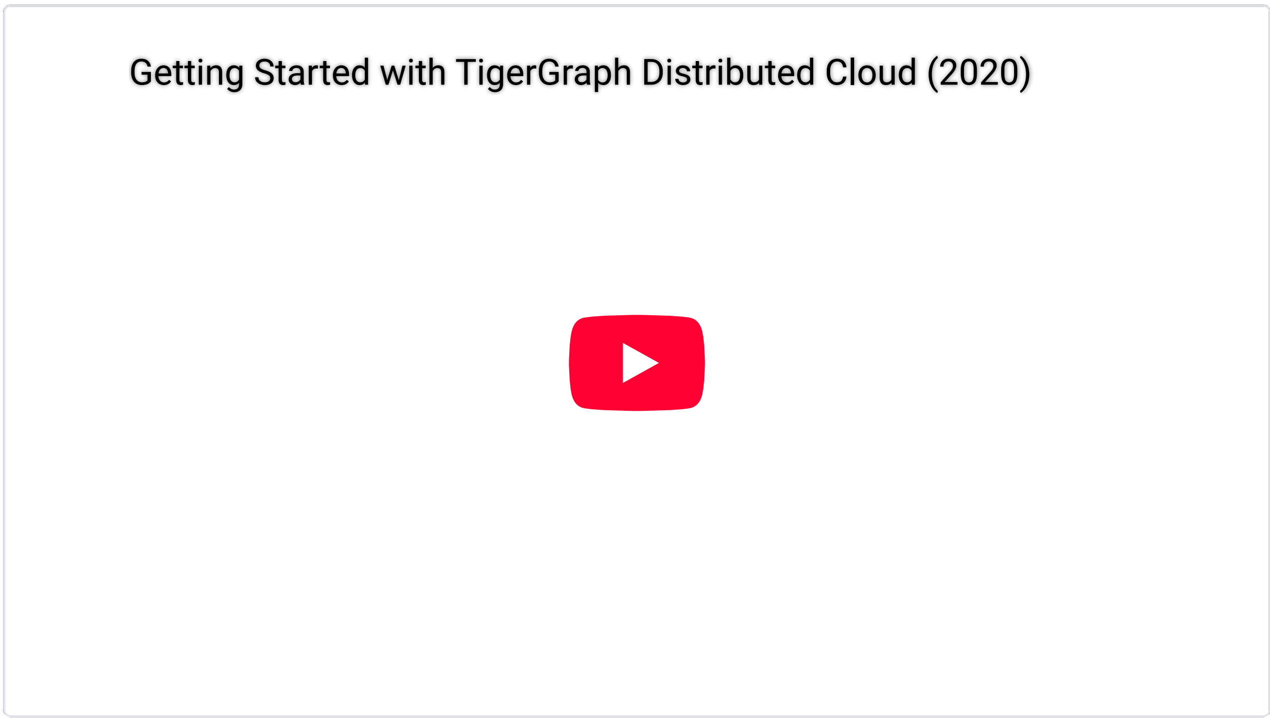
3.2

TigerGraph Documentation

To switch to a different version, select the version you want at the top of the menu on the left. For documentation of TigerGraph versions prior to 2.2, please contact TigerGraph Support.

Introducing: TigerGraph Distributed Cloud

- Learn more and sign up: www.tigergraph.com/cloud ↗
- Documentation: TigerGraph Cloud
- Watch Quick Start Video ↗ and Start Your TigerGraph Cloud Solutions in 5 Minutes!



TigerGraph Cloud Quickstart

TigerGraph Server 3.2

Major Sections	Quick Links
Developers Community	TigerGraph Community Forum ↗

FAQs and Troubleshooting	Release Notes - TigerGraph Server 3.2 Change Log Knowledge Base and FAQs Troubleshooting Guide Error Codes
Get Started	HW & SW Requirements Platform Installation GET STARTED Tutorial GSQL 101, GSQL 102
Platform Overview	Comparing TigerGraph Editions GSQL Graph Algorithm Library MultiGraph Overview TigerGraph Internal Architecture Transaction Processing and ACID Support
GSQL	GSQL 101 GSQL 102 Pattern Matching Accumulators Tutorial GSQL Spec - Data Definition & Loading GSQL Spec - Querying Interpreted GSQL
GraphStudio and UI	GraphStudio UI Guide Admin Portal UI Guide
Connectors and APIs	RESTPP API User Guide Kafka Loader User Guide S3 Loader User Guide Spark Connection Via JDBC Driver GSQL JSON Output Spec Connector Ecosystem ↗
Sys Mgmt and Admin	HA Cluster Configuration User Privileges and Authentication MultiGraph LDAP and Single Sign-On Data Encryption

> Backup and Restore

TigerGraph Cloud

TigerGraph Cloud is a fully-managed cloud database built on the same TigerGraph Enterprise Server engine that is delivering the fastest and most scalable graph database. TigerGraph Cloud handles all the complexity of deploying and managing your deployments on the cloud service provider of your choice (AWS, Azure, and GCP).

Most of the documentation for TigerGraph Enterprise Server also applies to TigerGraph Cloud. However, since TigerGraph Cloud does not have a command-line interface on the OS level, command-line features are not supported in the same way. In many cases, we introduced a GUI-based method for the same feature. For advanced OS-level commands, submit a support ticket for more information.

Watch Quick Start Video ↗ and Start Your TigerGraph Cloud Solutions in 5 Minutes!

Get started with TigerGraph Cloud

- Create a free TigerGraph Cloud account
- Create a free-tier TigerGraph Cloud solution
- Access your TigerGraph Cloud solution

Manage TigerGraph Cloud solutions

- Monitor a solution
- Security and user management
- Backup and restore

Billing

- Manage payment methods
- Cloud Marketplace subscriptions

FAQs

See the TigerGraph Cloud FAQs for answers to common questions.

Comparison with TigerGraph Enterprise Server

Topic	TigerGraph Enterprise Server	TigerGraph Cloud
GSQL Graph Algorithm Library	Yes	Installation not yet supported. Generated algorithms can be copied-and-pasted into the Write Queries code panel in GraphStudio
Release Notes	Yes	TigerGraph Cloud Provides Two Database Versions: V2.6.4 and V3.0.6
GSQL 101	Yes	Basic concepts still apply, most CREATE, INSTALL, LOAD, and RUN commands are replaced with the GraphStudio GUI-based approach.
GSQL 102 Pattern Matching	Yes	The same patterns can be used in queries.
MultiGraph Overview	Yes	Yes. Choose V3.0.6 in TigerGraph Cloud
Hardware and Software Requirements	Yes	N/A

Installation and Configuration	Yes	N/A. There are a few simple steps to install a Starter Kit. See the TigerGraph Cloud FAQs.
User Access Management	Yes	Yes. Choose V3.0.6 in TigerGraph Cloud for multiple users in one TigerGraph solution.
Data Encryption	Yes	N/A. TigerGraph Cloud is setup already for encrypted data at rest and at motion.
System Management	Yes	The Cloud Admin Portal is enhanced over the TigerGraph Server Admin Portal. Backup and Restore is done through the Cloud Admin Portal.
GraphStudio UI Guide	Yes	Yes
GSQL Demo Examples	Yes	Yes
GSQL Language Reference, Part 1 Data Definition and Loading	Yes	Yes. GSQL Web Shell (Paid Tiers) supports Data Definition and Loading. Data files can be imported into the filesystem through GraphStudio, or submit support request to import into the filesystem. In addition, the operations can be performed through the GraphStudio user interface.
GSQL Language Reference, Part 2 Querying	Yes	Yes
RESTPP API User Guide	Yes	Yes. Note the URL assigned to your solution.
Transaction Processing and ACID Support	Yes	Yes

Data Loader User Guides	Yes	S3 Loader can be used through the GraphStudio interface. In addition, S3 Loader and Kafka Loader

Getting Started

Overview

TigerGraph Cloud offers an easy way to analyze and query your graph data in the cloud. This tutorial guides you through creating your TigerGraph Cloud account, provisioning a free-tier TigerGraph Cloud Solution, and logging into GraphStudio.

Watch Quick Start Video ↗ and Start Your TigerGraph Cloud Solutions in 5 Minutes!

Create a TigerGraph Cloud account

Go to the TigerGraph Cloud landing page ↗.

You can create a TigerGraph Cloud account with a Google account, a LinkedIn account, or an email address.

Email

1. Click **Login/Register**.
2. Click **Sign Up**.
3. Enter your email and password.
4. A verification email with the subject "Verify your email" will be sent to the email address you provided.
5. Go to your email inbox. Find the verification email and click **Confirm my account.**

Google

1. Click **Login/Register.**
2. Click **Sign up.**

3. Click **Sign up with Google.**
4. Enter the password for your Google account.
5. Click **Next**.
6. Review and click the checkbox to accept the **Terms of Service** and the **Privacy Policy**.
7. Click **Submit**.
8. After being redirected back to TigerGraph Cloud, enter the information on the signup form and click the checkbox to accept TigerGraph's **Terms & Conditions** and **Privacy Policy**.
9. Click **Submit**.

LinkedIn

1. Click **Login/Register.**
2. Click **Sign up.**
3. Click Sign up with LinkedIn.
4. Enter your LinkedIn credentials and click Sign in.
5. Review the Terms and Conditions and click **Allow**.
6. After being redirected back to TigerGraph Cloud, enter the information on the signup form and click the checkbox to accept TigerGraph's **Terms & Conditions** and **Privacy Policy**.
7. Click **Submit**.

Log in to your TigerGraph Cloud account

If you already created a TigerGraph Cloud account, sign in to your account:

1. Go to the TigerGraph Cloud landing page ↗.
2. Click **Login / Register**.
3. Enter your email and password, or if you registered with an identity provider, select the correct provider and enter your credentials.

4. Click **Log In.**

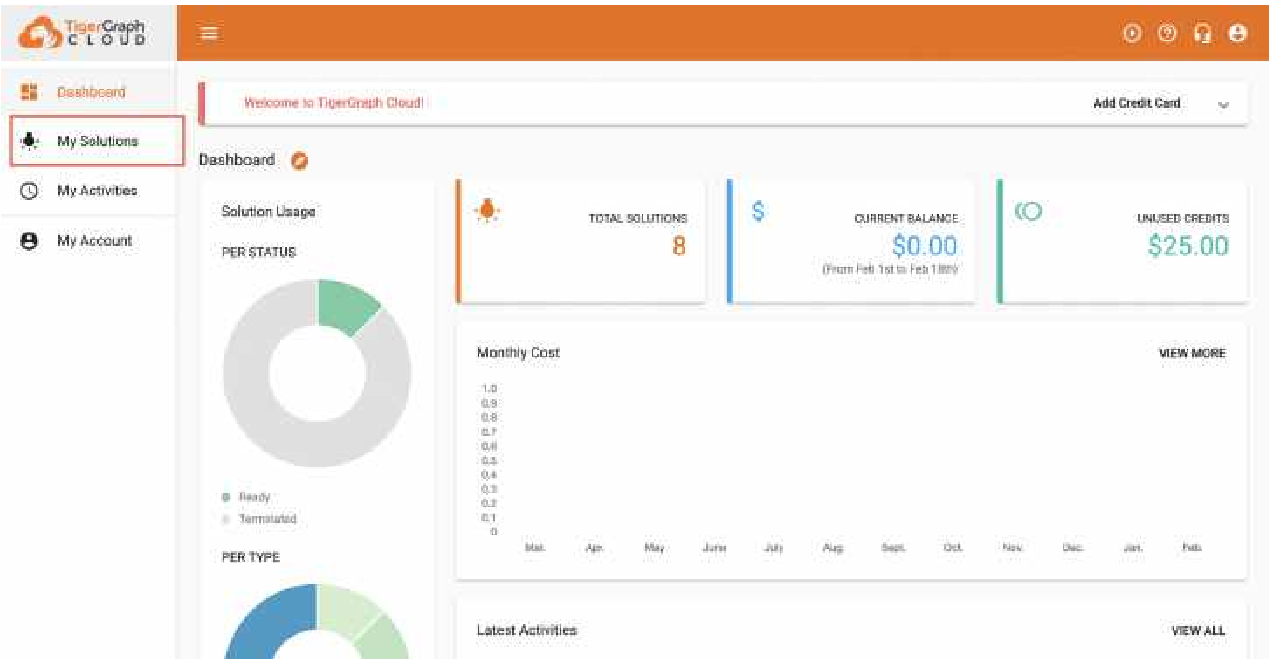
Create a free-tier solution

TigerGraph Cloud free-tier solutions (subject to limits) offer a small-scale analytics platform to store and query your graph data.

> Free-tier solutions are for training, learning, and light scale proof of concept (POC) use cases and do not support backup and restore. For in-depth POCs, consider adding a valid payment method and use paid-tier solutions to access the backup and restore feature.

1. Click My Solutions.

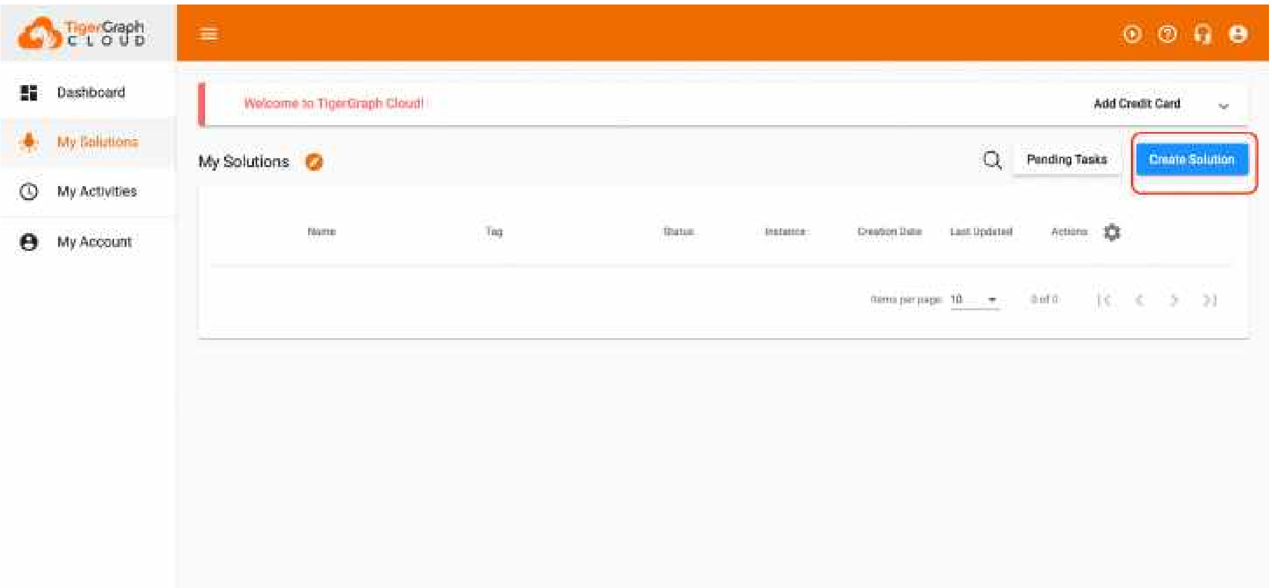
After logging in, you will be directed to the dashboard page, on the right side of the page, click **My Solutions.**



TigerGraph Cloud Dashboard view

2. Click Create Solution.

On the **My Solutions** page, click **Create Solution.**



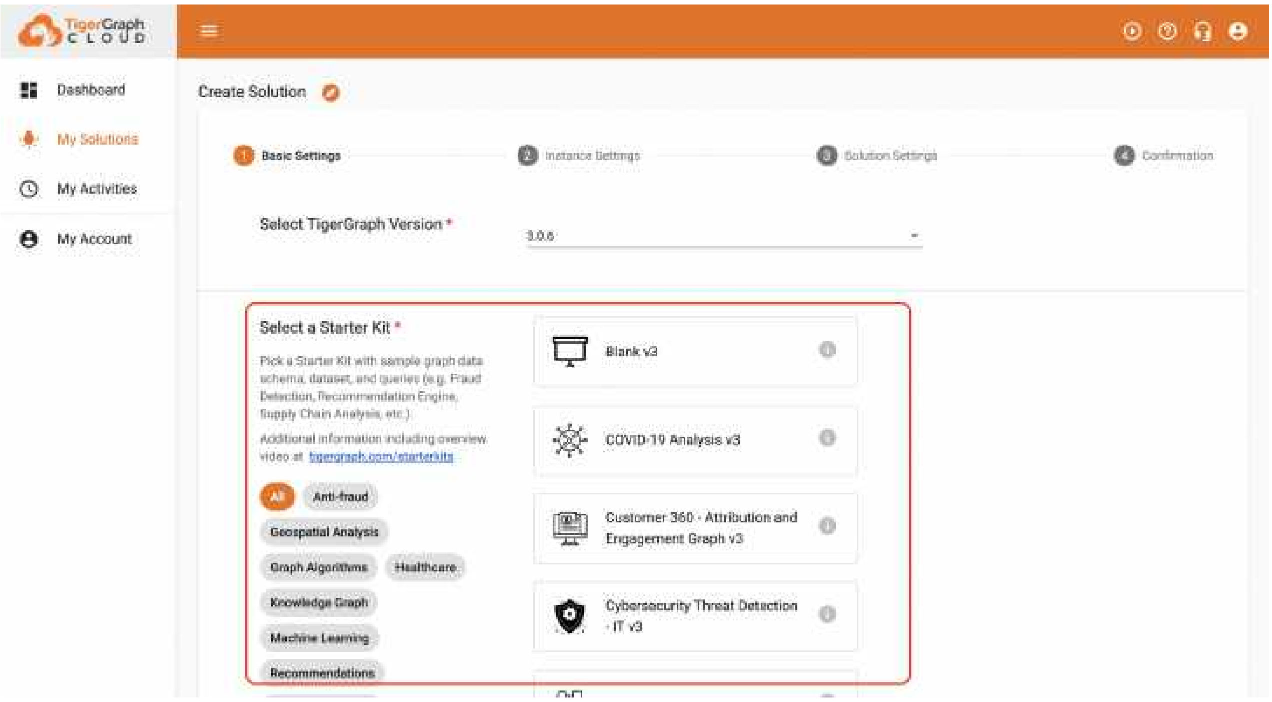
TigerGraph Cloud My Solutions view

3. Select TigerGraph version.

Not all TigerGraph versions that are offered on-prem are offered on TigerGraph Cloud. In the dropdown list, choose the version of TigerGraph you want to run.

4. Select a starter kit.

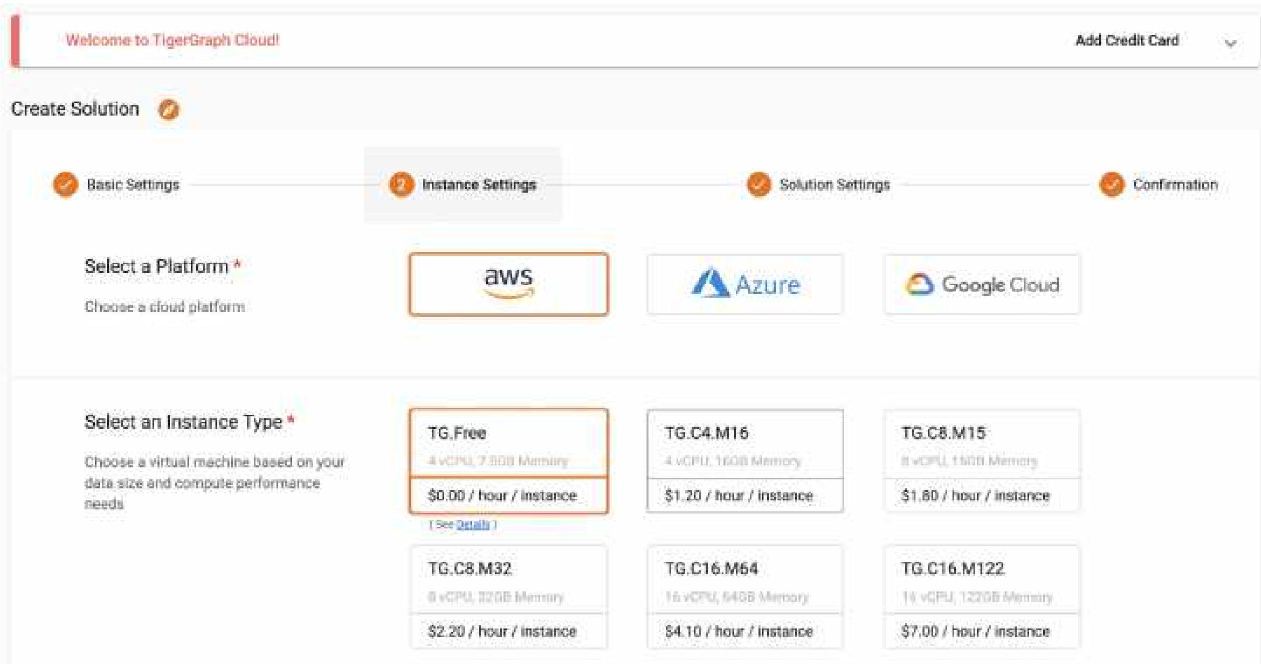
TigerGraph Cloud Starter Kits are built with sample graph data schema, dataset, and queries focused on a specific use case to help you get started quickly. Most of our starter kits have a demo video ↗ that walks you through the schema and the queries built for its use case. If you wish to proceed with your own schema and data, select **Blank.**



After you have selected a starter kit, scroll down to the bottom of the page and click **Next**.

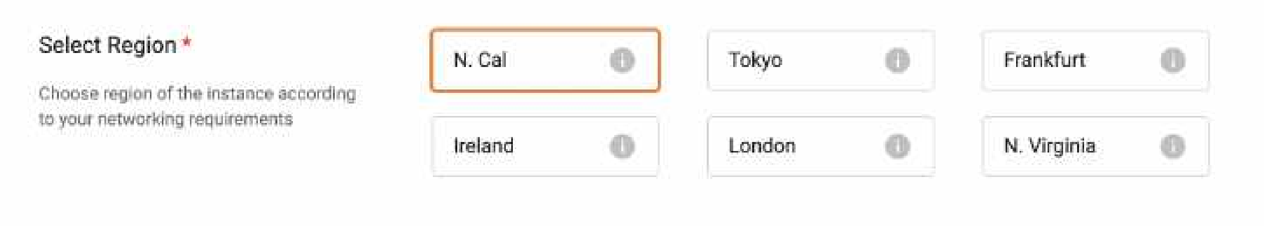
5. Select *Cloud Provider* and instance type.

Currently, TigerGraph Cloud supports free-tier solutions on AWS and GCP. Select **AWS or Google Cloud** as your cloud provider and select **TG.Free** as your instance type.



Cloud platform and instance type options

6. Select Region.



Regions for AWS

7. Select disk size.

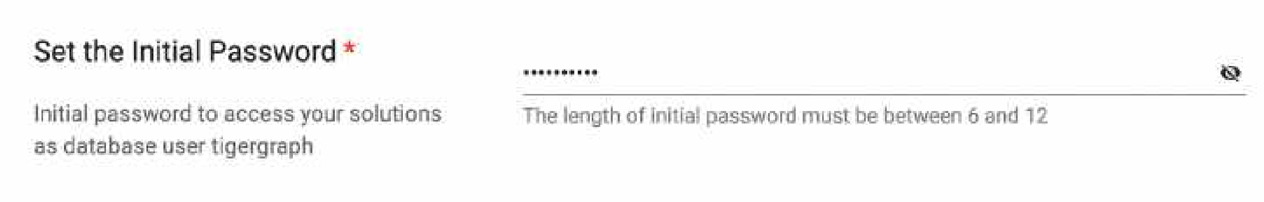
Select **50 GB** for your disk size. Scroll down to the bottom of the page and click **Next**.

8. Enter a name and a tag for your solution.

You can enter any name for your solution. A solution name may contain alphanumerics, dashes, underscores, and spaces, and cannot be longer than 20 characters. A solution tag can be no longer than 40 characters and can contain any character.

9. Set initial password.

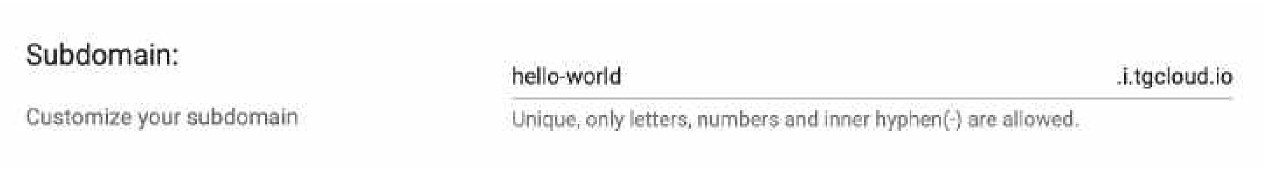
This is the initial password for the default TigerGraph user `tigergraph`. You can change the password later in Admin Portal.



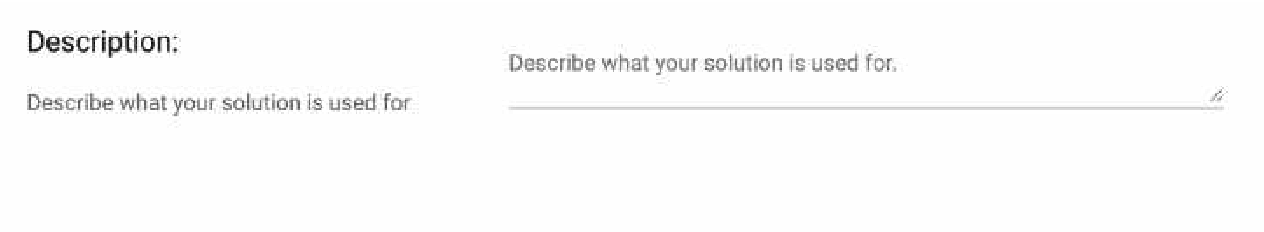
This initial password allows you to log into the database when your solution is in the ready state. If you forget this password, you will need to terminate and recreate the solution.

10. Enter a subdomain and description.

Customize the subdomain for your solution. If this field is left blank, a random subdomain will be automatically generated for the solution.



Enter an optional description of what your solution will be used for, and click **Next**.



11. Confirm your settings.

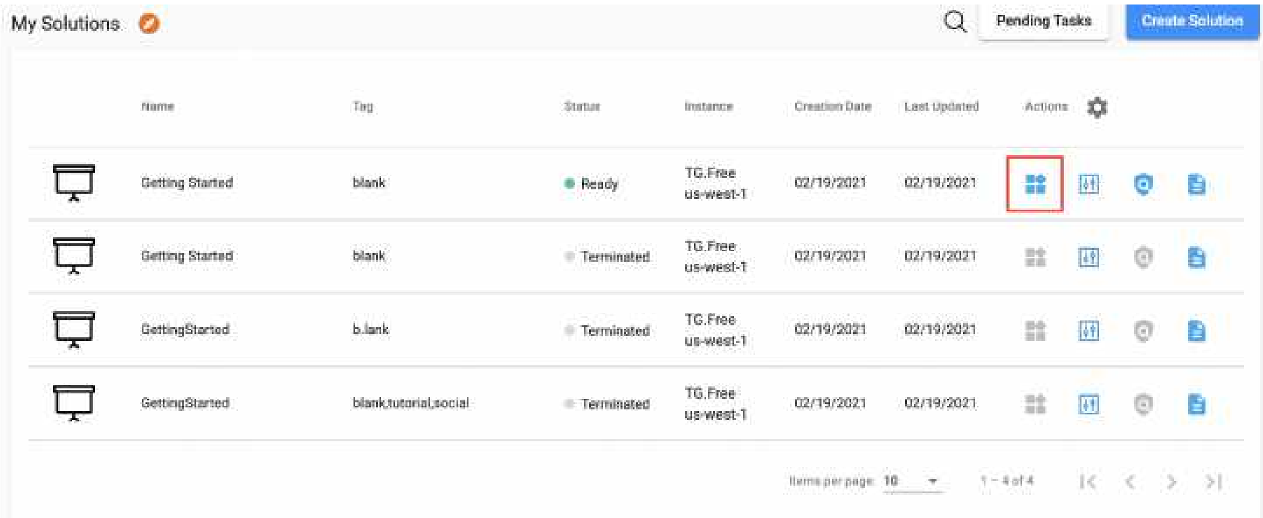
Confirm your settings and click **Submit.** You will be redirected back to the **My Solutions** view, and your solution will be ready in minutes.

Log in to GraphStudio

After you have created your solution, it may take a few minutes for it to be ready. Once you see the status of your solution turn to "Ready", you can proceed to log into GraphStudio.

1. Open GraphStudio for your solution.

Go to My Solutions. Find the solution that is ready, click the blue Applications icon, and Click **GraphStudio**.



2. Enter your credentials.

Enter the initial password set during **Create Solution** for the default database user `tigergraph` and click **Login**. This is not your TigerGraph Cloud account email and password.

GraphStudio login page

3. You are now ready to use your free-tier TigerGraph Solution.

You can learn how to use GraphStudio in our GraphStudio UI guide.

Solutions

Solutions are fully-managed TigerGraph deployments. Here are some resources you can use to create, configure and manage your solutions:

- Create a new solution
  - A step-by-step guide to creating a new solution
- Cloud Providers and Regions
  - Information on the different cloud providers and regions available on TigerGraph Cloud
- Sizing and Scaling
  - Learn how to choose the appropriate instance type and disk size
- Stop, Resume and Terminate a Solution
  - A step-by-step guide on how to stop, restart, and terminate a solution

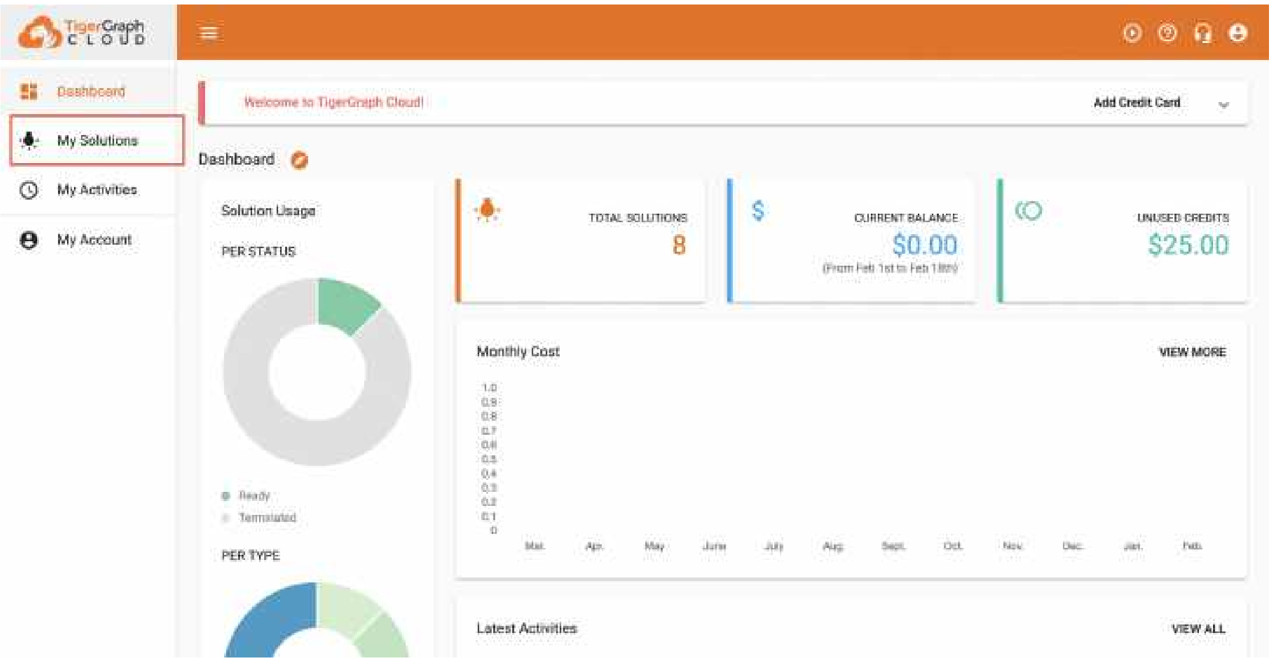
Create a New Solution

In this tutorial, you will learn how to provision a new TigerGraph Cloud solution.

Procedure

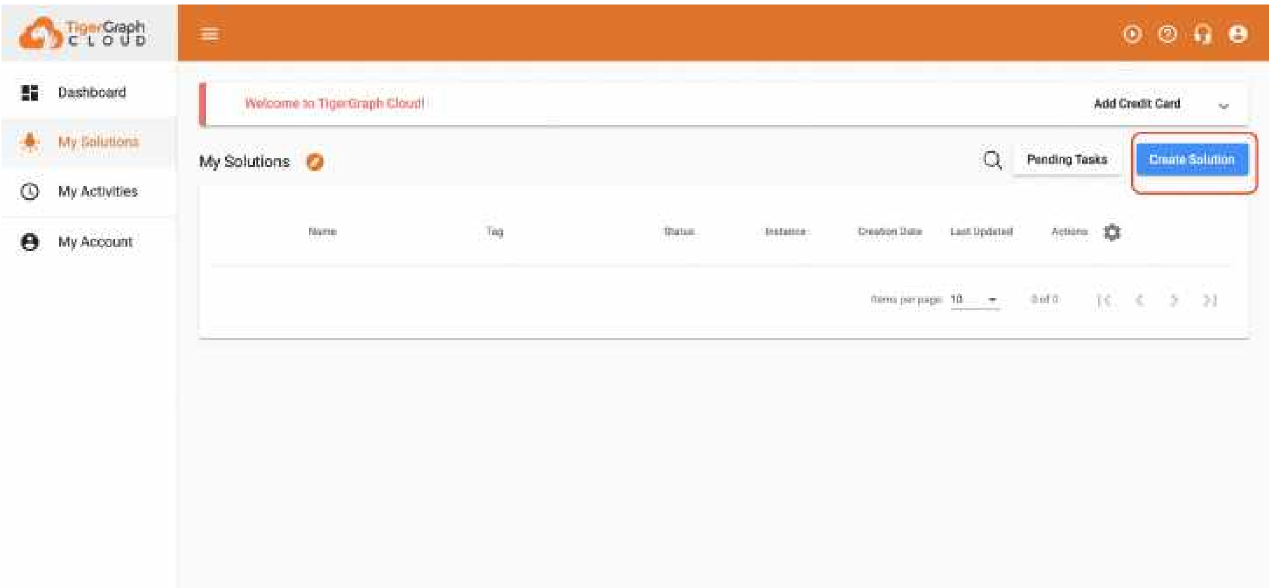
1. Navigate to My Solutions page

After logging into your TigerGraph Cloud account, click **My Solutions** on the left side menu to go to the **My Solutions** page.



2. Click Create Solution

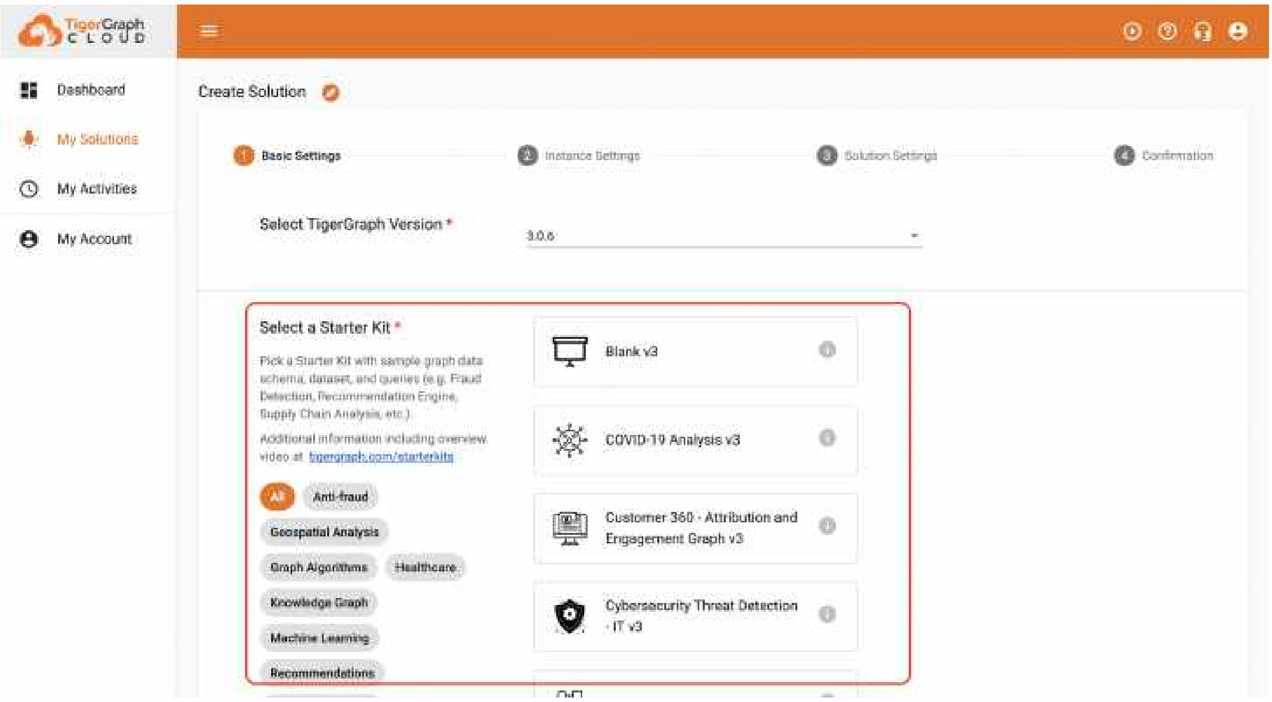
On the My Solutions page, click **Create Solution.**



3. Select version and starter kit

Not all TigerGraph versions that are offered on-prem are offered on TigerGraph Cloud. In the dropdown list, choose the version of TigerGraph you want to run.

TigerGraph Cloud Starter Kits are built with sample graph data schema, dataset, and queries focused on a specific use case to help you get started quickly. Most of our starter kits have a demo video ↗ that walks you through the schema and the queries built for its use case. If you wish to proceed with your own schema and data, select **Blank.**



After choosing the starter kit, scroll down to the bottom and click **Next**.

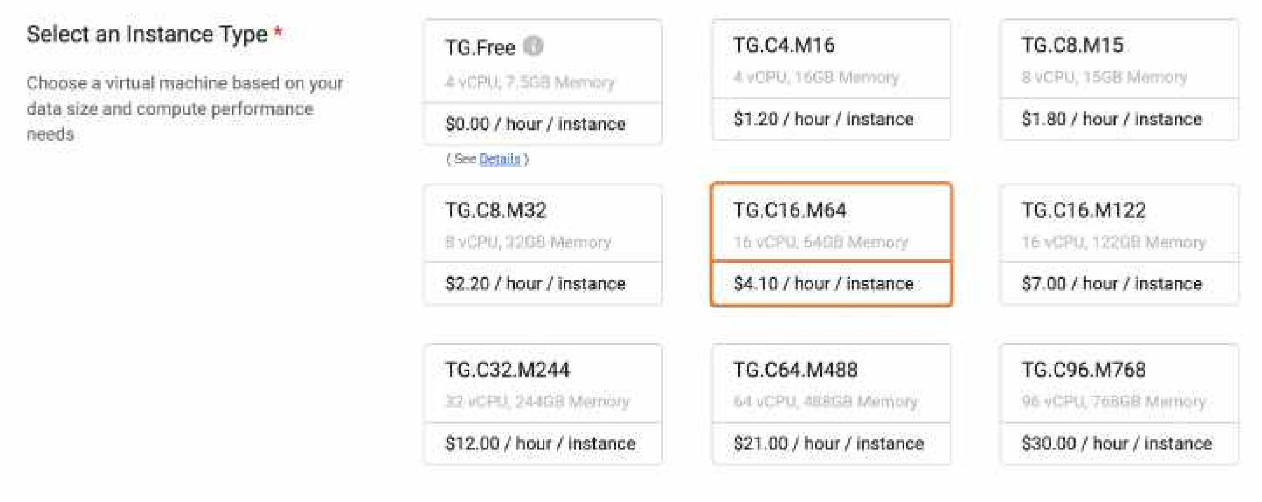
4. Configure your instance

4.1. Choose a cloud provider

The choice of cloud provider affects the configuration options for the available instance types and disk sizes, network latency for accessing your solution, the geographic location of your solution, and the cost of running your solutions. For more information, see [Cloud Providers and Regions].

4.2. Choose an instance type

Each instance type indicates its vCPU and RAM specifications as well as the hourly rate for running the instance. For suggestions on choosing the right instance type and disk storage, see [Sizing and Scaling].



4.3. Choose a region

Region reflects the physical location of the server running your solutions. It also affects the network latency for accessing your solution.

4.4. Choose a disk size

Disk storage refers to the root disk that is attached to your instance. If you need help choosing the right disk storage, see Sizing and Scaling.

> To ensure your solution has enough disk storage, TigerGraph mandates that any solution's disk size must be **at least 3 times the size of the solution instance's RAM**. For example, if you choose an instance type that has a RAM of 64 GB, then the solution's disk size must be more than 192 GB.

5. Enter cluster settings

5.1. Choose a partitioning factor

In a High Availability (HA) cluster, the partitioning factor refers to the number of parts or components your graph data is split into, which also equals the number of instances that collectively store one copy of the full graph.

If you only need a single instance to store your entire graph, enter 1 as your partition factor.

5.2. Choose a replication factor

The replication factor is the number of copies of data, each on a separate machine. TigerGraph Cloud currently supports a replication factor of up to three.

If you don't need additional copies of your data, enter 1 as your replication factor.

If you provision a cluster with a replication factor higher than 1:

- If your solution is on AWS or GCP, the replicas will automatically be provisioned on different availability zones.
- A load balancer would be automatically attached to your cluster to balance traffic between the replicas.

> If your cluster is on GCP and it has a replication factor higher than 1, you won't be able to use ports 9000 and 14240 to access API endpoints. Instead, you can access the endpoints using **HTTPS** requests on port 443:
>
> - To access the endpoints that normally listen on port 9000, append `/restpp` after the domain and before the endpoint. Since an HTTPS request already implies port 443, specifying the port is optional. However, you need to ensure that you are making HTTPS requests instead of HTTP requests.
>   - Example: To access the /echo endpoint: `curl "https://examplesolution.i.tgcloud.io/restpp/echo"`
> - To access the endpoints that normally listen on port 14240, use the original endpoint address, but on port 443. Similar to port 9000 requests, specifying the port is optional.
>   - Example: To access the health check endpoint on 14240: `curl "https://examplesolution.i.tgcloud.io/api/ping"`
> - Access to GraphStudio is not affected: opening the domain of the solution in a browser will directly take you to GraphStudio

6. Enter solution settings

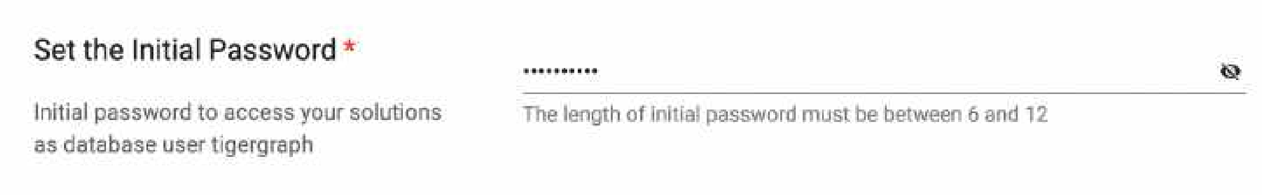
6.1. Name and tag your solution

Give your solution a name. A solution name can only contain alphanumeric characters, space, dash, and underscore and may be no longer than 20 characters.

Give your solution a tag, which helps you sort and identify your solutions. A solution tag may be no longer than 40 characters and can contain any character.

6.2. Set initial password

When a solution is provisioned, a default TigerGraph user `tigergraph` is created in the installation process. This is the initial password for the default user.



Set initial password

> This initial password allows you to log into the database when your solution is in the ready state. If you forget this password, you will need to terminate and recreate the solution.

6.3. Set a subdomain

Enter a unique subdomain for your solution. Only letters, numbers, and inner hyphen(-) are allowed in the subdomain. You can send HTTP requests to the REST endpoints on the domain once the solution is up and running. If you do not enter a subdomain, an automatically generated subdomain will be assigned to the solution.

6.4. Enter a description

Enter a description of what the solution is used for or anything else you would like to note about this solution.

7. Review and confirm.

After making sure all your settings are correct, click **Submit** to start your solution. Your solution should be ready after a brief warm-up period.

Cloud Providers and Regions

TigerGraph Cloud offers three cloud platform options: Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). The choice of cloud providers and regions affects the pricing and network latency of your solutions. The region indicates the physical location of your solution.

Each cloud provider has different offerings for instance type, disk storage, and backup storage. At the moment, TigerGraph Cloud only supports free-tier clusters on AWS. You may choose to deploy free-tier clusters on any region in AWS. For more information on the options of each cloud provider, see:

- Amazon Web Services (AWS)
- Microsoft Azure
- Google Cloud Platform (GCP)

Sizing and Scaling

TigerGraph Cloud offers different choices when it comes to instance types and disk sizes. This document helps you find the right instance type and disk size for your workload.

Choose the right instance type

Larger graphs require more RAM. Higher performance calls for both more CPUs and more RAM.

Our pricing table ↗ gives initial recommendations for which instance type to start with if you know how much data you will be loading into the graph. We call the input data your **raw data**. TigerGraph reorganizes your data into a graph, encoding and compressing it. We assume that your stored graph will be about 70% of the size of your raw data. This is a conservative estimate; it's often smaller.

You also need RAM for your graph querying and computation. Workloads vary considerably. In the pricing table, we recommend that your total RAM be about 1.5 times the maximum amount of raw data you think you will load.

Choose the right disk size

Disk storage refers to the root disk attached to your instance that holds the following files:

- OS installation files
- TigerGraph database installation files
- TigerGraph Graph storage
- Other data files uploaded to the instance
- Any output files generated by querying the database
- Logs generated while running TigerGraph

When you choosing the size of the disk, consider the size of the data for all the above installations and files.

To ensure adequate disk storage, TigerGraph Cloud mandates that any disk size of a solution must be **at least 3 times** the size of its instance's RAM. For example, if your solution instance has 64 GB of RAM, then the disk size must be higher than 192 GB.

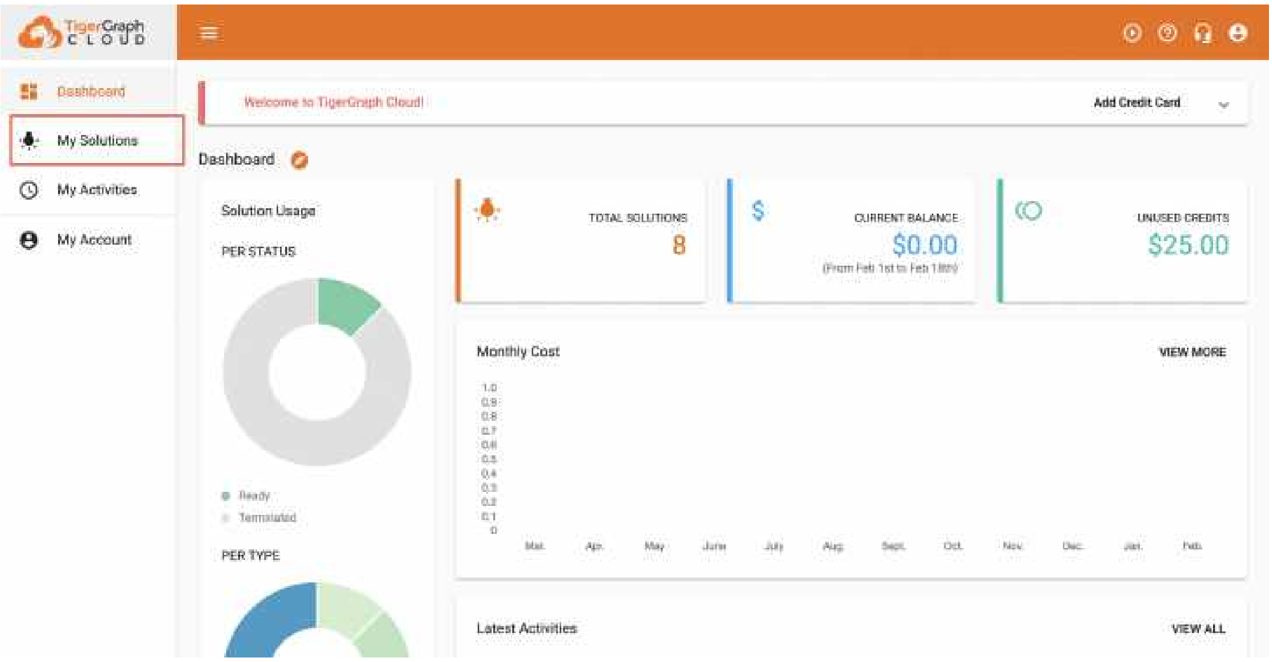
Stop, Restart, and Terminate a Solution

Stop a solution

When you stop a solution, you shut down the virtual machine instance. The TigerGraph database is frozen in its current state. Billing for that machine instance also stops. Scheduled backups also stop, but the backup copies are kept in EFS on AWS, Azure Files on Azure, and GCP Filestore on GCP. You will still incur charges for disk storage and backups when a solution is stopped. You can start the solution again. After a warm-up period, the solution will return to its previous state.

Below are the steps to stop a solution

1. Navigate to My Solutions.



2. Stop the solution.

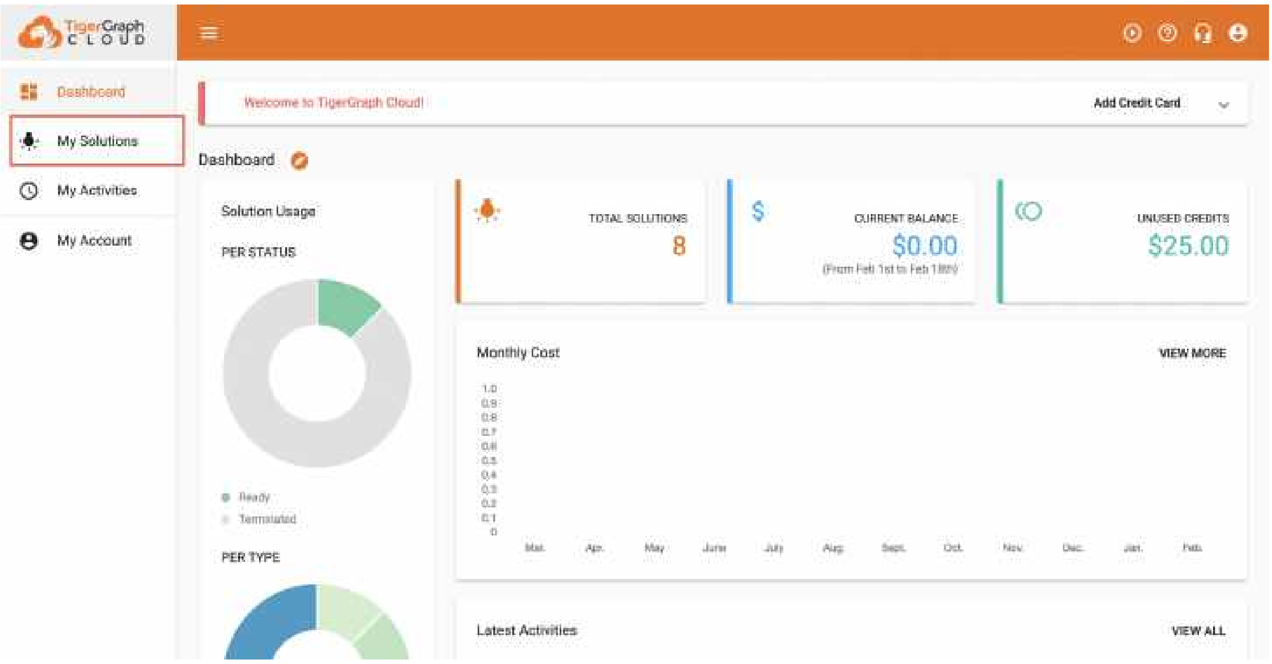
Find the solution you want to stop and click on the Solution Operations icon and click **Stop**.

Restart a solution

You can restart a stopped solution at any time. After a warm-up period, the solution will return to the state it was in when it was stopped.

Below are the steps to restart a solution:

1. Navigate to My Solutions



2. Restart the solution.

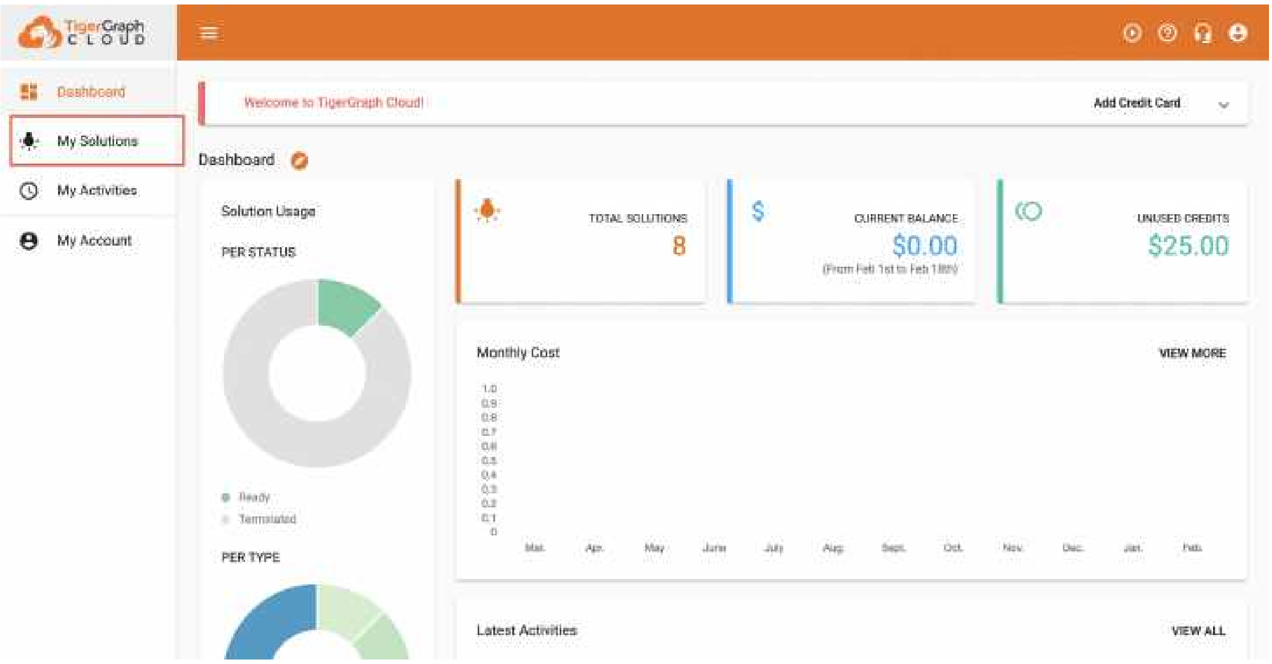
Find the stopped solution, and click the Solution Operations icon , and in the dropdown list, click **Start**.

Terminate a solution

By terminating a solution, you deprovision the virtual machine and the attached disk storage. The solution is deleted according to the cloud platform's policies and cannot be recovered, so is everything in the attached disk storage.

Below are the steps to terminating a solution.

1. Navigate to My Solutions.



2. Terminate the solution.

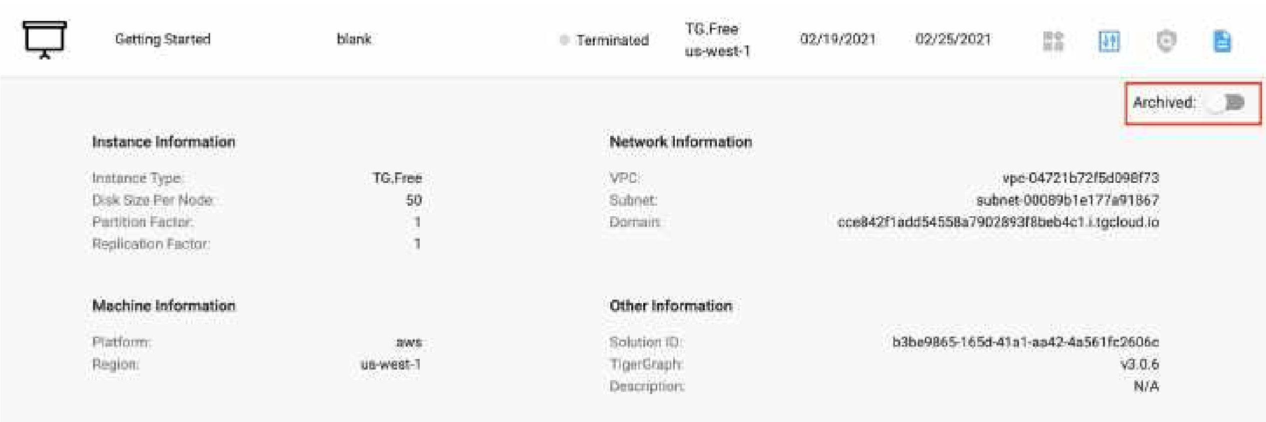
Find the solution you want to stop and click on the Solution Operations icon and click **Terminate**.

Archive a solution

After a solution has been terminated, it can be archived so that it no longer shows up on the default view in the list of solutions.

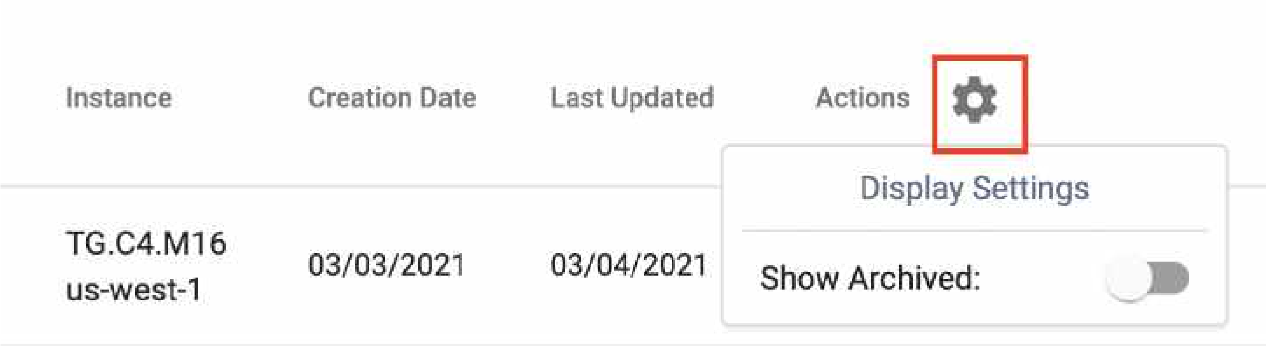
Archive a solution

To archive a solution, click on the solution row in My Solutions to expand it and click the **Archived** toggle.



Show archived solutions

To view archived solutions, click the settings icon to the right of the **Actions** column and toggle on **Show Archived** in the **Display Settings**.



Access a Solution

Once a solution has been provisioned, there are three ways to access the solution and interact with the database:

- GraphStudio
- REST endpoints
- GSQL web shell

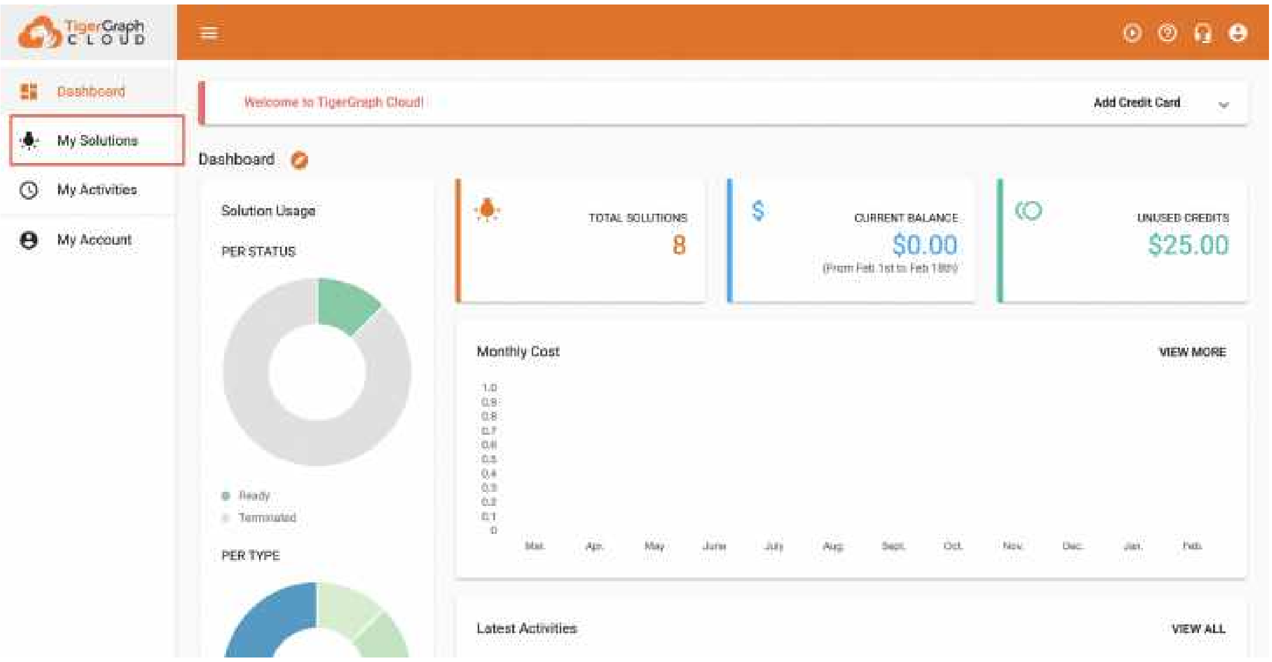
Access a Solution through GraphStudio

The TigerGraph GraphStudio™ UI (User Interface) provides an intuitive, browser-based interface that helps users get started quickly with graph-based application development tasks: designing a graph schema, creating a schema mapping, loading data, exploring the graph, and writing GSQL queries. GraphStudio is supported in the same way on TigerGraph Cloud as the on-premise version of TigerGraph.

Open GraphStudio

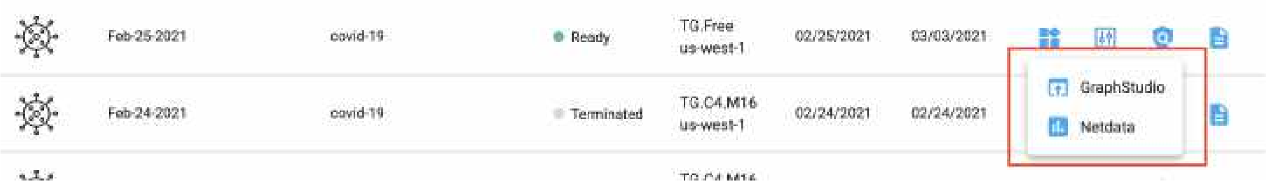
1. Navigate to My Solutions

After logging into TigerGraph Cloud ↗, go to **My Solutions**.



2. Open GraphStudio

Find the solution you want to access, and click the Applications icon in the same row, and click GraphStudios in the dropdown list.



Use GraphStudio

To learn how to use GraphStudio, read our GraphStudio UI Guide.

Send REST Requests to a Solution

Besides GraphStudio, another way to interact with your TigerGraph database is through our REST API. Use RESTful endpoints to send requests to TigerGraph solutions and develop applications.

On TigerGraph Cloud, RESTPP Authentication is enabled by default, and only the requests made with a valid authorization token in the request header will be accepted.

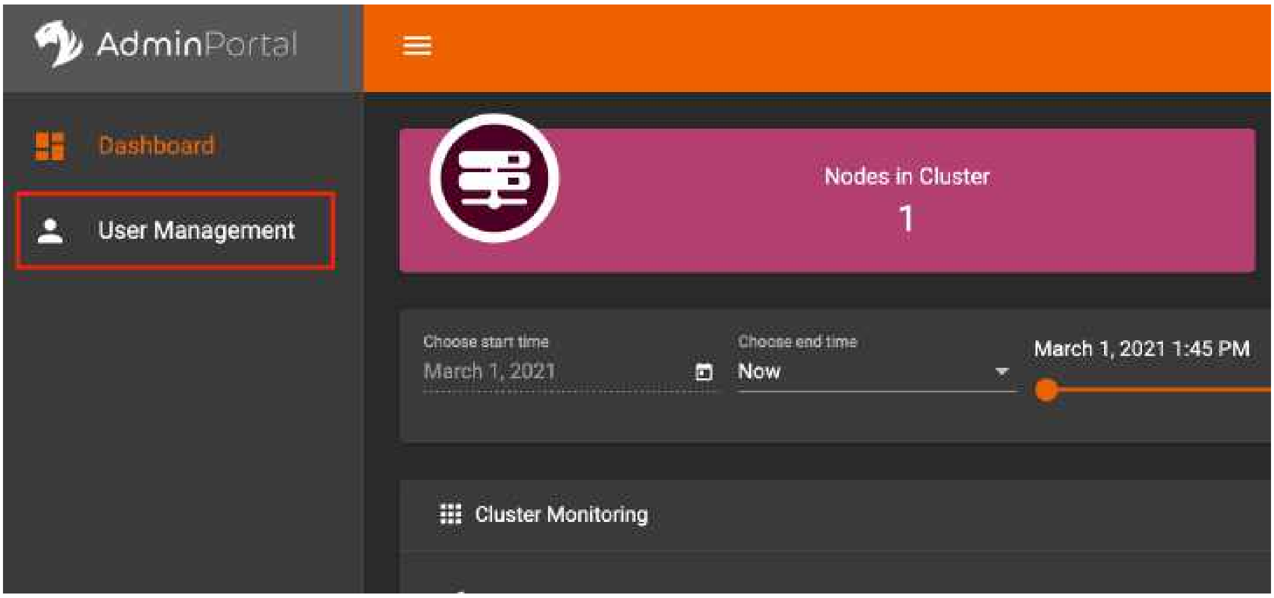
> If your cluster is on GCP and it has a replication factor higher than 1, you won't be able to use ports 9000 and 14240 to access API endpoints. Instead, you can access the endpoints using **HTTPS** requests on port 443:
>
> - To access the endpoints that normally listen on port 9000, append `/restpp` after the domain and before the endpoint. Since an HTTPS request already implies port 443, specifying the port is optional. However, you need to ensure that you are making HTTPS requests instead of HTTP requests.
>   - Example: To access the /echo endpoint: `curl "https://examplesolution.i.tgcloud.io/restpp/echo"`
> - To access the endpoints that normally listen on port 14240, use the original endpoint address, but on port 443. Similar to port 9000 requests, specifying the port is optional.
>   - Example: To access the health check endpoint on 14240: `curl "https://examplesolution.i.tgcloud.io/api/ping"`
> - Access to GraphStudio is not affected: opening the domain of the solution in a browser will directly take you to GraphStudio

Generate an authorization token

Here is a step-by-step guide to generating and using an authorization token for RESTPP:

1. Navigate to User Management

From GraphStudio, go to **Admin Portal**, and click **User Management**.

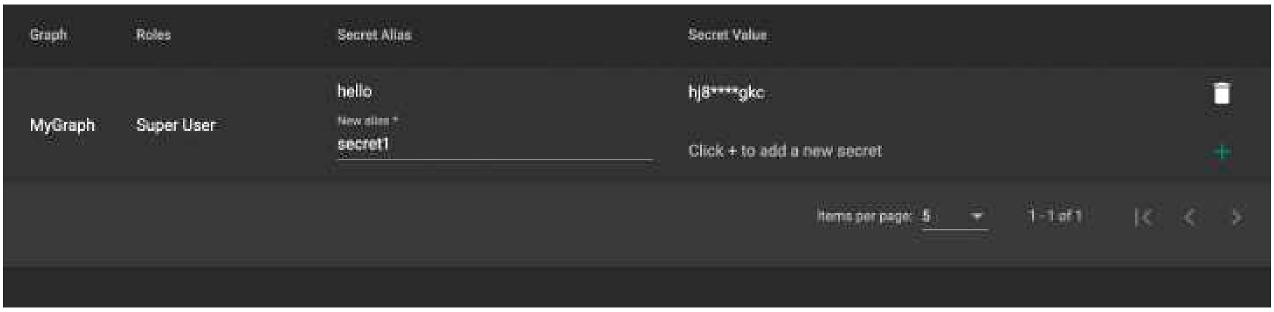


Click User Management

2. Generate a secret

In the **My Profile** tab, find the graph you want to generate the secret for, enter an alias for your secret and click the "**+**" symbol at the right side of the row.

Remember to copy and save the secret to a safe location. **This is the only time the secret will be exposed in full in Admin Portal, and you will not be able to see it again**.

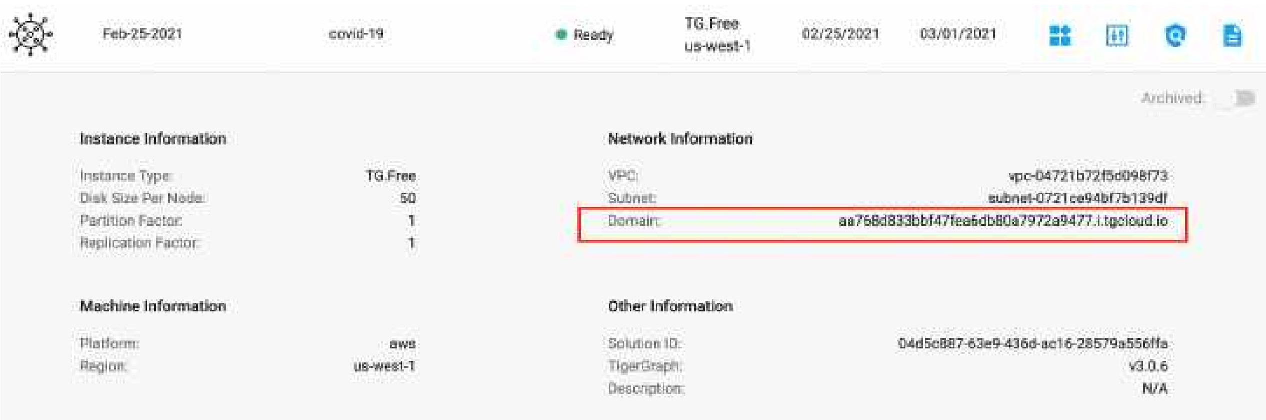


3. Generate a token

Use the `/requesttoken` endpoint to generate an authorization token for your solution. You can use either a GET or POST request. In this tutorial, we will be using a GET request.

1. Find the domain name of your solution

If you did not enter a subdomain when setting up your solution, a random subdomain will be automatically generated for you. Go to **My Solutions**, and click the solution you are trying to access, you will find the solution's domain name in the expanded view. Use this domain name as the server address when making REST requests.



2. Send the request to `/requesttoken`

The endpoint takes two parameters, `secret` and `lifetime`, and the latter is optional. Put the parameters in the query string and send the request using your favorite REST client. The below example uses curl to request the authentication token:

```
curl -X GET 'https://aa768d833bbf47fea6db80a7972a9477.i.tgcloud.io:9000/r
```

The response will look like this, where the string in the response with the key token is your authorization token:

```
{
  "code": "REST-0000",
  "expiration": 1617232169,
  "error": false,
  "message": "Generate new token successfully.",
  "token": "eq87e740arn73pp8chbf6j930pa7qorm"
}
```


4. Send a request using your token

Now that you have an authorization token, you can proceed to make requests to your solution. To use the authorization token, include it in the request header as a bearer token.

Refer to our RESTful API User Guide to learn about all the endpoints available. In this tutorial, we will make a request to the List vertices endpoint on a solution with the COVID-19 starter kit ↗ and list 5 patients and the patients' age.

```
curl -H "Authorization: Bearer fc6p0i2ijjt031n0sja6m9ci70p232h7" \
"https://aa768d833bbf47fea6db80a7972a9477.i.tgcloud.io:9000/graph/MyGraph
```

Response:

```
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "6100000100",
      "v_type": "Patient",
      "attributes": {
        "birth_year": 1959
      }
    },
    {
      "v_id": "6023000024",
      "v_type": "Patient",
      "attributes": {
        "birth_year": 0
      }
    },
    {
      "v_id": "6022000024",
      "v_type": "Patient",
      "attributes": {
        "birth_year": 1978
      }
    },
    {
      "v_id": "6020000020",
      "v_type": "Patient",
      "attributes": {
        "birth_year": 0
      }
    },
    {
      "v_id": "6015000008",
      "v_type": "Patient",
      "attributes": {
        "birth_year": 0
      }
    }
  ]
}
```

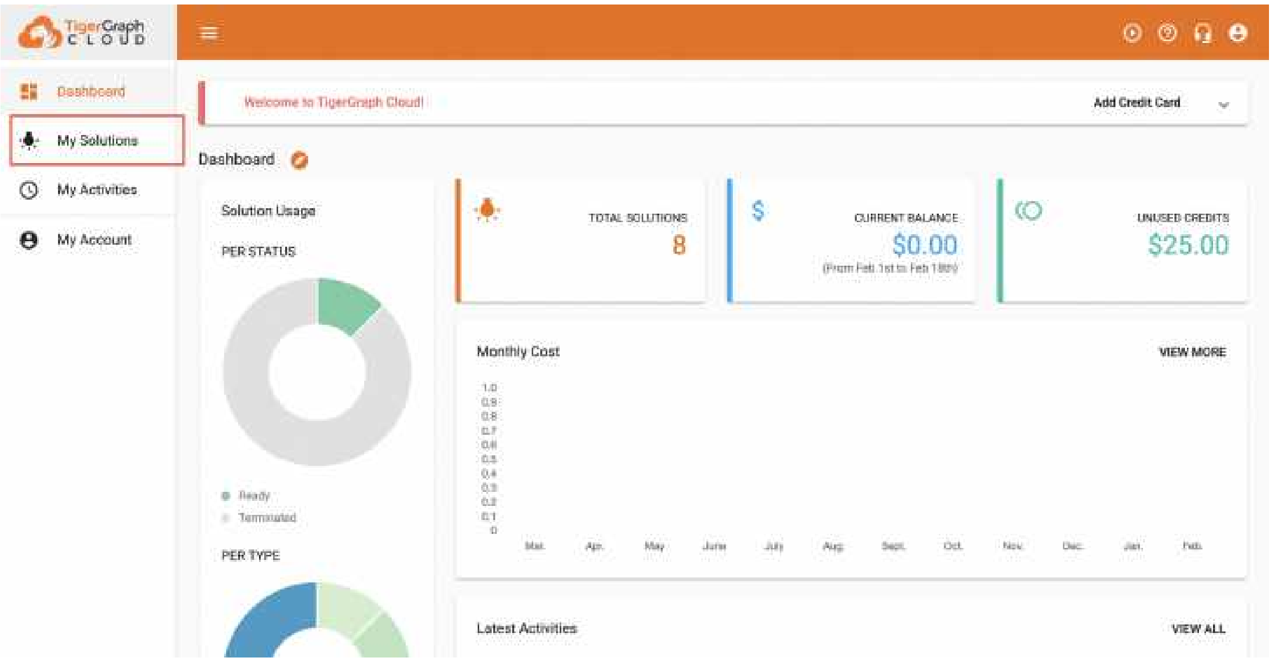
Access a Solution through GSQL Web Shell

With a paid tier solution and a valid payment method, you can also access the solution through the GSQL Web Shell. The web shell allows you to run GSQL commands on your TigerGraph Cloud solution.

Open GSQL Web Shell

1. Navigate to My Solutions

After logging into TigerGraph Cloud ↗, go to **My Solutions**.



2. Click GSQL Access

Find the solution you want to access, and click on the Admin Operations icon . Click **GSQL Access** from the dropdown menu.

3. Provide session information

In the pop-up window, provide the username of the TigerGraph user you'd like to proceed with, and specify a validity period for the session. The GSQL Web Shell session will terminate after the validity period is over.

Validity period and username

4. Enter your password

When the GSQL Web Shell opens, the shell will ask you for your password. Enter the password for the TigerGraph user whose username you just entered and press the enter key.

```
Password for tigergraph : ********
Welcome to TigerGraph.
GSQL > ls
---- Global vertices, edges, and all graphs
Vertex Types:
  - VERTEX City(PRIMARY_ID city_id STRING, city STRING, elementary_school_count UINT
erly_population_ratio DOUBLE, elderly_alone_ratio DOUBLE, nursing_home_count UINT) W
  - VERTEX Province(PRIMARY_ID province_id STRING, province STRING, population UINT,
alse"
  - VERTEX Country(PRIMARY_ID id STRING, country STRING) WITH STATS="OUTDEGREE_BY_ED
  - VERTEX SearchStat(PRIMARY_ID id STRING, cold FLOAT, flu FLOAT, pneumonia FLOAT,
UTE="false"
  - VERTEX WeatherStat(PRIMARY_ID id STRING, code UINT, avg_temp DOUBLE, min_temp DO
nd_direction DOUBLE, avg_relative_humidity DOUBLE) WITH STATS="OUTDEGREE_BY_EDGETYPE
  - VERTEX InfectionCase(PRIMARY_ID id STRING, infection_case STRING, confirmed UINT
E="true"
  - VERTEX Year_(PRIMARY_ID id STRING, year_ INT) WITH STATS="OUTDEGREE_BY_EDGETYPE"
  - VERTEX Month_(PRIMARY_ID id STRING, month_ INT) WITH STATS="OUTDEGREE_BY_EDGETYP
  - VERTEX Day_(PRIMARY_ID id STRING, day_ INT) WITH STATS="OUTDEGREE_BY_EDGETYPE",
  - VERTEX TravelEvent(PRIMARY_ID id STRING, latitude FLOAT, longitude FLOAT, visite
IMARY_ID_AS_ATTRIBUTE="false"
```

GSQL Web Shell

Monitor a Solution

TigerGraph Cloud offers users various ways to monitor the activities and the state of their solutions. This page lists a few resources that can help monitor a solution's usage:

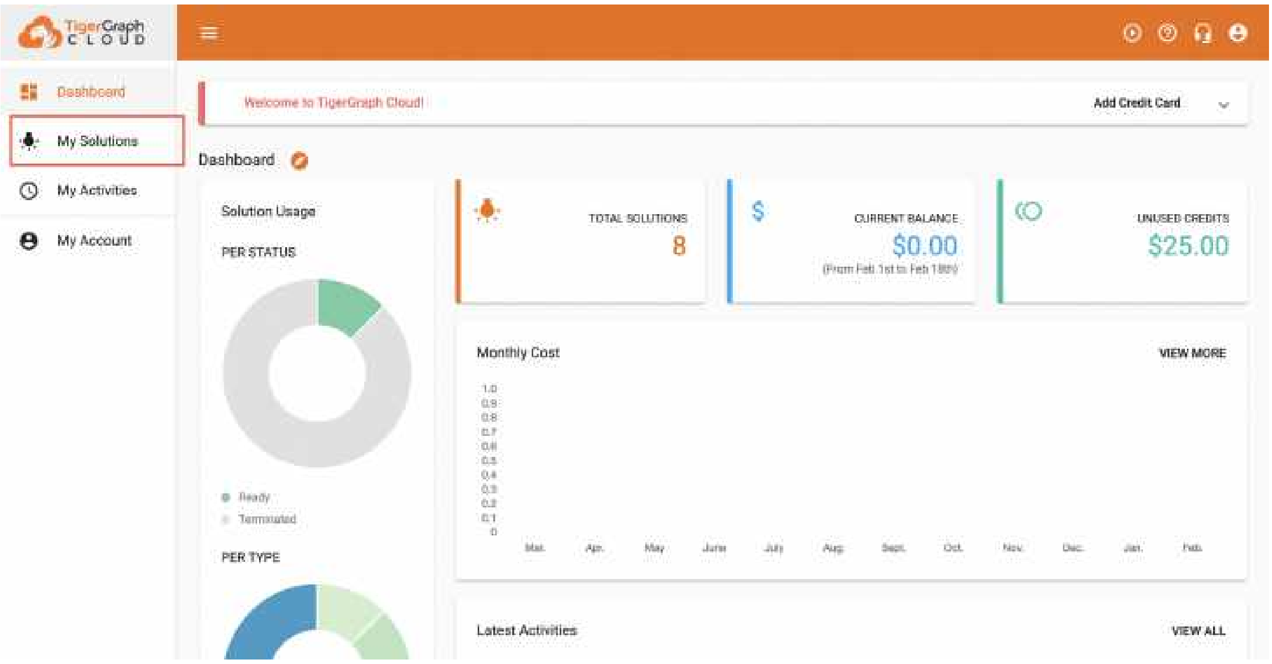
- Monitor Solution State
- View Activity Log

Monitor Solution State

TigerGraph Cloud allows you to view detailed operational data on a running solution, such as CPU usage, RAM usage and, and disk and network IO data.

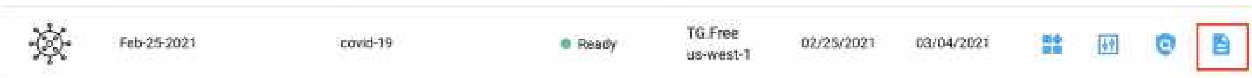
1. Navigate to My Solutions

After logging into TigerGraph Cloud ↗, go to **My Solutions**.



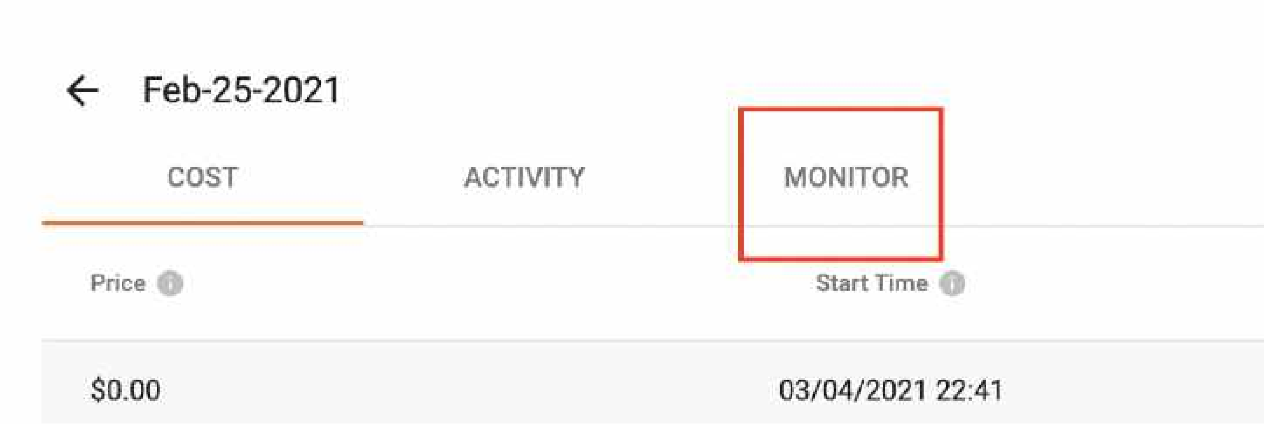
2. Click Solution Details

Find the solution you want to monitor, and click on the Solution Details icon on the right side of the same row.



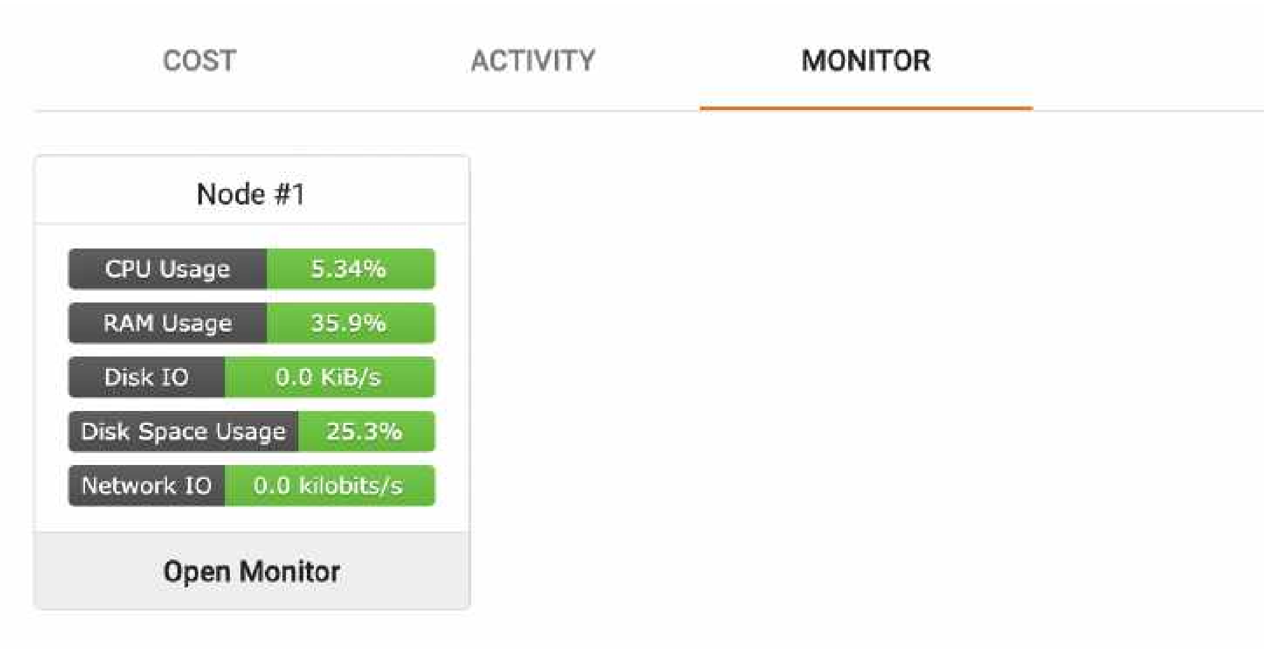
3. Click Monitor

Click the **Monitor** tab.



Click monitor

This will bring you a brief summary of your solution:



4. Click Open Monitor

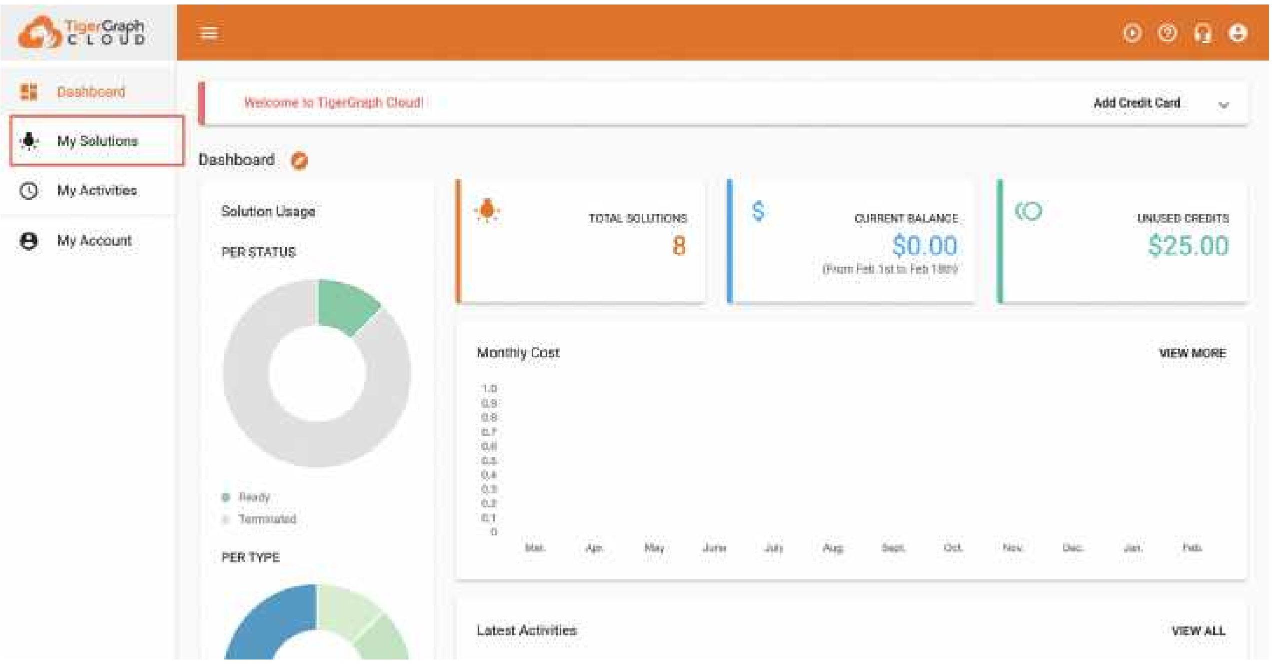
Click **Open Monitor** to view more detailed statistics of your solution by [Netdata monitor ↗].

View Activity Log

TigerGraph Cloud solutions log their activities in an activity log, which you can view on TigerGraph Cloud.

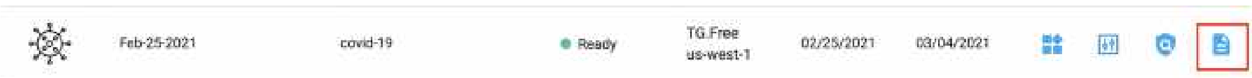
1. Navigate to My Solutions

After logging into TigerGraph Cloud ↗, go to **My Solutions**.



2. Click Solution Details

Find the solution you want to monitor, and click on the Solution Details icon on the right side of the same row.



