolean;
  // Escapes HTML content used on vertex/edge tooltip.
  escapeHtml: boolean;
  // Width used for legend area.
  legendWidth: number;
  // Number of hops used for expand and focus actions.
  numberOfHops: number;
  // Maximum number of hops possible while performing expand and focus
actions.
  // Default value is 5. If the specified value is less than or equal to one,
then the hop count slider panel will not be displayed.
  maxNumberOfHops: number;
  // Size mode determines the size of UI elements (like toolbar buttons,
search region etc).
  // Possible values are 'compact' and 'normal'. If not specified, it will be
computed based on the available page width.
  sizeMode: SizeMode;
  // Property used for live search feature.
  searchValue: string | undefined;
  // Edger marker, can be 'arrow' or 'none'. Default is 'arrow'.
  edgeMarker: EdgeMarker;
  // Limit of characters that are shown for vertex/edge label.
  charLimit: number;
  // Show title of edge/vertex components.
  showTitle: boolean;
  // Vertex property showed on the visualization.
  vertexLabelProperty: Nullable<string>;
  // Edge property showed on the visualization.
  edgeLabelProperty: Nullable<string>;
  // theme settings (default light theme).
  theme: Theme;
  // customized theme settings.
  customTheme: CustomTheme;
  // Limit of characters shown on the vertex/edge tooltip. A value of 0 means
no limit. Default is 100.
  tooltipCharLimit: Nullable<number>;
  // Styles applied to all vertices and edges
  baseStyles: Styles;
  // Rules correspond to Legend entries that also control visiblity/styling
highlights.
  ruleBasedStyles: RuleBasedStyleSetting[];
// Determines whether the view represented by the Settings (Schema View or
Graph View) is in 'expanded' or 'collapsed' state.
  viewMode?: ExpandedState;
  // Specifies the value shown in the label and tooltip to set the current
view's context ('Schema' or 'Graph')
  viewLabel?: string;
  // Specifies whether the graph Views' legend region is in expanded or
collapsed state. Not applicable for schema settings
```

```
  legendState?: ExpandedState;
  // Specifies whether accessibility mode (that includes features such as
keyboard interactions, voice over support, and so on) is enabled.
  // This will be enabled by default unless explicitly set to false.
  accessibilityEnabled?: Boolean;
  // Specifies how visibility of graph elements are determined when
visibility checkbox of the legend is toggled.
  // Defaults to 'hideWhenAnyUnchecked' when not specified.
  visibilityToggleMode?: VisibilityToggleMode;
  // Enables autogenerated legend entries based on graph data. Defaults to
true.
  // Set to false to rely solely on ruleBasedStyles for legend entries.
  defaultLegendEnabled?: boolean;
  // Specifies the default state of various aspects of GVT
  defaults: Partial<DefaultSettings>;
}

// Denotes the id involved in the validation failure.
type EntityValidationType = 'vertex' | 'edge' | 'schema vertex' | 'schema
edge';

interface EntityValidationError {
  // Denotes the type of the entity whose validation failed.
  entityType?: EntityValidationType;
  // Denotes the id of the entity involved in the validation failure.
  entityId?: string;
  // Denotes the label of the entity involved in the validation failure.
  entityLabel?: string[];
  // Denotes the property of the entity involved in the validation failure.
  property?: string;
  // Denotes the type of the property received.
  actualType?: string;
  // Denotes the type of the property expected.
  expectedType?: string;
  // Denotes the validation message as string.
  message?: string;
}

interface SchemaValidationError extends EntityValidationError {
  // Error codes returned when schema validation fails
  errorType:
    | 'MANDATORY_PROPS_IN_SCHEMA_MISSING_IN_GRAPH'
    | 'ENTITY_IN_SCHEMA_MISSING_IN_GRAPH'
    | 'ITEMS_IN_GRAPH_NOT_DEFINED_IN_SCHEMA'
    | 'TYPE_MISMATCH_BETWEEN_SCHEMA_AND_GRAPH';
}

interface StyleValidationError extends EntityValidationError {
  // Error codes returned when validation of rule based styles fail
  errorType: 'CONDITIONS_MISSING' | 'OPERATION_NOT_SUPPORTED';
  operator?: FilterOperator;
}

type FilterComponent = 'vertex' | 'edge';

type ApplyTarget = 'vertex' | 'source' | 'target' | 'edge' | 'ingoing' |
'outgoing';

type FilterOperator = '<' | '<=' | '>' | '>=' | '=' | '!=' | '~' | 'CONTAINS'
| 'CONTAINS_REGEX';

interface ElementProperty<T> {
  property: string;
  value: T;
}

interface BasicCondition extends ElementProperty<string | string[]> {
  operator: FilterOperator;
}

interface RuleCondition {
  rule: string;
}

interface ExpandCondition extends BasicCondition {
  component: FilterComponent;
}

type ConditionsOperator = 'and' | 'or';

interface Conditions<T extends FilterCondition | RuleCondition |
BasicCondition> {
  conditions: T[];
  operator?: ConditionsOperator;
}

// Graph animations are applied within the filter properties.
interface GraphAnimation {
  id?: string;
  duration: number;
  timingFunction: string;
  direction?: string;
  keyFrames: KeyFrame[];
  iterationCount?: number;
}

type FilterProperties = {
  colors?: string[];
  classes?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  sizes?: number[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  icons?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  iconColors?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  image?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
```

```
instead.
  label?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  style?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  animations?: GraphAnimation[][];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  legendTitle?: string[];
  // @Deprecated since version 25.1 - use the equivalent RuleBasedStyle
instead.
  legendDescription?: string[];
};

interface FilterInterpolation {
  // The property on which interpolation is applied.
  property: string;
  // The minimum range for interpolation.
  min?: number;
  // The maximum range for interpolation.
  max?: number;
}

// Different types of aggregation functions are supported.
type AggregationType = 'average' | 'min' | 'max' | 'sum' | 'count' |
'distinctCount';

interface PropertyAggregation {
  // The property of vertex or edge on which aggregation is computed.
  source: string;
  // The type of aggregation function used for computation.
  type: AggregationType;
}

type OverlayIndicatorIcon = 'pin' | 'bookmark';  // icon specified will be
internally converted to the actual icon name, based on the theme used.

interface RuleBasedStyleSetting extends FromTemplate {
  // Marks if Styling is enabled for a ruleBasedStyleSetting item and the
vertices/edges that it controls.
  stylingEnabled?: boolean;
  // Conditions deciding which vertices/edges will be affected by the
ruleBasedStyleSetting item.
  conditions?: Conditions<BasicCondition | RuleCondition>;
  // The component for which the ruleBasedStyleSetting is defined.
  component: FilterComponent;
  // The target on which this ruleBasedStyleSetting applies (vertex, source,
target, edge, ingoing, outgoing).
  target: ApplyTarget;
  // The various properties (like colors, icons, image, animations) of a
vertex/edge that this ruleBasedStyleSetting's state can affect.
  properties: FilterProperties;
  // Marks if aggregation is enabled for a ruleBasedStyleSetting item, based
on which computation is performed.
```

```
  aggregationEnabled?: boolean;
  // The various aggregation properties configured on this
ruleBasedStyleSetting.
  aggregation?: PropertyAggregation[];
  // The properties and range on which interpolation will apply.
  interpolation?: FilterInterpolation;
  // References of ruleBasedStyleSetting ids.
  filterReferenceIds?: number[];
  // Legend title for the rule.
  legendTitle?: string;
  // Style for modifiers. Keys can be selected, unselected, group, hover.
  modifierStyles?: TypedMap<VertexStyle | EdgeStyle>;
  // Properties for animations.
  animations?: GraphAnimation[][];
  // Marks if the rule is a default rule.
  isDefaultRule?: boolean;
  // Marks the legend item's preview region using the specified icon as an
overlay.
  overlayIndicator?: OverlayIndicatorIcon;
  // Marks if the legend entry is displayed in the legend area (Default true).
  legendDisplayed?: boolean;
}

interface LegendEntry extends RuleBasedStyleSetting{
  // The title of the legend entry when the ruleBasedStyleSetting is shown in
the legend area.
  legendTitle?: string[];
  // Marks if the legend entry is visible in the legend area.
  legendEntryVisible: boolean;
  // The style of legend entry in the legend area.
  style: Partial<VertexStyle> | Partial<EdgeStyle>;
  // The vertices / edges on which this legend entry has influence.
  filteredNodes: Vertex[] | Edge[];
  // The style is from RuleBasedSetting and will be applied to elements that
match the rule.
  toApplyStyle?: Partial<VertexStyle> | Partial<EdgeStyle>;
}

type LayoutSettings =
  | CircleLayoutSettings
  | ConcentricLayoutSettings
  | ForceLayoutSettings
  | GridLayoutSettings
  | HierarchicalLayoutSettings
  | PresetLayoutSettings
  | RadialLayoutSettings
  | RandomLayoutSettings;

type LayoutType = 'circle' | 'concentric' | 'force' | 'grid' | 'hierarchical'
| 'preset' | 'radial' | 'random';

interface BaseLayoutSettings {
  type: LayoutType;
}

interface SpacingLayoutSettings {
  // Spacing among vertices in multiples of vertex radius.
  spacing: number;
}

interface CircleLayoutSettings extends BaseLayoutSettings,
SpacingLayoutSettings {
  type: 'circle';
}

interface ClusterOptions {
  clusterBy?: string; //vertex property
  hideUnclusteredVertices?: boolean;
}

interface ConcentricLayoutSettings extends BaseLayoutSettings,
SpacingLayoutSettings {
  type: 'concentric';
}

interface ForceLayoutSettings extends BaseLayoutSettings,
SpacingLayoutSettings {
  type: 'force';
  alphaDecay: number; // (default 0.01)
  velocityDecay: number; // (default 0.1)
  edgeDistance: number; // (default 100)
  vertexCharge: number; // (default -60)
  clusterEnabled: boolean; // (default false)
  clusterOptions?: ClusterOptions;
}

// When selecting grid layout, if neither rows or columns are defined, the
graph will be displayed in a square grid.
// If rows are selected, it will be displayed in a grid with that many rows.
// If columns are selected it will be displayed in a grid witht that many
columns.
// If both rows and columns are selected, only the rows will be taken into
consideration.
interface GridLayoutSettings extends BaseLayoutSettings,
SpacingLayoutSettings {
  type: 'grid';
  rows?: number;
  columns?: number;
}

type HierarchicalRankDirection =
  | 'UL' // Up to left
  | 'UR' // Up to right
  | 'DL' // Down to left
  | 'DR' // Down to right
  | 'TB' // Top to bottom
  | 'BT' // Bottom to top
  | 'LR' // Left to right
  | 'RL'; // Right to left

type HierarchicalRanker = 'network-simplex' | 'tight-tree' | 'longest-path';
```

```
interface HierarchicalLayoutSettings extends BaseLayoutSettings {
  type: 'hierarchical';
  // Default is 'TB'.
  rankDirection: HierarchicalRankDirection;
  // Default is 'network-simplex'.
  ranker: HierarchicalRanker;
  vertexSeparation?: number;
  edgeSeparation?: number;
  rankSeparation?: number;
}

interface PresetLayoutSettings extends BaseLayoutSettings {
  type: 'preset';
  // Property of the vertex used as x coordinate.
  x: string;
  // Property of the vertex used as y coordinate.
  y: string;
}

interface RadialLayoutSettings extends BaseLayoutSettings,
SpacingLayoutSettings {
  type: 'radial';
}

interface RandomLayoutSettings extends BaseLayoutSettings {
  type: 'random';
}

interface MapMarker {
  longitude: number;
  latitude: number;
  content?: string;
}