3. Click Activity

Click the **Activity** tab, which shows you the activity log of that solution.

COST | **ACTIVITY** | MONITOR

	' Feb-25-2021 ' - the solution is ready
	' Feb-25-2021 ' - starting the solution
	' Feb-25-2021 ' - user requests to start the solution
	' Feb-25-2021 ' - solution is stopped

Resize a Solution

As your workload changes, you can resize your TigerGraph Cloud solution to maximize the efficiency of your resources. You can resize the instance type or upgrade the storage volume:

- Resize instance type
- Upgrade storage volume

Resize Instance Type

As your workload changes after you have created a solution, you can choose to resize your solution's instance type.

Before you begin

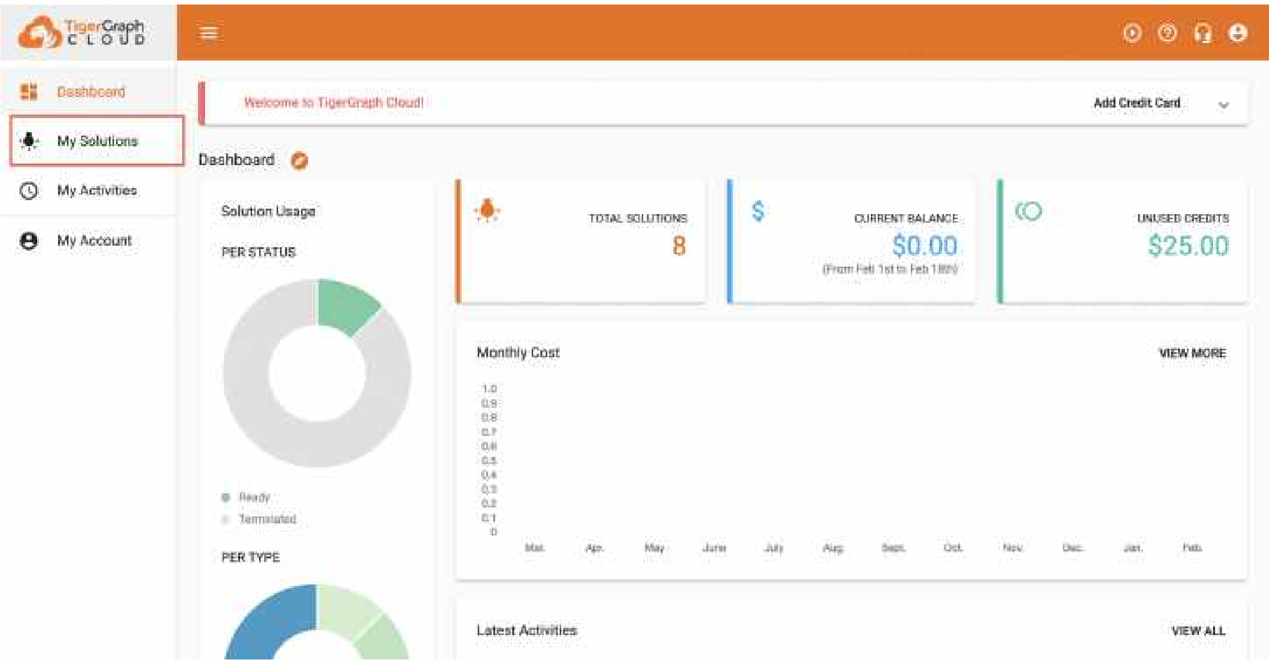
- If you choose to upgrade the instance type of a free-tier solution, you must also upgrade the disk volume of the solution.
- Resizing instance types would result in a short period of downtime.

Procedure

Below is the procedure to resize the instance type of a TigerGraph Cloud solution:

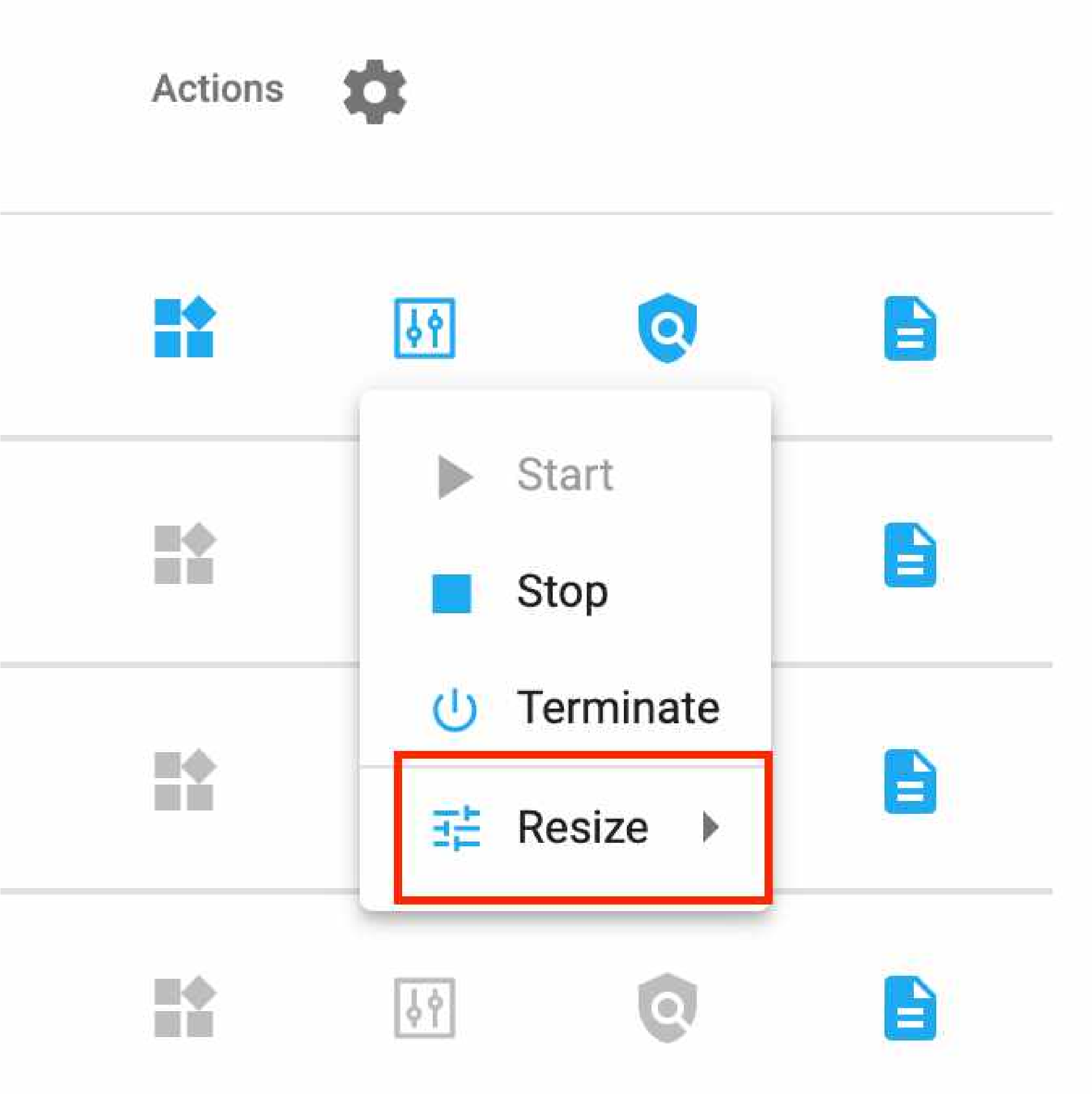
Step 1: Navigate to My Solutions page

After logging into your TigerGraph Cloud account, click **My Solutions** on the left side menu to go to the **My Solutions** page.



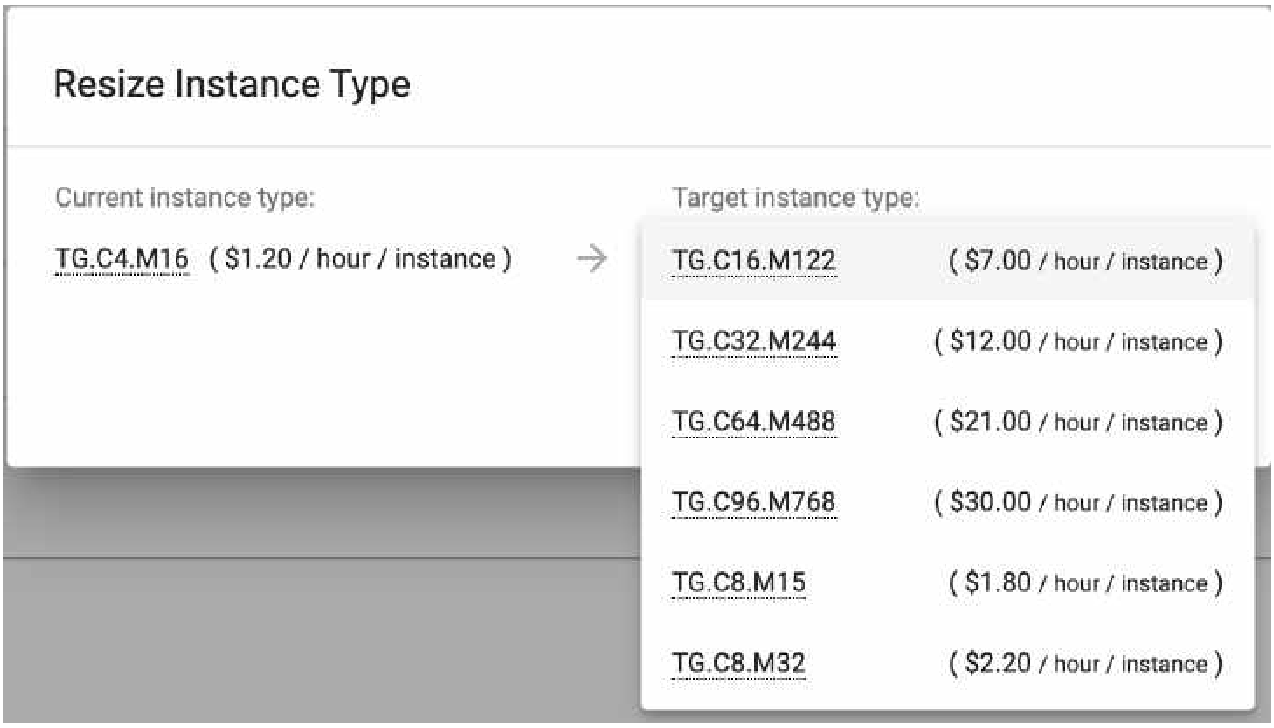
Step 2: Choose a new instance type

On the My Solutions page, click the Solutions Operations icon in the Actions column and hover over **Resize**. In the dropdown menu that appears, select **Instance Type**.



Hover over Resize

In the window that appears, choose the new instance type that you'd like to change your instance type to.



Resize instance type window

Click **Resize** to confirm and start the resizing operation.

Upgrade Storage Volume

If you decide that your solution needs more disk storage space, TigerGraph Cloud allows you to upgrade your disk volume.

Before you begin

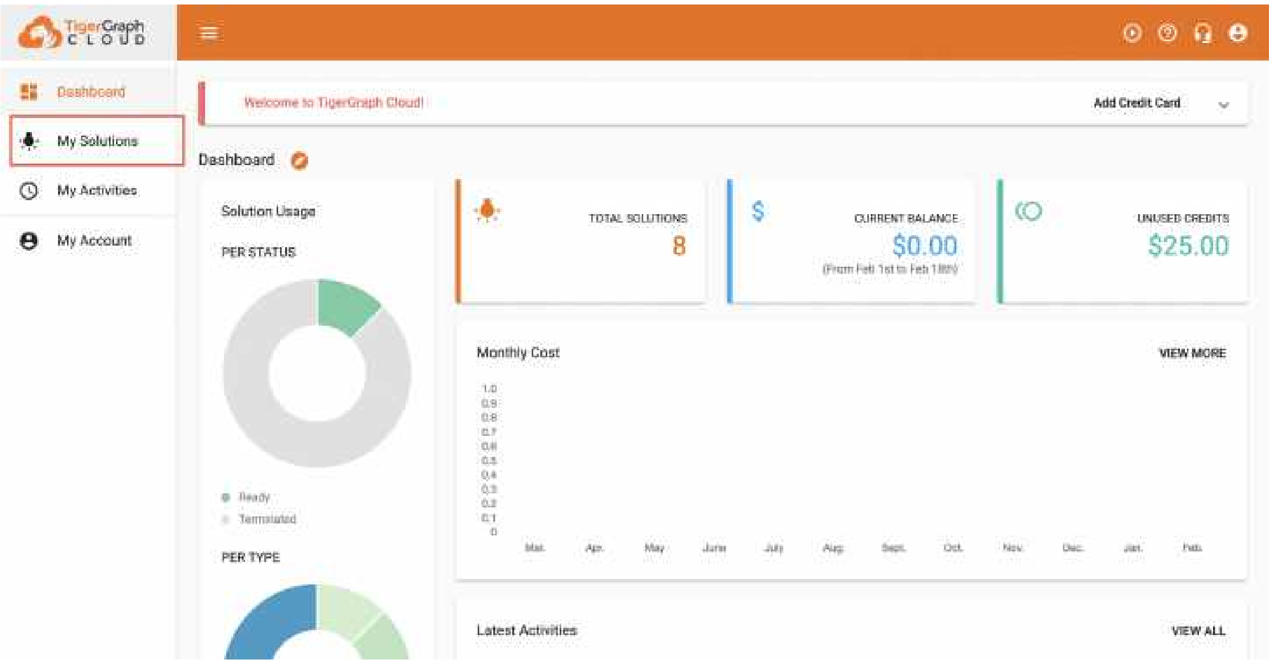
- You need a paid-tier solution in order to upgrade disk volume

Procedure

Below is the procedure to upgrade the disk volume of a solution.

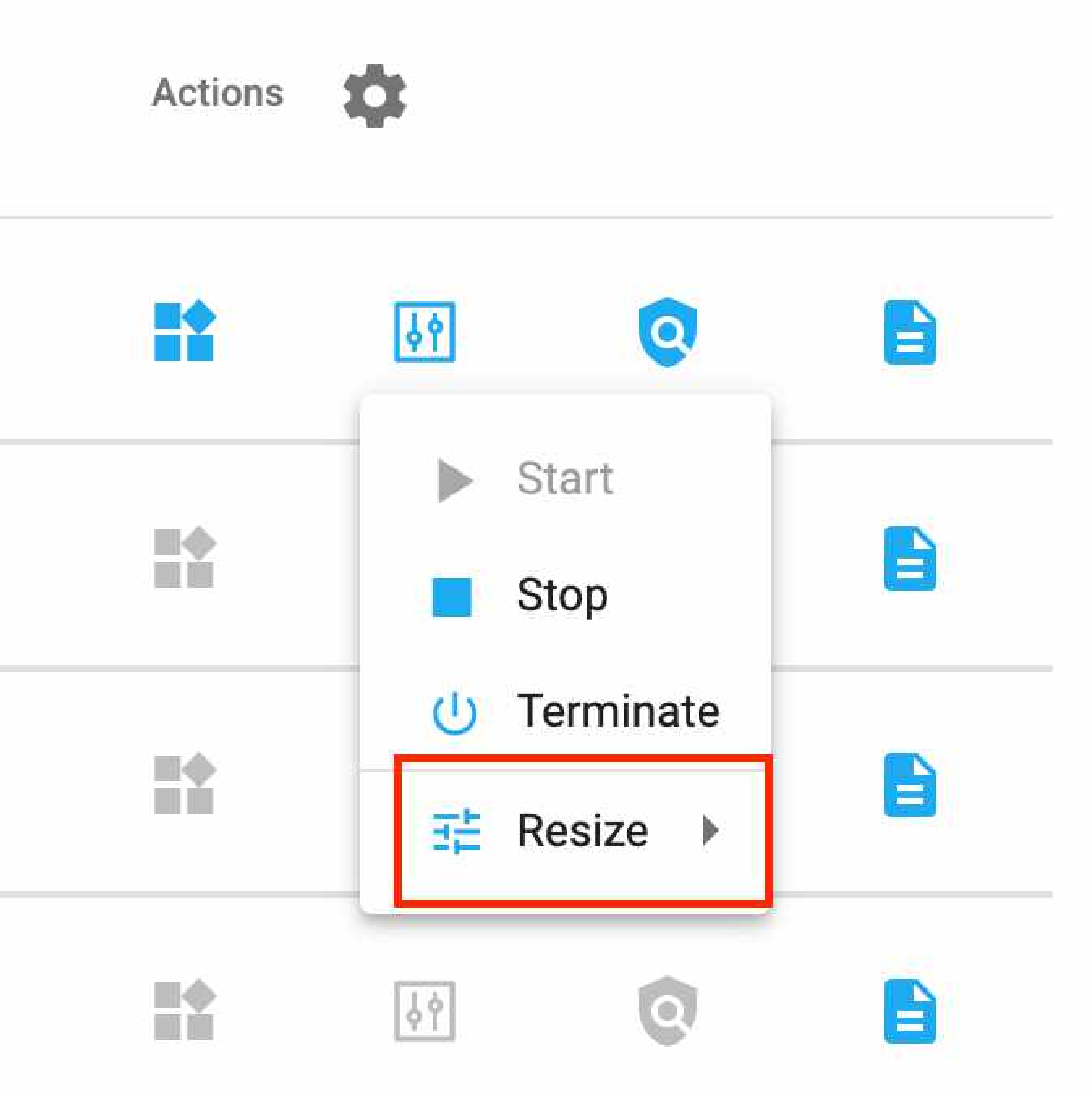
Step 1: Navigate to My Solutions page

After logging into your TigerGraph Cloud account, click **My Solutions** on the left side menu to go to the **My Solutions** page.



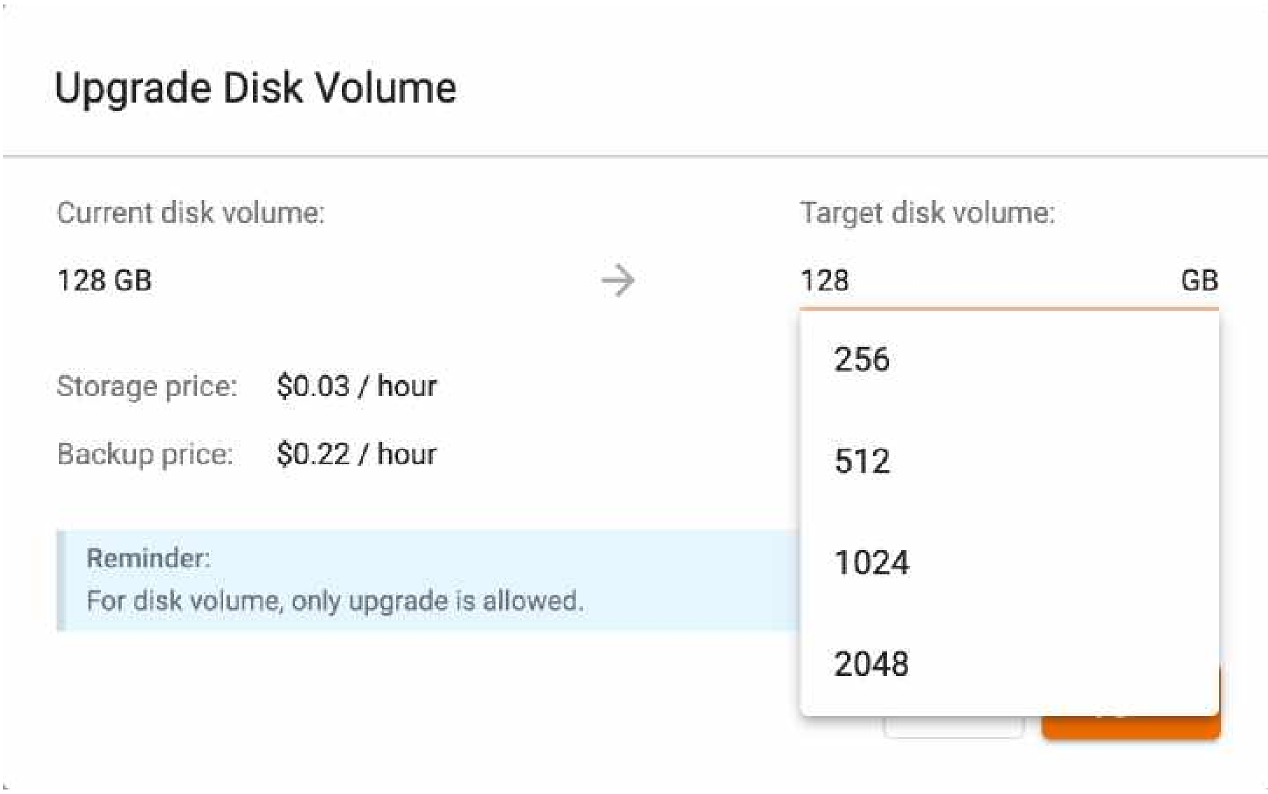
Step 2: Upgrade disk volume

On **My Solutions**, click the Solutions Operations icon in the Actions column and hover over **Resize**. In the dropdown menu that appears, select **Disk Volume**.



Hover over Resize

In the window that appears, choose your target disk volume:



Click **Upgrade** to confirm and start the upgrade.

Security and User Management

TigerGraph Cloud account

Your TigerGraph Cloud account serves as your credential on the platform level. It allows you to manage solutions as well as payments. It also gives you superuser access to all of the graphs hosted by the solutions on the account.

TigerGraph users

On the solution level, the same role-based access control used by the on-prem version of TigerGraph also applies to TigerGraph Cloud. Each solution or solution cluster can have many TigerGraph users and each user can have different roles for each graph with different privileges.

For more information on how to manage users and roles in Admin Portal, see User Management.

Preconfigured security features

Encryption-at-rest

All data loaded into TigerGraph Cloud, including data in disk storage and backups, are automatically encrypted-at-rest using the cloud provider's native encryption protocol with platform-managed keys.

Transport Layer Security (TLS)

TigerGraph Cloud uses TLS to encrypt connections to your graph databases.

RESTPP Authentication

By default, all TigerGraph Cloud solutions have RESTPP authentication turned on. To make requests to the RESTPP server, you must generate a secret in the Admin Portal and use that secret to generate an authentication token.

Manage Your TigerGraph Cloud Account

Reset your TigerGraph Cloud account password

1. Go to TigerGraph Cloud login portal ↗.
2. Click **Login / Register**
3. Click **Don't remember your password?**
4. Enter your email address. If you signed up using an identity provider, use the email address associated with your account with the identity provider. Click **SEND EMAIL**.
5. Go to your email inbox, and find an email titled **Reset your password**. Click on the link in the email
6. Enter your new password.

Manage Database Users

On the solution level, the same role-based access control used by the on-prem version of TigerGraph also applies to TigerGraph Cloud. Each solution or solution cluster can have many TigerGraph users and each user can have different roles for each graph with different privileges.

To learn how to manage users and roles in Admin Portal, refer to the User Management section in Admin Portal UI Guide.

Backup and Restore

Backups are snapshots of the state of your solutions at a given point in time. Currently, any paid solutions provisioned by an account with a valid payment method can have a maximum of 7 backups concurrently.

> The backup feature is not supported on free-tier instances or instances provisioned by an account without a valid payment method.

Data Storage and Encryption

Backups are saved to Amazon EFS on AWS, Azure Files on Microsoft Azure, and GCP Filestore on GCP. Backup storage is provisioned when you start a paid-tier solution and is based on the maximum number of backups allowed (currently fixed at 7). It only stops incurring charges when the solution is terminated. Stopping the solution does not stop charges for backup storage.

Backups are encrypted by default using the native encryption feature of the cloud provider with platform-managed keys.

Scheduled backup

By default, scheduled backup is turned on when instances are running and will be done daily. If the maximum retention number is reached when a new backup is made, the oldest scheduled backup will be deleted. To change the backup schedule, change your backup settings in Admin Portal.

Change backup schedule

1. Go to Admin Portal.

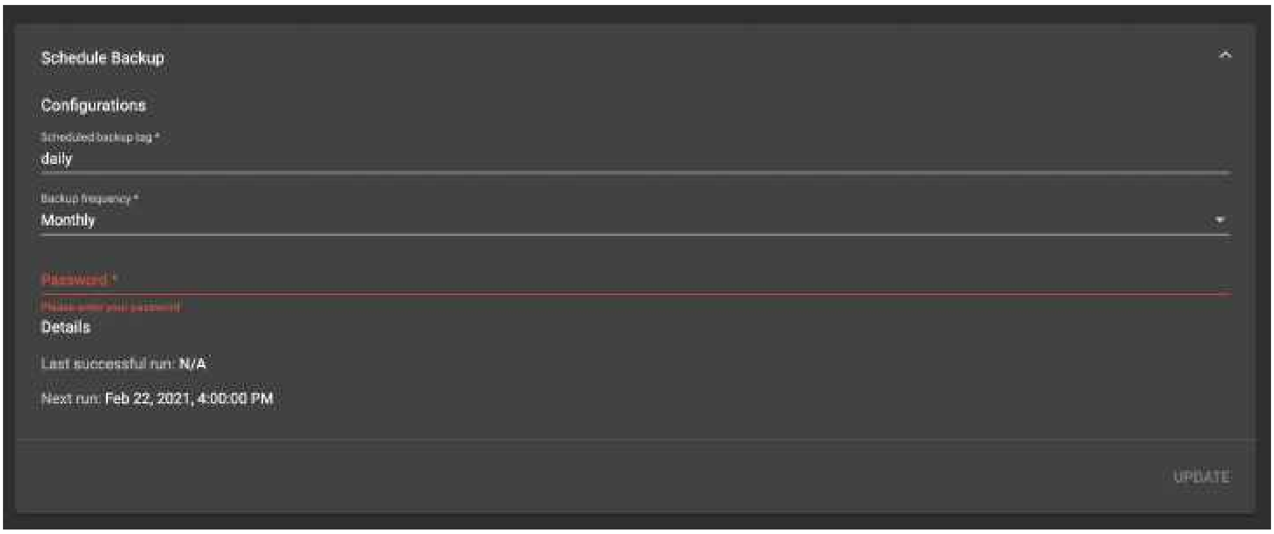
From **My Solutions** page, go to GraphStudio. Then click **Admin** on the upper-right corner to go to **Admin Portal**.

2. Click Backup & Restore.

On the left side of the screen, click **Backup & Restore.**

3. Specify backup schedule.

Choose the backup frequency you want, and supply the password of your TigerGraph user.



4. Click UPDATE.

Manual backup

Aside from scheduled backups, you can also make backups manually in Admin Portal.

- If you want to perform a manual backup when there are already seven copies of backups, you need to delete an older backup.
- The maximum number of manual backups is six, as the platform always reserves one spot for scheduled backups.

Below are the procedures to create a backup manually:

1. Go to Admin Portal.

2. Click Backup & Restore.

3. Provide tag and password.

In Backup/Restore, supply a tag for your backup and enter the password of your TigerGraph user.

4. Click BACK UP.

Restore a backup

You can use a backup to restore your solution.

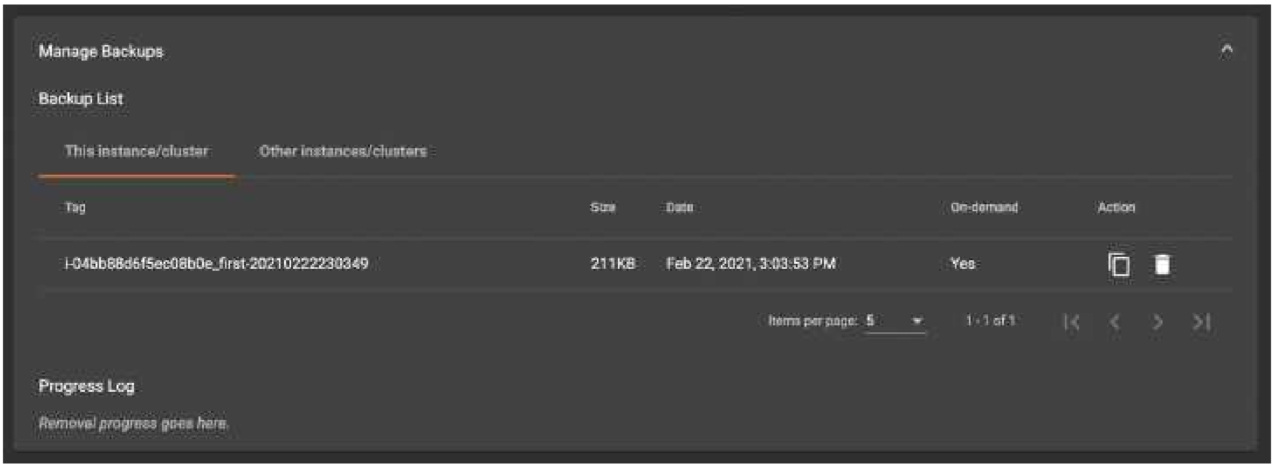
Below are the procedures to restore a backup:

1. Go to Admin Portal.

2. Click Back up & Restore.

3. Copy backup tag.

In **Manage Backups**, find the backup you want to use for the restore and click the clipboard icon to copy the tag of the backup.



Delete a backup

Below are the procedures to delete a backup

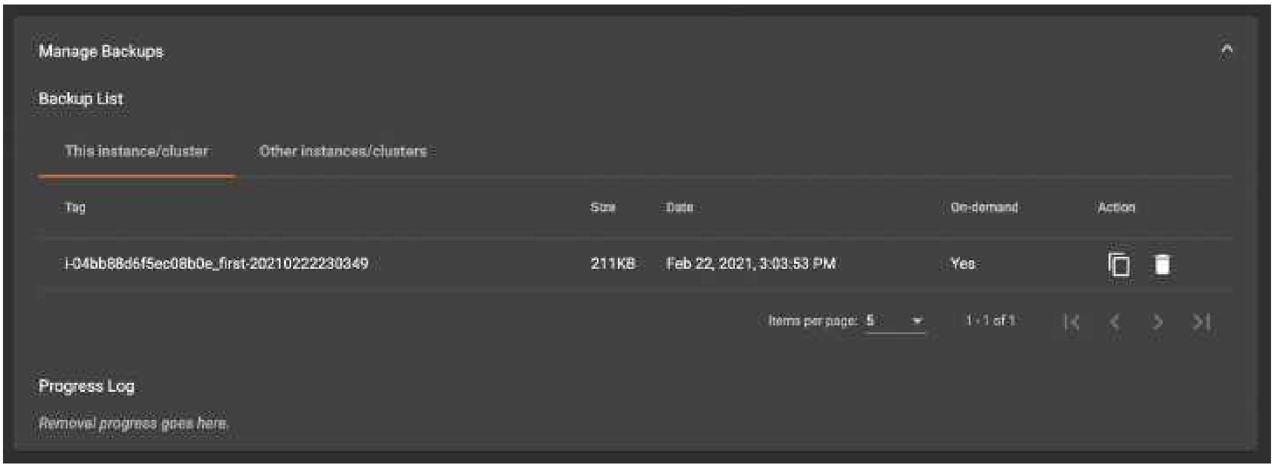
1. Go to Admin Portal.

2. Click to Backup & Restore

3. Go to Manage Backups

4. Delete the backup.

In Backup List, find the backup you want to delete and click the delete button in the **Action** column.



VPC Peering

TigerGraph Cloud supports VPC Peering connections for AWS and GCP clusters.

- Set up VPC peering connection on AWS
- Set up VPC peering connection on GCP

Set Up VPC Peering on AWS

This is a guide for setting up a VPC peering connection between your TigerGraph Cloud VPC and another AWS VPC.

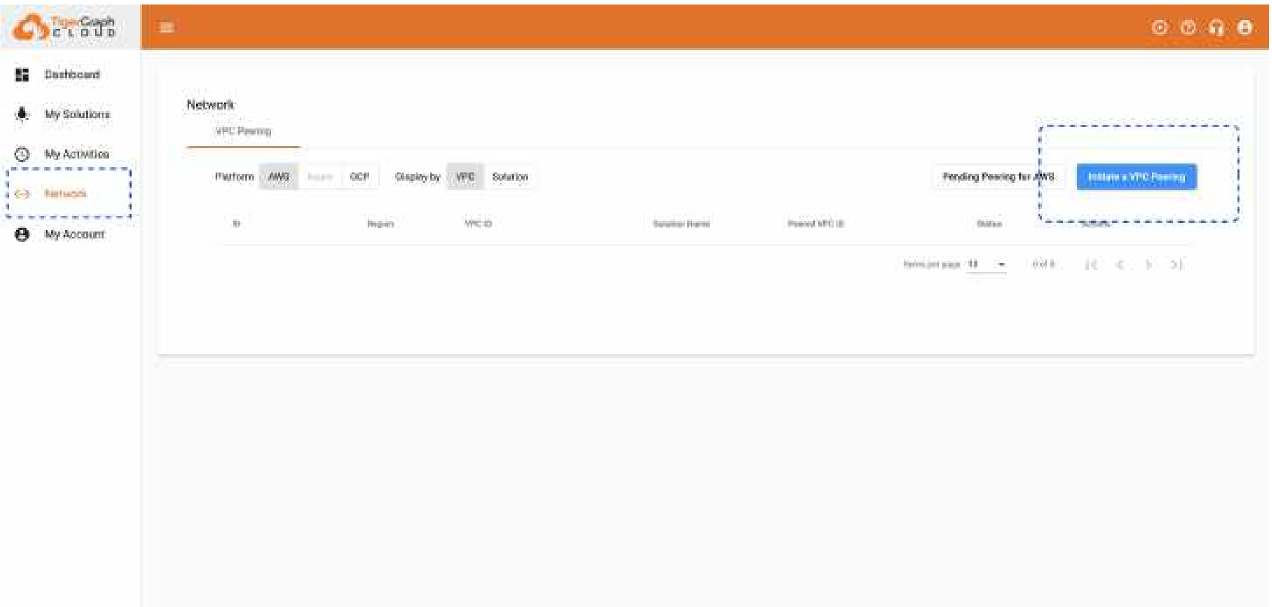
Prerequisites

- Unterminated TigerGraph Cloud solutions on AWS

Procedure

1. Initiate VPC peering

Log into the cloud portal, and navigate to the Network tab. Click Initiate VPC peering.



In the TigerGraph VPC section, Choose the TigerGraph VPC that you want to initiate the peering connection from.

In the Peered VPC section, fill in the information of the VPC that you hope to peer with:

- `OwnerID`
  - The ID of the VPC owner's AWS account, ex: `123456789012`
- `Region`
  - The region of the VPC you want to peer with, ex: `us-west-1`
- `VPC ID`
  - The ID of the VPC: `vpc-123456789abcdef01`
- `CIDR`
  - CIDR-block of the peered VPC, ex: `10.0.0.0/16`. You can also use a IPv6 CIDR here if your peered VPC has one.

2. Accept VPC peering request

After you have completed the first step, a VPC peering connection will be sent to your VPC. Follow the steps below to accept the peering connection on your AWS VPC. See more in AWS's documentation on Accept VPC Peering Request ↗.

1. Go to “VPC service” → “Peering Connections” in the AWS web console
2. Accept the connection from the TigerGraph VPC

3. Update route table for VPC peering connection

After accepting the peering request, you still need to update your route table for the peering connection.

To add a route for a VPC peering connection

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/ ↗.
2. In the navigation pane, choose **Route Tables**.
3. Select the check box next to the route table that's associated with the subnet in which your instance resides.

4. Choose **Actions**, **Edit routes**.
5. Choose **Add route**.
6. For **Destination**, enter the CIDR block of the TigerGraph VPC you are peering with. You can find the CIDR block of the TigerGraph VPC at **Peering Connections** in the VPC service where you accepted the peering connection. There is a column "**Requester CIDRs**".
7. For **Target**, select the VPC peering connection, and then choose **Save changes**.

Set Up VPC Peering on GCP

This is a guide for setting up a VPC peering connection between your TigerGraph Cloud VPC and another GCP VPC.

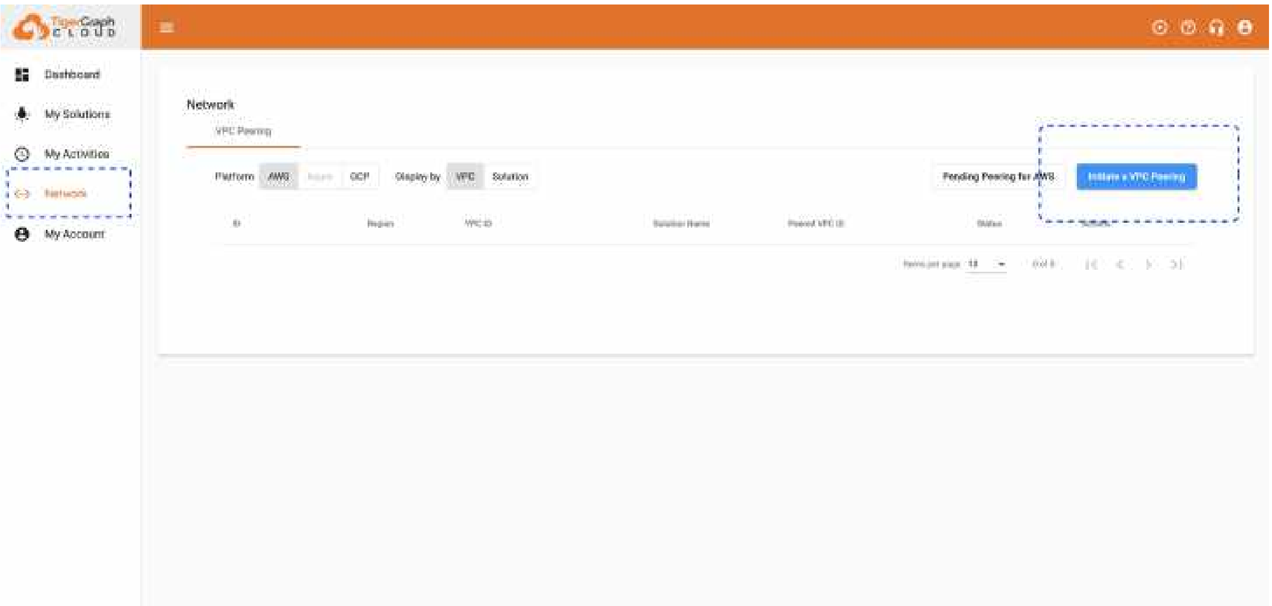
Prerequisite

- At least one unterminated TigerGraph Cloud solution on GCP

Procedure

1. Initiate VPC Peering

Log in to the Cloud portal and navigate to the Network tab, click "**Initiate VPC Peering**"



Network tab

In the TigerGraph VPC section, choose the TigerGraph VPC that you want to initiate the peering connection from.

In the Peered VPC section, fill in the information of the VPC that you hope to peer with:

- VPC ID: `projects/[projectID]/global/networks/[VPCName]`, where `projectID` is the ID of your GCP project, and `VPCName` is the name of the VPC.

2. Create VPC peering connection from peered VPC to host VPC

After completing Step 1, your TigerGraph VPC will create a peering configuration to the VPC you hope to peer with. At this time, the peering connection will appear as **Inactive**. To complete the peering connection, your still need to create a matching peering configuration on your peered VPC to the TigerGraph VPC:

1. In the Google Cloud Console, go to the **VPC Network Peering** page.
   Go to VPC Network Peering ↗
2. Click **Create connection**.
3. Click **Continue**.
4. Enter a **Name** for your peering connection.
5. Under **Your VPC network**, select a network you want to peer.
6. Select the TigerGraph network as the network to peer with.
   - Select **In another project**. You will need to specify the project ID that includes the network you want to peer with and the name of the VPC network:
     - Project ID: `tgcloud-prod`
     - VPC network name: Name of your TigerGraph Cloud VPC. You can find it in the **TigerGraph Cloud portal - Network tab - VPC Peering** page. Choose `GCP` as the platform, and in the VPC column is the VPC name.
7. You can leave the other fields as they are (with only **Export subnet routes with public IP** selected) unless you need special configurations.
8. Click **Create**.

Location of VPC name

After creating the matching peering configuration, the status of your VPC connection should change to `Active,` meaning the two VPCs are peered successfully.

Billing

Machine instances and storage are billed hourly, which means that you are charged for each continuous period of operation, measured in seconds. *Instances* incur charges from when they are started until they are stopped (or terminated). *Disk storage* attached to instances and *backup storage* incur charges from when an instance is first activated until the instance is terminated. Our billing system will total the charges at the end of each calendar month, generate an itemized invoice, and charge your card.

Data transfer costs from the cloud platforms are calculated according to the cloud platform charge and will be added to your invoice in the following month.

> This section only applies to the standard hourly service of TigerGraph Cloud. If you have a committed long-term contract with TigerGraph, please contact sales@tigergraph.com ↗ for questions.

Pricing Information

View our pricing table ↗ for detailed rates for each type of charge.

Manage Payment Methods

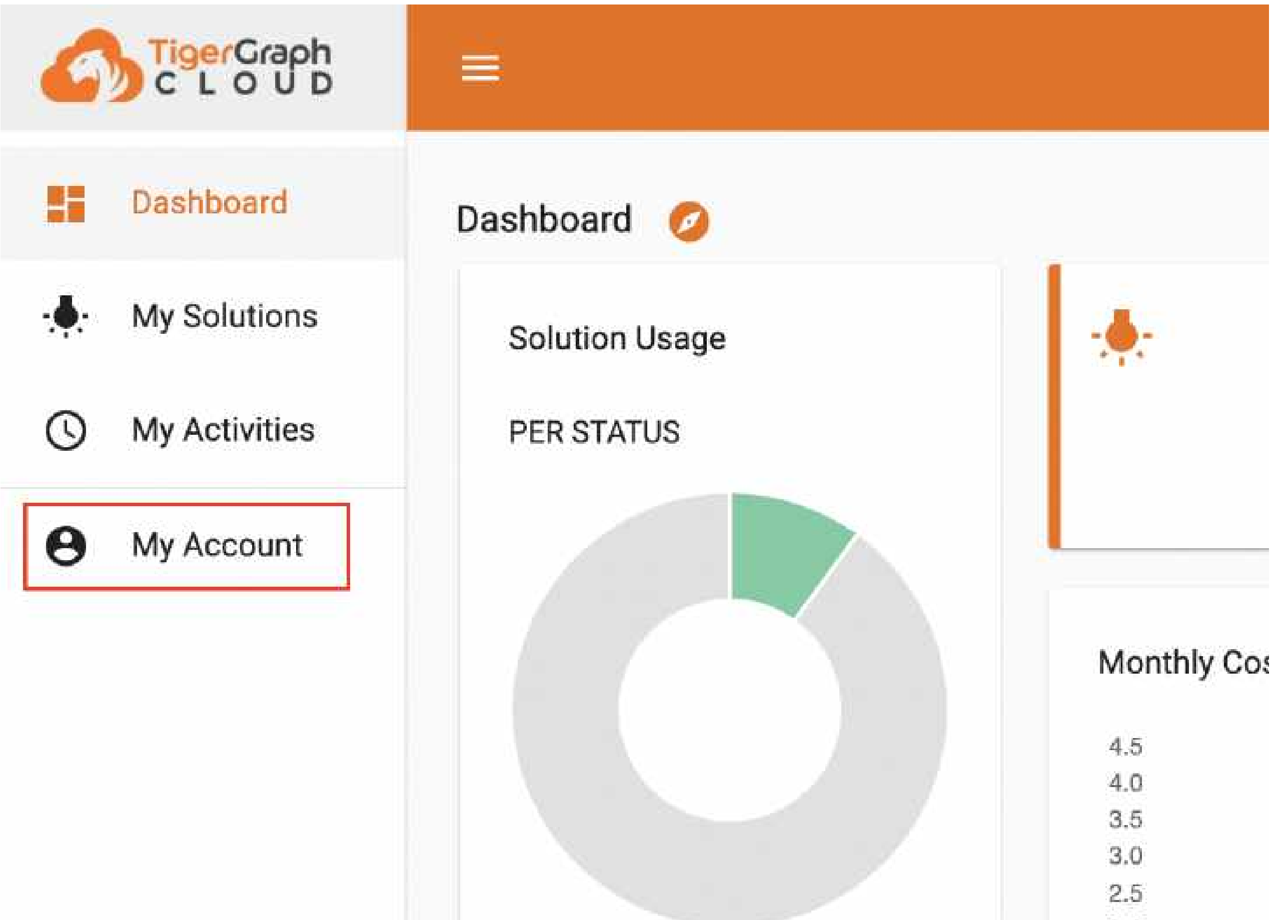
For standard hourly use of TigerGraph Cloud solutions, we currently support payment through cards. This document offers is a tutorial on how to add or delete a card and how to set a card as the default payment option.

If you need to use ACH payment, or purchase bulk cloud credits, please contact sales@tigergraph.com

Add a card

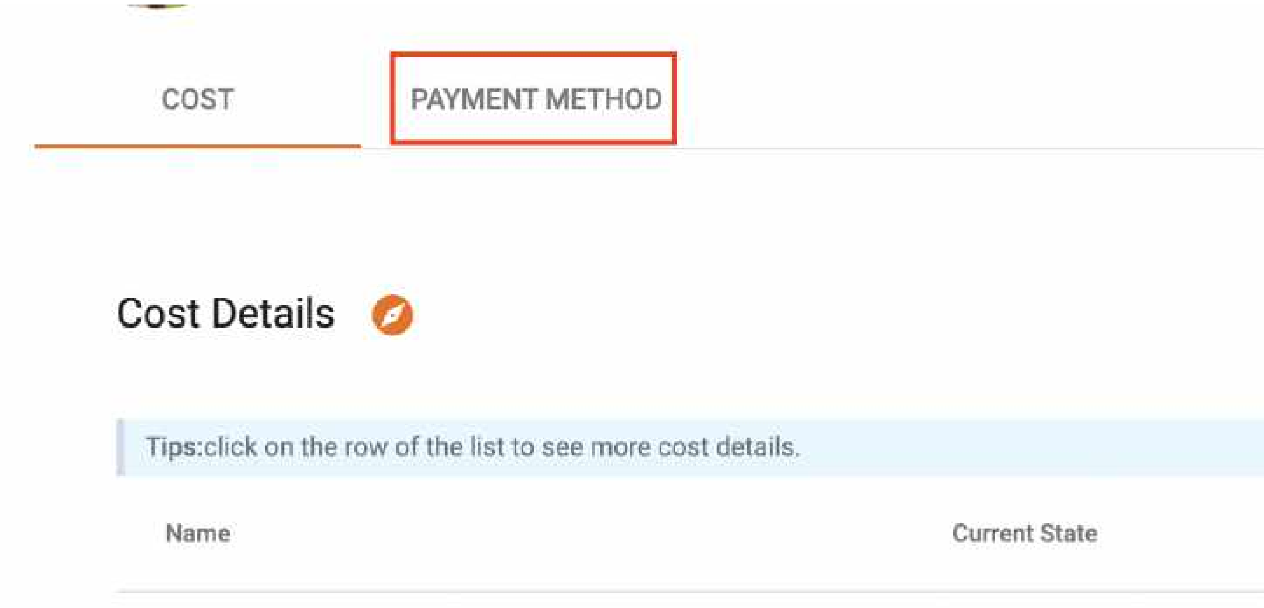
1. Navigate to My Account

On the left-side menu, click **My Account**.



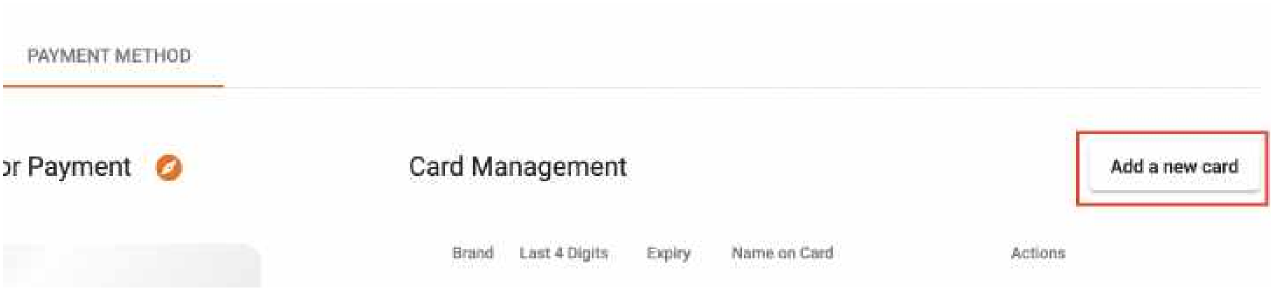
2. Click Payment Method

On the My Account page, click **Payment Method.**



3. Click Add a new card

In the **Payment Method** tab, click **Add a new card**.



4. Enter your card information

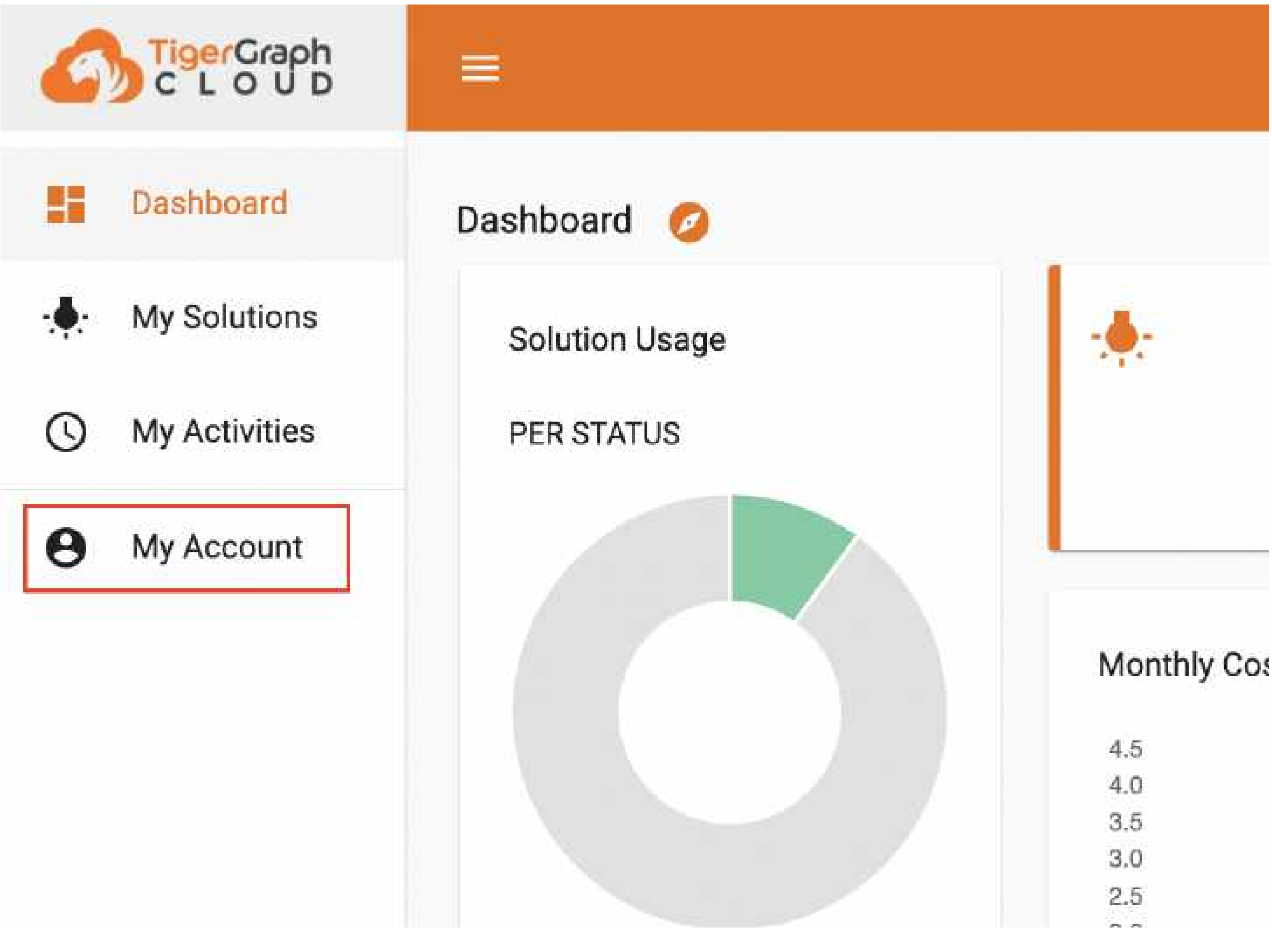
Enter your card information in the window that pops up, and click **Save**.

Delete a card

You can delete a card if there is more than one card associated with the account.

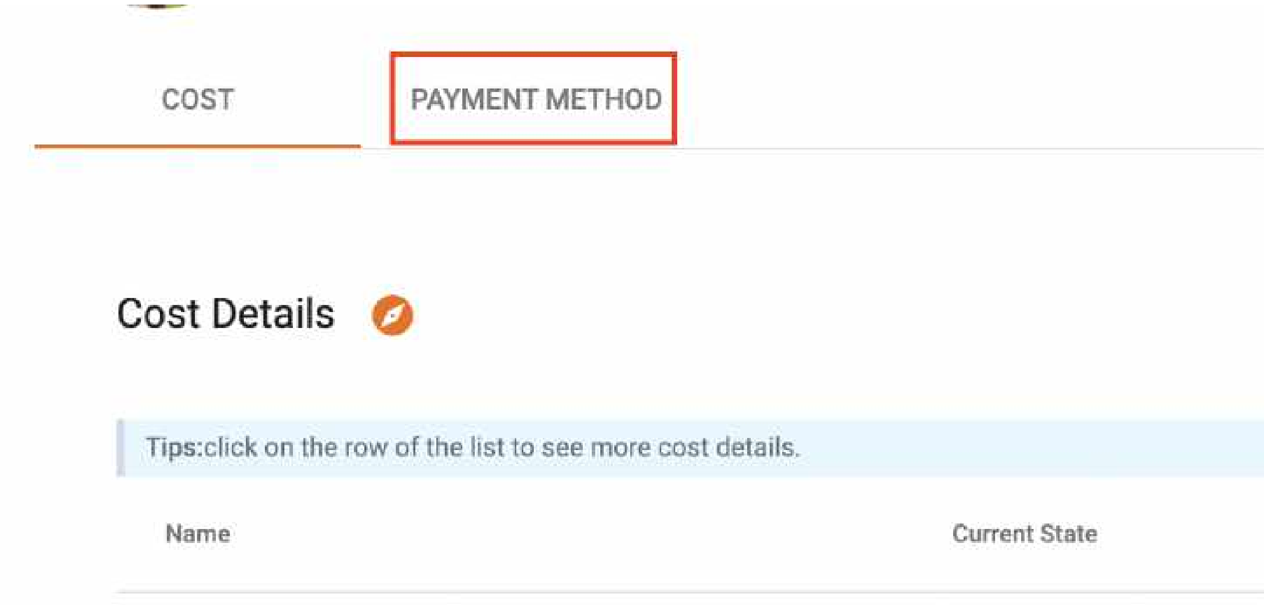
1. Navigate to My Account

On the left-side menu, click **My Account**.



2. Click Payment Method

On the My Account page, click **Payment Method.**



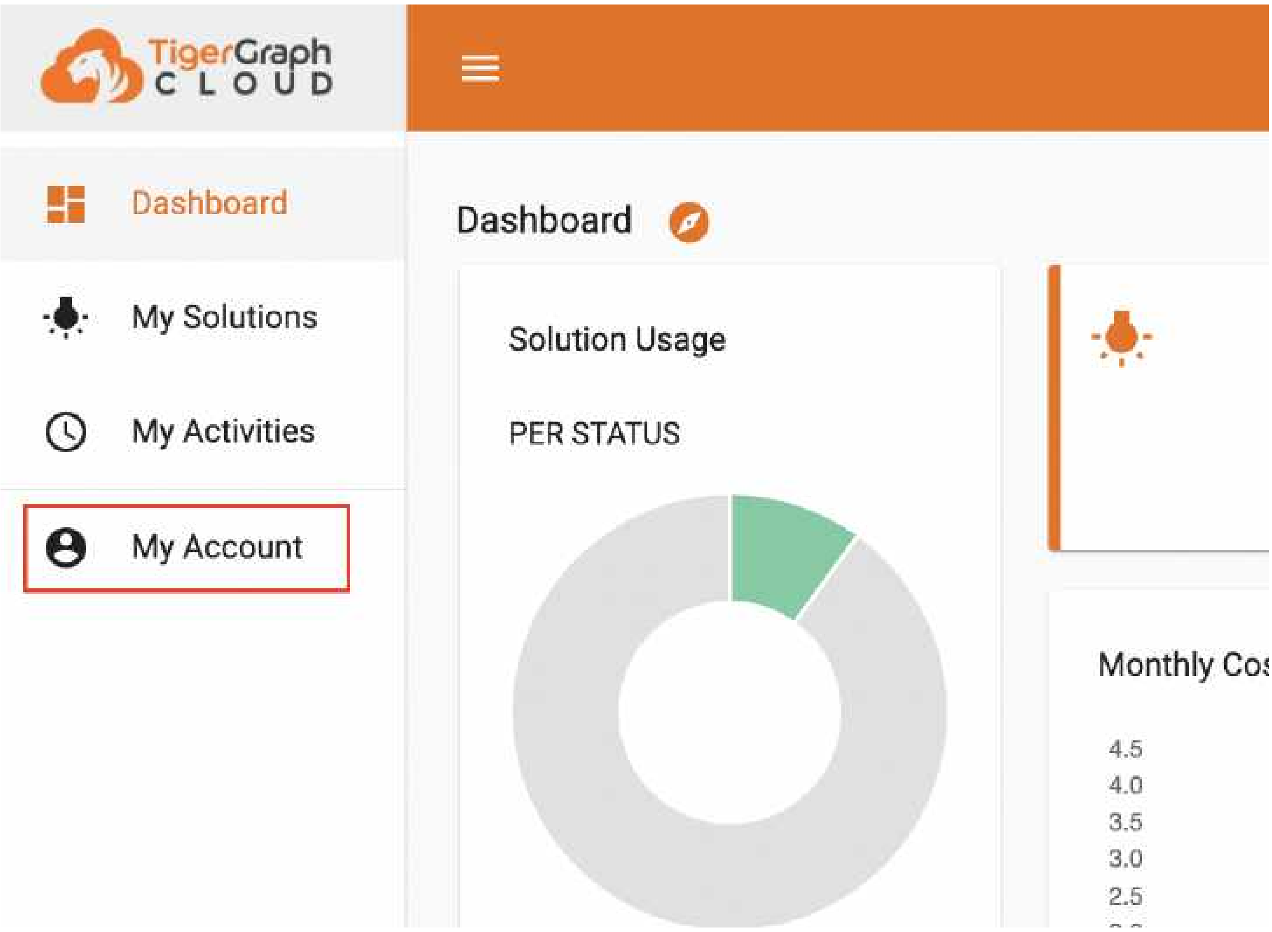
3. Delete the card

Find the card you want to delete in **Card Management**, and click **Delete**.

Set card as default

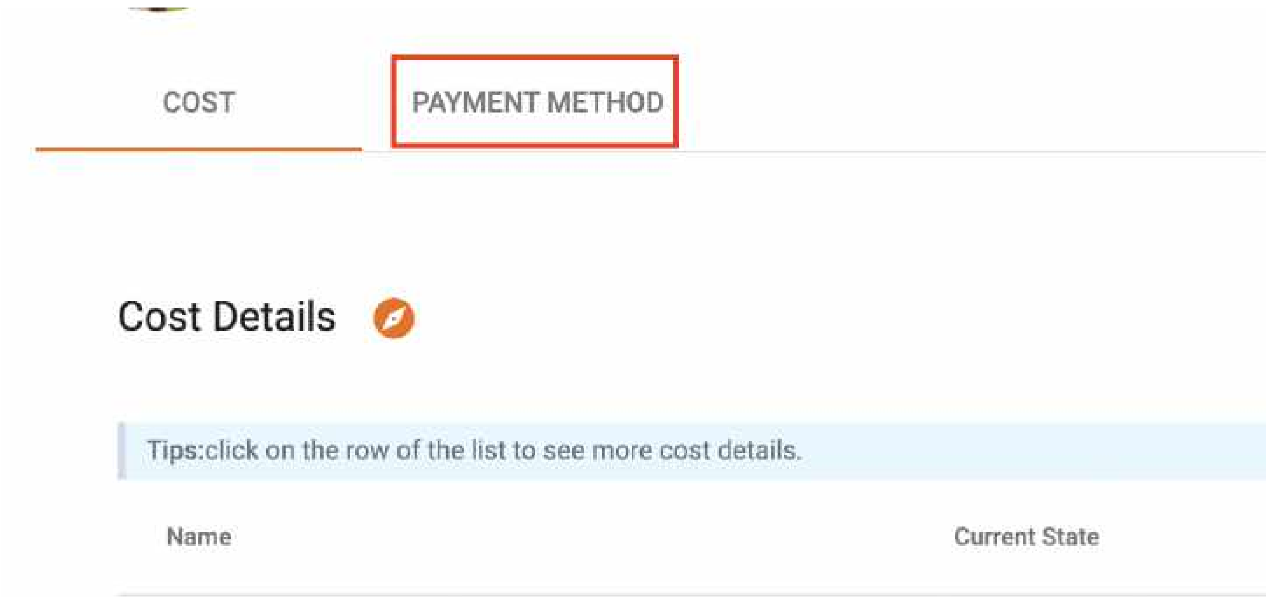
1. Navigate to My Account

On the left-side menu, click **My Account**.



2. Click Payment Method

On the My Account page, click **Payment Method.**



3. Click Set as Default

Find the card you want to set as the default card in **Card Management**, and click **Set as Default**.

Subscriptions

TigerGraph Cloud users can pay for TigerGraph Cloud via marketplace subscriptions through a cloud provider:

- AWS Marketplace ↗: Pay for TigerGraph Cloud via AWS
- Azure Marketplace ↗: Pay for TigerGraph Cloud via Azure
- GCP Marketplace ↗: Pay for TigerGraph Cloud via GCP

Release Notes

TigerGraph Cloud V3.10 - Sep 2021 Updates

Release Date: Sep 22, 2021

New features

- **TigerGraph V3.1.5 is now supported when creating new solutions.** Users can create solutions that include both single server and cluster deployment when they choose TigerGraph V3.1.5.
- **VPC Peering.** Users can now create a VPC peering connection between their TigerGraph Cloud VPC and another VPC on all supported cloud providers.
- **Automatic load balancing for HA clusters.** When users provision a cluster with replicas in TigerGraph Cloud, a load balancer will be automatically attached to the cluster.
- **Multi-AZ deployment.** When a user provisions an HA cluster, the replicas of the cluster will automatically be provisioned on different availability zones. This feature is supported on AWS and GCP.
- **Free-tier solutions on GCP.** We now offer the option to create a free-tier solution on GCP in addition to AWS.

TigerGraph Cloud V3.6 - May 2021 Updates

Release Date: May 17, 2021

New Features

- ✅ Support TigerGraph V3.1.1 for New Solutions (Single Server)

TigerGraph Cloud V3.5 - March 2021 Updates

Release Date: March 17, 2021

New Features

- ✅ Support GCP (Google Cloud Platform) as one of the Cloud Platforms

TigerGraph Cloud V3.4 - March 2021 Updates

Release Date: March 9, 2021

New Features

- ✅ Add Azure Central India and South India Regions to Cloud Platforms

TigerGraph Cloud V3.3 - Feb 2021 Updates

Release Date: February 17, 2021

New Features

- ✅ Add Cloud Portal Self Guided Tour
- ✅ Support Minimum Disk Size Check During New Solution Provisioning Process

TigerGraph Cloud V3.2 - Dec 2020 Updates

Release Date: December 16, 2020

New Features

- ✅ Pricing Change
  - Instance Pricing Change
  - Backup Pricing Change
  - Replica Pricing Change
  - New Data Transfer Cost (First 50GB free for 2 hour+ uptime per month)
- ✅ Support TigerGraph V3.0.6 for New Solutions
- ✅ Support TigerGraph v2.6.4 for New Solutions
- ✅ Support replication factor of 3 for TigerGraph V3.0.6 clusters
- ✅ New TigerGraph Cloud Instance Type TG.C4.M16 available from AWS
- ✅ SOC2 Type 2 report available to request
- ✅ 5 New TigerGraph Starter Kits
  - Graph Convolutional Networks (V3.0.6)
  - Healthcare Graph (Drug Interaction/FAERS) (V3.0.6)
  - Social Network Analysis (V2.6.4 and V3.0.6)
  - Enterprise Knowledge Graph (Corporate Data) (V2.6.4 and V3.0.6)
  - Enterprise Knowledge Graph (Crunchbase) (V2.6.4 and V3.0.6)

TigerGraph Cloud V3.1 - September 2020 Updates

Release Date: September 29, 2020

Patch

- ✅ Patch TigerGraph V3.0.5 with GSQL Bug Fixes for New Solutions

TigerGraph Cloud V3.0 - September 2020 Updates

Release Date: September 17, 2020

New Features

- ✅ Support TigerGraph V3.0.5 for New Solutions
- ✅ Support TigerGraph v2.6.3 for New Solutions
- ✅ Allow Users to choose v2.6.3 or V3.0.5 at New Solution Provisioning

TigerGraph Cloud V2.2 - July 2020 Updates

Release Date: July 10, 2020

New Features

- ✅ Support Microsoft Azure as one of the Cloud Platforms

TigerGraph Cloud V2.1 - April 2020 Updates

Release Date: April 27, 2020

New Features

- ✅ Advanced Developer Tool: GSQL Web Shell for Non-Free Tiers
- ✅ TigerGraph Database Version Upgrades to TigerGraph 2.6 For New Instances
- ✅ Starter Kit Categorization at Solution Provisioning
- ✅ 6 New TigerGraph Starter Kits
    - COVID-19 Starter Kit
    - In-Database Machine Learning Recommendation
    - Low-Rank Approximation Machine Learning
    - Graph Algorithms
        - Shortest Path
        - Centrality
        - Community Detection

TigerGraph Cloud V2.0

Release Date: January 08, 2020

New Features

- ✅ New product name: "TigerGraph Distributed Cloud"
- ✅ New URL: tgcloud.io
- ✅ Option to provision a TigerGraph Distributed Cluster
- ✅ Option to provision a Highly Available TigerGraph Cluster
- ✅ Added Steps in Cloud Portal Provision Workflow
- ✅ New TigerGraph Starter Kits

TigerGraph Cloud V1.0

Release Date: September 25, 2019

Deprecation Date: March 31, 2020 (See https://docs.tigergraph.com/tigergraph-cloud/tigergraph-cloud-v1-to-tigergraph-distributed-cloud-faqs ↗)

New Features

- ✅ Single Server Enterprise Edition
- ✅ Self-service Signup and Registration at tgcloud.us
- ✅ Free Credits for 1st Time Sign Up
- ✅ Available in 8 Tiers of Instances (Free Tier Included) and 6 Global Regions
- ✅ Flexible Billing and Pay As You Go Pricing
- ✅ Secure and Isolated Network (VPC) for Each TigerGraph Cloud Account
- ✅ Fast Data Loading through S3 and Local Files
- ✅ 13 TigerGraph Starter Kits with Sample Datasets and Queries
- ✅ Start/Stop/Terminate TigerGraph Solutions On Demand through Cloud Portal
- ✅ Built-in Backup and Restore through TigerGraph Admin Portal
- ✅ Rich Metrics from System Monitoring Panel on Cloud Portal
- ✅ Convenient Application Development Through RESTful Endpoints

FAQs

Feedback for TigerGraph Cloud is welcome at tgcloud-feedback@tigergraph.com

Top FAQs

Q: How does TigerGraph Cloud compare with other DBaaS offerings?

A: TigerGraph Cloud gets you up and running with the fastest and best price-performance graph platform in just minutes. TigerGraph's native parallel graph and deep link analytics give you both speed and scale, even on the most complex tasks. The GraphStudio visual design interface enables everyone on your team to be a guru in graph. And our starter kits for popular use cases mean you can have an example graph application working in minutes. Since it requires far fewer machines to achieve high performance, TigerGraph’s price performance sets a new bar for the graph database industry.

Q: How do I migrate to the new December 2020 pricing to lower my bill?

A: New December 2020 pricing applies to new paid tier solutions provisioned after December 16th 2020. You can keep the old pricing if you keep your paid tier instances provisioned before December 16th, 2020 and keep its state (running or stopped). If you want to take advantage of the new pricing to lower your cost, you can migrate your workload to a newly provisioned instance, or simply start and stop the old instance. Note that TigerGraph Cloud accounts with valid credit cards are eligible for the paid tier migration. Please submit a support ticket for migration assistance.

Q: Are there Free Tier Instances or Free Credits for the System?

A: Yes, in both instances.

Free Credits: A $25 credit will be automatically granted to each new registered account. The free credits is valid for 30 days after initial use.

Free Tier Instance: When you select an instance type, you will see that one instance type is designated as the Free Tier. For each registered account, you may provision one solution from the Free Tier. Free Tier instances do not include backup and do not include support.

If no user activity is detected for more than one hour, TigerGraph may automatically stop a Free Tier instance. Users can manually restart the free tier instances from their cloud portal. After 7 days of inactivity, TigerGraph may terminate the inactive free tier instances. For additional information, see TigerGraph Cloud Terms ↗.

Tip: If you need to save your work from a Free Tier instance, export the solution (which saves your graph and queries) and write queries which print all your data to files.

Q: I cannot start my free tier instances. Why is there a capacity error in "My Activities"?

A: **Known Capacity Issues:** Free Tier instances are provisioned on a common instance type in a data center region of the cloud provider (AWS). During the surge cloud usage period, such as the current global lockdown due to COVID-19, cloud providers across the globe are experiencing capacity issues.

It is possible that when you provision a free tier instance, it will not provision successfully on the first try because of the peak usage in a certain region in the backend cloud platform (AWS). Please be patient and try to provision in another region, or at a later time when there is enough capacity in the region. For any stopped free tier instances, when you restart the solution, it is possible that your solution cannot restart with a machine because of the capacity issue from the cloud provider. In this case, try to restart the solution at a later time when there are more machines available in the region of the cloud provider (AWS). In both scenarios, you can look at the log in "My Activities" and the capacity issue is logged. Submit a support ticket if the capacity issue persists.

To reserve an instance for guaranteed capacity, please submit a support ticket and contact sales@tigergraph.com ↗ to arrange a long term contract for reserved TigerGraph instances.

Q: In what cloud platforms and regions does TigerGraph operate?

A: TigerGraph Cloud offers three cloud platform options: AWS, Azure and GCP. In AWS, supported regions are: US-West-1 (N.Cal), US-East-1 (N. Virginia), EU-West-1(Ireland), EU-West-2(UK), EU-Central-1(Germany) and AP-Northeast-1(Japan). In Azure, supported regions are: West US (California, US West), West Europe (Amsterdam, Netherlands), Central India (Pune, India), South India (Chennai, India). In GCP, supported regions are: Europe-West-2 (London, UK), US-West-1 (oregon, US).

More regions and platforms will be added soon.

Q: Can I choose different versions of TigerGraph on TigerGraph Cloud?

A: Yes. When you create a solution, you can choose the database version you would like to use.

Q: Does TigerGraph Cloud support distributed databases?

A: Yes. In the latest version of TigerGraph Cloud, you can provision a highly available TigerGraph cluster by entering 2 or 3 for the replication factor during the provisioning process. Replication factor of 2 and 3 are supported for TigerGraph V3.0+. For TigerGraph V2.6, only replication factor of 2 is supported. The configuration is active-active, meaning that all copies of the data are available to answer queries.

NOTE: HA systems must have a minimum of 3 instances. This means that the smallest supported cluster configuration for a replicated system for TigerGraph V3.0+ is one-way partitioning X three-way replication = three instances. In comparison, the smallest supported cluster configuration for a replicated system for TigerGraph V2.6 is two-way partitioning X two-way replication = four instances.

Q: What type of server should I use for my data size and workload?

A: Please see the section on pricing on our website at www.tigergraph.com/cloud/ ↗. For further assistance, contact TigerGraph at sales@tigergraph.com ↗

Q: What kind of services do I get?

A: TigerGraph Cloud includes automatic scheduled backup, built-in encryption and other security features, patching, replication and distributed database option. In addition, you can now provision a distributed database as well as a replica cluster for high availability. As a cloud service, many of the administrative and operational tasks - for monitoring, restoring, upgrading, for example - are just a click away. In addition to operations and management for your TigerGraph databases, TigerGraph Cloud also offers Starter Kits to provide instant experience to various graph analytics use cases. The list of Starter Kits can be found here: https://www.tigergraph.com/starterkits/ ↗

Q: Is there a quota for the number of solutions I can provision in a single account?

A: Yes. For basic accounts with credit card billing, there is a limit of 160 vCPUs and 20 instances. A solution is one logical database, which may include several instances due to a distribution database configuration and replication. For accounts running exclusively on Free Credit and that have not yet entered a valid credit card, there is a limit of 16 vCPUs and two solutions. For each registered account (with or without a valid credit card), you may only provision one solution from the Free Tier. To allow more vCPUs and solutions, please contact TigerGraph at sales@tigergraph.com ↗.

Total number of instances in a cluster = replication factor x partitioning factor

	Account With a Valid Credit Card	Account Without a Valid Credit Card
Free Tier Solution Quota	1	1

Total Instance Quota	20	2
Total vCPU Quota	160	16
Cluster Size Limit	10 instances	1 (that is, distributed data is not enabled)
Replication Factor Limit	2	1 (that is, HA is not enabled)
Request Quota/Limit Increase	Yes, Please contact sales@tigergraph.com ↗.	N/A Please upgrade your account by entering a valid credit card.

Q: Can I run queries directly after sample starter kits are provisioned?

A: Data must be loaded and queries must be installed first. Please perform the following steps after the solution is provisioned:

1. Connect to GraphStudio through “Open Solution Via Domain” from the cloud portal.
2. On the Load Data tab, click the Load button to load the sample dataset.
3. On the Write Query tab, click the Install button to install the sample queries.

Now you can run queries on the starter kit's sample data. Please visit tigergraph.com/starterkits ↗ to watch the overview video for each starter kit.

Q: How can I monitor my TigerGraph Cloud service?

A: TigerGraph Cloud is instance-based and offers an administrator portal to monitor the performance and health of each machine instance.

Q: Is TigerGraph Cloud cloud-agnostic?

A: TigerGraph Cloud will provide teams with the flexibility to use the cloud vendor of their choice, so there will be no vendor lock-in. For the current version, TigerGraph Cloud offers instances on AWS, Azure and GCP platforms. If you require immediate assistance to manage TigerGraph on another cloud provider, please contact sales@tigergraph.com ↗.

Q: Is the support for TigerGraph Cloud the same as the support for TigerGraph Enterprise?

A: Yes, TigerGraph supports TigerGraph cloud users using paid tiers. See the Support Policy terms at https://www.tigergraph.com/support-policy/ ↗. For free tier instances, support is not included; therefore, support tickets for free tier solutions are answered when support staff bandwidths allows. See Section 1.4 in https://www.tigergraph.com/tigergraph-cloud-subscription-terms/ ↗. Additional developer resources for free tier users: join TigerGraph developer community at https://community.tigergraph.com ↗.

Q: What is the user interface for the TigerGraph Cloud instance?

A: The TigerGraph GraphStudio™ UI (User Interface) ↗ provides an intuitive, browser-based interface that helps users get started quickly with graph-based application development tasks: designing a graph schema, creating a schema mapping, loading data, exploring the graph, and writing GSQL queries.

In addition, for TigerGraph paid tier solutions, users can use GSQL Web Shell to write GSQL Commands in an interactive command line shell. Users can access GSQL Web Shell through the solution panel from their Cloud Portal. This GSQL Web Shell opens up more data loading options such as S3 Loader (parquet format) and Kafka Loader.

Q: What graph query language does TigerGraph support?

A: TigerGraph uses GSQL, the query language designed for fast and scalable graph operations and analytics. GSQL's similarity to SQL, high-level syntax, Turing completeness, and built-in parallelism brings faster performance, faster development and the ability to describe any algorithm.

You can start learning GSQL from our tutorial GSQL 101. We also support a RESTful API and JSON output for easy integration with application languages like Python, Java, and C++.

Start learning GSQL and become a TigerGraph Certified Associate today: www.tigergraph.com/certification/ ↗

Q: Can I have multiple graphs in one TigerGraph Cloud instance?

A: Yes. If you use V3.0.5+ in TigerGraph Cloud, MultiGraph is supported through GraphStudio. Please refer to https://docs.tigergraph.com/ui/graphstudio/design-schema ↗ for the new MultiGraph Support through GraphStudio starting TigerGraph V3.0.

Q: What methods do you support for importing data?

A: We support AWS S3 import and local file upload through GraphStudio. Spark loading is available through our open source JDBC Driver. See https://github.com/tigergraph/ecosys/tree/master/etl ↗

In addition, for TigerGraph paid tier solutions, users can use GSQL Web Shell to write GSQL Commands in an interactive command line shell. This GSQL Web Shell opens up more data loading options such as S3 Loader (parquet format), Kafka Loader and other complex loading jobs. Users can access GSQL Web Shell through the solution panel from their Cloud Portal.

For complex loading jobs through GSQL Web Shell for paid tier solutions, please submit a support ticket from your cloud portal for more information.

Q: If I don't enter a valid credit card in the account and use the initial $25 free credit, will the data in the provisioned solutions be deleted if I stop the solutions?

A: As long as there is remaining free credits in your account, the data of your solutions will be preserved by the provisioned disk after you stop the solutions even if you don't have a credit card in your account. The data will be available and accessible after you restart the solutions. The free credits is valid for 30 days after initial use. Your solutions will be deleted automatically if the following scenario is detected: there is no remaining valid free credits and there is no valid credit card in your account.

To use “backup and restore” functionality, you need to enter a valid credit card in your account and choose non free tier instances. The non free tier solutions provisioned after the credit card is entered have backup and restore functionality through TigerGraph’s Admin Portal. Note that any solutions provisioned before entering the credit card will not be upgraded with the backup and restore functionality.

If you choose to terminate the instance, your solutions and the data will not be preserved. Please see "Provisioning, Backup and Restore" section for detailed explanation on the difference between stopping and terminating a solution.

Q: I can’t find my registration activation email after registering for the first time. Where can I find it?

A: Please check your SPAM folder. The activation email could be sent to your SPAM folder. If you can't find it in SPAM folder, and does not receive verification email in your mailbox after resending the verification email, please contact support@tigergraph.com ↗ so that we can assist you with manual email verification process.

Q: How do I develop a customized application on top of TigerGraph Cloud?

A: Please see Graph Gurus episode 24 ↗, where we presented "How to Build Innovative Applications with a Native Graph Database".

Q: Can I upgrade from V2.6.x, V3.0.5, V3.0.6 to V3.1.1?

A: For free tier solutions, the upgrade is not supported unless you are migrating to paid tier solutions. If you want to change the database version in your free tier, you can terminate the existing free tier solution, and create a new free tier with the new version. In each TigerGraph Cloud account, you can have one free tier. If you want to migrate to paid tier solutions, please send a support ticket through your cloud portal.

For paid tier solutions, please submit a support ticket for upgrade assistance. Downtime is expected during this upgrade. It is recommended to create a new blank V3.1.1 solution, import your data and copy the GSQL queries to test and experiment before upgrading your production instance from V2.6.x, V3.0.5, V3.0.6 to V3.1.1.

Pricing

See www.tigergragraph.com/cloud ↗ for pricing information.

Q: How do I migrate to the new December 2020 pricing to lower my bill?

A: New December 2020 pricing applies to new paid tier solutions provisioned after December 16th 2020. You can keep the old pricing if you keep your paid tier instances provisioned before December 16th, 2020 and keep its state (running or stopped). If you want to take advantage of the new pricing to lower your cost, you can migrate your workload to a newly provisioned instance, or simply start and stop the old instance. Note that TigerGraph Cloud accounts with valid credit cards are eligible for the paid tier migration. Please submit a support ticket for migration assistance.

Q: Do you offer a discount for annual contracts/commitments?

A: Yes, there is a discount for contracting with TigerGraph for a one-year term. Please contact sales@tigergraph.com ↗ to request your discount.

Q: Can I purchase TigerGraph Cloud Credits?

A: Yes. You can purchase cloud credits applied to TigerGraph Cloud usage. Please refer to https://www.tigergraph.com/cloud-credits/ ↗

Q: If I delete my instance before the end of the month, what will I have to pay?

A: If you are subscribed to our standard hourly service, you will be charged only for your hours of use, as described above.

Q: If I add an instance mid-month, when do I start paying for it?

A: We bill you and charge your credit card at the end of each calendar month.

Q: If I have a billing problem that I can’t solve online, how do I contact you?

A: You can either open a support ticket from the cloud portal by clicking “Support” on the menu at the left of the page, or you can send an email to billing@tigergraph.com ↗.

Q: What happens if my monthly costs exceed the credit limit on my credit card?

A: If this happens you can contact sales@tigergraph.com ↗ and we will provide you with other payment options.

Q: Where can I find TigerGraph's terms and conditions for its products and services?

A: You can find our terms and conditions here: www.tigergraph.com/terms ↗.

Q: How is TigerGraph Cloud priced for development, test, or QA environments?

A: With TigerGraph Cloud, you only pay for what you use. For long term contracts, please contact sales@tigergraph.com ↗.

Sizing and Scaling

Q: How do I determine the instance needed for my workload?

TigerGraph Distributed Cloud offers eight different instances in AWS platform, ranging from 4 vCPUs with 7.5 GiB RAM, to 96 vCPUs with 768 GiB RAM. In Azure, TigerGraph Distributed Cloud offers seven different instances, ranging from 4 vCPUs with 16 GiB RAM, to 96 vCPUs with 672 GiB RAM. In GCP, TigerGraph Distributed Cloud offers seven different instances, ranging from 4 vCPUs with 16 GiB RAM, to 80 vCPUs with 640 GiB RAM. Larger graphs require more RAM. Higher performance calls for both more CPUs and more RAM.

Our pricing table ↗ gives initial recommendations for which instance to start with, if you know how much data you will be loading into the graph. We call the input data your "raw data". TigerGraph reorganizes your data into a graph, encoding and compressing it. We have assumed that your stored graph will be about 70% of the size of your raw data. This is a conservative estimate; it's often smaller.

You also need RAM for your graph querying and computation. Workloads vary considerably. In the pricing table, we recommend that your total RAM be about 1.5 times the maximum amount of raw data you think you will load.

If you see you need more space or more compute power, then just scale up.

Please submit a support ticket from TigerGraph Cloud Portal or contact support@tigergraph.com ↗ for more help.

Q: How do I expand the disk size?

Please submit a support ticket from TigerGraph Cloud Portal or contact support@tigergraph.com ↗ for more help.

Q: Can I migrate my database to a larger (or smaller) instance?

A: Contact TigerGraph Cloud Support for migrating between different instance types. Instance migration is not yet supported through one-click operation, however, migration can be achieved by using backup and restore for some cases.

Q: Can I deploy a distributed TigerGraph database across a cluster of instances?

A: Yes, you can provision a distributed database in TigerGraph Cloud. Simply provide the partition number and replication number you would like to have. Please see High Availability and Replication ↗ Section below.

Hardware

Q: What type of disks are attached to the provisioned TigerGraph Instances?

A: A Root Disk (EBS based) is attached to TigerGraph Instances.

Q: What does the provisioned disk include, and how should I approach disk sizing?

A: If you choose AWS as the backend, TigerGraph Cloud provisions persistent EBS volume as the disk attached to the EC2 machines to hold installation files and data files. The disk holds the following: OS installation, TigerGraph database installation, TigerGraph Graph storage, other data files uploaded to the instance, and any output files generated by querying the database, in addition to system logs generated during the process. Hence, when you consider the size of the disk, please consider the size of the data for all the above installations and files.

Q: What browsers are supported?

A: TigerGraph Cloud has been designed and tested for Chrome. Other browsers may not yet be fully supported.

Provisioning, Backup and Restore

Q: What is a Solution?

A: A TigerGraph Solution is a graph database image which can be deployed on a virtual machine instance. Most solutions also come with a starter kit, a sample graph schema, sample data, and sample queries for a common use case, such as Recommendation Engine, Anti-Fraud, and Healthcare Analytics. In a newly provisioned solution, the data files are ready to be loaded, and the queries are ready to be installed.

Q: Is there a warm-up period for TigerGraph instances and solutions?

A: When you provision or restart a solution, there will be a warm-up period for the machine instances and the disk attached to the instances. The larger the data size and the greater the number of instances, the longer the warm-up period will be.

Q: What is the difference between stopping and terminating a solution?

A: When you stop a solution, you shut down the virtual machine instance. The TigerGraph database is frozen in its current state. Billing for that machine instance also stops. Scheduled backups also stop, but the backup copies are kept in EFS in AWS, Azure Files in Azure, and GCP Filestores in GCP. You will still incur charges for disk storage and backups when a solution is stopped. You can start the solution again. After a warm up period, the solution will return to its previous state.

When you terminate a solution, you will deprovision the virtual machine and the attached disk space. The solution is deleted according to the cloud platforms policies and cannot be recovered. A backup file, however, can be used to restore to a new solution within 15 days.

Q: Do you offer backup in TigerGraph Cloud?

A: Yes. In GraphStudio, go to Admin Portal > Backup and Restore.

Q: What backup options are available?

A: TigerGraph Cloud offers full backups on a scheduled or on-demand basis. By default, backup is turned on when instances are running and will be done daily. There are four options you can choose from: backup daily, weekly, monthly, and ad hoc. In current version, the retention policy is to retain up to seven backups. Therefore, if you choose to backup daily; the retention of backup is for a week; if you choose to backup weekly, the retention of backup is for seven weeks; if you choose to backup monthly, the retention of backup is for seven months. if you want to perform an ad hoc manual backup when there are already seven copies of backups, you need to delete an older version of backup in order to save the most recent ad hoc copy. The maximum number of manual backups is six, as the platform always reserves one spot for scheduled backup. When a solution is stopped, backup in EFS, Azure Files and GCP Filestores will be charged for the time you have the solutions. Please see pricing for backup cost while your instance is stopped.

For a longer retention policy, more options will be coming soon.

Q: Can I restore from a backup from another solution in my account?

A: Only single server solutions can be restored from the backup of another solution in the same account within the same cloud platform. Distributed and replicated cluster cannot be restored from the backup of another cluster in the same account.

Q: What password do I use for backup and restore?

A: To perform an ad hoc backup, use the password of the tigergraph user of your solution. To restore from a previous backup, use the same password that was used to create the backup. This rule applies to restoring from a different solution in your account. All the backups of different solutions in your account can be found through the Admin Portal.

Q: What happens to my automated backups if I terminate my solution?

A: We will keep the latest backup for 15 days and then delete. Within that 15 days, you may use your backup to restore into another similar solution if it is a single server solution. To restore a cluster within 15 days, please submit a support ticket through the cloud portal.

Q: How soon can I restore from a backup that I just made?

A: After you perform a backup, you should wait at least 15 minutes.

Q: How can I delete my TigerGraph Cloud account?

A: Please submit a support ticket through the cloud portal.

High Availability and Replication

Q: What kind of replicas does TigerGraph Cloud support?

A: TigerGraph Distributed Cloud offers active-active replication, for increased availability and automatic failover.

A TigerGraph system with HA is a cluster of server machines which uses replication to provide continuous service when one or more servers are not available or when some service components fail. TigerGraph HA service provides loading balancing when all components are operational, as well as automatic failover in the event of a service disruption.

Q: What is the replication factor?

A: Replication factor means how many copies of data are stored, each on a separate machine. The default HA configuration has a replication factor of two, meaning that a fully-functioning system maintains two copies of the data, stored on separate machines. TigerGraph Distributed Cloud currently supports your choice of replication factor of one (only one copy of the data, not recommended for critical systems), two. or three if you choose TigerGraph v3.0.6+. If you choose TigerGraph v2.6.x, only replication factor of one or two are supported.

Limitation for TigerGraph v2.6.x: In TigerGraph Distributed Cloud, if replication is used, the total number of instances must be at least 3. For TigerGraph v2.6.x, If replication factor is 2, then the partition factor must be at least 2, for a total of 2×2 = 4 instances. For TigerGraph v3.0.6+, the configuration for a cluster with 1 partition and 3 replicas is supported.

Q: What is the partition factor?

A: Partition factor means the number of parts or components your graph data is split into, which also equals the number of instances that collectively store one copy of the full graph. For example, if you select a partition factor of 3, each instance will hold approximately 1/3 of your data. Please read https://docs.tigergraph.com/admin/admin-guide/installation-and-configuration/ha-cluster#examples ↗ for additional details about partitions and replications.

Q: How long do I need to wait for the cluster to be ready after provisioning process starts?

A: It currently takes about 4 minutes to provision a single instance. If you configure a 2×2 replicated and distributed graph database, it will take about 15 minutes.

Network, Security and User Management

Q: Can I use TigerGraph in Amazon Virtual Private Cloud (Amazon VPC)?

A: By default, you will be given your own VPC(s) for your TigerGraph Cloud account resources in AWS and GCP, and your own Azure Virtual Networks for your TigerGraph Cloud account resources in Azure. Your instances are separated from other accounts by different VPCs or Virtual Networks. Within your own account, you have different VPCs or Virtual Networks for different regions.

Q: Does TigerGraph Cloud support encrypting my data in transit and at rest?

A: Yes, TigerGraph Cloud encrypts all data in transit and at rest.

Q: Am I sharing data storage with other customers? Is TigerGraph a multi-tenant solution?

A: You are not sharing storage with other customers. Each TigerGraph solution is provisioned as one or more virtual machine instances of the TigerGraph engine, used only for your account, and provisioned with its own disk space. No two accounts are sharing the same TigerGraph database.

Q: How do I log in to the GraphStudio UI of my solution for the first time?

A: You provide the initial password through provisioning steps in your cloud portal. Hence, remember the initial password you provided to your tigergraph user. To log in to GraphStudio for the first time, use tigergraph/<initial password>.

Q: How do I change the password to my solutions?

A: You provide the initial password through provisioning steps in your cloud portal. Hence, remember the initial password you provided to your tigergraph user. To change the password, you need to log in to GraphStudio using tigergraph/<initial password>, then go to Admin Portal. The ability to change passwords is provided through administrator portal. You can access this page by GraphStudio > Admin Portal > User Management.

Q: Can I create multiple users with different roles to access TigerGraph solutions?

A: If you choose V3.0.5+, the default tigergraph user with superuser role can create other users with different roles through GraphStudio > Admin Portal > User Management for each solution. Please read the following documentation for more steps and information: https://docs.tigergraph.com/ui/admin-portal/user-management ↗

Additional information on Role Based Access Control can be found here: https://docs.tigergraph.com/ui/graphstudio/user-access-management ↗

Q: How do I access my TigerGraph Cloud account (e.g., username and password)?

A: When you register your account, you will select a username and password. You can then log in anytime at www.tgcloud.io ↗. You will also be given a URL, using a subdomain name that you select.

Q: How do I access my TigerGraph database and POST to TigerGraph?

A: You can access the database through TigerGraph's GraphStudio visual interface and through RESTful endpoints. Use RESTful endpoints to POST to TigerGraph solutions and develop applications. Please refer to the RESTful API User Guide ↗. To find the RESTful endpoints for queries created in GraphStudio, please read "Show Query Endpoint ↗". There is also a recorded webinar which demos the process in details: https://info.tigergraph.com/graph-gurus-24 ↗

Here is the step-by-step instructions:

TigerGraph cloud enables REST++ Authentication ↗ to securely connect TigerGraph Cloud solutions with your application through port 9000.

Step 1: [One time] Navigate to the TigerGraph solution's Admin Portal, and generate a secret from User Management.

For example, the URL for the solution is:

```
https://SOLUTIONID.i.tgcloud.io:14240/admin/#/user-management
```

and the generated secret is abcd1234 from Admin Portal.

Step 2: [Need to renew every lifetime] Use the secret generated in step 1 to get a RESTPP token (for example, xyz789) using curl command. Note that the port is 9000.

Here is an example where you obtain a token with a lifetime of 1,000,000 seconds (11 days):

```
curl -X GET 'https://SOLUTIONID.i.tgcloud.io:9000/requesttoken secret=abc
```

```
{
  "code":"REST-0000",
  "expiration":1570727825,
  "error":false,
  "message":"Generate new token successfully.",
  "token":"xyz789"
}
```

Step 3: Now in your application, use the token in the REST call, for example:

```
curl -X GET -H "Authorization: Bearer xyz789" 'https://SOLUTIONID.i.tgclo
```

Q: Do free tier instances expose RESTful endpoints?

A: Yes. Free tier instances expose RESTful endpoints through port 9000 to allow access to TigerGraph database, similar to non free tier instances.

Q: Does TigerGraph Cloud offer Role Based Access Control?

A: TigerGraph's role-based access control with MultiGraph and User Management is available if you use TigerGraph V3.0.5+.

Q: What happens to my data if I terminate an instance or if my account is closed?

A: When you terminate an instance in TigerGraph Distributed Cloud, the virtual machine instance and its associated storage volume are deleted according to the policies of the underlying cloud infrastructure vendor.

Q: How does TigerGraph Cloud secure my data?

A: TigerGraph Cloud encrypts data at rest and in transit, and SSL is enabled for secure access.

Q: Can I integrate TigerGraph Cloud into my single sign on system?

A: The ability to use cloud portal to integrate TigerGraph Cloud into an SSO system will be provided at a future date. If you are using a paid tier instance, please submit a support ticket to request advanced service to integrate into an SSO system TigerGraph supports. Please see https://docs.tigergraph.com/admin/admin-guide/user-access-management/single-sign-on ↗

https://docs.tigergraph.com/ui/admin-portal/management/security/sso ↗

Logs

Q: Does TigerGraph Cloud provide logs?

A: Access to TigerGraph system and component logs is coming soon via the TigerGraph Cloud portal and administrator portal for provisioned TigerGraph instances.

Upgrade

Q: Can I upgrade from V2.6.x, V3.0.5, V3.0.6 to V3.1.1?

A: For free tier solutions, the upgrade is not supported unless you are migrating to paid tier solutions. If you want to change the database version in your free tier, you can terminate the existing free tier solution, and create a new free tier with the new version. In each TigerGraph Cloud account, you can have one free tier. If you want to migrate to paid tier solutions, please send a support ticket through your cloud portal.

For paid tier solutions, please submit a support ticket for upgrade assistance. Downtime is expected during this upgrade. It is recommended to create a new blank V3.1.1 solution, import your data and copy the GSQL queries to test and experiment before upgrading your production instance from V2.6.x, V3.0.5, V3.0.6 to V3.1.1.

TigerGraph Cloud Support

Q: How do I submit a support ticket?

A: In your cloud portal, on the top bar, you can submit the support ticket by clicking on . This will generate a support ticket in TigerGraph's freshdesk support system. You can track the ticket status in freshdesk.

Q: How do I submit a support ticket if I cannot access the Cloud Portal?

A: If in rare cases, you cannot submit TigerGraph Cloud support tickets through the cloud portal for your account, you can also submit the support request by email to support@tigergraph.com ↗ directly. This action will automatically generates a new freshdesk support ticket and you can track the support ticket in freshdesk portal. Please remember to use the email account associated with your TigerGraph Cloud account to submit the support email to support@tigergraph.com ↗ so that TigerGraph Cloud Support can verify your account.

Performance

Q: Is TigerGraph built on a NoSQL database or a relational database?

A: TigerGraph is a native parallel graph database built on C++. It is not built on a NoSQL database or relational database.

Q: How can I monitor the speed or throughput of queries and data loading?

A: The GraphStudio visual design tool provides several monitors. The Load Data ↗ page includes a real time monitor and statistics. Query performance and many other measures are available on the administrator portal ↗.

Q: How can I improve the speed of my system?

A: Due to TigerGraph's massively parallel and hybrid in-memory database design, an instance with more vCPUs and more memory will usually run faster. For a given hardware configuration, performance can be improved by optimizing graph schema, loading jobs, and queries. In TigerGraph Distributed Cloud, you can also choose to provision a cluster with replication factor 2 to increase throughput. Contact sales@tigergraph.com ↗ to discuss for query optimization services.

Q: What third-party software is used in TigerGraph Cloud?

A: A list of third-party software used in the TigerGraph engine and TigerGraph Cloud is available at https://docs.tigergraph.com/legal/patents-and-third-party-software. ↗

Support

TigerGraph provides support for TigerGraph Cloud for customers with non-free-tier solutions. For more information on TigerGraph's support policy, visit our support policy page ↗.

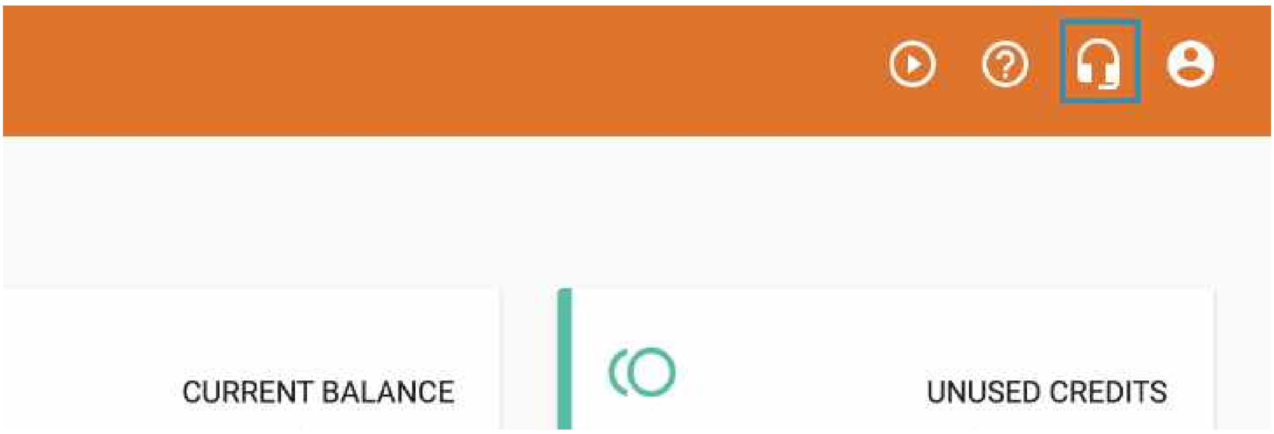
Submit a support ticket

You can submit a support ticket in the TigerGraph Cloud portal or by email.

1. Submit a support ticket

Click on the support icon, and fill out the form that pops up with information relevant to your issue. If you cannot access the TigerGraph Cloud portal, you can also email support@tigergraph.com to submit your support ticket. Either approach will automatically generate a support ticket in TigerGraph's FreshDesk support system ↗, and you will receive an email with the link to your ticket.

> Please remember to use the email account associated with your TigerGraph Cloud account to submit the support email to support@tigergraph.com ↗ so that TigerGraph Cloud Support can verify your account.



2. Setup your FreshDesk account (first ticket submission only)

> If you already have a FreshDesk account with TigerGraph, you can skip this step.

When you submit a support ticket through TigerGraph Cloud or through email, and you don't already have a FreshDesk account with TigerGraph, an account will be created for you automatically. An account setup email titled **Welcome to TigerGraph Support** will be sent to you at the email address associated with your TigerGraph Cloud account, and you can follow the instructions in the email to set up your account.

This setup email might arrive a few minutes later than the email containing the link to your ticket.

3. Follow up on your ticket

If you have a paid subscription with TigerGraph, a support representative will be reviewing your request and will send you a personal response (usually within 24 hours). You can follow up on your ticket either by replying to the email thread or in FreshDesk.

Reference

Amazon Web Services (AWS)

Regions

- US-West-1 (N.Cal)
- US-East-1 (N. Virginia)
- EU-West-1 (Ireland)
- EU-West-2 (UK)
- EU-Central-1 (Germany)
- AP-Northeast-1 (Japan)

Instance types

The table below shows the instance types available on AWS.

Instance name	vCPU	RAM (GiB)	Price per hour
TG.Free	4	7.5	0
TG.C4.M16	4	16	$1.20
TG.C8.M15	8	15	$1.80
TG.C8.M32	8	32	$2.20
TG.C16.M64	16	64	$4.10
TG.C16.M122	16	122	$7.00
TG.C32.M244	32	244	$12.00
TG.C64.M488	64	488	$21.00
TG.C96.M768	96	768	$30.00

Disk storage

On AWS, disk storage refers to the EBS volume ↗ attached to your TigerGraph Cloud instance. By default, data in disk storage is encrypted-at-rest with platform-managed keys.

Disk Storage is $0.15 /GB/month, equivalent to $0.0002083 /GB/hour

Backup storage

Solution backups on AWS are stored on Amazon Elastic File System ↗ (EFS) and automatically encrypted at rest with platform-managed keys. When a solution is provisioned, backup storage is part of the running cost of the solution and only stops incurring charges when the solution is terminated.

Backup Storage is $1.25 /GB/month = $0.001736 /GB/hour.

Microsoft Azure

Regions

- West US
- West Europe
- Central India
- South India

Instance types

Instance name	vCPU	RAM (GiB)	Price per hour
TG.C8.M16	8	16	$1.80
TG.C8.M32	8	32	$2.20
TG.C16.M64	16	64	$4.10
TG.C16.M128	16	128	$7.00
TG.C32.M256	32	256	$12.20
TG.C64.M432	64	432	$18.50
TG.C96.M672	96	672	$26.00

Disk storage

On Azure, disk storage refers to Premium SSD Managed Disks ↗. Encryption is turned on by default and uses platform-managed keys.

Disk Storage is $0.15 /GB/month, equivalent to $0.0002083 /GB/hour

Backup storage

Solutions deployed on Azure use Azure Files ↗ for storage. By default, backups are encrypted and decrypted transparently using 256-bit AES encryption ↗, one of the strongest block ciphers available, and is FIPS 140-2 compliant.

Backup Storage is $1.25 /GB/month = $0.001736 /GB/hour.

Google Cloud Platform (GCP)

Regions

- Oregon
- London

Instance types

Instance name	vCPU	RAM (GiB)	Price per hour
TG.Free	2	8	0
TG.C4.M16	4	16	$1.20
TG.C8.M32	8	32	$2.20
TG.C16.M64	16	64	$4.10
TG.C16.M128	16	128	$7.00
TG.C32.M256	32	256	$12.20
TG.C64.M512	64	512	$22.00
TG.C80.M640	80	640	$25.00

Disk storage

Disk storage on GCP uses GCP's SSD persistent disks ↗. Encryption-at-rest is turned on with GCP's native encryption protocol with keys managed by GCP.

Disk Storage is $0.25 /GB/month, equivalent to $0.000347222 /GB/hour.

Backup storage

Backup storage on GCP uses GCP Filestore ↗. Encryption-at-rest is turned on with GCP's native encryption protocol with keys managed by GCP.

Backup storage is $2.1/GB/Month = $ 0.0029166 /GB/hour.

Service Limits

By default, TigerGraph Cloud limits how many instances and how many vCPUs a single account can provision. These limits can be waived by request if you have a valid credit card on your account. To waive a service limit, please contact sales@tigergraph.com ↗.

The table below outlines the service limits for accounts with and without a valid credit card.

Service	**Accounts with a valid credit card**	**Accounts without a valid credit card**
Free Tier Solution	1	1
Total Instance	20	2
Total vCPU	160	16
Cluster Size (replication factor * partition factor)	10 instances	1
Replication Factor	2	1
Backup and Restore	Yes	No
GSQL Web Shell Access	Yes	No

Free tier solution limits

Each registered account is allowed one solution from the Free Tier. Free Tier instances do not include backup and do not include support.

If no user activity is detected for more than one hour, TigerGraph may automatically stop a Free Tier instance. Users can manually restart the free tier instances from their cloud portal. After 7 days of inactivity, TigerGraph may terminate the inactive free tier instances.

Glossary

backup

A snapshot of the state of your solutions at a given point in time.

instance

A virtual server in the cloud.

partitioning factor

An internal TigerGraph characteristic of a TigerGraph cluster that determines how the data in the database will be distributed

replication factor

A characteristic of the cluster design that will determine the number of copies of data that will be stored in the cluster.

role-based access control

A policy-neutral access-control mechanism defined around roles and privileges.

solution

Fully-managed TigerGraph deployment.

TigerGraph user

A database user with its own roles and privileges for different graphs stored on a solution.

Platform Overview

Continuous Availability Overview

Introduction:

HA (High Availability) is a generic term used to describe a computer system which has been architected to deliver higher levels of operational performance through enhanced uptime and throughput than would be expected from a traditional single server node.

With Continuous Availability, TigerGraph goes beyond the standard scope and definition of High Availability by providing the following functionality:

- Fault tolerance against loss of database server(s)
  - Automated recovery of services for Intra-cluster failure
- Full native HA support for user-facing applications - Seamless automatic client reconnection to standby GSQL server and GraphStudio servers
- Failover to remote cluster during Disaster Recovery
- Improve RoI with additional Replicas
  - Enhanced Query Throughput performance
  - Increased concurrency for operational workloads

In short, TigerGraph Continuous Availability provides not only the ability to keep the business application running without any noticeable downtime, but also delivers higher return on investment.

Architecture Design:

TigerGraph’s architecture design relies on active-active replication to keep multiple data copies in sync. This is transparent to the user. The underlying principles of uniform distribution of data are automatically applied no matter how many replicas are stored. Additionally, the placement of replicas is infrastructure-aware to tolerate hardware failures. Continuous availability is a production configuration that customers can pick at cluster installation time. Customers have flexibility to place replicas in specific availability zones or data centers based on their infrastructure requirements.

TigerGraph Continuous Availability design provides the following

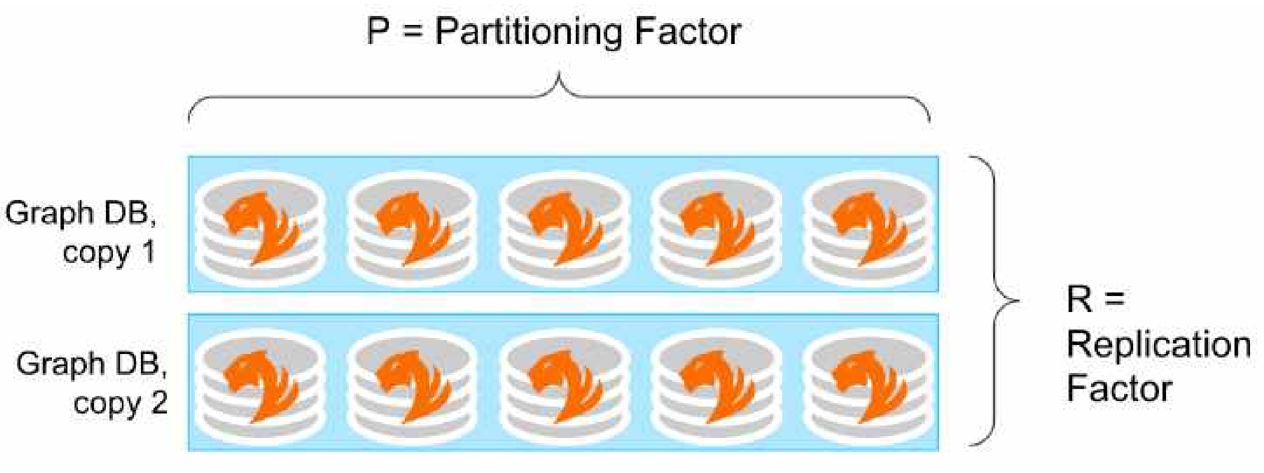
- Throughput: Each replica is always up to date and handles its share of read requests. This provides higher query concurrency and throughput.
- If a server goes offline for planned or unplanned reasons, TigerGraph's HA design with Automatic Failover will reroute work to the replica nodes of that server, maintaining continuous operation.
- Higher levels of replication provide more throughput and resilience.

Continuous Availability - Definitions:

TigerGraph is based on an MPP architecture. All services are distributed uniformly across the cluster. This requires data to be distributed across the cluster. There are two key concepts in the cluster design:

Replication Factor: Replication factor is a characteristic of the cluster design that will determine the number of copies of data that will be stored in the cluster. This is configurable and customers can choose at the time of installation.

Partitioning Factor: Partitioning factor is an internal TigerGraph characteristic of a TigerGraph cluster that determines how the data in the database will be distributed. Based on the nodes of the cluster size, TigerGraph will automatically pick a partitioning factor taking into account the replication factor.



Cluster dimention = P X R

In short, a TigerGraph cluster can be seen as a combination of data spread across P partitions with multiple copies equal to the number of R Replicas.

Some Key Cluster Design Considerations:

- Any cluster size is allowed, except 1×2
- Minimum number of servers needed for Continuous availability is 3 servers - This is due to Zookeeper quorum dependency.

Continuous Availability - Data Operations

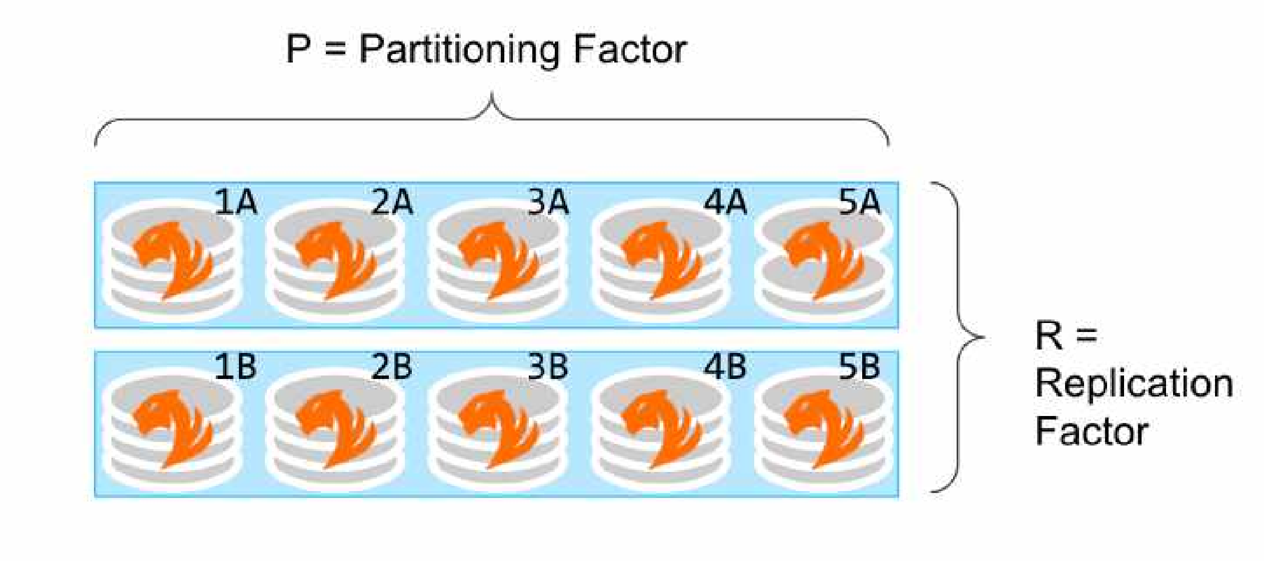
TigerGraph services are based on distributed masterless architecture - all replicas are equal and can service both read and write requests. This is a key differentiation that ensures that no single node can be the Single Point of Failure.

Write Operations: In order to keep all replicas in sync for full consistency of all data sets, all write operations are sent to all replicas synchronously by default. A write operation is considered complete only if all the replicas acknowledge that the writes are successful.

Read Operations: As all replicas are guaranteed to be in sync for all write operations, Read requests can be sent to any replica with no need to verify the data consistency with other replica copies. This optimizes the read performance for read-heavy analytical queries.

Example:

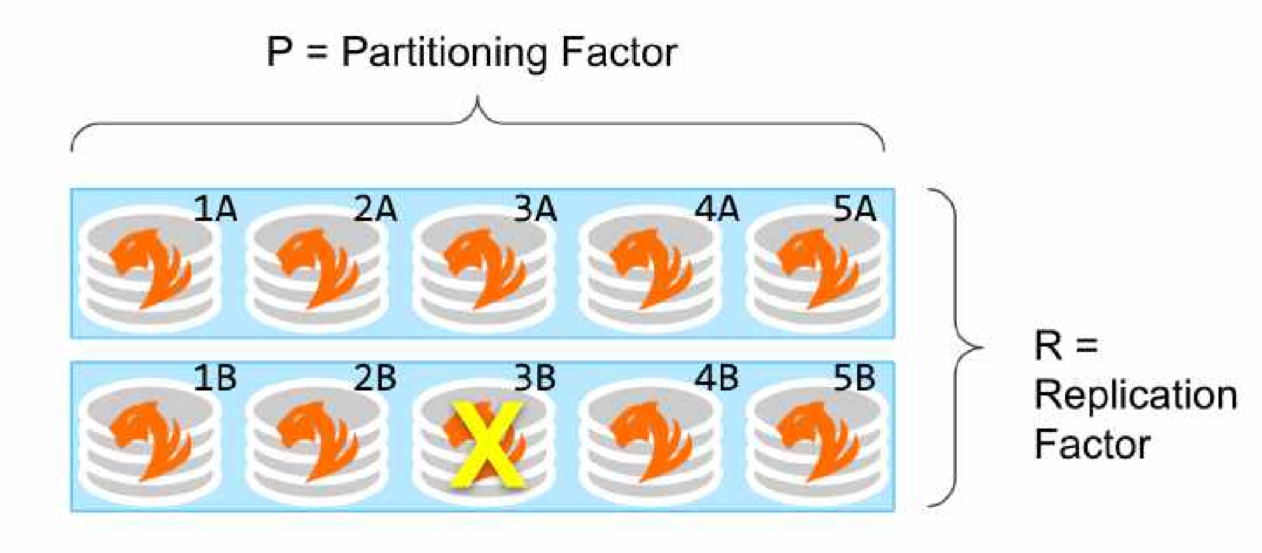
In the following example, the data sets in the cluster are spread across 5 partitions with 2 replicas of data sets i.e. Partitioning Factor of 5 and Replication Factor of 2.



- All writes go to all replicas (e.g. both 1A and 1B).
- Reads can be from any one replica (e.g. either 1A or 1B).
- Distributed queries can read from a mix replicas (e.g. {1A, 2B, 3B, 4A, 5B}.

Continuous Availability - Cluster failover operations:

TigerGraph design ensures Automatic Failover for Continuous Availability. If a server goes down (hardware or software, planned or unplanned), incoming DB operations can continue. The requests are automatically routed around the unavailable server. The TigerGraph Database scheduler tracks in real time the availability of servers and routes the request to the right servers.



Example:

In the event of server failure:

- If any single server is unavailable (expected or unexpected):
  - When it fails to respond after a certain number of tries, requests will automatically divert to another replica (e.g. 3B is unavailable, so use 3A)
  - If it fails in the middle of a transaction, that transaction might be aborted.
- System continues to operate, with reduced throughput, until the server is restored.

Comparing TigerGraph Editions

This document compares what is included in the Enterprise Edition vs. TigerGraph Cloud of the TigerGraph platform.

To see what has been added or changed in different releases (versions) of TigerGraph, see Change Log.

Overview

	TigerGraph Cloud	Enterprise Edition
Licensing	Pay as you go by the minute; Annual contracts available	A free license is available with up to 50 GB storage after compression - contact us ↗ for a paid version without storage limitations
Includes	The power of TigerGraph, as a Service + Instant Deployment + Automatic backups + Scaling out & Replication + Security + Pay for what you use	All TigerGraph Database and Enterprise features, including + Distributed Graph + MultiGraph + Security + User Management
Support	✅	Free version includes support from the Community Forum ↗. Contact us to get professional support.
How to Get It	www.tigergraph.com/cloud ↗	www.tigergraph.com/free-trial ↗

Database Features

Feature	TigerGraph Cloud	Enterprise Edition
Native MPP Graph	✅	✅
Real-Time Deep Link Analytics	✅	✅
Ultra-Fast Loading and Updates	✅	✅
GSQL Query and Loading Language	✅	✅
SQL-like syntax and built-in parallelism	✅	✅
Graph Size	Unlimited	Unlimited
Compressed Data Store	✅	✅
In-Memory Processing; ACID Transactions	✅	✅
Distributed, Auto-Partitioned Graph	✅	✅
MultiGraph	✅	✅
Dynamic Schema Change	✅	✅
GraphStudio Visual SDK and UI Design, Load, Explore, Query, Visualize	✅	✅
Admin Portal	✅	✅

Enterprise Features

Feature	TigerGraph Cloud	Enterprise Edition
Automated Deployment	✅	No
Automated Backups	✅	No

Backup and Restore	✅ built-in cloud feature	✅
Multiple Users	✅	✅
Continuous Availability	✅	✅
HA Replication	✅	✅
Cross-region replication	No	✅
Cluster resizing	No	✅

Security Features

Feature	TigerGraph Cloud	Enterprise Edition
Data Encryption At Rest and In Motion	✅	✅
Enterprise User Management LDAP and SSO	✅ through GSQL web shell	✅
Role-Based Access Control	✅	✅
User-defined roles	✅	✅
Audit compliance	✅ SOC 2 type 1 and type 2	Coming soon
Cloud Security: VPC for each account	✅	N/A

Graph Data Science Library

Please bookmark the new location:

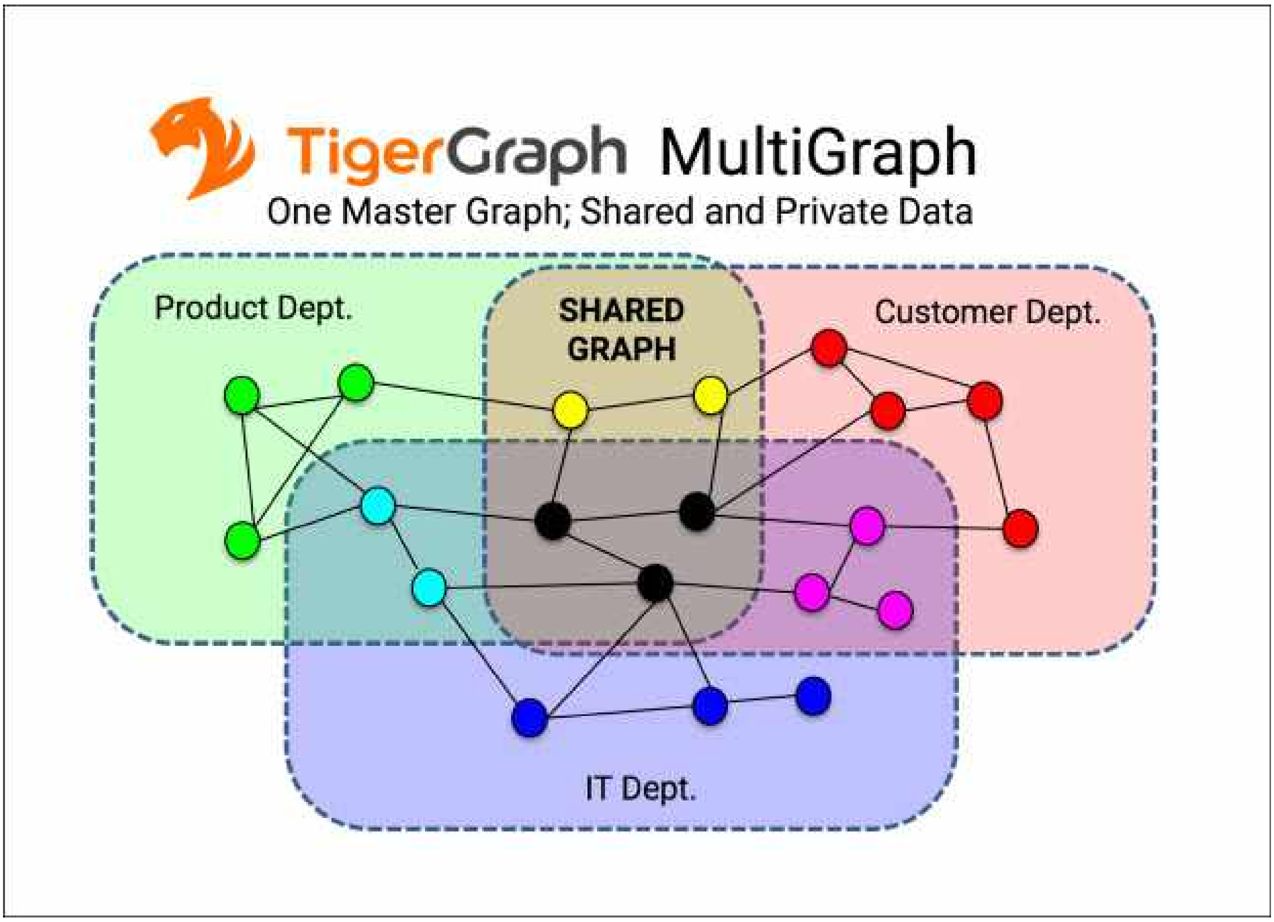
docs.tigergraph.com/graph-algorithm-library ↗

MultiGraph Overview

Beginning with Version 3.1,

- Vertex-Level Access Control **greatly improves the granularity of data access control from partitioning data at the type-level (MultiGraph) to the individual vertex-level (VLAC).**

One TigerGraph instance can manage multiple graphs, each with its own set of user privileges. This first-of-its-kind capability, dubbed MultiGraph, is available as an optional service in the TigerGraph platform.



MultiGraph allows one graph schema to be shared accross all departments, with different levels of access control at the type level.

MultiGraph enables several powerful use cases:

- Multiple Tenancy: Use one TigerGraph instance to support several completely separate data sets, each with its own set of users. Each user community cannot see the other user communities or other data sets.

- Fine-grained privileges on the same set of data: Role-based access control, available on single graphs, grants permission for the privilege to run queries (include data modification queries). In a single graph scheme, there is no way to say "Query X can be run by some users but not by others." Using multiple graphs defined over the same set of data, each graph can have its own set of queries and own set of users, in effect customizing who can run which queries.
- Overlapping graphs: Graphs can partially overlap, to enable a combination of shared and private data.
- Hierarchical subgraphs: A Graph X can be defined to cover the domains of Graphs Y and Z, that is, Graph X = (Graph Y) U (Graph Z). This provides an interesting way to describe a data partitioning or parent-child structure. (This is not the same as defining sub-classes of data types; data types are still independent.)

If you implement only one graph now, you can upgrade to MultiGraph and add additional graphs at any time, without having to redo your existing design.

Concepts

Graphs and Graph Domains

A graph is defined as a set of vertex types and edge types. More precisely, it is all the vertices and edges of that collection of types. The domain of a graph is its set of vertex types and edge types. Each graph contains its own data loading jobs and queries, which do not affect and are not visible to other graphs.

```
CREATE GRAPH <gname> (<list of vertex types & edge types>)
```

MultiGraph Principles

- A TigerGraph instance with a basic license key can have one graph. A TigerGraph instance with a MultiGraph license key can create multiple graphs.

  `superuser` and `globaldesigner` can define one or more graphs. The domains of the two graphs may be completely separate, may overlap, or may coincide exactly.

- A vertex type or edge type created by a superuser is a global type.
- A `superuser` or `globaldesigner` can include a global vertex or edge type in one or more graphs. Global types can be shared among multiple graphs.
- Users with the `admin` or `designer` role for a particular graph can add local vertex types and edge types to their own graph. Local types cannot be shared among multiple graphs.

Graph-Specific Roles and Privileges

The TigerGraph system includes several predefined roles. Each role is a fixed and logical set of privileges to perform operations. In order to access a graph, a user must be granted a role on that graph. Without a role, a user has no meaningful access.

Role-Based MultiGraph Access Control

- User roles are granted or revoked on a per-graph basis. Each GRANT or REVOKE statement specifies not only a role but also a graph.
- The `GRANT` / `REVOKE` privilege is reserved for superuser and admin users.
- The `superuser` can grant a role to any user on any graph.
- A `superuser` can pick a user to be an `admin` on a particular graph. The admin can then manage user privileges on their graph.
- A user may be granted different roles on different graphs.

For details about managing users, privileges, and roles, see User Privileges and Authentication. There you will find a chart describing each of the roles in detail.

Setting a Working Graph

- A user must set their working graph in order to access that graph, either using the -g flag with the GSQL command, or by using the `USE GRAPH` command.
- Users who have privileges on more than one graph (including superusers) may only work with one graph at a time. The `GLOBAL SCHEMA_CHANGE JOB` stretches

this rule.

Note that the `CREATE` commands for queries, loading jobs, and schema_change jobs require that the graph name be specified, even for systems with only one graph.

Effect on Other Specifications

1. RESTPP Endpoints: Endpoints that pertain to the graph data must include the name of the graph in the request URL.
   See RESTPP API User Guide .
2. User Authentication secrets and tokens: Our commands and procedures follow OAuth standards.
   See Managing User Privileges and Authentication.

There are many other details about using the MultiGraph feature, especially if your application has multiple users with different roles. In the documentation, the Multiple Graph logo is placed next to relevant topics:

TigerGraph Internal Architecture

As the world's first and only Native Parallel Graph (NPG) system, TigerGraph is a complete, distributed, graph analytics platform supporting web-scale data analytics in real time. The TigerGraph NPG is built around both local storage and computation, supports real-time graph updates, and works like a parallel computation engine. These capabilities provide the following unique advantages:

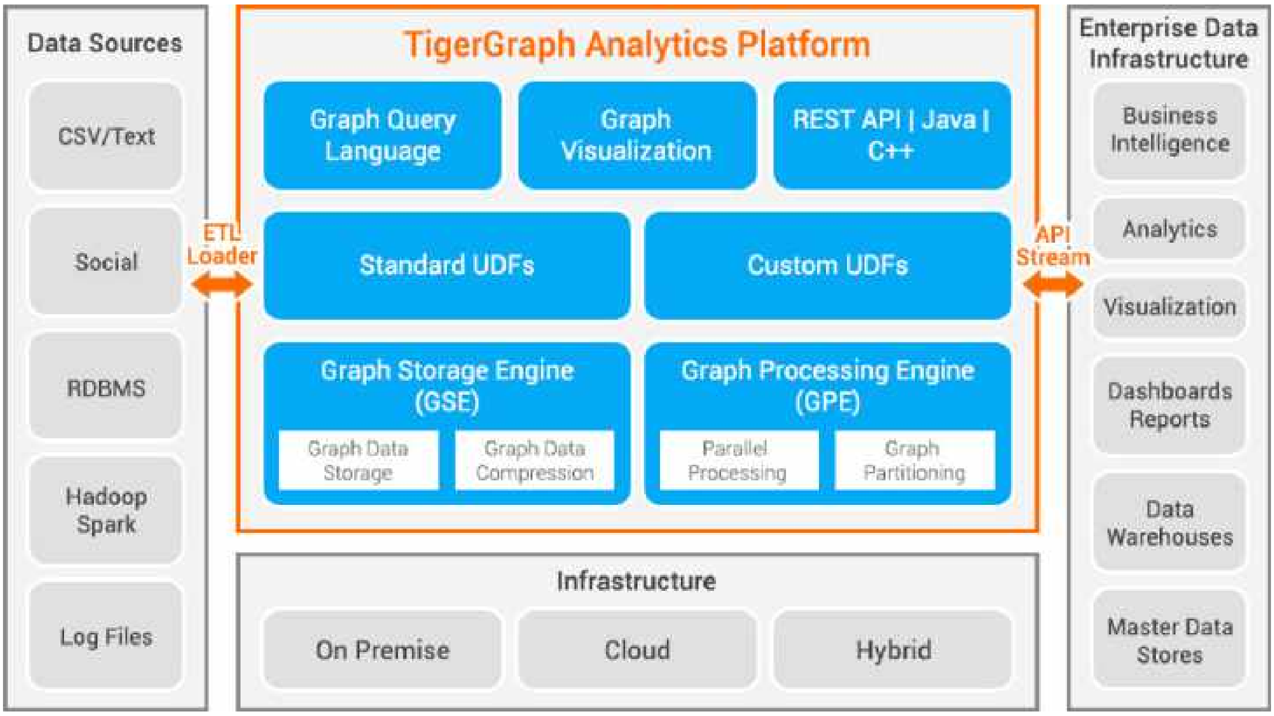
- Fast data loading speed to build graphs - able to load 50 to 150 GB of data per hour, per machine
- Fast execution of parallel graph algorithms - able to traverse hundreds of million of vertices/edges per second per machine
- Real-time updates and inserts using REST - able to stream 2B+ daily events in real-time to a graph with 100B+ vertices and 600B+ edges on a cluster of only 20 commodity machines
- Ability to unify real-time analytics with large scale offline data processing - the first and only such system

See the Resources ↗ section of our main website www.tigergraph.com ↗ to find white papers and other technical reports about the TigerGraph system.

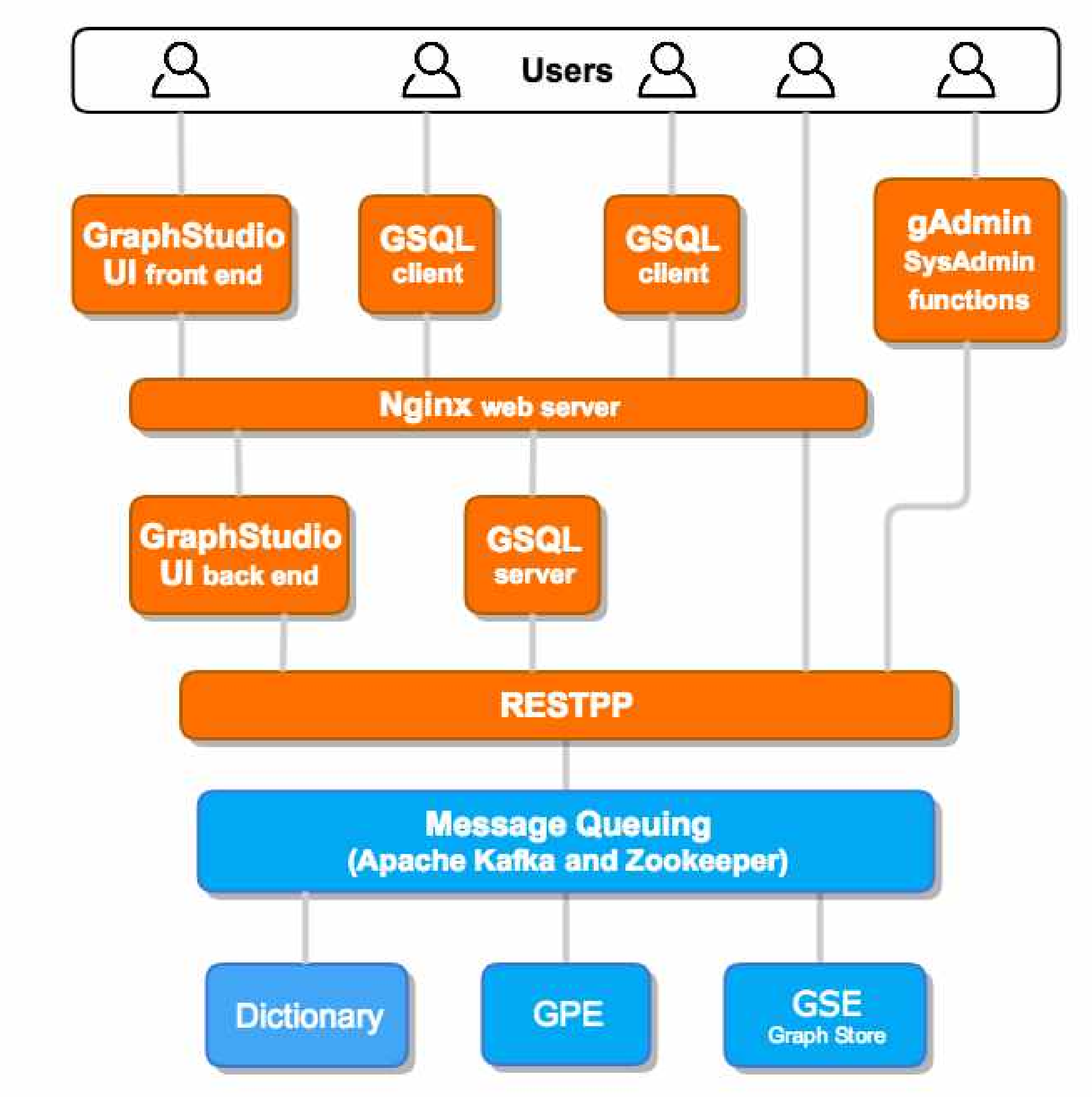
System Overview

The TigerGraph Platform runs on standard, commodity-grade Linux servers. The core components (GSE and GPE) are implemented in C++ for optimal performance. TigerGraph system is designed to fit into your existing environment with a minimum of fuss.

- Data Sources : The platform includes a flexible, high-performance data loader which can stream in tabular or semi-structured data, while the system is online.
- Infrastructure : The platform is available for on-premises, cloud, or hybrid use.
- Integration : REST APIs are provided to integrate your TigerGraph with your existing enterprise data infrastructure and workflow.



The figure below takes a closer look at the TigerGraph platform itself:



Within the TigerGraph system, a message-passing design is used to coordinate the activities of the components. RESTPP, an enhanced RESTful server, is central to the task management. Users can choose how they wish to interact with the system:

- GSQL client. One TigerGraph instance can support multiple GSQL clients, on remote nodes.
- GraphStudio - our graphical user interface, which provides most of the basic GSQL functionality, with a graphical and intuitive interface.
- REST API. Enterprise applications which need to run the same queries many times can maximize their efficiency by communicating directly with RESTPP.
- gAdmin is used for system adminstration.

Glossary

Name	Refers to
DDL	Data Definition Language - a generic term for a set of commands used to define a database schema. The GSQL Language includes DDL commands. In GraphStudio, the Design Schema function.
Dictionary (DICT)	The shared storage space for storing metadata about the graph store's configuration and state, including the catalog (graph schema, loading jobs, and queries).
DML	Data Manipulation Language - a generic term for a set of commands used to add, modify, and delete data from a database. Query commands are often considered a part of DML, even a pure query statement does not manipulate (change) the data. The GSQL Language includes full DML capability for query, add (insert), delete, and modify (update) commands.
gadmin	The system utility for configuring and managing the TigerGraph System. Analogous to mysqladmin.
gbar	Graph Backup and Restore. TigerGraph's utility program for backing up and restoring system data.
GPE	Graph Processing Engine. The server component which accepts requests from the REST++ server for querying and updating the graph store and which returns data.
Graph Store	The component which logically and physically stores the graph data and provides access to the data in a fast and memory-efficient way. We use the term

	graph store to distinguish it from conventional graph databases.
GraphStudio UI	The browser-based User Interface that enables the user to interact with the TigerGraph system in a visual and intuitive way, as an alternative to the GSQL Shell. The GraphStudio UI includes the following components: Schema Designer, Data Mapper, Data Loader, Graph Explorer, and Query Editor.
GSE	Graph Storage Engine. The processing component which manages the Graph Store.
GSQL	The user program which interprets and executes graph processing operations, including (a) schema definition, (b) data loading, and (c) data updates, and (d) data queries.
GSQL Language	The language used to instruct and communicate with the GSQL program.
GSQL Shell	The interactive command shell which may be used when running the GSQL program.
HA	High Availability - a generic term describing a computer system which has been architected to a higher level of operational performance (e.g., throughput and uptime) than would be expected from a traditional single server node.
IDS	ID Service. A subcomponent of the GSE which translates between user (external) IDs for data objects and graph store (internal) IDs.
Kafka	A free open-source "high-throughput distributed messaging system" from the Apache Software Foundation. Our distributed system architecture is based on message passing/queuing. Kafka is automatically included during TigerGraph system installation as one implementation

	of messaging passing. https://kafka.apache.org/ ↗
MultiGraph	A graph architecture and feature set which enables one global graph to be viewed as multiple logical subgraphs, each with its own set of user privileges. The subgraphs can overlap, meaning each subgraph can support both shared and private data.
Native Parallel Graph	An architecture and technology which provides for inherently highly-parallel and highly-scalable graph data storage and analytics. The use of vertex-level data+compute functionality is a key component of Native Parallel Graph design.
Nginx	A free, open-source, high-performance HTTP server and reverse proxy. Nginx is automatically included during TigerGraph system installation. https://nginx.org/en/ ↗
REST++ or **RESTPP**	A server component which accepts RESTful requests from clients, validates the requests, invokes the GPE, and sends responses back to the client. Additionally, REST++ provides a zero-coding interface for users to define RESTful endpoints.REST++ offers easy-to-use APIs for customizing the logic of handling requests and processing responses.
Single Sign-On (SSO)	A user authentication service that permits a user to use one set of login credentials to access multiple applications.
TigerGraph Platform	The TigerGraph real-time graph data analytics software system. The TigerGraph Platform offers complete functionality for creating and managing a graph database and for performing data queries and analyses. The platform includes the Graph Store and GSE , GPE, REST++, GSQL, GraphStudio, plus some third-party components, such as Apache Kafka and Zookeeper.

TigerGraph System	The TigerGraph platform and its languages. Based on context, the term may also include additional optional TigerGraph components which have been installed.
TS3	TigerGraph System Service State (TS3) is a TigerGraph sub-system which helps monitor the TigerGraph system. It serves as backend of TigerGraph Admin Portal.
Zookeeper	A free open-source program from the Apache Software Foundation, providing "a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services." Used for running the TigerGraph system on a cluster or other distributed system. Zookeeper is automatically included during TigerGraph system installation.

Transaction Processing and ACID Support

This document describes the transactional support provided by the TigerGraph platform. A TigerGraph ***transaction*** is a sequence of operations which acts as a single logical unit of work. A ***read-only operation*** in TigerGraph does not change any vertex/edge attribute value and doesn't insert any new or delete any existing vertex/edge. An ***update operation*** in TigerGraph is an operation which either changes some vertex/edge attribute value or insert some new or delete some existing vertex/edge.

The TigerGraph system provides full ACID transactions with sequential consistency. Transactions are defined as follows:

- Each GSQL query is a transaction. Each query may have multiple read or write operations.
- Each REST++ GET, POST, or DELETE operation (which may have multiple update operations within it) is a transaction.

Atomicity

A transaction with update operations may insert/delete multiple vertices/edges or update the attribute values of multiple edges/vertices. Such update requests are "all or nothing": either all changes are successful, or none is successful.

Consistency

The TigerGraph system provides traditional ACID consistency: A transaction can include data validation rules. The data validation rules can ensure any transaction will bring the system from one valid state to another.

The TigerGraph system also provides distributed system Sequential Consistency: every replica of the data performs the same operations in the same order. This is one of the strongest forms of consistency available, stronger than causal consistency, for example.

Isolation Level

TigerGraph supports the Serializable isolation level, the strongest form of isolation. Internally, TigerGraph uses MVCC to implement the isolation. MVCC, or Multi-Version Concurrency Control, makes use of multiple snapshots of portions of the database state in order to support isolated concurrent operations. In principle, there can be one snapshot per read or write operation.

No Dirty Reads

A read-only transaction R1 will not see any changes made by an uncommitted update transaction, whether that update transaction was submitted before or after R1 was submitted to the system.

Repeatable reads

Multiple same reads in a single transaction T1 will get the same results, even if there are update transactions which change vertex or edge attribute values read by T1 during T1's duration.

No phantom reads

Multiple reads in a single read-only transaction T1 will get the same results, even if there are update transactions which deleted/inserted vertices or edges read by T1 during T1's duration.

Durability

Committed transactions are written to disk (SSD or HDD). The TigerGraph platform implements write-ahead logging (WAL) to provide durability.

TigerGraph internal Snapshot Implementation

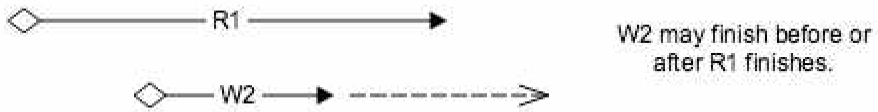
The TigerGraph platform uses Snapshot/MVCC (Multi-version Concurrency Control) to implement isolation of concurrent operations. At the high level, the platform can temporarily maintain multiple versions or snapshots of the graph data. When a transaction T1 is submitted to the system, it will work on the last consistent snapshot of the graph which has all the changes made by transactions committed before T1 was submitted but has no changes made by any transaction not yet committed when T1 was submitted. The version of the graph T1 is working on will not be changed by any transactions other than T1 , even if they commit before T1 is finished.

Example Scenarios

Let us examine a few transaction processing scenarios.

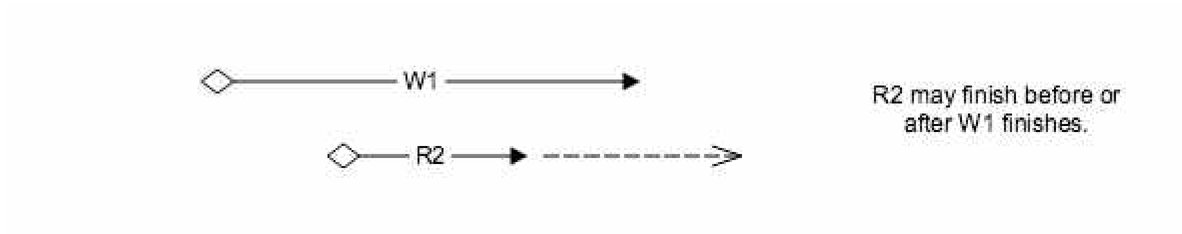
Scenario 1 Read - Write

A read-only transaction R1 is running. Before R1 finishes, an update transaction W2 comes in. W2 might finish before R1 is finished. But R1 will not see the changes made by W2 before W2 is committed (no dirty reads). Even if W2 is committed before R1 is finished, if R1 reads the same part of the graph multiple times, it will not see the changes made by W2 (repeatable reads). There are no phantom reads either. This is because the graph version R1 is working on cannot be changed by any of the W2 transaction aforementioned. Bottom line: If W2 starts when R1 is not yet committed, R1 will see results as though W2 did not exist.



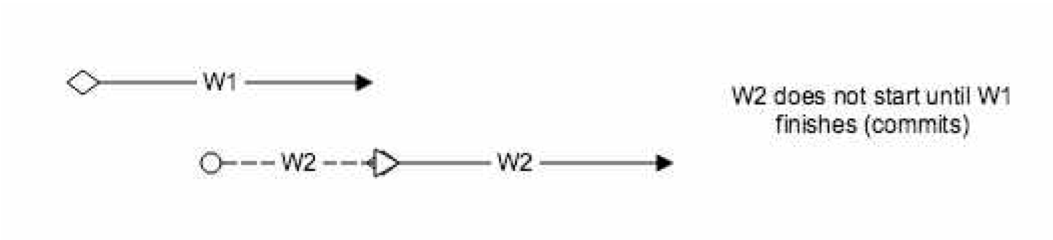
Scenario 2 Write - Read

An update transaction W1 is running. Before W1 is committed, a read-only transaction R2 comes in. R2 will not wait for W1 to finish and will be executed as if there is no W1. Later. even if W1 finishes and commits before R2 is finished, R2 will not see any changes made by W1. This is because the graph version R2 works on is 'fixed' at the time when R2 is submitted and will not include the changes to be made by W1. Bottom line: If R2 starts when W1 is not yet committed, R2 will see results as though W1 did not exist.



Scenario 3 Write - Write

An update transaction W1 is running. Before W1 finishes, a new update request W2 comes in. W2 will wait for W1 to finish before it is executed. When multiple update transactions come in, they will be executed sequentially by the system according to the time they are received by the system.



Tutorials and Start Guides

This section provides tutorials and getting-started guides for users who want to get working right away.

GSQL

- GET STARTED Tutorial
- GSQL 101
- Accumulators Tutorial
- GSQL102 Pattern Matching

GraphStudio

- GraphStudio UI Guide

Getting Started

Welcome to the TigerGraph™ Platform - the first real-time, Native Parallel Graph data analytics platform. This document covers the various options to get started with TigerGraph.

- If you are a laptop user (macOS or Windows), we recommend you use Docker to start up TigerGraph on your computer:

Get Started with Docker >

- If you have a Linux machine that meets our Software and Hardware Requirements, you can install TigerGraph on your machine directly:

Installation Guide >

- If you would like to run TigerGraph on a Virtual Machine with Virtual Box, follow this guide:

Get Started with Virtual Box >

- You can also start up TigerGraph instances from Cloud Images on AWS, Microsoft Azure, or Google Cloud Platform:

Get Started from Cloud Images >

Get Started with Docker

This document provides step-by-step instructions on how to pull the latest TigerGraph Enterprise Edition docker image to your host machine. You can follow the sections in sequence to set up the TigerGraph docker environment.

The latest TigerGraph docker image includes the following content:

- The latest version of TigerGraph
- Linux packages:
  - openssh-server
  - git
  - wget
  - curl
  - emac
  - vim
  - jq
  - tar
- Tutorial material
  - GSQL 101
  - GSQL 102
- The latest GSQL open-source graph algorithm library

This follow-along video shows the whole setup process:
https://www.youtube.com/watch?v=V5VvgJyjLxA ↗

1. Install Docker Desktop

Follow the steps below to install Docker Desktop on your machine and configure it with sufficient resources for TigerGraph:

1. Install Docker on your OS:

- To install Docker for Mac OS, follow this video: https://www.youtube.com/watch?v=MU8HUVlJTEY ↗
- To install Docker for Linux, follow these instructions:
  - Centos: https://docs.docker.com/install/linux/docker-ce/centos/ ↗
  - Ubuntu: https://docs.docker.com/install/linux/docker-ce/ubuntu/ ↗
- To install Docker for Windows OS, follow this video: https://www.youtube.com/watch?v=ymlWt1MqURY ↗

2. Configure Docker Desktop with sufficient resources:
   - Recommended: 4 cores and 16GB memory
   - Minimum: 2 cores and 10GB memory
   - Click the Docker Desktop icon, click **Preferences** >> **Resources**, drag the CPU and Memory sliders to the desired configuration, save and restart Docker Desktop

2. Prepare a Shared Folder on Host OS to be shared with Docker Container

Open a shell on your host machine and create or select a directory for sharing data between your host machine and docker container. Grant read+write+execute permission to the folder. For example, to create a folder called data in Linux:

```
mkdir data
chmod 777 data
```

You can mount (map) the data folder to a folder under the docker container, which allows you to share files between your host OS and Docker OS.

For example, if you mount the host OS folder `~/data` to the docker folder `/home/tigergraph/mydata`, then anything you put on `~/data` will be visible in the docker container under `/home/tigergraph/mydata`, and vice versa.

3. Run TigerGraph Docker image as a daemon

Run the following command to pull the TigerGraph docker image, bind ports, map a shared data folder, and start a container from the image. This command is very long; click the copy button at the end of the code block to copy the whole command:

```
docker run -d -p 14022:22 -p 9000:9000 -p 14240:14240 --name tigergraph -
```

- Here is a breakdown of the options and arguments in the command:
  - `-d`: make the container run in the background.
  - `-p`: map docker 22 port to host OS 14022 port, 9000 port to host OS 9000 port, 14240 port to host OS 14240 port.
  - `--name`: name the container tigergraph.
  - `--ulimit`: set the `ulimit` (the number of open file descriptors per process) to 1 million.
  - `-v`: mount the host OS `~/data` folder to the docker `/home/tigergraph/mydata` folder using the -v option. If you are using Windows, change the above ~/data to something using windows file system convention, e.g. `c:\data`
  - `-t`: allocate a pseudo-TTY
  - `docker.tigergraph.com/tigergraph:latest`: download the latest docker image from the TigerGraph docker registry URL docker.tigergraph.com/tigergraph.

> Replace "latest" with a specific version number if a dedicated version of TigerGraph is to be used. E.g., if you want to get the 3.0.5 version, the URL should be:
>
> `docker.tigergraph.com/tigergraph:3.0.5`
>
> To use the legacy developer editions, use:
>
> `docker.tigergraph.com/tigergraph-dev`

If you use Windows and have disk drive permission issues with the above command, try the following command instead (this command does not map the shared folder on your host machine to your container) :

```
docker run -d -p 14022:22 -p 9000:9000 -p 14240:14240 --name tigergraph -
```

4. Use SSH to connect to your container

After launching the container, you can use SSH to connect to your container:

1. Verify that the container is running. You should see a row that describes the running container after running the command below:

   ```
   docker ps | grep tigergraph
   ```

2. Use ssh to open a shell to the container. At the prompt, enter `tigergraph` as the password. Note that we have mapped the host 14022 port to the container's 22 port (the ssh default port), so on the host we use ssh to connect to port 14022.

   ```
   ssh -p 14022 tigergraph@localhost
   ```

5. Start TigerGraph

1. After connecting to the container via ssh, inside the container, start all TigerGraph services with the following command (which may take up to one minute):

   ```
   gadmin start all
   ```

2. Run the `gsql` command as shown below to start the GSQL shell. If you are new to TigerGraph, you can run the GSQL 101 tutorial now.

   ```
   $ gsql
   GSQL >
   ```

3. Start GraphStudio, TigerGraph's visual IDE, by visiting `http://localhost:14240`

in a browser on your laptop (host OS).

Operation Commands Cheat Sheet

- After you start Docker Desktop, use the commands below to stop and restart the container:

```
docker container stop tigergraph
docker container start tigergraph
```

- Start the TigerGraph service within the container:

```
gadmin start all
gadmin stop  all
```

- ssh to the container. Note: if localhost is not recognized, remove the localhost entry from ~/.ssh/known_hosts

```
sed -i.bak '/localhost/d' ~/.ssh/known_hosts
ssh -p 14022 tigergraph@localhost
```

> Linux users can access the container through its ip address directly:

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}
vssh tigergraph@<container_ip_address>
```

- Default user: `tigergraph`
- Default password: `tigergraph`
- After running `gadmin start`, you can go to GraphStudio. Open a browser on your laptop (host OS) and access GraphStudio at the following URL:

```
http://localhost:14240
```

- Check the version of GSQL:

```
gsql version
```

Get Started with Virtual Box

TigerGraph offers Virtual Machine images of Linux servers with the TigerGraph Enterprise Free Edition installed for users who want to run TigerGraph with Virtual Box.

Watch this tutorial video to get started with TigerGraph on Virtual Box.

Install TigerGraph on Linux

You can install TigerGraph on a Linux machine that meets the Hardware and Software Requirements. For a step-by-step guide on installing TigerGraph on your Linux machine, please visit the following page:

Installation Guide

Quickstart guide for new users

Installation Checklist

1. **CHECK** Hardware and Software Requirements
2. **DOWNLOAD** the TigerGraph platform: https://info.tigergraph.com/enterprise-free
3. **INSTALL** the Platform
   a. For simple single-server installation:
   Assuming your downloaded file is called <your_tigergraph_package>:

```
tar xzf <your_tigergraph_package>.tar.gz
cd tigergraph*/

# to install license in interactive mode
sudo ./install.sh

# to install license in non-interactive mode
sudo ./install.sh -n
```

   b. For additional options, see TigerGraph Platform Installation Guide

Get Started from Cloud Images

This chapter covers different options for getting started with TigerGraph in the cloud. To continue, choose the cloud platform you wish to get started on:

Get Started on AWS

Get Started on Microsoft Azure

Get Started on Google Cloud

For all editions on cloud marketplaces, please contact TigerGraph support and we'll assist you in upgrading from older images to the latest image.

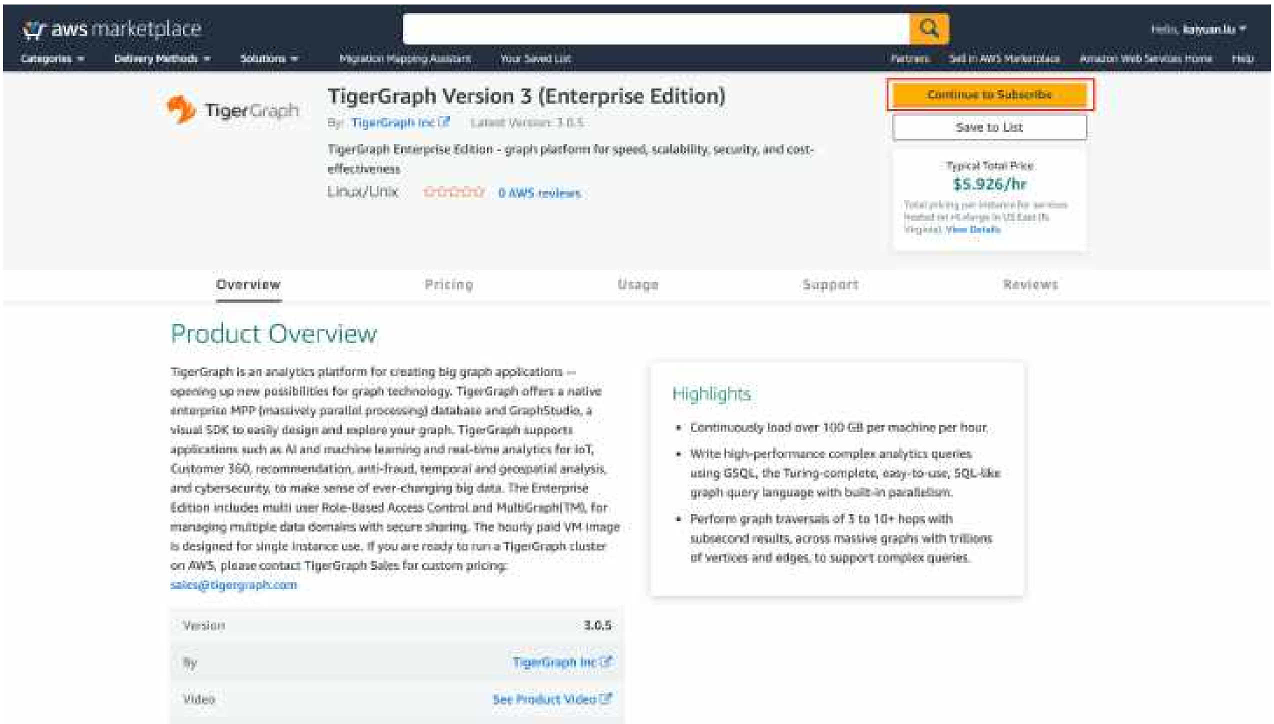
Get Started on AWS

This tutorial will show you how to start TigerGraph Enterprise Edition from an image on AWS.

Deploying Your Instance

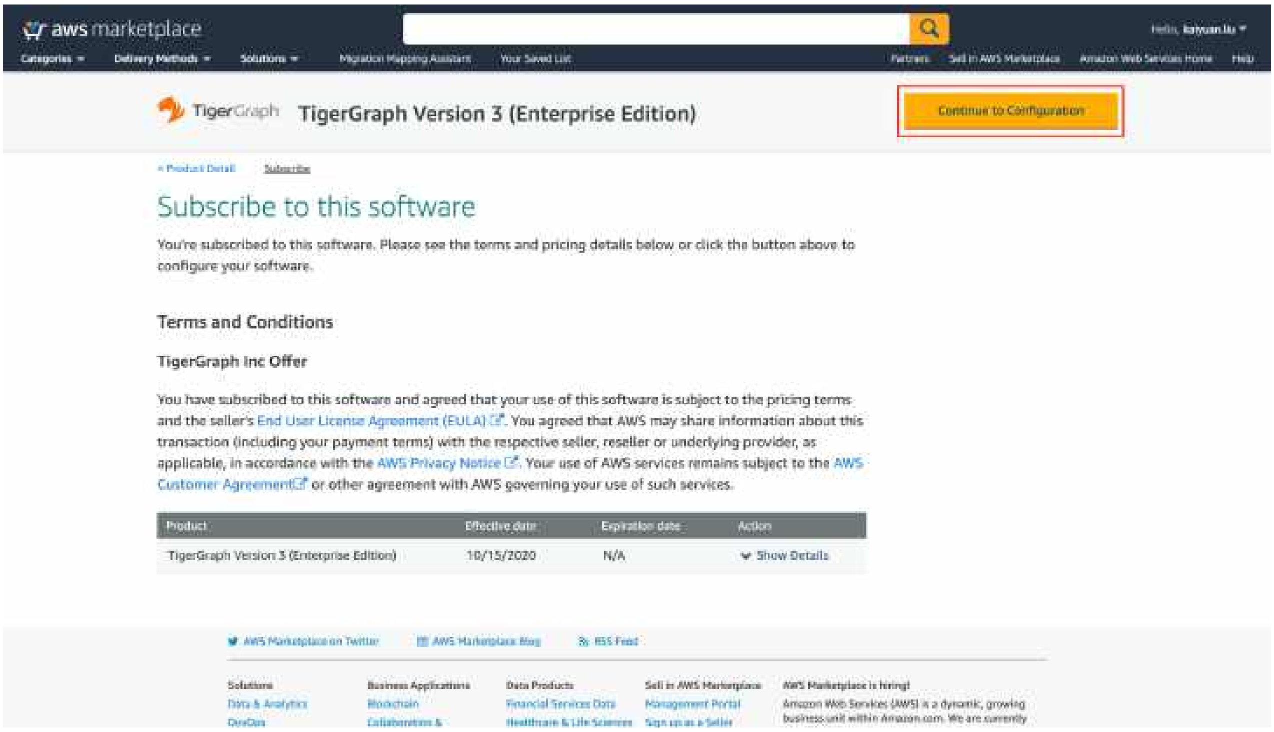
1. Go to [AWS Marketplace ↗](AWS Marketplace) and search for TigerGraph.

2. Click "Continue to Subscribe".



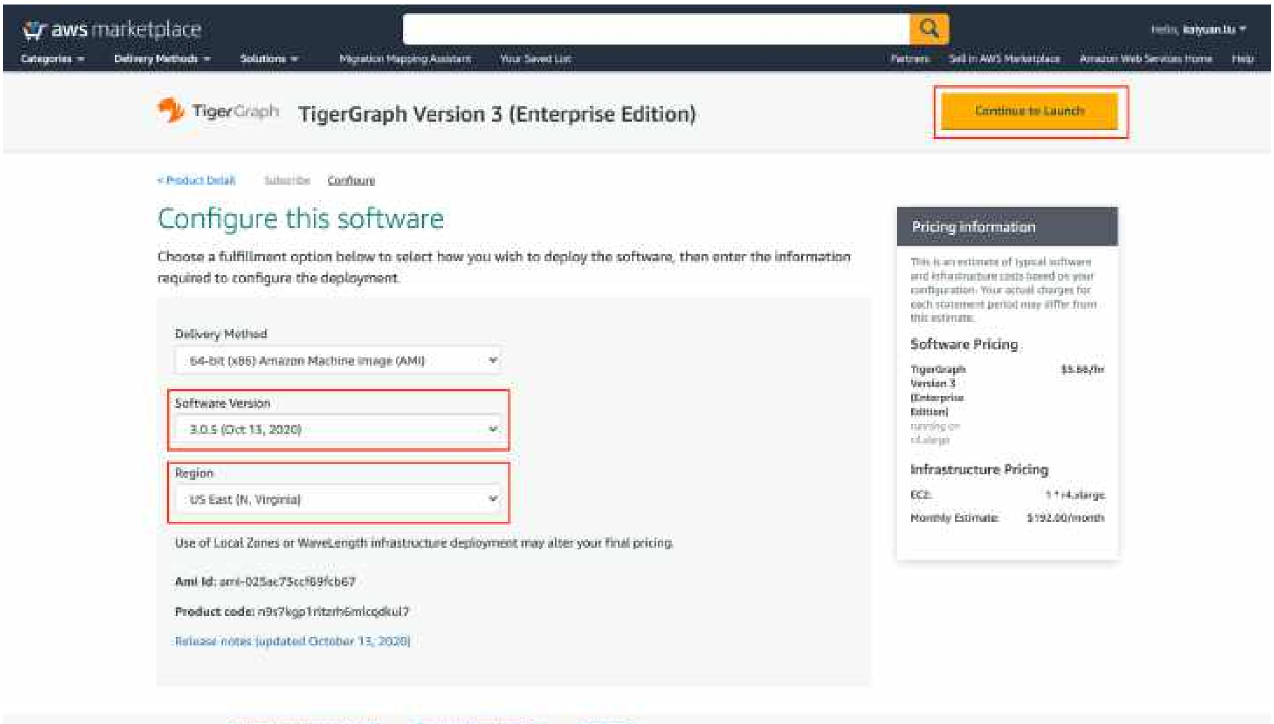
Subscribe Page

3. Click on "Continue to Configuration".



Pre Configuration Page

4. Select the Software Version and Region. We recommend selecting the latest version for the most up-to-date features. After making your selections, click on "Continue to Launch".



Configuration Page

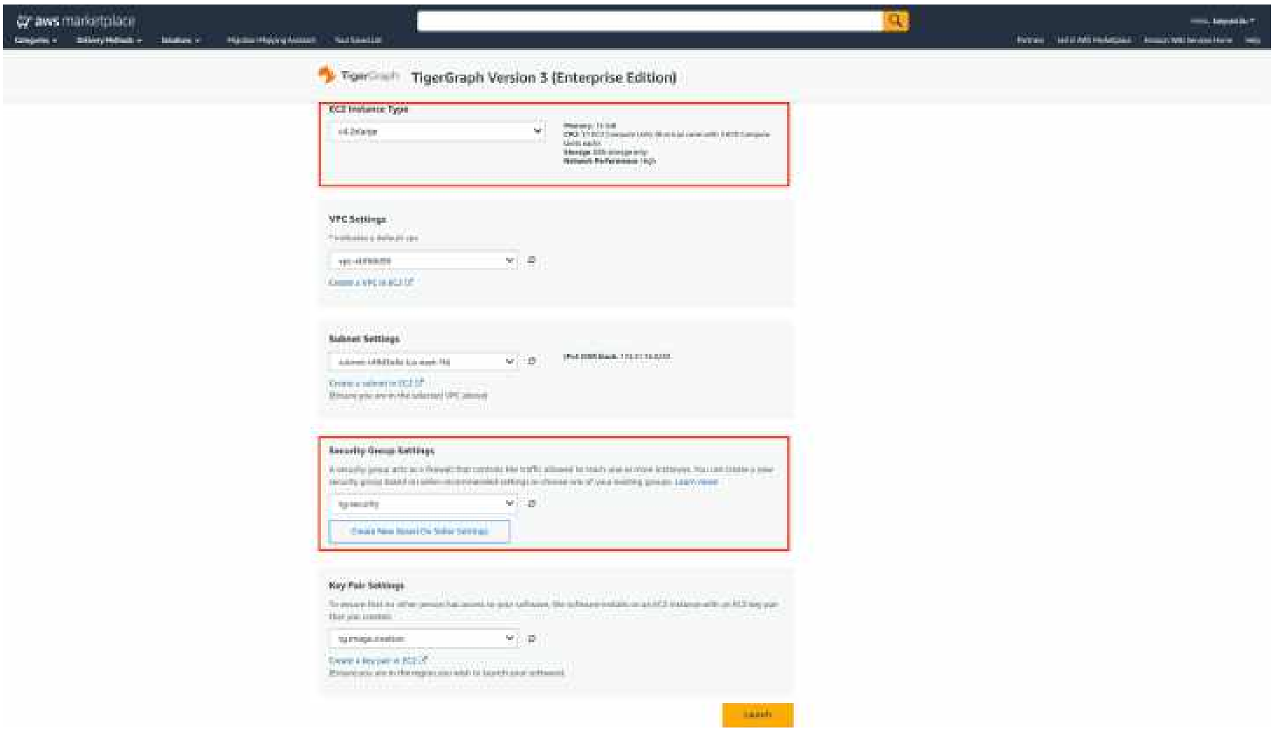
5. Select the instance type, security group settings, and other settings. The default settings are fine for most users, but feel free to modify them. Click "Launch" when finished.

Notes:
The instance type needs to have at least 4 CPUs and 16GB RAM for TigerGraph to work properly.

The security group must allow inbound TCP traffic to port 14240 if you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the GraphStudio UI Guide.

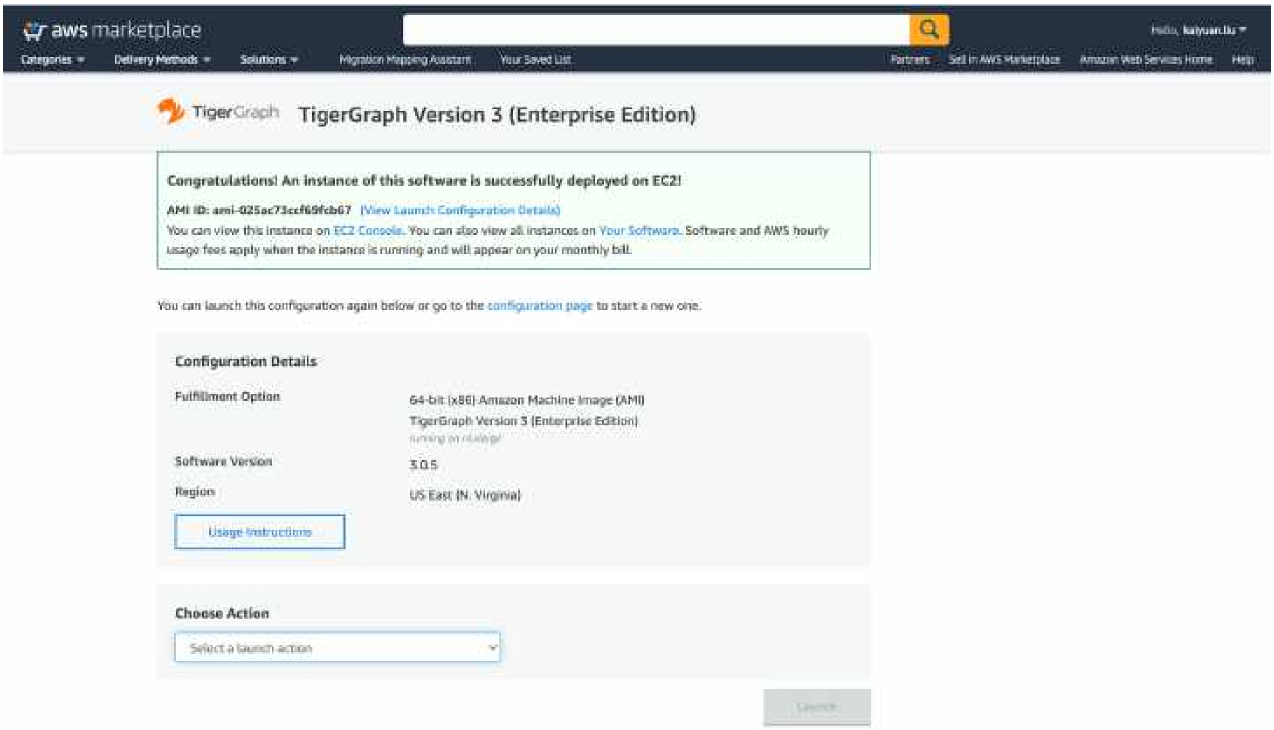
The security group must allow inbound TCP traffic to port 9000 if you want to send RESTful requests to TigerGraph from outside the instance (this includes configuring the GSQL client on a remote machine). For more about the REST API, see the TigerGraph RESTful API User Guide.

For more about the TigerGraph Platform, see the TigerGraph Platform Overview.



Configuration Page 2

6. That's it! The TigerGraph instance has been successfully deployed on AWS.

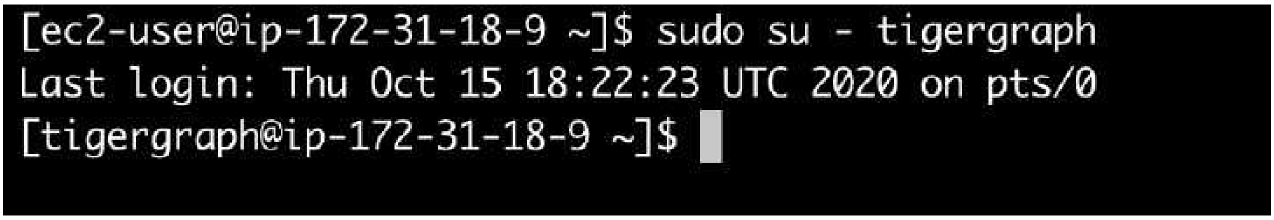


Deploying Page

Starting TigerGraph on Your Instance

1. Log on to the instance and switch to user `tigergraph` using the following command:

```
sudo su - tigergraph
```



TigerGraph Login

2. Run the following command to check the current status of TigerGraph. The services **"ADMIN", "CTRL", "ETCD", "IFM", "KAFKA", and "ZK" are started automatically** and should be up at this point. If any of them are not or you get the following error message, **please wait for 30 to 60 seconds and check the status again** before reporting it to TigerGraph support.

```
gadmin status
```

```
[tigergraph@ip-172-31-18-9 ~]$ gadmin status
+--------------------+-------------------------+-------------------------+
|    Service Name    |     Service Status      |      Process State      |
+--------------------+-------------------------+-------------------------+
|       ADMIN        |         Online          |         Running         |
|        CTRL        |         Online          |         Running         |
|        DICT        |          Down           |         Stopped         |
|        ETCD        |         Online          |         Running         |
|        GPE         |          Down           |         Stopped         |
|        GSE         |          Down           |         Stopped         |
|        GSQL        |          Down           |         Stopped         |
|        GUI         |          Down           |         Stopped         |
|        IFM         |         Online          |         Running         |
|       KAFKA        |         Online          |         Running         |
|     KAFKACONN      |          Down           |         Stopped         |
|    KAFKASTRM-LL    |          Down           |         Stopped         |
|       NGINX        |          Down           |         Stopped         |
|       RESTPP       |          Down           |         Stopped         |
|        TS3         |          Down           |         Stopped         |
|      TS3SERV       |          Down           |         Stopped         |
|         ZK         |         Online          |         Running         |
+--------------------+-------------------------+-------------------------+
[tigergraph@ip-172-31-18-9 ~]$
```

Output of gadmin status

```
[tigergraph@ip-172-31-18-9 ~]$ gadmin status
[  Error] ConnectionClosed (Unable to talk to ifm, are all infrastructure services up? You can start them by 'gadmin start infra')
[tigergraph@ip-172-31-18-9 ~]$
```

Gadmin status error message

3. Run the following command to start TigerGraph:

```
gadmin start
```

```
[tigergraph@ip-172-31-18-9 ~]$ gadmin start
[   Info] If no serviceID is specified, it only starts services excluding the infrastructure. Use 'gadmin start all' to start all services.
[   Info] Starting ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKASTRM-LL KAFKACONN TS3SERV GSQL TS3 GUI
[tigergraph@ip-172-31-18-9 ~]$
```

Output of gadmin start

4. Check the status again. All services should be up at this point:

```
gadmin status
```

```
[tigergraph@ip-172-31-18-9 ~]$ gadmin status
+--------------------+------------------------+------------------------+
|    Service Name    |     Service Status     |     Process State      |
+--------------------+------------------------+------------------------+
|       ADMIN        |         Online         |        Running         |
|        CTRL        |         Online         |        Running         |
|        DICT        |         Online         |        Running         |
|        ETCD        |         Online         |        Running         |
|        GPE         |         Warmup         |        Running         |
|        GSE         |         Warmup         |        Running         |
|        GSQL        |         Online         |        Running         |
|        GUI         |         Online         |        Running         |
|        IFM         |         Online         |        Running         |
|       KAFKA        |         Online         |        Running         |
|     KAFKACONN      |         Online         |        Running         |
|    KAFKASTRM-LL    |         Online         |        Running         |
|       NGINX        |         Online         |        Running         |
|       RESTPP       |         Online         |        Running         |
|        TS3         |         Online         |        Running         |
|      TS3SERV       |         Online         |        Running         |
|         ZK         |         Online         |        Running         |
+--------------------+------------------------+------------------------+
[tigergraph@ip-172-31-18-9 ~]$
```

Gadmin status after running gadmin start

5. TigerGraph has been successfully started on your cloud instance.

TigerGraph License on AWS Images

The TigerGraph Enterprise edition image comes with **a perpetual license** that will **only work on the AWS instance it's installed on**. Please run the following command to see it:

```
gadmin license status
```

```
[tigergraph@ip-172-31-18-9 ~]$ gadmin license status
[Warning] License will expire in 106751 days

   Issuer: TigerGraph Cloud Marketplace Keygen
 Audience: Marketplace User
IssueTime: 2020-10-15 18:13:35 +0000 UTC
  EndTime: 3344-05-20 13:14:05 +0000 UTC
  Edition: Cloud Marketplace

Host:
        MaxCPUCore: 9.007199254740991e+15
        MaxPhysicalMemorySize: 8589934592.00MB
        MaxClusterNodeNumber: 3.34452e+06

Topology:
        MaxVertexNumber: 9.007199254740991e+15
        MaxEdgeNumber: 9.007199254740991e+15
        MaxGraphNumber: 5.201314e+06
        MaxTopologySize: 8589934592.00MB

RuntimeMemory:
        MaxUserResidentSetSize: N/A
[tigergraph@ip-172-31-18-9 ~]$
```

Gadmin license status output

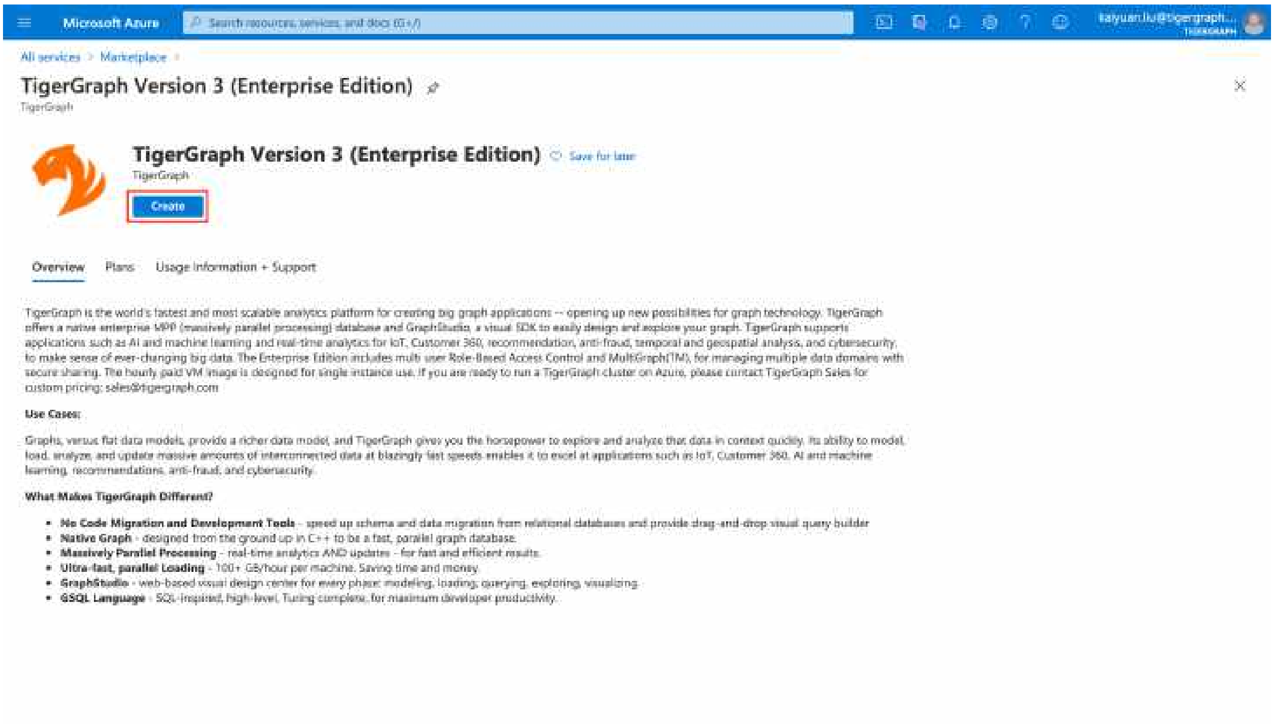
Get Started on Microsoft Azure

This tutorial will show you how to start TigerGraph Enterprise Edition from an image on Microsoft Azure.

Deploying Your Instance

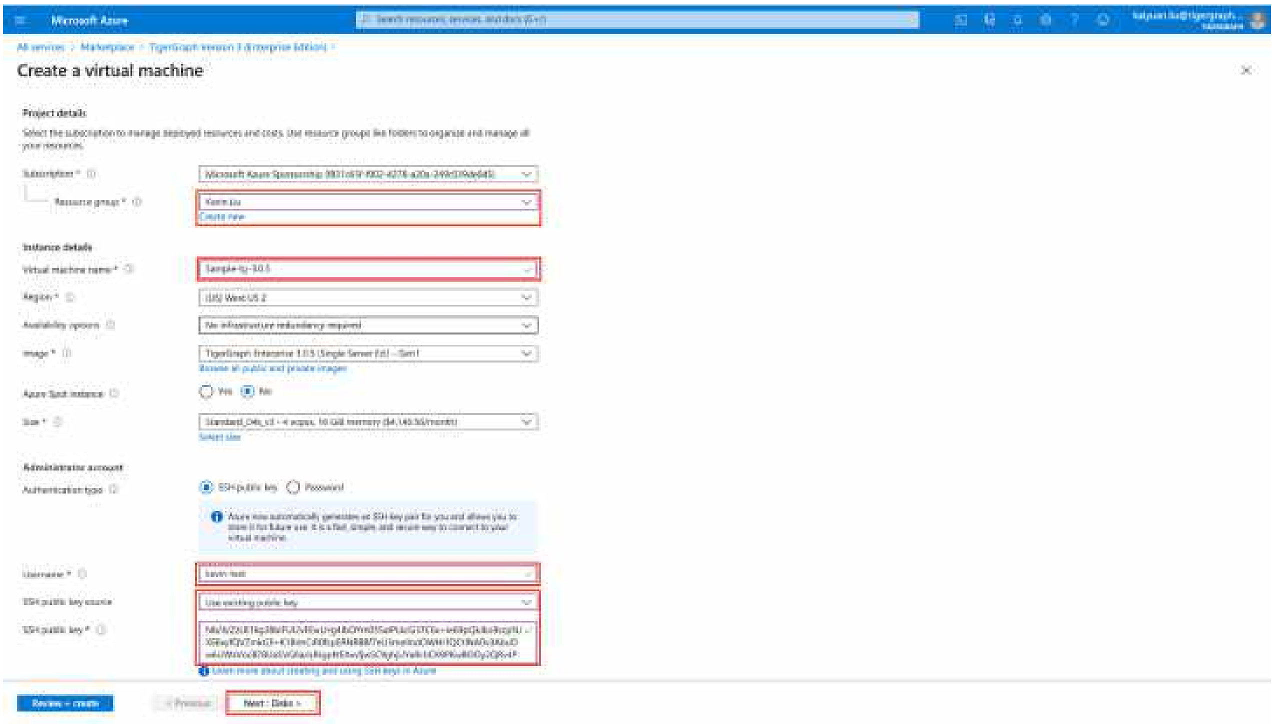
1. Go to Microsoft Azure Marketplace ↗ and search for "TigerGraph".

2. Select your software plan and Click "Create". Select the latest plan (with the highest version number) to access the latest features.



Create Page

3. Fill out the "Resource group", "Virtual machine name", "Username" and "SSH Public key" fields. The default values should work for the rest of the fields. Then click "**Next: Disks >**".



Basic Settings Page

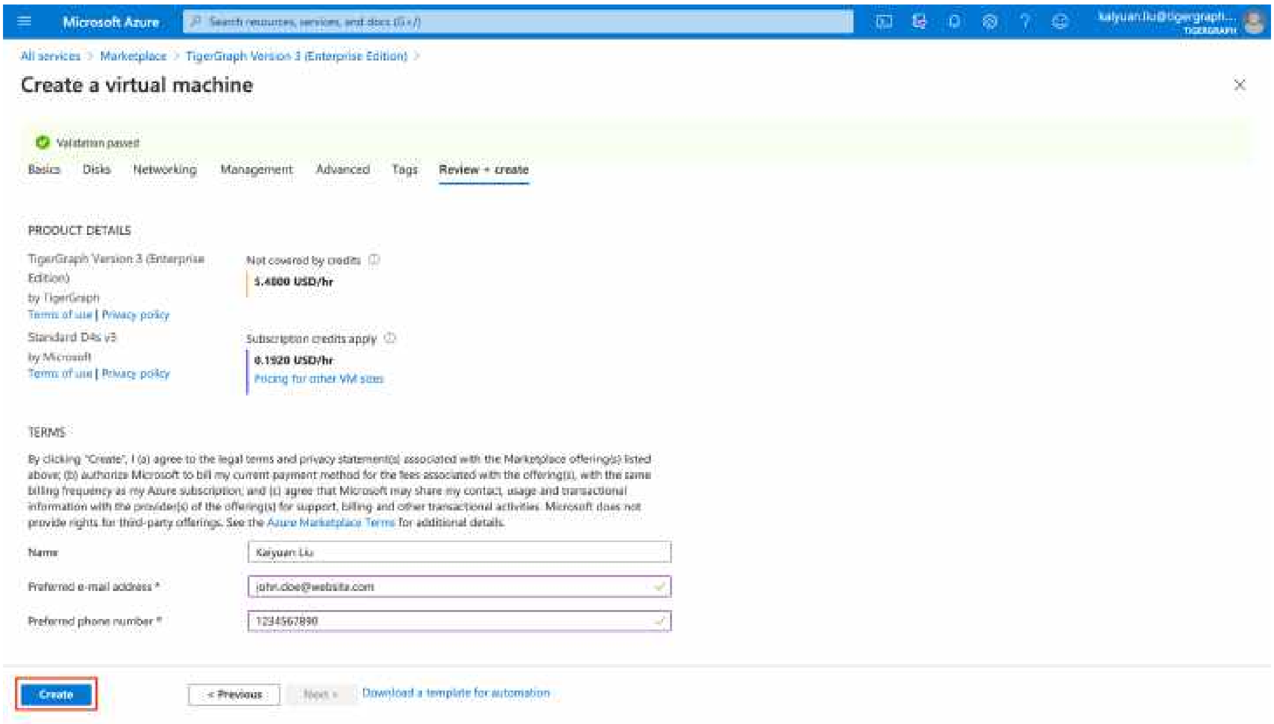
4. Keep the default values for all other settings and click "Next" until you see the "Review + Create" page below. Check all your settings and click "Create" when you are satisfied.

Notes:
The instance type needs to have at least 4 CPUs and 16GB RAM for TigerGraph to work properly.

The "NIC network security group" must allow inbound TCP traffic to port 14240 if you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the [GraphStudio UI Guide](#).

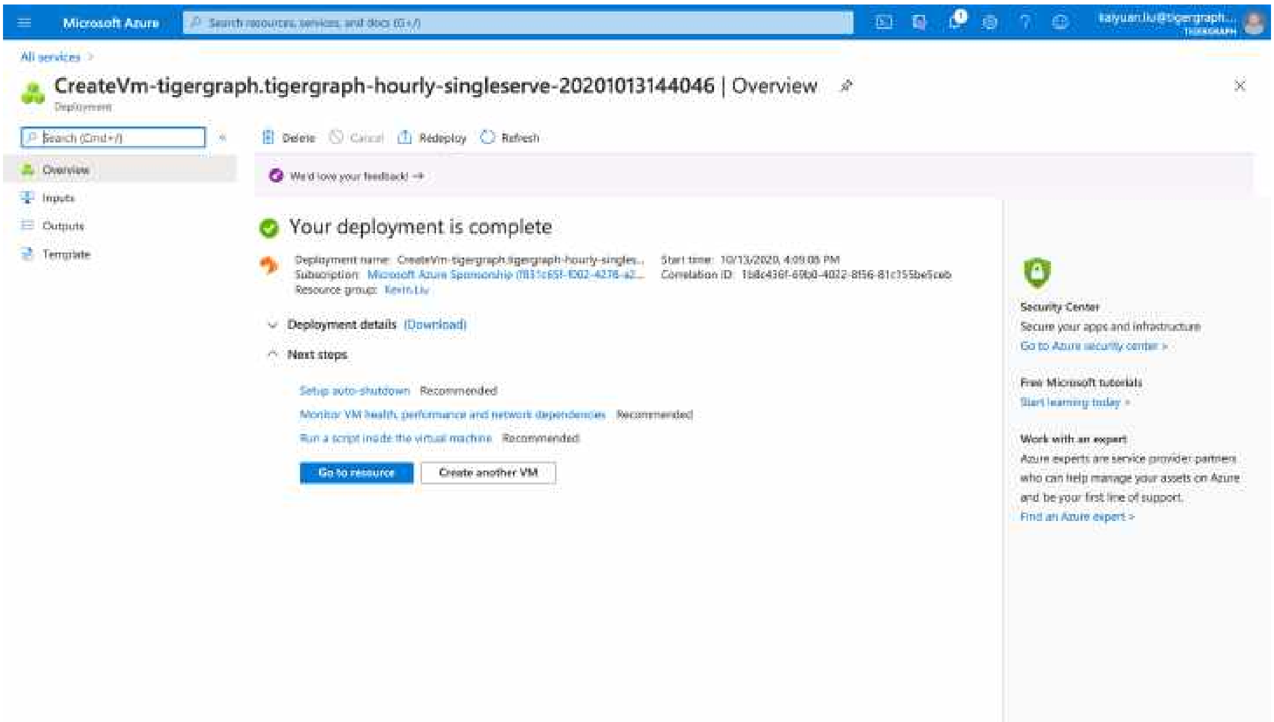
The "NIC network security group" must allow inbound TCP traffic to port 9000 if you want to send RESTful requests to TigerGraph from outside the instance (this includes configuring the GSQL client on a remote machine). For more about the REST API, see the [TigerGraph RESTful API User Guide](#).

For more about the TigerGraph Platform, see the [TigerGraph Platform Overview](#).



Review Page

5. That's it! The TigerGraph instance has been successfully deployed on Microsoft Azure.

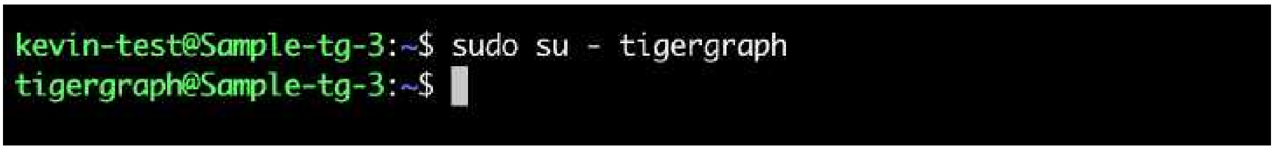


Deploying Page

Starting TigerGraph on Your Instance

1. Log on to the instance and switch to user `tigergraph` using the following command:

```
sudo su - tigergraph
```



TigerGraph Login

2. Run the following command to check the current status of TigerGraph. The services **"ADMIN", "CTRL", "ETCD", "IFM", "KAFKA", and "ZK" are started automatically** and should be up at this point. If any of them are not or you get the following error message, **please wait for 30 to 60 seconds and check the status again** before reporting it to TigerGraph support.