// Types of maps.
type MapType = 'osm_positron' | 'osm_bright' | 'osm_darkmatter' |
'world_map_mb' | 'custom_type';

interface GeographicalLayoutSettings extends BaseLayoutSettings {
  type: 'geographical';
  longitude: string;
  latitude: string;
  appId?: string;
  mapType?: MapType;
  showInfo?: boolean;
  showNavigation?: boolean;
  layers?: string;
  sources?: string;
  markers?: MapMarker[];
}

interface EvolutionEntity {
  // Start property.
  start: string;
  // End property.
  end?: string;
}

interface Evolution {
  // Height of the UI component (default is 100).
  height: number;
  // Type of the chart (default is 'bar').
  chart: 'bar' | 'line';
  // Aggregation granularity in given unit (default is 1).
  granularity: number;
  // Time unit or undefined for numbers (default is undefined).
  unit?: 'second' | 'minute' | 'hour' | 'day' | 'week' | 'month' | 'year';
  // Vertex Evolution properties (or just string specifying Start property).
  vertex?: string | EvolutionEntity;
  // Edge Evolution properties (or just string specifying Start property).
  edge?: string | EvolutionEntity;
  // Defines exclusion of values.
  exclude: {
    // Array of excluded values.
    values: (string | number)[];
    // Whether to always show or hide excluded values (default is false).
    show: boolean;
  };
  // Playback options.
  playback: {
    // Number of vertex / edge changes per step.
    step: number;
    // Number of milliseconds between steps.
    timeout: number;
  };
  // If turned on, network evolution will keep the original vertex positions
of the graph
  // when vertices and edges unfold during playback.
  preservePositions: boolean;
  // Requires a string that represents the format in which the date must be
displayed.
  // The format must include either YYYY, MM, or DD. Otherwise, it will be
ignored.
  // If not provided, the following defaults apply:
  // When displaying units of days, only the day will be displayed (1, 15,
30, and so on).
  // When displaying months only the tag of the month will be displayed (Jan,
Feb, and so on).
  // When displaying years, only the year wil be displayed (2001, 1999, and
so on).
  // If the time window between the first date in the graph and the last date
  // in the graph is too big, such that the displayed time label cannot fit,
it will
  // change to the next bigger unit. For example, if the unit is days and the
labels cannot fit,
  // then it will attempt to use a month label. In case a month label is too
big, then a year label will be used.
  labelFormat?: string;
  axis?: 'vertices' | 'edges' | 'both';
}

type RuleBasedExplorerType = 'expand' | 'group';
```

```
interface RuleBasedExplorer extends FromTemplate {
  readonly type: RuleBasedExplorerType;
  name: string;
}

interface RuleBasedGroup extends RuleBasedExplorer {
  readonly type: 'group';
  applyOnLoad: boolean;
  enabled: boolean;
  component: 'vertex' | 'edge';
  groupBy?: string;
  conditions: Conditions<ExpandCondition>;
  aggregations?: PropertyAggregation[];
}

interface RuleBasedExpand extends RuleBasedExplorer {
  readonly type: 'expand';
  numberOfHops: Optional<number>;
  navigation: Conditions<ExpandCondition>;
  destination: Conditions<ExpandCondition>;
}

// @Deprecated since version 26.2, use RuleBasedExplorerType instead
type SmartExplorerType = 'expand' | 'group';

interface FromTemplate {
  _fromTemplate?: boolean;
  _id?: number | string;
}

// @Deprecated since version 26.2, use RuleBasedExplorer instead
interface SmartExplorer extends FromTemplate {
  readonly type: SmartExplorerType;
  name: string;
}

// @Deprecated since version 26.2, use RuleBasedExpand instead
interface SmartExpand extends SmartExplorer {
  readonly type: 'expand';
  numberOfHops: Optional<number>;
  navigation: Conditions<ExpandCondition>;
  destination: Conditions<ExpandCondition>;
}

// @Deprecated since version 26.2, use RuleBasedGroup instead
interface SmartGroup extends SmartExplorer {
  readonly type: 'group';
  automatic: boolean;
  enabled: boolean;
  groupBy?: string;
  conditions: Conditions<ExpandCondition>;
}

type Theme = 'light' | 'dark';
```

```
interface CustomTheme {
  backgroundColor?: string;
  textColor?: string;
}

type Styles = TypedMap<VertexStyle | EdgeStyle>;

interface Style extends ElementPosition {
  // Default is (vertex: lightgray, edge: #C0C0C0).
  color: string;
  // Default is 1.
  opacity: number;
  // Css filter. Default is none.
  filter: string;
  // @deprecated since version 26.2. Use `caption` instead (label remains for
backward compatibility)
  label: Nullable<LabelStyle>;
  // Caption settings or just caption text or null for no caption
  caption?: Nullable<LabelStyle>;
  // Legend style or just legend text. It is null for no legend.
  legend: Nullable<this & { text: string }>;
  // Definitions of child elements (for example, vertex / edge badges).
  children: Nullable<TypedMap<_VertexStyle & Classable>>;
}

interface ImageStyle {
  // Image url. Default is undefined.
  url: string;
  // Image scale. Default is 1.
  scale?: number;
}

interface BorderStyle {
  // Border width. Default is 1.
  width?: number;
  // Border color. Default is #404040.
  color?: string;
}

interface IconStyle {
  // Icon class. For example, fa-bell. Default is undefined.
  class: string;
  // Icon text color. Default is white.
  color?: string;
}

interface VertexStyle extends Style {
  // Vertex radius. Default is 8.
  size: number;
  // Background image settings or just url. Null for no background.
  image: ImageStyle;
  // Vertex border settings or just color. Null for no border.
  border: BorderStyle;
  // Vertex icon settings or just class. Null for no icon.
  icon: IconStyle;
  // Vertex shape (default circle). May be a shape keyword or detailed
```

```
configuration.
  // For example: 'triangle', { type: 'ellipse', widthHeightRatio: 1.5 }
  shape?: ShapeType | VertexShape;
}

interface EdgeStyle extends Style {
  // Edge width. Default is 2.
  width: number;
  // Fill pattern. Default is undefined.
  // Dasharray values are: '1 5', '5', '5 10', '10 5', '5 1', '15 10 5', '15
10 5 10', '15 10 5 10 15', '5 5 1 5'
  dasharray: string;
}

// Position of the label or child vertex.
interface ElementPosition {
  // Angle position of label or child vertex (in degrees) w.r.t the parent
vertex.
  // Following are some values and its corresponding positioning of label or
child vertex:
  // null - inside the parent vertex
  // 0 - to the right side of the parent vertex
  // 90 - towards the top of the parent vertex
  // 180 - to the left side of the parent vertex
  // 270 - towards the bottom of the parent vertex (this is the default
position unless overridden)
  angle?: Nullable<number>;
  // Position on the edge. Value between -1 (edge start) and 1 (edge end).
  position?: number;
  // Offset from: vertex radius (> 0: outside, < 0: inside) or edge path.
  // (> 0: above, < 0: under)
  d: number;
}

interface FontStyle {
  // Font size. Default is 10.
  size?: number;
  // Font family. Default is inherited.
  family?: string;
  // Font style. Default is inherited.
  style?: string;
  // Font weight. Default is inherited.
  weight?: string;
}

interface LabelStyle extends ElementPosition {
  // Label text.
  text: string;
  // Color - Default is rgba(0, 0, 0, 0.8).
  color?: string;
  // Maximum label length. Default is 15. The whole label is displayed in
tooltip.
  maxLength: number;
  font: FontStyle;
 // When disableBackdrop is true, it hides the faded backdrop placed behind
vertex labels.
```

```
  // The backdrop that is enabled by default is particularly useful when
vertex label crosses over an edge
  // or when label is shown inside a vertex.
  disableBackdrop: boolean = false;
  // When resizeParent is true, vertices will adapt its size and shape to
suit the label's length.
  // Applies only when the label is shown within the vertex (that is label's
style.angle is null)
  resizeParent: boolean = false;
}
```

- Style Expressions
- Rule Expressions

3.1.3.1 Style Expressions

These expressions can access anything from the `ExpressionContext` which extends `Entity` so also all the properties of the vertex / edge that is styled.

```
interface ExpressionContext extends Entity {
  // Helper function for value interpolation
  // path: path to the ExpressionContext property that will be interpolated
  // (e.g. 'id', 'properties.someProperty')
  // min: minimum interpolation result value
  // max: maximum interpolation result value
  interpolate: (path: string, min: number, max: number) => number;
  // Previous value of evaluated property
  previous?: number | string;
}
```

Context is accessed through `${accessor}` syntax (that is JavaScript template literals). The following lists a few example expressions:

- `https://flagcdn.com/40x30/${properties.code}.png`: Constructs a URL using a given property.
- `${previous + 4}`: Returns a bigger value. This can be used, for example, to make vertices or edges bigger on hover.
- `${interpolate("group.size", 8, 16)`: Interpolation based on the grouped vertex size.

3.1.3.2 Rule Expressions

Rule expressions are used to specify the target element into which given style will be applied. It has the following structure:

```
elementName(.className)*(:modifier)*([conditionExpression])? (>
elementName(.className)*)
```

In the preceding format:

- `elementName` := `* | 'vertex' | 'edge'`
- `className` (*deprecated since 25.1*): Any `className` specified in input vertex or edge classes array.
- `modifier` := `'hover' | 'selected' | 'unselected' | 'group'`

- `conditionExpression` (*deprecated since 25.1*): JavaScript expression that can access any property of evaluated vertex or edge.
  It is recommended to use `settings.ruleBasedStyles`.

Also, note the following:

- `*`: Applies to all elements.
- `vertex`: Applies to all vertices.
- `edge`: Applies to all edges.
- `example` (*deprecated since 25.1*): Applies to all elements with example class specified.
- `vertex.example` (*deprecated since 25.1*): Applies to all vertices with example class.
- `vertex:selected`: Applies to all selected vertices.
- `vertex[id > 10]` (*deprecated since 25.1*): Applies to all vertices with `id > 10`.
  It is recommended to use `settings.ruleBasedStyles`.
- `vertex[properties.some === 'value']`: Applies to all vertices that have some property with value value.
- It is recommended to use `settings.ruleBasedStyles`. All the properties in settings are optional and have their defaults.

3.1.4 schema

This section describes the properties to visualize the graph schema.

```
interface GraphSchema {
  // Vertices of the schema.
  vertices: VertexLabelSchema[];
  // Edges of the schema.
  edges: EdgeLabelSchema[];
}

//Base interface to VertexLabelSchema and EdgeLabelSchema, holding properties
common to both.
interface EntityLabelSchema {
  // Specifies the label associated with the schema's vertex or edge.
  labelName: string;
  // Properties defined in the schema.
  properties: PropertySchema[];
}

interface VertexLabelSchema extends EntityLabelSchema {}

interface EdgeLabelSchema extends EntityLabelSchema {
  // Specifies the origin Schema for the vertex and edge.
  sourceVertexLabel?: string;
  // Specifies the target Schema for the vertex and edge.
  targetVertexLabel?: string;
}

interface PropertySchema {
  // Name of the property.
  name: string;
  // Data type of the property value.
  dataType: 'string' | 'boolean' | 'number' | 'date' | 'timestamp';
  // Limits used for validation like Maximum length, Precision/Scale, and so
on. This depends on the data type of the property.
  limits?: number[];
  // Specifies if the property should always have a value.
  mandatory?: boolean;
}
```

3.1.5 schemaSettings

You can configure specific settings for the schema visualization.

See Schema View Configuration Parameters for more information.

3.1.6 featureFlags

This section describes the hierarchical flags to hide specified features or group of features.

```
type FeatureFlags =
  | false
  | NestedFlags<{
      // Use false to hide the whole exploration.
      exploration: {
        // Use false to hide expand.
        expand: boolean;
        focus: boolean;
        group: boolean;
        ungroup: boolean;
        drop: boolean;
        undo: boolean;
        redo: boolean;
        reset: boolean;
      };
      // Use false to hide all modes.
      modes: {
        // Use false to hide interaction mode.
        interaction: boolean;
        fitToScreen: boolean;
        sticky: boolean;
        evolution: boolean;
        reachabilityPaths: boolean;
      };
      // Use false to hide display size control.
      displaySizeControl: boolean;
    }>;

type NestedFlags<T> = {
  readonly [P in keyof T]?: T[P] extends object ? false | NestedFlags<T[P]> :
T[P];
};
```

3.1.7 fetchMore

This section describes the callback to retrieve a given page of graph data.

```
// Returns Graph for additional data.
type FetchMore = (start: number, size: number) => Promise<Graph>;
```

If provided, `data` does not have to be set and graph visualization will automatically call `fetchMore` to retrieve data on initial load. If not provided, only `data` will be displayed.

3.1.8 expand

This section describes the callback to retrieve n-hops neighbors of specified vertices.

```
// ids: To expand from the ids of the selected vertices.
// hops: Number of hops to fetch from selected vertices.
// Returns the expanded graph.
type ExpandActionType = 'expand' | 'focus';

type Expand = (ids: Id[], hops: number, action: ExpandActionType,
templateId?: number | null) => Promise<Graph>;
```

By default, expand or focus is hidden.

3.1.9 eventHandlers

This section describes the callbacks to handle events triggered by the graph entities (vertices or edges).

```
// id: Id of the child vertex targeted with the event(if any).
// entity: The entity or parent of the vertex(identified by the id parameter)
targeted with the event.
type EntityEventCallback = (event: Event, id: Optional<string>, entity:
Entity) => void;

// eventType: Supported <g> element event attributes without the -on- prefix.
// children: Event handlers for child entities.
interface _EntityEventHandlers {
  [eventType: string]: EntityEventCallback | _EntityEventHandlers;
  children?: EntityEventHandlers;
}

type EntityEventHandlers = Optional<_EntityEventHandlers>;

// vertex: Callbacks that handle events fired by vertices.
// edge: Callbacks that handle events fired by edges.
interface _AllEventHandlers {
  vertex: EntityEventHandlers;
  edge: EntityEventHandlers;
}

type AllEventHandlers = Partial<_AllEventHandlers>;
```

3.1.10 persist

This section describes the callback to save any graph modification to a datasource.