```
gadmin status
```

```
tigergraph@Sample-tg-3:~$ gadmin status
+--------------------+--------------------------+--------------------------+
|    Service Name    |      Service Status      |      Process State       |
+--------------------+--------------------------+--------------------------+
|       ADMIN        |          Online          |         Running          |
|        CTRL        |          Online          |         Running          |
|        DICT        |           Down           |         Stopped          |
|        ETCD        |          Online          |         Running          |
|        GPE         |           Down           |         Stopped          |
|        GSE         |           Down           |         Stopped          |
|        GSQL        |           Down           |         Stopped          |
|        GUI         |           Down           |         Stopped          |
|        IFM         |          Online          |         Running          |
|       KAFKA        |          Online          |         Running          |
|     KAFKACONN      |           Down           |         Stopped          |
|    KAFKASTRM-LL    |           Down           |         Stopped          |
|       NGINX        |           Down           |         Stopped          |
|       RESTPP       |           Down           |         Stopped          |
|        TS3         |           Down           |         Stopped          |
|      TS3SERV       |           Down           |         Stopped          |
|         ZK         |          Online          |         Running          |
+--------------------+--------------------------+--------------------------+
tigergraph@Sample-tg-3:~$
```

Output of gadmin status

```
tigergraph@Sample-tg-3:~$ gadmin status
[   Error] ConnectionClosed (Unable to talk to ifm, are all infrastructure services up? You can start them by 'gadmin start infra')
tigergraph@Sample-tg-3:~$
```

Gadmin status error message

3. Run the following command to start TigerGraph:

```
gadmin start
```

```
tigergraph@Sample-tg-3:~$ gadmin start
[    Info] If no serviceID is specified, it only starts services excluding the infrastructure. Use 'gadmin start all' to start all services.
[    Info] Starting ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKASTRM-LL KAFKACONN TS3SERV GSQL TS3 GUI
tigergraph@Sample-tg-3:~$
```

Output of gadmin status

4. Check the status again. All services should be up at this point:

```
gadmin status
```

Gadmin status after running gadmin start

5. TigerGraph has been successfully started on your cloud instance.

TigerGraph License on Microsoft Azure Images

The TigerGraph Enterprise edition image comes with **a perpetual license** that will **only work on the Microsoft Azure instance it's installed on**. Please run the following command to see it:

```
gadmin license status
```

```
tigergraph@Sample-tg-3:~$ gadmin license status
[Warning] License will expire in 106751 days

   Issuer: TigerGraph Cloud Marketplace Keygen
 Audience: Marketplace User
IssueTime: 2020-10-13 23:11:04 +0000 UTC
  EndTime: 3344-05-20 13:14:05 +0000 UTC
  Edition: Cloud Marketplace

Host:
        MaxCPUCore: 9.007199254740991e+15
        MaxPhysicalMemorySize: 8589934592.00MB
        MaxClusterNodeNumber: 3.34452e+06

Topology:
        MaxVertexNumber: 9.007199254740991e+15
        MaxEdgeNumber: 9.007199254740991e+15
        MaxGraphNumber: 5.201314e+06
        MaxTopologySize: 8589934592.00MB

RuntimeMemory:
        MaxUserResidentSetSize: N/A
tigergraph@Sample-tg-3:~$
```

Gadmin license status output

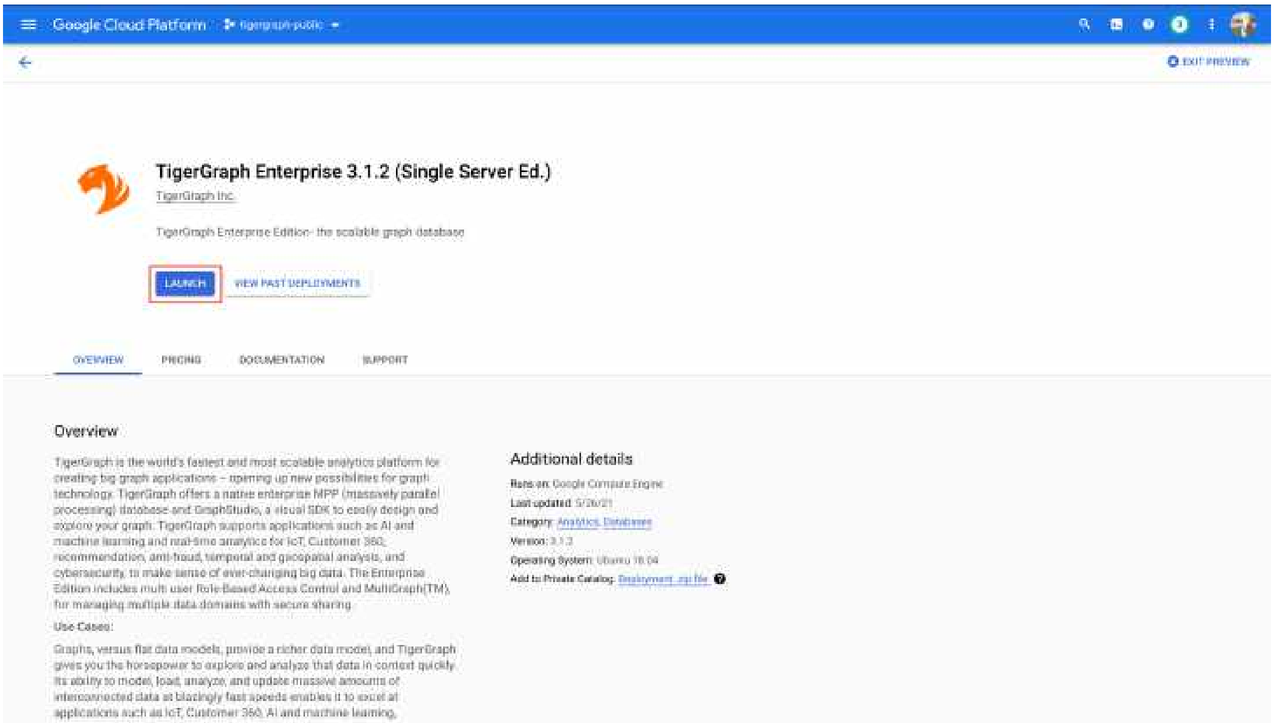
Get Started on Google Cloud

This tutorial will show you how to start TigerGraph from an image on Google Cloud. Please select your edition from below:

Deploying Your Instance

1. Go to Google Cloud Marketplace https://console.cloud.google.com/marketplace ↗ and search for "TigerGraph Enterprise Edition". Choose the latest edition to access the most up-to-date features.

2. Click on "Launch".



3. The default settings are fine for most users, but feel free to modify them. When ready, click on "Deploy".

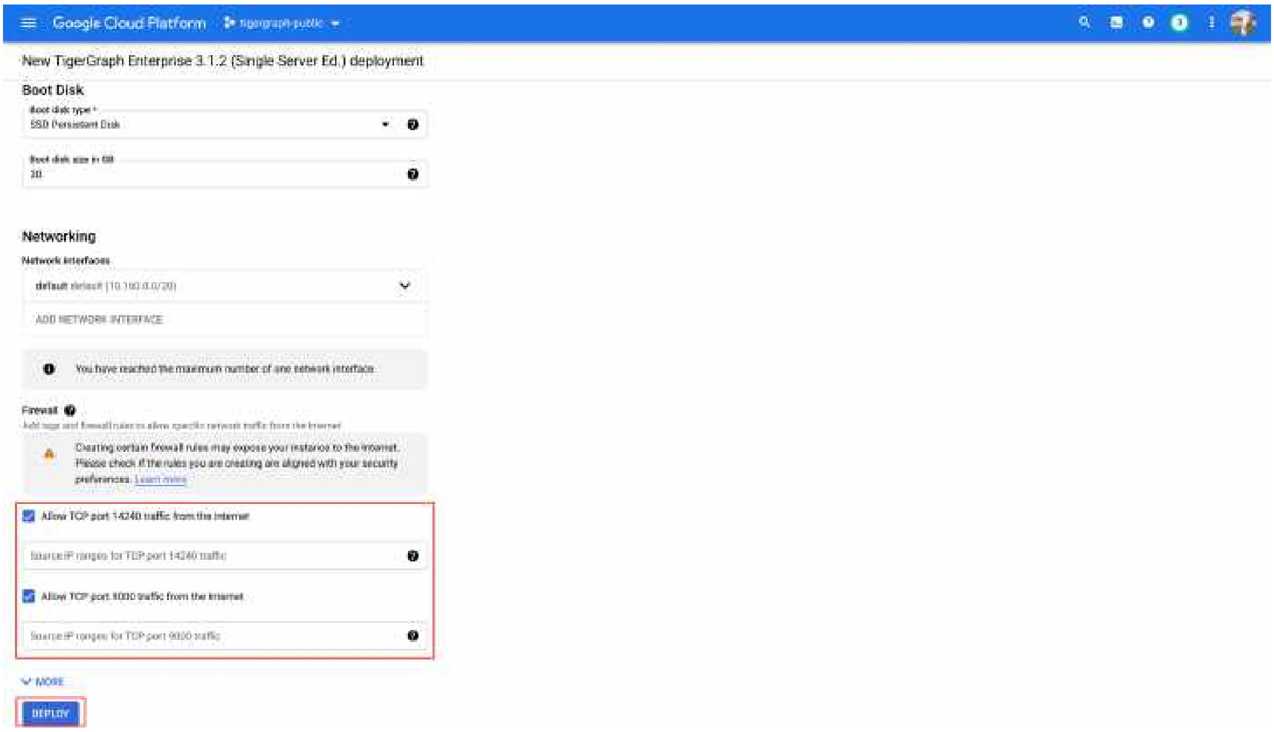
Notes:
The instance type needs to have at least 4 CPUs and 16GB RAM for TigerGraph to work properly.

The "Allow TCP port 14240 traffic from the Internet" checkbox must be checked if you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the GraphStudio UI Guide.

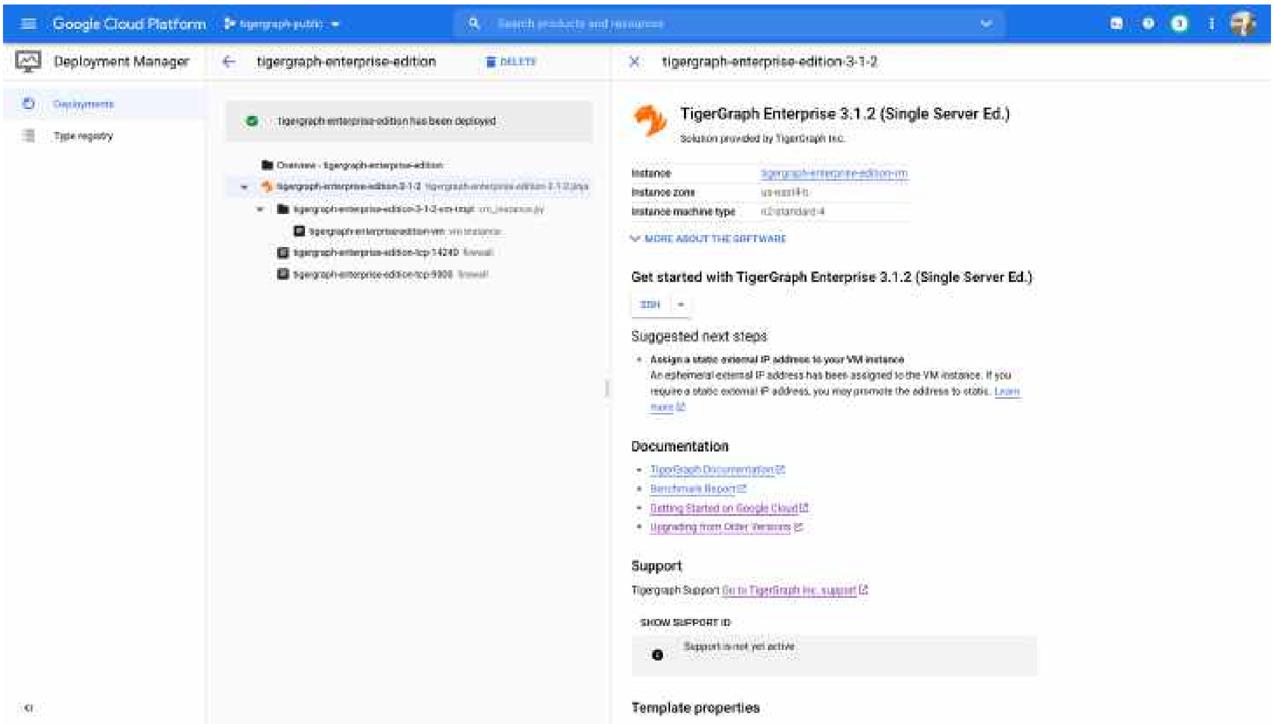
For information on how to set up authentication please see User access management.

The "Allow TCP port 9000 traffic from the Internet" checkbox must be checked if you want to send RESTful requests to TigerGraph from outside the instance (this includes configuring the GSQL client on a remote machine). For more about the REST API, see the TigerGraph RESTful API User Guide.

For more about the TigerGraph Platform, see the TigerGraph Platform Overview.



4. That's it! The TigerGraph instance has been successfully deployed on Google Cloud.



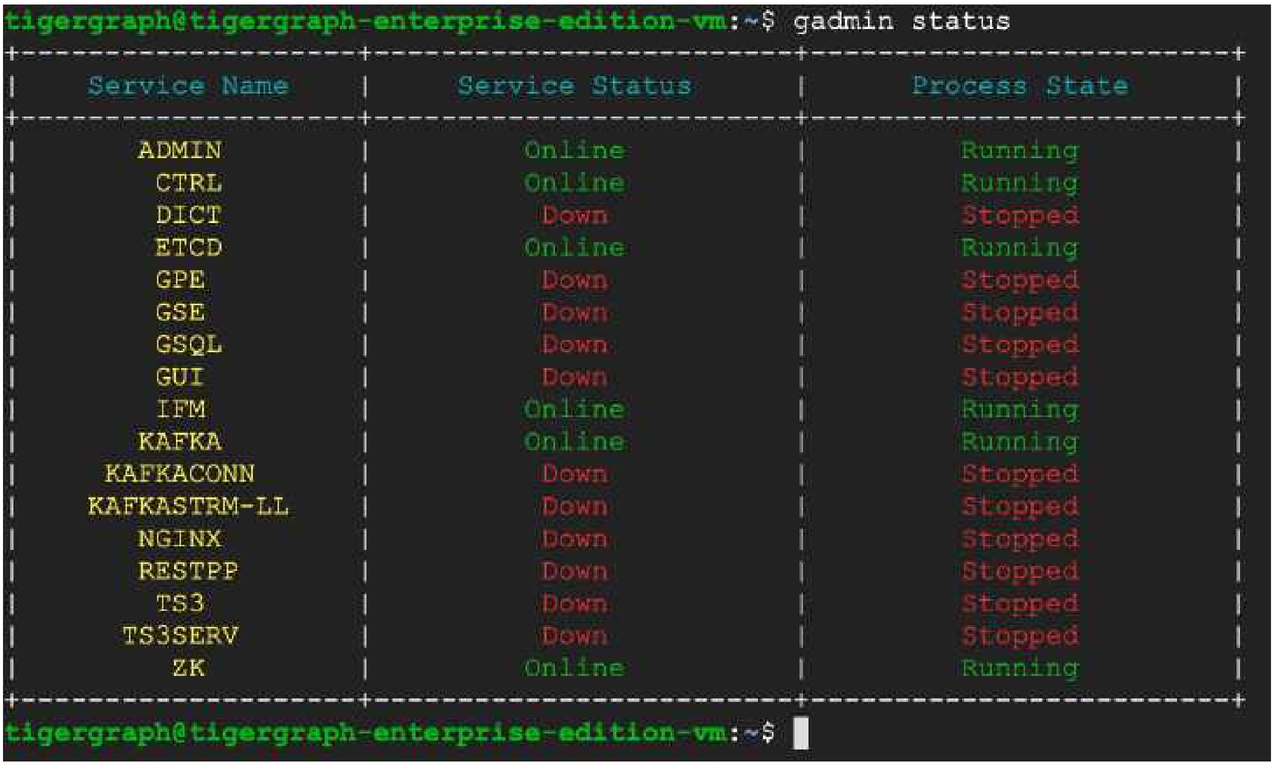
Starting TigerGraph on Your Instance

1. Log on to the instance and switch to user `tigergraph` using the following command:

```
sudo su - tigergraph
```

2. Run the following command to check the current status of TigerGraph. The services **"ADMIN", "CTRL", "ETCD", "IFM", "KAFKA", and "ZK" are started automatically** and should be up at this point. If any of them are not or you get the following error message, **please wait for 30 to 60 seconds and check the status again** before reporting it to TigerGraph support.

```
gadmin status
```

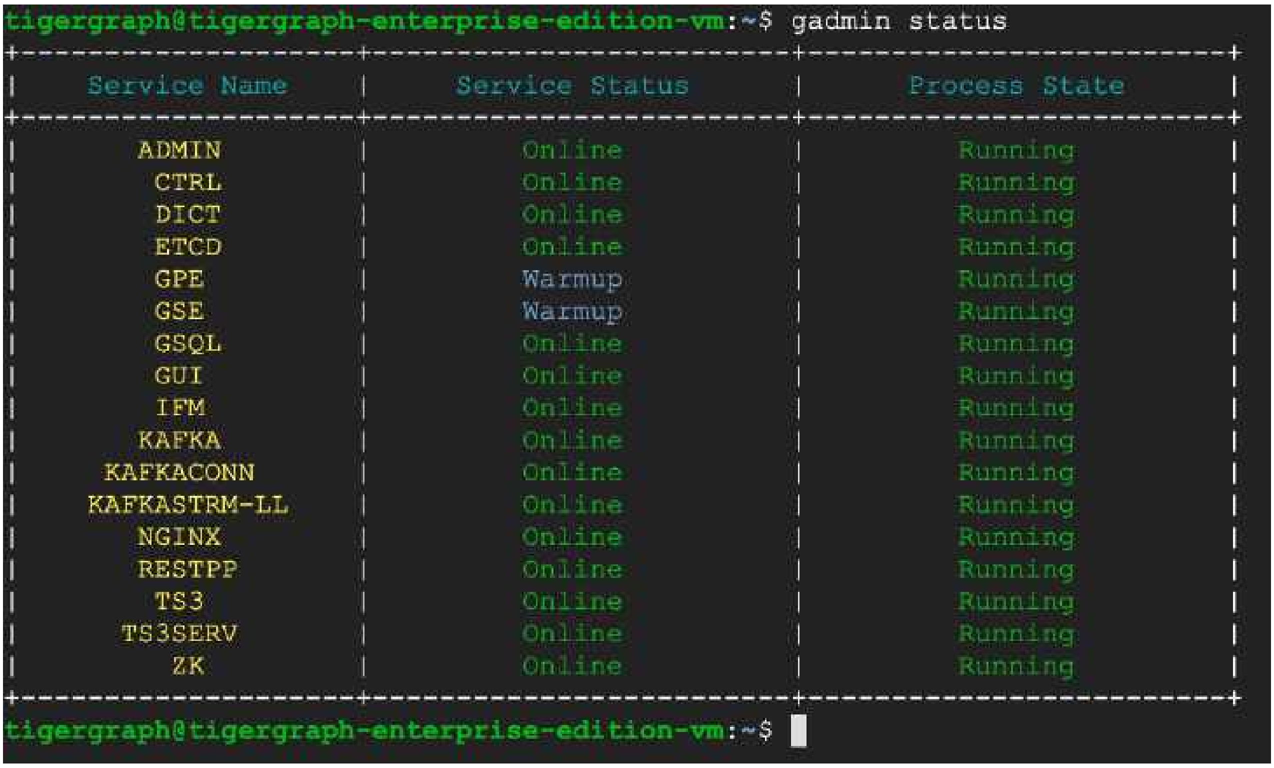


3. Run the following command to start TigerGraph:

```
gadmin start
```

4. Check the status again. All services should be up at this point:

```
gadmin status
```



5. TigerGraph has been successfully started on your cloud instance.

TigerGraph License on Google Cloud Images

The TigerGraph Enterprise edition image comes with **a perpetual license** that will **only work on the Google Cloud instance it's installed on**. Please run the following command to see it:

```
gadmin license status
```

```
tigergraph@tigergraph-enterprise-edition-vm:~$ gadmin license status
[Warning] License will expire in 106751 days

   Issuer: TigerGraph Cloud Marketplace Keygen
 Audience: Marketplace User
IssueTime: 2021-04-30 01:25:07 +0000 UTC
  EndTime: 3344-05-20 13:14:05 +0000 UTC
  Edition: Cloud Marketplace

Host:
        MaxCPUCore: 9.007199254740991e+15
        MaxPhysicalMemorySize: 8589934592.00MB
        MaxClusterNodeNumber: 3.34452e+06

Topology:
        MaxVertexNumber: 9.007199254740991e+15
        MaxEdgeNumber: 9.007199254740991e+15
        MaxGraphNumber: 5.201314e+06
        MaxTopologySize: 8589934592.00MB

RuntimeMemory:
        MaxUserResidentSetSize: N/A
tigergraph@tigergraph-enterprise-edition-vm:~$
```

GSQL 101

In this exercise, we will go through the 3-step process of writing GSQL-- define a schema, load data, and write a query.

This tutorial is written so that you can follow along and perform the steps on your TigerGraph system as your read.

Get Set

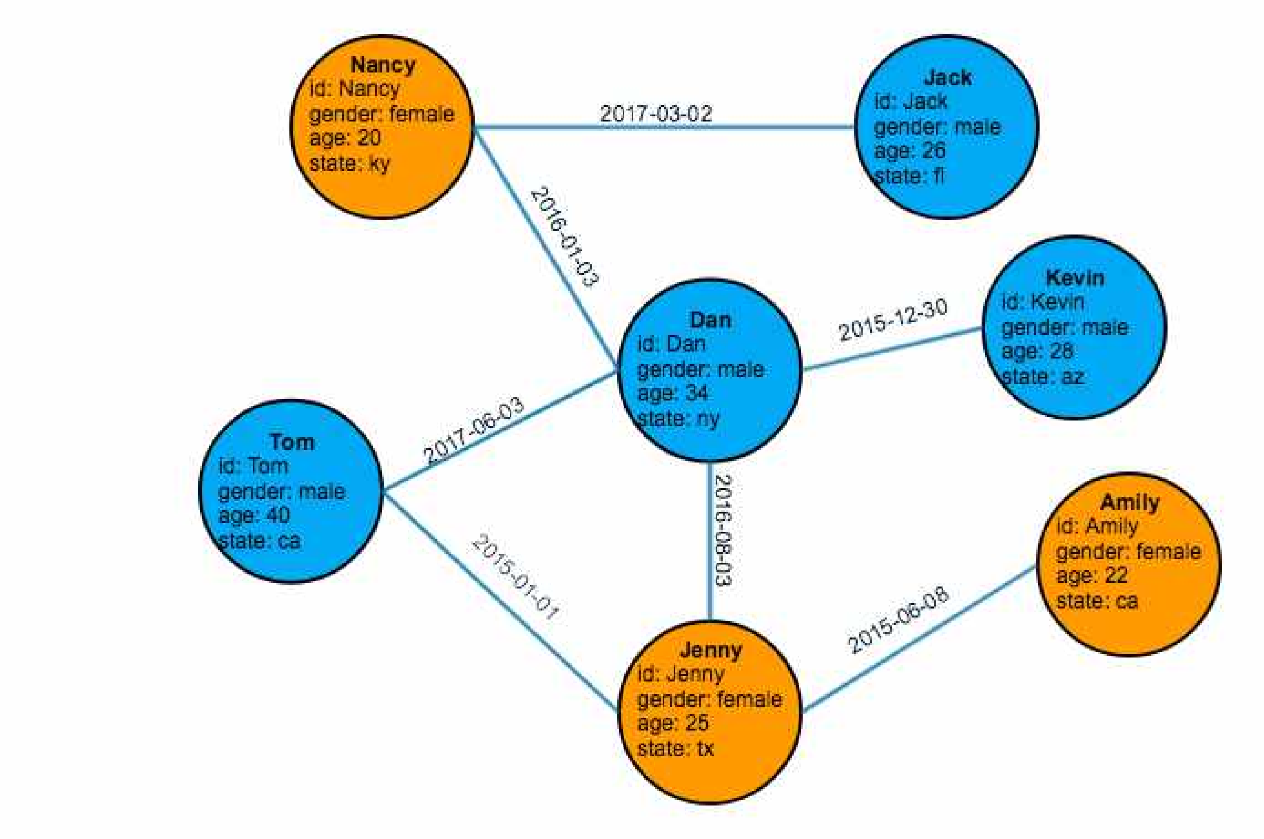
Introduction

In this tutorial, we will show you how to create a graph schema, load data in your graph, write simple parameterized queries, and run your queries. Before you start, you need to have installed the TigerGraph system, verified that it is working, and cleared out any previous data. It'll also help to become familiar with our graph terminology.

What is a Graph?

A graph is a collection of data entities and the connections between them. That is, it's a network of data entities.

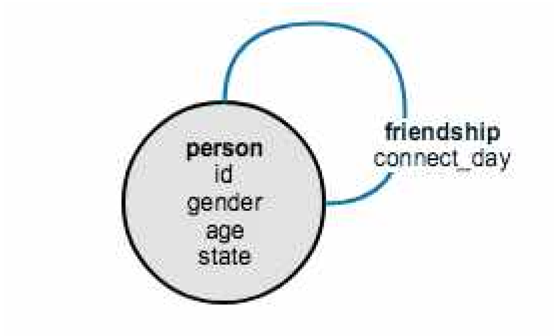
Many people call a data entity a **node** ; at TigerGraph we called it a **vertex**. The plural is **vertices**. We call a connection an edge. Both vertices and edges can have properties or attributes. The figure below is a visual representation of a graph containing 7 vertices (shown as circles) and 7 edges (the lines).



Friendship Social Graph

A graph schema is the model which describes the **types** of vertices (nodes) and edge (connections) which can appear in your graph. The graph above has one type of vertex (person) and one type of edge (friendship).

A schema diagram looks like a small graph, except each node represents one **type** of vertex, and each link represents one **type** of edge.



Friendship Social Graph Schema

The friendship loop shows that a friendship is between a person and another person.

Data Set

For this tutorial, we will create and query the simple friendship social graph shown in Figure Friendship Social Graph. The data for this graph consists of two files in csv (comma-separated values) format. To follow along with this tutorial, please save these two files, person.csv and friendship.csv, to your TigerGraph local disk. In our running example, we use the `/home/tigergraph/` folder to store the two csv files.

person.csv

```
name,gender,age,state
Tom,male,40,ca
Dan,male,34,ny
Jenny,female,25,tx
Kevin,male,28,az
Amily,female,22,ca
Nancy,female,20,ky
Jack,male,26,fl
```

friendship.csv

```
person1,person2,date
Tom,Dan,2017-06-03
Tom,Jenny,2015-01-01
Dan,Jenny,2016-08-03
Jenny,Amily,2015-06-08
Dan,Nancy,2016-01-03
Nancy,Jack,2017-03-02
Dan,Kevin,2015-12-30
```

Prepare Your TigerGraph Environment

First, let's check that you can access GSQL.

1. Open a Linux shell.
2. Type gsql as below. A GSQL shell prompt should appear as below.

   Linux Shell

```
$ gsql
GSQL >
```

3. If the GSQL shell does not launch, try resetting the system with "gadmin start all". If you need further help, please see manage TigerGraph with gadmin ↗ and TigerGraph Knowledge Base and FAQs .

If this is your first time using GSQL, the TigerGraph data store is probably empty. However, if you or someone else has already been working on the system, there may already be a database. You can check by listing out the database catalog with the "ls" command. This is what should look like if it is empty:

GSQL shell - an empty database catalog

```
GSQL > ls
---- Global vertices, edges, and all graphs
Vertex Types:
Edge Types:

Graphs:
Jobs:

Json API version: v2
```

If the data catalog is not empty, you will need to empty it to start this tutorial. We'll assume you have your coworkers' permission. Use the command DROP ALL to delete all the database data, its schema, and all related definitions. This command takes about a minute to run.

GSQL shell - DROP ALL

```
GSQL > drop all

Dropping all, about 1 minute ...
Abort all active loading jobs
[ABORT_SUCCESS] No active Loading Job to abort.

Shutdown restpp gse gpe ...
Graph store /usr/local/tigergraph/gstore/0/ has been cleared!
Everything is dropped.
```

Restarting TigerGraph

If you need to restart TigerGraph for any reason, use the following command sequence:

Linux Shell - Restarting TigerGraph services

```
# Switch to the user account set up during installation
# The default is user=tigergraph, password=tigergraph
$ su tigergraph
Password:tigergraph

# Start all services
$ gadmin restart -y
```

Running GSQL commands from Linux

You can also run GSQL commands from a Linux shell. To run a single command, just use "gsql" followed by the command line enclosed in single quotes. (The quotes aren't necessary if there is no parsing ambiguity; it's safer to just use them.) For example,

Linux shell - GSQL commands from a Linux shell

```
# "-g graphname" is need for a given graph
gsql -g social 'ls'
gsql 'drop all'
gsql 'ls'
```

You can also execute a series of commands which you have stored in a file, by simply invoking "gsql" following by the name of the file.

When you are done, you can exit the GSQL shell with the command "quit" (without the quotes).

Define a Schema

Introduction

For this tutorial, we will work mostly in the GSQL shell, in interactive mode. A few commands will be from a Linux shell. The first step in creating a GSQL graph is to define its schema. GSQL provides a set of DDL (Data Definition Language) commands, similar to SQL DDL commands, to model vertex types, edge types and a graph.

Create a Vertex Type

Use CREATE VERTEX command to define a vertex type named **person.** Here, PRIMARY_ID is required: each person must have a unique identifier. The rest is the optional list of attributes which characterize each person vertex, in the format `attribute_name data_type, attribute_name data_type, ...`

GSQL command

```
CREATE VERTEX person (
    PRIMARY_ID name STRING,
    name STRING, age INT,
    gender STRING, state STRING
)
```

We show GSQL keywords in ALL CAPS to highlight them, but they are case-insensitive.

GSQL will confirm the creation of the vertex type.

GSQL shell

```
GSQL > CREATE VERTEX person (PRIMARY_ID name STRING, name STRING, age INT
The vertex type person is created.
GSQL >
```

You can create as many vertex types as you need.

Create an Edge Type

Next, use the CREATE ... EDGE command to create an edge type named **friendship**. The keyword UNDIRECTED indicates this edge is a bidirectional edge, meaning that information can flow starting from either vertex. If you'd rather have a unidirectional connection where information flows only from the FROM vertex, use the DIRECTED keyword in place of UNDIRECTED. Here, FROM and TO are required to specify which two vertex types the edge type connects. An individual edge is specifying by giving the primary_ids of its source (FROM) vertex and target (TO) vertex. These are followed by an optional list of attributes, just as in the vertex definition.

GSQL command

```
CREATE UNDIRECTED EDGE friendship (FROM person, TO person, connect_day DA
```

GSQL will confirm the creation of the edge type.

GSQL shell

```
GSQL > CREATE UNDIRECTED EDGE friendship (FROM person, TO person, connect
The edge type friendship is created.
GSQL >
```

You can create as many edge types as you need.

Create a Graph

Next, use the CREATE GRAPH command to create a graph named **social.** Here, we just list the vertex types and edge types that we want to include in this graph.

GSQL command

```
CREATE GRAPH social (person, friendship)
```

GSQL will confirm the creation of the first graph after several seconds, during which it pushes the catalog information to all services, such as the GSE, GPE and RESTPP.

GSQL shell

```
GSQL > CREATE GRAPH social (person, friendship)

Restarting gse gpe restpp ...

Finish restarting services in 16.554 seconds!
The graph social is created.
```

At this point, we have created a **person** vertex type, a **friendship** edge type, and a **social** graph that includes them. You've now built your first graph schema! Let's take a look what's in the catalog by typing the `ls` command in the GSQL shell.

GSQL shell

```
GSQL > ls
---- Global vertices, edges, and all graphs
Vertex Types:
  - VERTEX person(PRIMARY_ID name STRING, name STRING, age INT, gender ST
Edge Types:
  - UNDIRECTED EDGE friendship(FROM person, TO person, connect_day DATETI

Graphs:
  - Graph social(person:v, friendship:e)
Jobs:


Json API version: v2
```

Load Data

After creating a graph schema, the next step is to load data into it. The task here is to instruct the GSQL loader how to associate ("map") the fields in a set of data files to the attributes in your vertex types and edge types of the graph schema we just defined.

You should have the two data files person.csv and friendship.csv on your local disk. It's not necessary that they are in the same folder with you.

If you need to exit the GSQL shell for any reason, you can do so by typing "quit" without the quotes. Type gsql to enter again.

Define a Loading Job

The loading job below assumes that your data files are in the folder `/home/tigergraph`. If they are elsewhere, then in the loading job script below replace `/home/tigergraph/person.csv` and `/home/tigergraph/friendship.csv` with their corresponding file path respectively. Assuming you're (back) in the GSQL shell, enter the following set of commands.

GSQL commands to define a loading job

```
USE GRAPH social
BEGIN
CREATE LOADING JOB load_social FOR GRAPH social {
   DEFINE FILENAME file1="/home/tigergraph/person.csv";
   DEFINE FILENAME file2="/home/tigergraph/friendship.csv";

   LOAD file1 TO VERTEX person VALUES ($"name", $"name", $"age", $"gender
   LOAD file2 TO EDGE friendship VALUES ($0, $1, $2) USING header="true",
}
END
```

Let's walk through the commands:

- `USE GRAPH social` :
  Tells GSQL which graph you want to work with.

- `BEGIN ... END`:
  Indicates multiple-line mode. The GSQL shell will treat everything between these markers as a single statement. These is only needed for interactive mode. If you run GSQL statements that are stored in a command file, the command interpreter will study your whole file, so it doesn't need the BEGIN and END hints.
- `CREATE LOADING JOB`:
  One loading job can describe the mappings from multiple files to multiple graph objects. Each file must be assigned to a filename variable. The field labels can be either by name or by position. By-name labelling requires a header line in the source file. By-position labelling uses integers to indicate source column position 0, 1,... In the example above, the first LOAD statement refers to the source file columns by name, whereas the second LOAD statement refers to the source file columns by position. Note the following details:
  - The column "name" in file1 gets mapped to two fields, both the PRIMARY_ID and the "name" attribute of the person vertex.
  - In file1, gender comes before age. In the person vertex, gender comes after age. When loading, state your attributes in the order needed by the target object (in this case, the person vertex).
  - Each LOAD statement has a USING clause. Here it tells GSQL that both files contain a header (whether we choose to use the names or not, GSQL still needs to know whether to consider the first line as data or not). It also says the column separator is comma. GSQL can handle any single-character separator, not just commas.

When you run the CREATE LOADING JOB statement, GSQL checks for syntax errors and checks that you have data files in the locations specified. If it detects no errors, it compiles and saves your job.

GSQL shell

```
GSQL > USE GRAPH social
Using graph 'social'
GSQL > BEGIN
GSQL > CREATE LOADING JOB load_social FOR GRAPH social {
GSQL >    DEFINE FILENAME file1="/home/tigergraph/person.csv";
GSQL >    DEFINE FILENAME file2="/home/tigergraph/friendship.csv";
GSQL >
GSQL >    LOAD file1 TO VERTEX person VALUES ($"name", $"name", $"age", $
GSQL >    LOAD file2 TO EDGE friendship VALUES ($0, $1, $2) USING header=
GSQL > }
GSQL > END
The job load_social is created.
```

Run a Loading Job

You can now run your loading job to load data into your graph:

GSQL command

```
RUN LOADING JOB load_social
```

The result is shown below.

GSQL shell

```
GSQL > run loading job load_social
[Tip: Use "CTRL + C" to stop displaying the loading status update, then u
[Tip: Manage loading jobs with "ABORT/RESUME LOADING JOB jobid"]
Starting the following job, i.e.
  JobName: load_social, jobid: social_m1.1528095850854
  Loading log: '/home/tigergraph/tigergraph/logs/restpp/restpp_loader_log

Job "social_m1.1528095850854" loading status
[FINISHED] m1 ( Finished: 2 / Total: 2 )
  [LOADED]
  +-----------------------------------------------------------------------
  |                      FILENAME |   LOADED LINES |   AVG SPEED |   DUR
  |/home/tigergraph/friendship.csv |              8 |       8 l/s |     1
  |    /home/tigergraph/person.csv |              8 |       7 l/s |     1
  +-----------------------------------------------------------------------
```

Notice the location of the loading log file. The example assumes that you installed TigerGraph in the default location, `/home/tigergraph/`. In your installation folder is the main product folder, tigergraph. Within the tigergraph folder are several subfolders, such as logs, document, config, bin, and gstore. If you installed in a different location, say `/usr/local/`, then you would find the product folder at `/usr/local/tigergraph`.

Run Built-in Queries

You now have a graph with data! You can run some queries using the built-in REST endpoint calls.

Get Vertex/Edge Statistics

Below we call two functions, stat_vertex_number and stat_edge_number to return the cardinality of each vertex and edge type.

REST endpoints return results in JSON format. JSON data are used for various purposes. But JSON data can’t be read easily from JSON file by using bash script like other normal files. **jq** tool is used to solve this problem

We recommend you install jq and redirect the REST call result to jq before it is output.

Linux Shell

```
#get vertex cardinality
curl -X POST 'http://localhost:9000/builtins/social' -d  '{"function":"st
```

```
#results
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_type": "person",
      "count": 7
    }
  ]
}
```

Linux Shell

```
#get edge cardinality
curl -X POST 'http://localhost:9000/builtins/social' -d  '{"function":"st
```

```
#results
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "friendship",
      "count": 7
    }
  ]
}
```

Select Vertices

If you want to lookup the details about a vertex with its primary_id, you can use the following REST call.

Linux Shell

```
curl -X GET "http://localhost:9000/graph/{graph_name}/vertices/{vertex_ty
```

Example. Find a person vertex whose primary_id is "Tom".

Linux Shell

```
  curl -X GET "http://localhost:9000/graph/social/vertices/person/Tom" | j
```

```
#result
{
 "version": {
   "edition": "enterprise",
   "api": "v2",
   "schema": 0
 },
 "error": false,
 "message": "",
 "results": [
   {
     "v_id": "Tom",
     "v_type": "person",
     "attributes": {
       "name": "Tom",
       "age": 40,
       "gender": "male",
       "state": "ca"
     }
   }
 ]
}
```

Select Edges

In similar fashion, we can see details about edges. To describe an edge, you name the types of vertices and edges in the two parts or three parts of a URL.

Linux Shell

```
#two parts
curl -X GET "http://localhost:9000/graph/edges/{source_vertex_type}/{sour

#three parts
curl -X GET "http://localhost:9000/graph/edges/{source_vertex_type}/{sour
```

Example. Find all friendship edges whose source vertex's primary_id is "Tom".

Linux Shell

```
curl -X GET "http://localhost:9000/graph/social/edges/person/Tom/friendsh
```

```
#result
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "friendship",
      "directed": false,
      "from_id": "Tom",
      "from_type": "person",
      "to_id": "Dan",
      "to_type": "person",
      "attributes": {
        "connect_day": "2017-06-03 00:00:00"
      }
    },
    {
      "e_type": "friendship",
      "directed": false,
      "from_id": "Tom",
      "from_type": "person",
      "to_id": "Jenny",
      "to_type": "person",
      "attributes": {
        "connect_day": "2015-01-01 00:00:00"
      }
    }
  ]
}
```

For more built-in REST endpoints, you can refer to the Built-in Endpoints page.

Develop Parameterized Queries

Develop, install, and run parameterized GSQL queries

We just saw how easy and quick it is to run simple built-in queries. However you'll undoubtedly want to create more customized or complex queries. GSQL puts maximum power in your hands through parameterized vertex set queries. Parameterized queries let you traverse the graph from one vertex set to an adjacent set of vertices, again and again, performing computations along the way, with built-in parallel execution and handy aggregation operations. You can even have one query call another query. But we'll start simple.

A GSQL parameterized query has three steps.

1. Define your query in GSQL. This query will be added to the GSQL catalog.
2. Install one or more queries in the catalog, generating a REST endpoint for each query.
3. Run an installed query, supplying appropriate parameters, either as a GSQL command or by sending an HTTP request to the REST endpoint.

A Simple 1-Hop Query

Now, let's write our first GSQL query. We'll display all the direct (1-hop) neighbors of a person, given as an input parameter.

GSQL command

```
USE GRAPH social
CREATE QUERY hello(VERTEX<person> p) {
  Start = {p};
  Result = SELECT tgt
           FROM Start:s-(friendship:e) ->person:tgt;
  PRINT Result;
}
```

This query features one SELECT statement. The SELECT statements here are much more powerful than the ones in built-in queries. Here you can do the following:The query starts by seeding a vertex set "Start" with the person vertex identified by parameter *p* passed in from the query call. The curly braces tell GSQL to construct a set containing the enclosed items.

Next, the SELECT statement describes a 1-hop traversal according to the pattern described in the FROM clause:

`Start:s -(friendship:e)-> person:tgt`

This is basically the same syntax we used for the built-in select edges query. Namely, we select all edges beginning from the given source set (Start), which have the given edge type (friendship) and which end at the given vertex type (person). A feature we haven't seen before is the use of vertex and edge set aliases defined by ":alias": "s" is the alias for the source vertex set, "e" is the edge set alias, and "tgt" is the target vertex set alias.

Refer back to the initial clause and the assignment ("`Result = SELECT tgt`"). Here we see the target set's alias tgt. This means that the SELECT statement should return the target vertex set (as filtered and processed by the full set of clauses in the SELECT query block) and assign that output set to the variable called Result.

Last, we print out the Result vertex set, in JSON format.

Create A Query

Rather than defining our query in interactive mode, we can store the query in a file and invoke the file from within the GSQL shell, using the @filename syntax. Copy and paste the above query into a file `/home/tigergraph/hello.gsql`. Then, enter the GSQL shell and invoke the file using @hello.qsql (Note that if you are not in the /home/tigergraph folder when you start gsql, then you can use the absolute path to invoke a gsql file. e.g., `@/home/tigergraph/hello.gsql`) Then run the "ls" command to see that the query is now in the catalog.

GSQL shell

```
GSQL > @hello.gsql
Using graph 'social'
The query hello has been added!
GSQL > ls
---- Graph social
Vertex Types:
  - VERTEX person(PRIMARY_ID name STRING, name STRING, age INT, gender ST
Edge Types:
  - UNDIRECTED EDGE friendship(from person, to person, connect_day DATETI

Graphs:
  - Graph social(person:v, friendship:e)
Jobs:
  - CREATE LOADING JOB load_social FOR GRAPH social {
      DEFINE FILENAME file2 = "/home/tigergraph/friendship.csv";
      DEFINE FILENAME file1 = "/home/tigergraph/person.csv";

      LOAD file1 TO VERTEX person VALUES($"name", $"name", $"age", $"gend
      LOAD file2 TO EDGE friendship VALUES($0, $1, $2) USING SEPARATOR=",
    }

Queries:
  - hello(vertex<person> p)
```

Install a Query

However, the query is not installed yet; it is not ready to run. In the GSQL shell, type the following command to installed the just added query "hello".

GSQL command

```
INSTALL QUERY hello
```

GSQL shell

```
GSQL > INSTALL QUERY hello
Start installing queries, about 1 minute ...
hello query: curl -X GET 'http://127.0.0.1:9000/query/social/hello?p=VALU

[========================================================================
```

It takes about 1 minute for the database to install this new query. Be patient! For queries on large datasets, this small investment pays off many times over in faster query execution, particularly if you will run the query many times, with different parameters. The installation will generate machine instructions and a REST endpoint. After the progress bar reaches 100%, we are ready to run this query.

Run a Query in GSQL

To run a query in GSQL, use "RUN QUERY" followed by the query name and a set of parameter values.

GSQL command - run query examples

```
RUN QUERY hello("Tom")
```

The result is presented in JSON format. Tom has two 1-hop neighbors, namely Dan and Jenny.

GSQL shell

```
GSQL > RUN QUERY hello("Tom")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [
    {
      "v_id": "Dan",
      "attributes": {
        "gender": "male",
        "name": "Dan",
        "state": "ny",
        "age": 34
      },
      "v_type": "person"
    },
    {
      "v_id": "Jenny",
      "attributes": {
        "gender": "female",
        "name": "Jenny",
        "state": "tx",
        "age": 25
      },
      "v_type": "person"
    }
  ]}]
}
```

Run a Query as a REST Endpoint

Under the hood, installing a query will also generate a REST endpoint, so that the parameterized query can be invoked by an http call. In Linux, the curl command is the most popular way to submit an http request. In the example below, the portion that is standard for all queries is shown in bold ; the portion in normal weight pertains to this particular query and parameter value. The JSON result will be returned to the Linux shell's standard output. So, our parameterized query becomes a http service!

Linux shell

```
curl -X GET 'http://localhost:9000/query/social/hello?p=Tom'
```

Finally, to see the GSQL text of a query in the catalog, you can use

GSQL command - show query example

```
#SHOW QUERY query_name. E.g.
SHOW QUERY hello
```

Congratulations! At this point, you have gone through the whole process of defining, installing, and running a query.

Running Anonymous Queries Without Installing

Installing query will give the fastest query speed, but user needs to wait for the installation overhead.

The Interpreted Mode for GSQL, introduced in TigerGraph 2.4, let us skip the INSTALL step, and even run a query as soon as we create it, to offer a more interactive experience. These one-step interpreted queries are unnamed (anonymous) and parameterless, just like SQL. Please refer to GSQL 102 for this mode.

A More Advanced Query

Now, let's do a more advanced query. This time, we are going to learn to use the powerful built-in accumulators, which serves as the runtime attributes (properties) attachable to each vertex visited during our traversal on the graph. Runtime means they exist only while the query is running; they are called accumulators because they are specially designed to gather (accumulate) data during an implicitly parallel processing of the query.

GSQL command file - hello2.gsql

```
USE GRAPH social
CREATE QUERY hello2 (VERTEX<person> p) {
  OrAccum  @visited = false;
  AvgAccum @@avgAge;
  Start = {p};

  FirstNeighbors = SELECT tgt
                  FROM Start:s -(friendship:e)-> person:tgt
                  ACCUM tgt.@visited += true, s.@visited += true;

  SecondNeighbors = SELECT tgt
                   FROM FirstNeighbors -(:e)-> :tgt
                   WHERE tgt.@visited == false
                   POST_ACCUM @@avgAge += tgt.age;

  PRINT SecondNeighbors;
  PRINT @@avgAge;
}
INSTALL QUERY hello2
RUN QUERY hello2("Tom")
```

In this query we will find all the persons which are exactly 2 hops away from the parameterized input person. Just for fun, let's also compute the average age of those 2-hop neighbors.

In the standard approach for this kind of graph traversal algorithm, you use a boolean variable to mark the first time that the algorithm "visits" a vertex, so that it knows not to count it again. To fit this need, we'll define a local accumulator of the type OrAccum. To declare a local accumulator, we prefix an identifier name with a single "@" symbol. Each accumulator type has a default initial value; the default value for boolean accumulators is false. Optionally, you can specify an initial value.

We also want to compute one average, so we will define a global AvgAccum. The identifier for a global accumulator begins with two "@"s.

After defining the Start set, we then have our first one 1-hop traversal. The SELECT and FROM clauses are the same as in our first example, but there is an additional ACCUM clause. The += operator within an ACCUM clause means that for each edge matching the FROM clause pattern, we accumulate the right-hand-side expression (true) to the left-hand-accumulator (tgt.@visited as well as s.@visited).

Note that a source vertex or target vertex may be visited multiple times. Referring to Figure 1, if we start at vertex Tom, there are two edges incidents to Tom, so the ACCUM clause in the first SELECT statement will visit Tom two times. Since the accumulator type is OrAccum, the cumulative effect of the two traversals is the following:

Tom.@visited ⇐ (initial value: false) OR (true) OR (true)

Note that it does not matter which of the two edges was processed first, so this operation is suitable for multithreaded parallel processing. The net effect is that as long as a vertex is visited at least once, it will end up with @visited = true. The result of this first SELECT statement is assigned to the variable FirstNeighbors.

The second SELECT block will do one hop further, starting from the FirstNeighbors vertex set variable, and reaching the 2-hop neighbors. Note that this time, we have omitted the edge type friendship and the target vertex type person from the FROM clause, but we retained the aliases. If no type is mentioned for an alias, then it is interpreted as ALL types. Since our graph has only one vertex type and one edge type, it is logically the same as if we had specified the types. The WHERE clause filters out the vertices which have been marked as visited before (the 1-hop neighbors and the starting vertex *p*). This SELECT statement uses POST_ACCUM instead of ACCUM. The reason is that POST_ACCUM traverses the vertex sets instead of the edge sets, guaranteeing that we do not double-count any vertices. Here, we accumulate the ages of the 2-hop neighbors to get their average.

Finally, the SecondNeighbors of p are printed out.

This time, we put all of the following GSQL commands into one file hello2.gsql:

- USE GRAPH social
- The query definition
- Installing the query
- Running the query

We can execute this full set of commands *without* entering the GSQL shell. Please copy and paste the above GSQL commands into a Linux file named

/home/tigergraph/hello2.gsql.

In a Linux shell, under /home/tigergraph, type the following:

Linux shell

```
gsql hello2.gsql
```

GSQL Query Summary:

- Queries are installed in the catalog and can have one or more input parameters, enabling reuse of queries.
- A GSQL query consists of a series of SELECT query blocks, each generating a named vertex set.
- Each SELECT query block can start traversing the graph from any of the previously defined vertex sets (that is, the sequence does not have to form a linear chain).
- Accumulators are runtime variables with built-in accumulation operations, for efficient multithreaded computation.
- Query can call another query.
- Output is in JSON format.

Review

You have learned a lot in GSQL 101!

With just the knowledge from GSQL 101 and a little practice, you should be able to do the following:

- Create a graph schema containing multiple vertex types and edge types.
- Define a loading job that takes one or more CSV files and maps the data directly to the vertices and edges of your graph.
- Write and run simple parameterized queries which start at one vertex and then traverse one or more hops to generate a final vertex set. Make a simple additive computation and return the results.

Want to learn more?

- To learn to do the same types of operations using the GraphStudio Visual SDK and UI, see the TigerGraph GraphStudio UI Guide.
- Continue learning Accumulators and Pattern Matching.
- To see more GSQL examples, see IS ↗, IC ↗, and BI ↗ workloads based on LDBC ↗ benchmark schema ↗, and GSQL Demo Examples .
- To get answers to common questions, see the TigerGraph Knowledge Base and FAQs .
- To see the full GSQL specification (whose table of contents with give you and idea of what is available) see
  - GSQL Language Reference Part 1 - Defining Graphs and Loading Data
  - GSQL Language Reference Part 2 - Querying
- To discuss and get help with fellow GSQL users and GSQL developers, join GSQL community forum ↗.

Accumulators Tutorial

Introduction

GSQL is a Turing complete Graph Database query language. Comparing to other graph query languages, the biggest advantages is its support of accumulators -- global or vertex local.

In addition to provide the classic pattern match syntax, which is easy to master, GSQL supports powerful run-time vertex attributes (a.k.a local accumulators) and global state variables (a.k.a global accumulators).

This short tutorial aims to shorten the learning curve of accumulator. Supposedly, after reading this article, everyone can master the essence of accumulator by heart, and start solving real-life graph problems with this handy language feature.

What is an Accumulator?

Figure 1. The left box is a GSQL query with different accumulators being accumulated to. The right box shows the accumulator variables' final results.

An accumulator is a state variable in GSQL. Its state is mutable throughout the life cycle of a query. It has an initial value, and users can keep accumulating (using its "+=" built-in operator) new values into it. Each accumulator variable has a type. The type decides what semantics the declared accumulator will use to interpret the "+=" operation.

In Figure 1's left box, from line 3 to line 8, six different accumulator variables (those with prefix @@) are declared, each with a unique type. Below we explain the semantic and usage of them.

- **SumAccum<INT>** allows user to keep adding INT values into its internal state variable. As the line 10 and 11 have shown, we added 1 and 2 to the accumulator, and end up with the value 3 (shown on line 3 in the right box).

- **MinAccum<INT>** keeps the smallest INT number it has seen. As the line 14 and 15 have shown, we accumulated 1 and 2 to the MinAccum accumulator, and end up with the value 1 (shown on line 6 in the right box).
- **MaxAccum<INT>** is symmetric to MinAccum. It returns the MAX INT value it has seen. Lines 18 and 19 show that we send 1 and 2 into it, and end up with the value 2 (shown on line 9 in the right box).
- **OrAccum** keeps OR-ing the internal boolean state variable with new boolean variables that accumulate to it. The initial default value is FALSE. Lines 22 and 23 show that we send TRUE and FALSE into it, and end up with the TRUE value (shown on line 12 in the right box).
- **AndAccum** is symmetric to OrAccum. Instead of using OR, it uses the AND accumulation semantics. Line 26 and 27 show that we accumulate TRUE and FALSE into it, and end up with the FALSE value (shown on line 15 in the right box).
- **ListAccum<INT>** keeps appending new integer(s) into its internal list variable. Line 30 - 32 show that we append 1, 2, and [3,4] to the accumulator, and end up with [1,2,3,4] (shown on lines 19-22 in the right box).

Global vs. Vertex-attached Accumulator

At this point, we have seen that accumulators are special typed variable in GSQL language. We are ready to explain their global and local scopes.

Global accumulator belongs to the entire query. Anywhere in a query, a statement can update its value. Local accumulator belongs to each vertex. It can only be updated when its owning vertex is accessible. To differentiate them, we use special prefixes in the identifier when we declare them.

- **@@** prefix is used for declaring global accumulator variable. It is always used stand-alone. E.g
  `@@cnt +=1`
- **@** prefix is used for declaring local accumulator variable. It must be used with a vertex alias in a query block. E.g. `v.@cnt += 1,` where v is a vertex alias specified in a FROM clause of a SELECT-FROM-WHERE query block.

Figure 2. A social graph with 7 person vertices and 7 friendship edges connecting them.

Consider a toy social graph modeled by a person vertex type and a person-to-person friendship edge type shown in Figure 2. Below we write a query, which accepts a person, and does a 1-hop traversal from the input person to its neighbors. We use the @@global_edge_cnt accumulator to accumulate the total number of edges we traverse. And we use @vertex_cnt to write to the input person's each friend vertex an integer 1.

Figure 3. The top box shows a query that given a person, accumulate the edge count into @@global_edge_cnt. The bottom box shows that for each friend of the input person, we accumulate 1 into its @vertex_cnt.

As Figure 2 shows, Dan has 4 direct friends -- Tom, Kevin, Jenny, and Nancy, each of which holds a local accumulator @vertex_cnt. And the @@global_edge_cnt has value 4, reflecting the fact that for each edge, we have accumulated 1 into it.

ACCUM vs. POST-ACCUM

ACCUM and POST-ACCUM clauses are computed in stages, where-in a SELECT-FROM-WHERE query block, ACCUM is executed first, followed by the POST-ACCUM clause.

- **ACCUM** executes its statement(s) once for each matched edge (or path) of the FROM clause pattern. Further, ACCUM parallelly executes its statements for all the matches.
- **POST-ACCUM** executes its statement(s) once for each involved vertex. Note that each statement within the POST-ACCUM clause can refer to either source vertices or target vertices but not both. Its statements can access the aggregated accumulator result computed in the ACCUM clause.

Conclusion

We have explained the mechanism of accumulators, their types, and the two different scopes--global and local. We also elaborate the ACCUM and POST-ACCUM clause semantics. Once you master the basics, the rest is to practice more. We have made available 46 queries based on the LDBC ↗ schema. These 46 queries are divided into three groups.

- **IS: https://github.com/tigergraph/ecosys/tree/master/tools/ldbc_benchmark/tigergraph/queries_pattern_match/interactive_short ↗**
- **IC: https://github.com/tigergraph/ecosys/tree/master/tools/ldbc_benchmark/tigergraph/queries_pattern_match/interactive_complex ↗**
- **BI: https://github.com/tigergraph/ecosys/tree/master/tools/ldbc_benchmark/tigergraph/queries_pattern_match/business_intelligence ↗**

You can follow GSQL 102 to setup the environment. You can also post your feedback and questions on the GSQL community forum ↗. Our community members and developers love to hear any feedback from your graph journey of using GSQL and are ready to help clarifying any doubts.

GSQL102 Pattern Matching

A guide to GSQL's multi-hop SELECT statements which make it convenient to express pattern-matching queries.

Get Set

Introduction

In this tutorial, we will show you how to write and run Pattern Matching queries. Pattern Matching is available in TigerGraph v2.4+.

We assume you have finished GSQL 101. If not, please complete GSQL 101 first.

> This tutorial was updated for TigerGraph 3.0. If you are using an older version, please change to the documentation for that version.

What is a Graph Pattern?

A graph pattern is a traversal trace on the graph schema. A pattern can contain repeated steps. A pattern can be a linear trace, or a non-linear trace (tree, circle etc.). For example, imagine a simple schema consisting of a Person vertex type and a Friendship edge type. A pattern could be a trace on this simple schema,

```
Person - (Friendship) - Person - (Friendship) - Person
```

or, use *2 to denote the two consecutive Friendship edges,

```
Person - (Friendship*2) - Person
```

What is Pattern Matching?

Pattern matching is the process of finding subgraphs in a data graph that conform to a given query pattern.

Prepare Your TigerGraph Environment

We assume you are running your own TigerGraph instance as the sole user with full privileges. If you are on a multiuser Enterprise Edition, consult with your DB administrator. You need to have Designer or Admin privilege on an empty graph. At various points in this tutorial, there are links to download files. Most are small, but the graph data file is 1GB when uncompressed.

First, let's check that you can access GSQL, and that your version is 3.0 or higher.

1. Open a Linux shell.
2. Type `gsql` as below. A GSQL shell prompt should appear as below.
3. Type version in GSQL shell. It should show 2.4 or higher as below. If not, please download and install the latest version from https://www.tigergraph.com/download/ ↗

   Linux Shell

   ```
   $ gsql
   GSQL > version
   GSQL version: 3.0
   ```

4. If the GSQL shell does not launch, try resetting the system with "gadmin start all". This will launch each service if they have not been started yet. If you need further help, please see the manage TigerGraph with gadmin, and TigerGraph Knowledge Base and FAQs.
5. You need to start from an empty data catalog. If necessary, run `drop all` to clear the catalog first.

Cheatsheet

The following general use commands were introduced in GSQL 101.

- The `%` prefix indicates Linux shell commands. You need TigerGraph admin privilege to run most gadmin commands.
- The `GSQL>` prefix indicates GSQL shell commands.

Command	Description
% gsql	Enter the GSQL shell in interactive mode
% gsql '<GSQL command string>'	Run one GSQL command
% gadmin status	Check the status of TigerGraph services (If your graph store is empty, it is normal for some statuses to be flagged in red.)
% gadmin restart	Restart TigerGraph services
GSQL> ls	List the graph schema, loading jobs, and queries
GSQL> show user	Show your user name and roles
GSQL> drop all	Delete the entire schema, all data, all jobs, and all queries
GSQL> exit	Exit GSQL interactive shell

Define the Schema

Data Set

We will use the LDBC Social Network Benchmark ↗ (LDBC SNB) data set. This data set models a typical social forum, where communities of persons can post messages on a forum to discuss a topic. It comes with a data generator, which allows you to generate data at different scale factors. Scale factor 1 generates roughly 1GB of raw data, scale factor 10 generates roughly 10GB of raw data, etc.

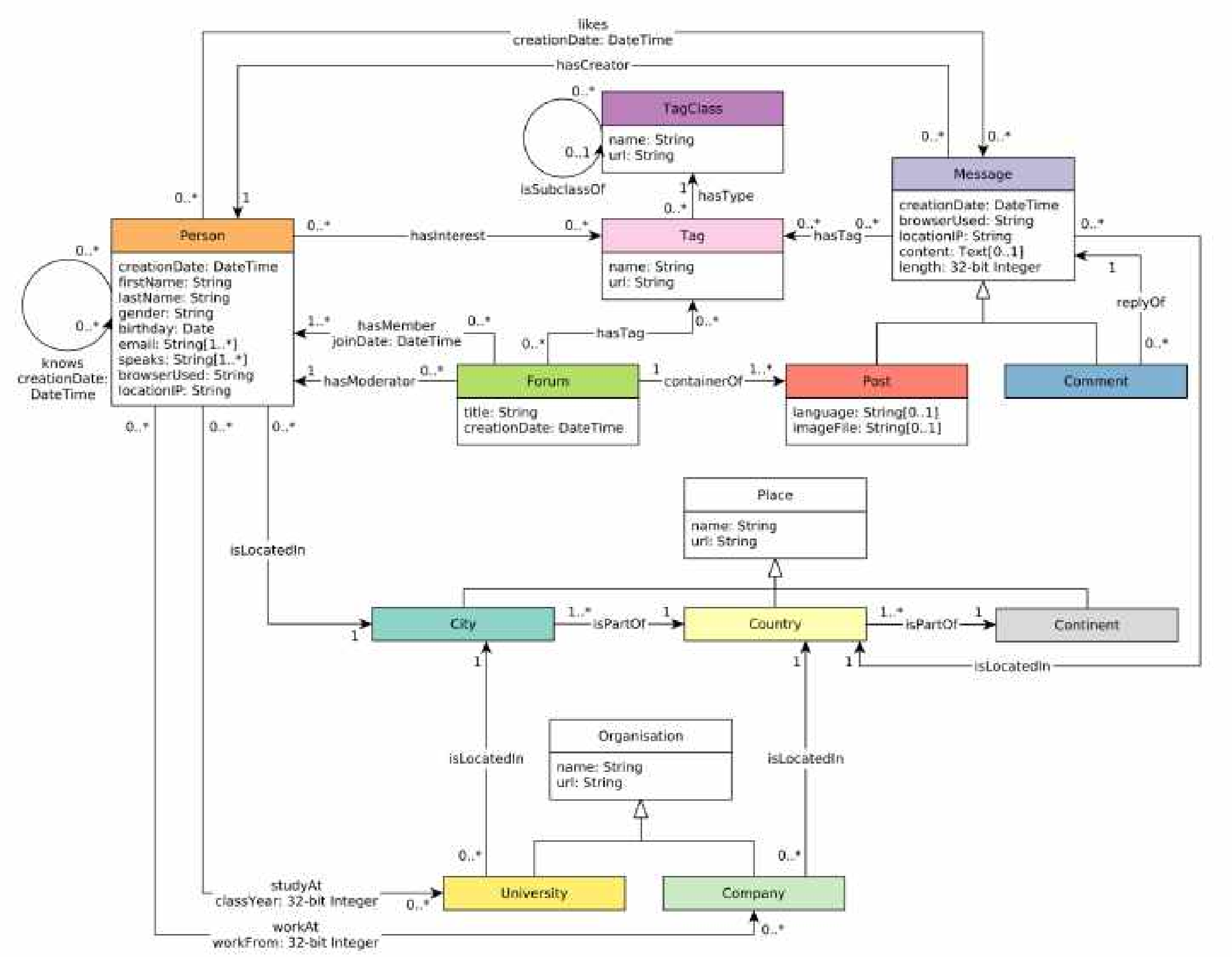


Figure 1. LDBC SNB Schema

Figure 1 shows the schema (from the LDBC SNB specification ↗). It models the activities and relationships of social forum participants. For example, a forum Member can publish Posts on a Forum, and other Members of the Forum can make a Comment on the Post or on someone else's Comment. A Person's home location is a hierarchy (Continent>Country>City), and a person can be affiliated with a University or a Company. A Tag can be used to classify a Forum, a Message, or a Person's interests. Tags can further be classified by TagClass. The relationships between entities are modeled as directed edges, except person KNOWS person is modeled as an undirected edge. For example, Person connects to Tag by the hasInterest edge. Forum connects to Person by two different edges, hasMember and hasModerator.

LDBC SNB schema uses **inheritance** to model certain relationships:

- **Message** is the superclass of Post and Comment.
- **Place** is the superclass of City, Country, and Continent.
- **Organization** is the superclass of University and Company.

We do not use the superclasses in our graph model. When there is an edge type connecting an entity to a superclass, we instead create an edge type from the entity to each of the subclasses of the superclass. For example, Message has an *isLocatedIn* relationship to Country. Since Message has two subclasses, Post and Comment, both connected with Country by the *isLocatedIn* relationship, we create an edge type IS_LOCATED_IN, connecting both vertex type pairs using the **compound edge type** available in TigerGraph 3.0.

- **CREATE DIRECTED EDGE** IS_LOCATED_IN (**FROM** Comment, **TO** Country
  | **FROM** Post, **TO** Country)

This new DDL syntax allows a general edge type to be defined over multiple vertex pairs. For example, there are many relationships in the LDBC schema all called *isLocatedIn* which connect something to a geographical entity. We can model them all as a single edge type IS_LOCATED_IN. The result is a more succinct graph model and less GSQL code when expressing pattern matching queries.

- **CREATE DIRECTED EDGE** IS_LOCATED_IN (**FROM** Comment, **TO** Country | **FROM** Post, **TO** Country | **FROM** Company, **TO** Country | **FROM** Person, **TO** City | **FROM** University, **TO** City) **WITH REVERSE_EDGE**="IS_LOCATED_IN_REVERSE"

Schema Naming Conventions

The folowing nameing conventions are followed in the DDL language:

- Vertex types
    - For each entity in Figure 1 (the rectangular boxes), we create a vertex type with the entity's name in UpperCamelCase.
    - Examples
        - **Person** is a person who participates in a forum.
        - **Forum** is a place where persons discuss topics.
        - **City, Country**, and **Continent** are geographic locations of other entities.
        - **Company** and **University** are organizations with which a person can be affiliated.
        - **Comment** and **Post** are the interaction messages created by persons in a forum.
        - **Tag** is a topic or a concept.
        - **TagClass** is a class or a category. TagClass can form a hierarchy of tags.
- Edge types
    - For each relationship type, we create an edge type with the relationship name **(all capitalized and words are separated by an underscore).**
    - When multiple relationships share the same semantics in Figure 1, we merge them into a single compound edge type. For example:
    - **Examples**
        - **CONTAINER_OF**: Forum is the container of posts.
        - **HAS_INTERESTS**: Person has interest in tag(s).
        - **IS_SUBCLASS_OF:** Tag is a subclass of another Tag.
        - **IS_LOCATED_IN:** Comment or Post is located in Country, Company is located in Country, Person is located in City, and University is located in City.

GSQL Schema DDL

Two GSQL scripts for defining the LDBC-SNB schema are shown below. Choose the one that serve your needs. They are not equivalent if you have different organizations using the same graph database instance.

Method 1. Bottom-up DDL

In this method, global vertex/edge type containers are created first. Next, graphs are created to group them. In other words, the global vertex/edge type containers can be shared across graphs.

#LDBC-SNB schema

```
//clear the current catalog.
// It may take a while since it restarts the subsystem services.
DROP ALL

//create vertex types

## Post and Comment
CREATE VERTEX Comment (PRIMARY_ID id UINT, creationDate DATETIME, locatio
    browserUsed STRING, content STRING, length UINT) WITH primary_id_as_a
CREATE VERTEX Post (PRIMARY_ID id UINT, imageFile STRING, creationDate DA
    locationIP STRING, browserUsed STRING, lang STRING, content STRING,
    length UINT) WITH primary_id_as_attribute="TRUE"
## organisation
CREATE VERTEX Company (PRIMARY_ID id UINT, name STRING, url STRING) WITH
CREATE VERTEX University (PRIMARY_ID id UINT, name STRING, url STRING) WI
## place
CREATE VERTEX City (PRIMARY_ID id UINT, name STRING, url STRING) WITH pri
CREATE VERTEX Country (PRIMARY_ID id UINT, name STRING, url STRING) WITH
CREATE VERTEX Continent (PRIMARY_ID id UINT, name STRING, url STRING) WIT
## etc
CREATE VERTEX Forum (PRIMARY_ID id UINT, title STRING, creationDate DATET
CREATE VERTEX Person (PRIMARY_ID id UINT, firstName STRING, lastName STRI
   creationDate DATETIME, locationIP STRING, browserUsed STRING, speaks s
   WITH primary_id_as_attribute="TRUE"
CREATE VERTEX Tag (PRIMARY_ID id UINT, name STRING, url STRING) WITH prim
CREATE VERTEX TagClass (PRIMARY_ID id UINT, name STRING, url STRING) WITH

// create edge types
CREATE DIRECTED EDGE CONTAINER_OF (FROM Forum, TO Post) WITH REVERSE_EDGE
CREATE DIRECTED EDGE HAS_CREATOR (FROM Comment|Post, TO Person) WITH REVE
CREATE DIRECTED EDGE HAS_INTEREST (FROM Person, TO Tag) WITH REVERSE_EDGE
CREATE DIRECTED EDGE HAS_MEMBER (FROM Forum, TO Person, joinDate DATETIME
CREATE DIRECTED EDGE HAS_MODERATOR (FROM Forum, TO Person) WITH REVERSE_E
CREATE DIRECTED EDGE HAS_TAG (FROM Comment|Post|Forum, TO Tag) WITH REVER
CREATE DIRECTED EDGE HAS_TYPE (FROM Tag, TO TagClass) WITH REVERSE_EDGE="
CREATE DIRECTED EDGE IS_LOCATED_IN (FROM Comment, TO Country
                                  | FROM Post, TO Country
                                  | FROM Company, TO Country
                                  | FROM Person, TO City
                                  | FROM University, TO City) WITH REVERS
CREATE DIRECTED EDGE IS_PART_OF (FROM City, TO Country
                               | FROM Country, TO Continent) WITH REVERSE
CREATE DIRECTED EDGE IS_SUBCLASS_OF (FROM TagClass, TO TagClass) WITH REV
CREATE UNDIRECTED EDGE KNOWS (FROM Person, TO Person, creationDate DATETI
CREATE DIRECTED EDGE LIKES (FROM Person, TO Comment|Post, creationDate DA
CREATE DIRECTED EDGE REPLY_OF (FROM Comment, TO Comment|Post) WITH REVERS
CREATE DIRECTED EDGE STUDY_AT (FROM Person, TO University, classYear INT)
CREATE DIRECTED EDGE WORK_AT (FROM Person, TO Company, workFrom INT) WITH
```

```
//create graph type
CREATE GRAPH ldbc_snb (*)
```

Method 2. Top-down DDL

In this method, an empty graph is created first. Next, local vertex/edge type containers are added to the empty graph via a schema change job. The vertex/egde type containers added this way will be private to the graph, no other graph can see them.

```
#LDBC-SNB schema
```

```
//clear the current catalog.
// It may take a while since it restarts the subsystem services.
DROP ALL

# 1. Create graph
CREATE GRAPH ldbc_snb ()

# 2. Create schema_change job to include all vertex/edge types
CREATE SCHEMA_CHANGE JOB change_schema_of_ldbc  FOR GRAPH ldbc_snb {

  ## Post and Comment
  ADD VERTEX Comment (PRIMARY_ID id UINT, creationDate DATETIME, location
    browserUsed STRING, content STRING, length UINT) WITH primary_id_as_a

  ADD VERTEX Post (PRIMARY_ID id UINT, imageFile STRING, creationDate DAT
    locationIP STRING, browserUsed STRING, lang STRING, content STRING,
    length UINT) WITH primary_id_as_attribute="TRUE";
  ## organisation
  ADD VERTEX Company (PRIMARY_ID id UINT, name STRING, url STRING) WITH p
  ADD VERTEX University (PRIMARY_ID id UINT, name STRING, url STRING) WIT
  ## place
  ADD VERTEX City (PRIMARY_ID id UINT, name STRING, url STRING) WITH prim
  ADD VERTEX Country (PRIMARY_ID id UINT, name STRING, url STRING) WITH p
  ADD VERTEX Continent (PRIMARY_ID id UINT, name STRING, url STRING) WITH
  ## etc
  ADD  VERTEX Forum (PRIMARY_ID id UINT, title STRING, creationDate DATET
  ADD  VERTEX Person (PRIMARY_ID id UINT, firstName STRING, lastName STRI
   creationDate DATETIME, locationIP STRING, browserUsed STRING, speaks s
   WITH primary_id_as_attribute="TRUE";
  ADD VERTEX Tag (PRIMARY_ID id UINT, name STRING, url STRING) WITH prima
  ADD VERTEX TagClass (PRIMARY_ID id UINT, name STRING, url STRING) WITH

  // create edge types
  ADD DIRECTED EDGE CONTAINER_OF (FROM Forum, TO Post) WITH REVERSE_EDGE=
  ADD  DIRECTED EDGE HAS_CREATOR (FROM Comment|Post, TO Person) WITH REVE
  ADD  DIRECTED EDGE HAS_INTEREST (FROM Person, TO Tag) WITH REVERSE_EDGE
  ADD DIRECTED EDGE HAS_MEMBER (FROM Forum, TO Person, joinDate DATETIME)
  ADD DIRECTED EDGE HAS_MODERATOR (FROM Forum, TO Person) WITH REVERSE_ED
  ADD DIRECTED EDGE HAS_TAG (FROM Comment|Post|Forum, TO Tag) WITH REVERS
  ADD DIRECTED EDGE HAS_TYPE (FROM Tag, TO TagClass) WITH REVERSE_EDGE="H
  ADD  DIRECTED EDGE IS_LOCATED_IN (FROM Comment, TO Country
                                  | FROM Post, TO Country
                                  | FROM Company, TO Country
                                  | FROM Person, TO City
                                  | FROM University, TO City) WITH REVERS
  ADD DIRECTED EDGE IS_PART_OF (FROM City, TO Country
                               | FROM Country, TO Continent) WITH REVERSE
  ADD DIRECTED EDGE IS_SUBCLASS_OF (FROM TagClass, TO TagClass) WITH REVE
```

```
  ADD UNDIRECTED EDGE KNOWS (FROM Person, TO Person, creationDate DATETIM
  ADD DIRECTED EDGE LIKES (FROM Person, TO Comment|Post, creationDate DAT
  ADD DIRECTED EDGE REPLY_OF (FROM Comment, TO Comment|Post) WITH REVERSE
  ADD DIRECTED EDGE STUDY_AT (FROM Person, TO University, classYear INT)
  ADD DIRECTED EDGE WORK_AT (FROM Person, TO Company, workFrom INT) WITH
}

# 3. Run schema_change job
RUN SCHEMA_CHANGE JOB change_schema_of_ldbc

# 4. Drop schema_change job
DROP JOB change_schema_of_ldbc
```

Load Data

Define the Loading Job

Below, we use the GSQL loading language to define a loading job script, which encodes all the mappings from the source CSV file (generated by the LDBC SNB benchmark data generator) to our schema.

The script to load the LDBC-SNB data is below.

```
# GSQL script for loading LDBC-SNB data
```

```
USE GRAPH ldbc_snb
CREATE LOADING JOB load_ldbc_snb FOR GRAPH ldbc_snb {
  // define vertex
  DEFINE FILENAME v_comment_file;
  DEFINE FILENAME v_post_file;
  DEFINE FILENAME v_organisation_file;
  DEFINE FILENAME v_place_file;
  DEFINE FILENAME v_forum_file;
  DEFINE FILENAME v_person_file;
  DEFINE FILENAME v_tag_file;
  DEFINE FILENAME v_tagclass_file;

  // define edge
  DEFINE FILENAME forum_containerOf_post_file;
  DEFINE FILENAME comment_hasCreator_person_file;
  DEFINE FILENAME post_hasCreator_person_file;
  DEFINE FILENAME person_hasInterest_tag_file;
  DEFINE FILENAME forum_hasMember_person_file;
  DEFINE FILENAME forum_hasModerator_person_file;
  DEFINE FILENAME comment_hasTag_tag_file;
  DEFINE FILENAME post_hasTag_tag_file;
  DEFINE FILENAME forum_hasTag_tag_file;
  DEFINE FILENAME tag_hasType_tagclass_file;
  DEFINE FILENAME organisation_isLocatedIn_place_file;
  DEFINE FILENAME comment_isLocatedIn_place_file;
  DEFINE FILENAME post_isLocatedIn_place_file;
  DEFINE FILENAME person_isLocatedIn_place_file;
  DEFINE FILENAME place_isPartOf_place_file;
  DEFINE FILENAME tagclass_isSubclassOf_tagclass_file;
  DEFINE FILENAME person_knows_person_file;
  DEFINE FILENAME person_likes_comment_file;
  DEFINE FILENAME person_likes_post_file;
  DEFINE FILENAME comment_replyOf_comment_file;
  DEFINE FILENAME comment_replyOf_post_file;
  DEFINE FILENAME person_studyAt_organisation_file;
  DEFINE FILENAME person_workAt_organisation_file;

  // load vertex
  LOAD v_comment_file
    TO VERTEX Comment VALUES ($0, $1, $2, $3, $4, $5) USING header="true"
  LOAD v_post_file
    TO VERTEX Post VALUES ($0, $1, $2, $3, $4, $5, $6, $7) USING header="
  LOAD v_organisation_file
    TO VERTEX Company VALUES ($0, $2, $3) WHERE $1=="company",
    TO VERTEX University VALUES ($0, $2, $3) WHERE $1=="university" USING
  LOAD v_place_file
    TO VERTEX City VALUES ($0, $1, $2) WHERE $3=="city",
    TO VERTEX Country VALUES ($0, $1, $2) WHERE $3=="country",
```

```
    TO VERTEX Continent VALUES ($0, $1, $2) WHERE $3=="continent" USING h
  LOAD v_forum_file
    TO VERTEX Forum VALUES ($0, $1, $2) USING header="true", separator="|
  LOAD v_person_file
    TO VERTEX Person VALUES ($0, $1, $2, $3, $4, $5, $6, $7, SPLIT($8,";"
  LOAD v_tag_file
    TO VERTEX Tag VALUES ($0, $1, $2) USING header="true", separator="|";
  LOAD v_tagclass_file
    TO VERTEX TagClass VALUES ($0, $1, $2) USING header="true", separator

  // load edge
  LOAD forum_containerOf_post_file
    TO EDGE CONTAINER_OF VALUES ($0, $1) USING header="true", separator="
  LOAD comment_hasCreator_person_file
    TO EDGE HAS_CREATOR VALUES ($0 Comment, $1) USING header="true", sepa
  LOAD post_hasCreator_person_file
    TO EDGE HAS_CREATOR VALUES ($0 Post, $1) USING header="true", separat
  LOAD person_hasInterest_tag_file
    TO EDGE HAS_INTEREST VALUES ($0, $1) USING header="true", separator="
  LOAD forum_hasMember_person_file
    TO EDGE HAS_MEMBER VALUES ($0, $1, $2) USING header="true", separator
  LOAD forum_hasModerator_person_file
```

Prepare The Raw Data

Linux Bash

```
wget https://s3-us-west-1.amazonaws.com/tigergraph-benchmark-dataset/LDBC
```

```
    TO EDGE HAS_MODERATOR VALUES ($0, $1) USING header="true", separator=
  LOAD post_hasTag_tag_file
    TO EDGE HAS_TAG VALUES ($0 Post, $1) USING header="true", separator="
  LOAD forum_hasTag_tag_file
    TO EDGE HAS_TAG VALUES ($0 Forum, $1) USING header="true", separator=
  LOAD tag_hasType_tagclass_file
    TO EDGE HAS_TYPE VALUES ($0, $1) USING header="true", separator="|";
```

Linux Bash

```
tar -xzf  ldbc_snb_data-sf1.tar.gz
```

```
  LOAD comment_isLocatedIn_place_file
    TO EDGE IS_LOCATED_IN VALUES ($0 Comment, $1 Country) USING header="t
  LOAD post_isLocatedIn_place_file
    TO EDGE IS_LOCATED_IN VALUES ($0 Post, $1 Country) USING header="true
  LOAD person_isLocatedIn_place_file
    TO EDGE IS_LOCATED_IN VALUES ($0 Person, $1 City) USING header="true"
  LOAD place_isPartOf_place_file
    TO EDGE IS_PART_OF VALUES ($0 Country, $1 Continent) WHERE to_int($0)
    TO EDGE IS_PART_OF VALUES ($0 City, $1 Country) WHERE to_int($0) > 11
  LOAD tagclass_isSubclassOf_tagclass_file
    TO EDGE IS_SUBCLASS_OF VALUES ($0, $1) USING header="true", separator
  LOAD person_knows_person_file
    TO EDGE KNOWS VALUES ($0, $1, $2) USING header="true", separator="|";
  LOAD person_likes_comment_file
```

```
    TO EDGE LIKES VALUES ($0, $1 Comment, $2) USING header="true", separa
  LOAD person_likes_post_file
    TO EDGE LIKES VALUES ($0, $1 Post, $2) USING header="true", separator
  LOAD comment_replyOf_comment_file
    TO EDGE REPLY_OF VALUES ($0, $1 Comment) USING header="true", separat
  LOAD comment_replyOf_post_file
    TO EDGE REPLY_OF VALUES ($0, $1 Post) USING header="true", separator=
  LOAD person_studyAt_organisation_file
    TO EDGE STUDY_AT VALUES ($0, $1, $2) USING header="true", separator="
  LOAD person_workAt_organisation_file
    TO EDGE WORK_AT VALUES ($0, $1, $2) USING header="true", separator="|
}
```

Run The Loading Job

command to specify its location. Then, start your TigerGraph services if needed. Finally, run the **setup_schema.gsql** script to create your LDBC Social Network graph.

Linux Bash

```
#change the directory to your raw file directory
export LDBC_SNB_DATA_DIR=/home/tigergraph/ldbc_snb_data/social_network/

#start all TigerGraph services
gadmin start all

#setup schema and loading job
gsql setup_schema.gsql
```

Download the loading job script ↗ and invoke it on the command line. #

Linux Bash

```
./load_data.sh
```

Sample Loading Progress Output

```
tigergraph/gsql_102$ ./load_data.sh
[Tip: Use "CTRL + C" to stop displaying the loading status update, then u
[Tip: Manage loading jobs with "ABORT/RESUME LOADING JOB jobid"]
Starting the following job, i.e.
  JobName: load_ldbc_snb, jobid: ldbc_snb.load_ldbc_snb.file.m1.155805315
  Loading log: '/mnt/data/tigergraph/logs/restpp/restpp_loader_logs/ldbc_

Job "ldbc_snb.load_ldbc_snb.file.m1.1558053156447" loading status
[FINISHED] m1 ( Finished: 31 / Total: 31 )
  [LOADED]
  +-----------------------------------------------------------------------
  |
  |                        /mnt/data/download/ldbc_snb_data/social_network
  |     /mnt/data/download/ldbc_snb_data/social_network/comment_hasCreato
  |            /mnt/data/download/ldbc_snb_data/social_network/comment_ha
  |     /mnt/data/download/ldbc_snb_data/social_network/comment_isLocated
  |       /mnt/data/download/ldbc_snb_data/social_network/comment_replyOf
  |          /mnt/data/download/ldbc_snb_data/social_network/comment_repl
  |                          /mnt/data/download/ldbc_snb_data/social_netwo
  |        /mnt/data/download/ldbc_snb_data/social_network/forum_containe
  |        /mnt/data/download/ldbc_snb_data/social_network/forum_hasMembe
  |     /mnt/data/download/ldbc_snb_data/social_network/forum_hasModerato
  |              /mnt/data/download/ldbc_snb_data/social_network/forum_ha
  |                  /mnt/data/download/ldbc_snb_data/social_network/orga
  |/mnt/data/download/ldbc_snb_data/social_network/organisation_isLocated
  |                        /mnt/data/download/ldbc_snb_data/social_networ
  |        /mnt/data/download/ldbc_snb_data/social_network/person_hasInte
  |      /mnt/data/download/ldbc_snb_data/social_network/person_isLocated
  |           /mnt/data/download/ldbc_snb_data/social_network/person_know
  |          /mnt/data/download/ldbc_snb_data/social_network/person_likes
  |             /mnt/data/download/ldbc_snb_data/social_network/person_li
  |   /mnt/data/download/ldbc_snb_data/social_network/person_studyAt_orga
  |    /mnt/data/download/ldbc_snb_data/social_network/person_workAt_orga
  |                          /mnt/data/download/ldbc_snb_data/social_netwo
  |          /mnt/data/download/ldbc_snb_data/social_network/place_isPart
  |                           /mnt/data/download/ldbc_snb_data/social_netw
  |        /mnt/data/download/ldbc_snb_data/social_network/post_hasCreato
  |               /mnt/data/download/ldbc_snb_data/social_network/post_ha
  |        /mnt/data/download/ldbc_snb_data/social_network/post_isLocated
  |                            /mnt/data/download/ldbc_snb_data/social_net
  |          /mnt/data/download/ldbc_snb_data/social_network/tag_hasType_
  |                      /mnt/data/download/ldbc_snb_data/social_network/
  |/mnt/data/download/ldbc_snb_data/social_network/tagclass_isSubclassOf_
  +-----------------------------------------------------------------------
```

After loading, you can check the graph's size using built-in REST endpoint calls.

Below we call two functions, stat_vertex_number and stat_edge_number to return the cardinality of each vertex and edge type.

Linux shell

```
curl -X POST 'http://localhost:9000/builtins/ldbc_snb' -d  '{"function":"
curl -X POST 'http://localhost:9000/builtins/ldbc_snb' -d  '{"function":"
```

One-hop patterns

Introduction

Pattern matching by nature is declarative. It enables users to focus on specifying what they want from a query without worrying about the underlying query processing.

A pattern usually appears in the FROM clause, the most fundamental part of the query structure. The pattern specifies sets of vertex types and how they are connected by edge types. A pattern can be refined further with conditions in the WHERE clause. In this tutorial, we'll start with simple one-hop path patterns, and then extend it multi-hop patterns and finally multiple-path patterns.

Currently, pattern matching may only be used in read-only queries. DML support will be added in the near future release.

Pattern matching queries support nested queries

1-Hop Pattern

The easiest way to understand patterns is to start with a simple 1-Hop pattern. Even a single hop has several options. After we've tackled single hops, then we'll see how to add repetition to make variable length patterns and how to connect single hops to form bigger patterns.

In classic GSQL queries, described in GSQL 101, we used the punctuation `-( )->` in the FROM clause to indicate a 1-hop query, where the arrow specifies the vertex flow from left to right, and `( )` encloses the edge types.