```
type GraphActionType = 'drop' | 'expand' | 'focus' | 'group' | 'ungroup' |
'undo' | 'redo' | 'reset';

// vertexIds: Ids of the vertices targeted with the action.
// edgeIds: Ids of the edges targeted with the action.
interface GraphAction {
  type: GraphActionType;
  vertexIds?: NonEmptyArray<Id>;
  edgeIds?: NonEmptyArray<Id>;
  template?: Nullable<number | string>;
}

// action: Graph action to persist to a datasource.
type Persist = (action: GraphAction) => Promise<void>;
```

3.1.11 fetchActions

This section describes the callback to retrieve actions from a data source and apply them during the initial loading of the graph.

```
// This gets executed only once when the graph loads for the first time.
// It contains code to retrieve graph actions to apply on the graph initially.
type FetchActions = () => Promise<GraphAction[]>;
```

3.1.12 search

This section describes the callback to retrieve a list of vertices and edges that matches a search.

Note

The `search` callback is deprecated. Instead, use the conditional search functionality. See savedSearches and savedSearchProvider for more information.

```
// Function for live search feature.
//It returns the list of vertices and edges that matches the keyword.
type Search = (keyword: string, searchGraph?: Optional<Graph>) =>
Promise<SearchResult>;
```

3.1.13 updateEvolution

This section describes the callback to enable or disable the evolution feature.

```
// Enables or disables the network evolution feature.
type UpdateEvolution = (enabled: boolean) => Promise<void>;
```

3.1.14 updateSelectedOption

This section describes the callback to update the selected option for rule-based expand or rule-based group.

```
// Updates the selected option for smart group or smart expand.
type UpdateSelectedOption = (option: number | null, tag: SmartExplorerType)
=> Promise<void>;
```

3.1.15 updateSearchValue

This section describes the callback to update the value used for live search.

```
// Updates the search value for the live search feature.
type UpdateSearchValue = (value: string) => Promise<void>;
```

3.1.16 updateGraphData

This section describes the callback to handle events when the graph data is updated.

```
// This gets executed when the graph data gets updated.
// Vertices and edges params contains all vertices and edges of the graph.
type UpdateGraphData = (Vertices: Vertex[], edges: Edge[]) => Promise<void>;
```

3.1.17 updateRuleBasedStyle

This section describes the callback to perform custom handling when visibility or styling toggle, or reordering is performed in the legend area.

The following describes in detail the different scenarios where the `updateRuleBasedStyle` gets invoked:

- **Toggling the visibility checkbox of a legend item:** A `RuleBasedStyleSetting` containing the style with the updated `visibilityEnabled` state will be passed as the `style` parameter, while the `action` parameter will carry the value `visibilityToggle`. The consuming application can use the `style` parameter to update the `visibilityEnabled` state of the style at its end.
- **Toggling the styling using the legend item's preview:** The consuming application can use the `style` parameter to update the `stylingEnabled` state of the style at its end.
- **Reordering legend items using mouse drag and drop or by using the keyboard up/down arrows:** A `RuleBasedStyleSetting[]` array containing all the styles in the updated order will be passed as the `style` parameter, while the `action` parameter will carry the value `reorder`. The consuming application can then use the `style` parameter to update the order of styles at its end.

```
type UpdateRuleBasedStyle = (
  style: RuleBasedStyleSetting | RuleBasedStyleSetting[],
  action?: StyleAction
) => Promise<void>;
```

```
type StyleAction = 'visibilityToggle' | 'stylingToggle' | 'reorder';
```

3.1.18 editRuleBasedStyle

This section describes the callback to handle the editing of rule based styles using an external user interface.

It is triggered when `Edit Style` is invoked on items in the legend area. If the callback method is not provided, `Edit Style` button will not appear next to the legend items.

```
// Gets executed when the 'Edit Style' button of a legend item is invoked.
// A single RuleBasedStyleSetting representing the legend item on which 'Edit
Style' was invoked will be passed as the 'style' parameter.
// The consuming application can use the 'style' parameter to load the
specific style for further processing.

type EditRuleBasedStyle = (style: RuleBasedStyleSetting) => Promise<void>;
```

3.1.19 onManyClick

This section describes the callback to retrieve vertex and edge counts by executing the provided graph query and vertex/edge ID column names.

```
type OnManyClick = (
  graphQuery: string,
  vertexIdColumnNames: string[],
  edgeIdColumnNames: string[]
) => Promisable<{ vertexCount: number; edgeCount: number }>;
```

3.1.20 savedSearches

This section provides the list of saved conditional search configurations. When provided, these configurations will be available in the user interface for conditional search.

```
type SavedSearchId = string;

interface SavedSearch {
  id?: SavedSearchId;
  name: string;
  selectedLabels: string[];
  // Use unknown to avoid a hard dependency on internal filter types.
  filterRoot: unknown;
  createdAt?: string;
  updatedAt?: string;
}

// Component prop
savedSearches?: SavedSearch[];
```

3.1.21 savedSearchProvider

This section describes the provider callbacks for listing, saving, or deleting saved conditional searches.

savedSearches and `savedSearchProvider` are typically used together.

- `savedSearches`: This is the list displayed in the user interface for conditional search.
- `savedSearchProvider`: This handles persistence operations (such as save, update, and delete).

If `savedSearchProvider` is not provided, then savedSearches is disabled.

The embedding application is also responsible for keeping `savedSearches` synchronized with `savedSearchProvider`. For example, it can call `savedSearchProvider.list()` on startup and update `savedSearches` after any save or delete operation.

```
interface SavedSearchProvider {
  list?: () => Promise<SavedSearch[]>;
  upsert?: (config: SavedSearch) => Promise<SavedSearch | void> | SavedSearch
| void;
  remove?: (idOrName: string) => Promise<void> | void;
}

// Component prop
savedSearchProvider?: SavedSearchProvider;
```

- SavedSearchAction
  This section describes the union type that represents actions related to saved searches.

3.1.21.1 SavedSearchAction

This section describes the union type that represents actions related to saved searches.

```
type SavedSearchAction =
  | { type: 'saved_search/save'; config: SavedSearch }
  | { type: 'saved_search/delete'; idOrName: string }
  | { type: 'saved_search/load'; idOrName: string };
```

3.2 Events

The following events are supported:

- `graph`: This event occurs on any changes to the graph and returns the current state of the graph.
- `selection`: This event occurs on any changes to the selection of vertices and edges on the graph. It returns the currently selected vertices and edges on the graph.

3.3 Methods

The following method is supported:

`validateRuleBasedStyle(style: Partial &lt;RuleBasedStyleSetting&gt;): StyleValidationError[]`: This validation method can be invoked by the consuming application to validate the conditions in a rule based style. The style will be validated against the *schema* if present or will be validated against *data*. Any errors found will be returned as an array.

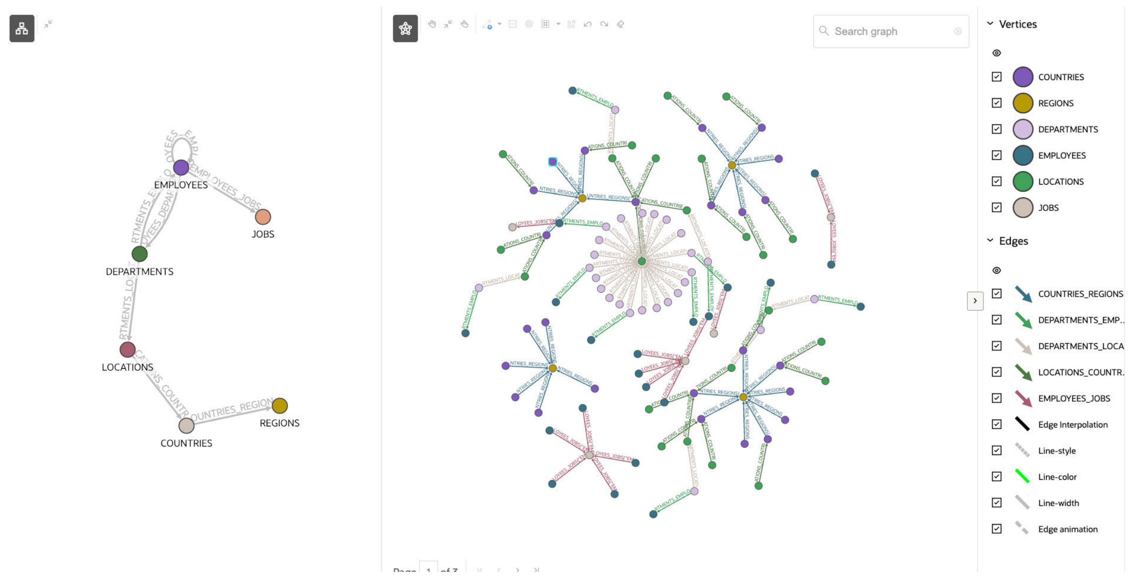
3.4 Schema View

The Graph Visualization library allows you to visualize a graph's schema in the form of a property graph.

In order to enable schema view, you must configure the properties described in Schema View Configuration Parameters.

Once the schema view is enabled, it will appear along side the graph in the graph visualization panel as shown:

Figure 3-1 Visualizing Schema and Graph Views

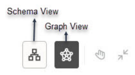


- Schema View Modes
  You can switch between **Schema** view, **Graph** view, or use both in the graph visualization panel using the toggle buttons in the toolbar.
- Schema Validation
  In addition to rendering the schema of a graph, the Graph Visualization library automatically validates the provided graph `data` against the `schema` if one is available.
- Schema View Configuration Parameters
  In order to enable **Schema View** in your visualization, you must configure a few related properties.
- Validation Rules
  The graph `data` must conform to the `schema` based on the certain validation rules.

3.4.1 Schema View Modes

You can switch between **Schema** view, **Graph** view, or use both in the graph visualization panel using the toggle buttons in the toolbar.

Figure 3-2 Schema View Modes



Depending on which toggle button is active the corresponding view will be displayed. The following describes more about the supported modes:

- **Schema**: In this mode, the schema view is displayed spanning the entire visualization.
- **Graph** *(default)*: In this mode, the graph is displayed spanning the entire visualization.
- **Schema** and **Graph**: In this mode, when both the toggle buttons are active, the schema and graph regions will appear side-by-side with a separator in between, as shown in Figure 3-1. The separator can be dragged to resize the regions.

It is important to note that depending on your data and settings configurations, either the **Schema** or **Graph** view will be always enabled in the toolbar. If both the toggle buttons are manually turned off, then it automatically defaults to **Graph** view.

3.4.2 Schema Validation

In addition to rendering the schema of a graph, the Graph Visualization library automatically validates the provided graph `data` against the `schema` if one is available.

If the graph `data` conforms to the `schema` based on the defined validation rules (see Validation Rules), then the Graph Visualization library initializes without any problems. Also, in the absence of a `schema`, the **Schema** view will not be displayed, and the schema validations will not be executed.

If the schema validation fails, then the errors will be displayed in the following JSON format:

```
[{
    errorType : string              // Used by consuming application to 
summarize or detail the error
    entityType: 'vertex' | 'edge'   // Can be used to format the error 
message to specify the context
    entityId?: string               // Used to identify the entity in the 
message
    entityDescriptor?: string[]     // Used to represent name or labels of 
the entity in the message
    property?: string               // Name of the property that failed 
validation
    expectedType?: string           // Expected type of the property
    actualType?: string             // Actual type of the property
    message?: string                // Ready to use message if no further 
customization is needed at the consuming side
}]
```

The consuming applications can then use the `message` property to obtain the complete message for display on its interface. They can also derive their own custom message, summarize errors, or categorize errors using `errorType` and other properties provided in the error message.

3.4.3 Schema View Configuration Parameters

In order to enable **Schema View** in your visualization, you must configure a few related properties.

The following properties control the behavior of **Schema View**:

- `schema` *(optional)*: This provides the data that is shown in a **Schema View**. This property takes data in the `GraphSchema` format that is described in settings. **Schema View** will not be supported if this property is not configured.
- `schemaSettings`*(optional)*: This defines various settings (for example, styling) to control the rendering of **Schema View**. This property follows the `Settings` format that is described in settings. `schemaSettings.baseStyles` and `schemaSettings.ruleBasedStyles` allow custom styling of the **Schema View**. This is similar to `settings.baseStyles` and `settings.ruleBasedStyles` that customize styling for the existing graph view.
  If styling is not specified through `schemaSettings`, then the Graph Visualization library will use a default styling to render vertices and edges in **Schema View**. Also, certain features like pagination, legends, and so on will not apply to **Schema View**. Therefore, such attributes will not have any effect even if configured in `Settings`.
- The `Settings` interface supports the following additional properties to customize **Schema** and **Graph** views display in the graph visualization panel. See settings for more information on each of the following properties.
  - `viewMode` *(optional)*: This controls the **Schema View** and **Graph View** display. It can also be used to switch between views at runtime. Supported values are `expanded` and `collapsed`. **Schema View** mode will be determined by `schemaSettings.viewMode` and **Graph View** mode will be determined by `settings.viewMode`.
  - `viewLabel` *(optional)*: This specifies a label that is displayed in **Schema View** or **Graph View**. If configured, the value will appear as a label on the top center of the corresponding view. It will also be used in the tooltip for the **Schema View** and **Graph View** toggle buttons.
  - `legendState` *(optional)*: This specifies whether the legend region in the **Graph View** is in expanded or collapsed state. Supported values are `expanded` and `collapsed`. This property will not have any effect when specified in `schemaSettings` for **Schema View** as there is no legend region for this view.

  The following shows an example `schemaSettings` for **Schema View**.

  ```
  {"viewMode": "expanded", "viewLabel": "Schema View", "baseStyles":
  { "vertex": { "caption": "${properties.labelName}", "color": "blue" }}}
  ```

  The following shows an example `settings` for **Graph View**.

  ```
  {"viewMode": "collapsed", "viewLabel": "Graph View", "legendState":
  "collapsed", "baseStyles": { "vertex": { "caption": "$
  {properties.labelName}", "color": "red"}}}
  ```

3.4.4 Validation Rules

The graph `data` must conform to the `schema` based on the certain validation rules.

The graph `data` is validated against the `schema` by verifying the following rules:

- **MANDATORY_PROPS_IN_SCHEMA_MISSING_IN_GRAPH**: This rule verifies that all the properties that are marked as **mandatory** in the `schema` have values in the graph `data`.
  The following shows a sample error if this validation fails:

  ```
  {
      errorType: "MANDATORY_PROPS_IN_SCHEMA_MISSING_IN_GRAPH",
      entityType: "vertex",
      entityId: "0",
      entityDescriptor: ["Hermione"],
      property: "id",
      message: "Vertex 'Hermione' with id '0' doesn't have a property 'id'
  while it is marked as mandatory in the schema"
  }
  ```

- **ENTITY_IN_SCHEMA_MISSING_IN_GRAPH**: This rule verifies that when entities (vertices or edges) are defined in the `schema`, the provided graph `data` also has those entities in it.
  The following shows a sample error if this validation fails:

  ```
  {
      errorType: "ENTITY_IN_SCHEMA_MISSING_IN_GRAPH",
      entityType: "edge",
      message: "The graph's data doesn't have Edges in it, but is specified
  in the schema"
  }
  ```

- **ITEMS_IN_GRAPH_NOT_DEFINED_IN_SCHEMA**: This rule verifies that entities (vertices or edges, labels, or properties) that are present in the graph, have definitions in the `schema`.
  The following shows a sample error if this validation fails due to edges being present in the graph `data` without having edges defined in the schema:

  ```
  ```
  {
      errorType: "ITEMS_IN_GRAPH_NOT_DEFINED_IN_SCHEMA",
      entityType: "edge",
      message: "Edge that is present in the graph is not defined in the
  schema."
  }
  ```
  ```

  The following shows a sample error if this validation fails due to a label being present in the graph `data` without having that label defined in the schema:

  ```
  ```
  {
      errorType: "ITEMS_IN_GRAPH_NOT_DEFINED_IN_SCHEMA",
      entityType: "vertex",
  ```
  ```

~~~
    entityDescriptor: ["misc"],
    message: "Vertex with label 'misc' that is present in the graph is not
defined in the schema."
}
```
~~~

The following shows a sample error if this validation fails due to a property being present in the graph `data` without having that property defined in the schema:

~~~
```
{
    errorType: "ITEMS_IN_GRAPH_NOT_DEFINED_IN_SCHEMA",
    entityType: "vertex",
    entityId: "0",
    entityDescriptor: ["Tom Jones"],
    property: "date",
    message: "Property 'date' of vertex 'Tom Jones' with id '0' that is
present in the graph is not defined in the schema."
}
```
~~~

- **TYPE_MISMATCH_BETWEEN_SCHEMA_AND_GRAPH**: This rule verifies that the data type of properties in the graph `data` conforms with its definition in the `schema`.
  The following shows a sample error if this validation fails:

  ```
  {
      errorType: "TYPE_MISMATCH_BETWEEN_SCHEMA_AND_GRAPH",
      entityType: "vertex",
      entityDescriptor: ["Mary"],
      property: "id",
      expectedType: "string",
      actualType: "number",
      message: "Vertex 'Mary' has an attribute 'id' of type 'number' while
  it is specified to be of type 'string' in the schema"
  }
  ```

- **MISMATCH_BETWEEN_LABELS_IN_SCHEMA**: This rule verifies that all properties defined in the `schema` across multiple labels do not contradict, when those labels are used in a vertex or edge of the graph `data`.
  The following shows a sample error if this validation fails:

  ```
  {
      errorType: "MISMATCH_BETWEEN_LABELS_IN_SCHEMA",
      entityType: "edge",
      entityId: "0",
      entityDescriptor: ["default", "Label 1", "Label 2", "Label 3"],
      property: "id",
      message: "Edge 'default' with id '0' has mismatch in the schema
  definition of 'id' property among its labels 'Label 1', 'Label 2', and
  'Label 3'"
  }
  ```

4

Usage Examples

This section provides several usage examples using the Graph Visualization library.

- Base Styles
- Default Legend Styles
- Themes
- Children
- Interpolation
- Rule-Based Styles
- Vertex Shapes
- Captions
- Animations
- Icons
- Graph Schema Visualization
- Saved Conditional Searches

4.1 Base Styles

If base styles or any rule based styles are not defined (or applied), then the following default base styles are applied to the graph:

```
const border = {
  width: 1,
  color: '#404040'
};

const badge = {
  size: 6,
  color: '#FF584A',
  caption: {
    text: '${group.size}',
    angle: null,
    color: 'white',
    font: {
      weight: 'bold'
    }
  }
};

const defaults: Styles = {
  '*': {
    filter: 'none',
    caption: {
      maxLength: 15,
      font: {
        size: 10
      }
    }
  },
  vertex: {
    size: 8,
    color: 'lightgray',
    image: {
      scale: 1
    },
    border,
    icon: {
      color: 'white'
    },
    caption: {
      angle: 270,
      d: 2
    }
  },
  'vertex:group': {
    size: '${interpolate("group.size", 8, 16)}',
    opacity: 1,
    color: '#75BBF0',
    border,
    caption: {
      text: '',
      angle: 270,
      d: 2
    },
    icon: null,
    image: null,
    legend: null,
    children: {
      size: badge
    }
  },
  edge: {
    width: 2,
    color: '#C0C0C0',
    caption: {
      position: 0,
      d: 1
    }
  },
  'edge:group': {
    width: 2,
    opacity: 1,
    caption: null,
    children: {
      size: badge
    }
  },
  '* > *': {
    size: 5,
    d: 0,
    color: 'darkgray',
    border: null,
    icon: {
      color: 'white'
    },
    image: {
      scale: 1
    },
    caption: {
      d: 1
    }
  },
  'vertex > *': {
    angle: 45
  },
  'edge > *': {
    position: 0
  },
  ':unselected': {
    filter: 'grayscale(100%)'
  },
  'vertex:unselected': {
    opacity: 0.3
  },
  'edge:unselected': {
    opacity: 0.3
  },
  'vertex:hover': {
    size: '${previous + 4}'
  },
  'edge:hover': {
    width: '${previous + 2}'
  },
  'edge:hover > *': {
    size: '${previous + 2}'
  }
};
```

If you wish to create a custom base style, then you can provide your own `settings.baseStyles`, which overrides the `defaults` shown in the preceding code.

The following shows an usage example to create a custom base style that applies for all vertices and edges:

Note

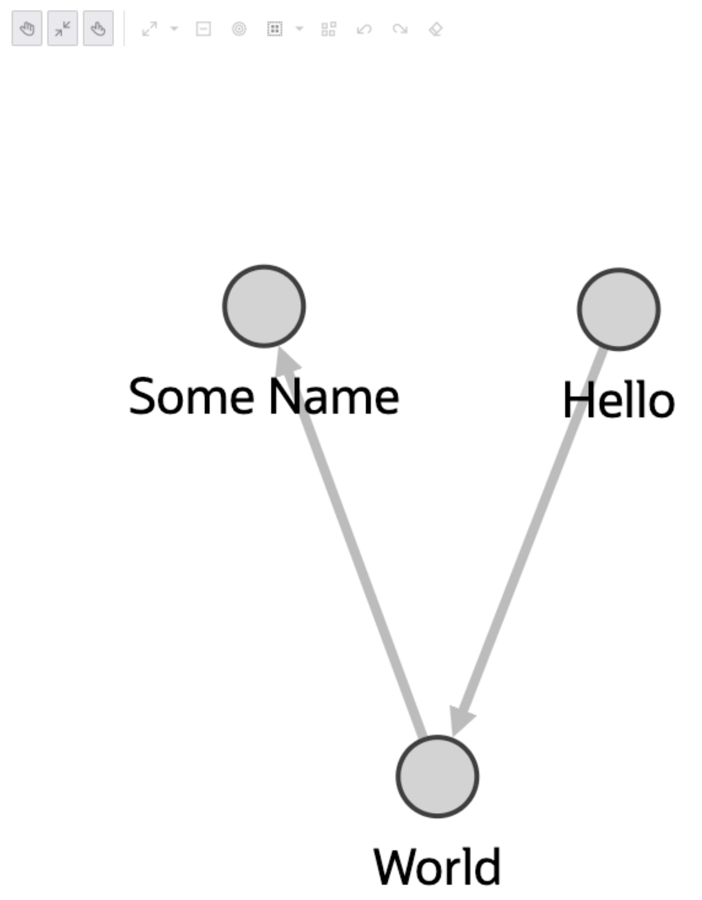
The Graph Visualization library also contains TypeScript definitions if you are using TypeScript).

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';
```

```
const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello'
    },
    labels: ['color']
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World'
    },
    labels: ['color']
  },
  {
    id: 3,
    properties: {
      name: 'Some Name'
    },
    labels: ['text']
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2,
    labels: ['edge']
  },
  {
    id: 2,
    source: 2,
    target: 3,
    labels: ['edge']
  }
];

const settings = {
  baseStyles: {
    vertex: {
      caption: { text: '${properties.name}' }
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The following shows the graph visualization using the preceding custom base style:

Figure 4-1 Using a Custom Base Style



It is important to note that some specified properties of `settings.baseStyles` (such as `color`) may get overridden if the vertices or edges have labels in it. This is due to **ruleBasedStyles** getting automatically generated which applies distinct colors to vertices and edges belonging to each label.

You can still enforce the styling of your choice by fetching the `ruleBasedStyles` using `getCurrentRuleBasedStyles` method and overriding the default styles (those with `isDefaultRule` property **true**) as follows:

- Updating the default ruleBasedStyles `stylingEnabled` property to `false`. This will disable the generated ruleBasedStyles and cause the styles defined in `settings.baseStyles` to apply.
- Updating the properties of the default ruleBasedStyles to reflect the styling you require.

See Default Legend Styles for more information.

4.2 Default Legend Styles

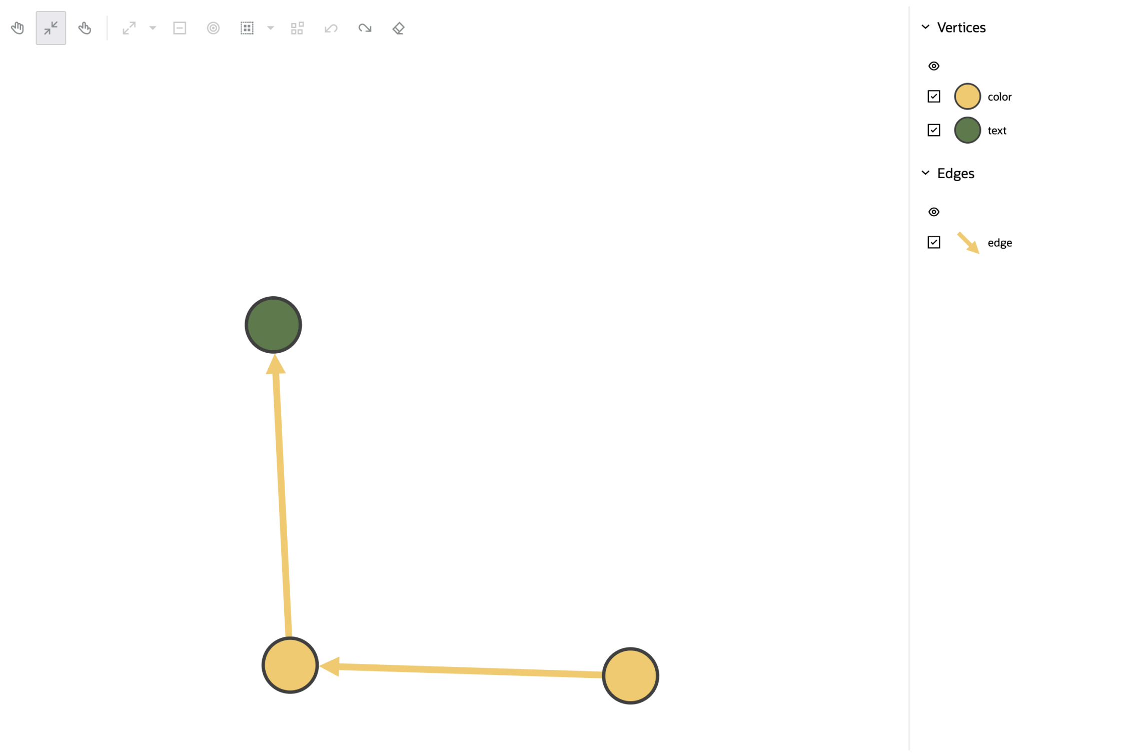
When the vertices or edges include labels, then the corresponding legend entries are automatically generated based on those labels. However, if the vertex has a property named `caption` or `title`, that property's value is used as the text shown for that vertex in the default legend style.