```
Person:p -(LIKES:e)-> Message:m          /* Classic GSQL example */
```

In pattern matching, we use the punctuation `-( )-` to denote a 1-hop pattern, where the edge type(s) is enclosed in the parentheses `()` and the hyphens `-` symbolize connection without specifying direction. Instead, directionality is explicitly stated for *each* edge type.

- For an undirected edge E, no added decoration: `E`
- For a directed edge E from left to right, use a suffix: `E>`
- For a directed edge E from right to left, use a prefix: `<E`

For example, in the LDBC SNB schema, there are two directed relationships between Person and Message: person *LIKES* message, and message *HAS_CREATOR* person. Despite the fact that these relationships are in opposite directions, we can include both of them in the same pattern very concisely using an alternation separator `|` :

```
Person:p -((LIKES>|<HAS_CREATOR):e)- Message:m         /* Pattern example
```

Edge Type Wildcards

The underscore `_` is a wildcard meaning *any edge type.* Arrowheads are still used to indicate direction, e.g., `_>` or `<_` or `_`
The empty parentheses `()` means any edge, directed or undirected.

Vertex Type Wildcards and Path Symmetry

Prior to TigerGraph 3.0, the source (leftmost) vertex set needed to be defined as an explicit set, prior to the SELECT statement. A typical approach is shown here.

Seed set definition required in classic GSQL

```
CREATE QUERY seedSet() FOR GRAPH ldbc_snb SYNTAX v1 {
    Source = {Person}; // Seed set
    SELECT t FROM Source:s -(IS_LOCATED_IN:e)- :t;
    PRINT t;
}
```

Beginning in TigerGraph 3.0, SYNTAX V2 treats the source vertex set the same as the target vertex set. That is, the source or the target vertex set may be:

- a vertex type
  `SELECT t FROM Person:s -(IS_LOCATED_IN>:e) - City:t`
- an alternation of vertex types
  `SELECT t FROM (Post|Comment):s -(IS_LOCATED_IN>:e) - Country:t`
- omitted, with only an alias, meaning any vertex type
  `SELECT s FROM :s -(IS_LOCATED_IN>:e) - Country:t`
- omitted, without an alias, meaning any vertex type
  `SELECT t FROM -(IS_LOCATED_IN>:e) - Country:t`

> Performance may be better when types are explicitly provided.

Examples of 1-Hop Patterns

1. `FROM X:x - (E1:e1) - Y:y`
   - E1 is an undirected edge, x and y bind to the end points of E1, and e1 is the alias of E1.
2. `FROM X:x - (E2>:e2) - Y:y`
   - Right directed edge x binds to the source of E2; y binds to the target of E2.
3. `FROM X:x - (<E3:e3) - Y:y`
   - Left directed edge; y binds to the source of E3; x binds to the target of E3.
4. `FROM X:x - (_:e) - Y:y`
   - Any undirected edge between a member of X and a member of Y.
5. `FROM X:x - (_>:e) - Y:y`
   - Any right directed edge with source in X and target in Y.
6. `FROM X:x - (<_:e) - Y:y`
   - Any left directed edge with source in Y and target in X.
7. `FROM X:x - ((<_|_):e) - Y:y`
   - Any left directed or any undirected; "|" means OR, and parentheses enclose the group of edge descriptors; e is the alias for the edge pattern (<_|_).

8. `FROM X:x - ((E1|E2>|<E3):e) - Y:y`
    - Any one of the three edge patterns.
9. `FROM X:x - () - Y:y`
    - any edge (directed or undirected)
    - Same as `(<_|_>|_)`

Entering Pattern Match Syntax Mode

To use pattern matching, you need to either set a session parameter or specify it in the query. There are currently two syntax versions for queries:

- "v1" is the classic syntax, traversing one hop per SELECT statement. This is the default mode.
- "v2" enhances the v1 syntax with pattern matching.

syntax_version Session Parameter

You can use the SET command to assign a value to the *syntax_version* session parameter: v1 for classic syntax; v2 for pattern matching. If the parameter is never set, the classic v1 syntax is enabled. Once set, the selection remains valid for the duration of the GSQL client session, or until it is changed with another SET command.

GSQL: Set Syntax Version By A Session Parameter

```
SET syntax_version="v2"
```

Query-Level SYNTAX option

You can also select the syntax by using the SYNTAX clause in the CREATE QUERY statement: v1 for classic syntax (default); v2 for pattern matching. The query-level SYNTAX option overrides the syntax_version session parameter.

GSQL: Set Syntax Version By Specifying The Version After Graph Name In The Query

```
CREATE QUERY test10 (string str ) FOR GRAPH ldbc_snb SYNTAX v2
{
  ...
}
```

Running Anonymous Queries Without Installing

In this tutorial, we will use Interpreted Mode for GSQL, introduced in TigerGraph 2.4. Interpreted mode lets us skip the INSTALL step, and even run a query as soon as we create it, to offer a more interactive experience. These one-step interpreted queries are unnamed (anonymous) and parameterless, just like SQL.

To run an anonymous query, replace the keyword CREATE with INTERPRET. Remember, no parameters:

```
INTERPRET QUERY () FOR GRAPH graph_name SYNTAX v2 { <query body> }
```

Recommendation: Increase the query timeout threshold.

Interpreted queries may run slower than installed queries, so we recommend increasing the query timeout threshold:

GSQL: Set Longer Timeout

```
# set query time out to 1 minutes
# 1 unit is 1 millisecond
SET query_timeout = 60000
```

Examples of 1-Hop Fixed Length Query

Example 1. Find persons who know the person named "Viktor Akhiezer" and return the top 3 oldest such persons.

Example 1. Left Directed Edge Pattern

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {
   #1-hop pattern.
   friends = SELECT p
             FROM Person:s -(KNOWS:e)- Person:p
             WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
             ORDER BY p.birthday ASC
             LIMIT 3;

    PRINT  friends[friends.firstName, friends.lastName, friends.birthday]
}
```

Syntax Enhancement in TigerGraph 3.0+

- In Example 1, "FOR GRAPH ldbc_snb" is not used after () in the query signature. It's an optional component in 3.0+ when "USE GRAPH graphName" is used; Or from the command line, "gsql -g graphName " precedes any query invocation.
- In the FROM clause, we directly use vertex type Person as the starting vertex set. This syntax enhancement is available in syntax V2 only.

You can copy the above GSQL script to a file named example1.gsql and invoke this script file in Linux.

Linux Bash

```
gsql example1.gsql
```

Output of Example 1

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "developer",
    "api": "v2"
  },
  "results": [{"friends": [
    {
      "v_id": "10995116279461",
      "attributes": {
        "friends.birthday": "1980-05-13 00:00:00",
        "friends.lastName": "Cajes",
        "friends.firstName": "Gregorio"
      },
      "v_type": "Person"
    },
    {
      "v_id": "4398046517846",
      "attributes": {
        "friends.birthday": "1980-04-24 00:00:00",
        "friends.lastName": "Glosca",
        "friends.firstName": "Abdul-Malik"
      },
      "v_type": "Person"
    },
    {
      "v_id": "6597069776731",
      "attributes": {
        "friends.birthday": "1981-02-25 00:00:00",
        "friends.lastName": "Carlsson",
        "friends.firstName": "Sven"
      },
      "v_type": "Person"
    }
  ]}]
}
```


Example 2. Find the total number of comments and total number of posts liked by Viktor. A Person can reach Comments or Posts via a directed edge LIKES.

Example 2. Right-directed Edge Pattern

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {
   SumAccum<int> @commentCnt= 0;
   SumAccum<int> @postCnt= 0;

   #1-hop pattern.
   Result = SELECT s
            FROM Person:s -(LIKES>)- :tgt
            WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
            ACCUM CASE WHEN tgt.type == "Comment" THEN
                           s.@commentCnt += 1
                       WHEN tgt.type == "Post" THEN
                           s.@postCnt += 1
                   END;

    PRINT  Result[Result.@commentCnt, Result.@postCnt];
}
```

You can copy the above GSQL script to a file named example2.gsql, and invoke this script file in Linux.

Linux Bash

```
gsql example2.gsql
```

Output of Example 2.

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "28587302323577",
    "attributes": {
      "Result.@commentCnt": 108,
      "Result.@postCnt": 51
    },
    "v_type": "Person"
  }]}]
}
```


Example 3. Solve the same problem as in Example 2, but use a left-directed edge pattern.

Note below (line 8) that the source vertex set are now Comment and Post, and the target is Person.

Example 3. Left-directed Edge Pattern

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{
   SumAccum<int> @commentCnt= 0;
   SumAccum<int> @postCnt= 0;

   Result = SELECT tgt
            FROM Person:tgt -(<LIKES_REVERSE)- (Comment|Post):src
            WHERE tgt.firstName == "Viktor" AND tgt.lastName == "Akhiezer
            ACCUM CASE WHEN src.type == "Comment" THEN
                               tgt.@commentCnt += 1
                           WHEN src.type == "Post" THEN
                               tgt.@postCnt += 1
                       END;

  PRINT Result[Result.@commentCnt, Result.@postCnt];
}
```

You can copy the above GSQL script to a file named example3.gsql, and invoke this script file in linux command line. The output should be the same as in Example 2.

Example 4. Find Viktor Akhiezer's total number of related comments and total number of related posts. That is, a comment or post is either created by Viktor or is liked by Viktor. Note that the HAS_CREATOR edge type starts from Comment|Post, and the LIKES edge type starts from Person.

Example 4. Disjunctive 1-hop edge pattern.

```
USE GRAPH ldbc_snb
set query_timeout=60000

INTERPRET QUERY () SYNTAX v2{
  SumAccum<int> @commentCnt= 0;
  SumAccum<int> @postCnt= 0;

  Result = SELECT tgt
           FROM Person:tgt -(<HAS_CREATOR|LIKES>)- (Comment|Post):src
           WHERE tgt.firstName == "Viktor" AND tgt.lastName == "Akhiezer"
           ACCUM CASE WHEN src.type == "Comment" THEN
                          tgt.@commentCnt += 1
                      WHEN src.type == "Post" THEN
                          tgt.@postCnt += 1
                 END;

  PRINT Result[Result.@commentCnt, Result.@postCnt];
}
```

You can copy the above GSQL script to a file named example4.gsql, and invoke this script file in Linux:

Linux Bash

```
gsql example4.gsql
```

Output of Example 4.

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "28587302323577",
    "attributes": {
      "Result.@commentCnt": 152,
      "Result.@postCnt": 96
    },
    "v_type": "Person"
  }]}]
}
```


Example 5. Find the total number of comments or posts related to "Viktor Akhiezer". This time, we count them together and, we use wildcard "_" to represent the two types of edges: HAS_CREATOR and LIKES_REVERSE. Both are following the same direction.

Example 5. Disjunctive 1-hop edge pattern.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{
  SumAccum<int> @@cnt= 0;

  Result = SELECT tgt
           FROM Person:tgt -(<_)- (Comment|Post):src
           WHERE tgt.firstName == "Viktor" AND tgt.lastName == "Akhiezer"
           ACCUM  @@cnt += 1;

  PRINT @@cnt;
}
```

You can copy the above GSQL script to a file named example5.gsql, and invoke this script file in Linux:

Linux Bash

```
gsql example5.gsql
```

Output of Example 5.

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 248}]
}
```

Repeating a 1-Hop Pattern

A common pattern is the two-step "Friend of a Friend". Or, how many entities might receive a message if it is passed up to three times? Do you have any known change of connections to a celebrity?

GSQL pattern matching makes it easy to express such variable-length patterns which repeat a single-hop. Everything else stays the same as introduced in the previous section, except we append an asterisk (or *Kleene star* for you regular expressionists) and an optional *min..max* range to an edge pattern.

- (E*) means edge type E repeats any number of times (including zero!)
- (E*1..3) means edge type E occurs one to three times.

Below are more illustrative examples:

- **1-hop star pattern — repetition of an edge pattern 0 or more times**
  1. `FROM X:x - (E*) - Y:y`
  2. `FROM X:x - (F>*) - Y:y`
  3. `FROM X:x - (<G*) - Y:y`
  4. `FROM X:x - (_*) - Y:y`
     - Any undirected edge can be chosen at each repetition.
  5. `FROM X:x - (_>*) - Y:y`
     - Any right-directed edge can be chosen at each repetition.
  6. `FROM X:x - (<_*) - Y:y`
     - Any left-directed edge can be chosen at each repetition.
  7. `FROM X:x - ((E|F>|<G)*) - Y:y`
     - Either E, F> or <G can be chosen at each repetition.
- **1-hop star pattern with bounds**
  1. `FROM X:x - (E*2..) - Y:y`
     - Lower bounds only. There is a chain of at least 2 E edges.
  2. `FROM X:x - (F>*..3) - Y:y`
     - Upper bounds only. There is a chain of between 0 and 3 F edges.

3. `FROM X:x - (<G*3..5) - Y:y`
   - Both Lower and Upper bounds. There is a chain of 3 to 5 G edges.
4. `FROM X:x - ((E|F>|<G)*3) - Y:y`
   - Exact bound. There is a chain of exactly 3 edges, where each edge is either E, F>, or <G.

Remarks

- **No alias allowed for edge with Kleene star**
  An edge alias may not be used when a Kleene star is used. The reason is that when there are a variable number of edges, we cannot associate or bind the alias to a specific edge in the pattern.
- **Shortest path semantics**
  When an edge is repeated with a Kleene star, only the shortest matching occurrences are selected. See the example below:

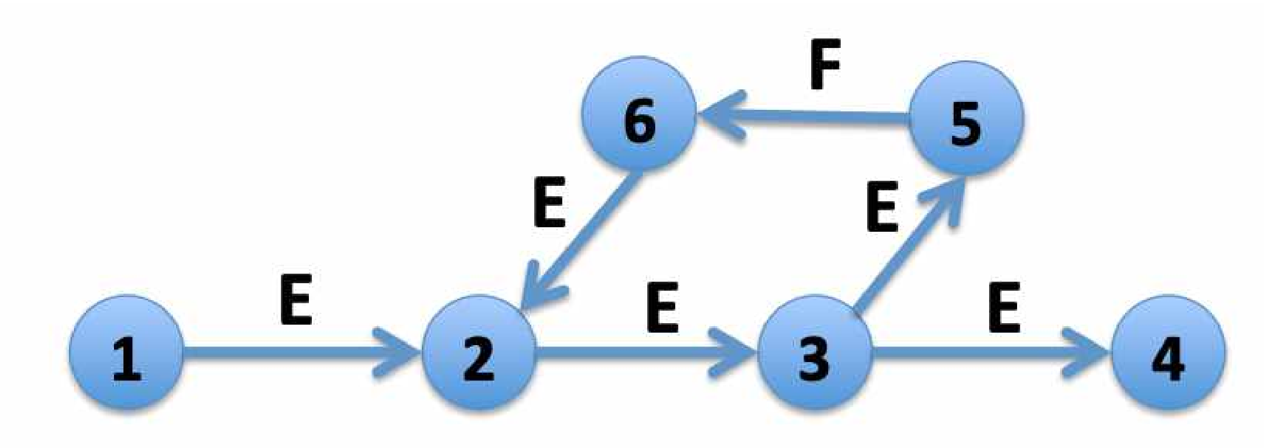


Figure 2 Shortest Path Illustration.

In Figure 2, or Pattern `1 - (E>*) - 4`, any of the following paths reach 4 from 1.

- 1→2→3→4
- 1→2→3→5→6→2→3→4
- any path that goes through the cycle 2→3→5→6→2 two or more times and jumps out at 3.

The first path is shorter than the rest; it is considered the only match.

Examples of Variable Hop Queries

In this tutorial, we will use the Interpreted Mode for GSQL, introduced in TigerGraph 2.4. Interpreted mode lets us skip the INSTALL step, and even run a query as soon as we create it, to offer a more interactive experience. These one-step interpreted queries are unnamed (anonymous) and parameterless, just like SQL.

Example 1. Find the direct or indirect superclass (including the self class) of the TagClass whose name is "TennisPlayer".

Example 1. Directed Edge Pattern Unconstrained Repetition

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  TagClass1 =  SELECT t
               FROM TagClass:s - (IS_SUBCLASS_OF>*) - TagClass:t
               WHERE s.name == "TennisPlayer";

    PRINT  TagClass1;
}
```

You can copy the above GSQL script to a file named example1.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example1.gsql
```

Note below that the starting vertex s, whose name is TennisPlayer, is also a match, using a path with zero hops.

Output of Example 1

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"TagClass2": [
    {
      "v_id": "211",
      "attributes": {
        "name": "Person",
        "id": 211,
        "url": "http://dbpedia.org/ontology/Person"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "0",
      "attributes": {
        "name": "Thing",
        "id": 0,
        "url": "http://www.w3.org/2002/07/owl#Thing"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "149",
      "attributes": {
        "name": "Athlete",
        "id": 149,
        "url": "http://dbpedia.org/ontology/Athlete"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "59",
      "attributes": {
        "name": "TennisPlayer",
        "id": 59,
        "url": "http://dbpedia.org/ontology/TennisPlayer"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "239",
```

```
      "attributes": {
        "name": "Agent",
        "id": 239,
        "url": "http://dbpedia.org/ontology/Agent"
      },
      "v_type": "TagClass"
    }
  ]}]
}
```


Example 2. Find the immediate superclass of the TagClass whose name is "TennisPlayer". (This is equivalent to a 1-hop non-repeating pattern.)

Exmaple 2. Exactly 1 Repetition of A Directed Edge

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  TagClass1 =  SELECT t
               FROM TagClass:s - (IS_SUBCLASS_OF>*1) - TagClass:t
               WHERE s.name == "TennisPlayer";

  PRINT TagClass1;
}
```

You can copy the above GSQL script to a file named example2.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example2.gsql
```

Output of Example 2

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"TagClass2": [{
    "v_id": "149",
    "attributes": {
      "name": "Athlete",
      "id": 149,
      "url": "http://dbpedia.org/ontology/Athlete"
    },
    "v_type": "TagClass"
  }]}]
}
```


Example 3. Find the 1 to 2 hops direct and indirect superclasses of the TagClass whose name is "TennisPlayer".

Example 3. 1 to 2 Repetition Of A Directed Edge.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  TagClass1 =  SELECT t
               FROM TagClass:s - (IS_SUBCLASS_OF>*1..2) - TagClass:t
               WHERE s.name == "TennisPlayer";

    PRINT  TagClass1;
}
```

You can copy the above GSQL script to a file named example3.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example3.gsql
```

Output of Example 3


```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"TagClass2": [
    {
      "v_id": "149",
      "attributes": {
        "name": "Athlete",
        "id": 149,
        "url": "http://dbpedia.org/ontology/Athlete"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "211",
      "attributes": {
        "name": "Person",
        "id": 211,
        "url": "http://dbpedia.org/ontology/Person"
      },
      "v_type": "TagClass"
    }
  ]}]
}
```


Example 4. Find the superclasses within 2 hops of the TagClass whose name is "TennisPlayer".

Example 4. Up-to 2 Repetition Of A Directed Edge.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  TagClass1 =  SELECT t
               FROM TagClass:s - (IS_SUBCLASS_OF>*..2) - TagClass:t
               WHERE s.name == "TennisPlayer";

    PRINT  TagClass1;
}
```

You can copy the above GSQL script to a file named example4.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example4.gsql
```

Output of Example 4

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"TagClass2": [
    {
      "v_id": "211",
      "attributes": {
        "name": "Person",
        "id": 211,
        "url": "http://dbpedia.org/ontology/Person"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "149",
      "attributes": {
        "name": "Athlete",
        "id": 149,
        "url": "http://dbpedia.org/ontology/Athlete"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "59",
      "attributes": {
        "name": "TennisPlayer",
        "id": 59,
        "url": "http://dbpedia.org/ontology/TennisPlayer"
      },
      "v_type": "TagClass"
    }
  ]}]
}
```

Example 5. Find the superclasses at least one hop from the TagClass whose name is "TennisPlayer".

Example 5. At Least 1 Repetition Of A Directed Edge.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  TagClass1 =  SELECT t
               FROM TagClass:s - (IS_SUBCLASS_OF>*1..) - TagClass:t
               WHERE s.name == "TennisPlayer";

    PRINT  TagClass1;
}
```

You can copy the above GSQL script to a file named example5.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example5.gsql
```

Output of Example 5

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"TagClass2": [
    {
      "v_id": "211",
      "attributes": {
        "name": "Person",
        "id": 211,
        "url": "http://dbpedia.org/ontology/Person"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "0",
      "attributes": {
        "name": "Thing",
        "id": 0,
        "url": "http://www.w3.org/2002/07/owl#Thing"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "149",
      "attributes": {
        "name": "Athlete",
        "id": 149,
        "url": "http://dbpedia.org/ontology/Athlete"
      },
      "v_type": "TagClass"
    },
    {
      "v_id": "239",
      "attributes": {
        "name": "Agent",
        "id": 239,
        "url": "http://dbpedia.org/ontology/Agent"
      },
      "v_type": "TagClass"
    }
  ]}]
}
```

```
}
```

Example 6. Find the 3 most recent comments that are liked or created by Viktor Akhiezer, and the total number of comments related to (created or liked by) Viktor Akhiezer.

Example 6. Disjunctive 1-Repetition Directed Edge.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{
  SumAccum<INT> @@commentCnt = 0;

  # find top 3 latest comments that is liked or created by Viktor Akhieze
  # and the total number of comments related to Viktor Akhiezer
  Top3Comments = SELECT p
                 FROM Person:s - ((<HAS_CREATOR|LIKES>)*1) - Comment:p
                 WHERE s.firstName == "Viktor" AND s.lastName == "Akhieze
                 ACCUM @@commentCnt += 1
                 ORDER BY p.creationDate DESC
                 LIMIT 3;

  PRINT Top3Comments;
  # total number of comments related to Viktor Akhiezer
  PRINT  @@commentCnt;
}
```

You can copy the above GSQL script to a file named example6.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example6.gsql
```

Output of Example 6

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [
    {"Top3Comments": [
      {
        "v_id": "2061584720640",
        "attributes": {
          "browserUsed": "Chrome",
          "length": 4,
          "locationIP": "194.62.64.117",
          "id": 2061584720640,
          "creationDate": "2012-09-06 06:46:31",
          "content": "fine"
        },
        "v_type": "Comment"
      },
      {
        "v_id": "2061586872389",
        "attributes": {
          "browserUsed": "Chrome",
          "length": 90,
          "locationIP": "31.216.177.175",
          "id": 2061586872389,
          "creationDate": "2012-08-28 14:54:46",
          "content": "About Hector Berlioz, his compositions Symphonie fa
        },
        "v_type": "Comment"
      },
      {
        "v_id": "2061590804929",
        "attributes": {
          "browserUsed": "Chrome",
          "length": 83,
          "locationIP": "194.62.64.117",
          "id": 2061590804929,
          "creationDate": "2012-09-04 16:16:56",
          "content": "About Muttiah Muralitharan, mit by nine degrees, fi
        },
        "v_type": "Comment"
      }
    ]},
```

```
    ]],
    {"@@commentCnt": 152}
  ]
}
```

Multiple Hop Patterns and Accumulation

Multiple Hop Pattern Shortest Path Semantics

Repeating the same hop is useful sometimes, but the real power of pattern matching comes from expressing multi-hop patterns, with specific characteristics for each hop. For example, the well-known product recommendation phrase "People who bought this product also bought this other product", is expressed by the following 2-hop pattern:

```
FROM This_Product:p -(<Bought:b1)- Customer:c -(Bought>:b2)- Product:p2
WHERE p2 != p
```

As you see, a 2-hop pattern is a simple concatenation and merging of two 1-hop patterns where the two patterns share a common endpoint. Below, Y:y is the connecting end point.

2-hop pattern

```
FROM X:x - (E1:e1) - Y:y - (E2>:e2) - Z:z
```

Similarly, a 3-hop pattern concatenates three 1-hop patterns in sequence, each pair of adjacent hops sharing one end point. Below, Y:y and Z:z are the connecting end points.

3-hop pattern

```
FROM X:x - (E2>:e2) - Y:y - (<E3:e3) - Z:z - (E4:e4) - U:u
```

In general, we can connect N 1-hop patterns into a N-hop pattern. The database will search the graph topology to find subgraphs that match this N-hop pattern.

Unnamed Intermediate Vertex Set

A multi-hop pattern has two endpoint vertex sets and one or more intermediate vertex sets. If the query does not need to express any conditions for an intermediate vertex set, then the vertex set can be omitted and the two surrounding edge sets can be joined with a simple ".". For example, in the 2-hop pattern example above, if we do not need to specify the type of the intermediate vertex Y, nor need to refer to it in any of the query's other clauses (such as WHERE or ACCUM), then the pattern can be reduced as follows:

```
FROM X:x - (E1.E2>) - Z:z
```

Note that when we abbreviate that path in this way, we do not support aliases for the edges or intermediate vertices in the abbreviated section.

Shortest Paths Only for Variable Length Patterns

If a pattern has a Kleene star to repeat an edge, GSQL pattern matching selects only the *shortest paths* which match the pattern. If we did not apply this restriction, computer science theory tells us that the computation time could be unbounded or extreme (NP = non-polynomial, to be technical). If we instead matched ALL paths regardless of length when a Kleene star is used without an upper bound, there could be an infinite number of matches, if there are loops in the graph. Even without loops or with an upper bound, the number of paths to check grows exponentially with the number of hops.

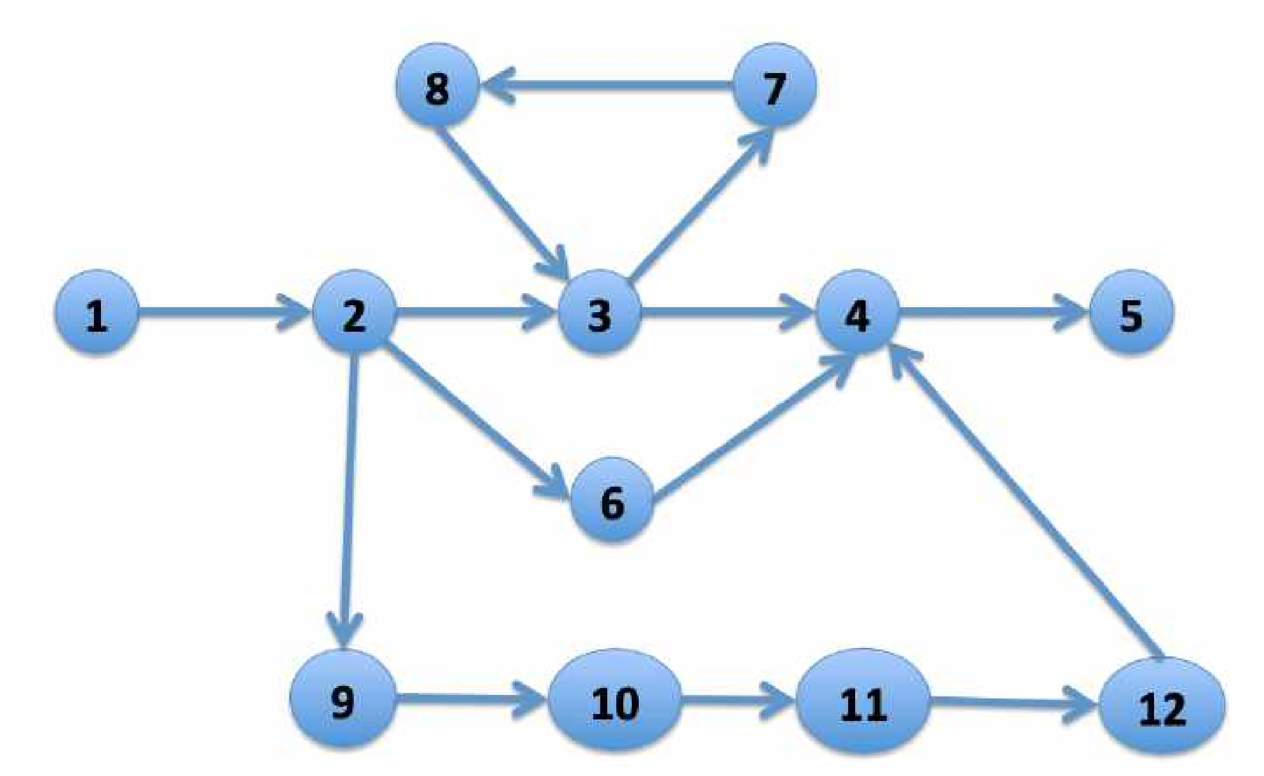


Figure 3. Shortest Path Illustration

For the pattern `1 - (_*) - 5` in Figure 3 above, you can see the following:

- **There are TWO shortest paths: 1-2-3-4-5 and 1-2-6-4-5**
  - These have 4 hops, so we can stop searching after 4 hops. This makes the task tractable.
- If we search for ALL paths which do not repeat any vertices:
  - There are THREE non-repeated-vertex paths: 1-2-3-4-5, 1-2-6-4-5, and 1-2-9-10-11-12-4-5
  - The actual number of matches is small, but the number of paths is theoretically very large.
- If we search for ALL paths which do not repeat any edges:
  - There are FOUR non-repeated-edge paths: 1-2-3-4-5, 1-2-6-4-5, 1-2-9-10-11-12-4-5, and 1-2-3-7-8-3-4-5
  - The actual number of matches is small, but number of paths to consider is NP.
- If we search for ALL paths with no restrictions:
  - There are an infinite number of matches, because we can go around the 3-7-8-3 cycle any number of times.

Additional Details about Pattern Matching

In the early version of Pattern Matching (TigerGraph v2.4 to v2.6), there were a number of restrictions on the WHERE, ACCUM and POST-ACCUM clauses In TigerGraph 3.0, most of these restrictions are lifted.

Each vertex set or edge set in a pattern (except edges with Kleene stars) can have an alias variable associated with it. When the query runs and finds matches, it associates, or binds, each alias to the matching vertices or edges in the graph.

SELECT Clause

The SELECT clause specifies the output vertex set of a SELECT statement. For a multiple-hop pattern, we can select any vertex alias in the pattern. The example below shows the 4 possible choices for the given pattern:

SELECT Clause Can Select Any Vertex Alias

```
#select starting end point x
SELECT x
FROM X:x-(E2>:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u;

#select y
SELECT y
FROM X:x-(E2>:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u;

#select z
SELECT z
FROM X:x-(E2>:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u;

#select ending end point u
SELECT u
FROM X:x-(E2>:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u;
```

FROM Clause

For a multiple-hop pattern, if you don't need to refer to the intermediate vertex points, you can just use "." to connect the edge patterns, giving a more succinct representation. For example, below we remove y and z, and connect E2>, <E3 and E4 using the period symbol. Note that you cannot have an alias for a multi-hop sequence like E2>.<E3.E4.

Omitting intermediate vertex y and z.

```
#select starting end point x
SELECT x
FROM X:x-(E2>:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u;

#if we don't need to access y, z, we can write
SELECT u
FROM X:x-(E2>.<E3.E4)-U:u;
```

WHERE Clause

Beginning with TigerGraph v3.0, each predicate (simple true/false condition) can refer to any of the aliases in the path. As with any database query, more complex conditions may not be as performant as simpler queries with simpler, more local predicate conditions. Consider the pattern and query below:

```
FROM X1:x1-(E1:e1)-X2:x2-(E2:e2)-X3:x3-(E3:e3)-X4:x4
```

WHERE Clause Support both Local Predicate and Cross-hop predicates.

```
# (x1, e1, x2) belongs to the 1st-hop
# (x2, e2, x3) belongs to the 2nd-hop
# (x3, e3, x4) belongs to the last-hop
# below x1.age > x2.age is a local predicate
# x2.@cnt != x4.@cnt is a cross-hop predicate
# (x1.salary + x3.salary) < x4.salary is a cross-hop predicate
SELECT x
FROM X1:x1-(E1:e1)-X2:x2-(E2:e2)-X3:x3-(E3:e3)-X4:x4
WHERE x1.age>x2.age AND x2.@cnt!=x4.@cnt AND (x1.salary+x3.salary)<x4.sal
```

Path Patterns as a Regular Expression Language

GSQL's pattern matching syntax provides the essentials for a regular expression language for paths in graphs. Consider the three basic requirements for a regular expression language:

1. **The empty set** → A path of length zero (no match)
2. **Concatenation** → Form a path by adding one on two another. You can write an N-hop pattern, and M-hop pattern, and then combine them to have a (N+M)-hop pattern.
3. **Alternation (either-or)** → You can use alternation for both vertex sets and edge sets, e.g.
   `FROM (Source1 | Source2) -(Edge1> | <Edge 2)- (Target1 | Target2)`
   Note: This is not the same as
   `FROM (Source1 -(Edge1>)- Target 1)   |   (Source2 -(<Edge2)- Target 2)`
   The latter can be achieved by writing two SELECT query blocks and getting the UNION of their results.

Working with Your Pattern Matches

The point of pattern matching is to identity sets of graph entities that match your input pattern. Once you've done that, GSQL enables you to do advanced and efficient computation on that data, from simply counting the matches to advanced algorithms and analytics. This section compares accumulation in the current Pattern Matching syntax to earlier versions, but it does not attempt to explain accumulators in full. You may want to consult the Accumulators Tutorial and and the GSQL Language Reference's section on the ACCUM and POST-ACCUM clauses.

ACCUM Clause

TigerGraph 3.0 removes the Pattern Matching (SYNTAX v2)-related restrictions on the ACCUM and POST-ACCUM clause.

Just as in classic GSQL syntax, the ACCUM clause it executed once (in parallel) for each set of vertices and edges in the graph which match the pattern and constraints given in the FROM and WHERE clauses. You can think of FROM-WHERE as producing a virtual table. The columns of this matching table are the alias variables from the FROM clause pattern, and the rows are each possible set of vertex and edge aliases (e.g. a path) which fit the pattern.

A simple pattern 1-hop pattern, which could be syntax v1 or v2, like this:

```
FROM Person:A -(IS_LOCATED_IN:B)- City:C
```

produces a match table with 3 columns: A, B, and C. Each row is a tuple (A,B,C) where there is a `has_lived_in` edge B from a `Person` vertex A to a `City` vertex C. We say that the match table provides a *binding* between the pattern aliases and graph's vertices and edges. A multi-hop pattern simply has more columns than a 1-hop pattern.

The ACCUM clause iterates through ALL matches. If you do not have an alias on every vertex in the pattern, then the number of **distinct** matches may be less than that number of matches.

For, example, consider

```
FROM Person:A -(KNOWS.KNOWS)- Person.C
WHERE C.email = "Andy@www.com"
ACCUM C.@patternCount += 1
```

This asks who are the friends of friends of Andy@www.com. Suppose Andy knows 3 persons (Larry, Moe, and Curly) who know Wendy. The accumulator `C.@patternCount` will be incremented 3 times for C = Wendy. This is similar to a SQL `SELECT C, COUNT(*) ... GROUP BY C` query. There is no alias for the vertex in the middle of `KNOWS.KNOWS` so the identities of Larry, Moe, and Curly cannot be reported.

POST-ACCUM Clause

As of TigerGraph 3.0, Pattern Matching (V2) syntax supports multiple POST-ACCUM clauses.

At the end of the ACCUM clause, all the requested accumulation (+=) operators are processed in bulk, and the updated values are now visible. You can now use POST-ACCUM clauses to perform a second, different round of computation on the results of your pattern matching.

The **ACCUM** clause executes **for each full path** that matches the pattern in the FROM clause. In contrast, the **POST-ACCUM** clause executes **for each vertex** in one vertex set (e.g. one vertex column in the matching table); its statements can access the aggregated accumulator result computed in the ACCUM clause. New for v3.0, if you want to perform per-vertex updates for more than one vertex alias, you should use a separate POST-ACCUM clause for each vertex alias. The multiple POST-ACCUM clauses are processed in parallel; it doesn't matter in what order you write them. (For each binding, the statements within a clause are executed in order.)

For example, below we have two POST-ACCUM clauses. The first one iterates through s, and for each s, we do `s.@cnt2 += s.@cnt1`. The second POST-ACCUM iterations through t.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @cnt1;
  SumAccum<int> @cnt2;

  R   =  SELECT s
         FROM Person:s-(LIKES>) -:msg - (HAS_CREATOR>)-Person:t
         WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
               AND t.lastName LIKE "S%" AND year(msg.creationDate) == 201
         ACCUM s.@cnt1 +=1 //execute this per match of the FROM pattern.
         POST-ACCUM s.@cnt2 += s.@cnt1 //execute once per s.
         POST-ACCUM t.@cnt2 +=1;//execute once per t

  PRINT R [R.firstName, R.lastName, R.@cnt1, R.@cnt2];
}
```

which produces the result

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [
    {"R": [{
      "v_id": "28587302323577",
      "attributes": {
        "R.firstName": "Viktor",
        "R.@cnt1": 3,
        "R.lastName": "Akhiezer",
        "R.@cnt2": 3
      },
      "v_type": "Person"
    }]},
  ]
}
```

However, the following is not allowed, since it involves two aliases (t and s) in one POST-ACCUM clause.

```
POST-ACCUM t.@cnt1 += 1,
           s.@cnt1 += 1
```

Also, you may not use more than one alias in a single assignment. The following is not allowed:

```
POST-ACCUM t.@cnt1 += s.@cnt + 1
```

Examples of Multiple Hop Pattern Match

Example 1. Find the 3rd superclass of the Tag class whose name is "TennisPlayer".

Example1. Succinct Representation Of Multiple-hop Pattern

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  TagClass1 =
       SELECT t
       FROM TagClass:s-(IS_SUBCLASS_OF>.IS_SUBCLASS_OF>.IS_SUBCLASS_OF>)-
       WHERE s.name == "TennisPlayer";

  PRINT TagClass1;
}
```

You can copy the above GSQL script to a file named example1.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example1.gsql
```

Output of Example 1

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"TagClass2": [{
    "v_id": "239",
    "attributes": {
      "name": "Agent",
      "id": 239,
      "url": "http://dbpedia.org/ontology/Agent"
    },
    "v_type": "TagClass"
  }]}]
}
```

Example 2. Find in which continents were the 3 most recent messages in Jan 2011 created.

Example1. Disjunction In A Succinct Representation Of Multiple-hop Pattern

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{

  SumAccum<String> @continentName;

  accMsgContinent =
                SELECT s
                FROM (Comment|Post):s-(IS_LOCATED_IN>.IS_PART_OF>)-Conti
                WHERE year(s.creationDate) == 2011 AND month(s.creationD
                ACCUM s.@continentName = t.name
                ORDER BY s.creationDate DESC
                LIMIT 3;

  PRINT accMsgContinent;
}
```

You can copy the above GSQL script to a file named example2.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example2.gsql
```

Output of Example 2

```
Using graph 'ldbc_snb'
```

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"accMsgContinent": [
    {
      "v_id": "824640012997",
      "attributes": {
        "browserUsed": "Firefox",
        "length": 7,
        "locationIP": "27.112.21.246",
        "@continentName": "Asia",
        "id": 824640012997,
        "creationDate": "2011-01-31 23:54:28",
        "content": "no way!"
      },
      "v_type": "Comment"
    },
    {
      "v_id": "824636727408",
      "attributes": {
        "browserUsed": "Firefox",
        "length": 3,
        "locationIP": "31.2.225.17",
        "@continentName": "Europe",
        "id": 824636727408,
        "creationDate": "2011-01-31 23:57:46",
        "content": "thx"
      },
      "v_type": "Comment"
    },
    {
      "v_id": "824634837528",
      "attributes": {
        "imageFile": "",
        "browserUsed": "Internet Explorer",
        "length": 115,
        "locationIP": "87.251.6.121",
        "@continentName": "Asia",
        "id": 824634837528,
        "creationDate": "2011-01-31 23:58:03",
        "lang": "tk",
        "content": "About Adolf Hitler, jews. His writings and methods we
```

```
        },
        "v_type": "Post"
      }
    ]}]
  }
```

Example 3. Find Viktor Akhiezer's favorite author of 2012 whose last name begins with character 'S'. Also find how many LIKES Viktor has given to the author's post or comment.

Example 3. Multiple-hop Pattern With Accumulator Applied To All Matched Paths

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{
  SumAccum<int> @likesCnt;

  FavoriteAuthors =
            SELECT t
            FROM Person:s-(LIKES>) -:msg - (HAS_CREATOR>)-Person:t
            WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
                      AND t.lastName LIKE "S%" AND year(msg.creationDate)
            ACCUM t.@likesCnt +=1;

  PRINT FavoriteAuthors[FavoriteAuthors.firstName, FavoriteAuthors.lastNa
}
```

You can copy the above GSQL script to a file named example3.gsql, and invoke this script file in a Linux shell.

Linux Bash

```
gsql example3.gsql
```

Output of Example 3

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"FavoriteAuthors": [
    {
      "v_id": "8796093025410",
      "attributes": {
        "FavoriteAuthors.firstName": "Priyanka",
        "FavoriteAuthors.lastName": "Singh",
        "FavoriteAuthors.@likesCnt": 1
      },
      "v_type": "Person"
    },
    {
      "v_id": "2199023260091",
      "attributes": {
        "FavoriteAuthors.firstName": "Janne",
        "FavoriteAuthors.lastName": "Seppala",
        "FavoriteAuthors.@likesCnt": 1
      },
      "v_type": "Person"
    },
    {
      "v_id": "15393162796846",
      "attributes": {
        "FavoriteAuthors.firstName": "Mario",
        "FavoriteAuthors.lastName": "Santos",
        "FavoriteAuthors.@likesCnt": 1
      },
      "v_type": "Person"
    }
  ]}]
}
```

Multi-Block Queries

We have shown how complex multi-hop patterns, containing even a conjunctive of patterns, can be expressed in a single FROM clause of a single SELECT query. There are times, however, when it is better or necessary to write query as more than one SELECT block. This could be because of the need to do computation and decision matching in stages, to make the query easier to read, or to optimize performance.

Regardless of the reason, GSQL has always supported writing procedural queries containing multiple SELECT query blocks. Moreover, each SELECT statement outputs a vertex set. This vertex set can be used in the FROM clause of an subsequence SELECT block.

For example, if Set1, Set2, and Set3 were the outputs of three previous SELECT blocks in this query, then each of these FROM clauses can take place later in the query:

- `FROM    Set1:x1 -(mh1)- :x2 -(mh2)- Set3:x3`
- `FROM    :x1 -(mh1)- :x2 -(mh2)- Set3:x3`
- `FROM    Set2:x1 -(mh1)- :x2 -(mh2)- Set2:x3`

Example 1. Find Viktor Akhiezer's liked messages' authors, whose last name starts with letter S. Find these authors alumni count.

```
USE GRAPH ldbc_snb

# a computed vertex set F is used to constrain the second pattern.
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  F  =  SELECT t
        FROM :s -(LIKES>:e1)- :msg -(HAS_CREATOR>)- :t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer" AND t.

  Alumni = SELECT p
           FROM Person:p -(STUDY_AT>) -:u - (<STUDY_AT)- F:s
           WHERE s != p
           Per (p)
           POST-ACCUM @@cnt+=1;


  PRINT @@cnt;

}

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 216}]
}
```

Example 2. Find Viktor Akhiezer's liked posts' authors A, and his liked comments' authors B. Count the common universities that both A and B have members studied at.

```
USE GRAPH ldbc_snb

#A and B are used to constraint the third pattern.
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  A  =  SELECT t
        FROM :s -(LIKES>:e1)- Post:msg -(HAS_CREATOR>)- :t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer" ;


  B  =  SELECT t
        FROM :s -(LIKES>:e1)- Comment:msg -(HAS_CREATOR>)- :t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer" ;

  Univ = SELECT u
         FROM A:p -(STUDY_AT>) -:u - (<STUDY_AT)- B:s
         WHERE s != p
         Per (u)
         POST-ACCUM @@cnt+=1;


  PRINT @@cnt;

}

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 4}]
}
```

Example 3. Find Viktor Akhiezer's liked posts' authors A. See how many pair of persons in A that one person likes a message authored by another person.

```
USE GRAPH ldbc_snb

# a computed vertex set A is used twice in the second pattern.
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  A  =  SELECT t
        FROM :s -(LIKES>:e1)- Post:msg -(HAS_CREATOR>)- :t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer" ;

  A = SELECT p
      FROM A:p -(LIKES>) -:msg - (HAS_CREATOR>) - A:p2
      WHERE p2 != p
      Per (p, p2)
      ACCUM @@cnt +=1;


  PRINT @@cnt;

}

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 14833}]
}
```

Example 4. Find how many messages are created and liked by the same person whose first name begins with letter T.

```
USE GRAPH ldbc_snb

# the same alias is used twice in a pattern
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  A  =  SELECT msg
        FROM :s -(LIKES>:e1)- :msg -(HAS_CREATOR>)- :s
        WHERE s.firstName LIKE "T%"
        PER (msg)
        ACCUM @@cnt +=1;


  PRINT @@cnt;

}
#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 207}]
}

#to further verify, we picked one message from the above query result.
#see if there exists a person who like her own message.
INTERPRET QUERY () SYNTAX v2 {

    R = SELECT s
        FROM :msg -(HAS_CREATOR>)- :s
        WHERE msg.id == 1374390714042;

    T =  SELECT s
         FROM R:s -(LIKES>)- :msg
         WHERE msg.id == 1374390714042;

  PRINT R;
  PRINT T;

}

#result
{
```

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [
    {"R": [{
      "v_id": "13194139533433",
      "attributes": {
        "birthday": "1985-11-26 00:00:00",
        "firstName": "Taras",
        "lastName": "Kofler",
        "gender": "female",
        "speaks": [
          "uk",
          "ro",
          "en"
        ],
        "browserUsed": "Internet Explorer",
        "locationIP": "31.131.28.133",
        "id": 13194139533433,
        "creationDate": "2011-01-29 01:14:27",
        "email": [
          "Taras13194139533433@gmail.com",
          "Taras13194139533433@yahoo.com"
        ]
      },
      "v_type": "Person"
    }]},
    {"T": [{
      "v_id": "13194139533433",
      "attributes": {
        "birthday": "1985-11-26 00:00:00",
        "firstName": "Taras",
        "lastName": "Kofler",
        "gender": "female",
        "speaks": [
          "uk",
          "ro",
          "en"
        ],
        "browserUsed": "Internet Explorer",
        "locationIP": "31.131.28.133",
        "id": 13194139533433,
        "creationDate": "2011-01-29 01:14:27",
        "email": [
```

```
            "Taras13194139533433@gmail.com",
            "Taras13194139533433@yahoo.com"
          ]
        },
        "v_type": "Person"
      }]}
    ]
  }
```


A Recommendation Application

In this example, we want to recommend some messages (comments or posts) to the person Viktor Akhiezer.

How do we do this?

One way is to find Others who likes the same messages Viktor likes. And then recommend the messages that Others like but Viktor have not seen. The pattern is roughly like below

- Viktor - (Likes>) - Message - (<Likes) - Others
- Others - (Likes>) - NewMessage
- Recommend NewMessage to Viktor

However, this is too fine granularity, and we are overfitting the message level data with a collaborative filtering algorithm. The intuition is that two persons are similar to each other when their "liked" messages fall into the same category (tag). This makes more sense and common than finding two persons that "likes" the same set of messages. As a result, one way to avoid this overfitting is to go one level above. That is, instead of finding common messages as a similarity base, we find common messages' tags as a similarity base. Person A and Person B are similar if they like messages that belong to the same tag. This scheme fixes the overfitting problem. In pattern match vocabulary, we have

- Viktor - (Likes>) - Message - (Has>) - **Tag** - (<Has) - Message - (<Likes) - Others
- Others - (Likes>) - NewMessage

- Recommend NewMessage to Viktor

GSQL. RecommendMessage Application.

This time, we create the query first, and interpret the query by calling the query name with parameters.
If we are satisfied with this query, we can use "install query queryName" to get the compiled query installed which has the best performance.

GSQL Recommendation Algorithm

```
use graph ldbc_snb
set query_timeout=60000
DROP QUERY RecommendMessage

CREATE QUERY RecommendMessage (String fn, String ln) SYNTAX v2{

  SumAccum<int> @TagInCommon;
  SumAccum<float> @SimilarityScore;
  SumAccum<float> @Rank;
  OrAccum @Liked = false;

   #1. mark messages liked by Viktor
   #2. calculate log similarity score for each persons share the same
   #   interests at Tag level.
    Others =
       SELECT p
       FROM Person:s-(LIKES>)-:msg - (HAS_TAG>.<HAS_TAG.<LIKES)- :p
       WHERE s.firstName == fn AND s.lastName == ln
       ACCUM msg.@Liked = true, p.@TagInCommon +=1
       POST-ACCUM p.@SimilarityScore = log (1 + p.@TagInCommon);

    # recommend new messages to Viktor that have not liked by him.
    RecommendedMessage =
             SELECT msg
             FROM Others:o-(LIKES>) - :msg
             WHERE  msg.@Liked == false
             ACCUM msg.@Rank +=o.@SimilarityScore
             ORDER BY msg.@Rank DESC
             LIMIT 2;

  PRINT   RecommendedMessage[RecommendedMessage.content, RecommendedMessa
}


INTERPRET QUERY RecommendMessage ("Viktor", "Akhiezer")
#try the second person with just parameter change.
INTERPRET QUERY RecommendMessage ("Adriaan", "Jong")
```

You can copy the above GSQL script to a file named app1.gsql, and invoke this script file in linux command line.

Linux Bash

```
gsql app1.gsql
```

Output of App1

```
Using graph 'ldbc_snb'
The query RecommendMessage is dropped.
The query RecommendMessage has been added!
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"RecommendedMessage": [
    {
      "v_id": "549760294602",
      "attributes": {
        "RecommendedMessage.@Rank": 4855.49219,
        "RecommendedMessage.content": "About Indira Gandhi, Gandhi establ
      },
      "v_type": "Post"
    },
    {
      "v_id": "549760292109",
      "attributes": {
        "RecommendedMessage.@Rank": 4828.7251,
        "RecommendedMessage.content": "About Ho Chi Minh, nam, as an anti
      },
      "v_type": "Post"
    }
  ]}]
}
```

Install the query

When you are satisfied with your query in the GSQL interpret mode, you can now install it as a generic service which has a much faster speed. Since we have been using "CREATE QUERY .." syntax, the query is added into the catalog, we can set the syntax version and install it.

GSQL Prepare Install Query

```
#before install the query, need to set the syntax version
SET syntax_version="v2"
USE GRAPH ldbc_snb

#install query
INSTALL QUERY RecommendMessage
```

GSQL Run the Installed Query

```
GSQL > install query RecommendMessage
Start installing queries, about 1 minute ...
RecommendMessage query: curl -X GET 'http://127.0.0.1:9000/query/ldbc_snb

[========================================================================
GSQL > run query RecommendMessage("Viktor", "Akhiezer")
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"RecommendedMessage": [
    {
      "v_id": "549760294602",
      "attributes": {
        "RecommendedMessage.@Rank": 4855.49219,
        "RecommendedMessage.content": "About Indira Gandhi, Gandhi establ
      },
      "v_type": "Post"
    },
    {
      "v_id": "549760292109",
      "attributes": {
        "RecommendedMessage.@Rank": 4828.7251,
        "RecommendedMessage.content": "About Ho Chi Minh, nam, as an anti
      },
      "v_type": "Post"
    }
  ]}]
}
```

Linux Bash: Shutdown The System

```
#when you are not using the TigerGraph System on your laptop,
# to save resource, you can stop it by
gadmin stop
#when you need to start it again, use
gadmin start
```

The above use log-cosine as a similarity measurement. We can also use cosine similarity by using two persons liked messages.

GSQL Recommendation Algorithm 2

```
use graph ldbc_snb
set query_timeout=60000
DROP QUERY RecommendMessage

CREATE QUERY RecommendMessage (String fn, String ln) SYNTAX v2{

  SumAccum<int> @MsgInCommon = 0;
  SumAccum<int> @MsgCnt = 0 ;
  SumAccum<int> @@InputPersonMsgCnt = 0;
  SumAccum<float> @SimilarityScore;
  SumAccum<float> @Rank;
  SumAccum<float> @TagCnt = 0;
  OrAccum @Liked = false;
  float sqrtOfInputPersonMsgCnt;

   #1. mark messages liked by input user
   #2. find common msg between input user and other persons
    Others =
       SELECT p
       FROM Person:s-(LIKES>)-:msg -(<LIKES)-:p
       WHERE s.firstName == fn AND s.lastName == ln
       ACCUM msg.@Liked = true, @@InputPersonMsgCnt += 1,
             p.@MsgInCommon += 1;

    sqrtOfInputPersonMsgCnt = sqrt(@@InputPersonMsgCnt);

    #calculate cosine similarity score.
    #|AxB|/(sqrt(Sum(A_i^2)) * sqrt(Sum(B_i^2)))
    Others  =
        SELECT o
        FROM Others:o-(LIKES>)-:msg
        ACCUM o.@MsgCnt += 1
        POST-ACCUM o.@SimilarityScore = o.@MsgInCommon/(sqrtOfInputPerson

   #recommend new messages to input user that have not liked by him.
    RecommendedMessage =
             SELECT msg
             FROM Others:o-(LIKES>) - :msg
             WHERE  msg.@Liked == false
             ACCUM msg.@Rank +=o.@SimilarityScore
             ORDER BY msg.@Rank DESC
             LIMIT 3;

  PRINT   RecommendedMessage[RecommendedMessage.content, RecommendedMessa
}

INTERPRET QUERY RecommendMessage ("Viktor", "Akhiezer")
#try the second person with just parameter change.
```

```
INTERPRET QUERY RecommendMessage ("Adriaan", "Jong")
```

Advanced Features

This section includes some advanced features related to pattern match. It includes using the PER clause to fine control the ACCUM execution, DML support in pattern match, and the conjunctive pattern match syntax which allows multiple patterns in one FROM clause. We dedicate a subsection for each topic.

Per Clause (Beta)

Introduction

Pattern matching produces a virtual match table, and the ACCUM clause acts like a FOREACH loop, executing the clause's statement once for each row of the match table.

Patterns are paths in the graphs, and each row in the match table is a distinct path. However, paths may share some vertices or edges. Some applications do not want to do aggregations per path. Instead, they want to execute the ACCUM clause per distinct group of vertex aliases.

For example, consider the following query which counts the number of paths in a simple 2-hop pattern:

```
SumAccum<int> @@cnt;

S = SELECT t
    FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
    ACCUM @@cnt += 1;
```

Suppose the query produces the following match table.

```
(s, edge1, m , edge2, t)//match table schema
v1, e1, v3, e2, v2 //match 1
v1, e3, v4, e4, v2 //match 2
v5, e5, v6, e6, v7 //match 3
v8, e7, v9, e8, v7 //match 4
```

By default, the ACCUM clause will execute the `@@cnt += 1` statement 4 times, for each row in the match table. The result will be `@@cnt = 4`.

For the same query, what if the user wants to

- count the number of distinct path endings in the match table? For this case, we would want to iterate on the alias `t`.
- count the number of distinct (start, end) pairs in the match table? For that case, we would want to iterate on distinct pairs of the aliases `(s, t)`.

To provide users with this added flexibility and finer control over ACCUM iteration, TigerGraph 3.0 adds the PER clause to pattern matching (V2) syntax.

Syntax

The PER Clause is an optional clause that comes at the start of the ACCUM clause in a SELECT statement. As illustrated below, it starts with the keyword PER, and followed by a pair of parenthesis, in which user can put one or more distinct vertex aliases found in the FROM pattern.

```
selectBlock := SELECT alias
               FROM pattern
               [sampleClause]
               [whereClause]
               [[perClause] accumClause]
               [postAccumClause]*
               [havingClause]
               [orderClause]
               [limitClause]

perClause := PER (vertex_alias_1, vertex_alias_2, ...)
```

Examples. Below are multiple examples of the PER Clause using the same FROM clause.

```
S1 = SELECT s
     FROM S:s - (E1:edge1) - M:m - (E2:edge2) - T:t
     PER (s)
     ACCUM @@cnt += 1;

S2 = SELECT t
     FROM S:s - (E1:edge1) - M:m - (E2:edge2) - T:t
     PER (t)
     ACCUM @@cnt += 1;

S3 = SELECT m
     FROM S:s - (E1:edge1) - M:m - (E2:edge2) - T:t
     PER (m)
     ACCUM @@cnt += 1;

S4 = SELECT t
     FROM S:s - (E1:edge1) - M:m - (E2:edge2) - T:t
     PER (s, t)
     ACCUM @@cnt += 1;

S5 = SELECT t
     FROM S:s - (E1:edge1) - M:m - (E2:edge2) - T:t
     PER (s, m, t)
     ACCUM @@cnt += 1;
```

Semantics

The PER Clause specifies a list of vertex aliases, which are used to group the rows in the match table, one group per distinct value of the alias or of the alias list. If there are N distinct groups, we will execute the ACCUM clause N times, once per distinct vertex aliases' binding. Note that the PER clause has no effect on POST-ACCUM clauses semantic, except confining the POST-ACCUM vertex alias.

Suppose s, m, and t are vertex aliases in a pattern. Below are some interpretations of the PER Clause based on the graph element bindings found in the match table.

- `PER (s) ACCUM` means that per each distinct s vertex, execute the ACCUM clause once.
- `PER (s,t) ACCUM` means that per each distinct (s, t) pair, execute the ACCUM clause once.

- `PER (s,m,t) ACCUM` means that per each distinct (s, m, t) tuple, execute the ACCUM clause once.

Examples to show PER clause semantics.

```
//match table
(s, edge1, m , edge2, t)//schema
v1, e1, v3, e2, v2 //match 1
v1, e3, v4, e4, v2 //match 2
v5, e5, v6, e6, v7 //match 3
v8, e7, v9, e8, v7 //match 4

//since we have v1, v5, and v8 three distinct vertices bind to s,
//we execute ACCUM clause 3 times.
S1 = SELECT s
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (s)
     ACCUM @@cnt += 1;

//since we have v2, v7 two distinct vertices bind to t,
//we execute ACCUM clause twice.
S2 = SELECT t
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (t)
     ACCUM @@cnt += 1;

//since we have v3, v4, v6, v9 four distinct vertices bind to m,
//we execute ACCUM clause 4 times.
S3 = SELECT m
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (m)
     ACCUM @@cnt += 1;

//since we have (v1, v2), (v5, v7) and (v8, v7) three distinct vertex pai
//bind to (s,t), we execute ACCUM clause 3 times.
S4 = SELECT t
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (s, t)
     ACCUM @@cnt += 1;


//since we have (v1, v3, v2), (v1, v4, v2), (v5, v6, v7) and (v8, v9, v7)
//distinct vertex groups bind to (s,m,t), we execute ACCUM clause 4 times
S5 = SELECT t
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (s, m, t)
     ACCUM @@cnt += 1;
```

If the PER Clause is used in a SELECT query block, then the vertex aliases used in the SELECT, ACCUM , and POST-ACCUM clauses must be confined to the aliases that appear in the PER clause.

Below are some illegal cases.

```
//semantic error. SELECT t, but t doesn't appear in PER clause.
S1 = SELECT t
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (s, m)
     ACCUM @@cnt += 1;

//semantic error. ACCUM t.@cnt, but t doesn't appear in PER clause.
S2 = SELECT t
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (s, m)
     ACCUM t.@cnt += 1;

//semantic error. POST-ACCUM t.@cnt, but t doesn't appear in PER clause.
S3 = SELECT s
     FROM S:s - (E1:edge1) - M:m -(E2:edge2) - T:t
     PER (s)
     ACCUM s.@cnt += 1
     POST-ACCUM t.@cnt =1;
```

PER Clause Examples

Example 1. Count the number of Countries that has a City which has a resident that likes a post.

```
//Example 1.
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {
  SumAccum<int> @@cnt;

 R =   SELECT c
       FROM   Country:c -(<IS_PART_OF.<IS_LOCATED_IN.LIKES>)- Post:p
       PER    (c)
       ACCUM  @@cnt +=1;

 PRINT @@cnt;
}

//results
Using graph 'ldbc_snb'
The query AA is dropped.
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 111}]
}
```

Example 2. Count the number of posts liked by a person who is located in a city that belongs to a country. (All cities are in a country, but humor us. We are reusing the same FROM pattern in several examples.)

```
//Example 2.
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {
  SumAccum<int> @@cnt;

 R =   SELECT p
       FROM   Country:c -(<IS_PART_OF.<IS_LOCATED_IN.LIKES>)- Post:p
       PER    (p)
       ACCUM  @@cnt +=1;

 PRINT @@cnt;

//result
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 70668}]
}
```

Example 3. Find for each country in ("Dominican_Republic","Angola", "Cambodia") the number of posts that is liked by a person living in that country.

```
//Exmaple 3
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{

 MapAccum<string, SumAccum<int>> @@postPerCountry;

 R =   SELECT p
       FROM   Country:c -(<IS_PART_OF.<IS_LOCATED_IN.LIKES>)- Post:p
       WHERE  c.name in  ("Dominican_Republic","Angola", "Cambodia")
       PER    (c, p)
       ACCUM  @@postPerCountry += (c.name -> 1);

 PRINT @@postPerCountry;
}

//results
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@postPerCountry": {
    "Dominican_Republic": 395,
    "Angola": 12,
    "Cambodia": 4002
  }}]
}
```

Example 4. Find for each country in ("Dominican_Republic","Angola", "Cambodia") the number of posts that is liked by a person living in that country. Use local accumulators this time.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2{

 SumAccum<int> @postCnt;

 R =   SELECT c
       FROM   Country:c -(<IS_PART_OF.<IS_LOCATED_IN.LIKES>)- Post:p
       WHERE  c.name in  ("Dominican_Republic","Angola", "Cambodia")
       PER    (c, p) //per (country, post), add 1 to c.@postCnt
       ACCUM  c.@postCnt += 1;

 PRINT R;
}

//results
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"R": [
    {
      "v_id": "2",
      "attributes": {
        "@postCnt": 12,
        "name": "Angola",
        "id": 2,
        "url": "http://dbpedia.org/resource/Angola"
      },
      "v_type": "Country"
    },
    {
      "v_id": "67",
      "attributes": {
        "@postCnt": 4002,
        "name": "Cambodia",
        "id": 67,
        "url": "http://dbpedia.org/resource/Cambodia"
      },
      "v_type": "Country"
    },
    {
      "v_id": "11",
```

```
        "attributes": {
          "@postCnt": 395,
          "name": "Dominican_Republic",
          "id": 11,
          "url": "http://dbpedia.org/resource/Dominican_Republic"
        },
        "v_type": "Country"
      }
    ]}]
  }
```

Performance and Best Practices

The PER Clause not only helps users to control the semantics of the ACCUM clause, it also boosts the performance of the pattern match query, as it uses the PER clause to optimize the query execution.

To get the best performance, we recommend three guidelines for writing efficient queries.

Use PER (target) If Possible

Per target is in general faster than Per source. In the example below, query q2 is faster than q1. The only difference between these two queries is q2's FROM pattern is the flip of q1's FROM pattern.

```
USE GRAPH ldbc_snb

# not recommended, since it does per (src).
CREATE QUERY q1 () SYNTAX v2 {

  SumAccum<int> @@cnt ;

  T = SELECT c
      FROM Comment:c - (<LIKES) - Person:ps - (IS_LOCATED_IN>) - City:cit
      WHERE year(c.creationDate) >= 2006
      PER (c)
      ACCUM @@cnt += 1;

  PRINT @@cnt;
}

# recommended, since it does per (tgt)
CREATE QUERY q2 () SYNTAX v2 {

  SumAccum<int> @@cnt ;

  T = SELECT c
      FROM City:city - (<IS_LOCATED_IN) - Person:ps - (LIKES>) - Comment:
      WHERE year(c.creationDate) >= 2006
      PER (c)
      ACCUM @@cnt += 1;

  PRINT @@cnt;
}
```

Write Patterns with Smallest Expected Vertex Set on the Left

The match table is built by traversing the pattern from left to right. Follow the basic principle of pruning early rather than late by orienting the query the smaller cardinality set on the left. This practice will result in producing the least number of candidate matches during the query computation. For example, if there are fewer distinct tags than persons, then query q4 is faster than q3.

```
USE GRAPH ldbc_snb

# not recommended, since the pattern starts from a large cardinality vert
# (Person), and ends at a small cardinality vertex type (Tag).
CREATE QUERY q3 () SYNTAX v2 {

  SumAccum<int> @cnt;

  V = SELECT s
      FROM Person:s- (LIKES>)-Post:p - (<CONTAINER_OF)-:f - (HAS_TAG>) -
      PER (s)
      ACCUM s.@cnt += 1;

  PRINT V.size();
}

# recommended, start from small cardinality end (Tag), and use per tgt
CREATE QUERY q4 () SYNTAX v2 {

  SumAccum<int> @cnt;

  V = SELECT s
      FROM Tag:t-(<HAS_TAG)-Forum:f -(CONTAINER_OF>)-Post:p  - (<LIKES)-
      PER (s)
      ACCUM s.@cnt += 1;

  PRINT V.size();
}
```

Specify Complete Type Information

Specifying complete type information improves performance. For example, query q6 is faster than q5 even though they are known to be logically identical. `Forum` is the `CONTAINER_OF` `Post`, so it does not need to be specified in q5, but explicitly saying `Forum` in q6 speeds up performance.

```
USE GRAPH ldbc_snb

#we do not put Forum befoe :f, even if we know it.
CREATE QUERY q5 () SYNTAX v2 {

  SumAccum<int> @@person_cnt;

  V = SELECT s
      FROM Person:s- (LIKES>)-Post:p - (<CONTAINER_OF)-:f - (HAS_TAG>) -
      PER (s)
      ACCUM @@person_cnt += 1;

  PRINT @@person_cnt;
}


#recommended: we put Forum as the type info.
CREATE QUERY q6 () SYNTAX v2 {

  SumAccum<int> @@person_cnt;

  V = SELECT s
      FROM Person:s- (LIKES>)-Post:p - (<CONTAINER_OF)-Forum:f - (HAS_TAG
      PER (s)
      ACCUM @@person_cnt += 1;

  PRINT @@person_cnt;
}
```

LDBC Benchmark Queries

Using the PER clause and linear regular path pattern, we have translated all of the LDBC-SNB queries. You can find them on github at https://github.com/tigergraph/ecosys/tree/ldbc/ldbc_benchmark/tigergraph/queries_linear/queries ↗. Most of the queries are installed as functions. You can find sample parameter(s) of the functions from https://github.com/tigergraph/ecosys/tree/ldbc/ldbc_benchmark/tigergraph/queries/seeds ↗.

Data Modification

Pattern Matching GSQL supports Insert, Update, and Delete operations. The syntax is identical to that in classic GSQL (v1), though the full range of data modification operations are not yet support.

In general, data modification can be at two levels in GSQL:

1. Top level. The statement does not need to within any other statement.
2. Within a SELECT query statement. The FROM-WHERE clauses define a match table, and the data modification is performed based on the vertex and edge information in the match table. The GSQL specifications calls these within-SELECT statements **DML-sub statements**.

- Insert, Update, and Delete currently work in compiled mode only (e.g., you must run INSTALL QUERY before RUN QUERY.) Data Modification in interpreted mode is not yet available.
- SELECT queries with data modification may only have one POST-ACCUM clause.

Insert vertices and edges

Pattern matching Insert is supported at both the top-level and within-SELECT levels, using the same syntax as in classic GSQL. You can insert vertices and edges.

- For a top-level statement, use INSERT INTO,
- Inside an ACCUM or POST-ACCUM clause, use the DML-sub INSERT statement.

Example 1. Create a Person vertex, whose name is Tiger Woods. Next, find Viktor's favorite 2012 posts' authors, whose last name is prefixed with S. Finally, insert KNOWS edges connecting Tiger Woods with Viktor's favorite authors.

```
USE GRAPH ldbc_snb

#find Viktor's 2012 favorite posts' authors, whose lastName prefix with S
INTERPRET QUERY() SYNTAX V2 {

  R  =  SELECT t
        FROM Person:s -(LIKES>)- :msg -(HAS_CREATOR>)- Person:t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
         AND t.lastName LIKE "S%" AND year(msg.creationDate) == 2012;

  PRINT R[R.id, R.firstName, R.lastName];
}

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"R": [
    {
      "v_id": "8796093025410",
      "attributes": {
        "R.id": 8796093025410,
        "R.firstName": "Priyanka",
        "R.lastName": "Singh"
      },
      "v_type": "Person"
    },
    {
      "v_id": "2199023260091",
      "attributes": {
        "R.id": 2199023260091,
        "R.firstName": "Janne",
        "R.lastName": "Seppala"
      },
      "v_type": "Person"
    },
    {
      "v_id": "15393162796846",
      "attributes": {
        "R.id": 15393162796846,
        "R.firstName": "Mario",
        "R.lastName": "Santos"
      },
```

```
      "v_type": "Person"
    }
  ]}]
}

# create a Person node, whose name is tiger,
# and connect this Person with above Victor's favorite authors
CREATE QUERY InsertEdgeAndVertex () SYNTAX v2{

  #add a celebrity person node using INSERT INTO statement.
  INSERT INTO Person VALUES (100000000,"Tiger", "Woods", "m", _, _,_,_,_,

  R  =  SELECT t
        FROM Person:s -(LIKES>)- :msg -(HAS_CREATOR>)- Person:t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
         AND t.lastName LIKE "S%" AND year(msg.creationDate) == 2012
        PER (s, t)
        ACCUM
           #add edges connecting "tiger" and t with a 6/1/2020 time stamp
          INSERT INTO KNOWS VALUES(100000000, t, to_datetime("2020-06-01"

  PRINT R [R.id, R.firstName, R.lastName];
}

INSTALL QUERY InsertEdgeAndVertex
RUN QUERY InsertEdgeAndVertex()
```

You can verify the result by running a simple built-in REST endpoint.

Check the inserted vertex.

Linux Shell

```
#check the inserted vertex
curl -X GET "http://localhost:9000/graph/ldbc_snb/vertices/Person/1000000

#result
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 1
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "100000000",
      "v_type": "Person",
      "attributes": {
        "id": 100000000,
        "firstName": "Tiger",
        "lastName": "Woods",
        "gender": "m",
        "birthday": "1970-01-01 00:00:00",
        "creationDate": "1970-01-01 00:00:00",
        "locationIP": "",
        "browserUsed": "",
        "speaks": [],
        "email": []
      }
    }
  ]
}
```

Check the inserted edges.

Linux file

```
#check the inserted edges using tiger's id (100,000,000)
curl -X GET "http://localhost:9000/graph/ldbc_snb/edges/Person/100000000/
#result
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "KNOWS",
      "directed": false,
      "from_id": "100000000",
      "from_type": "Person",
      "to_id": "8796093025410",
      "to_type": "Person",
      "attributes": {
        "creationDate": "2020-06-01 00:00:00"
      }
    },
    {
      "e_type": "KNOWS",
      "directed": false,
      "from_id": "100000000",
      "from_type": "Person",
      "to_id": "2199023260091",
      "to_type": "Person",
      "attributes": {
        "creationDate": "2020-06-01 00:00:00"
      }
    },
    {
      "e_type": "KNOWS",
      "directed": false,
      "from_id": "100000000",
      "from_type": "Person",
      "to_id": "15393162796846",
      "to_type": "Person",
      "attributes": {
        "creationDate": "2020-06-01 00:00:00"
      }
    }
  ]
}
#note you can use the vertex lookup API to verify the three connected aut
```

```
curl -X GET "http://localhost:9000/graph/ldbc_snb/vertices/Person/8796093
```

Update data

To update vertex attributes, use assignment statements in a `POST-ACCUM` clause. To update edge attributes, use assignment statements in an `ACCUM` clause. In addition, data updates can only be performed if the `FROM` statement only contains a single-hop and fix-length pattern.

Query-body level `UPDATE` statements are not yet supported in syntax v2.

Example 2. For all `KNOWS` edges that connect Viktor Akhiezer and his friends whose `lastName` begins with "S", update the edge `creationDate` to "2020-10-01". Also, for the Person vertex (Tiger Woods) update the vertex's creationDate and language he speaks.

```
USE GRAPH ldbc_snb

DROP QUERY UpdateKnowsTS

CREATE QUERY UpdateKnowsTS () SYNTAX v2 {

  # update the vertex tiger's attributes
  # creationDate and languages spoken in POST-ACCUM
  R = SELECT p
      FROM Person:p
      WHERE p.firstName == "Tiger" AND p.lastName == "Woods"
      POST-ACCUM
            # update simple base type attribute
            p.creationDate = to_datetime("2020-6-1"),
            # update  collection-type attribute
            p.speaks = ("english", "golf");

  #DML-sub level, update KNOWS edge attribute "creationDate" in ACCUM
  R  =  SELECT t
        FROM Person:s-(KNOWS:e) -:t
        WHERE s.firstName == "Tiger" and s.lastName == "Woods"
        #update the KNOWS edge time stamp
        ACCUM e.creationDate = to_datetime("2020-10-01");
}

INSTALL QUERY UpdateKnowsTS
RUN QUERY UpdateKnowsTS()
```

To verify the update, we can use REST calls.

Check Tiger Woods' creationDate and language he speaks.

Linux Shell

```
curl -X GET "http://localhost:9000/graph/ldbc_snb/vertices/Person/1000000
#result
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "100000000",
      "v_type": "Person",
      "attributes": {
        "id": 100000000,
        "firstName": "Tiger",
        "lastName": "Woods",
        "gender": "m",
        "birthday": "1970-01-01 00:00:00",
        "creationDate": "2020-06-01 00:00:00",
        "locationIP": "",
        "browserUsed": "",
        "speaks": [
          "english",
          "golf"
        ],
        "email": []
      }
    }
  ]
}
```

Check KNOWS edges whose source is tiger woods.

Linux Shell

```
curl -X GET "http://localhost:9000/graph/ldbc_snb/edges/Person/100000000/

#result
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "KNOWS",
      "directed": false,
      "from_id": "100000000",
      "from_type": "Person",
      "to_id": "8796093025410",
      "to_type": "Person",
      "attributes": {
        "creationDate": "2020-10-01 00:00:00"
      }
    },
    {
      "e_type": "KNOWS",
      "directed": false,
      "from_id": "100000000",
      "from_type": "Person",
      "to_id": "2199023260091",
      "to_type": "Person",
      "attributes": {
        "creationDate": "2020-10-01 00:00:00"
      }
    },
    {
      "e_type": "KNOWS",
      "directed": false,
      "from_id": "100000000",
      "from_type": "Person",
      "to_id": "15393162796846",
      "to_type": "Person",
      "attributes": {
        "creationDate": "2020-10-01 00:00:00"
      }
    }
  ]
}
```

Delete vertices and edges

You can use delete () function to delete edges and vertices in ACCUM and POST-ACCUM clauses.

- Top-levels DELETE statements are not yet supported in SYNTAX v2.
- Edges can only be deleted in the ACCUM clause.
- For best performance, vertices should be deleted in the POST-ACCUM clause.
- To perform within-SELECT deletes, the FROM pattern can only be a ***single hop, fixed length*** pattern.

Example 3. Delete vertex Tiger Woods and its KNOWS edges.

```
USE GRAPH ldbc_snb

DROP QUERY  DeleteEdgeAndVertex

CREATE QUERY DeleteEdgeAndVertex () SYNTAX v2{

  R  =  SELECT t
        FROM Person:s -(KNOWS:e)- Person:t
        WHERE s.firstName == "Tiger" AND s.lastName == "Woods"
        ACCUM
           //delete edges
           DELETE(e)
        POST-ACCUM DELETE(s); //delete src vertex


  PRINT  R [R.id, R.firstName, R.lastName];
}

INSTALL QUERY DeleteEdgeAndVertex
RUN QUERY DeleteEdgeAndVertex()
```

To verify the result, you can use built-in REST calls.

```
curl -X GET "http://localhost:9000/graph/ldbc_snb/vertices/Person/1000000
#vertexresults
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": true,
  "message": "The input vertex id '100000000' is not a valid vertex id fo
  "code": "601"
}

curl -X GET "http://localhost:9000/graph/ldbc_snb/edges/Person/100000000/
#edge results
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": true,
  "message": "The input source_vertex_id '100000000' is not a valid verte
  "code": "601"
}
```

Conjunctive Pattern Matching (Beta)

What is Conjunctive Pattern Matching?

So far, we have described pattern matching as one path pattern in a FROM clause. In this section, we introduce GSQL's capability to match **multiple patterns** in one FROM clause. This extension is called Conjunctive Pattern Matching (CPM), because the query asks for the conjunction (logical AND) of the patterns. To get a match, all of the patterns must be satisfied, and the patterns can interrelate. Visually, you can think of patterns formed by a set of intersecting line segments. This feature, introduced as a Beta feature in TigerGraph 3.0, enables you to express complex patterns concisely in a single query block.

In general, a CPM query block consists of multiple patterns in the FROM clause. It has a structure illustrated below.

Conjunctive Pattern Matching Syntax

```
SelectBlock := SELECT alias
               FROM pattern
               [sampleClause]
               [whereClause]
               [ [perClause] accumClause]
               [postAccumClause]*
                    ...

pattern := vertexPattern | edgePattern | (pathPattern ["," pathPattern])
# vertexPattern and edgePattern are from classic GSQL
```

We elaborate on each of the clause.

SELECT Clause

The SELECT clause selects only one vertex alias from all the patterns in the FROM clause.

FROM Clause - Conjunctive Matching

This is where the conjunctive matching is expressed. The FROM clause consists of a list of path patterns, which are separated by commas. Evaluating each pattern against the underlying graph data produces a match table. If two patterns share a vertex alias, then we form the natural join of the two match tables.

For example, consider this CPM:

```
FROM X:x - (E1:e1) - Y:y - (E2>:e2) - Z:z,
     Z:z - (E3:e3) - U:u - (E4>:e4) - V:v
```

The first pattern's variables are x, e1, y, e2, and z; the second pattern's variables are z, e3, u, e4, and v. Considering the two patterns independently would yield the follwing match table schemas:

```
#first match table
(x, e1, y, e2, z)
#second match table
(z, e3, u, e4, v)
```

Natural Join of Match Tables

Natural joining two match tables compares all the shared vertex aliases between the two tables, and the resulting joined table contains all non-shared variables plus one copy of each of the shared vertex variables. Here is the match table for the CPM above:

```
#natural join result; the shared vertex variable z appears once.
(x, e1, y, e2, z, e3, u, e4, v)
```

Valid Conjunctive Patterns

The match table of the conjunctive pattern match is the natural join of all the patterns' match tables. By design, a row in the CPM match table must simultaneously satisfy all the match tables.

If the match tables of the patterns in a FROM clause can be naturally joined into one match table, then the FROM clause has a valid CPM input. Otherwise, the FROM clause has an invalid pattern input list.

For example, below we show two valid CPM inputs and one invalid CPM input.

```
# a valid CPM, since the two patterns natrually join on :tgt
SELECT
FROM Person:p - (KNOWS) - :tgt,
     Post:s -(<LIKES) - :tgt

# a valid CPM, since the two patterns naturally join on :f
SELECT
FROM Person:p - (KNOWS) - :f - (LIKES>) - Post:tgt,
     :f - (LIKES>) - Comment:c

# an invalid CPM, since the two patterns do not share any vertex variable
# they cannot be naturally joined.
# One pattern has (p, tgt); the other has (s, t).
SELECT
FROM Person:p - (KNOWS) - :tgt,
     Post:s - (<LIKES) - Person:t
```

WHERE Clause

The predicates in the WHERE clause can use any of the vertex or edge aliases in any of the patterns, including predicates which combine variables from different constituent paths. CPM queries do not have any special restrictions on the WHERE predicate. Distance matters, however, for performance. Conditions that are local, measured both cross-path and within-path, can be resolved earlier and therefore are faster.

In the example below, `x2.age > x4.age` is a cross-pattern predicate, `e1.timestamp < e3.timestamp` is a cross-pattern predicate, and `x1.gender == x4.gender` is a local predicate of the second pattern.

```
FROM X1:x1-(E1:e1)-X2:x2-(E2:e2)-X3:x3,
     X1:x1-(E3:e3)-X4:x4
WHERE x2.age > x4.age AND e1.timestamp < e3.timestamp AND x1.gender == x4
```

ACCUM Clause

You can ACCUM to any vertex variable in a CPM block.

The ACCUM clause by default will execute ***as many times as*** the row (match) count of the CPM match table; each execution uses one row from the match table.

ACCUM To The Three Vertex Variables of A CPM Pattern

```
#accum to x1, x2 and x4.
FROM X1:x1-(E1:e1)-X2:x2-(E2:e2)-X3:x3,
     X1:x1-(E3:e3)-X4:x4
ACCUM x1.@cnt +=1, x2.@cnt += x3.quantity, x4.@cnt += x3.quantity
```

POST-ACCUM Clause

POST-ACCUM for CPM behaves the same as POST-ACCUM for single path patterns. That is, each POST-ACCUM clause can refer to one vertex alias, and the clause executes iteratively over that vertex set (e.g. one vertex column in the matching table). Its statements can access the aggregated accumulator result computed in the ACCUM clause. The query can have multiple POST-ACCUM clauses, one for each vertex alias you wish to work on. The multiple POST-ACCUM clauses are processed in parallel; it doesn't matter in what order you write them. (For each binding, the statements within a clause are executed in order.)

For example, below we have three POST-ACCUM clauses. The first one iterates through `x1`, and for each x1, we do `@@cnt += x1.@cnt`. The second and third POST-ACCUMs iterate through `x2` and `x3` respectively, and accumulates their `@cnt` accumulator value into `@@cnt`.

POST-ACCUM to a global accumulator @@cnt, using three CPM Vertex Variables

```
FROM X1:x1-(E1:e1)-X2:x2-(E2:e2)-X3:x3,
     X1:x1-(E3:e3)-X4:x4
ACCUM x1.@cnt +=1, x2.@cnt += x3.quantity, x4.@cnt += x3.quantity
POST-ACCUM @@cnt += x1.@cnt
POST-ACCUM @@cnt += x2.@cnt
POST-ACCUm @@cnt += x3.@cnt;
```

Examples

Example 1. Find Viktor Akhiezer's liked messages (100+ days after their creation) whose author's last name begin with letter S. Output the message's forum.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  F  =  SELECT f
        FROM Person:s - (LIKES>:e1) - :msg - (HAS_CREATOR>) - Person:t,
             Forum:f - (CONTAINER_OF>:e2) - :msg
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
              AND t.lastName LIKE "S%"
              AND e1.creationDate >DATETIME_ADD(msg.creationDate, INTERVA

  PRINT F;
}

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"F": [{
    "v_id": "962072688797",
    "attributes": {
      "id": 962072688797,
      "title": "Album 12 of Mario Santos",
      "creationDate": "2011-04-12 09:36:50"
    },
    "v_type": "Forum"
  }]}]
}
```

Example 2. Find any authors who wrote posts that Viktor Akhiezer's liked and whose last name begins with S. Find the country for each of these authors and report on the countries.

```
USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  C  =  SELECT ctry
        FROM Person:s - (LIKES>:e1) - Post:msg - (HAS_CREATOR>) - Person:
             :t - (WORK_AT>:e2) - Company:c,
             :c - (IS_LOCATED_IN>) - Country:ctry
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
              AND t.lastName LIKE "S%" ;

  PRINT C;
}

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"C": [{
    "v_id": "93",
    "attributes": {
      "name": "Portugal",
      "id": 93,
      "url": "http://dbpedia.org/resource/Portugal"
    },
    "v_type": "Country"
  }]}]
}
```

Example 3. Given a TagClass and a Country, find all the Forums created in the given Country, containing at least one Post with Tags belonging directly to the given TagClass. The location of a Forum is identified by the location of the Forum’s moderator.

```
USE GRAPH ldbc_snb

DROP QUERY bi_4

CREATE QUERY bi_4(string tcName, string cName) for graph ldbc_snb syntax
  SetAccum<vertex<Post>> @postSet;
  SumAccum<int> @personId, @postCount;

  ForumSet =
    SELECT f
    FROM Forum:f -(HAS_MODERATOR>)- Person:a -(IS_LOCATED_IN>.IS_PART_OF>
         :f -(CONTAINER_OF>)- Post:p -(HAS_TAG>.HAS_TYPE>)- TagClass:tc
    WHERE c.name == cName and tc.name == tcName
    ACCUM f.@personId = a.id, f.@postSet += p
    POST-ACCUM f.@postCount = f.@postSet.size(), f.@postSet.clear()
    ORDER BY f.@postCount DESC, f.id ASC
    LIMIT 3;

  PRINT ForumSet[ForumSet.id, ForumSet.title, ForumSet.creationDate,
                 ForumSet.@personId, ForumSet.@postCount];
}
```

```
INSTALL QUERY bi_4

RUN QUERY bi_4("MusicalArtist", "Burma")

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"ForumSet": [
    {
      "v_id": "81903",
      "attributes": {
        "ForumSet.title": "Wall of Donald Steele-Perkins",
        "ForumSet.@personId": 5226,
        "ForumSet.id": 81903,
        "ForumSet.@postCount": 65,
        "ForumSet.creationDate": "2010-02-15 06:48:04"
      },
      "v_type": "Forum"
    },
    {
      "v_id": "137438953686",
      "attributes": {
        "ForumSet.title": "Wall of Eric Law-Yone",
        "ForumSet.@personId": 2199023262994,
        "ForumSet.id": 137438953686,
        "ForumSet.@postCount": 65,
        "ForumSet.creationDate": "2010-04-25 22:10:32"
      },
      "v_type": "Forum"
    },
    {
      "v_id": "687194810508",
      "attributes": {
        "ForumSet.title": "Wall of Hector Hugh Michie",
        "ForumSet.@personId": 10995116283784,
        "ForumSet.id": 687194810508,
        "ForumSet.@postCount": 39,
        "ForumSet.creationDate": "2010-12-19 15:33:30"
      },
      "v_type": "Forum"
    }
  ]}]
```

```
}
```

Example 4. For a given country, count all the distinct triples of Persons such that:

- a is a friend of b.
- b is a friend of c
- c is a friend of a.

Distinct means that if a certain 3 vertices appear once in the results, it will not be repeated: it will appear only once. KNOWS is an undirected relationship, so it doesn't matter in what order we list the 3 vertices.

```
USE GRAPH ldbc_snb

CREATE QUERY bi_17(string cName) FOR GRAPH ldbc_snb SYNTAX v2 {
  TYPEDEF TUPLE <uint a, uint b, uint c> triplet;
  SetAccum<triplet> @@tripletSet;
  SumAccum<int> @@tripletCount;

  C =
    SELECT c
    FROM Country:c -(<IS_PART_OF.<IS_LOCATED_IN)- Person:p1,
         :c -(<IS_PART_OF.<IS_LOCATED_IN)- Person:p2,
         :c -(<IS_PART_OF.<IS_LOCATED_IN)- Person:p3,
         :p1 -(KNOWS)- :p2 -(KNOWS)- :p3 -(KNOWS)- :p1
    WHERE c.name == cName AND p1.id < p2.id AND p2.id < p3.id
    ACCUM @@tripletSet += triplet(p1.id, p2.id, p3.id);

  @@tripletCount = @@tripletSet.size();
  @@tripletSet.clear();
  PRINT @@tripletCount;
}


INSTALL QUERY bi_17

RUN QUERY bi_17("Spain")

#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@tripletCount": 242}]
}
```

More Examples. We translated LDBC-SNB ↗ BI and IC queries using CPM, and shared the translation in github. Please refer to the query translation here ↗. Most of the queries are installed as functions, you can find sample parameter(s) of the functions from here ↗.

Source Vertex Set Flexibility

As mentioned when we first described pattern matching, in One-hop patterns, the source (leftmost) vertex set can be a vertex type, an alternation of types, or even omitted.

Example 1. Find Viktor Akhiezer's favorite messages' creators whose last name begins with letter S. Count them.

```
USE GRAPH ldbc_snb

#start from a vertex type "Person"
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  F  =  SELECT t
        FROM Person:s -(LIKES>:e1)- :msg -(HAS_CREATOR>)- Person:t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
              AND t.lastName LIKE "S%"
        POST-ACCUM @@cnt+=1;

  PRINT  @@cnt;

}
#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 8}]
}
```

Example 2. Same query as example 1, but without beginning with vertex types. GSQL compiler can infer the types of :s.

```
USE GRAPH ldbc_snb

#both end points of the pattern do not have vertex types.
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  F  =  SELECT t
        FROM :s -(LIKES>:e1)- :msg -(HAS_CREATOR>)- :t
        WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer" AND t.
        POST-ACCUM @@cnt+=1;

  PRINT  @@cnt;

}
#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 8}]
}
```

Example 3. Count the LIKES edge.

```
USE GRAPH ldbc_snb

# a pattern starts without any information.
INTERPRET QUERY () SYNTAX v2 {

  SumAccum<int> @@cnt;

  F  =  SELECT msg
        FROM  -(LIKES>:e1)- :msg
        ACCUM @@cnt+=1;

  PRINT  @@cnt;

}
#result
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 2190095}]
}
```

Summary

We have covered a lot of territory in GSQL 102:

- Showed how to invoke GSQL Pattern Matching syntax
- Explained how Pattern Matching extends the classic FROM clause grammar:
    - Each hop can be a choice of multiple, individually directed edge types
    - The Kleene star and a min...max range enable each hop to be repeated.
    - GSQL automatically finds the shortest paths that satisfy a variable length path.
    - A virtual match table has a column for each vertex or edge alias in a multi-hop path, and a row for each graph path that satisfies the pattern.
    - The ACCUM clause iterates on each row in the match table.
    - A POST-ACCUM clause iterates on one vertex alias; a query can have multiple POST-ACCUM clauses.
- Described the improvements to Pattern Matching in TigerGraph 3.0:
    - The source (leftmost) vertex set can be specified with the same flexibility as the other vertex sets: a vertex type, an alteration of types, or omitted. Explicit seed sets are no longer needed
    - Restrictions on which vertex aliases may be used in the ACCUM clause have been lifted.
    - Described three major advanced options:
        - The PER <vertex alias> clause enables users to fine tune the ACCUM iteration.
        - Data modification (insert, update, delete) are now supported.
        - Conjunctive Pattern Matching let users express a complex pattern as a set of path patterns which must all be satisfied.
- Provided best practices for writing queries, especially pattern matching queries:
    - Put the smaller vertex set on the left end.
    - Specify all vertex and edge types explicitly.
    - Use the PER clause to reduce the match table size
- Provided numerous examples and the full set of LDBC Social Network benchmark queries.

With a little practice, you will be writing GSQL pattern matching queries to efficiently solve real-world problems. You can post your feedback and questions on the GSQL community forum ↗. Our community members and developers love to hear any feedback from your graph journey of using GSQL and are ready to help clarifying any doubts.

GSQL Demo Examples

This is our our original Get-Started tutorial for 2017. Today, we'd suggest you start with GET STARTED Tutorial or one of our video tutorials.

Using the TigerGraph™ platform is as easy as 1-2-3. In this tutorial we will show you how to use the TigerGraph platform and the GSQL language by developing solutions for several use cases, using the following three-step method:

1. Create a Graph Model for the use case using the GSQL™ language, TigerGraph's high-level graph definition and manipulation language.
2. Load Initial Data : load and transform data to TigerGraph's graph engine.
3. Write a Graph-based Solution by writing queries in the GSQL language.

In addition, this guide will also show you how to update your data: load more data, revise your data, or delete selected data.

Each example involves a data set and simple example of a real-life query or task. We develop a graph model, a loading job to load the data, and one or more queries to answer the question at hand. The applications for graph-based queries are limitless. The goal of these examples is to demonstrate the expressive power of GSQL queries, as well as how business intelligence is a natural fit for the graph analytics world.

We assume the user has a working installation of the TigerGraph system. If you have not installed the system, please refer to the TigerGraph Platform Installation Guide.

This tutorial uses the console-based GSQL Shell. If you prefer to use the browser-based GraphStudio UI, see the TigerGraph GraphStudio UI Guide first. You can then return to this document in learn more about the language itself.

To start the GSQL Shell:
type the command `gsql` to exit, type `exit` or `quit` to run a command file from within the shell, precede the file name with "@":

```
GSQL> @load_demo.gsql
```

You can also run GSQL commands directly from Linux:

- For single-line commands, type "gsql" followed by the command enclosed in single-quotation marks:
- ```
  $ gsql 'RUN QUERY topCopLiked("id1", 5)'
  ```
- For command files, just type "gsql" followed by the filename:
- ```
  $ gsql cf_model.gsql
  ```

The loading jobs have been updated to v2.0 syntax. The output examples have been updated to JSON output API version "v2", which is the default output format for TigerGraph platform version 1.1 or higher.

Common Graph Schema of Demo Examples

The examples in Part 1 of this tutorial have been designed so that all them can be loaded together in one global graph, gsql_demo. This has several benefits:

1. You can quickly load several demo examples by running just one script.
2. After they are loaded, you can switch from one example to another with no delay.
3. The format is modular, so additional examples can be added easily.

If you want to learn how to design your own graph data analyses, we recommend reading and doing Example 1, then Example 2, etc., rather than running the entire batch of examples at once.

Common Applications >

Classic Graph Algorithms

Example 8. Single Pair Shortest Path (unweighted)

The shortest path problem is to find the path(s) between two given vertices S and T in a graph such that the path's total edge weight is minimized. If the edge weights are all the same (e.g., weight=1), then the shortest paths are the ones with the fewest edges or steps from S to T. The classic solution to this graph problem is to start the search from one vertex S and walk one step at a time on the graph until it meets the other input vertex T (unidirectional Breadth-First Search). In addition, we present a more sophisticated way to solve this problem on the TigerGraph advanced graph computing platform. Instead of starting the search from one input vertex, our solution will launch the search agents from both input vertices, walking the graph concurrently until they meet. This greatly improves the algorithm performance. To simplify this problem, this article will assume the graph is undirected and unweighted.

1. Graph Schema and Data

The following examples will use the graph that is presented below. Before we show the algorithms, their implementation and examples, we present the graph schema and data used to create the graph. All files in this document are available here:

16KB DemoExamples_2.0.tar.gz

DemoExamples_2.0.tar.gz

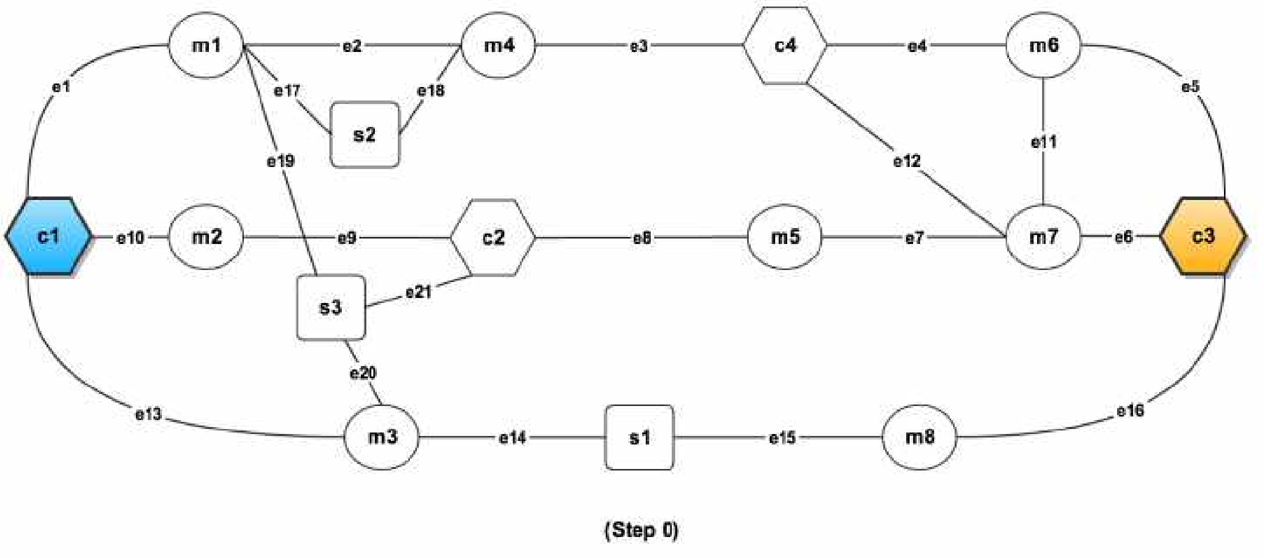


Figure 1: Example Graph used to demonstrate the Shortest Path algorithms.

Graph Schema

First, we give the graph schema. This will create the graph with vertices of type *company* , *persons* and *skill*. It also creates undirected edges that go from *person* to *company*, from *person* to *person*, from any type to *skill*, and from any type to *company*.

Graph Schema

```
DROP ALL
CREATE VERTEX company (PRIMARY_ID companyId string, id string, company_na
CREATE VERTEX persons (PRIMARY_ID pId string, id string, profileId string
CREATE VERTEX skill (PRIMARY_ID skillId uint, id string)
CREATE UNDIRECTED EDGE person_work_company (FROM persons, TO company, pos
CREATE UNDIRECTED EDGE person_person (FROM persons, TO persons, connect_t
CREATE UNDIRECTED EDGE all_to_skill (FROM *, TO skill)
CREATE UNDIRECTED EDGE all_to_company (FROM *, TO company)
CREATE GRAPH work_graph(*)
```

Data Set

Data source for *company* vertices.

small_companies

```
c1,1,com1,us
c2,0,com2,jp
c3,1,com3,de
c4,0,com4,cn
```

Data source for *person* vertices and *skill* vertices. The first line,

`m1,i1,0,"s2|s3"`

means that person m1 has skills s2 and s3.

small_persons

```
m1,i1,0,"s2|s3"
m2,i1,1,""
m3,i1,2,"s1|s3"
m4,i1,3,"s2"
m5,i2,4,""
m6,i2,5,""
m7,i2,6,""
m8,i3,7,"s1"
```

Data source for *person_work_company* edges. The first line means that person m1 works for company c1.

small_person_company

```
m1,c1,1,1,1
m2,c1,2,1,3
m2,c2,2,1,4
m3,c1,2,1,5
m4,c4,2,2,6
m5,c2,3,2,7
m6,c3,3,2,8
m6,c4,3,2,9
m7,c3,3,2,10
m7,c4,3,2,11
m8,c3,3,3,12
```

Data source for *person_person* edges.

small_person_person

```
m1,m4,1
m6,m7,3
m7,m5,4
```

Data source for *all_to_skill* edges such as *all_to_skill* (m1, s2) or *all_to_skill* (c2, s3). While the schema supports *all_to_company* edges, this particular data set does not use any..

small_all_to_all

```
s,m,m1,s2,0
s,m,m4,s2,0
s,m,m1,s3,0
s,m,m3,s3,0
s,c,c2,s3,0
s,m,m3,s1,1
s,m,m8,s1,1
```

Loading the Data

To load all of this data into the graph, we can use the following GSQL command file (which also includes the graph schema creation commands).

graph_create.gsql

```
DROP ALL
CREATE VERTEX company (PRIMARY_ID companyId string, id string, company_na
CREATE VERTEX persons (PRIMARY_ID pId string, id string, profileId string
CREATE VERTEX skill (PRIMARY_ID skillId uint, id string)
CREATE UNDIRECTED EDGE person_work_company (FROM persons, TO company, pos
CREATE UNDIRECTED EDGE person_person (FROM persons, TO persons, connect_t
CREATE UNDIRECTED EDGE all_to_skill (FROM *, TO skill)
CREATE UNDIRECTED EDGE all_to_company (FROM *, TO company)
CREATE GRAPH work_graph(*)

USE GRAPH work_graph
SET sys.data_root="./"
CREATE LOADING JOB load_data FOR GRAPH work_graph {
    LOAD "$sys.data_root/small_companies"
        TO VERTEX company VALUES ($0, $0, $2)
        USING HEADER="false", SEPARATOR=",", QUOTE="double";
    LOAD "$sys.data_root/small_persons"
        TO VERTEX persons VALUES ($0, $0, $1, $2)
            WHERE to_int($2) >= 0
        USING HEADER="false", SEPARATOR=",", QUOTE="double";
# Example of flattening a multi-valued field
    LOAD "$sys.data_root/small_persons"
        TO temp_table member_skill_table (memberID, skillID)
            VALUES ($0, flatten($3, "|", 1))
        USING HEADER="false", SEPARATOR=",", QUOTE="double";
    LOAD temp_table member_skill_table
        TO VERTEX skill VALUES ($"skillID", $"skillID");

    LOAD "$sys.data_root/small_person_company"
        TO EDGE person_work_company VALUES($0, $1, $2, $3, $4)
            WHERE to_int($4) >= 0
        USING HEADER="false", SEPARATOR=",",  QUOTE="double";
    LOAD "$sys.data_root/small_person_person"
        TO EDGE person_person VALUES($0, $1, $2)
            WHERE to_int($2) >= 0
        USING HEADER="false", SEPARATOR=",", QUOTE="double";

# Note how $0 and $1 indicate what type of data is in $3 and $2, respecti
# so that the VALUES $2 and $3 can explicitly state the data type.
    LOAD "$sys.data_root/small_all_to_all"
        TO EDGE all_to_skill VALUES ($2 company, $3 skill)
            WHERE $0 == "s" AND $1 == "c",
        TO EDGE all_to_skill VALUES ($2 persons, $3 skill)
            WHERE $0 == "s" AND $1 == "m",
        TO EDGE all_to_skill VALUES ($2 skill, $3 skill)
            WHERE $0 == "s" AND $1 == "s",
        TO EDGE all_to_company VALUES ($2 company, $3 company)
            WHERE $0 == "c" AND $1 == "c",
```

```
        TO EDGE all_to_company VALUES ($2 persons, $3 company)
            WHERE $0 == "c" AND $1 == "m",
        TO EDGE all_to_company VALUES ($2 skill, $3 company)
            WHERE $0 == "c" AND $1 == "s"
        USING HEADER="false", SEPARATOR=",", QUOTE="double";
}
RUN LOADING JOB load_data
```

To run a command file, simply enter gsql *name_of_file*

Create Graph and Load Data

```
> gsql graph_create.gsql
```

2. Unidirectional (BFS) Algorithm

If the edges are unweighted, then the shortest path can be found using the classic Breadth-First Search (BFS) algorithm. Below is an implementation in the GSQL Query Language:

shortest_path_1D.gsql (v2.0)

```
/**
 * This query assumes every edge in the graph is undirected.
 * It uses breadth-first-search to find the shortest path between s and t
 */
// 1 May 2018: v2.0 - ListAccum "+" behavior changed.  Need to use FOREAC

CREATE QUERY shortest_path_1D (VERTEX<company> S, VERTEX<company> T, INT

  OrAccum @@found  = false;
  OrAccum @notSeen = true;
  ListAccum<STRING>  @pathResult;
  Start (ANY) = {S};
  Start = SELECT v
    FROM Start:v
    //assume each vertex has an id attribute
    ACCUM v.@notSeen = false, v.@pathResult = v.id;

  WHILE NOT @@found LIMIT maxDepth DO
    Start = SELECT v
      FROM Start - (:e) -> :v
      WHERE v.@notSeen
      ACCUM v.@notSeen = false,
            //add partial result paths to target v. v2.0 ListAccum requir
            FOREACH path IN Start.@pathResult DO
                v.@pathResult += (path + "-" + v.id)
            END,
            CASE WHEN v == T
              THEN @@found += true
            END;
  END;

  IF @@found THEN
    Result = {T};
    #PRINT Result.@pathResult;          #JSON output API version v1
    PRINT Result [Result.@pathResult]; #JSON output API version v2
  ELSE
    PRINT "Can't find shortest path within max steps";
  END;
}
INSTALL QUERY shortest_path_1D
```

The algorithm works by expanding the search path through all vertices that were seen in the previous step. Each step is taken by one iteration of the WHILE loop. In the first iteration of the WHILE loop, we start at vertex S and travel to all its neighbors. In each of the following iterations, we travel from previously reached vertices to their neighbors that have not already been seen by the path.

To install the query, run the following command:

Install Query

```
> gsql -g work_graph shortest_path_1D.gsql
```

Example of Unidirectional BFS Search

Let us show a running example of this algorithm. We will be trying to find the shortest path from c1 to c3. First, we have our initial graph, where we have not traveled along any edges yet.

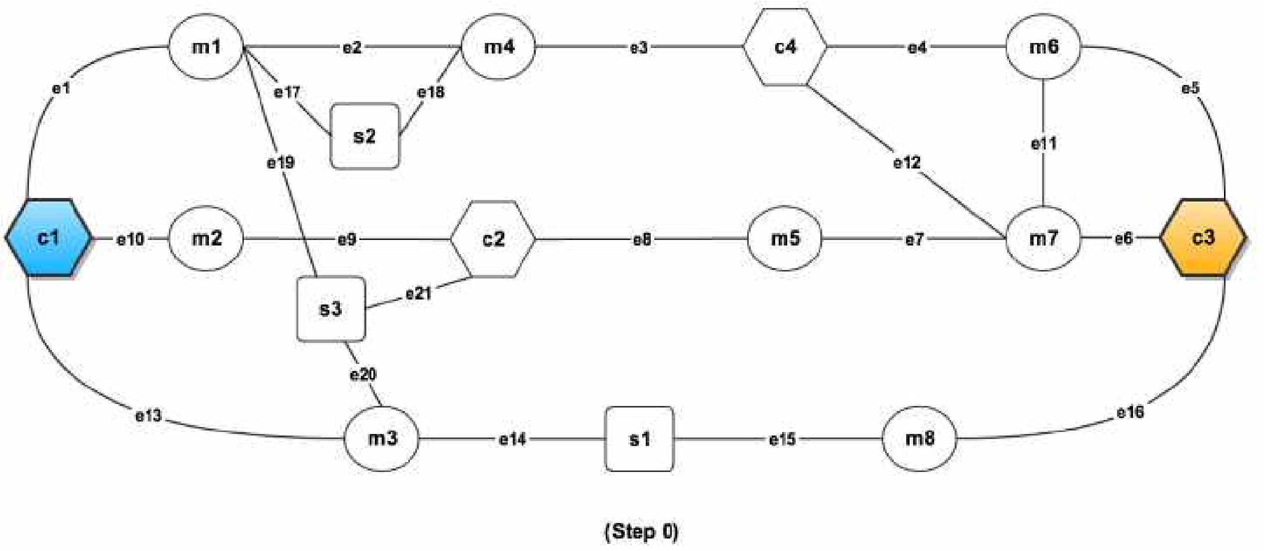


Figure 2: The starting state for our graph. From here, we go on to the first step of the algorithm. We start at c1, and go along each of its edges.

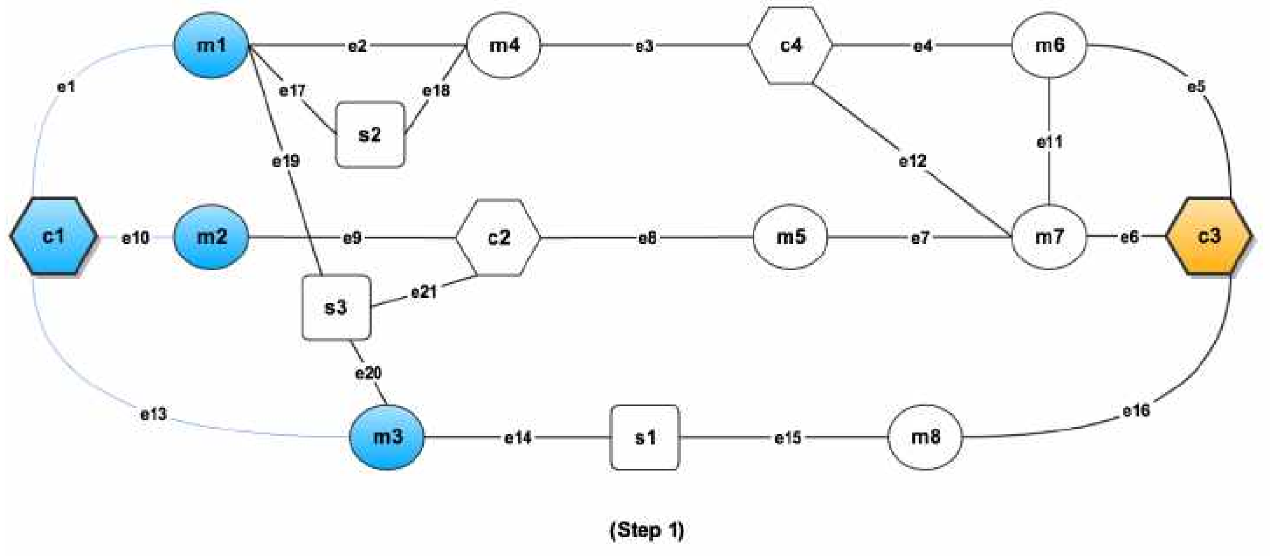


Figure 3: This is the graph after one step. We have traveled from c1 to all of its neighbors, labeling them as visited. For each one that we visit, we update its @pathResult accumulator value in order to keep track of our path as we traverse the graph.

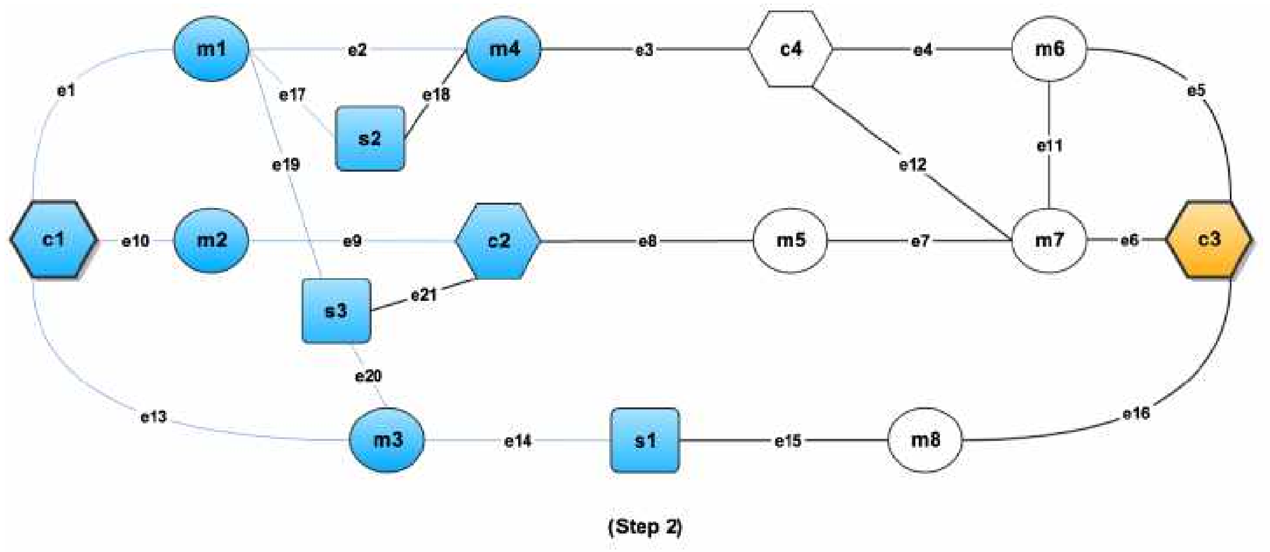


Figure 4: This graph shows where we have traveled after two steps. We traveled to our new vertices s1, s2, s3, c2 and m4 by traveling one edge away from the nodes that we had visited in step 1. Note that the blue edges also tell us how we can get from c1 to a vertex. For example, we notice that e21 is not labeled blue. This means that we did not travel along this edge. That is, we must have gotten to c2 using a different edge. Indeed, we can see that the path c1-m2-c2 is shorter than c1-m3-s3-c2. This explains why e9 is blue, but e21 is not.

Each time that the query travels from a starting vertex (m1, m2, or m3) to a target vertex (s1, s2, s3, c3, or m4), the target vertex's @pathResult ListAccum<string> is updated (Line 22 of the query). A new string is added to the list (the += operator), which means that there is a path string for each time that the target vertex is reached. The path string consists of the path string from the source vertex, followed by this target vertex. That is equivalent to the path from the query's starting vertex (e.g., c1) to the current target vertex.

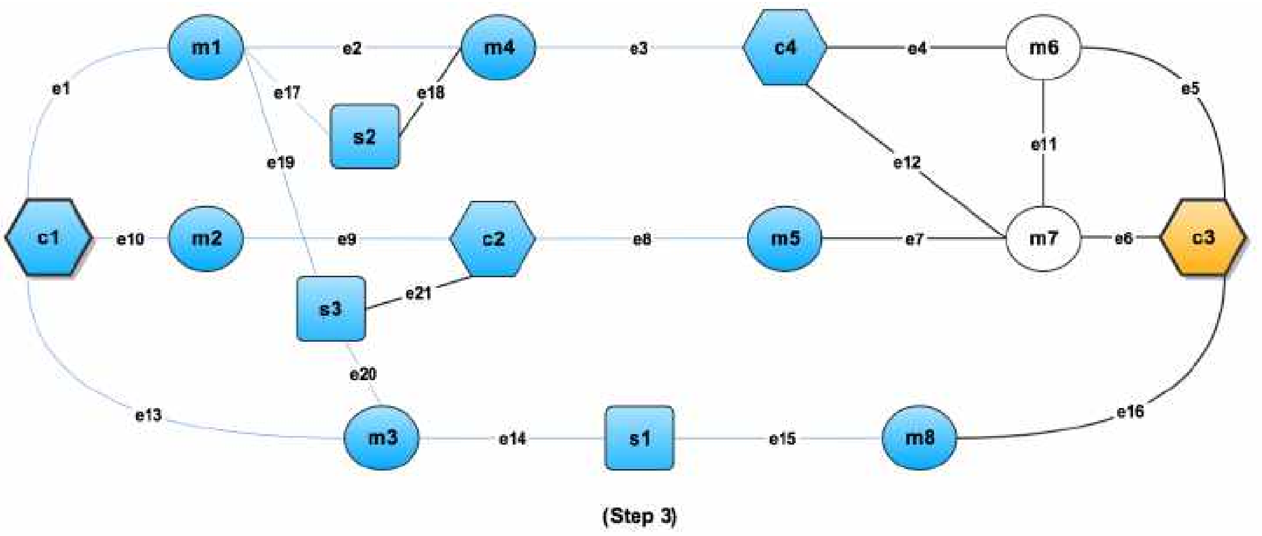


Figure 5: At the third step of our algorithm, we have reached the nodes m8, m5 and c4. We got here by moving one edge away from the vertices that we reached in step 2.

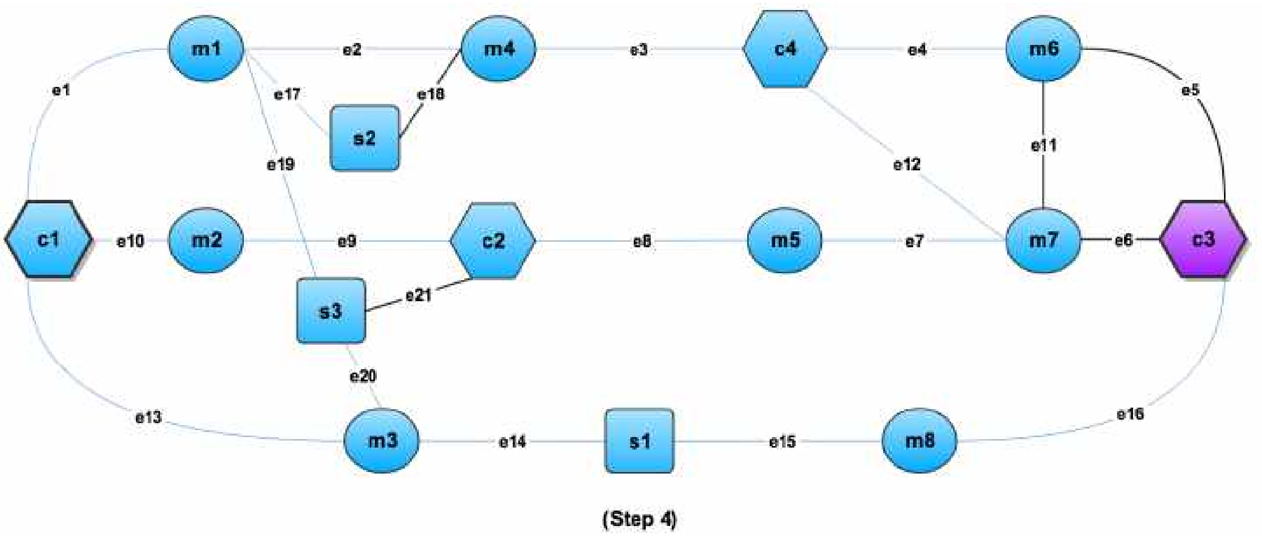


Figure 6: Finally, we have reached the end of our algorithm. Note that when we travel one edge away from m8, we arrive at our target node of c3. Working

backwards, we can reconstruct the shortest path. We reached c3 from m8, m8 from s1, s1 from m3 and m3 from c1. Thus, we get that the shortest path is indeed c1-m3-s1-m8-c3.

To run the query with starting vertex c1, ending vertex c3, and a maximum distance of 10:

Query

```
> gsql -g work_graph 'RUN QUERY shortest_path_1D("c1","c3",10)'
```

This will give the following result.

Results


```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "c3",
    "attributes": {"Result.@pathResult": ["c1-m3-s1-m8-c3"]},
    "v_type": "company"
  }]}]
}
```


As we can see, the algorithm tells us that the shortest path from c1 to c3 is going through m3, followed by s1, then m8, then finally arriving at c3. However, this result also tells us that this is the unique shortest path. Indeed, if we instead run:

Multiple Shortest Paths Query

```
> gsql -g work_graph 'RUN QUERY shortest_path_1D("c3", "c4", 10)'
```

Our results are:

Multiple Shortest Paths Results

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "c4",
    "attributes": {"Result.@pathResult": [
      "c3-m6-c4",
      "c3-m7-c4"
    ]},
    "v_type": "company"
  }]}]
}
```


Note that here we have two paths. The first is from c3 to m6, and then to c4. The other path is from c3, to m7, to c4. We are presented with both paths because each of these consists of the least possible weight: exactly two edges. As explained earlier, this is because we arrive at a vertex at the same time through two different paths. When we started at c3, we traveled to m6, m7 and m8. At the second step, both m6 and m7 arrive at c4 at the exact same time. That means that two path strings will be written to c4.@queryResult, recording two shortest paths.

3. Bi-Directional Shortest Path Search Algorithm

Bi-Directional search will launch two search agents, each from a given vertex. The two agents concurrently walk one step at a time, until they meet at an intermediate vertex. The shortest path length may be odd or even. For example, in Figure 7 below, Case II is an even-length case, and Case III is an odd-length case. Case I is a special case of an odd-length path.

The core of this solution is that in each step, a set of previously unvisited vertices will be discovered by the search frontiers of S and T. The newly visited vertices will become the new frontier of S or T. The algorithm will repeat this process until the frontiers of the two agents meet.

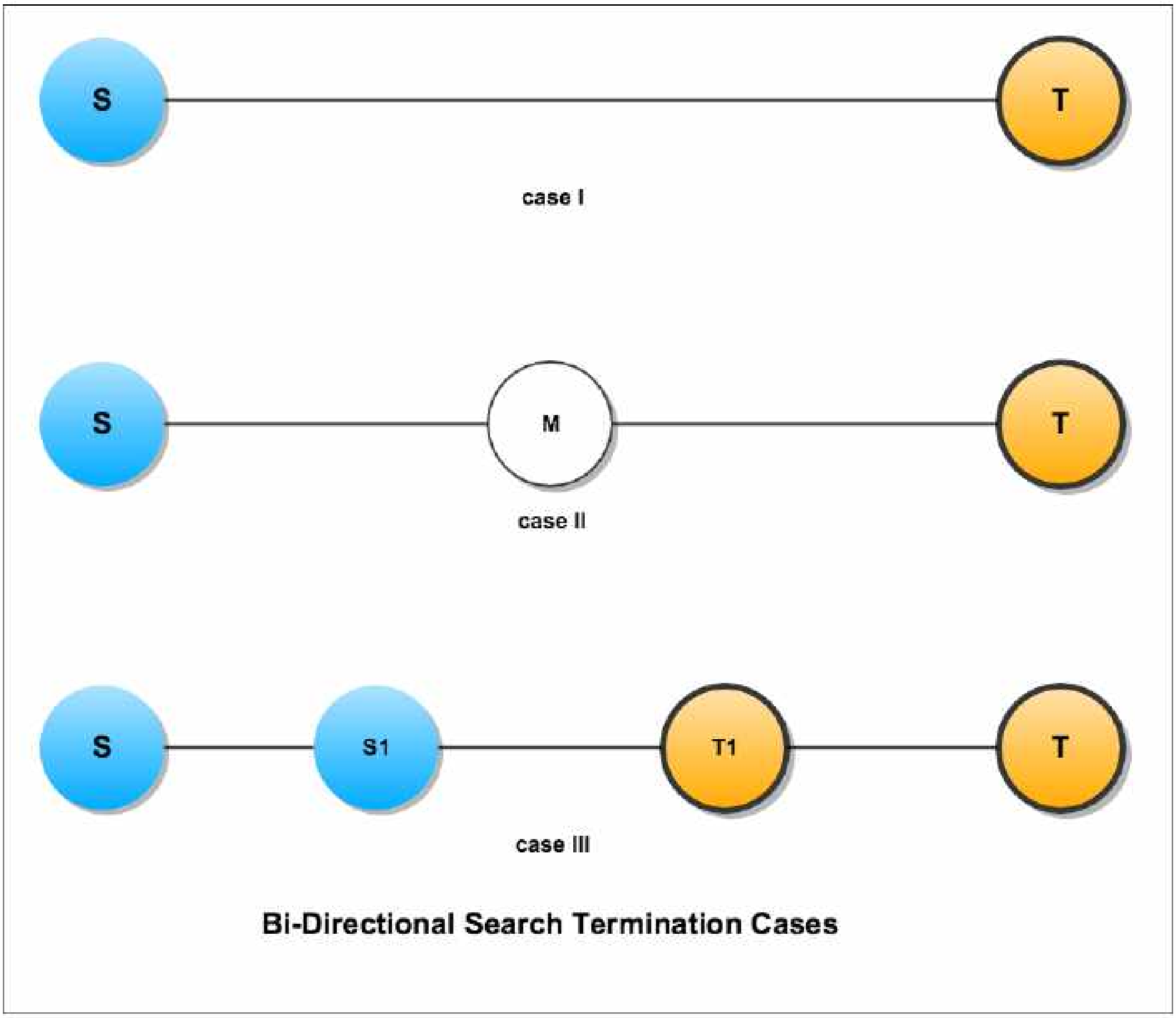


Figure 7 : Three cases for terminating a bi-directional path search.

Because this algorithm is more complicated than one directional search, we first give pseudocode to help explain the algorithm.

bi-directional shortest path search algorithm

```
void find_shortest_path_bi_directional_search(Vertex S, Vertex T) {
    bool stop = false;
    vertex.pathFromS = "";

    vertex.pathFromT = "";
    vertex.visitedByS = false;
    vertex.visitedByT = false;
    final_path = "";

    Activate vertex S and T as the starting vertices;
    S.visitedByS = true;
    T.visitedByT = true;

    // VERTEX GROUP, if a vertex V is visited by a vertex originating fro
    // then V is part of vertex group T.  The vertices who have the longe
    // from its origin vertex are called the FRONTIER of the vertex group
    // Initially, S is the frontier and only member of vertex group S,
    // and T is the frontier and only member of vertex group T.
    while (!stop) {
        VS = (frontier of Vertex Group S) union (frontier of Vertex Group
        for each vertex v in VS {
            for each neighbor vertex b of v {
                if ((v.visitedByS && b.visitedByT) || (v.visitedByT && b.
                    // If the frontiers of S and T are neighbors (Case II
                    if (v.visitedByS) {
                        final_path = v.pathFromS + v.ID + b.ID + b.pathFr
                    }
                    if (v.visitedByT) {
                        final_path = v.pathFromT + v.ID + b.ID + b.pathFr
                    }
                    stop = true;
                    break;
                } else if ((v.visitedByS && not b.visitedByS) || (v.visit
                    // If b (the neighbor of v) is not yet part of v's ve
                    // then add b to the vertex group, and update b's pat
                    if (v.visitedByS)  {
                        b.visitedByS = true;
                        b.pathFromS = v.pathFromS + v.ID;
                    }
                    if (v.visitedByT) {
                        b.visitedByT = true;
                        b.pathFromT = v.pathFromT + v.ID;
                    }
                }
            }

            // if a vertex is visited by S & T in the same iteration (Cas
            if (v.visitedByT && v.visitedByS) {
```

```
                final_path = v.pathFromS + v.ID + v.pathFromT;
                stop = true;
                break;
            }
        }
    }
    print out final_path;
}
```

This algorithm essentially works by running two versions of the algorithm from the first example at the same time, just with different starting vertices. The algorithm continues with these two paths until there is an intersection. Once the two paths cross, we know that the shortest path goes through this intersection, as explained in the previous section.

Below is an implementation in the GSQL Query Language.

shortest_path_2D.gsql (v2.0)

```
// 1 May 2018: v2.0 - ListAccum "+" behavior changed.  Need to use FOREAC
CREATE QUERY shortest_path_2D (VERTEX<company> S, VERTEX<company> T , INT
{
  OrAccum @@stop = false;       // global variable controlling whether to
  OrAccum  @seenS = false;      // a vertex has been seen by S
  OrAccum  @seenT = false;      // a vertex has been seen by T
  OrAccum @meet = false;        // vertex flag indicating whether it is '
  SumAccum<int>   @sLength = 0;  // vertex runtime attribute: # steps fro
  SumAccum<int>   @tLength = 0;  // vertex runtime attribute: # steps fro
  SumAccum<int>   @resultLength = 0; // the final length of shortest path
  ListAccum<string>  @pathS;  //list of paths so far from S
  ListAccum<string>  @pathT;  //list of paths so far from T
  ListAccum<string>  @pathResults; //final set of shortest paths

  Start = {S,T};

  //initialize S, T vertices
  StartSet (ANY) = SELECT v         // _ means StartSet can contain any v
              FROM Start:v
              ACCUM CASE WHEN v==S THEN v.@seenS=true, v.@pathS +=  ""
                         WHEN v==T THEN v.@seenT=true, v.@pathT +=  ""
                    END;
  WHILE @@stop == false LIMIT maxDepth DO
    StartSet  = SELECT v
        // Consider each edge from S or T's frontier (u) to outside (v),
        // i.e., each edge that moves "out" from the frontier.
        // Note how StartSet is updated to be v (pushing the frontier for
        FROM StartSet:u-(:e)->:v
        WHERE ((u.@seenS==true AND v.@seenS!=true) OR // from S frontier
               (u.@seenT==true AND v.@seenT!=true))   // from T frontier
        ACCUM
            // If u->v joins the S and T frontiers, an odd-length path is
            CASE WHEN ((u.@seenS == true AND v.@seenT == true) OR
                       (u.@seenT == true AND v.@seenS == true))
                THEN @@stop += true,
                    // we don't want to print the results twice
                    // v.@pathResults  stores all shortest paths
                    // between S and T where v is in the middle of
                    // every such path.
                    // only need to print out the result once, see above
                    CASE WHEN (u.@seenS == true AND v.@seenT == true)
                        THEN
                            STRING joiner = u.id + "-" + v.id + "-",
                            FOREACH pathS IN u.@pathS DO
                                FOREACH pathT in v.@pathT DO
                                    v.@pathResults += pathS + joiner + pa
                                END
                            END,
```

```
                              v.@meet = true,
                              v.@resultLength =  u.@sLength + v.@tLength +
                          END
              // Else, since u->v does not complete a path, move the fronti
              // If u is in S's frontier, then extend S's frontier to v. As
              WHEN u.@seenS == true
                  THEN v.@seenS = true,
                      FOREACH uPath IN u.@pathS DO
                          v.@pathS += uPath + ( u.id + "-")
                      END,
                      v.@sLength = u.@sLength + 1
              // If u is in T's frontier, then extend T's frontier to v. As
              WHEN  u.@seenT == true
                  THEN v.@seenT =true,
                      FOREACH uPath IN u.@pathT DO
                          v.@pathT += (u.id + "-") + uPath
                      END,
                      v.@tLength = u.@tLength + 1
              END
          POST-ACCUM
              // If the two frontiers meet at v, an even-length path is fou
              CASE WHEN (v.@seenS == true AND v.@seenT == true AND @@stop =
                  THEN @@stop += true,
                      // Insert v.id between the source paths and the targe
                      FOREACH pathS in v.@pathS DO
                          FOREACH pathT in v.@pathT DO
                              v.@pathResults += pathS + v.id + "-" + pathT
                          END
                      END,
                      v.@resultLength = v.@sLength + v.@tLength,
                      v.@meet = true
              END;
  END;

  // print out the final result stored at the vertex who marked
  // as meet vertex
  StartSet = SELECT v
             FROM StartSet:v
             WHERE v.@meet == true;

  #PRINT StartSet.@resultLength, StartSet.@pathResults;           # JSON
  PRINT StartSet [StartSet.@resultLength, StartSet.@pathResults]; # JSON
}
INSTALL QUERY shortest_path_2D
```

Example of Bidirectional BFS Search

The following is a running example to demonstrate the algorithm of finding the shortest path in a bi-directional way. The graph below (Figure 8) shows vertices c1 and c3, with several other vertices between them. The algorithm will demonstrate the two search directions by using two different colors and border thicknesses:

- Blue and thin border for c1's search frontier
- Orange and thick border for c3's search frontier

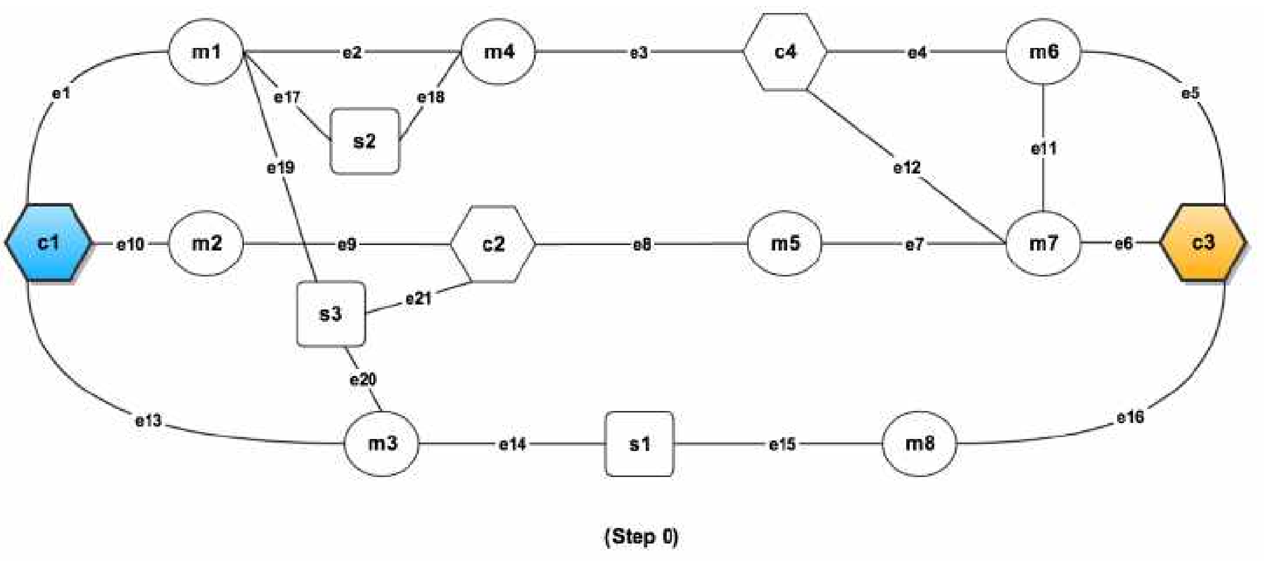


Figure 8: Initialization - prepare to start the search process. The two given vertices (c1 and c3) are activated and colored as Blue and Orange respectively. The rest of the graph remains untouched.

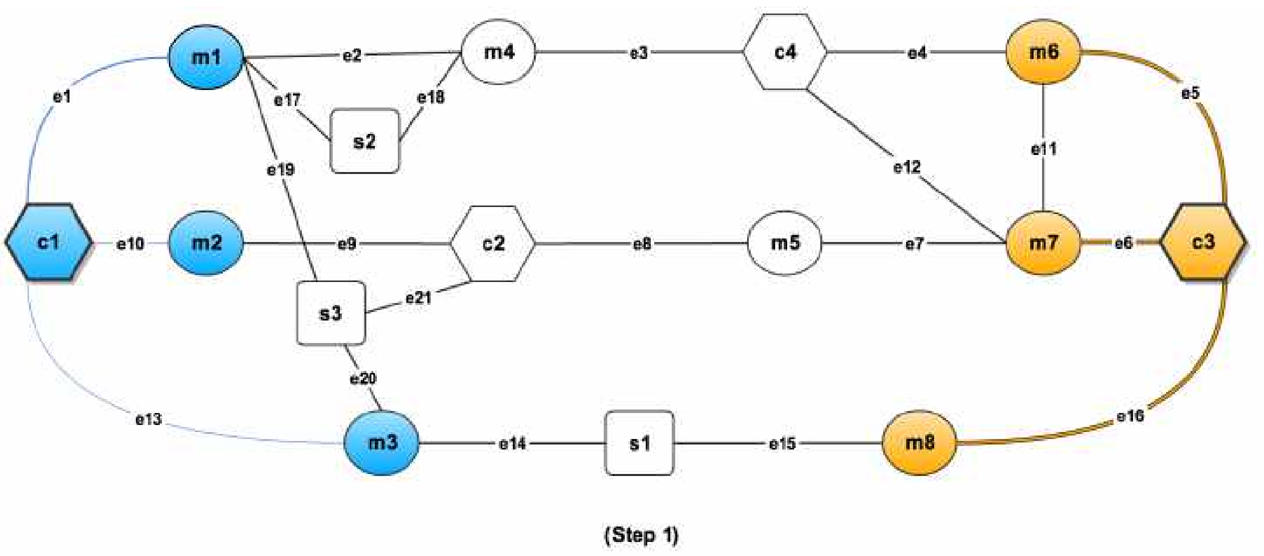


Figure 9: The graph after the first step. The search process starts simultaneously from c1 and c3. If a vertex is seen by the agent starting from c1 (c3), we will say it is seen by c1 (c3).

- From the vertex c1, the algorithm goes to the neighbors of c1 that have not yet been seen. As a result, the unseen vertices m1, m2 and m3 are discovered and become the frontier of c1's vertex group.
- From the vertex c3, in a similar fashion, the vertices m5, m7 and m8 are discovered and become the frontier of c3's vertex group.

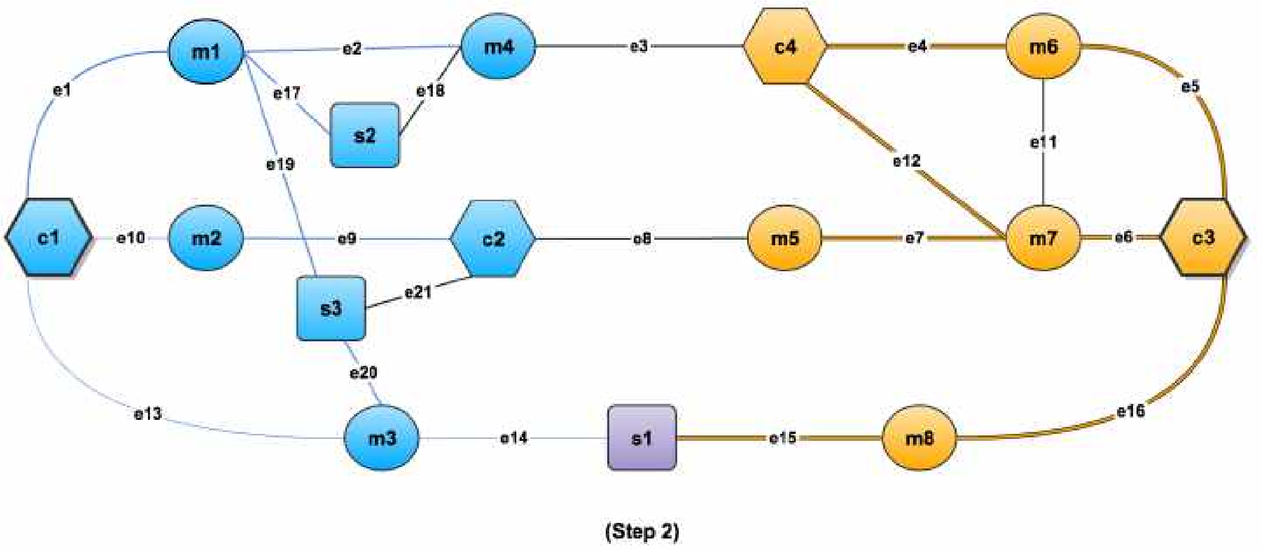


Figure 10: As the two groups have not been met yet, the search process continues.

- From c1's search agent, the vertices m4, s2, c2, s3 and s1 are all discovered.
- From c3's search agent, the vertices c4, m5 and s1 are all discovered.

Notice that both search agents have found the vertex s1. Thus, the algorithm should stop, and return the path going through s1. In this case, this path is c1-m3-s1-m8-c3.

In order to get this result in the TigerGraph Query Language (GSQL), first install the query, for which the code was given earlier.

Install Query

```
> gsql -g work_graph shortest_path_2D.gsql
```

Now, run the query using c1 as a starting node, c3 as the ending node, and a maximum distance of 10:

Query

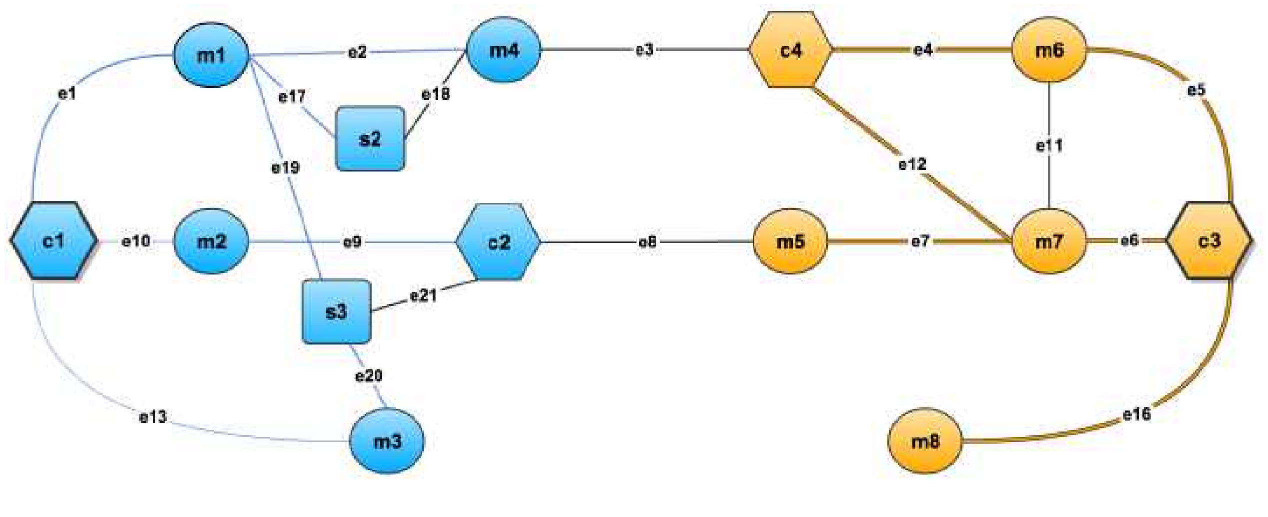
```
> gsql -g work_graph 'RUN QUERY shortest_path_2D("c1","c3",10)'
```

This will return the following result:

Results

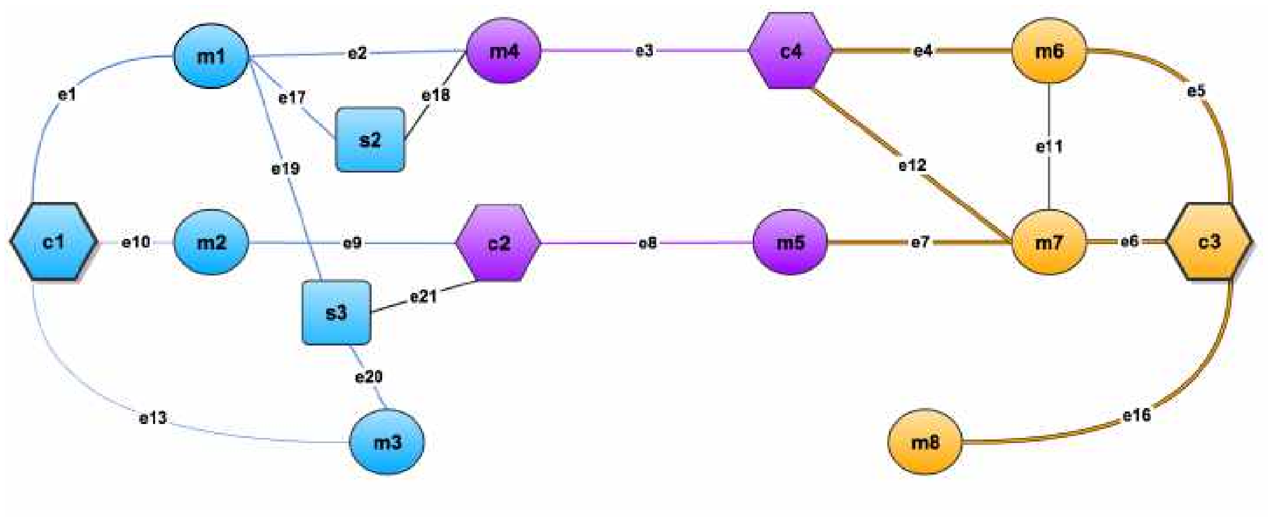
```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "api": "v2"
  },
  "results": [{"StartSet": [{
    "v_id": "s1",
    "attributes": {
      "StartSet.@pathResults": ["c1-m3-s1-m8-c3-"],
      "StartSet.@resultLength": 4
    },
    "v_type": "skill"
  }]}]
}
```

However, in order to demonstrate the odd-length case, assume that s1 does not exist.



(Alternative Step 2)

Figure 11: 2nd Iteration in a modified graph in which s1 does not exist. We got here by traveling one edge away form the vertices that were visited in the previous step. However, as we do not yet have a crossing, we must complete one more iteration.



(Alternative Step 3)

Figure 12: Here, the paths from c1 have finally found a vertex that was previously found by the paths from c3 (and vice versa). That is, the blue paths traveled from c2 to m5 and from m4 to c4. In Figure 11, m5 and c4 were both orange. In Figure 12, we change a vertex's color to purple when one frontier meets the other. This tells us that the shortest path from c1 to c3 either goes through e8 or e3. If we go through e8, we go along the path c1-m2-c2-m5-m7-c3. Note that if we go through e3, we are given two paths. This is almost identical to the multiple path example from the first algorithm. From c4, we can either take e4 or e12 to get to c3. Thus,

when going from c1 to c3 through e3, we are actually given two paths. These paths are c1-m1-m4-c4-m6-c3 and c1-m1-m4-c4-m7-c3.

The * operator in Lines 41 and 63 handle the case of multiple paths from one direction merging with multiple paths from the other direction. For example, we know there are two shortest paths from c4 to c3. Pretend for a moment that there are 3 shortest paths from c1 to m4. Then, when m4 and c4 meet, there would then be (3 ∗ 2) = 6 shortest paths from c1 to c3.

Once again, we can implement this alternate graph in GSQL by using the DELETE keyword. First, we delete the vertex s1 from the graph by doing the following:

Remove "s1"

```
> gsql -g work_graph 'DELETE FROM skill WHERE primary_id=="s1"'
```

Now, we can run our query once again:

Query

```
> gsql -g work_graph 'RUN QUERY shortest_path_2D("c1","c3",10)'
```

Notice that this time, we are given the three paths that we previously described.

Results

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "api": "v2"
  },
  "results": [{"StartSet": [
    {
      "v_id": "m5",
      "attributes": {
        "StartSet.@pathResults": ["c1-m2-c2-m5-m7-c3-"],
        "StartSet.@resultLength": 5
      },
      "v_type": "persons"
    },
    {
      "v_id": "c4",
      "attributes": {
        "StartSet.@pathResults": [
          "c1-m1-m4-c4-m6-c3-",
          "c1-m1-m4-c4-m7-c3-"
        ],
        "StartSet.@resultLength": 5
      },
      "v_type": "company"
    }
  ]}]
}
```

Common Applications

16KB DemoExamples_2.0.tar.gz

DemoExamples_2.0.tar.gz

Example 1. Collaborative Filtering

Here is an observation about social networks: If a set of persons likes me, and many of them also like another person Z, it is probably true that person Z and I have some things in common. The same observation works for products and services: if a set of customers likes product X, and many of them also like product Z, then product X and Z probably have something in common. We say X and Z are "co-liked". This observation can be turned around into a search for recommendations: Given a user X, find a set of of users Z which are highly co-liked. For social networks, this can be used as friend recommendation: find a highly co-liked person Z to introduce to X. For e-commerce, this can be used for purchase recommendation: someone who bought X may also be interested in buying Z. This technique of finding the top co-liked individuals is called collaborative filtering.

A graph analytics approach is a natural fit for collaborative filtering because the original problem is in a graph (social network), and the search criteria can easily be expressed as a path in the graph. We first find all people Y who like user X, then find other users Z who are liked by someone in group Y, and rank members of Z according to how many times they're liked by Y.

Figure 1 below shows a simple graph according to our model. The circles represent three User vertices with id values id1, id2, and id3. There are two directed edges labeled "Liked" which show that User id2 likes id1, and id2 also likes id3. (In this model, friendship is directional because in online social networks, one of the two persons initiates the friendship.) There are two more directed edges in the opposite directions labeled "Liked_By". Since id2 likes both id1 and id3, id1 and id3 are co-liked.

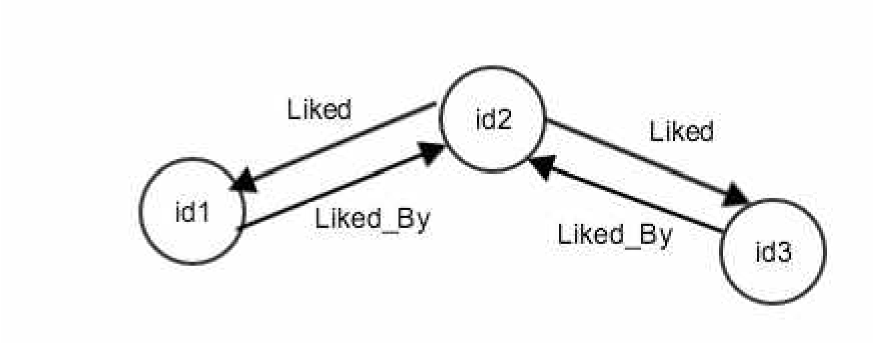


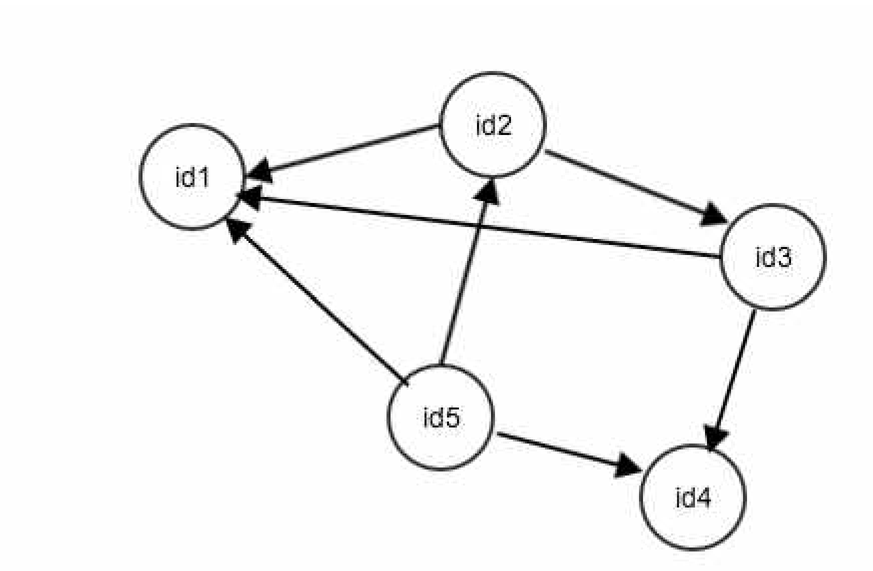
Figure 1 - Example graph for the collaborative filtering model

Quick Demo

To just see the basic operation of the TigerGraph system, follow the easy instructions below . You can then continue to read for the explanation of the command files so you can learn to design your own examples.

Quick Demo Instructions

This example uses the graph below and asks the following query: "Who are the top co-liked persons of id1"?



Step 1: Obtain the data and command files. Create a graph model.

This example uses 4 small files: 3 command files (cf_model.gsql , cf_load.gsql , cf_query.gsql) and one data file (cf_data.csv) . Their contents are shown below, so you can either copy from this document or download the files (look in the "cf" subfolder of Examples.zip)

```
> gsql 'DROP ALL'
> gsql cf_model.gsql
> gsql 'CREATE GRAPH gsql_demo(*)'
```

Step 2: Load data:

The command below loads our new data.

```
> gsql -g gsql_demo cf_load.gsql
```

Step 3: Install and execute the query:

The file cf_query.gsql creates a query called topCoLiked. Then we install the query. The creation step runs fast, but the installation (compiling) step may take about 1 minute. We then run the query, asking for the top 20 Users who are co-liked with User id1.

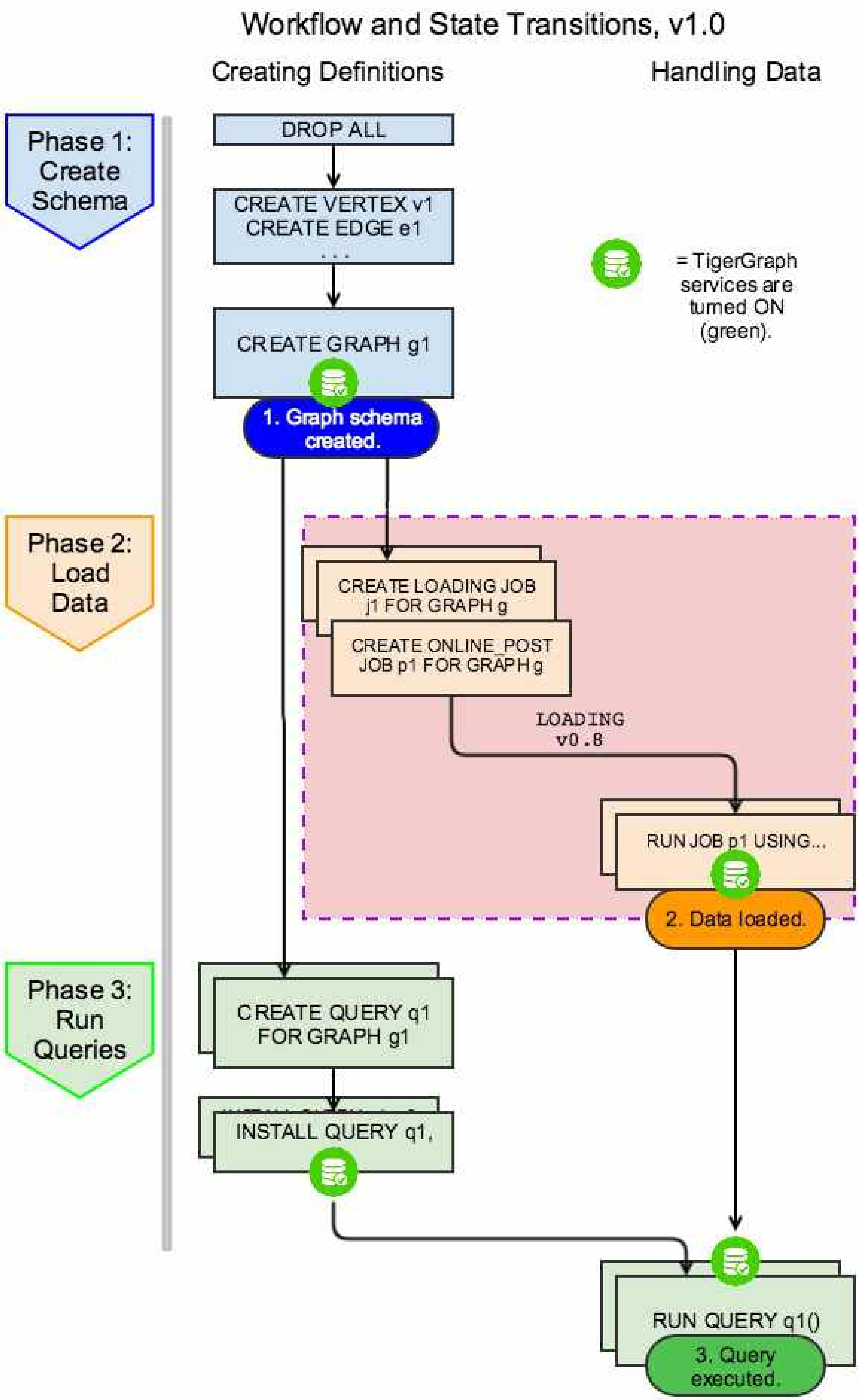
```
> gsql -g gsql_demo cf_query.gsql
> gsql -g gsql_demo 'INSTALL QUERY topCoLiked'
> gsql -g gsql_demo 'RUN QUERY topCoLiked("id1", 20)'
```

The query results should be the following. Interpretation: id4 has as score (@cnt) = 2, which means there are two persons who like both id1 and id4. Next, id2 and id3 each have 1 co-friend in common with id1.

```
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"L2": [
    {
      "v_id": "id4",
      "attributes": {"@cnt": 2},
      "v_type": "User"
    },
    {
      "v_id": "id3",
      "attributes": {"@cnt": 1},
      "v_type": "User"
    },
    {
      "v_id": "id2",
      "attributes": {"@cnt": 1},
      "v_type": "User"
    }
  ]}]
}
```


We now begin a tutorial-style explanation of this TigerGraph example and the workflow in general.

The figure below outlines the steps to progress from an empty graph to a query solution. Each of the blocks below corresponds to one of the steps in the Quick Demo above. The tutorial below will give you a deeper understanding of each step, so you can learn how it works and so you can design your own graph solutions.


Workflow and State Transitions, v1.0
Creating Definitions
Handling Data
Phase 1: Create Schema
DROP ALL
CREATE VERTEX v1
CREATE EDGE e1
. . .
CREATE GRAPH g1
1. Graph schema created.
= TigerGraph services are turned ON (green).
Phase 2: Load Data
CREATE LOADING JOB j1 FOR GRAPH g
CREATE ONLINE_POST JOB p1 FOR GRAPH g
LOADING v0.8
RUN JOB p1 USING...
2. Data loaded.
Phase 3: Run Queries
CREATE QUERY q1 FOR GRAPH g1
INSTALL QUERY q1,
RUN QUERY q1()
3. Query executed.

Step 1: Create a graph model.

The first step is to create a model for your data which describes the types of vertices and edges you will have.

> This example is written to be compatible with older TigerGraph platforms which support only one graph model at a time (though the user can make the model simple or complex, to handle multiple needs). To clear an existing model and old data, so you can install a new one, run the `DROP ALL` command.

The statements below describe the vertex types and edge types in our Co-Liked model

```
CREATE VERTEX User (PRIMARY_ID id string)
CREATE DIRECTED EDGE Liked (FROM User, TO User) WITH REVERSE_EDGE = "Like
```

The first CREATE statement creates one vertex type called User. The second statement creates one directed edge type called Liked. The `WITH REVERSE_EDGE` clause means that for every two vertices (x,y) connected by a Liked type of edge, the system will automatically generate a corresponding edge of type Liked_By pointing from y to x, and both edges will have the same edge attributes.

After defining all your vertex and edge types, execute the following command to create a graph which binds the vertices and edges into one graph model:

CREATE GRAPH command

```
CREATE GRAPH gsql_demo(*)
```

The name of the graph is gsql_demo. Within the parentheses, you can either list the specific vertex and edge types (User, Liked), or you can use *, which means include everything. We chose to use * so that the same command can be used for all of our examples.

The CREATE commands can be stored in one file and executed together.

cf_model.gsql

```
CREATE VERTEX User (PRIMARY_ID id string)
CREATE DIRECTED EDGE Liked (FROM User, TO User) WITH REVERSE_EDGE = "Like
#CREATE GRAPH gsql_demo(*)
```

The CREATE GRAPH command is commented out for the following reason:

Our examples have been designed to run either as individual graphs or merged together into one multi-application graph. The CREATE GRAPH command may be run only once, after all the vertex and edge types have been created. (Each of our demo examples uses unique vertex and edge names, to avoid conflicts.) In other words, we run **`CREATE GRAPH gsql_demo(*)`** as a separate command after creating all the vertex and edge types. If you decide you want to modify the schema after running CREATE GRAPH, you can create and run a SCHEMA_CHANGE JOB.

Newer TigerGraph platforms (i.e., version 1.1 or higher) can support multiple graphs, but this tutorial has been designed to be compatible with older single-graph platforms.

- To execute these statements (DROP ALL, CREATE VERTEX, etc.), you can type them individually at the GSQL shell prompt, or you can first save them to a file, such as `cf_model.gsql`, and then run the command file. From within the shell, you would run
  `@cf_model.gsql`
  From outside the shell, you would run
  `>` **`gsql cf_model.gsql`**

Normally a user would put all their CREATE VERTEX, CREATE EDGE, and the final CREATE GRAPH statements in one file. In our example files, we have separated out the CREATE GRAPH statement because we want to merge all our example schemas together into one common graph.

- The vertex, edge, and graph types become part of the *catalog* . To see what is currently in your catalog, type the **`ls`** command from within the GSQL shell to see a report as below:

Catalog contents, as reported by the "ls" command

```
Vertex Types:
  - VERTEX User(PRIMARY_ID id STRING) WITH STATS="OUTDEGREE_BY_EDGETYPE"

Edge Types:
  - directed edge Liked(from User, to User) with reverse_edge="Liked_By"
  - directed edge Liked_By(from User, to User) with reverse_edge="Liked"

Graphs:

Jobs:
Queries:

Json API version: v2
```

- To remove a definition from the catalog, use some version of the `DROP` command. Use the `help` command to see a summary of available GSQL commands.
- In our examples, we typically show keywords in ALL UPPERCASE to distinguish them from user-defined identifiers. Identifiers are case-sensitive but keywords are not.

In this example, the vertices and edges don't have attributes. In general, a TigerGraph graph can have attributes on both vertices and edges, and it can also have different types of edges connecting the same two vertices. Please see GSQL Language Reference Part 1 - Defining Graphs and Loading Data which provides a more complete description of the graph schema definition language with additional examples.

Step 2: Load initial data.

Figure 2 shows a larger graph with five vertices and several edges. To avoid crowding the figure, only the Liked edges are shown: For every Liked edge, there is a corresponding Liked_By edge in the reverse direction.

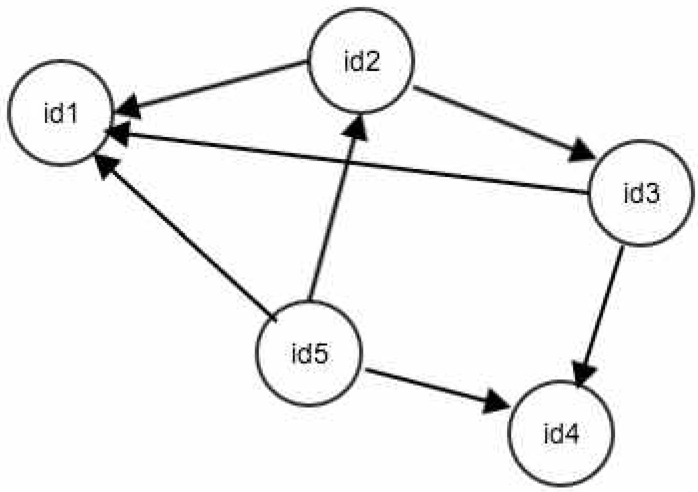


Figure 2 - Graph for Collaborative Filtering Calculation

The data file below describes the five vertices and seven edges of Figure 2.

cf_data.csv

```
id2,id1
id2,id3
id3,id1
id3,id4
id5,id1
id5,id2
id5,id4
```

The loading job below will read from a data file and create vertex and edge instances to put into the graph.

Per Example: Load data into the graph (file: cf_load.gsql)

```
# define the loading job
USE GRAPH gsql_demo # added for v1.2
CREATE LOADING JOB load_cf FOR GRAPH gsql_demo {
  DEFINE FILENAME f;
  LOAD f
    TO VERTEX User VALUES ($0),
    TO VERTEX User VALUES ($1),
    TO EDGE Liked VALUES ($0, $1);
}

# load the data
RUN LOADING JOB load_cf USING f="../cf/data/cf_data.csv"
```

Now that we have defined a graph (in Step 1), GSQL commands or sessions should specify that you want to use a particular graph. Line 2 (new for v1.2) sets the working graph to be gsql_demo. Another way to set the working graph is to specify each time you invoke the gsql command, e.g.,

```
gsql -g gsql_demo cf_load.gsql
```

The CREATE LOADING JOB statement (line 3) defines a job called load_cf. The job will read each line of the input file, creates one vertex based on the value in the first column (referenced with column name $0), another vertex based on the value in the second column ($1), and one Liked edge pointing from the first vertex to the second vertex. In addition, since the Liked edge type definition includes the WITH REVERSE_EDGE clause, a Liked_By edge pointing in the opposite direction is also created.

After the job has been created, we run the job (line 12). the RUN LOADING JOB command line includes details about the data source: the name of the file is cf_data.csv, commas are used to separate columns, and \n is used to end each line. (Data files should not contain any extra spaces before or after the separator character.)

The TigerGraph loader automatically filters out duplicates. If either of the two column values has already been seen before, that vertex won't be created. Instead the existing vertex will be used. For example, if we read the first two data lines in data file cf_data.csv , the first line will generate two `User` vertices, one edge type of `Liked` , and one edge type of `Liked_By` . For the second row, however, only one new vertex will be created since `id2` has been seen already. Two edges will be created for the second row.

1. It is okay to run an LOADING JOB again, or to run a different loading job, to add more data to a graph store which already has some data. For example, you could do the following:

```
RUN LOADING JOB load_cf USING f="../cf/cf_data1.csv"
RUN LOADING JOB load_cf USING f="../cf/cf_data2.tsv"
```

2.After loading, you can use the GraphStudio UI to visually inspect your data. Refer to the TigerGraph GraphStudio UI Guide .

To clear all your data but to keep your graph model, run the "CLEAR GRAPH STORE -HARD" command. -HARD must be in all capital letters.

Be very careful using CLEAR GRAPH STORE; there is no UNDO command.

For the querying and updating examples in the remainder of this use case, we will assume that Figure 2 has been loaded.

This loading example is basic. The GSQL language can do complex data extraction and transformation, such as dealing with JSON input format and key-value list input, all in high-level syntax. Please see GSQL Language Reference Part 1 - Defining Graphs and Loading Data for more examples.

Step 3: Write a graph-based query solution

The GSQL language includes not only data definition and simple inspection of the data, but also advanced querying which traverses the graph and which supports aggregation and iteration.

Built-In Queries

First , we can run some simple queries to verify that the data were loaded correctly. Below are some examples of some built-in GSQL queries which can be run in GSQL shell:

Simple Query for Validation	Meaning & Comments
`SELECT count(*) FROM User`	Display the number of User vertices,
`SELECT count(*) FROM User-(Liked)->User`	Display the number of directed Liked edges from User type to User type
`SELECT approx_count(*) FROM User`	Display the number of User vertices according to cached statistics. Response time may be faster than count(*). See note below.
`SELECT approx_count(*) FROM User-(Liked)->User`	Display the number of directed Liked edges from User type to User Type, according to cached statistics. Response time may be faster than count(*). See note below.
`SELECT * FROM User LIMIT 3`	Display all id, type, and attribute information for up to 3 User vertices. A LIMIT or WHERE condition is required, to prevent the output from being too large. Note that there is also a system limit of 10240 vertices or edges returned by SELECT *.
`SELECT * FROM User WHERE primary_id=="id2"`	Display all id, type and attribute information for the User vertex whose primary_id is "id2". The WHERE clause can also specify non-ID attributes.
`SELECT * FROM User-(ANY)->ANY WHERE from_id=="id1"`	Display all id,type, and attribute information about any type of edge which starts from vertex "id1". To guard against queries which select too many edges, the WHERE clause is mandatory when selecting edges.

```
GSQL > SELECT * FROM User LIMIT 5
[
  {
    "v_id": "id2",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id5",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id3",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id4",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id1",
    "attributes": {},
    "v_type": "User"
  }
]
```

Create a Query

> Note on approx_count(*)
>
> The approx_count(*) function relies on statistics which may not account for recent insertions and deletions. If there has been no recent activity, they will give accurate results. In contrast, the count(*) function insures that recent data insertions and deletions are processed, so that it returns an accurate count.

`SELECT *` displays information in JSON format. Below is an example of query output.

Now let's solve our original problem: find users who are co-liked with a user X. The following query demonstrates a 2-step traversal with aggregation.

The query below performs the co-liked collaborative filtering search. The concept behind this query is to describe a "graph path" which represents the relationship between a person (the starting point) and a person that is co-liked (the ending point). Figure 1 illustrates this path: id3 is a co-liked user of id1, because id2 likes both of them. The path from id1 to co-liked users is: (1) traverse a Liked_By edge to a User, and then (2) traverse a Liked edge to another User. This query also calculates the magnitude of the relationship between the starting point and each ending point. The more users there are such as id2 which connect id1 and id3, the stronger the co-like relationship between id1 and id3. Counting the number of paths that end at id3 serves to calculate this magnitude.

cf_query.gsql - Define the collaborative filtering query

```
CREATE QUERY topCoLiked( vertex<User> input_user,  INT topk) FOR GRAPH gs
{
  SumAccum<int>  @cnt = 0;
  # @cnt is a runtime attribute to be associated with each User vertex
  # to record how many times a user is liked.
    L0 = {input_user};
    L1 = SELECT tgt
        FROM L0-(Liked_By)->User:tgt;
    L2 =  SELECT tgt
          FROM L1-(Liked)->:tgt
          WHERE  tgt != input_user
          ACCUM tgt.@cnt += 1
          ORDER BY tgt.@cnt DESC
          LIMIT topk;
  PRINT L2;
}
```

This query is structured like a procedure with two input parameters: an input vertex and value of k for the top-K ranking. The query contains three SELECT statements executed in order. The L0 statement defines our initial list of vertices: a set containing a single user supplied by the `input_user` parameter. Suppose the input user is `id1`. Next, the L1 statement starts from every vertex in the set L0, traverses every connected edge of type Liked_By and returns every target vertex (that is, the other ends of the connected edges). As a result, L1 is the set of all users who liked the input user. Referring to the graph in Figure 2, the query travels backwards along every Liked edge which points to `id1`, arriving at `id2`, `id3`, and `id5`. These three vertices form L1. Next, the L2 statement starts from each user in L1, travels to every user liked by that starting user (via the Liked type of edges), and increments the count for each User reached. That is, the algorithm counts how many times each vertex is visited by a query path. The WHERE condition makes sure the original input user will not be returned in the result.ORDER BY and LIMIT have the same meaning as in SQL. Below, we show how the L2 step tallies the counts for each vertex encountered:

1. From id2, Liked edges lead to id1 and id3. id1 is excluded due to the WHERE clause. The cnt count for id3 is incremented from 0 to 1.
2. From id3, Liked edges lead to id1 and id4. id1 is excluded due to the WHERE clause. The cnt count for id4 is incremented from 0 to 1.
3. From id5, Liked edges lead to id1, id2, and id4. id1 is excluded to to the WHERE clause. The cnt count of id2 is incremented from 0 to 1. The cnt count of id4 is incremented from 1 to 2.

The three co-liked users and their cnt scores: id3 (cnt score = 1), id4 (cnt = 2), and id2 (cnt = 1). The ORDER BY clause indicates that the sorting should be in descending order, such that the LIMIT clause trims L2 to the 20 vertices with the highest (as opposed to lowest) cnt values. For the test graph, there are only 3 vertices which are co-liked, less than the limit of 20. id4 has the strongest co-liked relationship.

Install and Run a Query

After the query is defined (in the CREATE QUERY block), it needs to be installed. The INSTALL QUERY command compiles the query.