The following example describes the graph data for vertices and edges as shown:

```
const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello'
    },
    labels:['color']
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World'
    },
    labels:['color']
  },
  {
    id: 3,
    properties: {
      name: 'Some Name'
    },
    labels:['text']
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2,
    labels: ['edge']
  },
  {
    id: 2,
    source: 2,
    target: 3,
    labels: ['edge']
  }
];

const settings = {baseStyles: {}};

const graphViz = new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The legends are then generated from the vertex and edge labels (used in the preceding code) as shown:

Figure 4-2   Default Legends

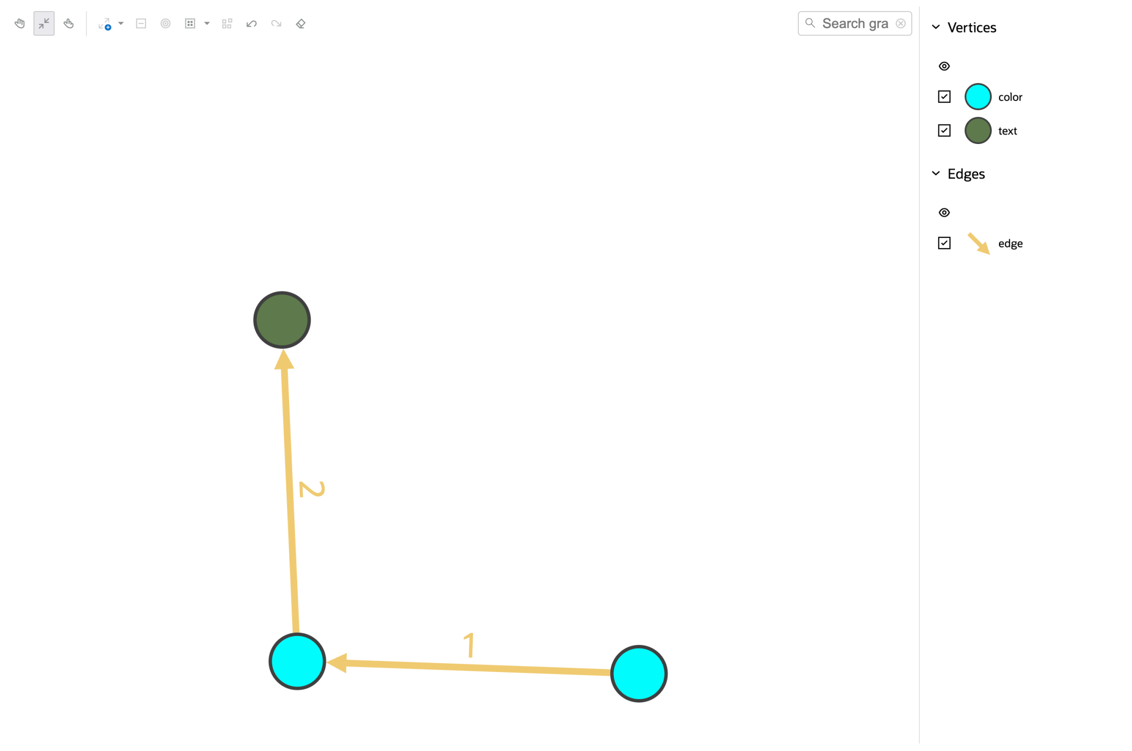


The `getCurrentRuleBasedStyles` function returns the currently defined rule-based styles, including both default and custom styles. You can use this function to change the default rule-based styles as shown in the following example:

```
const graphViz = new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});

const receivedRules = grpahViz.getCurrentRuleBasedStyles();

/*
Here is example of receivedRules. If the rule is default, rule.isDefault is
true.
[
  {
    "_id": "2",
    "stylingEnabled": true,
    "target": "vertex",
    "conditions": {
      "operator": "and",
      "conditions": [
        {
          "property": "labels",
          "operator": "~",
          "value": "color"
        }
      ]
    },
    "component": "vertex",
    "style": {
      "color": "#F0CC71"
    },
    "legendTitle": "color",
    "isDefaultRule": true
  },
  ...
]
*/

//Modify receivedRules. Note: Do not edit _id, isDefaultRule, and conditions.
//Modify _id = 2 default rule, color to aqua
receivedRules = [
  {
    "_id": "2",
    "stylingEnabled": true,
    "target": "vertex",
    "conditions": {
      "operator": "and",
      "conditions": [
        {
          "property": "labels",
          "operator": "~",
          "value": "color"
        }
      ]
    },
    "component": "vertex",
    "style": {
      "color": "aqua"
    },
    "legendTitle": "color",
    "isDefaultRule": true
  },
  ...
]

//Assgin the modified received rules to settings.ruleBasedStyles
settings.ruleBasedStyles = recivedRules;
graphViz.$set({setting});
```

The updated styles are then reflected in the legend panel as shown:

Figure 4-3 Custom Legends



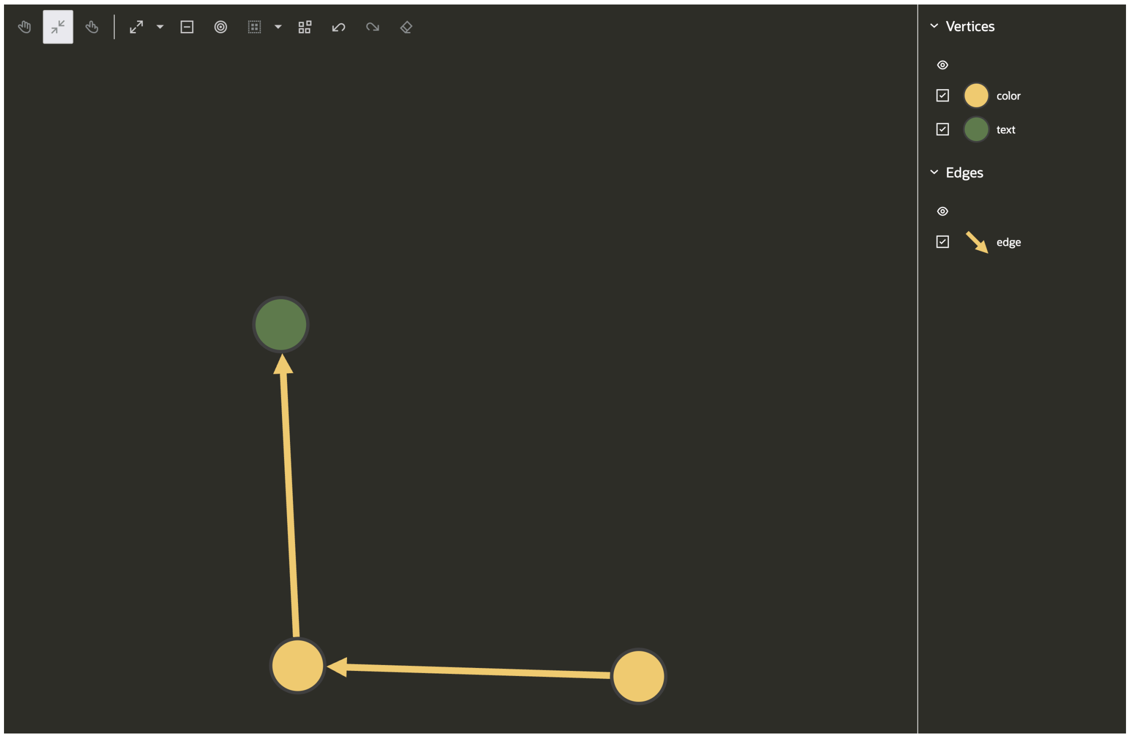
4.3 Themes

You can enable a dark theme through settings as shown:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello'
    },
    labels:['color']
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World'
    },
    labels:['color']
  },
  {
    id: 3,
    properties: {
      name: 'Some Name'
    },
    labels:['text']
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2,
    labels: ['edge']
  },
  {
    id: 2,
    source: 2,
    target: 3,
    labels: ['edge']
  }
];

const settings = {
  theme: 'dark',
  baseStyles: {
    vertex: {
      label: { text: '${properties.name}' }
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The corresponding visualization appears as shown:

Figure 4-4 Applying Dark Theme

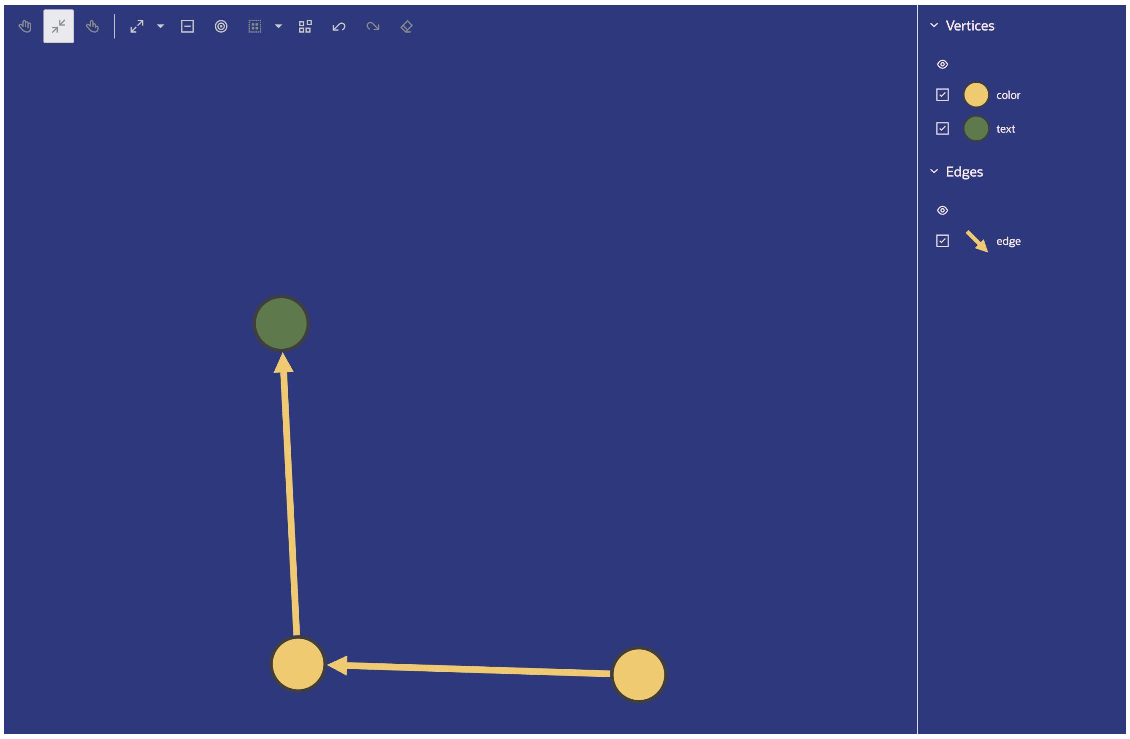


You can also create a customized theme to modify the background and foreground colors.

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello'
    },
    labels:['color']
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World'
    },
    labels:['color']
  },
  {
    id: 3,
    properties: {
      name: 'Some Name'
    },
    labels:['text']
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2,
    labels: ['edge']
  },
  {
    id: 2,
    source: 2,
    target: 3,
    labels: ['edge']
  }
];

const settings = {
  customTheme: {
    'backgroundColor': '#2F3C7E',
    'textColor': '#FBEAEB'
  },
  baseStyles: {
    vertex: {
      label: { text: '${properties.name}' }
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The custom theme gets applies as shown:

Figure 4-5   Applying Custom Theme



Also, note the following:

- If both dark theme and custom theme are applied simultaneously, the colors defined in the custom theme will take precedence over the dark theme colors.
- If the settings specify a label color, the label will use the color from the label settings rather than the color from the theme settings.

4.4 Children

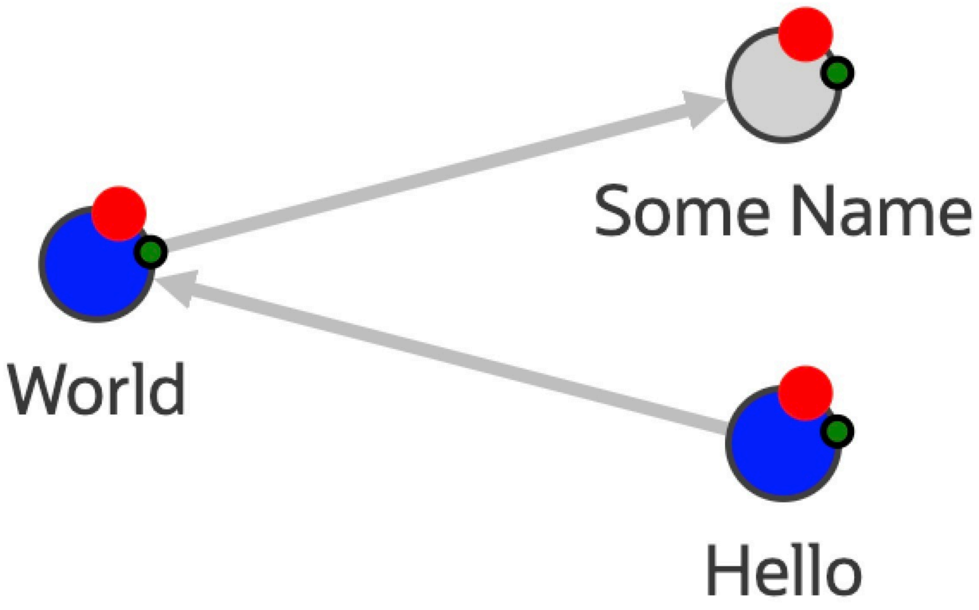
You can use the `children` attribute to create children nodes that appear on the circumference of the nodes where they are indicated. Styles for the children nodes are applied similarly to the parent nodes.

Consider the following example:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello'
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World'
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name'
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {
  baseStyles: {
    vertex: {
      label: { text: '${properties.name}' },
      // This would add two children to every vertex, any name can be
assigned to these children nodes.
      children: {
        firstChild: {
          size: '4',
          color: 'red',
          children: {
            size: '2'
          }
        },
        secondChild: {
          size: '2',
          color: 'green',
          border: {
            'width': 1,
            'color': 'black'
          }
        }
      }
    }
  }
};

settings.ruleBasedStyles = [{
  component: 'vertex',
  target: 'vertex',
  stylingEnabled: true,
  conditions: {
    conditions: [{
      property: "label",
      operator: "=",
      value: "blue"
    }]
  },
  style: { color: 'blue' }
}];

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The corresponding visualization appears as shown:

Figure 4-6 Visualizing Children Nodes



4.5 Interpolation

Interpolation can be applied to the size or color of the vertices or edges. The following interpolation types are supported:

4.5.1 Linear Interpolation

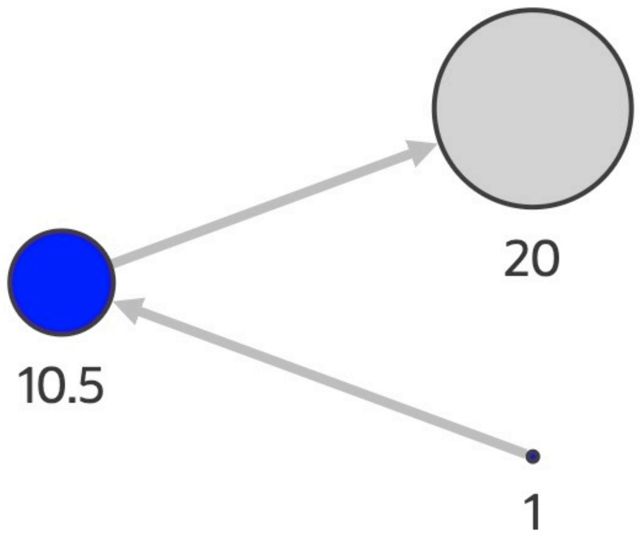
The default linear interpolation can be used to define the size of nodes or edges within a range using a property value to interpolate in the given range.

Consider the following example:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World',
      age: 20
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name',
      age: 30
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {
  ruleBasedStyles: [{
    component: 'vertex',
    target: 'vertex',
    stylingEnabled: true,
    conditions: {
      conditions: [{
        property: "label",
        operator: "=",
        value: "blue"
      }]
    },
    style: { color: 'blue' }
  }],
  baseStyles: {
    vertex: {
      // The label is changed to see the size of the node on it.
      label: { text: '${interpolate("properties.age", 1, 20)}' },
      // The size will be defined by the interpolation of properties.age in
the range of 1 -> 20.
      size: '${interpolate("properties.age", 1, 20)}'
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings}
});
```