```
INSTALL QUERY topCoLiked
```

If you have several queries, you can wait to install them in one command, which runs faster than installed each one separately. E.g.,

```
INSTALL QUERY query1, query2
```

or

```
INSTALL QUERY ALL
```

is faster than

```
INSTALL QUERY query1
INSTALL QUERY query2
```

After a query has been installed, it can be run as many times has desired. The command RUN QUERY invokes the query, with the given input arguments.

Using "id1" as the starting point and allowing up to 5 vertices in the output, the RUN QUERY command and its output on our test graph is shown below:

```
GSQL > RUN QUERY topCoLiked("id1", 5)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"L2": [
    {
      "v_id": "id4",
      "attributes": {"@cnt": 2},
      "v_type": "User"
    },
    {
      "v_id": "id3",
      "attributes": {"@cnt": 1},
      "v_type": "User"
    },
    {
      "v_id": "id2",
      "attributes": {"@cnt": 1},
      "v_type": "User"
    }
  ]}]
}
```

Instead of using the RUN QUERY command within the GSQL shell, the query can be invoked from the operating system via a RESTful GET endpoint (which is automatically created by the `INSTALL QUERY` command):

```
curl -X GET "http://hostName:port/query/gsql_demo/topCoLiked?input_user=i
```

If you followed the standard installation instructions for the TigerGraph system, *hostName* for the REST server is **`localhost`** and **`port`** is **`9000`**.

> As of TigerGraph 1.2, the URL for query REST endpoints includes the graph name after query/. Prior to 1.2, the URL for the example above was http://hostName:port /query/topCoLiked

Step 4 (Optional): Update Your Data.

You can update the stored graph at any time, to add new vertices and edges, to remove some, or to update existing values. The GSQL language includes ADD, DROP, ALTER, UPSERT, and DELETE operations which are similar to the SQL operations of the same name. The UPSERT operation is a combined UPDATE-INSERT operation: If *object* exists, then UPDATE, else INSERT. Note that this is the default behavior for The GSQL language's 'smart' loading described above. There are three basic types of modifications to a graph:

1. Adding or deleting objects
2. Altering the schema of the graph
3. Modifying the attributes of existing objects

We'll give a quick example of each type. To show the effect each modification, we'll use the following simple built-in queries:

cf_mod_check.gsql

```
SELECT * FROM User LIMIT 1000
SELECT * FROM User-(Liked)->User WHERE from_id=="id2"
```

The current results, before making any modifications, are shown below.

Users vertices and Edges from id2, before any modifications

```
GSQL > SELECT * FROM User LIMIT 1000
[
  {
    "v_id": "id2",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id5",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id3",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id4",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id1",
    "attributes": {},
    "v_type": "User"
  }
]
GSQL > SELECT * FROM User-(Liked)->User WHERE from_id=="id2"
[
  {
    "from_type": "User",
    "to_type": "User",
    "directed": true,
    "from_id": "id2",
    "to_id": "id3",
    "attributes": {},
    "e_type": "Liked"
  },
  {
    "from_type": "User",
    "to_type": "User",
    "directed": true,
    "from_id": "id2",
    "to_id": "id1",
    "attributes": {},
    "e_type": "Liked"
  }
```

```
]
```

⚠ Graph modification operations are performed by a distributed computing model which satisfies Sequential Consistency. For these examples, a brief one second pause between the updating and querying the graph should be sufficient.

Modification Type 1: Adding or deleting

Adding is simply running a loading job again with a new data file. More details are in the GSQL Language Reference Part 1.

Deleting: Suppose we want to delete vertex id3 and all its connections:

cf_mod1.gsql

```
DELETE FROM User WHERE primary_id=="id3"
```

Users vertices and Edges from id2, after Modification 1

```
GSQL > SELECT * FROM User LIMIT 1000
[
  {
    "v_id": "id2",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id5",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id4",
    "attributes": {},
    "v_type": "User"
  },
  {
    "v_id": "id1",
    "attributes": {},
    "v_type": "User"
  }
]
GSQL > SELECT * FROM User-(Liked)->User WHERE from_id=="id2"
[{
  "from_type": "User",
  "to_type": "User",
  "directed": true,
  "from_id": "id2",
  "to_id": "id1",
  "attributes": {},
  "e_type": "Liked"
}]
```

Modification Type 2: Altering the schema

The GSQL DELETE operation is a cascading deletion. If a vertex is deleted, then all of the edges which connect to it are automatically deleted as well.

Result: one fewer vertex and one fewer edge from id2.

The GSQL language supports four types of schema alterations:

1. Adding a new type of vertex or edge: `ADD VERTEX | DIRECTED EDGE | UNDIRECTED EDGE`
2. Removing a type of vertex or edge: `DROP VERTEX | DIRECTED EDGE | UNDIRECTED EDGE`
3. Adding attributes to a vertex or edge type: `ALTER VERTEX vertex_type | EDGE edge_type ADD ATTRIBUTE (name type)`
4. Removing attributes of a vertex or edge type: `ALTER VERTEX vertex_type | EDGE edge_type DROP ATTRIBUTE (name)`

To make schema changes, create a SCHEMA_CHANGE job. Running the SCHEMA_CHANGE JOB will automatically stop all services, update the graph store, and restart the service. For example, suppose we wish to add a name for Users and a weight to Liked edges to indicate *how much* User A likes User B.

cf_mod2.gsql

```
CREATE GLOBAL SCHEMA_CHANGE JOB cf_mod2 {
    ALTER VERTEX User ADD ATTRIBUTE (name string);
    ALTER EDGE Liked ADD ATTRIBUTE (weight float DEFAULT 1);
}
RUN JOB cf_mod2
```

As of v1.2, the schema_change job here needs to be GLOBAL because the User vertex and Liked edge are global types (they were defined before an active graph was set.)

Changing the schema may necessitate changing queries and other tasks, such as REST endpoints. In this example, the collaborative filtering query will still run with the the new weight attribute, but it will ignore the weight in its calculations.

Users vertices and Edges from id2, after Modification 2

```
GSQL > SELECT * FROM User LIMIT 1000
[
  {
    "v_id": "id2",
    "attributes": {"name": ""},
    "v_type": "User"
  },
  {
    "v_id": "id5",
    "attributes": {"name": ""},
    "v_type": "User"
  },
  {
    "v_id": "id4",
    "attributes": {"name": ""},
    "v_type": "User"
  },
  {
    "v_id": "id1",
    "attributes": {"name": ""},
    "v_type": "User"
  }
]
GSQL > SELECT * FROM User-(Liked)->User WHERE from_id=="id2"
[{
  "from_type": "User",
  "to_type": "User",
  "directed": true,
  "from_id": "id2",
  "to_id": "id1",
  "attributes": {"weight": 1},
  "e_type": "Liked"
}]
```

Modification Type 3: Modifying the attributes of existing objects

Now that we have added a weight attribute, we probably want to assign some weight values to the graph. The following example updates the weight values of two edges. For edge upserts, the first two arguments in the VALUES list specify the FROM vertex id and the TO vertex_id, respectively. Similarly, for vertex upserts, the first argument in the VALUES list specifies the PRIMARY_ID id. Since id values may not be updated, the GSQL shell implicitly applies a conditional test: "If the specified id value(s) exist, than update the non-id attributes in the VALUES list; otherwise, insert a new data record using these values."

cf_mod3.gsql

```
UPSERT User VALUES ("id1", "Aaron")
UPSERT User VALUES ("id2", "Bobbie")
UPSERT User-(Liked)->User VALUES ("id2","id1",2.5)
```

Users vertices and Edges from id2, after Modification 3

```
GSQL > SELECT * FROM User LIMIT 1000
[
  {
    "v_id": "id2",
    "attributes": {"name": "Bobbie"},
    "v_type": "User"
  },
  {
    "v_id": "id5",
    "attributes": {"name": ""},
    "v_type": "User"
  },
  {
    "v_id": "id4",
    "attributes": {"name": ""},
    "v_type": "User"
  },
  {
    "v_id": "id1",
    "attributes": {"name": "Aaron"},
    "v_type": "User"
  }
]
GSQL > SELECT * FROM User-(Liked)->User WHERE from_id=="id2"
[{
  "from_type": "User",
  "to_type": "User",
  "directed": true,
  "from_id": "id2",
  "to_id": "id1",
  "attributes": {"weight": 2.5},
  "e_type": "Liked"
}]
```

Other Modes for Graph Updates

In addition to making graph updates within the GSQL Shell, there are two other ways: sending a query string directly to the Standard Data Manipulation REST API, or writing a custom REST endpoint. For details about the first method, see the GET, POST, and DELETE /graphendpoints in the **RESTPP API User Guide .** The functionality in GSQL and in the Standard Query API is the same; GSQL commands are translated into REST GET, POST, and DELETE requests and submitted to the Standard Query API.

The REST API equivalent of the GSQL Modification 3 upsert example above is as follows:

```
curl -X POST --data-binary @ data/cf_mod3_input.json http://hostName:9000
```

where *serverIP* is the IP address of your REST server (default = `localhost`) and *data/cf_mod3_input.json* is a text file containing the following JSON-encoded data:

cf_upsert.json

```
{
    "vertices": {
        "User":{
            "id1":{
                "name":{
                    "value":"Aaron"
                }
            }
        },
        "User":{
            "id2":{
                "name":{
                    "value":"Bobbie"
                }
            }
        }
    },
    "edges": {
         "User":{
             "id2":{
                 "Liked":{
                     "User":{
                         "id1":{
                             "weight" : {
                                 "value":2.5
                             }
                         }
                     }
                 }
             }
         }
    }
}
```


Example 2. Page Rank

This example shows the use of WHILE loop iteration, global variables , and the built-in outdegree attribute.

It is recommended that you do the Collaborative Filtering Use Case first, because it contains additional tips on running the TigerGraph system.

Remember that if you have a text file containing GSQL commands (e.g., commands.gsql), you can run it one of two ways:

- From Linux: `gsql commands.gsql`
- From inside the GSQL shell: `@commands.gsql`

To run a single command (such as DROP ALL):

- From Linux: `gsql 'DROP ALL'`
- From inside the GSQL shell: `DROP ALL`

Setting the working graph

If a graph has been defined, then all subsequent gsql commands need to specify which graph is being used. If your command file does not contain a "USE GRAPH" statement, then you can specify the graph when invoking gsql:gsql -g graph_name commands.gsql

```
gsql -g graph_name commands.gsql
```

If you are always using the same graph, you can define a Linux alias to automatically include your graph name:

```
alias gsql='gsql -g graph_name'
```

You can add this line to the .bashrc in your home directory so that the alias is defined each time you open a bash shell.

Step 1: Create a graph model.

In this example, there is only one type of vertex and one type of edge, and edges are directed.

pagerank_model.gsql

```
CREATE VERTEX Page (PRIMARY_ID pid string, page_id string)
CREATE DIRECTED EDGE Linkto (FROM Page, TO Page)
#CREATE GRAPH gsql_demo(*)
```

Note how the Page vertex type has both a PRIMARY_ID and a page_id attribute. As will be seen in step 2, the same data will be loaded into both fields. While this seems redundant, this is a useful technique in TigerGraph graph stores. The PRIMARY_ID is not treated as an ordinary attribute. In exchange for high-performance storage, the PRIMARY_ID lacks some of the filtering and querying features available to regular attributes. The Linkto edge does not have any attributes. In general, a TigerGraph graph can have attributes on both vertices and edges, and it can also have different types of edges connecting the same two vertices.

The CREATE GRAPH command is commented out for the following reason:

Our examples have been designed to run either as individual graphs or merged together into one multi-application graph. The CREATE GRAPH command should be run only once, after all the vertex and edge types for all the examples have been created. (Naturally, every model uses unique vertex and edge names, to avoid conflicts.) In other words, run ' `CREATE GRAPH gsql_demo(*)` ' as a separate command after you have created all your vertex and edge types.

Please see the GSQL Language Reference which provides a more complete description of the graph schema definition language with additional examples .

Step 2: Load initial data

A similar graph to what was used for the Collaborative Filtering user-user network example can be used for an example here. That is, each row has two values which are node IDs, meaning that there is a connection from the first node to the second node. However, we will introduce a difference to demonstrate the flexibility of the TigerGraph loading system. We will modify the data file to use the tab character as a field separator instead of the comma.

pagerank_data.tsv

```
1   2
1   3
2   3
3   4
4   1
4   2
```

Loading job:

Create your loading job and load the data.

Per Example: Load data into the graph (file: pagerank_load.gsql)

```
# define the loading job
CREATE LOADING JOB load_pagerank FOR GRAPH gsql_demo {
  DEFINE FILENAME f;
  LOAD f
      TO VERTEX Page VALUES ($0, $0),
      TO VERTEX Page VALUES ($1, $1),
      TO EDGE Linkto VALUES ($0, $1)
      USING SEPARATOR="\t";
}

# load the data
RUN LOADING JOB load_pagerank USING f="../pagerank/pagerank_data.tsv"
```

The above loading job will read each line of the input file (pagerank_data.tsv), create one vertex based on the value in the first column (referenced as $0), another vertex based on the value in the second column ($1), and one edge pointing from the first vertex to the second vertex. If either of the two column values has already been seen before, that vertex won't be created. Instead the existing vertex will be used. For example, the first row of pagerank_data.tsv, will create two vertices, with ids 1 and 2, and one edge (1, 2). The second row, however, will create only one new vertex, id 3, and one edge (1, 3), because id 1 already exists.

Note how the LOAD statement specifies the SEPARATOR character is the tab character.

Step 3: Write a Graph-based query solution

GSQL includes not only data definition and simple inspection of the data, but also advanced querying which traverses the graph and which supports aggregation and iteration. This example uses iterations, repeating the computation block until the maximum score change at any vertex is no more than a user-provided threshold, or until it reaches a user-specified maximum number of allowed iterations. Note the arrow `->` in the `FROM` clause used to represent the direction of a directed edge.

pagerank_query.gsql

```
CREATE QUERY pageRank (float maxChange, int maxIteration, float dampingFa
FOR GRAPH gsql_demo
{
  # In each iteration, compute a score for each vertex:
  #   score = dampingFactor + (1-dampingFactor)* sum(received scores from
  # The pageRank algorithm stops when either of the following is true:
  #  a) it reaches maxIterations iterations;
  #  b) max score difference of any vertex compared to the last iteration
  #   @@ prefix means a global accumulator;
  #   @ prefix means an individual accumulator associated with each verte

  MaxAccum<float> @@maxDifference = 9999; # max score change in an iterat
  SumAccum<float> @received_score = 0; # sum of scores each vertex receiv
  SumAccum<float> @score = 1;   # initial score for every vertex is 1.

  AllV = {Page.*};   #  Start with all vertices of type Page
  WHILE @@maxDifference > maxChange LIMIT maxIteration DO
    @@maxDifference = 0;
    S = SELECT s
         FROM AllV:s-(Linkto)->:t
         ACCUM t.@received_score += s.@score/s.outdegree()
         POST-ACCUM s.@score = dampingFactor + (1-dampingFactor) * s.@rec
                    s.@received_score = 0,
                    @@maxDifference +=   abs(s.@score - s.@score');
    PRINT @@maxDifference; # print to default json result
  END; # end while loop
  #PRINT AllV.page_id, AllV.@score;       # print the results, JSON outpu
  PRINT AllV[AllV.page_id, AllV.@score];  # print the results, JSON outpu
} # end query
```

For JSON output API v2, the PRINT syntax for a vertex set variable is different than the v1 syntax.

After executing the CREATE QUERY command, remember to install the query, either by itself or together with other queries:

Install the query

```
INSTALL QUERY pageRank
```

Run the query:

We will use the typical dampingFactor of 0.15, iterate until the pagerank values change by less than 0.001, up to a maximum of 100 iterations. For these conditions, the PageRank values for the 4 vertices (1,2,3,4) are (0.65551, 0.93379, 1.22156, 1.18914), respectively.

```
RUN QUERY pageRank(0.001, 100, 0.15)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@maxDifference": 0.425},
    {"@@maxDifference": 0.36125},
    {"@@maxDifference": 0.15353},
    {"@@maxDifference": 0.19575},
    {"@@maxDifference": 0.16639},
    {"@@maxDifference": 0.09429},
    {"@@maxDifference": 0.08014},
    {"@@maxDifference": 0.05961},
    {"@@maxDifference": 0.04705},
    {"@@maxDifference": 0.03999},
    {"@@maxDifference": 0.017},
    {"@@maxDifference": 0.02},
    {"@@maxDifference": 0.017},
    {"@@maxDifference": 0.00953},
    {"@@maxDifference": 0.0081},
    {"@@maxDifference": 0.00616},
    {"@@maxDifference": 0.00479},
    {"@@maxDifference": 0.00407},
    {"@@maxDifference": 0.00178},
    {"@@maxDifference": 0.00205},
    {"@@maxDifference": 0.00174},
    {"@@maxDifference": 9.6E-4},
    {"AllV": [
      {
        "v_id": "2",
        "attributes": {
          "AllV.page_id": "2",
          "AllV.@score": 0.93379
        },
        "v_type": "Page"
      },
      {
        "v_id": "4",
        "attributes": {
          "AllV.page_id": "4",
          "AllV.@score": 1.18914
        },
        "v_type": "Page"
```

```
        },
        {
          "v_id": "1",
          "attributes": {
            "AllV.page_id": "1",
            "AllV.@score": 0.65551
          },
          "v_type": "Page"
        },
        {
          "v_id": "3",
          "attributes": {
            "AllV.page_id": "3",
            "AllV.@score": 1.22156
          },
          "v_type": "Page"
        }
      ]}
    ]
  }
```


Step 4 (Optional): Update Your Data.

Details about updating were discussed in Use Case 1 (Collaborative Filtering). We will go right to the graph modification examples for the PageRank case.

To show the effect of each modification, we use two built-in queries. The first one lists all the Page vertices. The second one lists all the edges which start at Page 4.

pagerank_mod_check.gsql

```
SELECT * FROM Page LIMIT 1000
SELECT * FROM Page-(Linkto)->Page WHERE from_id=="4"
```

These are the results of the diagnostic queries, before any graph modifications. There are 4 vertices total and 2 edges which start at page 4.

Page vertices and Linkto edges from Page 4, before modifications

```
SELECT * FROM Page LIMIT 1000
[
  {
    "v_id": "2",
    "attributes": {"page_id": "2"},
    "v_type": "Page"
  },
  {
    "v_id": "4",
    "attributes": {"page_id": "4"},
    "v_type": "Page"
  },
  {
    "v_id": "1",
    "attributes": {"page_id": "1"},
    "v_type": "Page"
  },
  {
    "v_id": "3",
    "attributes": {"page_id": "3"},
    "v_type": "Page"
  }
]
SELECT * FROM Page-(Linkto)->Page WHERE from_id=="4"
[
  {
    "from_type": "Page",
    "to_type": "Page",
    "directed": true,
    "from_id": "4",
    "to_id": "2",
    "attributes": {},
    "e_type": "Linkto"
  },
  {
    "from_type": "Page",
    "to_type": "Page",
    "directed": true,
    "from_id": "4",
    "to_id": "1",
    "attributes": {},
    "e_type": "Linkto"
  }
]
```

Modification 1: Adding or deleting

Adding is simply running a loading job again with a new data file.

Deleting: Suppose we want to delete vertex url4 and all its connections:

pagerank_mod1.gsql

```
DELETE FROM Page WHERE page_id=="1"
```

The GSQL DELETE operation is a cascading deletion. If a vertex is deleted, then all of the edges which connect to it are automatically deleted as well.

Result: one fewer vertex and one fewer edge from Page 4.

Page vertices and Linkto edges from Page 4, after Modification 1

```
SELECT * FROM Page LIMIT 1000
[
  {
    "v_id": "2",
    "attributes": {"page_id": "2"},
    "v_type": "Page"
  },
  {
    "v_id": "4",
    "attributes": {"page_id": "4"},
    "v_type": "Page"
  },
  {
    "v_id": "3",
    "attributes": {"page_id": "3"},
    "v_type": "Page"
  }
]
SELECT * FROM Page-(Linkto)->Page WHERE from_id=="4"
[{
  "from_type": "Page",
  "to_type": "Page",
  "directed": true,
  "from_id": "4",
  "to_id": "2",
  "attributes": {},
  "e_type": "Linkto"
}]
```

Modification Type 2: Altering the schema

For example, suppose we wish to add an attribute to the Page vertices to classify what type of Page it is and also a date to the edges.

pagerank_mod2.gsql

```
CREATE GLOBAL SCHEMA_CHANGE JOB pagerank_mod2 {
    ALTER VERTEX Page ADD ATTRIBUTE (pageType string DEFAULT "");
    ALTER EDGE Linkto ADD ATTRIBUTE (dateLinked string DEFAULT "");
}
RUN JOB pagerank_mod2
```

⚠ Changing the schema may necessitate revising and reinstalling loading jobs and queries. In this case, adding the pageType attribute does not harm the pageRank query.

⚠ This schema_change job is GLOBAL because the Page vertex and Linkto edge types are global (defined before setting an active graph).

Page vertices and Linkto edges from Page 4, after Modification 2

```
SELECT * FROM Page LIMIT 1000
[
  {
    "v_id": "2",
    "attributes": {
      "page_id": "2",
      "pageType": ""
    },
    "v_type": "Page"
  },
  {
    "v_id": "4",
    "attributes": {
      "page_id": "4",
      "pageType": ""
    },
    "v_type": "Page"
  },
  {
    "v_id": "3",
    "attributes": {
      "page_id": "3",
      "pageType": ""
    },
    "v_type": "Page"
  }
]
SELECT * FROM Page-(Linkto)->Page WHERE from_id=="4"
[{
  "from_type": "Page",
  "to_type": "Page",
  "directed": true,
  "from_id": "4",
  "to_id": "2",
  "attributes": {"dateLinked": ""},
  "e_type": "Linkto"
}]
```

Modification Type 3: Modifying the attributes of existing objects

The following example updates the type values of two vertices and one edge.

```
UPSERT Page VALUES (2,2,"info")
UPSERT Page VALUES (3,3,"commerce")
UPSERT Page-(Linkto)->Page VALUES (4,2,"2016-08-31")
```

Page vertices and Linkto edges from Page 4, after Modification 3

```
SELECT * FROM Page LIMIT 1000
[
  {
    "v_id": "2",
    "attributes": {
      "page_id": "2",
      "pageType": "info"
    },
    "v_type": "Page"
  },
  {
    "v_id": "4",
    "attributes": {
      "page_id": "4",
      "pageType": ""
    },
    "v_type": "Page"
  },
  {
    "v_id": "3",
    "attributes": {
      "page_id": "3",
      "pageType": "commerce"
    },
    "v_type": "Page"
  }
]
SELECT * FROM Page-(Linkto)->Page WHERE from_id=="4"
[[{
  "from_type": "Page",
  "to_type": "Page",
  "directed": true,
  "from_id": "4",
  "to_id": "2",
  "attributes": {"dateLinked": "2016-08-31"},
  "e_type": "Linkto"
}]
```

Other Modes for Graph Updates

In addition to making graph updates within the GSQL Shell, there are two other ways: sending a query string directly to the Standard Data Manipulation REST API, or writing a custom REST endpoint. For details about the first method, see the Standard Data Manipulation REST API User Guide . The functionality in GSQL and in the Standard Query API is essentially the same; GSQL commands are translated into REST GET, POST, and DELETE requests and submitted to the Standard Query API.

The REST API equivalent of the GSQL Modification 3 upsert example above is as follows:

```
curl -X POST --data-binary @data/pagerank_mod3_input.json http://hostName
```

where *hostName* is the IP address of your REST server, and *data/pagerank_mod3_input.json* is a text file containing the following JSON-encoded data:

```
{
   "vertices": {
        "Page":{
            "2":{
                "pageType" : {
                    "value":"info"
                }
            }
        },
        "Page":{
            "3":{
                "pageType" : {
                    "value":"commerce"
                }
            }
        }
    },
    "edges": {
         "Page":{
             "4":{
                 "Linkto":{
                     "Page":{
                         "2":{
                             "dateLinked" : {
                                 "value":"2016-08-31"
                             }
                         }
                     }
                 }
             }
        }
    }
}
```

Example 3. Simple Product Recommendation

This example introduces the technique of flattening – splitting a data field which contains a set of elements into individual vertices and edges, one for each element.

Input Data: A list of products. Each Product has a 64-bit image hash value and a list of words describing the product.
Query Task **:** Find the products which are most similar to a given product. Formally,

given a product id P and an integer K,return the top K products similar to the product P. The similarity between a product P and another product Q is based on the number of words found in the product descriptions for both product P and product Q.

Step 1: Create a graph model for the use case, using the data definition language (DDL) aspect of the GSQL language.

simprod_model.gsql

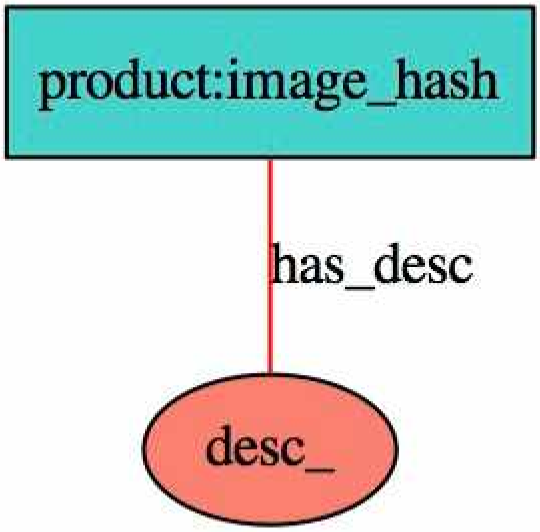
```
CREATE VERTEX Product (PRIMARY_ID pid string, image_hash uint)
CREATE VERTEX DescWord (PRIMARY_ID id string)
CREATE UNDIRECTED EDGE Has_desc (FROM Product, TO DescWord)
```

Then run

```
CREATE GRAPH gsql_demo(*)
```

The above statements create two types of vertices, Product and DescWord, and one type of edge connecting the two vertex types. The edge is undirected so that you can just as easily traverse from a Product to its descriptive words or from a descriptive word to Products which are described by it.

The generated graph schema for this case is shown below. The GSQL Language Reference manual provides a more complete description of the language with more examples .



Step 2: Load Input Data.

In this example, the input data are all stored in a single file having a 3-column format with a header column. Below are the test data:

simprod_data.csv

```
id,hash,words
62abcax334,15243242,"word1,word2,word3"
dell laptop,1837845,"word2,word4,word5"
mac book, 128474373,"word4"
surface pro,8439828,"word1,word3,word6"
hp book,29398439828,"word2,word3,word1"
linux abc,298439234,"word4,word2,word1"
linux def,295839234,"word4,word2,word6,word7"
```

Column 1 is the product id; column 2 is the image hash code, and column 3 is a list of words describing the product. Note how double quotation marks are used to enclose the list of words. Each row from the input file may lead to the creation of one Product vertex, multiple DescWord vertices, and multiple edges, one edge connecting the Product to each DescWord vertex.

The loading job below makes use of several features of the loading language to intelligently transform this data file into the appropriate vertices and edges.

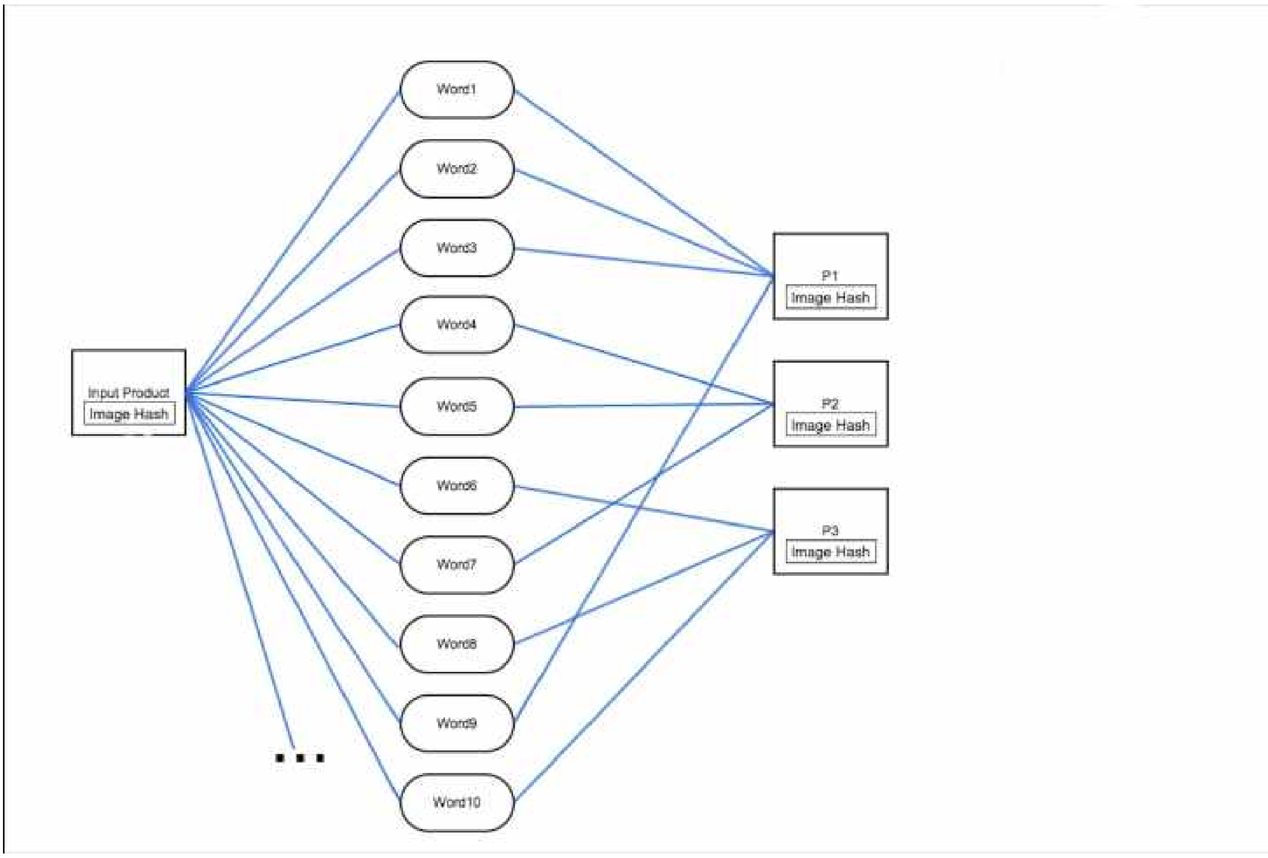
1. The HEADER="true" option tells the loader that the data file's first line contains column headings instead of data. It will read the column headings and permit these heading names to be used instead of index numbers $1, $2, etc.
2. DEFINE HEADER and USER_DEFINED_HEADER allow the loading job to define its own names for the columns ("id", "hash", "words"), instead of the index numbers ($0, $1, $2) and overriding the file's own headings.
3. QUOTE="double" informs the loader that double quotation marks enclose strings. This allows the separator character (e.g., comma) to appear in the string, without triggering the end of the token. QUOTE="single" is also available.
4. The special TEMP_TABLE and flatten() function are used to split the list of tokens into separate items and to store them temporarily. The temporary items are then used to assemble the final edge objects.

simprod_load.gsql

```
# define the loading job
CREATE LOADING JOB load_simprod FOR GRAPH gsql_demo {
  DEFINE HEADER head1 = "id","hash","words";
  DEFINE FILENAME f1;
  LOAD f1
    TO VERTEX Product values ($"id", $"hash"),
    TO TEMP_TABLE t (pid, description) VALUES ($"id", flatten($"words", "
    USING QUOTE="double", HEADER="true", USER_DEFINED_HEADER="head1";
  LOAD TEMP_TABLE t
    TO VERTEX DescWord VALUES ($"description"),
    TO EDGE Has_desc VALUES ($"pid", $"description");
}

# load the data
RUN LOADING JOB load_simprod USING f1="../simprod/data/simprod_data.csv"
```

In general, the GSQL language can map and transform multiple input files to multiple vertex and edge types. More advanced data transformation and filtering features are also available. See the GSQL Language Reference manual for more information.

An example of the resulting data graph is shown below. Products (P1, P2, etc.) connect to various DescWords (Word1, Word2, etc.). Each Product connects to many DescWords, and each DescWord is used in multiple Products.



Step 3: Write a graph-based solution using TigerGraph's high-level GSQL query language, to solve the use case and auto-generate the REST GET/POST endpoints for real-time accesses to TigerGraph's system.

simprod_query.gsql

```
CREATE QUERY productSuggestion (vertex<Product> seed, int threshold_cnt,
FOR GRAPH gsql_demo
{
    # an accumulator variable attachable to any vertex
    SumAccum<int>     @cnt = 0;

    # T0 is the set of products from which we want to start traversal in
    T0={seed};

    /**
    *  Compute the collection of words describing the input
    *  product. tgt is the alias of vertex type DescWord.
    *  In other words, for every edge of the given type (Has_desc)
    *  that has one vertex in the set T0 and the other vertex being of
    *  the DescWord type, add its DescWord vertex to the output set.
    */
    ProductWords = SELECT tgt
                   FROM T0-(Has_desc)-DescWord:tgt;

    /**
     *  The output set of the previous query (ProductWords) becomes the i
     *  of this query. From each word in ProductWords, activate all other
     *  which contain the word in their description, and accumulate a cou
     *  each activated product to record how many words it has in common
     *  input product.  Then rank each related product using the count of
     *  words; the count must exceed the query parameter threshold_cnt.
     */
    Results = SELECT tgt
        FROM ProductWords-(Has_desc)->Product:tgt
        WHERE tgt != seed
        ACCUM tgt.@cnt += 1
        HAVING tgt.@cnt > threshold_cnt
        ORDER BY tgt.@cnt DESC
        LIMIT k;

    PRINT Results; # default print output is the REST call response in JS
}
```

Query Result:

For product id= 62abcax334, find the top 3 similar products, which have more than 1 descriptive word in common with product 62abcax334.

```
//INSTALL QUERY productSuggestion
RUN QUERY productSuggestion("62abcax334", 1, 3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Results": [
    {
      "v_id": "hp book",
      "attributes": {
        "@cnt": 3,
        "image_hash": 29398439828
      },
      "v_type": "Product"
    },
    {
      "v_id": "surface pro",
      "attributes": {
        "@cnt": 2,
        "image_hash": 8439828
      },
      "v_type": "Product"
    },
    {
      "v_id": "linux abc",
      "attributes": {
        "@cnt": 2,
        "image_hash": 298439234
      },
      "v_type": "Product"
    }
  ]}]
}
```

When installing the above GSQL query, a REST GET endpoint for this query will automatically be generated. Instead of running the query as a GSQL command, clients can also invoke the query by formatting the query as a HTTP request query string and sending a GET request, e.g.,

Example 4. Same Name Search

This example introduces the CASE...WHEN...THEN structure, which can also be used as an if...then block.

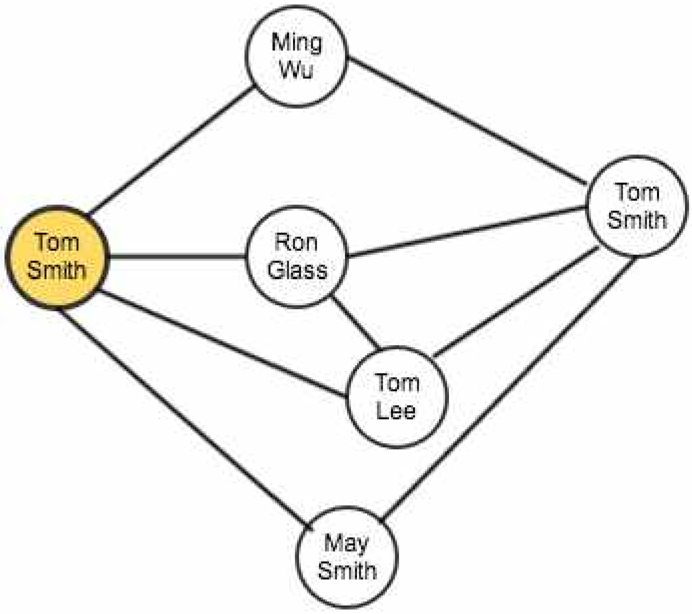
Input Data: A social network, where each person has a first and last name and may also display a picture of themselves.

Query Task : Find the users who are most "similar" to a user X. Specifically, a user X searches for other users whose first or last name matches user X's name. The query returns the list of users (Y1,Y2,...Yk) within two steps (two steps means friend-of-friend), who have matching names, and who offer a picture. The list is sorted and ranked by the relevance score between X and another user Yi, where the score is a linear function of four factors:

```
curl -X GET "http://hostName:port/query/gsql_demo/productSuggestion?seed=
```

For the standard TigerGraph configuration, `hostName:port` is **`localhost:9000`**

1. depth : how far X is from Yi (the shortest distance)
2. count : the number of shortest paths between X and Yi
3. match : whether Yi matches the input first name (match=1), the input last name (match=2), or both input names (match=3)
4. profile : whether Yi has a profile picture



Using the graph above as an example, suppose we want to compute relevance scores for the social network of the Tom Smith on the left.

- match=1 for Tom Lee (first names are the same)
- match=2 for May Smith (last names are the same)
- match=3 for Tom Smith on the right (both names are the same).

There is no direct connection to the other Tom Smith, but there are several paths:

- Paths with a depth = 2:
  - Tom Smith → Ming Wu → Tom Smith
  - Tom Smith → Ron Glass → Tom Smith
  - Tom Smith → Tom Lee → Tom Smith
  - Tom Smith → May Smith → Tom Smith

There are also some longer paths (e.g., Tom Smith → Ron Glass → Tom Lee → Tom Smith), but since they are longer, they are not of interest. Therefore, for the relationship (Tom Smith, Tom Smith), depth = 2 and count = 4.

The four factors (depth, count, match, hasPicture) are combined to compute an overall relevance score:

```
score = match * 100 + (4-depth) * 50 + count + hasPicture? 200 : 0
```

The clause `hasPicture? 200 : 0` uses the ternary conditional operator. If `hasPicture` is TRUE, evaluate to 200. Otherwise, evaluate to 0.

To design the graph schema, consider what attributes are needed for each vertex and attribute. The User vertices need to have a first name, a last name, and a profile picture. We assume that the social network is stored in two data files, one for vertices and one for edges.

Vertex file format: id, firstname, lastname, img_url
Edge file format: user1, user2

The following code creates the schema and loads the data:

name_model.gsql

```
CREATE VERTEX NameUser (PRIMARY_ID id string, firstname string, lastname
CREATE UNDIRECTED EDGE NameConn (FROM NameUser, TO NameUser)
```

```
CREATE GRAPH gsql_demo(*)
```

name_load.gsql

```
# define the loading job
CREATE LOADING JOB load_nameV FOR GRAPH gsql_demo {
  DEFINE FILENAME f1;
  LOAD f1 TO VERTEX NameUser VALUES ($0, $1, $2, $3);
}
CREATE LOADING JOB load_nameE FOR GRAPH gsql_demo {
  DEFINE FILENAME f2;
  LOAD f2 TO EDGE NameConn VALUES ($0, $1);
}

# load the data
RUN lOADING JOB load_nameV USING f1="../name/data/name_search_vertex.csv"
RUN LOADING JOB load_nameE USING f2="../name/data/name_search_edge.csv"
```

Test data files

name_search_vertex.csv

```
0,michael,jackson,
1,michael,franklin,abc.com
2,michael,lili,def.com
3,franklin,lili,
4,lucia,franklin,lucia.org
5,michael,jackson,
6,michael,jackson,abc.com
7,lucia,jackson,
8,hahah,jackson,haha.net
```

name_search_edge.csv

```
0,1
0,3
0,4
1,5
1,3
1,2
2,6
2,7
2,1
2,0
3,0
3,1
3,5
3,7
4,5
5,6
5,7
6,7
6,1
6,2
6,0
6,4
7,5
8,5
```

The query algorithm is a bit long but straightforward:

1. Select all the depth=1 neighbors. For each neighbor:
    a. Use a CASE structure to check for matching first and last names and assign a match value.

   b. Check for an image.
   c. We know depth=1 and count=1, so compute the relevance score.
2. Starting from the depth=1 neighbors, move to the depth=2 neighbors. For each such neighbor:
   a. Use a CASE structure to check for matching first and last names and assign a match value.
   b. Use ACCUM to count up the paths.
   c. Check for an image.
   d. Depth=2. Compute the relevance score.

The complete query is shown below:

name_query.gsql

```
CREATE QUERY namesSimilar (vertex<NameUser> seed, string firstName, strin
FOR GRAPH gsql_demo
{
  # define a tuple to store neighbor score
  typedef tuple<vertex<NameUser> uid, float score> neighbor;
  # runtime variables used to compute neighbor score
  SumAccum<int> @count = 0;
  SumAccum<int> @depth = 0;
  SumAccum<int> @match= 0;
  SumAccum<float> @score = 0.0;
  SumAccum<int> @hasImgURL = 0;

  # global heap variable used to store final top k users, sorted by score
  #  in the neighbor tuple
  HeapAccum<neighbor>(k, score DESC) @@finalTopKUsers;

  # starting user
  StartP = {seed};

  # flag first level neighbor with @depth = 1
  # count number of incoming connections
  # flag match category
  # flag img_url count greater than 0
  # finally, push the user and their score into global top-k heap.
  FirstLevelConnection = SELECT u
        FROM   StartP -(NameConn)-> :u
        ACCUM u.@depth = 1, u.@count += 1,
            CASE WHEN u.firstname == firstName AND u.lastname == lastName
                    THEN  u.@match = 3
                WHEN u.firstname != firstName AND u.lastname == lastName
                    THEN  u.@match = 2
                WHEN u.firstname == firstName AND u.lastname != lastName
                    THEN u.@match = 1
            END,
            CASE WHEN u.imag_url != ""
                    THEN u.@hasImgURL = 1
            END
        POST-ACCUM @@finalTopKUsers += neighbor(u, u.@match * 100 + (4-u.

  # similarly, do the topk heap update using second level neighbor
  SecondLevelConnection = SELECT u2
        FROM   FirstLevelConnection -(NameConn)-> :u2
        WHERE  u2 != seed AND  u2.@depth != 1
        ACCUM  u2.@depth = 2, u2.@count +=1,
            CASE WHEN u2.firstname == firstName AND u2.lastname == lastNa
                    THEN  u2.@match = 3
                WHEN u2.firstname != firstName AND u2.lastname == lastNam
                    THEN  u2.@match = 2
```

```
                WHEN u2.firstname == firstName AND u2.lastname != lastNam
                    THEN u2.@match = 1
            END,
            CASE WHEN u2.imag_url !=""
                    THEN u2.@hasImgURL = 1
            END
        POST-ACCUM @@finalTopKUsers += neighbor(u2, u2.@match*100 + (4-u2

    #  print the result
    PRINT @@finalTopKUsers;
}
```

Query result

Starting from user 0, who is named "Michael Jackson", find the top 100 most similar persons, according to the scoring function described above.

```
//INSTALL QUERY namesSimilar
RUN QUERY namesSimilar (0,"michael","jackson",100)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@finalTopKUsers": [
    {
      "uid": "6",
      "score": 651
    },
    {
      "uid": "2",
      "score": 451
    },
    {
      "uid": "1",
      "score": 451
    },
    {
      "uid": "5",
      "score": 404
    },
    {
      "uid": "4",
      "score": 351
    },
    {
      "uid": "7",
      "score": 303
    },
    {
      "uid": "3",
      "score": 151
    }
  ]}]
}
```

Example 5. Content-Based Filtering Recommendation of Videos

This example demonstrates conditional loading to be selective about which data records to load into which vertices or edges.

Input Data: A network of video programs, a set of tags which describe each video, and a set of users who have watched and rated videos.

Query Task: Recommend video programs that a given user might like.

Step 1: Create Graph Schema

The principle behind content-based recommendation is that people are often interested in products which have attributes similar to the ones which they have selected in the past. Suppose we have a video store. If the store tracks what videos each customer has selected in the past, and also records attributes about its videos, it can use this data to recommend more videos to the customer. Formally, for an input user (seed), first find which videos the user has watched. Then, from all the watched videos, find the top k attributes. From the top k attributes, find the top n videos that the seed user has not watched.

This suggests that we should have a graph with three types of vertices: user, video, and attributes (of a video). The schema is shown below.

video_model.gsql

```
CREATE VERTEX VidUser (PRIMARY_ID user_id uint, content string, date_time
CREATE VERTEX Video (PRIMARY_ID content_id uint, content string, date_tim
CREATE VERTEX AttributeTag (PRIMARY_ID tag_id string, content string, att
CREATE UNDIRECTED EDGE Video_AttributeTag (FROM Video, TO AttributeTag, w
CREATE UNDIRECTED EDGE User_Video (FROM VidUser, TO Video, rating float D
```

Then run

```
CREATE GRAPH gsql_demo(*)
```

Step 2: Load Input Data

In this example, there is one data file which contains data for all three type of vertices – VidUser, Video, and AttributeTag. The first field of each line indicates the vertex type. Similarly, there is one edge data file for two types of edges – User_Video and Video_AttributeTag. The WHERE clause is used to conditionally load only certain data into each type of vertex or edge. Further, these data files do not contain information for every attribute. When "_" is used in the VALUES list of a LOAD statement, it means not to load data from the input. The default value will be written (or it will remain as it is, if there is already a vertex or edge with that ID).

Test data files

video_recommendation_v.csv

```
type,id,content
User,0,
User,1,
User,2,
User,3,
Video,0,v0
Video,1,v1
Video,2,v2
Video,3,v3
Video,4,v4
Tag,action,
Tag,comedy,
Tag,mystery,
Tag,technical,
```

video_recommendation_e.csv

```
type,from,to,rating
UV,0,0,6.8
UV,0,2,5.2
UV,0,3,10.0
UV,1,1,1.2
UV,2,0,7.4
UV,3,0,6.6
UV,3,4,8.4
VA,0,action,
VA,0,comedy,
VA,1,mystery,
VA,2,technical,
VA,2,mystery,
VA,2,action,
VA,3,comedy,
VA,4,technical,
VA,4,action,
```

Loading jobs

video_load.gsql

```
# define the loading job
CREATE LOADING JOB load_videoV FOR GRAPH gsql_demo {
  DEFINE FILENAME f1;
  LOAD f1
    TO VERTEX VidUser VALUES  ($1,_,_) WHERE $0 == "User",
    TO VERTEX Video VALUES ($1,$2,_) WHERE $0 == "Video",
    TO VERTEX AttributeTag VALUES ($1,_,_) WHERE $0 == "Tag";
}
CREATE LOADING JOB load_videoE FOR GRAPH gsql_demo {
  DEFINE FILENAME f2;
  LOAD f2
    TO EDGE User_Video VALUES  ($1,$2,$3, _) WHERE $0 == "UV",
    TO EDGE Video_AttributeTag VALUES  ($1,$2,_, _) WHERE $0 == "VA";
}

# load the data
RUN LOADING JOB load_videoV USING f1="../video/data/video_recommendation_
RUN LOADING JOB load_videoE USING f2="../video/data/video_recommendation_
```

Step 3: Query the data

The query has the three stages characteristic of content-based recommendation:

1. Find products (videos) previously selected
2. Find the top attributes of those products
3. Find the products which have the most attributes in common with the seed products

video_query.gsql

```
CREATE QUERY videoRecommendation (vertex<VidUser> seed, int k, int n) FOR
{
  OrAccum          @viewedBySeed;
  SumAccum<float>  @score;

  Start = {seed};

  # get viewed videos
  Viewed = SELECT v
           FROM   Start -(User_Video:e)-> Video:v
           ACCUM  v.@viewedBySeed += true,
                  v.@score += e.rating;

  # get attribute
  Attribute = SELECT att
              FROM Viewed:v -(Video_AttributeTag)-> AttributeTag:att
              ACCUM att.@score += v.@score
              ORDER BY  att.@score
              LIMIT k;

 # get recommended videos
  Recommend = SELECT v
              FROM Attribute:att -(Video_AttributeTag)-> Video:v
              WHERE v.@viewedBySeed != true
              ACCUM v.@score += att.@score
              ORDER BY v.@score DESC
              LIMIT n;

  PRINT Recommend;
}
```

Query result

Recommend up to 10 videos to user 0, using the top 10 attributes from the client's favorite videos.

```
//INSTALL QUERY videoRecommendation
RUN QUERY videoRecommendation (0, 10, 10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Recommend": [
      {
      "v_id": "4",
      "attributes": {
        "date_time": 0,
        "@score": 17.2,
        "@viewedBySeed": false,
        "content": "v4"
      },
      "v_type": "Video"
    },
    {
      "v_id": "1",
      "attributes": {
        "date_time": 0,
        "@score": 5.2,
        "@viewedBySeed": false,
        "content": "v1"
      },
      "v_type": "Video"
    }
  ]}]
}
```

Example 6. People You May Know

This example shows a technique for passing intermediate results from one stage to another.

Input Data : A social network with weighted connections.

Query Task: Recommend the Top K people you may know but who are not yet in your set of connections. Scoring is based on a variation of cosine similarity of two users:

```
      tgt.@edge_strength = e.strength
```

This is a way to "transport" a value as the query travels through the graph .

The graph schema and loading jobs:

Step 1

people_model.gsql

```
CREATE VERTEX Person (PRIMARY_ID id uint)
CREATE UNDIRECTED EDGE PersonConn (FROM Person, TO Person, strength float
```

$$score(A,B) = \sum \frac{connectionStrength(A \to x) \cdot connectionStrength(x \to B)}{degree(A) \cdot degree(B)}$$

This example shows that the computation of a moderately complex formula is simple in the GSQL language. It also demonstrates a technique of copying an attribute from an edge or a source vertex to the (temporary) accumulator of the edge's target vertex:

Then

```
CREATE GRAPH gsql_demo(*)
```

Step 2

people_load.gsql

```
# define the loading jobs
CREATE LOADING JOB load_peopleV FOR GRAPH gsql_demo {
  DEFINE FILENAME f1;
  LOAD f1 TO VERTEX Person VALUES ($0);
}
CREATE LOADING JOB load_peopleE FOR GRAPH gsql_demo {
  DEFINE FILENAME f2;
  LOAD f2 TO EDGE PersonConn VALUES ($0,$1,$2);
}

# load the data
RUN LOADING JOB load_peopleV USING f1="../people/data/people_user.dat"
RUN LOADING JOB load_peopleE USING f2="../people/data/people_conn.dat"
```

Test data:

people_user.dat

```
1
2
3
4
5
6
7
8
```

people_conn.dat

```
1,2,0.6
2,3,0.5
2,6,0.5
3,6,0.3
3,5,0.2
3,4,0.8
5,8,0.8
6,8,0.2
```

Step 3

If you have worked through the previous examples, you perhaps can now see that we need a two-stage query: from A to A's neighbors, and then from A's neighbors to their neighbors. Also, you may realize that we will use the ACCUM clause to perform summation in the second stage. But, how will we know during the second stage what was the strength of the first stage edge? By storing a copy of the edge's weight in an accumulator attached to the edge's target vertex, which becomes a source vertex in the second stage.

people_query.gsql

```
CREATE QUERY peopleYouMayKnow(vertex<Person> startP, int TopK) FOR GRAPH
{
  SumAccum<float> @edge_strength = 0;
  SumAccum<int>   @depth = 0;
  SumAccum<float> @sum = 0;
  SumAccum<float> @score = 0;
  SumAccum<int>   @@startPdegree = 0;

  Start = {startP};
  L1 = SELECT tgt
        FROM Start:src-(PersonConn:e)->Person:tgt
        ACCUM tgt.@edge_strength = e.strength, tgt.@depth=1, # copy edge
                  @@startPdegree += src.outdegree(); #  save seed outdegr

  # second level connections
  L2 = SELECT tgt2
        FROM L1:u-(PersonConn:e)->Person:tgt2
        WHERE tgt2 != startP AND tgt2.@depth != 1
        ACCUM tgt2.@sum += u.@edge_strength*e.strength
        POST-ACCUM tgt2.@score += tgt2.@sum/(@@startPdegree * tgt2.outdeg
        ORDER BY tgt2.@score DESC
        LIMIT TopK;
  #PRINT L2.@score;          # JSON output API version v1
  PRINT L2 [L2.@score];     # JSON output API version v2
}
```

⚠ In JSON output API v2, the PRINT syntax for a vertex set variable is different than the v1 syntax.

Query result:

Recommend up to 10 persons whom Person 1 might like to get to know.

```
# INSTALL QUERY peopleYouMayKnow
RUN QUERY peopleYouMayKnow (1, 10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"L2": [
    {
      "v_id": "6",
      "attributes": {"L2.@score": 0.1},
      "v_type": "Person"
    },
    {
      "v_id": "3",
      "attributes": {"L2.@score": 0.075},
      "v_type": "Person"
    }
  ]}]
}
```

Example 7. More Social Network Queries

Input Data: A social network in which each user has two attributes (besides their name): the time that they joined the network, and a boolean flag which says whether they are active or not.

Query Tasks: We show several query examples, making use the the time attribute and directed links between users.

Part 1: Create Graph

social_model.gsql

```
CREATE VERTEX SocialUser (PRIMARY_ID uid string, name string, isActive bo
CREATE DIRECTED EDGE SocialConn (FROM SocialUser, TO SocialUser) WITH REV
```

Then

```
CREATE GRAPH gsql_demo(*)
```

Part 2: Load Data

Test data:

social_users.csv

```
id,name,active,timestamp
0,luke,1,1400000000
1,john,1,1410000000
2,matthew,0,1420000000
3,mark,1,143000000
4,paul,1,144000000
5,steven,0,145000000
6,peter,1,146000000
7,james,1,147000000
8,joseph,1,148000000
9,thomas,1,149000000
```

social_connections.csv

```
0,1
0,2
0,3
0,4
0,5
1,3
1,4
1,5
1,6
0,7
7,0
7,3
7,4
7,5
0,8
8,3
8,4
0,9
9,3
```

We have two data files. For variety, we will create two loading jobs, one for each file. Moreover, we will define the specific file locations in the loading jobs themselves. Then, it is not necessary to provide the filepaths in the RUN LOADING JOB statements. Also, the file `social_users.csv` has a header, so we can use the column headings to refer to the columns.

social_load.gsql

```
# define the loading job
CREATE LOADING JOB load_social1 FOR GRAPH gsql_demo {
  LOAD "../social/data/social_users.csv"
    TO VERTEX SocialUser VALUES ($"id",$"name",$"active",$"timestamp")
    USING HEADER="true", QUOTE="double";
}
CREATE LOADING JOB load_social2 FOR GRAPH gsql_demo {
  LOAD "../social/data/social_connection.csv"
    TO EDGE SocialConn VALUES ($0, $1);
}

# load the data
RUN LOADING JOB load_social1
RUN LOADING JOB load_social2
```

Part 3 : Create, install, and run queries.

This case study presents four queries and their results, one at a time, so there are four separate "INSTALL QUERY" commands. Alternately, all four can be installed at once, which will execute faster than separate install commands:
`INSTALL QUERY socialFromUser, socialToUser, socialMutualConnections, socialOneWay`

or

`INSTALL QUERY ALL`

Q1 (socialFromUser): find users who have a direct connection from a given input user, with some filtering conditions on the candidate users' attributes

socialFromUser from social_query.gsql

```
CREATE QUERY socialFromUser(vertex<SocialUser> uid, bool is_active, int r
int reg_time_max, int k) FOR GRAPH gsql_demo
{
        L0={uid};
    RESULT = SELECT tgt
             FROM L0:u-(SocialConn)->SocialUser:tgt
             WHERE tgt.registration_timestamp >= reg_time_min AND
                   tgt.registration_timestamp <= reg_time_max AND
                   tgt.isActive == is_active
             LIMIT k;
    PRINT RESULT;
}
```

Test query and result:

```
#INSTALL QUERY socialFromUser
RUN QUERY socialFromUser("0", "true", 0, 147000000, 10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"RESULT": [
    {
      "v_id": "7",
      "attributes": {
        "registration_timestamp": 147000000,
        "name": "james",
        "isActive": true
      },
      "v_type": "SocialUser"
    },
    {
      "v_id": "4",
      "attributes": {
        "registration_timestamp": 144000000,
        "name": "paul",
        "isActive": true
      },
      "v_type": "SocialUser"
    },
    {
      "v_id": "3",
      "attributes": {
        "registration_timestamp": 143000000,
        "name": "mark",
        "isActive": true
      },
      "v_type": "SocialUser"
    }
  ]}]
}
```

Q2 (socialToUser): similar to Q1, but return users who have a connection pointing *to* the input user.

socialToUser from social_query.gsql

```
CREATE QUERY socialToUser(vertex<SocialUser> uid, bool is_active, int reg
int reg_time_max, int K) FOR GRAPH gsql_demo
{
        L0={uid};
    Result = SELECT tgt
             FROM L0:u-(reverse_conn)->SocialUser:tgt
             WHERE tgt.registration_timestamp >= reg_time_min AND
                   tgt.registration_timestamp <= reg_time_max AND
                   tgt.isActive == is_active
             LIMIT K;
    PRINT Result;
}
```

Test query and result:

```
#INSTALL QUERY socialToUser
RUN QUERY socialToUser("4", "true", 0, 150000000, 10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [
    {
      "v_id": "8",
      "attributes": {
        "registration_timestamp": 148000000,
        "name": "joseph",
        "isActive": true
      },
      "v_type": "SocialUser"
    },
    {
      "v_id": "7",
      "attributes": {
        "registration_timestamp": 147000000,
        "name": "james",
        "isActive": true
      },
      "v_type": "SocialUser"
    }
  ]}]
}
```

Q3 (socialMutualConnections): return the set of users who have connections from both input user A and input user B.

socialMutualConnection from social_query.gsql

```
CREATE QUERY socialMutualConnections(vertex<SocialUser> uid1, vertex<Soci
int reg_time_min, int reg_time_max, int k) FOR GRAPH gsql_demo
{
    SumAccum<int> @cnt =0;

    Start = {uid1,uid2};
    Result = SELECT tgt
             FROM  Start-(SocialConn)->SocialUser:tgt
             WHERE tgt.registration_timestamp >= reg_time_min AND
                   tgt.registration_timestamp <= reg_time_max AND
                   tgt.isActive == is_active
             ACCUM tgt.@cnt +=1
             HAVING tgt.@cnt == 2
             LIMIT k;
PRINT Result;
}
```

Test query and result:

```
#INSTALL QUERY socialMutualConnections
RUN QUERY socialMutualConnections("1", "7", "false", 0, 2000000000, 10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "5",
    "attributes": {
      "registration_timestamp": 145000000,
      "@cnt": 2,
      "name": "steven",
      "isActive": false
    },
    "v_type": "SocialUser"
  }]}]
}
```

Q4 (socialOneWay): find all A→B user relationships such that there is an edge from A to B but there is no edge from B to A, and also requires that A and B connect to at least some number of common friends.

socialOneWay from social_query.gsql

```
CREATE QUERY socialOneWay(int mutual_contacts_min = 5, int mutual_contact
FOR GRAPH gsql_demo
{
    typedef tuple<vertex<SocialUser> id, string name, int cnt> recTuple;
    # SumAccum<list<recTuple>> @recList; # v0.1 to v0.1.2
    ListAccum<recTuple> @recList; # v0.2

    Start = {SocialUser.*};

    Result = SELECT B
        FROM Start:A-(SocialConn)->SocialUser:B
          #  B.neighbors() is a built-in function which returns the list
          #  B.neighbors('edgeType1') returns only the neighbors connecte
        WHERE B NOT IN A.neighbors("reverse_conn") AND
            COUNT(A.neighbors("SocialConn") INTERSECT B.neighbors("Social
            COUNT(A.neighbors("SocialConn") INTERSECT B.neighbors("Social
        ACCUM  B.@recList += recTuple(A, A.name, COUNT(A.neighbors("Socia
    PRINT Result; # the result includes B's static attributes and B.@from
}
```

Test query and result: There are three such pairs

1. From vertex 0 to 1. Vertices 0 and 1 have 3 neighbors in common.
2. From vertex 0 to 8. Vertices 0 and 8 have 2 neighbors in common.
3. From vertex 0 to 9. Vertices 0 and 9 have 1 neighbor in common.

```
//INSTALL QUERY socialOneWay
RUN QUERY socialOneWay(1,10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [
    {
      "v_id": "8",
      "attributes": {
        "registration_timestamp": 148000000,
        "name": "joseph",
        "isActive": true,
        "@recList": [{
          "name": "luke",
          "cnt": 2,
          "id": "0"
        }]
      },
      "v_type": "SocialUser"
    },
    {
      "v_id": "9",
      "attributes": {
        "registration_timestamp": 149000000,
        "name": "thomas",
        "isActive": true,
        "@recList": [{
          "name": "luke",
          "cnt": 1,
          "id": "0"
        }]
      },
      "v_type": "SocialUser"
    },
    {
      "v_id": "1",
      "attributes": {
        "registration_timestamp": 1410000000,
        "name": "john",
        "isActive": true,
        "@recList": [{
          "name": "luke",
          "cnt": 3,
```

```
            "id": "0"
          }]
        },
        "v_type": "SocialUser"
      }
    ]}]
  }
```


Suggested variant query:

- socialTwoWay: Find all A↔B relationships such that there are connected edges both from A to B and from B to A, and A and B have some common neighbors.

Test query and result:

There is one such pair (0, 7), but the query reports it twice: first as (7, 0) and then as (0, 7). Vertices 7 and 0 have 3 neighbors in common.

```
RUN QUERY socialTwoWay(1,10)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [
    {
      "v_id": "1",
      "attributes": {
        "registration_timestamp": 1400000000,
        "name": "luke",
        "isActive": true,
        "@recList": [{
          "name": "james",
          "cnt": 3,
          "id": "7"
        }]
      },
      "v_type": "SocialUser"
    },
    {
      "v_id": "7",
      "attributes": {
        "registration_timestamp": 147000000,
        "name": "james",
        "isActive": true,
        "@recList": [{
          "name": "luke",
          "cnt": 3,
          "id": "0"
        }]
      },
      "v_type": "SocialUser"
    }
  ]}]
}
```

Example 8.

TigerGraph Graph Data Science Library

Overview

TigerGraph In-Database Graph Data Science Library is a collection of expertly written GSQL queries, each of which implements a standard graph algorithm. Each algorithm is ready to be installed and used, either as a stand-alone query or as a building block of a larger analytics application.

We renamed our library (formerly known as GSQL Graph Algorithm Library) to emphasize our focus on graph data science. As the world's only scalable graph analytics platform, TigerGraph is committed to providing the best graph analytics framework for data scientists.

GSQL running on the TigerGraph platform is particularly well-suited for graph algorithms for several reasons:

- **Turing-complete** with full support for imperative and procedural programming, ideal for algorithmic computation.
- **Parallel and Distributed Processing,** enabling computations on larger graphs.
- **User-Extensible**. Because the algorithms are written in standard GSQL and compiled by the user, they are easy to modify and customize.
- **Open-Source**. Users can study the GSQL implementations to learn by example, and they can develop and submit additions to the library.

Library Structure

You can download the library from Github: https://github.com/tigergraph/gsql-graph-algorithm ↗

The library contains two folders: `algorithms` and `graphs`.

`algorithms`

The `algorithms` folder contains the GSQL implementation of all the graph algorithms offered by the library. Within the algorithms folder are six subfolders that group the algorithms into six classes:

- Centrality
- Classification
- Community
- Node Embeddings
- Path
- Similarity

`graphs`

The `graphs` folder contains small sample graphs that you can use to experiment with the algorithms. In this document, we use the test graphs to show you the expected result for each algorithm. The graphs are small enough that you can manually calculate and sometimes intuitively see what the answers should be.

Release Branches

Starting with TigerGraph product version 2.6, the Library has release branches:

- **Product version branches** (2.6, 3.0, etc.) are snapshots created shortly after a product version is released. They contain the best version of the graph algorithm library at the time of that product version's initial release. They will not be updated, except to fix bugs.
- **Master branch**: the newest released version. This should be at least as new as the newest. It may contain new or improved algorithms.
- Other branches are development branches.

It is possible to run newer algorithms on an older product version, as long as the algorithm does not rely on features available only in newer product versions.

Run an algorithm

All GSQL graph algorithms are schema-free, which means they are ready to use with any graph, regardless of the graph's data model or schema. The algorithms have run-time input parameters for the vertex type(s), edge type(s), and attributes which the user wishes to use.

To use an algorithm, the algorithm (GSQL query) must first be installed. If your database is on a distributed cluster, you should use the `DISTRIBUTED` option when installing the query to install it in Distributed Query Mode.

Running an algorithm is the same as running a GSQL query. For example, if you selected the JSON option for `pageRank`, you could run it from GSQL as below:

```
GSQL > RUN QUERY tg_pageRank("Page","Links_to",_,30,_,50,_,_,_)
```

Installing a query also creates a REST endpoint. The same query could be run thus:

```
curl -X GET 'http://127.0.0.1:9000/query/alg_graph/pageRank?v_type=Page&e
```

GSQL lets you run queries from within other queries. This means you can use a library algorithm as a building block for more complex analytics.

Centrality Algorithms

PageRank

The PageRank algorithm measures the influence of each vertex on every other vertex. PageRank influence is defined recursively: a vertex's influence is based on the influence of the vertices which refer to it. A vertex's influence tends to increase if (1) it has more referring vertices or if (2) its referring vertices have higher influence. The analogy to social influence is clear.

A common way of interpreting PageRank value is through the Random Network Surfer model. A vertex's PageRank score is **proportional to the probability that a random network surfer will be at that vertex at any given time. A vertex with a high PageRank score is a vertex that is frequently visited**, assuming that vertices are visited according to the following Random Surfer scheme:

- Assume a person travels or surfs across a network's structure, moving from vertex to vertex in a long series of rounds.
- The surfer can start anywhere. This start-anywhere property is part of the magic of PageRank, meaning the score is a truly fundamental property of the graph structure itself.
- Each round, the surfer randomly picks one of the outward connections from the surfer's current location. The surfer repeats this random walk for a long time.
- But wait. The surfer doesn't always follow the network's connection structure. There is a probability (*1-damping*, to be precise), that the surfer will ignore the structure and will magically teleport to a random vertex.

For more information, see the Google paper on PageRank ↗.

Specifications

```
tg_pageRank (STRING v_type, STRING e_type,  FLOAT max_change=0.001, INT m
```

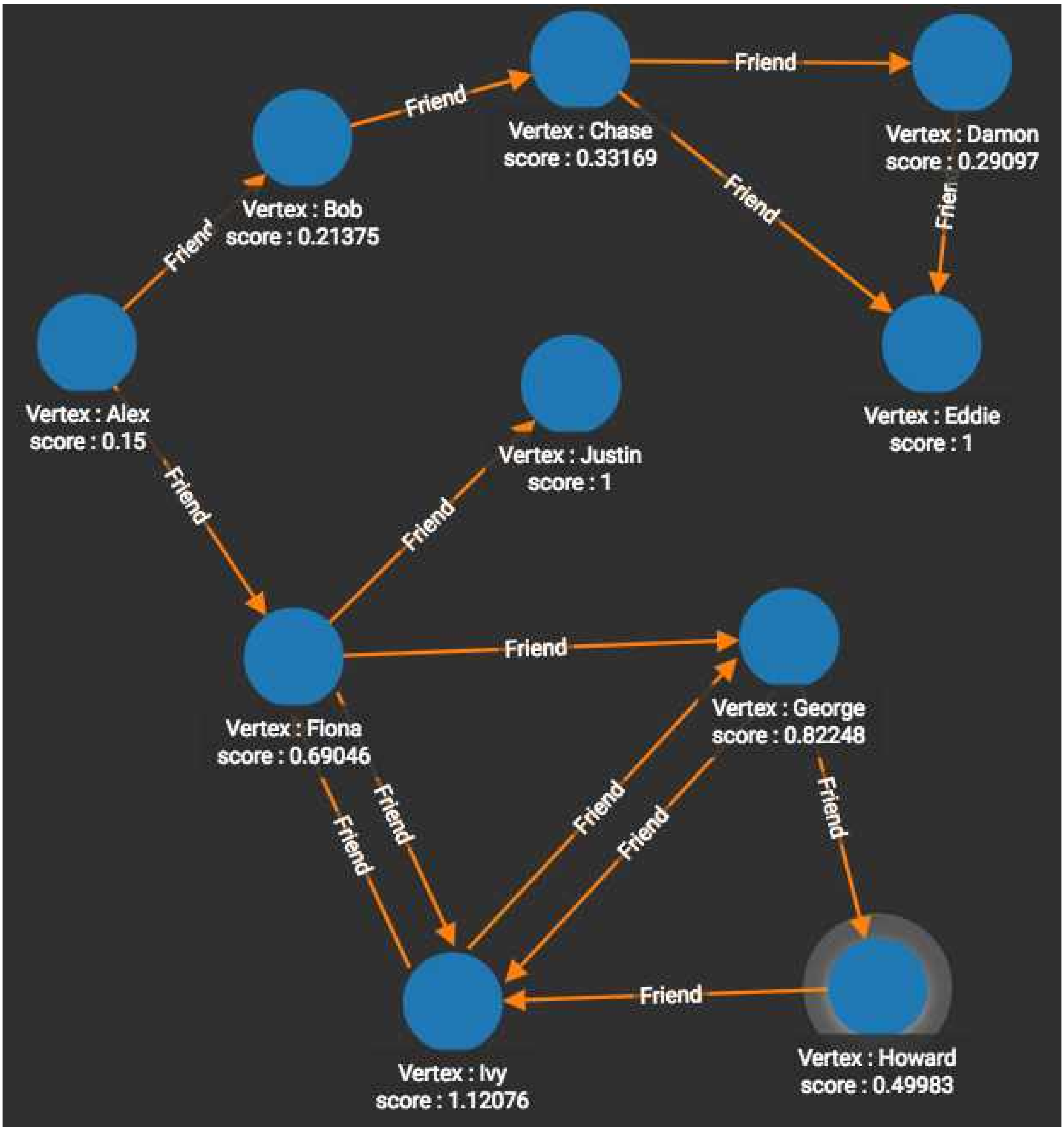
Characteristic	Value
Result	Computes a PageRank value (FLOAT type) for each vertex.

Input Parameters	• `STRING v_type`: Names of vertex type to use • `STRING e_type`: Names of edge type to use • `FLOAT max_change`: PageRank will stop iterating when the largest difference between any vertex's current score and its previous score ≤ `max_change.` That is, the scores have become very stable and are changing by less than `max_change` from one iteration to the next. • `INT max_iter`: Maximum number of iterations. • `FLOAT damping`: Fraction of score that is due to the score of neighbors. The balance (1 - damping) is a minimum baseline score that every vertex receives. • `INT top_k`: Sort the scores highest first and output only this many scores • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store PageRank values (FLOAT) to this attribute • `STRING file_path`**:** If not empty, write output to this file. • `BOOL display_edges`: If true, include the graph's edges in the JSON output, so that the full graph can be displayed.
Result Size	V = number of vertices
Time Complexity	O(E*k), E = number of edges, k = number of iterations. The number of iterations is data-dependent, but the user can set a maximum. Parallel processing reduces the time needed for computation.

Example

```
# Use _ for default values RUN QUERY tg_pageRank("Person", "Friend", 0.0
```

We ran pageRank on our test10 graph (using Friend edges) with the following parameter values: damping=0.85, max_change=0.001, and max_iter=25. We see that Ivy (center bottom) has the highest pageRank score (1.12). This makes sense since there are 3 neighboring persons who point to Ivy, more than for any other person. Eddie and Justin have scores of exactly 1 because they do not have any out-edges. This is an artifact of our particular version pageRank. Likewise, Alex has a score of 0.15, which is (1-damping), because Alex has no in-edges.



Visualized results of example query on social10 graph, with Friend edges

Article Rank (Beta)

ArticleRank is an algorithm that has been derived from the PageRank algorithm to measure the influence of journal articles.

Page Rank assumes that relationships originating from low-degree nodes have a higher influence than relationships from high-degree nodes. Article Rank modifies the formula in such a way that it retains the basic PageRank methodology but lowers the influence of low-degree nodes.

The Article Rank of a node *v* at iteration *i* is defined as:

$$ArticleRank_i(v) = (1-d) + d \sum_{w \in N_{in}(v)} \frac{ArticleRank_{i-1}(w)}{|N_{out}(w)| + \overline{N_{out}}}$$

Within the formula:

- *Nin(v)* are the incoming neighbors and *Nout(v)* are the outgoing neighbors of node *v*.
- *d* is a damping factor in *[0, 1]*, usually set to 0.85.
- *Nout* is the average out-degree

For more information, see [ArticleRank: a PageRank-based alternative to numbers of citations for analysing citation network ↗](#).

Specifications

```
CREATE QUERY tg_article_rank (STRING v_type, STRING e_type,
 FLOAT max_change=0.001, INT max_iter=25, FLOAT damping=0.85, INT top_k =
 BOOL print_accum = TRUE, STRING result_attr =  "", STRING file_path = ""
```

Parameters

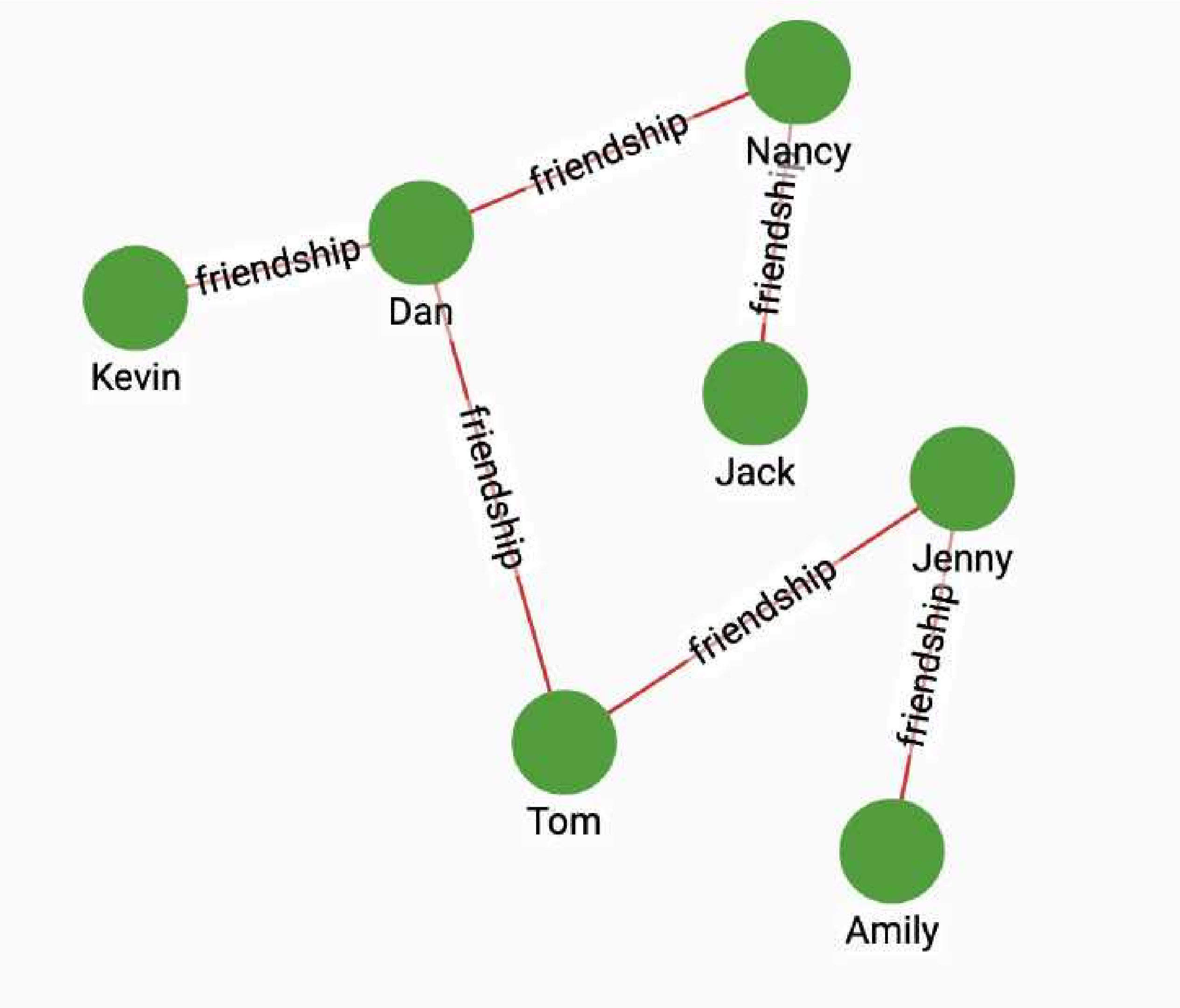
Name	Description	Data type
`v_type`	A vertex type.	`STRING`
`e_type`	An edge type.	`STRING`
`max_change`	Article Rank will stop iterating when the largest difference between any vertex's current score and its previous score ≤ `max_change.` That is, the scores have become very stable and are changing by less than `max_change` from one iteration to the next.	`FLOAT`
`max_iter`	Maximum number of iterations.	`INT`
`damping`	The damping factor. Usually set to 0.85.	`FLOAT`
`top_k`	The number of results with the highest scores to return.	`INT`
`print_accum`	If true, print JSON output.	`BOOL`
`result_attr`	If true, store the article rank score of each vertex in this attribute.	`STRING`
`file_path`	If true, output CSV to this file.	`STRING`

Return value

The article rank score for each vertex.

Example

Suppose we have the following graph:



By running Article Rank on the graph, we will see that the vertex with the highest score is Dan:

Query

```
RUN QUERY tg_article_rank ("person", "friendship", _, _, _, _, _)
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@topScores": [
    {
      "score": 2348294.75,
      "Vertex_ID": "Dan"
    },
    {
      "score": 1863160.625,
      "Vertex_ID": "Jenny"
    },
    {
      "score": 1442890.5,
      "Vertex_ID": "Tom"
    },
    {
      "score": 1053484.625,
      "Vertex_ID": "Nancy"
    },
    {
      "score": 739327.9375,
      "Vertex_ID": "Kevin"
    },
    {
      "score": 703562.75,
      "Vertex_ID": "Amily"
    },
    {
      "score": 498013.25,
      "Vertex_ID": "Jack"
    }
  ]}]
}
```

Weighted PageRank

The only difference between weighted PageRank and standard PageRank is that edges have weights, and the influence that a vertex receives from an in-neighbor is multiplied by the weight of the in-edge.

Specifications

```
tg_pageRank_wt (SET<STRING> v_type, SET<STRING> e_type, STRING wt_attr,
  FLOAT max_change=0.001, INT max_iter=25, FLOAT damping=0.85, INT top_k=
   BOOL print_accum = TRUE, STRING result_attr =  "", STRING file_path =
   BOOL display_edges = FALSE)
```

Characteristic	Value
Result	Computes a weighted PageRank value (FLOAT type) for each vertex.
Input Parameters	 • **`STRING v_type`**: Names of vertex type to use • **`STRING e_type`**: Names of edge type to use • **`STRING wt_attr`**: Name of edge weight attribute • **`FLOAT max_change`**: PageRank will stop iterating when the largest difference between any vertex's current score and its previous score ≤ `max_change`. That is, the scores have become very stable and are changing by less than `max_change` from one iteration to the next. • **`INT max_iter`**: Maximum number of iterations. • **`FLOAT damping`**: Fraction of score that is due to the score of neighbors. The balance (1 - damping) is a minimum

	baseline score that every vertex receives. • `INT top_k`: Sort the scores highest first and output only this many scores • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store PageRank values (FLOAT) to this attribute • `STRING file_path`: If not empty, write output to this file. • `BOOL display_edges`: If true, include the graph's edges in the JSON output, so that the full graph can be displayed.
Result Size	V = number of vertices
Time Complexity	O(E*k), E = number of edges, k = number of iterations. The number of iterations is data-dependent, but the user can set a maximum. Parallel processing reduces the time needed for computation.
Graph Types	Directed edges

Personalized PageRank

In the original PageRank, the damping factor is the probability of the surfer continues browsing at each step. The surfer may also stop browsing and start again from a random vertex. In personalized PageRank, the surfer can only start browsing from a given set of source vertices both at the beginning and after stopping.

Specifications

```
tg_pageRank_pers(SET<VERTEX> source, STRING e_type,
FLOAT max_change=0.001, INT max_iter=25, FLOAT damping = 0.85, INT top_k
BOOL print_accum = TRUE, STRING result_attr = "", STRING file_path = "")
```

Characteristic	Value
Result	Computes a personalized PageRank value (FLOAT type) for each vertex.
Input Parameters	• **`SET<VERTEX> source`**: Set of seed vertices • **`STRING e_type`**: Names of edge type to use • **`FLOAT max_change`**: PageRank will stop iterating when the largest difference between any vertex's current score and its previous score ≤ `max_change`. That is, the scores have become very stable and are changing by less than `max_change` from one iteration to the next. • **`INT max_iter`**: Maximum number of iterations. • **`FLOAT damping`**: Fraction of score that is due to the score of neighbors. The balance (1 - damping) is a minimum baseline score that every vertex receives.

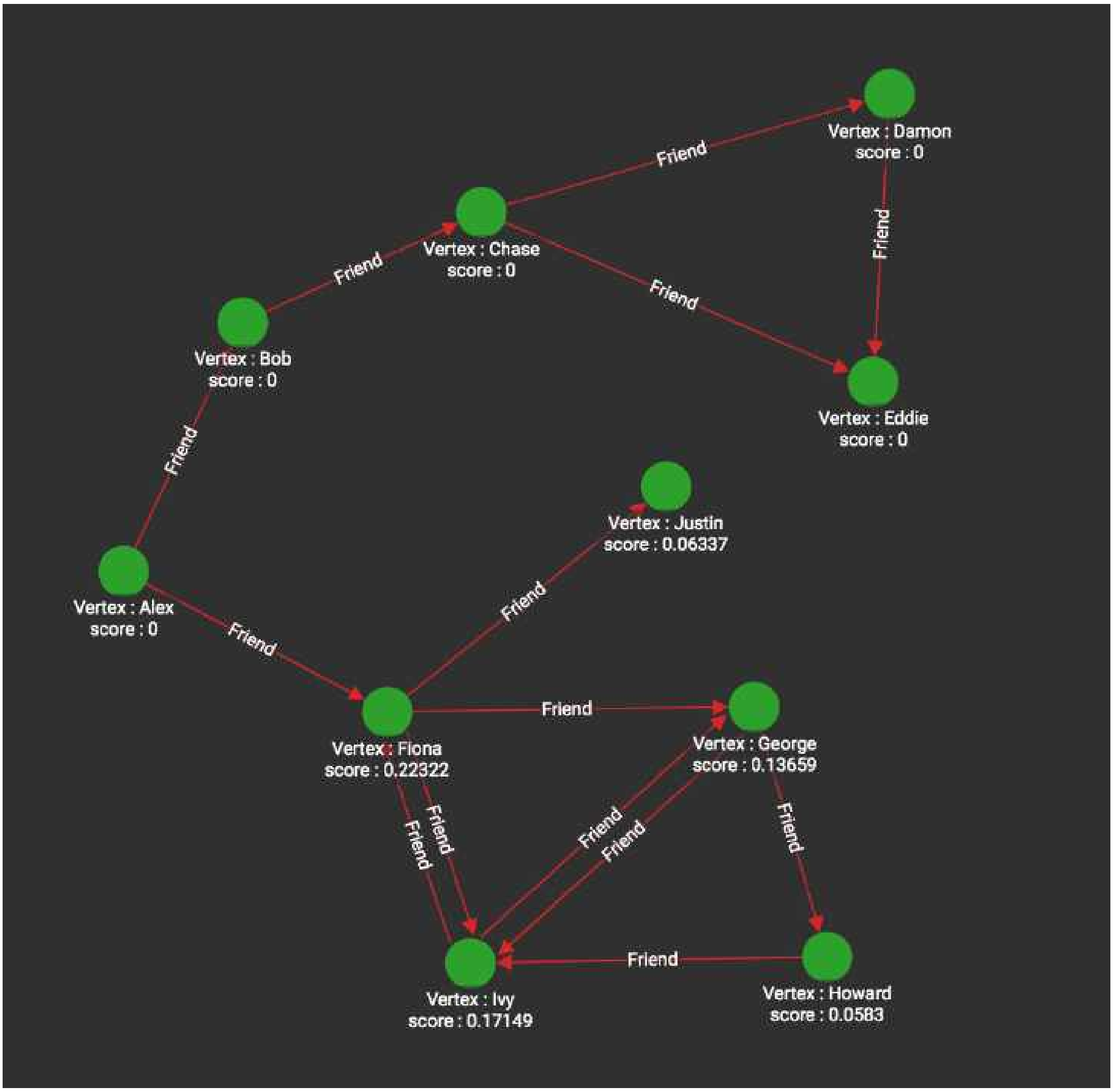
	• `INT top_k`: Sort the scores highest first and output only this many scores • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store PageRank values (FLOAT) to this attribute • `STRING file_path`: If not empty, write output to this file.
Result Size	V = number of vertices
Time Complexity	O(E*k), E = number of edges, k = number of iterations. The number of iterations is data-dependent, but the user can set a maximum. Parallel processing reduces the time needed for computation.
Graph Types	Directed edges

Example

We ran Personalized PageRank on the graph `social10` using Friend edges with the following parameter values:

```
# Using "_" to use default values
RUN QUERY tg_pageRank_pers([("Fiona","Person")], "Friend", _, _, _, _, _,
_)
```

In this case, the random walker can only start or restart walking from Fiona. In the figure below, we see that Fiona has the highest PageRank score in the result. Ivy and George have the next highest scores because they are direct out-neighbors of Ivy and there are looping paths that lead back to them again. Half of the vertices have a score of 0 since they can not be reached from Fiona.



Visualized results of example query on social10 graph, with Friend edges

Betweenness Centrality

The Betweenness Centrality of a vertex is defined as the number of shortest paths that pass through this vertex, divided by the total number of shortest paths. That is

$$BC(v) = \sum_{s \neq v \neq t} PD_{st}(v) = \sum_{s \neq v \neq t} SP_{st}(v)/SP_{st},$$

where $PD$is called the pair dependency, $SP_{st}$is the total number of shortest paths from node `s` to node `t` and $SP_{st}(v)$is the number of those paths that pass through `v`.

The TigerGraph implementation is based on A Faster Algorithm for Betweenness Centrality by Ulrik Brandes, Journal of Mathematical Sociology 25(2):163-177, (2001). For every vertex `s` in the graph, the pair dependency starting from vertex `s` to all other vertices `t` via all other vertices `v` is computed first,

$$PD_{s*}(v) = \sum_{t: s \in V} PD_{st}(v).$$

Then betweenness centrality is computed as

$$BC(v) = \sum_{s: s \in V} PD_{s*}(v)/2.$$

According to Brandes, the accumulated pair dependency can be calculated as

$$PD_{s*}(v) = \sum_{w: v \in P_s(w)} SP_{sv}(v)/SP_{sw} \cdot (1 + PD_{s*}(w)),$$

where$P_s(w)$, the set of predecessors of vertex `w` on shortest paths from `s`, is defined as

$$P_s(w) = \{u \in V : \{u, w\} \in E, dist(s, w) = dist(s, u) + dist(u, w)\}.$$

For every vertex, the algorithm works in two phases. The first phase calculates the number of shortest paths passing through each vertex. Then starting from the vertex on the most outside layer in a non-incremental order with pair dependency initial value of 0, traverse back to the starting vertex

This algorithm query employs a subquery called **bc_subquery**. Both queries are needed to run the algorithm.

Specifications

```
CREATE QUERY tg_betweenness_cent(SET<STRING> v_type, SET<STRING> e_type,
STRING re_type,INT max_hops=10, INT top_k=100, BOOL print_accum = True,
STRING result_attr = "", STRING file_path = "", BOOL display_edges = FALS
```

Parameters

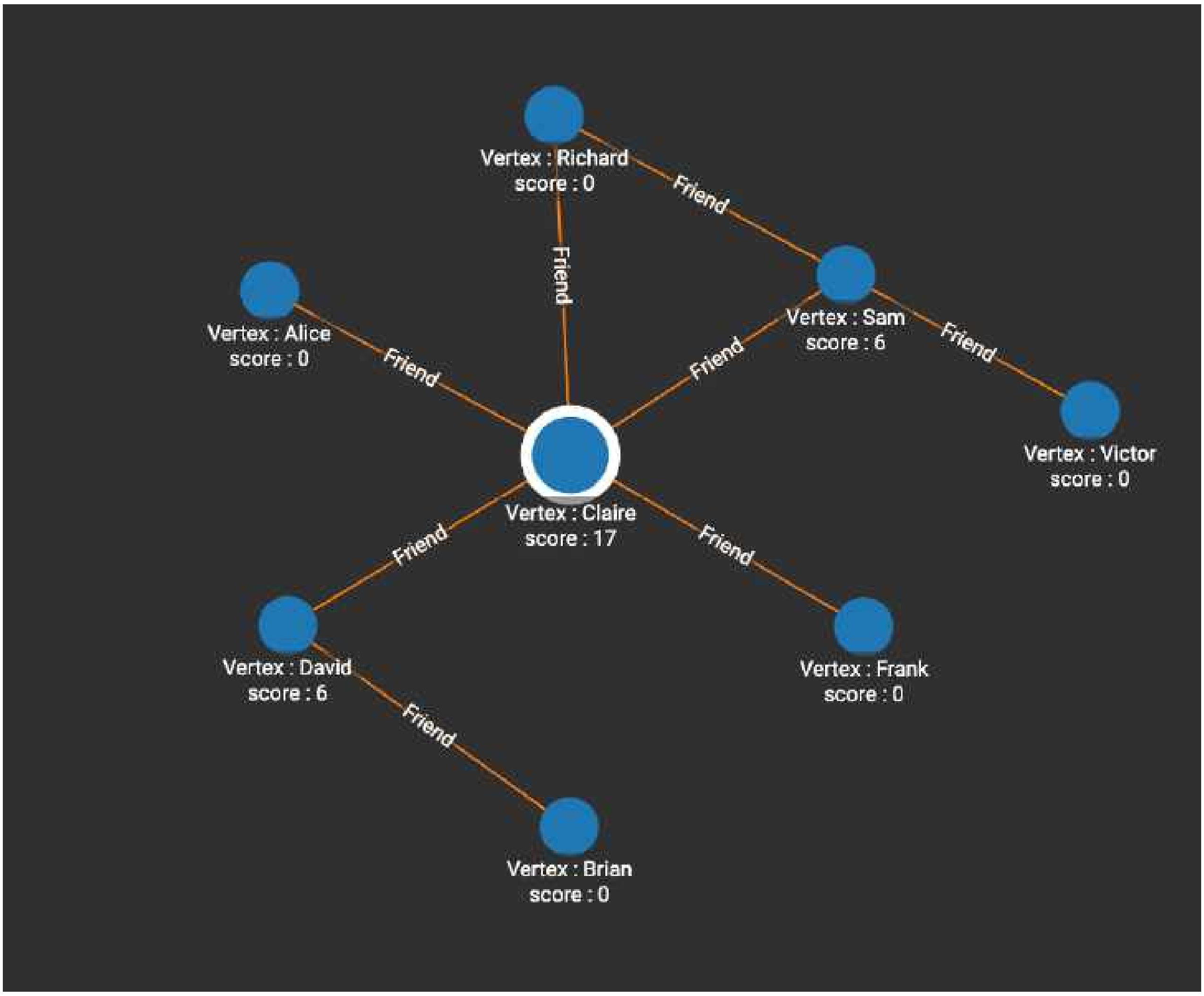
Characteristic	Value
Result	Computes a Betweenness Centrality value (FLOAT type) for each vertex.
Required Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `SET<STRING> re_type:` Names of reverse edge types to use • `INT max_hops`: If >=0, look only this far from each vertex • `INT top_k`: Sort the scores highest first and output only this many scores • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store centrality values (FLOAT) to this attribute • `STRING file_path`: If not empty, write output to this file. • `BOOL display_edges`: If true, include the graph's edges in the JSON output,

	so that the full graph can be displayed.
Result Size	V = number of vertices
Time Complexity	O(E*V), E = number of edges, V = number of vertices. Considering the high time cost of running this algorithm on a big graph, the users can set a maximum number of iterations. Parallel processing reduces the time needed for computation.
Graph Types	Undirected edges, Unweighted edges

Example

In the example below, Claire is in the very center of the graph and has the highest betweenness centrality. Six shortest paths pass through Sam (i.e. paths from Victor to all other 6 people except for Sam and Victor), so the score of Sam is 6. David also has a score of 6, since Brian has 6 paths to other people that pass through David.

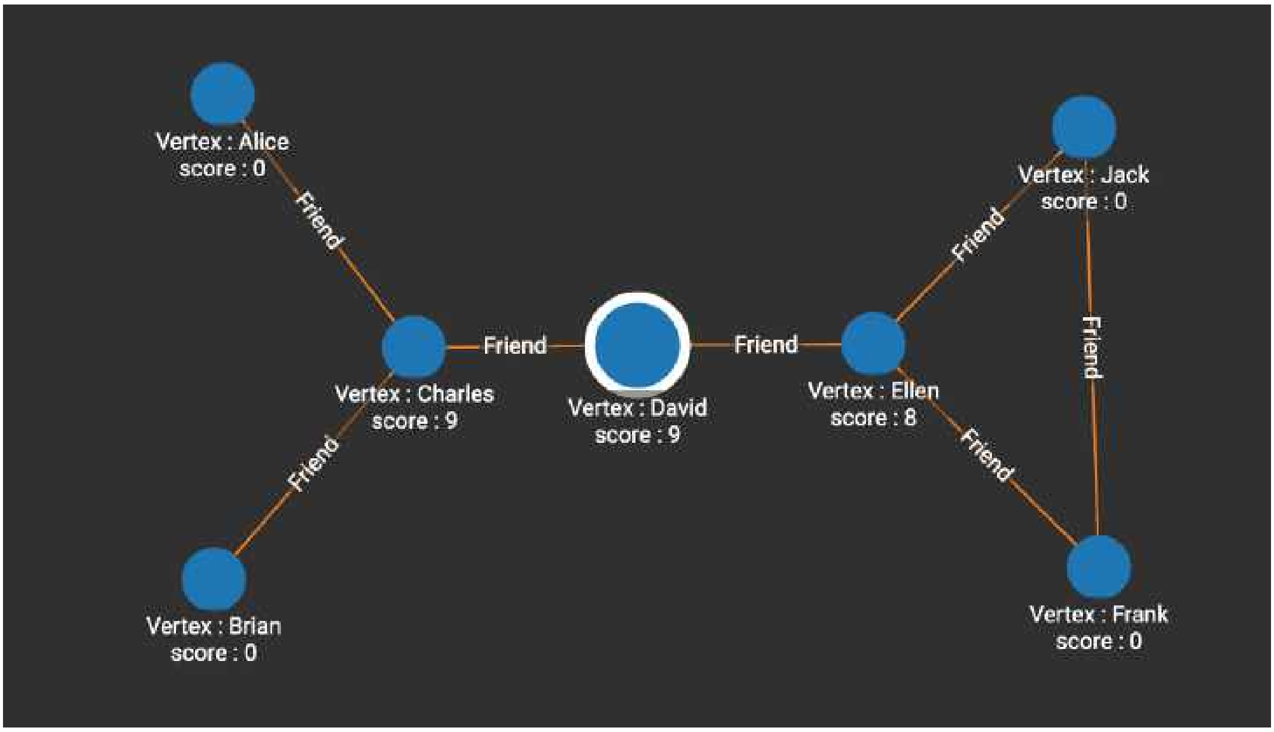
```
# Use _ for default values
RUN QUERY tg_betweenness_cent(["Person"], ["Friend"], _, _, _, _, _, _)
```



Visualized results of example query on a social graph with undirected edges Friend