The corresponding visualization appears as shown:

Figure 4-7 Normal Linear Interpolation

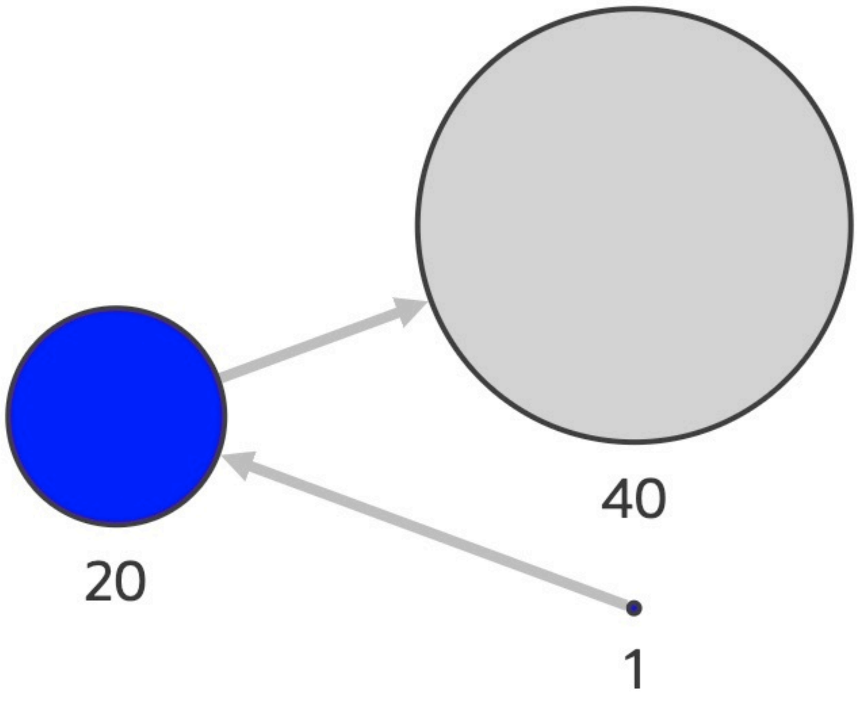


Alternatively, you can also use multiple values for interpolation instead of just using one range.

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World',
      age: 20
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name',
      age: 30
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {
  ruleBasedStyles: [{
    component: 'vertex',
    target: 'vertex',
    stylingEnabled: true,
    conditions: {
      operator: 'and',
      conditions: [{
        property: "label",
        operator: "=",
        value: "blue"
      }]
    },
    style: { color: 'blue' }
  }],
  baseStyles: {
    vertex: {
      // The label is changed to see the size of the node on it.
      label: { text: '${interpolate("properties.age", 1, 20, 40)}' },
      // The size will be defined by the interpolation of properties.age 
using the values of 1, 20, 40.
      size: '${interpolate("properties.age", 1, 20, 40)}'
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings}
});
```

The visualization for the preceding settings appear as shown:

Figure 4-8 Linear Interpolation for a Range of Values



4.5.2 Discrete Interpolation

Discrete interpolation can be used to define the size of vertices or edges within a defined range using a property as the value to interpolate in the given range. Unlike linear interpolation, the resulting values can only be the exact start or end value of the range. If the property value falls in the first half between the minimum and maximum values, it will be rounded up; otherwise, it will be rounded down.

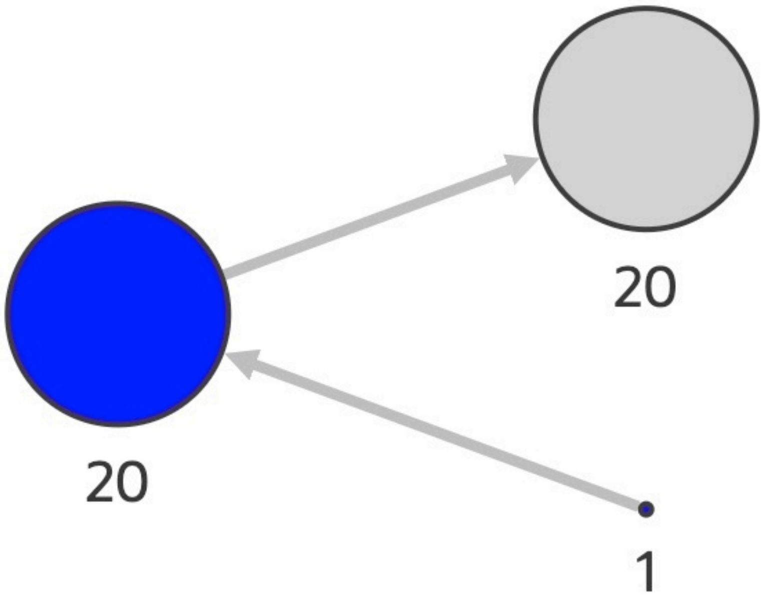
Consider the following example:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World',
      age: 20
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name',
      age: 30
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {
  ruleBasedStyles: [{
    component: 'vertex',
    target: 'vertex',
    stylingEnabled: true,
    conditions: {
      conditions: [{
        property: "label",
        operator: "=",
        value: "blue"
      }]
    },
    style: { color: 'blue' }
  }],
  baseStyles: {
    vertex: {
      // The label is changed to see the size of the node on it.
      label: { text: '${interpolate.discrete("properties.age", 1, 20)}' },
      // The size will be defined by the interpolation of properties.age in
the range of 1 -> 20.
      // In this example since the node with age 20 is exactly in the middle,
it will be rounded up to 20.
      size: '${interpolate.discrete("properties.age", 1, 20)}'
    }
  }
};
```

```
new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings}
});
```

The corresponding graph visualization is as shown:

Figure 4-9 Discrete Interpolation



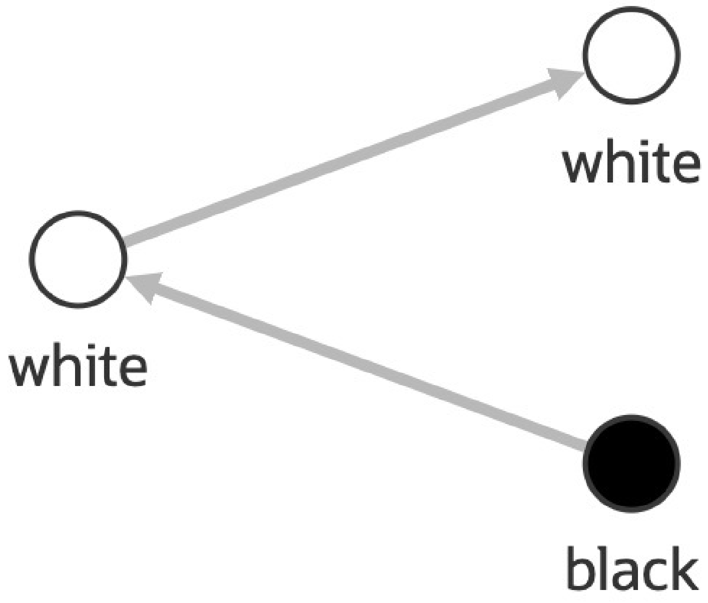
Discrete interpolation can also be applied using colors. You only need to define the colors that are to be discretely interpolated.

Consider the following example:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World',
      age: 20
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name',
      age: 30
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {
  baseStyles: {
    vertex: {
      label: { text: '${interpolate.discrete("properties.age", "black",
"white")}' },
      color: '${interpolate.discrete("properties.age", "black", "white")}'
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The corresponding visualization appears as shown:

Figure 4-10 Discrete Interpolation Using Colors



4.5.3 Color Interpolation

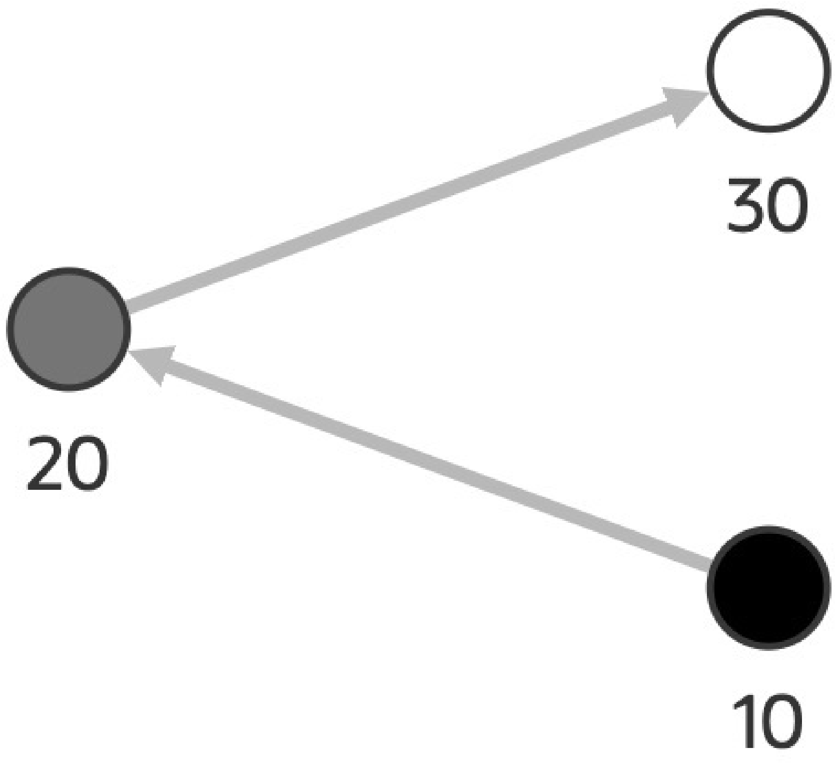
Colors can also be linearly interpolated using the `interpolate.color` function. You need to define the colors to interpolate the desired property.

Consider the following example:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World',
      age: 20
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name',
      age: 30
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {
  baseStyles: {
    vertex: {
      label: { text: '${properties.age}' },
      color: '${interpolate.color("properties.age", "black", "white")}'
    }
  }
};

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The corresponding visualization appears as shown:

Figure 4-11 Color Interpolation



4.6 Rule-Based Styles

Rule-based styles can be applied on any vertex or edge property values. You can define a rule-based styling using one or more defined properties. The set condition is verified and the vertices or edges are filtered based on the given condition.

The following operators can be used to determine if the property value matches the set rule: =, >, <, >=, <=, !=, and ~.

Consider the following example which describes a rule to color vertices with *person* label in *blue* and vertices having *first_name* = *'John'* in *red*.

```
const vertices = [
  {
    id: 1,
    properties: {
      first_name: 'Angel',
      last_name: 'Rodriguez'
      age: 10
    }
    labels: ['person']
  },
  {
    id: 2,
    properties: {
      first_name: 'John',
      last_name: 'Anderson'
      age: 20
    },
    labels: ['person']
  },
  {
    id: 3,
    properties: {
      first_name: 'Matt',
      last_name: 'Judge'
      age: 30
    },
    labels: ['person']
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {};
settings.baseStyles = {
  vertex: {
    label: { text: '${properties.name}' }
  }
};

settings.ruleBasedStyles = [
  {
    // Rule's target
    target: 'vertex',
    component: 'vertex',
    // The conditions on which the filter will be applied
    conditions: {
      conditions: [
        // This condition will apply to all the vertices that has 'person' on
their labels
        {
          property: "labels",
          operator: "CONTAINS",
          value: "person"
        }
      ]
    },
    // The title for the filter that will show in the legend
    legendTitle: 'Rule by labels',
    // The colors to apply to the nodes that match the rule
    style: {
      color: 'blue'
    },
    stylingEnabled: true
  },
  {
    // Rule's target
    target: 'vertex',
    component: 'vertex',
    // The conditions on which the filter will be applied
    conditions: {
      conditions: [
        // This condition will verify if the property first_name on a vertex
is equals to 'John'
        {
          property: "first_name",
          operator: "=",
          value: "John"
        }
      ]
    },
    // The title for the filter that will show in the legend
    legendTitle: 'Rule by property named first_name being equals to John',
    // The colors to apply to the nodes that match the rule
    style: {
      color: 'red'
    },
    stylingEnabled: true
  }
];

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

Rule-based styles can also be applied to adjust the size of nodes. Also, you can define a rule to match multiple conditions simultaneously. These conditions can be configured using `and` or `or` operators. In such a case, filtering is applied only when all the specified conditions are met for the `and` operator, or when any one of the conditions is satisfied for the `or` operator.

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      first_name: 'Angel',
      last_name: 'Rodriguez'
      age: 10
    }
    labels: ['person']
  },
  {
    id: 2,
    properties: {
      first_name: 'John',
      last_name: 'Anderson'
      age: 20
    },
    labels: ['person']
  },
  {
    id: 3,
    properties: {
      first_name: 'Matt',
      last_name: 'Judge'
      age: 30
    },
    labels: ['person']
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {};
settings.baseStyles = {
  vertex: {
    label: {text: '${properties.name}'}
  }
};

settings.ruleBasedStyles = [
  {
    // The target in the rule.
    target: 'vertex',
    component: 'vertex',
    // The conditions on which the rule will be applied.
    conditions: {
      operator: 'and',
      conditions: [
        // This condition will verify that the name contains the letter o.
        {
          property: "name",
          operator: "~",
          value: "o"
        },
        // This condition will verify that the name contains the letter l.
        {
          property: "name",
          operator: "~",
          value: "l"
        }
      ]
    },
    // The title for the filter that will show in the legend.
    legendTitle: 'Rule by name',
    // The colors to apply to the nodes that match the rule.
    style: {
      size: 15
    },
    stylingEnabled: true
  }
];

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

4.7 Vertex Shapes

Vertices are rendered as circles by default. The `shape` style property allows using alternative geometries either by passing a shape keyword or by providing additional configuration for more control. This property is available anywhere vertex styles can be specified (for example in `settings.baseStyles.vertex`, `settings.baseStyles['vertex:group']`, or within `settings.ruleBasedStyles`).

Supported values are *circle*, *ellipse square*, *rectangle*, *roundedRectangle*, *triangle*, *diamond*, *pentagon*, *hexagon*, and *octagon*.

The following describes a few additional configurations:

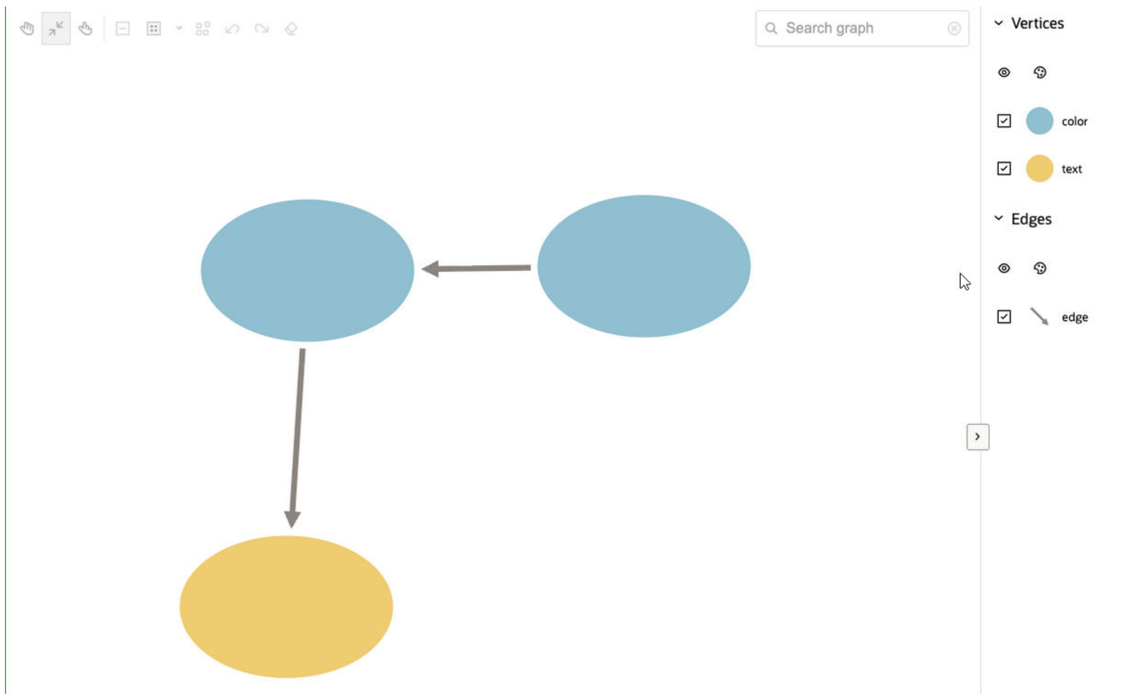
- `ellipse`, `rectangle`, and `roundedRectangle` can set `widthHeightRatio` to control aspect ratio.
- `roundedRectangle` also supports `cornerRadiusRatio` (defaults to `0.1`).
- `triangle` can rotate using `direction: 'up' | 'down' | 'left' | 'right'`.
- Applying `shape: null` removes any shape override and falls back to the default `circle`.
- When combined with `caption.angle: null`, enabling `caption.resizeParent` lets the chosen shape adapt to the caption length.

The following shows an example of applying `ellipse` styling to vertices:

```
const settings = {
  baseStyles: {
    vertex: {
      shape: { type: 'ellipse', widthHeightRatio: 1.5 }
    }
  }
};
```

The preceding code produces the following output:

Figure 4-12   Applying Ellipse Styling to Vertices



4.8 Captions

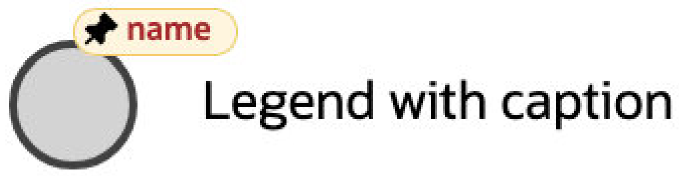
The legend panel displays caption over a legend item when the rule based style of the legend item has `label.text` specified.

The following shows a sample rule based style to display a caption.

```
{
  _id: 1683557130420,
  component: 'vertex',
  stylingEnabled: true,
  target: 'vertex',
  visibilityEnabled: true,
  conditions:{
    operator: 'and',
    conditions: []
  },
  style: {
    label: {
      text: '${properties.name}',
      color: 'brown'
    }
  },
  legendTitle: 'Legend with caption'
}
```

As shown in the preceding code, the property `name` gets extracted and is shown as caption if the `label.text` is specified as an expression using dot notation (`${properties.name}`) or bracket notation (`${properties["name"]}`. The following figure shows the caption displayed over the legend item:

Figure 4-13 Legend With Caption



The default `pin` icon shown inside the caption can be changed using the rule based style's `overlayIndicator` property as shown:

```
{
  ....
  overlayIndicator: 'bookmark',
  style: {
    label: {
      text: '${properties.name}',
      color: 'brown'
    }
  },
  legendTitle: 'Legend with caption'
}
```

4.9 Animations

Using animations. you can show dynamic movement of the graph vertices and/or edges. You can apply animations through the `settings` filter.

Consider the following example which show a graph with the vertices' stroke width animated:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';

import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
```

```
        }
      },
      {
        id: 2,
        properties: {
          label: 'blue',
          name: 'World',
          age: 20
        }
      },
      {
        id: 3,
        properties: {
          name: 'Some Name',
          age: 30
        }
      }
    ];

    const edges = [
      {
        id: 1,
        source: 1,
        target: 2
      },
      {
        id: 2,
        source: 2,
        target: 3
      }
    ];

    const settings = {};
    settings.baseStyles = {
      vertex: {
        label: { text: '${properties.name}' }
      }
    };

    settings.ruleBasedStyles= [
      {
        target: 'vertex',
        component: 'vertex',
        legendTitle: 'Vertex animation',
        animations: [[
          {
            duration: 1,
            keyFrames: [
              {
                percentage: 0,
                style: {
                  strokeWidth: '3px'
                }
              },
              {
                percentage: 50,
                style: {
                  strokeWidth: '7px'
                }
              },
              {
                percentage: 100,
                style: {
                  strokeWidth: '3px'
                }
              }
            ]
          }
        ]],
        conditions: {
          // This rule is applied to all vertices.
          conditions: []
        }
      }
    ];

    new GraphVisualization({
      target: document.body,
      props: { data: { vertices, edges }, settings }
    });
```

You can configure the animation using the following values:

- `duration`: Defines the duration of the animation in seconds.
- `keyframes`: An array representing all the changes that have to be applied to the entity during the animation.

The `keyframes` properties that need to be provided are:

- `percentage`: Represents at what percentage of the animation duration should the keyframe be applied. To generate smooth animations:
  - **Multiple keyframes**: Values must start from zero and end in 100.
  - **Single keyframe**: Percentage value must be 100.

  Note that this option is only meaningful when working with `strokeDashoffset`.
- `style`: Styles that need to be applied to the entity at each keyframe.

The supported style properties are:

- `strokeWidth`: Defines the width of the stroke that surrounds the vertices and also the width of the edges. This can be passed as any valid `css` value (`px` is recommended).
- `stroke`: Defines the color of the stroke that surrounds the vertices and the edges.
- `opacity`: Defines the opacity on a scale of 0 to 1; 0 indicates that the element is completely hidden, while 1 signifies that the element is fully visible with maximum opacity.
- `r` *(applies only for the vertices)*: Defines the radius of the vertices to which it is applied.
- `strokeDashoffset` *(applies only for the edges)*: Defines the amount of offset that has to be applied to the dashed pattern on the edges. Negative values make the offset go in the opposite direction. Note that you must apply the dashed pattern to the edges for this animation to be visible. Otherwise, nothing will appear on the graph.

The following example describes how to apply edge animation using `strokeDashoffset`:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';

import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello',
      age: 10
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World',
      age: 20
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name',
      age: 30
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {};

settings.baseStyles = {
  vertex: {
    label: { text: '${properties.name}' }
  }
};

settings.ruleBasedStyles = [
  {
    target: 'edge',
    component: 'edge',
    legendTitle: 'Edge animation',
    style: {
      dasharray: 'dashed'
    },
    animations: [
      [
        {
          duration: 1,
          keyFrames: [
            {
              percentage: 100,
              style: {
                strokeDashoffset: 50
              }
            }
          ]
        }
      ]]
    ,
    conditions: {
      conditions: []
    }
  }
];

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

4.10 Icons

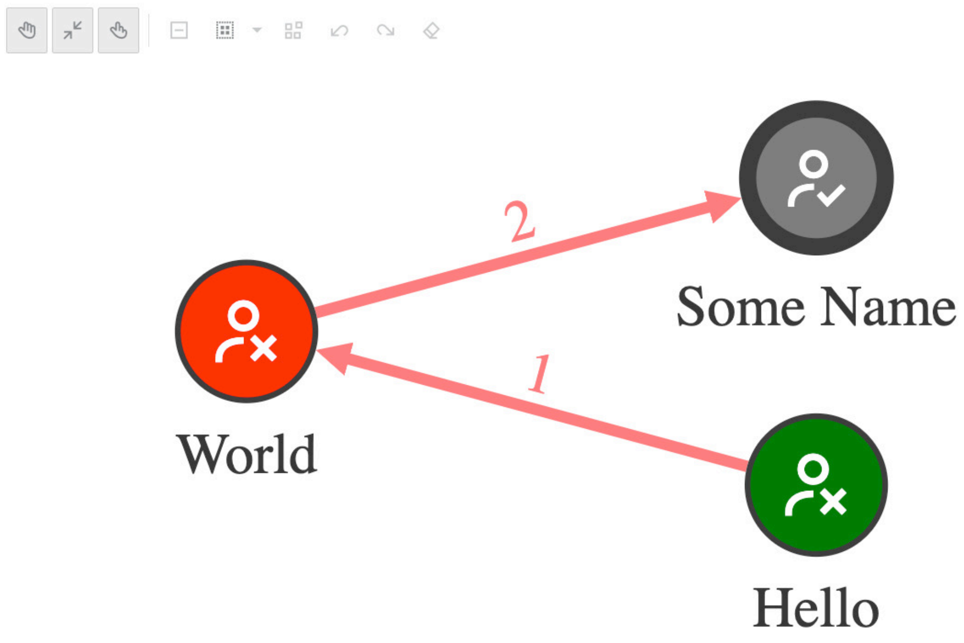
The Graph Visualization Toolkit supports Redwood as native icon library.

Consider the following example using the icon library:

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const vertices = [
  {
    id: 1,
    properties: {
      label: 'blue',
      name: 'Hello'
    }
  },
  {
    id: 2,
    properties: {
      label: 'blue',
      name: 'World'
    }
  },
  {
    id: 3,
    properties: {
      name: 'Some Name'
    }
  }
];

const edges = [
  {
    id: 1,
    source: 1,
    target: 2
  },
  {
    id: 2,
    source: 2,
    target: 3
  }
];

const settings = {};
settings.baseStyles = {
  // Style applies for all the vertices.
  vertex: {
    size: 12,
    label: '${properties.name}',
    color: 'red',
    icon: 'oj-ux-ico-user-not-available',
  },

  "vertex:hover": {
    size: '${previous + 4}'
  },
  // Style applies for all the edges.
  "edge": {
    label: '${id}',
    color: '#FF8080'
  }
};

settings.ruleBasedStyles = [
  {
    component: 'vertex',
    target: 'target',
    conditions: {
      conditions: [
        {
          property: 'name',
          operator: '=',
          value: 'Hello'
        }
      ]
    },
    style: {
      color: 'green'
    },
    stylingEnabled: true,
  },
  {
    component: 'vertex',
    target: 'target',
    conditions: {
      conditions: [
        {
          rule: 'id % 3 === 0'
        }
      ]
    },
    style: {
      color: 'gray',
      icon: { class: 'oj-ux-ico-user-available' }
    },
    stylingEnabled: true,
    legendTitle: 'advanced conditions'
  }
];

new GraphVisualization({
  target: document.body,
  props: { data: { vertices, edges }, settings }
});
```

The resulting graph visualization is as shown:

Figure 4-14   Using Redwood Icons in Graph Visualization



4.11 Graph Schema Visualization

You can visualize the underlying database schema for your property graph using the Graph Visualization library.

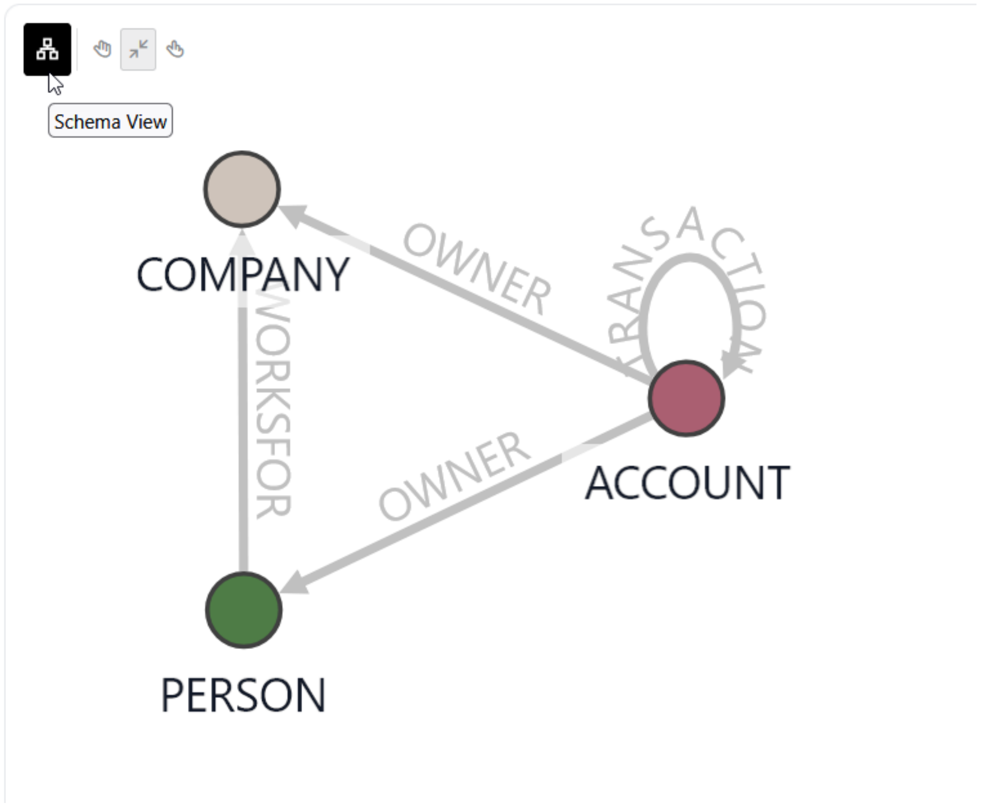
The following shows an example JSON configuration for a schema view:

```
const schema = {
  "vertices": [
    {
      "labels": [
        "COMPANY"
      ],
      "properties": [
        {
          "name": "ID",
          "dataType": "number",
          "limits": [],
          "mandatory": false
        },
        {
          "name": "NAME",
          "dataType": "string",
          "limits": [
            15
          ],
          "mandatory": false
        }
      ]
    },
    {
      "labels": [
          "PERSON"
      ],
      "properties": [
        {
          "name": "ID",
          "dataType": "number",
          "limits": [],
          "mandatory": false
        },
        {
          "name": "NAME",
          "dataType": "string",
          "limits": [
            15
          ],
          "mandatory": false
        },
        {
          "name": "AMOUNT",
          "dataType": "number",
          "limits": [],
          "mandatory": false
```

```
   }
  ]
 },
 {
  "labels": [
    "ACCOUNT"
  ],
  "properties": [
   {
    "name": "NUMBER",
    "dataType": "number",
    "limits": [],
    "mandatory": false
   }
  ]
 }
],
"edges": [
 {
  "sourceVertexLabels": [
   "ACCOUNT"
  ],
  "targetVertexLabels": [
   "ACCOUNT"
  ],
  "labels": [
   "TRANSACTION"
  ],
  "properties": [
   {
    "name": "AMOUNT",
    "dataType": "number",
    "limits": [],
    "mandatory": false
   }
  ]
 },
 {
  "sourceVertexLabels": [
   "ACCOUNT"
  ],
  "targetVertexLabels": [
   "PERSON"
  ],
  "labels": [
   "OWNER"
  ],
  "properties": []
 },
 {
  "sourceVertexLabels": [
   "ACCOUNT"
  ],
  "targetVertexLabels": [
   "COMPANY"
  ],
      "labels": [
        "OWNER"
      ],
      "properties": []
    },
    {
      "sourceVertexLabels": [
        "PERSON"
      ],
      "targetVertexLabels": [
        "COMPANY"
      ],
      "labels": [
        "WORKSFOR"
      ],
      "properties": []
    }
  ]
}

const instance = new GraphVisualization({
  target: vizContainer,
  props: {
    schemaSettings: { layout: "force" }
  }
});

return instance;
```

The corresponding schema visualization is as shown:

Figure 4-15 Visualizing Database Schema for the Property Graph



4.12 Saved Conditional Searches

The conditional search user interface optionally supports **saved searches** (list of saved conditional search configurations) when you provide:

- `savedSearchProvider`: Callbacks for persistence operations
- `savedSearches`: List that is displayed in the user interface.

If `savedSearchProvider` is not provided, `savedSearches` is disabled.

In the following example, saved-search support is enabled by providing both `savedSearches` and `savedSearchProvider` to the conditional search functionality in the user interface. The sample code uses `localStorage` as the data store. The `upsert` and `remove` implementations update and delete entries, respectively, persist the result, and then refresh `savedSearches` and the component so the user interface remains synchronized with the persisted state.

```
// This import is not necessary if you are using Oracle JET.
import '@ovis/graph/alta.css';
import Visualization from '@gvt/graphviz';

const SAVED_KEY = 'my-app-saved-searches';

function loadSavedSearches() {
  try {
    const parsed = JSON.parse(localStorage.getItem(SAVED_KEY) || '[]');
    return Array.isArray(parsed) ? parsed : [];
  } catch {
    return [];
  }
}

function persistSavedSearches(list) {
  localStorage.setItem(SAVED_KEY, JSON.stringify(list));
}

let savedSearches = loadSavedSearches();

let graphViz;

const savedSearchProvider = {
  // Optional: the UI does not call this directly; it can be used by your app
to fetch initial state.
  list: async () => loadSavedSearches(),

  upsert: async (config) => {
    const list = loadSavedSearches();
    const idx = list.findIndex((s) => s.name === config.name);

    if (idx >= 0) {
      list[idx] = { ...list[idx], ...config };
    } else {
      list.push({ ...config });
    }

    persistSavedSearches(list);

    // Keep the UI list in sync
    savedSearches = list;
    graphViz?.$set?.({ savedSearches });
  },

  remove: async (idOrName) => {
    const list = loadSavedSearches().filter((s) => s.name !== idOrName);
    persistSavedSearches(list);

    // Keep the UI list in sync
    savedSearches = list;
    graphViz?.$set?.({ savedSearches });
  }
};

graphViz = new GraphVisualization({
  target: document.body,
  props: {
    // ...other props like data/settings...
    savedSearches,
    savedSearchProvider
  }
});
```

Index