```
[
  {
    "@@BC": {
      "Alice": 0,
      "Frank": 0,
      "Claire": 17,
      "Sam": 6,
      "Brian": 0,
      "David": 6,
      "Richard": 0,
      "Victor": 0
    }
  }
]
```


In the following example, both Charles and David have 9 shortest paths passing through them. Ellen is in a similar position as Charles, but her centrality is weakened due to the path between Frank and Jack.



Visualized results of example query on a social graph with undirected edges Friend

```
[
  {
    "@@BC": {
      "Alice": 0,
      "Frank": 0,
      "Charles": 9,
      "Ellen": 8,
      "Brian": 0,
      "David": 9,
      "Jack": 0
    }
  }
]
```

Eigenvector Centrality (Beta)

Eigenvector centrality (also called **eigencentrality** or **prestige score**) is a measure of the influence of a vertex in a network. Relative scores are assigned to all vertices in the network based on the concept that connections to high-scoring vertices contribute more to the score of the vertex in question than equal connections to low-scoring vertices. A high eigenvector score means that a vertex is connected to many vertices who themselves have high scores.

For more information, see Eigenvector centrality ↗.

Specification

```
CREATE QUERY tg_eigenvector_cent(SET<STRING> v_type, SET<STRING> e_type,
 INT maxIter = 100, FLOAT convLimit = 0.000001, INT top_k = 100,
 BOOL print_accum = True, STRING result_attr = "", STRING file_path = ""
 )
```

Parameters

Name	Description	Data type
`v_type`	Vertex types to assign scores to.	`SET<STRING>`
`e_type`	Edge types to traverse.	`SET<STRING>`
`maxIter`	Maximum number of iteration.	`INT`
`convLimit`	The convergence limit.	`FLOAT`
`top_k`	The number of vertices with the highest scores to return.	`INT`

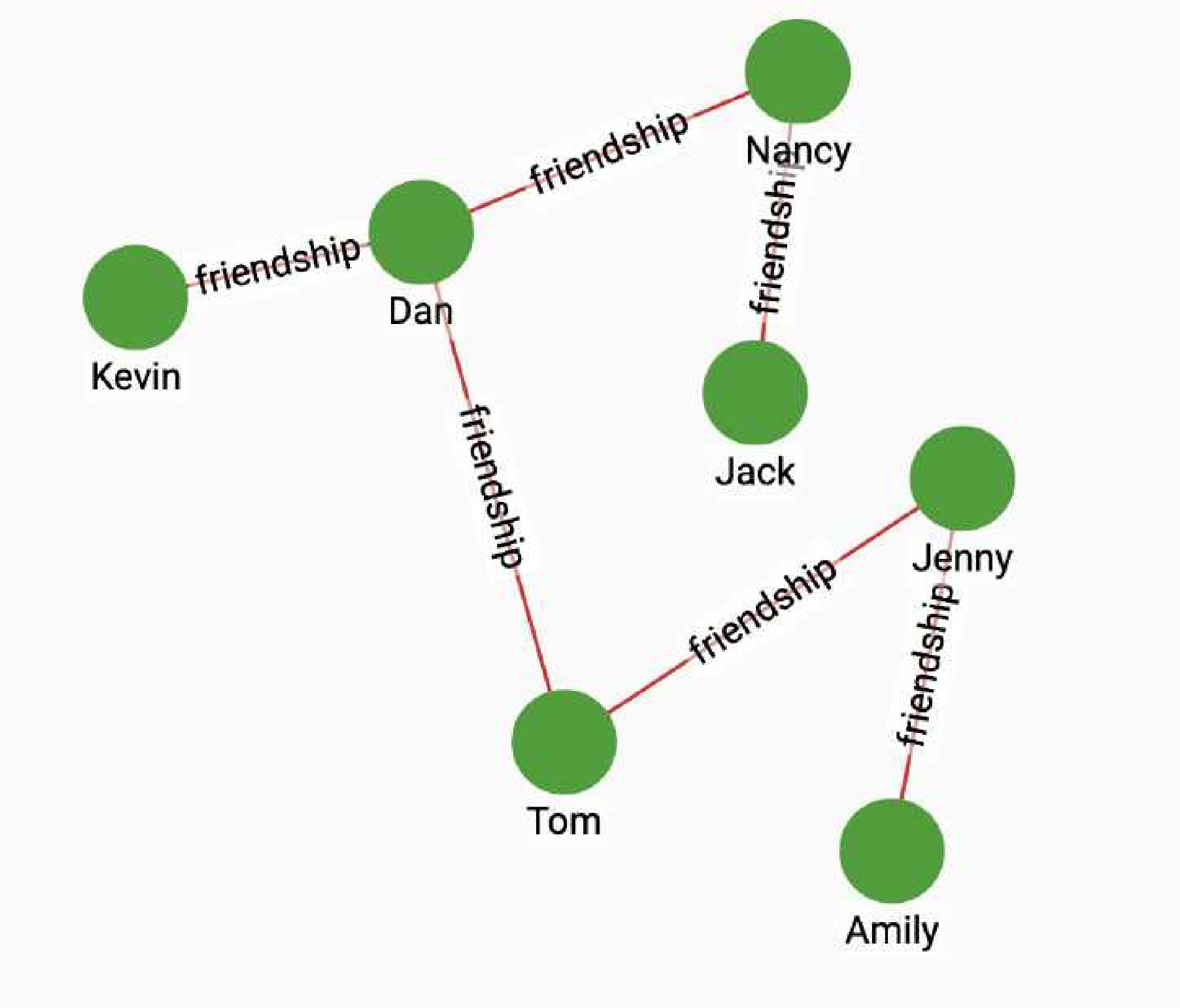
`print_accum`	If true, print results to JSON output.	`BOOL`
`result_attr`	If not empty, save the score of each vertex to this attribute.	`STRING`

Return value

The vertices with the highest Eigenvector centrality scores along with their score.

Example

Suppose we have the following graph:



Running the algorithm on the graph will show that Dan has the highest centrality score.

Query

```
RUN QUERY tg_eigenvector_cent(["person"], ["friendship"])
```

Result

```json
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"top_scores": [
    {
      "score": 0.59598,
      "Vertex_ID": "Dan"
    },
    {
      "score": 0.50223,
      "Vertex_ID": "Jenny"
    },
    {
      "score": 0.44381,
      "Vertex_ID": "Tom"
    },
    {
      "score": 0.28786,
      "Vertex_ID": "Nancy"
    },
    {
      "score": 0.24085,
      "Vertex_ID": "Kevin"
    },
    {
      "score": 0.20296,
      "Vertex_ID": "Amily"
    },
    {
      "score": 0.11633,
      "Vertex_ID": "Jack"
    }
  ]}]
}
```

Degree Centrality (Beta)

Degree centrality is defined as the number of edges incident upon a node (i.e., the number of ties that a node has). The degree can be interpreted in terms of the immediate risk of a node for catching whatever is flowing through the network (such as a virus, or some information).

Specification

```
CREATE QUERY tg_degree_cent(SET<STRING> v_type, SET<STRING> e_type,
  SET<STRING> re_type, BOOL in_degree = TRUE, BOOL out_degree = TRUE,
  INT top_k=100, BOOL print_accum = True, STRING result_attr = "",
  STRING file_path = "")
```

Parameters

Name	Description	Data type
`v_type`	A set of vertex types.	`SET<STRING>`
`e_type`	A set of edge types.	`SET<STRING>`
`re_type`	A set of reverse edge types. If an edge is undirected, put the edge name in the set as well.	\` `SET<STRING>`
`in_degree`	Boolean value that indicates whether to count the incoming edges as part of a vertex's degree centrality.	`BOOL`
`out_degree`	Boolean value that indicates whether to count the outgoing edges as part of a vertex's degree centrality.	`BOOL`

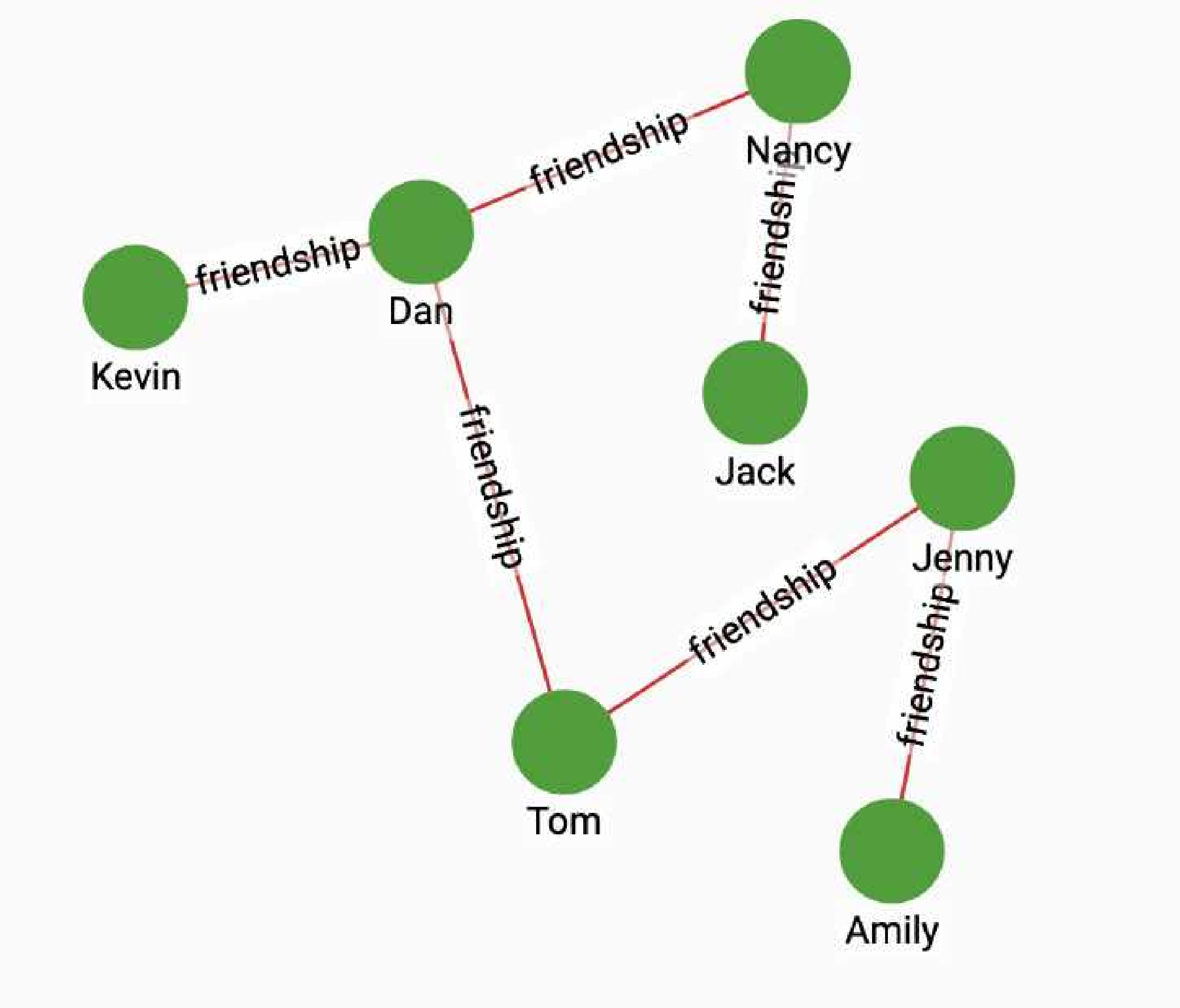
`top_k`	The number of vertices with the highest scores to return.	`INT`
`print_accum`	If true, print results to JSON output.	`BOOL`
`result_attr`	If not empty, save the degree centrality score of each vertex to this attribute.	`STRING`

Return value

The vertices with the highest degree centrality scores along with their scores.

Example

Suppose we have the following graph:



Running the query on the graph will show that Dan has the highest degree centrality

Query

```
RUN QUERY tg_degree_cent(["person"], ["friendship"],["friendship"])
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"top_scores": [
    {
      "score": 8,
      "Vertex_ID": "Dan"
    },
    {
      "score": 6,
      "Vertex_ID": "Jenny"
    },
    {
      "score": 4,
      "Vertex_ID": "Nancy"
    },
    {
      "score": 2,
      "Vertex_ID": "Kevin"
    },
    {
      "score": 2,
      "Vertex_ID": "Amily"
    },
    {
      "score": 2,
      "Vertex_ID": "Jack"
    }
  ]}]
}
```

Closeness Centrality

We all have an intuitive understanding when we say a home, an office, or a store is "centrally located." Closeness Centrality provides a precise measure of how "centrally located" a vertex is. The steps below show the steps for one vertex `v`:

Step	Mathematical Formula
1. Compute the average distance from vertex v to every other vertex:	$d_{avg}(v) = \sum_{u \neq v} dist(v, u)/(n-1)$
2. Invert the average distance, so we have average closeness of v:	$CC(v) = 1/d_{avg}(v)$

TigerGraph's closeness centrality algorithm uses multi-source breadth-first search (MS-BFS) to traverse the graph and calculate the sum of a vertex's distance to every other vertex in the graph, which vastly improves the performance of the algorithm. The algorithm's implementation of MS-BFS is based on the paper The More the Merrier: Efficient Multi-source Graph Traversal by Then et al ↗.

This algorithm query employs a subquery called `cc_subquery`. Both queries are needed to run the algorithm.

Specifications

```
tg_closeness_cent (SET<STRING> v_type, SET<STRING> e_type, INT max_hops=1
  INT top_k=100, BOOL wf = TRUE, BOOL print_accum = True, STRING result_a
  STRING file_path = "", BOOL display_edges = FALSE)
```

Parameters

Characteristic	Value

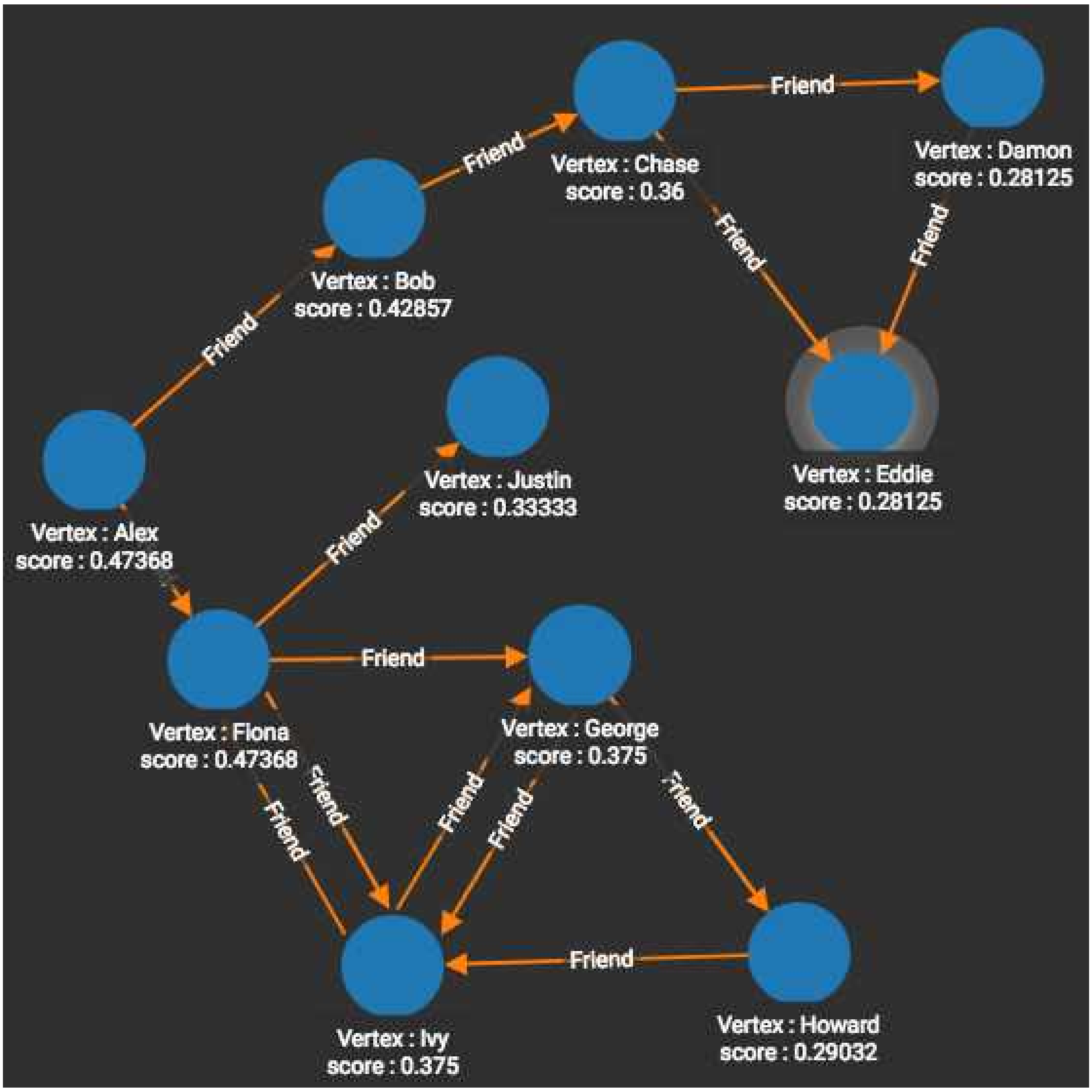
Result	Computes a Closeness Centrality value (FLOAT type) for each vertex.
Required Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `SET<STRING>` `re_type`: Names of reverse edge types to use • `INT max_hops`: If >=0, look only this far from each vertex • `INT top_k`: Sort the scores highest first and output only this many scores • `BOOL wf`: If True, use Wasserman-Faust normalization for multi-component graphs • `BOOL print_accum`: If true, output JSON to standard output • `STRING result_attr`: If not empty, store centrality values (FLOAT) to this attribute • `STRING file_path`: If not empty, write output to this file in CSV. • `BOOL display_edges`: If true, include the graph's edges in the JSON output, so that the full graph can be displayed.
Result Size	V = number of vertices
Time Complexity	O(E), E = number of edges. Parallel processing reduces the time needed for computation.

Example

Closeness centrality can be measured for either directed edges (from `v` to others) or for undirected edges. Directed graphs may seem less intuitive, however, because if the distance from Alex to Bob is 1, it does not mean the distance from Bob to Alex is also 1.

For our example, we wanted to use the topology of the Likes graph, but to have undirected edges. We emulated an undirected graph by using both `Friend` and `Also_Friend` (reverse-direction) edges.

```
# Use _ for default values
RUN QUERY tg_closeness_cent(["Person"], ["Friend", "Also_Friend"], _, _,
_, _, _, _, _)
```



Visualized results of example query on social10 graph, with Friend and Also_Friend edges

Approximate Closeness Centrality

In the Closeness Centrality algorithm, to obtain the closeness centrality score for a vertex, we measure the distance from the source vertex to every single vertex in the graph. In large graphs, running this calculation for every vertex can be highly time-consuming.

The Approximate Closeness Centrality algorithm (based on Cohen et al. 2014 ↗) calculates the approximate closeness centrality score for each vertex by combining two estimation approaches - sampling and pivoting. This hybrid estimation approach offers near-linear time processing and linear space overhead within a small relative error. It runs on graphs with unweighted edges (directed or undirected).

> ⓘ This query uses another subquery `closeness_cent_approx_sub` ↗, which needs to be installed before `closeness_approx` can be installed.

Specifications

```
tg_closeness_approx (
    SET<STRING> v_type,
    SET<STRING> e_type,
        INT k = 100,  # sample num
        INT max_hops = 10,  # max BFS explore steps
        DOUBLE epsilon = 0.1,  # error parameter
    BOOL print_accum = true, # output to console
        STRING file_path = "",  # output file
        INT debug = 0,  # debug flag -- 0: No LOG;1: LOG without the samp
        INT sample_index = 0,  # random sample group
        INT maxsize = 1000,  # max size of connected components using exa
        BOOL wf = True # Wasserman and Faust formula
)
```

Parameters

Name	Description
`v_type`	Vertex types to calculate approximate closeness centrality for.
`e_type`	Edge types to traverse.
`k`	Size of the sample.
`max_hops`	Upper limit of how many jumps the algorithm will perform from each vertex.
`epsilon`	The maximum relative error, between 0 and 1. Setting a lower value produces a more accurate estimate but increases run time.
`print_accum`	Boolean value that indicates whether or not to output to console in JSON format.
`file_path`	If provided, the algorithm will output to this file path in CSV format
`debug`	There are many conditional logging statements inside the query. If the input is 0, nothing will be logged. If the input is 1, everything else but the breadth-first-search process from the sample-node. If the input is 2, everything will be logged.
`sample_index`	The algorithm will partition the graph based on the sample size. This index indicates which partition to use when estimating closeness centrality.
`maxsize`	If the number of vertices in the graph is lower than `maxsize`, the exact closeness centrality is calculated instead and nothing will be approximated.
`wf`	Boolean value that indicates whether to use the Wasserman and Faust ↗formula to calculate closeness centrality rather than the classic formula.

Result

The result is a list of all vertices in the graph with their approximate closeness centrality score. It is available both in JSON and CSV format.

Example

Below is an example of running the algorithm on the social10 test graph and an excerpt of the response.

```
RUN QUERY tg_closeness_aprox(["Person"], ["Friend", "Coworker"], 6, 3   \
0.1, true, "", 0, 0, 100, false)

[
  {
    "Start": [
      {
        "attributes": {
          "Start.@closeness": 0.58333
        },
        "v_id": "Fiona",
        "v_type": "Person"
      },
      {
        "attributes": {
          "Start.@closeness": 0.44444
        },
        "v_id": "Justin",
        "v_type": "Person"
      },
      {
        "attributes": {
          "Start.@closeness": 0.53333
        },
        "v_id": "Bob",
        "v_type": "Person"
      }
]
```

Harmonic Centrality

The Harmonic Centrality algorithm calculates the harmonic centrality of each vertex in the graph. Harmonic Centrality is a variant of Closeness Centrality. In a (not necessarily connected) graph, the **harmonic centrality** reverses the sum and reciprocal operations in the definition of closeness centrality:

$$H(x) = \sum_{y \neq x} \frac{1}{d(x,y)}$$

If your graph has many unconnected clusters, the harmonic centrality could be a better indicator of centrality than closeness centrality.

For more information, see Harmonic Centrality ↗.

Specifications

```
CREATE QUERY harmonic_cent(SET<STRING> v_type, SET<STRING> e_type,
SET<STRING> re_type,INT max_hops=10, INT top_k=100, BOOL wf = TRUE,
BOOL print_accum = True, STRING result_attr = "", STRING file_path = "",
BOOL display_edges = FALSE)
```

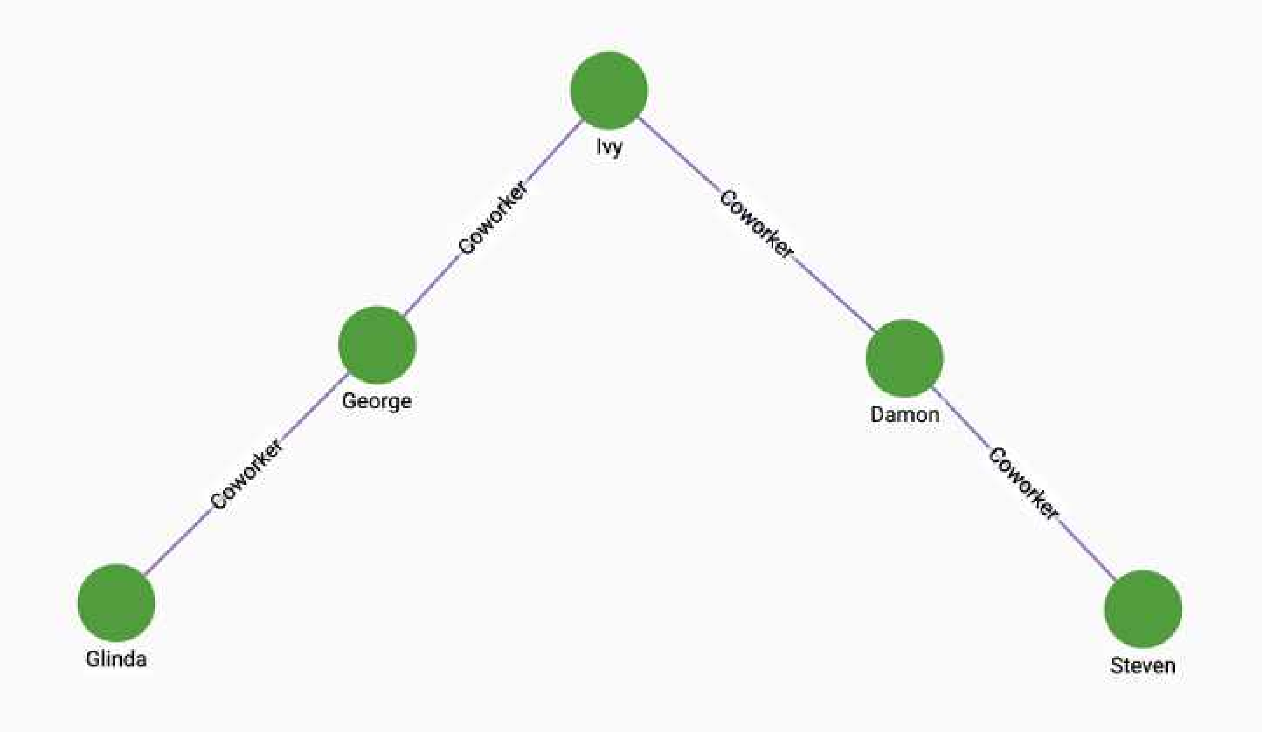
Parameters

Name	Description	Data type
`v_type`	Vertex types to calculate harmonic centrality for	`SET<STRING>`
`e_type`	Edge types to traverse	`SET<STRING>`
`re_type`	Reverse edge types. For undirected edges, fill in the edge type name in this	`SET<STRING>`

	parameter as well as the `e_type` parameter.	
`max_hops`	The maximum number of hops the algorithm would consider for each vertex. If set to a non-positive value, the limit is ignored.	`INT`
`top_k`	Sort the scores high to low and output the highest `k` scores	`INT`
`wf`	If `true`, use the Wasserman-Faust normalization for multi-component graphs	`BOOL`
`print_accum`	If `true`, output JSON to standard output	`BOOL`
`result_attr`	If not empty, store centrality values (FLOAT) to this attribute	`STRING`
`file_path`	If not empty, write output to this file in CSV.	`STRING`
`display_edges`	If `true`, include the graph's edges in the JSON output, so that the full graph can be displayed.	`BOOL`

Example

If we have the following graph, we can see that Ivy is the most central of the five vertices. Running the algorithm on the graph shows that Ivy has the highest centrality score:



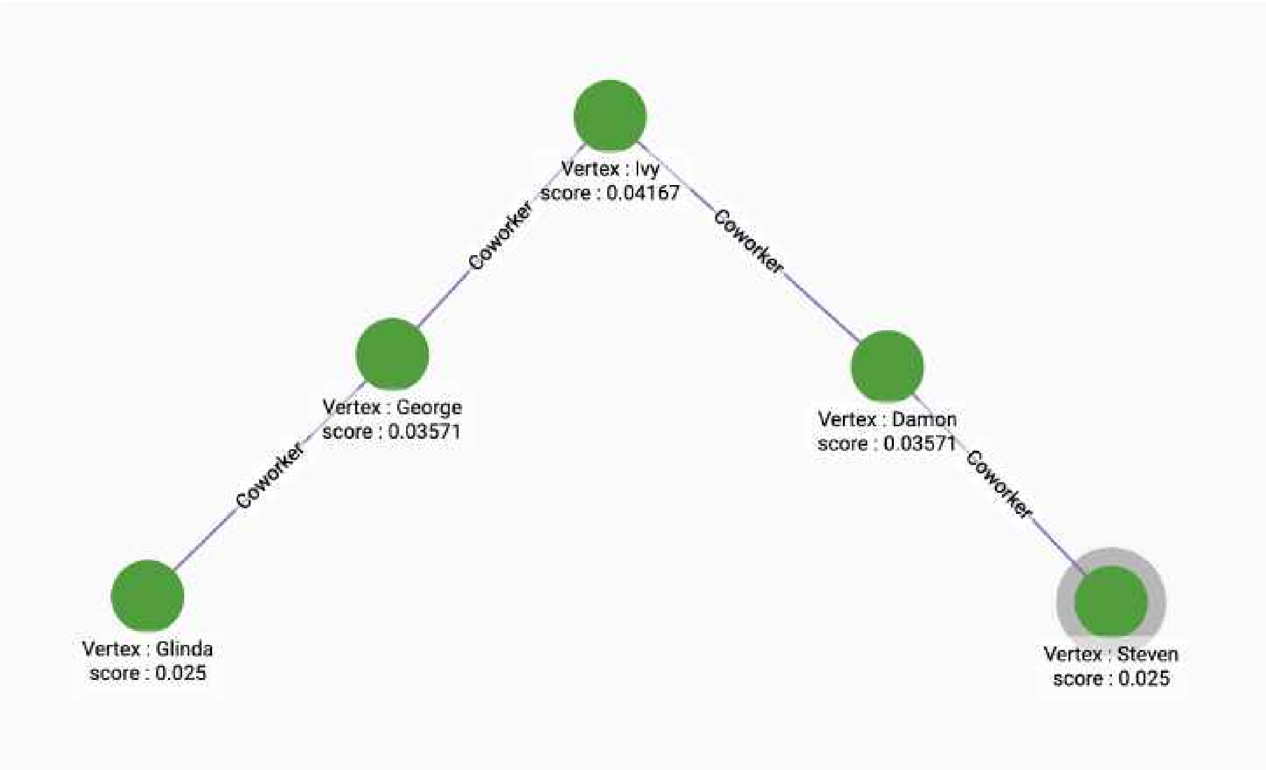
GSQL Command

```
RUN QUERY harmonic_cent(["Person"], ["Coworker"], ["Coworker"], 4, 5,
true, true, _, _, _)
```

Results

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"top_scores": [
    {
      "score": 0.04167,
      "Vertex_ID": "Ivy"
    },
    {
      "score": 0.03571,
      "Vertex_ID": "Damon"
    },
    {
      "score": 0.03571,
      "Vertex_ID": "George"
    },
    {
      "score": 0.025,
      "Vertex_ID": "Steven"
    },
    {
      "score": 0.025,
      "Vertex_ID": "Glinda"
    }
  ]}]
}
```


Visualized results


Vertex : Ivy
score : 0.04167
Coworker
Coworker
Vertex : George
score : 0.03571
Vertex : Damon
score : 0.03571
Coworker
Coworker
Vertex : Glinda
score : 0.025
Vertex : Steven
score : 0.025

Influence Maximization (Beta)

Influence maximization is the problem of finding a small subset of vertices in a social network that could maximize the spread of influence.

There are two versions of the Influence Maximization algorithm. Both versions find `k` vertices that maximize the expected spread of influence in the network. The CELF version improves upon the efficiency of the greedy version and should be preferred in analyzing large networks.

The two versions of the algorithm are implemented on the following papers:

- CELF: [Cost-effective Outbreak Detection in Networks ↗]
- Greedy: [Maximizing the Spread of Influence through a Social Network ↗]

Specifications

CELF

```
CREATE QUERY tg_influence_maximization_CELF(STRING v_type,STRING e_ty
  STRING weight, INT top_k, BOOL print_accum = True, STRING file_path
```

Greedy

```
CREATE QUERY tg_influence_maximization_greedy(STRING v_type,
  STRING e_type,STRING weight,INT top_k,
  BOOL print_accum = True, STRING file_path = "")
```

Parameters

The CELF version and the greedy version of the algorithms share the same set of parameters.

Name	Description	Data type
`v_type`	A vertex type	`STRING`
`e_type`	An edge type	`STRING`
`weight`	The name of the weight attribute on the edge type	`STRING`
`top_k`	The number of vertices with the highest influence score to return	`INT`
`print_accum`	If true, print results to JSON output.	`BOOL`
`file_path`	If not empty, save results in CSV to this file.	`STRING`

Return value

The ID of the vertices with the highest influence scores along with their scores.

Classification Algorithms

Greedy Graph Coloring

This algorithm assigns a unique integer value known as its color to the vertices of a graph such that no neighboring vertices share the same color. The reason why this is called color is that this task is equivalent to assigning a color to each nation on a map so that no neighboring nations share the same color.

Given a set of *k* vertices, the algorithm first colors all vertices with the same color - the first color. It then starts from all the vertices and has each vertex send its own colors to its neighbors. If there are two neighboring vertices with the same color, the algorithm will reassign colors where there is a conflict. The same process is repeated until all conflicts are resolved.

The algorithm has a worst-case time complexity of *O(V^2 + E),* where *V* is the number of vertices and *E* is the number of edges.

Specifications

```
CREATE QUERY tg_greedy_graph_coloring(SET<STRING> v_type,SET<STRING> e_ty
```

Parameters

Name	Description
`v_type`	A set of all vertex types to color.
`e_type`	A set of all edge types to traverse.
`max_colors`	The Maximum number of colors that can be used. Use a large number like 999999 unless there is a strict limit.
`print_color_count`	If set to true, the total number of colors used will be displayed

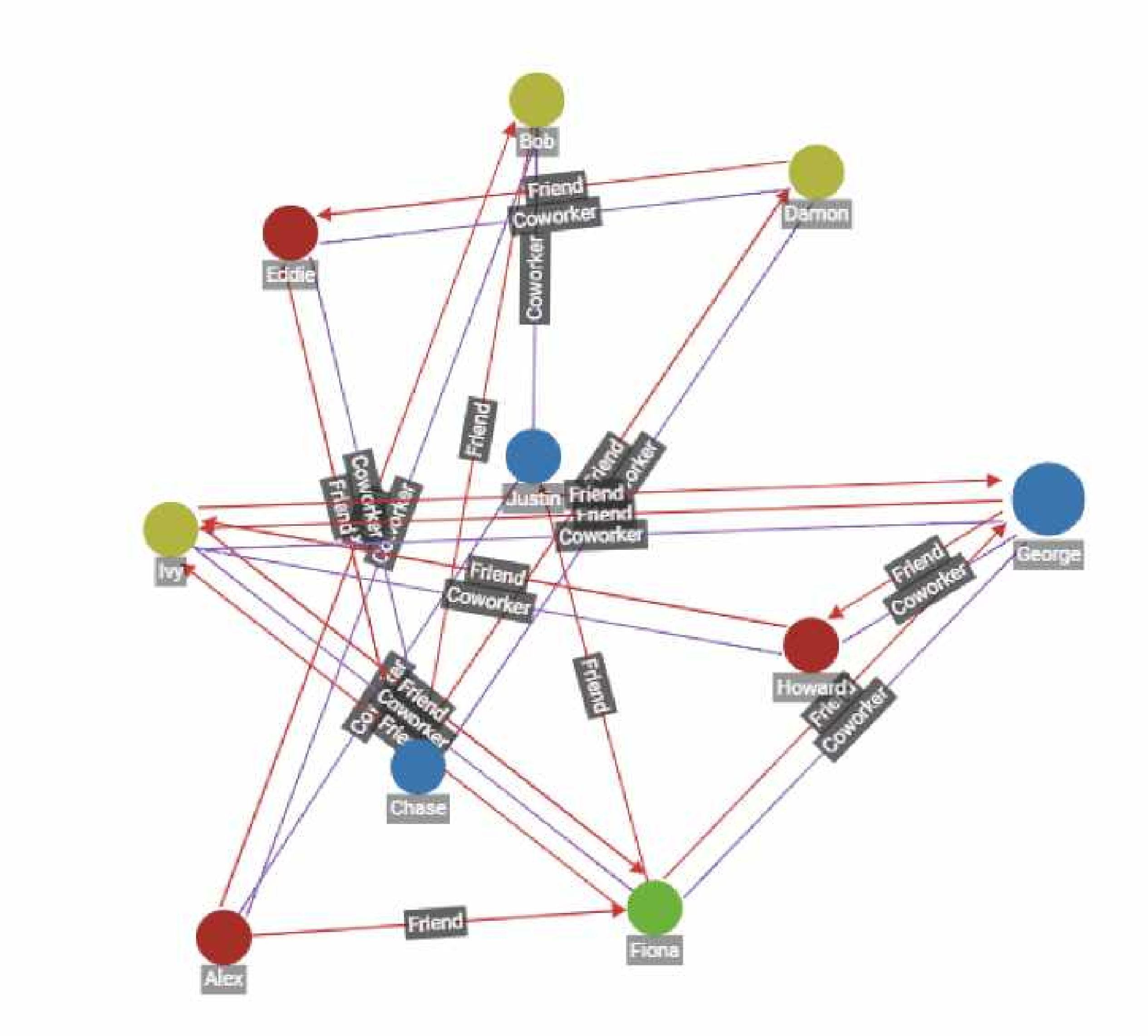
`display`	If set to true, the output will display all vertices and their associated color
	If a file path is provided, the output will be

Example

On the [social10 ↗](social10) graph, say we want to color the `Person` vertices in such a way that any two vertices that are either connected by a `Friend` edge or a `Coworker` edge do not have the same color. By running the `greedy_graph_color` algorithm, we get the following result:

```
GSQL > RUN QUERY greedy_graph_coloring(["Person"], ["Friend", "Coworker"]
 999999, true, true, "")

 [
  {
    // Total number of colors used
    "color_count": 4
  },
  {
    "start": [
      {
        "attributes": {
          "start.@colorvertex": 4
        },
        "v_id": "Fiona",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 3
        },
        "v_id": "Justin",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 2
        },
        "v_id": "Bob",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 3
        },
        "v_id": "Chase",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 2
        },
        "v_id": "Damon",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 1
```

```
        },
        "v_id": "Alex",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 3
        },
        "v_id": "George",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 1
        },
        "v_id": "Eddie",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 2
        },
        "v_id": "Ivy",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@colorvertex": 1
        },
        "v_id": "Howard",
        "v_type": "Person"
      }
    ]
  }
]
```



Visualized result - no neighboring vertices share the same color

k-Nearest Neighbors

The k-Nearest Neighbors (kNN) algorithm is one of the simplest classification algorithms. It assumes that some or all the vertices in the graph have already been classified. The classification is stored as an attribute called the label. The goal is to predict the label of a given vertex, by seeing what are the labels of the nearest vertices.

Given a source vertex in the dataset and a positive integer *k*, the algorithm calculates the distance between this vertex and all other vertices and selects the *k* vertices that are nearest. The prediction of the label of this node is the majority label among its k-nearest neighbors.

The distance can be physical distance as well as the reciprocal of similarity score, in which case "nearest" means "most similar". In our algorithm, the distance is the reciprocal of cosine neighbor similarity. The similarity calculation used here is the same as the calculation in Cosine Similarity of Neighborhoods, Single Source ↗. Note that in this algorithm, vertices with zero similarity to the source node are not considered in prediction. For example, if there are 5 vertices with non-zero similarity to the source vertex, and 5 vertices with zero similarity, when we pick the top 7 neighbors, only the label of the 5 vertices with non-zero similarity score will be used in prediction.

Specifications

```
tg_knn_cosine_ss (VERTEX source, SET<STRING> v_type, SET<STRING> e_type,
  re_type, STRING weight, STRING label, INT top_k,
  BOOL print_accum = TRUE, STRING file_path = "", STRING attr = "")
  RETURNS (STRING)
```

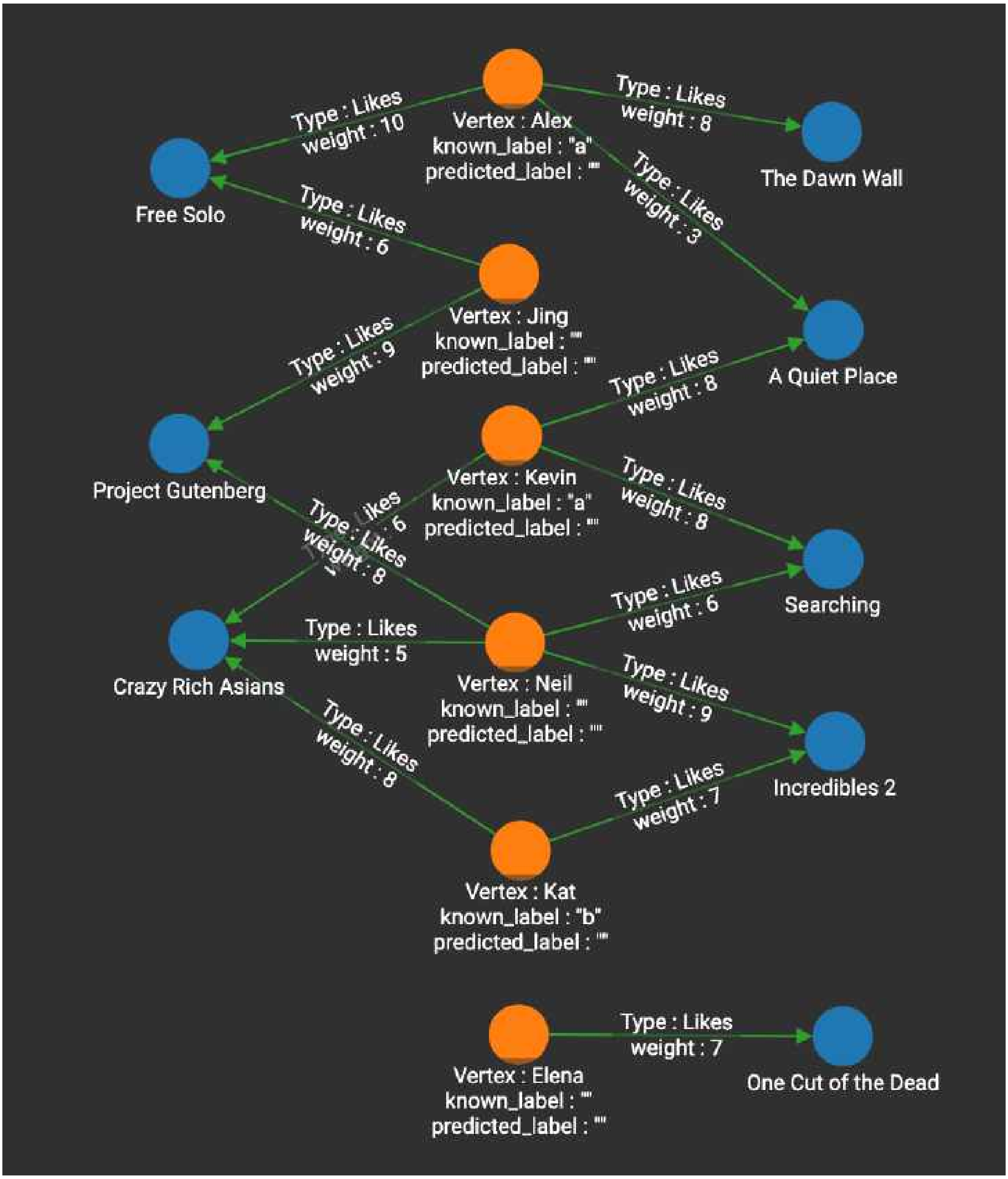
Characteristic	Value
Result	The predicted label for the source vertex. The result is available in three forms: • streamed out in JSON format

	• written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• `VERTEX source`: The vertex which you want to predict the label • `SET<STRING> v_type`: Vertex types to calculate distance to source vertex for • `SET<STRING> e_type`: Edge types to traverse • `SET<STRING> re_type`: Reverse edge types to traverse • `STRING weight`: Edge attribute to use as the weight of the edge • `STRING label`: Vertex attribute to recognize as the label of the vertex • `INT top_k`: number of nearest neighbors to consider • `BOOL print_accum`: If true, the algorithm will output the result to the console in JSON format. • `STRING filepath`: If provided, the algorithm will output to this file path in CSV format • `STRING attr`: Vertex attribute to save the predicted label as.
Result Size	V = number of vertices
Time Complexity	O(D^2), D = outdegree of vertex v
Graph Types	Undirected or directed edges, weighted edges

The algorithm will not output more than K vertex pairs, so the algorithm may arbitrarily choose to output one vertex pair over another if there are tied similarity scores.

Example

For the movie graph, we add the following labels to the Person vertices.



Movie graph with labels

When we install the algorithm, answer the questions like:

```
Vertex types: Person
Edge types: Likes
Second Hop Edge type: Reverse_Likes
Edge attribute that stores FLOAT weight, leave blank if no such attribute
Vertex attribute that stores STRING label:known_label
```

We then run kNN, using Neil as the source person and k=3. This is the JSON output :

```
[
  {
    "predicted_label": "a"
  }
]
```

If we run cosine_nbor_ss, using Neil as the source person and k=3, we can see the persons with the top 3 similarity score:

```
[
  {
    "neighbours": [
      {
        "v_id": "Kat",
        "v_type": "Person",
        "attributes": {
          "neighbours.@similarity": 0.67509
        }
      },
      {
        "v_id": "Jing",
        "v_type": "Person",
        "attributes": {
          "neighbours.@similarity": 0.46377
        }
      },
      {
        "v_id": "Kevin",
        "v_type": "Person",
        "attributes": {
          "neighbours.@similarity": 0.42436
        }
      }
    ]
  }
]
```

Kat has a label "b", Kevin has a label "a", and Jing does not have a label. Since "a" and "b" are tied, the prediction for Neil is just one of the labels.

If Jing had label "b", then there would be 2 "b"s, so "b" would be the prediction.

If Jing had label "a", then there would be 2 "a"s, so "a" would be the prediction.

k-Nearest Neighbors (Batch Version)

This algorithm is a batch version of the k-Nearest Neighbors, Cosine Neighbor Similarity, single vertex ↗. It makes a prediction for every vertex whose label is not known (i.e., the attribute for the known label is empty), based on its k nearest neighbors' labels.

Specifications

```
tg_knn_cosine_all(SET<STRING> v_type, SET<STRING> e_type, SET<STRING> re_
  STRING weight, STRING label, INT top_k, BOOL print_accum = TRUE,
  STRING file_path = "", STRING attr = "")
```

Characteristic	Value
Result	The predicted label for the vertices whose label attribute is empty. The result is available in three forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• `SET<STRING> v_type`: Vertex types to calculate distance to source vertex for • `SET<STRING> e_type`: Edge types to traverse • `SET<STRING> re_type`: Reverse edge types to traverse • `STRING weight`: Edge attribute to use as the weight of the edge • `STRING label`: Vertex attribute to recognize as the label of the vertex • `INT top_k`: number of nearest neighbors to consider • `BOOL print_accum`: Boolean value that indicates whether to output to console

	in JSON • `STRING filepath`: If provided, the algorithm will output to this file path in CSV format • `STRING attr`: Vertex attribute to save the predicted label as.
Result Size	V = number of vertices
Time Complexity	O(E^2 / V), V = number of vertices, E = number of edges
Graph Types	Undirected or directed edges, weighted edges

Example

For the movie graph shown in the single vertex version, run knn_cosine_all, using topK=3. Then you get the following result:

```
  {
    "Source": [
      {
        "v_id": "Jing",
        "v_type": "Person",
        "attributes": {
          "name": "Jing",
          "known_label": "",
          "predicted_label": "",
          "@predicted_label": "a"
        }
      },
      {
        "v_id": "Neil",
        "v_type": "Person",
        "attributes": {
          "name": "Neil",
          "known_label": "",
          "predicted_label": "",
          "@predicted_label": "b"
        }
      },
      {
        "v_id": "Elena",
        "v_type": "Person",
        "attributes": {
          "name": "Elena",
          "known_label": "",
          "predicted_label": "",
          "@predicted_label": ""
        }
      }
    ]
  }
]
```

k-Nearest Neighbors (Cross-Validation Version)

k-Nearest Neighbors (kNN) is often used for machine learning. You can choose the value for `topK` based on your experience, or using cross-validation to optimize the hyperparameters. In our library, Leave-one-out cross-validation for selecting optimal *k* is provided. Given a *k* value, we run the algorithm repeatedly using every vertex with a known label as the source vertex and predict its label. We assess the accuracy of the predictions for each value of *k*, and then repeat for different values of k in the given range. The goal is to find the value of k with highest predicting accuracy in the given range, for that dataset.

Specifications

```
tg_knn_cosine_cv(SET<STRING> v_type, SET<STRING> e_type, SET<STRING> re_t
STRING weight, STRING label, INT min_k, INT max_k) RETURNS (INT)
```

Characteristic	Value
Result	A list of prediction accuracy for every k value in the given range, and the value of k with the highest predicting accuracy in the given range. The result is available in JSON format
Input Parameters	• `SET<STRING> v_type`: Vertex types to calculate distance to source vertex for • `SET<STRING> e_type`: Edge types to traverse • `SET<STRING> re_type`: Reverse edge types to traverse • `STRING weight`: Edge attribute to use as the weight of the edge • `STRING label`: Vertex attribute to recognize as the label of the vertex

	• `INT min_k`: lower bound of k (inclusive) • `INT max_k`: upper bound of k (inclusive)
Result Size	max_k-min_k+1
Time Complexity	O(max_k*E^2 / V), V = number of vertices, E = number of edges
Graph Types	Undirected or directed edges, weighted edges

Example

Run knn_cosine_cv with min_k=2, max_k = 5. The JOSN result:

```
[
  {
    "@@correct_rate_list": [
      0.33333,
      0.33333,
      0.33333,
      0.33333
    ]
  },
  {
    "best_k": 2
  }
]
```

Community Algorithms

Weakly Connected Components

A component is the maximal set of vertices, plus their connecting edges, which are interconnected. That is, you can reach each vertex from each other vertex. In the example figure below, there are three components.

This particular algorithm deals with undirected edges. If the same definition (each vertex can reach each other vertex) is applied to directed edges, then the components are called Strongly Connected Components. If you have directed edges but ignore the direction (permitting traversal in either direction), then the algorithm finds Weakly Connected Components.

Specifications

```
tg_conn_comp (SET<STRING> v_type, SET<STRING> e_type, INT output_limit =
  BOOL print_accum = TRUE, STRING result_attr = "", STRING file_path = ""
```

Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value.
Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `INT output_limit`: If >=0, max number of vertices to output to JSON. • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store community ID values (INT) to this attribute • `STRING file_path`**:** If not empty, write output to this file.

Result Size	V = number of vertices
Time Complexity	O(E*d), E = number of edges, d = max(diameter of components)

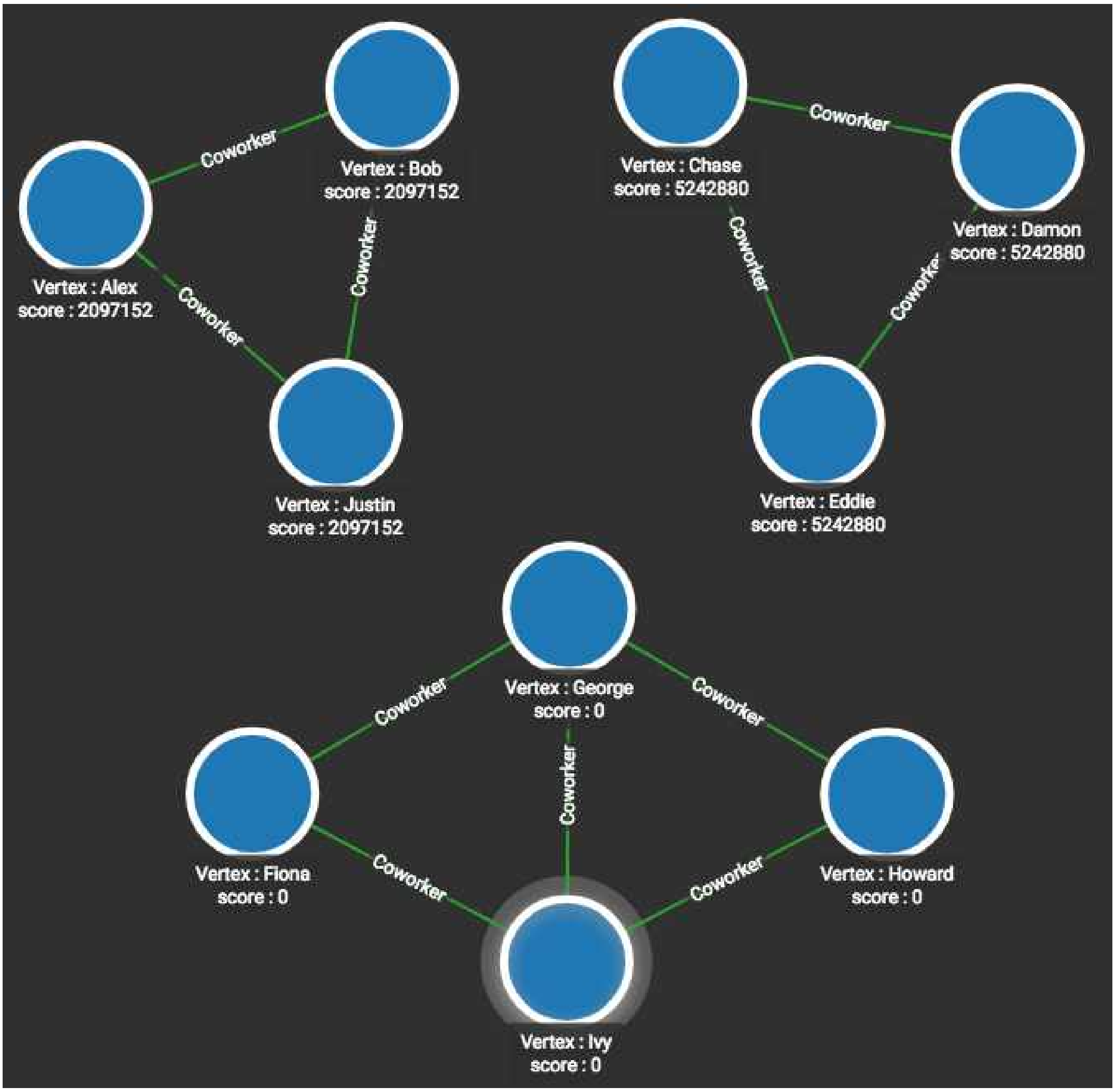
Example

It is easy to see in this small graph that the algorithm correctly groups the vertices:

- Alex, Bob, and Justin all have Community ID = 2097152
- Chase, Damon, and Eddie all have Community ID = 5242880
- Fiona, George, Howard, and Ivy all have Community ID = 0

Our algorithm uses the TigerGraph engine's internal vertex ID numbers; they cannot be predicted.

```
RUN QUERY tg_conn_comp(["Person"], ["Coworker"], _, _, _, _)
```



Visualized results of example query on social10 graph with Coworker edges

Small-World Optimized Version

In addition to the regular weakly connected component algorithm, we also provide a version that is optimized for small-world graphs. A small world graph means the graph has a hub community, where the vast majority of the vertices in the graph are weakly connected.

This version improves upon the performance of the original algorithm when dealing with small-world graphs by combining several different methods used to find connected components in a multi-step process proposed by Slota et al. in *BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems*.

The algorithm starts by selecting an initial pivot vertex *v* with a high product of indegree and outdegree. From the initial pivot vertex , the algorithm uses Breadth-First Search to determine the massive weakly connected component. The vertices that are not included in this SCC are passed off to the next step.

After identifying the first WCC, the algorithm uses the coloring method to idenify the WCCs in the remaining vertices.

For more details, see [Slota et al., *BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems.*] ↗

Specifications

```
CREATE QUERY tg_wcc_small_world(STRING v_type, STRING e_type,
UINT threshold = 100000, BOOL to_show_cc_count=FALSE)
```

Parameters

Name	Description	Data type

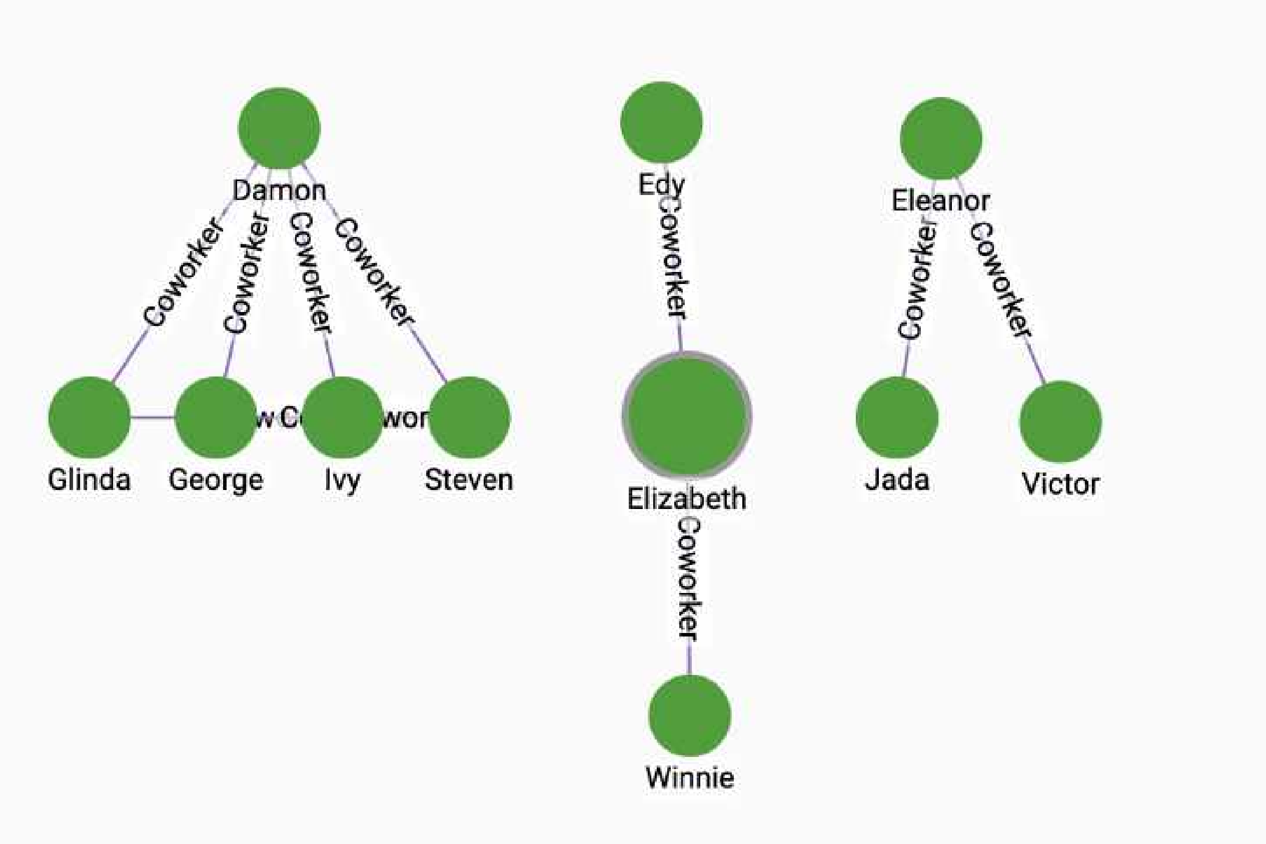
`v_type`	The vertex type to count as part of a connected component	`STRING`
`e_type`	The edge type to traverse	`STRING`
`threshold`	The threshold used to choose initial pivot vertices. Only vertices whose product of indegree and outdegree exceed this threshold will be considered candidates for the pivot vertex. This is an attempt to increase the chances that the initial pivot is contained within the largest WCC. The default value for this parameter is 100000. It is suggested that you keep this default value when running the algorithm.	`UINT`
`to_show_cc_count`	If true, the algorithm will return the number of vertices in each connected	`BOOL`

Result

If `to_show_cc_count` is set to true, the algorithm will return the number of vertices in each weakly connected component.

Example

Suppose we have the following graph. We can see that there are three connected components. The first one has 5 vertices, while the two others have 3 vertices.



Running the algorithm on the graph will show that there are three weakly connected components, and have 5, 3, and 3 vertices respectively.

GSQL Command

```
RUN QUERY tg_wcc_small_world("Person", "Coworker", _, true)
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@CC_count": {
    "1048576": 5,
    "1048577": 3,
    "4194306": 3
  }}]
}
```

K-Core Decomposition

A **k-core** of a graph is a maximal connected subgraph in which every vertex is connected to at least k vertices in the subgraph. To obtain the k-core of a graph, the algorithm first deletes the vertices whose outdegree is less than *k*. It then updates the outdegree of the neighbors of the deleted vertices, and if that causes a vertex's outdegree to fall below *k*, it will also delete that vertex. The algorithm repeats this operation until every vertex left in the subgraph has an outdegree of at least *k*.

Our algorithm takes a range of values for *k* and returns the set of the vertices that constitute the k-core with the highest possible value of *k* within the range. It is an implementation of Algorithm 2 in [Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure, Tripathy et al., IEEE Big Data 2018](#) ↗.

Time complexity

O(E), where E is the number of edges in the graph.

Specifications

```
tg_kcore(STRING v_type, STRING e_type, INT k_min = 0, INT k_max = -1,
BOOL print_accum = TRUE, STRING result_attr = "", STRING file_path = "",
BOOL print_all_k = FALSE, BOOL show_shells=FALSE)
```

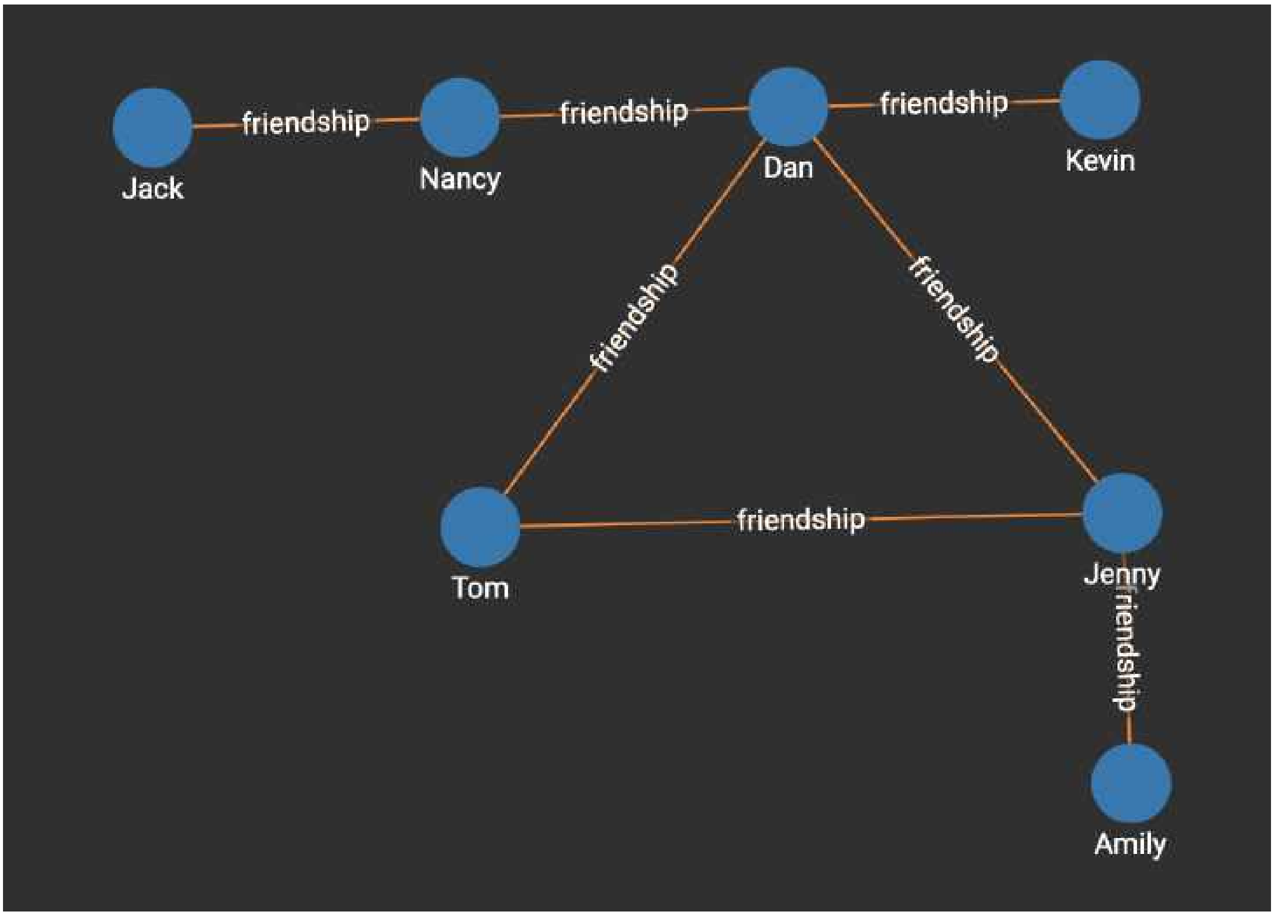
Parameters

Parameter	Description
`v_type`	Vertex type to include in the k-core
`e_type`	Edge type to count for k-core connections

`k_min`	Minimum value of k. If the actual maximum core is below `k_min`, the algorithm will return an empty set.
`k_max`	Maximum value of k. If `k_max` is smaller than `k_min`, the algorithm will ignore this parameter and keep looking for k-cores until it reaches a value of k where a k-core cannot be found.
`show_membership`	If `show_membership` is `true`, the algorithm will return the k-cores found for every value of k within the range provided. For each k-core, the results will include its member vertices.
`show_shells`	The ***k*-shell** is the set of vertices that are part of the *k*-core but not part of the (*k*+1)-core. If `show_shells` is `true`, the algorithm will return the k-shells found for every value of k. within the range provided. For each k-shell, the results will include its member vertices.
`print_accum`	Boolean value that decides whether the algorithm will return output in JSON
`attr`	Optional. An attribute of the vertex to save the core level of the vertex to. If `attr` is provided, the core level of the vertex will be saved to this attribute of the vertex.
`file_path`	Optional. If `file_path` is provided, the algorithm will output results to a file

Example

In the example below based on the `social` ↗ graph from GSQL 101, we can see that Dan, Tom, and Jenny make up a 2-core, which is the max-core of the graph:



If we run the `kcore` algorithm on this small graph like so:

```
RUN QUERY tg_kcore("person", "friendship", 0, -1, TRUE, "", "", FALSE, FA
```

Here is the returned JSON response, which includes a 2-core that is comprised of Dan, Jenny, and Tom:

```
[
  {
    "core_size": 3,
    "k": 2,             // the k-core with the highest possible k is retu
    "max_core": [
      {
        "attributes": {
          "@core": 2,
          "@deg": 0,
          "age": 40,
          "gender": "male",
          "name": "Tom",
          "state": "ca"
        },
        "v_id": "Tom",
        "v_type": "person"
      },
      {
        "attributes": {
          "@core": 2,
          "@deg": 0,
          "age": 34,
          "gender": "male",
          "name": "Dan",
          "state": "ny"
        },
        "v_id": "Dan",
        "v_type": "person"
      },
      {
        "attributes": {
          "@core": 2,
          "@deg": 0,
          "age": 25,
          "gender": "female",
          "name": "Jenny",
          "state": "tx"
        },
        "v_id": "Jenny",
        "v_type": "person"
      }
    ]
  }
]
```

Strongly Connected Components

A strongly connected component (SCC) is a subgraph such that there is a path from any vertex to every other vertex. A graph can contain more than one separate SCC. An SCC algorithm finds the maximal SCCs within a graph. Our implementation is based on the Divide-and-Conquer Strong Components (DCSC) algorithm[1]. In each iteration, pick a pivot vertex `v` randomly, and find its descendant and predecessor sets, where descendant set `D_v` is the vertex reachable from `v`, and predecessor set `P_v` is the vertices which can reach `v` (stated another way, reachable from `v` through reverse edges). The intersection of these two sets is a strongly connected component `SCC_v`. The graph can be partitioned into 4 sets: `SCC_v`, descendants `D_v` excluding `SCC_v`, predecessors `P_v` excluding `SCC`, and the remainders `R_v`. It is proved that *any SCC* is a subset of one of the 4 sets [1]. Thus, we can divide the graph into different subsets and detect the SCCs independently and iteratively.

The problem of this algorithm is unbalanced load and slow convergence when there are a lot of small SCCs, which is often the case in real-world use cases [3]. We added two trimming stages to improve the performance: size-1 SCC trimming[2] and weakly connected components[3].

The implementation of this algorithm requires reverse edges for all directed edges considered in the graph.

[1] Fleischer, Lisa K., Bruce Hendrickson, and Ali Pınar. "On identifying strongly connected components in parallel." International Parallel and Distributed Processing Symposium. Springer, Berlin, Heidelberg, 2000.

[2] Mclendon Iii, William, et al. "Finding strongly connected components in distributed graphs." Journal of Parallel and Distributed Computing 65.8 (2005): 901-910.

[3] Hong, Sungpack, Nicole C. Rodia, and Kunle Olukotun. "On fast parallel detection of strongly connected components (SCC) in small-world graphs." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM, 2013.

Specifications

```
tg_scc (SET<STRING> v_type, SET<STRING> e_type, SET<STRING> rev_e_type,
  INT top_k_dist, INT output_limit, INT max_iter = 500, INT iter_wcc = 5,
  BOOL print_accum = TRUE, STRING attr= "", STRING file_path="")
```

Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value.
Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `SET<STRING> rev_e_type`: Names of reverse direction edge types to use • `INT top_k_dist`: top k result in SCC distribution • `INT output_limit`: If >=0, max number of vertices to output to JSON. • `INT max_iter`: number of maximum iteration of the algorithm • `INT iter_wcc`: find weakly connected components for the active vertices in this iteration, since the largest SCCs are already found after several iterations; usually a small number(3 to 10) • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store community ID values (INT) to this attribute • `STRING file_path`**:** If not empty, write output to this file.
Result Size	V = number of vertices

Time Complexity	O(iter*d), d = max(diameter of components)
Graph Types	Directed edges with reverse direction edges

Example

We ran `scc` on the social26 graph. A portion of the JSON result is shown below.

```
[
  {
    "i": 1
  },
  {
    "trim_set.size()": 8
  },
  {
    "trim_set.size()": 5
  },
  {
    "trim_set.size()": 2
  },
  {
    "trim_set.size()": 2
  },
  {
    "trim_set.size()": 0
  },
  {
    "@@cluster_dist_heap": [
      {
        "csize": 9,
        "num": 1
      },
      {
        "csize": 1,
        "num": 17
      }
    ]
  },
```

The first element `"i"=1` means the whole graph is processed in just one iteration. The 5 `"trim_set.size()"` elements mean there were 5 rounds of size-1 SCC trimming. The final `"@@.cluster_dist_heap" object"` reports on the size distribution of SCCs.There is one SCC with 9 vertices, and 17 SCCs with only 1 vertex in the graph.

Small-World Optimized Version

In addition to the regular strongly connected component algorithm, we also provide a version that is optimized for small-world graphs. A small world graph in this context means the graph has a hub community, where a vast majority of the vertices of the graph are weakly connected.

This version improves upon the performance of the original algorithm when dealing with small-world graphs by combining several different methods used to find connected components in a multi-step process proposed by Slota et al. in *BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems*.

The algorithm starts by trimming the graph, which removes all vertices whose indegree or outdegree is 0. Then in the second phase, the algorithm selects an initial pivot vertex *v* with a high product of indegree and outdegree. From the initial pivot vertex , the algorithm uses one iteration of the Forward-Backward method to identify all vertices reachable by *v* (descendants) and all vertices that can reach *v* (predecessors). The intersection of the descendants and the predecessors form a strongly connected component (SCC). The vertices that are not included in this SCC are passed off to the next step.

After identifying the first SCC, the algorithm uses the coloring method and Tarjan's serial algorithm to identify the SCCs in the remaining vertices.

For more details, see Slota et al., *BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems.* ↗

Specifications

```
CREATE QUERY tg_scc_small_world(STRING v_type, STRING e_type, STRING re_t
 UINT threshold = 100000, BOOL to_show_cc_count=FALSE)
```

Time complexity

The algorithm has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Parameters

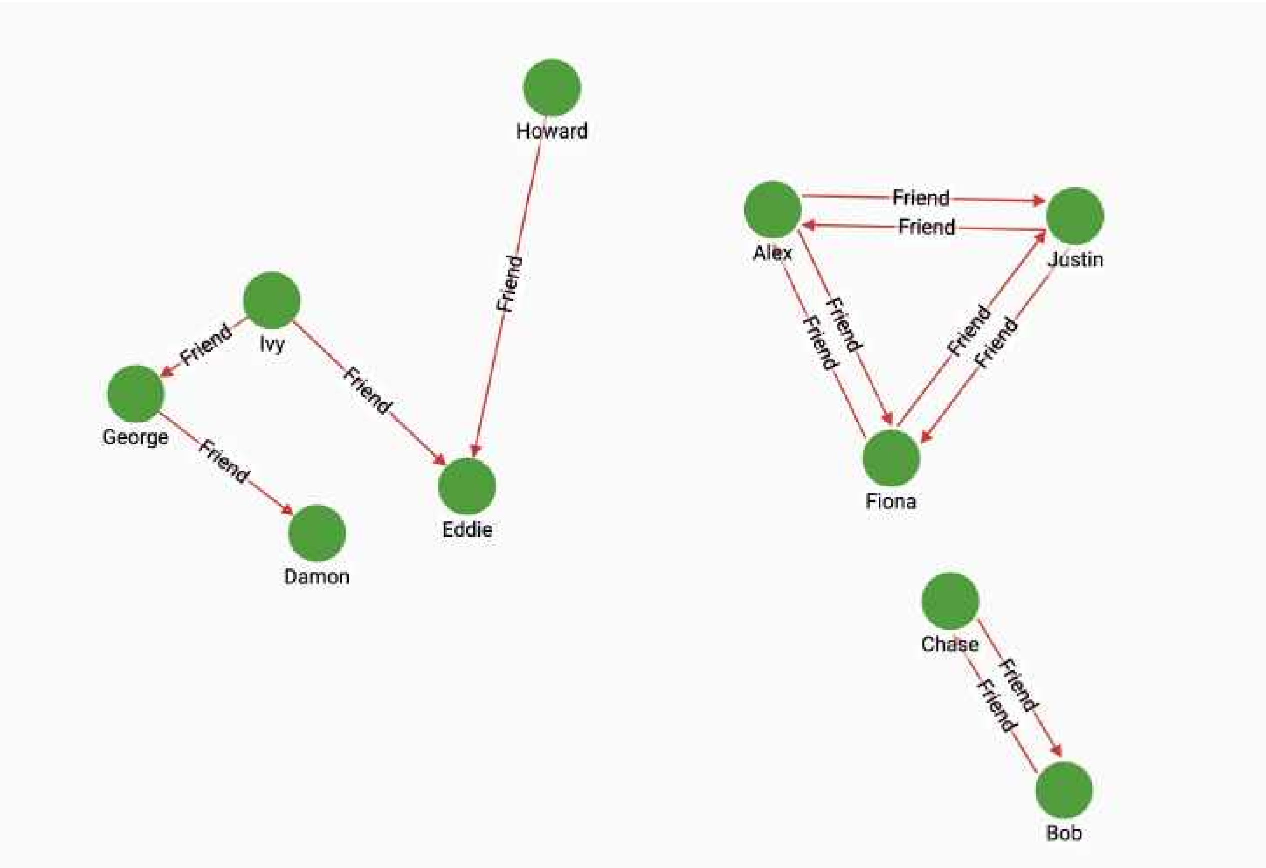
Name	Description	Data type
`v_type`	The vertex type to count as part of a strongly connected component	`STRING`
`e_type`	The edge type to traverse	`STRING`
`re_type`	The reverse edge type to traverse. If the graph is undirected, fill in the name of the undirected edge here as well as for `e_type.`	`STRING`
`threshold`	The threshold used to choose initial pivot vertices. Only vertices whose product of indegree and outdegree exceed this threshold will be considered candidates for the pivot vertex. This is an attempt to increase the chances that the initial pivot is contained within the largest SCC. The default value for this parameter is 100000. It is suggested that you keep this default value when running the algorithm.	`UINT`
`to_show_cc_count`	If set to `TRUE`, the algorithm will return the number of vertices in each strongly connected component.	`BOOL`

Result

When `to_show_cc_count` is set to true, the algorithm will return the number of strongly connected components in the graph.

Example

Suppose we have the following graph. We can see there are 7 strongly connected components. With two of them containing more than 1 vertices. The five vertices on the left are each a strongly connected component individually.



Graph with 7 SCCs

Running the algorithm on the graph will return a result of 7:

GSQL Command

```
RUN QUERY tg_scc_small_world("Person", "Friend", "Friend", _, TRUE)
```

Result


```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@CC_count.size()": 7}]
}
```

Label Propogation

Label Propagation is a heuristic method for determining communities. The idea is simple: If the plurality of your neighbors all bear the label X, then you should label yourself as also a member of X. The algorithm begins with each vertex having its own unique label. Then we iteratively update labels based on the neighbor influence described above. It is important that the order for updating the vertices be random. The algorithm is favored for its efficiency and simplicity, but it is not guaranteed to produce the same results every time.

In a variant version, some vertices could initially be known to belong to the same community. If they are well-connected to one another, they are likely to preserve their common membership and influence their neighbors,

Specifications

```
tg_label_prop (SET<STRING> v_type, SET<STRING> e_type, INT max_iter, INT
BOOL print_accum = TRUE, STRING file_path = "", STRING attr = "")
```

Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value.
Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `INT max_iter`: Number of maximum iteration of the algorithm • `INT output_limit`: If >=0, max number of vertices to output to JSON. • `BOOL print_accum`: If True, output JSON to standard output

	• **STRING attr**: If not empty, store community id values (INT) to this attribute • **STRING file_path**: If not empty, write output to this file.
Result Size	V = number of vertices
Time Complexity	O(E*k), E = number of edges, k = number of iterations.
Graph Types	Undirected edges

Example

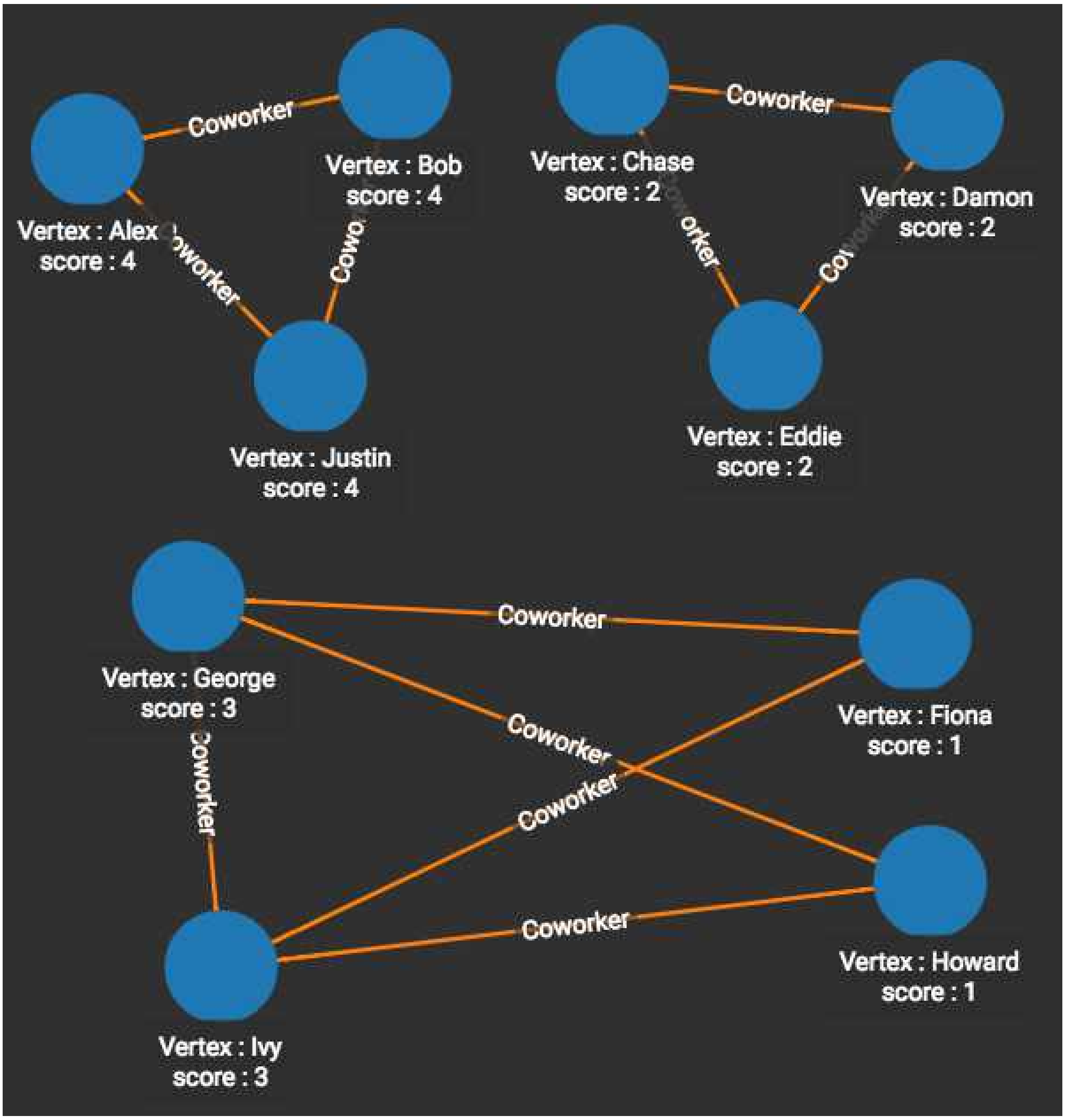
This is the same graph that was used in the Connected Component example. The results are different, though. The quartet of Fiona, George, Howard, and Ivy have been split into 2 groups:

- (George & Ivy) each connect to (Fiona & Howard) and to one another.
- (Fiona & Howard) each connect to (George & Ivy) but not to one another.

Label Propagation tries to find natural clusters and separations within connected components. That is, it looks at the quality and pattern of connections. The Component Component algorithm simply asks the Yes or No question: Are these two vertices connected?

We set `max_iter` to 10, but the algorithm reaches a steady state after 3 iterations:

```
# Use _ for default/empty values
RUN QUERY tg_label_prop(["Person"], ["Coworker"], 10, -1, _, _, _)
```



Visualized results of example query on social10 graph with Coworker edges

Local Clustering Coefficient

The Local Clustering Coefficient algorithm computes the local clustering coefficient of every vertex in a graph. The *local clustering coefficient* of a vertex (node) in a graph quantifies how close its neighbors are to being a complete graph, where every two distinct vertices are connected by an edge. It is obtained by dividing the number edges between a vertex's neighbors by the number of edges that could possibly exist.

The algorithm does not report the local clustering coefficient for vertices that have only one neighbor.

Time complexity

`O(n)`, where `n` is the number of vertices in the graph.

Specifications

```
tg_lcc(STRING v_type, STRING e_type,INT top_k=100,
  BOOL print_accum = True, STRING result_attr = "",
  STRING file_path = "", BOOL display_edges = FALSE)
```

Parameters

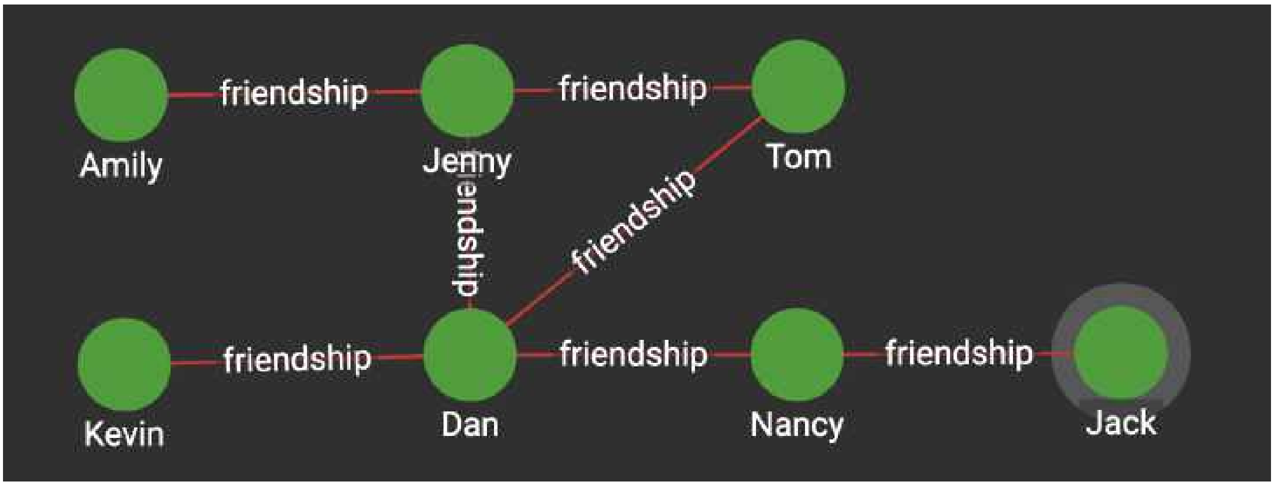
Parameter	Description	Data type
`v_type`	Vertex type to calculate local clustering coefficient for	`STRING`
`e_type`	Edge type to traverse. Only vertices that are connected by edges of this type are	`STRING`

	treated as connected in this algorithm.	
`top_k`	Number of the highest local clustering coefficients to report.	`INT`
`print_accum`	If `true`, output JSON to standard output.	`BOOL`
`result_attr`	If provided, the local clustering coefficient of a vertex will be saved to this attribute.	`STRING`
`file_path`	If provided, write output to this file in CSV.	`STRING`
`display_edges`	If `display_edges` is set to true, the algorithm will also return edges, which help produce better visualized results in GraphStudio.	`BOOL`

Example

Using the `social` graph below as an example, Jenny has three neighbors - Tom, Dan and Amily, but only Tom and Dan are connected. Between the three neighbors, the maximum number of edges that could exist is 3. Therefore, the local clustering coefficient for Jenny is 1/3.

On the other hand, Tom has two neighbors that are connected by one edge. Since the maximum number of edges that could exist between two vertices is 1, Tom has a local clustering coefficient of 1/1 = 1.



By running the algorithm in GSQL, we can confirm that Tom has the highest local clustering coefficient with a score of 1 and Jenny has a score of 1/3.

```
RUN QUERY tg_lcc("person", "friendship", _, _, _, _, _)

{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"top_scores": [
    {
      "score": 1,
      "Vertex_ID": "Tom"
    },
    {
      "score": 0.33333,
      "Vertex_ID": "Jenny"
    },
    {
      "score": 0.16667,
      "Vertex_ID": "Dan"
    },
    {
      "score": 0,
      "Vertex_ID": "Nancy"
    }
  ]}]
}
```

Louvain Method with Parallelism and Refinement

The Louvain Method for community detection [1] partitions the vertices in a graph by approximately maximizing the graph's modularity score. The modularity score for a partitioned graph assesses the difference in density of links within a partition vs. the density of links crossing from one partition to another. The assumption is that if a partitioning is good (that is, dividing up the graph into communities or clusters), then the within-density should be high and the inter-density should be low.

The most efficient and empirically effective method for calculating modularity was published by a team of researchers at the University of Louvain. The Louvain method uses agglomeration and hierarchical optimization:

1. Optimize modularity for small local communities.
2. Treat each optimized local group as one unit, and repeat the modularity operation for groups of these condensed units.

The original Louvain Method contains two phases. The first phase incrementally calculates the modularity change of moving a vertex into every other community and moves the vertex to the community with the highest modularity change. The second phase coarsens the graph by aggregating the vertices which are assigned in the same community into one vertex. The first phase and second phase make up a pass. The Louvain Method performs the passes iteratively. In other words, the algorithm assigns an initial community label to every vertex, then performs the first phase, during which the community labels are changed until there is no modularity gain. Then it aggregates the vertices with the same labels into one vertex and calculates the aggregated edge weights between new vertices. For the coarsened graph, the algorithm conducts the first phase again to move the vertices into new communities. The algorithm continues until the modularity is not increasing, or runs to the preset iteration limits.

However, phase one is sequential, and thus slow for large graphs. An improved Parallel Louvain Method Louvain Method (PLM) calculates the best community to move to for each vertex in parallel [2]. In Parallel Louvain Method(PLM), the positive modularity gain is not guaranteed, and it may also swap two vertices to each other's community. After finishing the passes, there is an additional refinement phase, which is running the first phase again on each vertex to do some small adjustments for the resulting communities. [3].

[1] Blondel, Vincent D., et al. "Fast unfolding of communities in large networks." Journal of statistical mechanics: theory and experiment 2008.10 (2008): P10008.

[2] Staudt, Christian L., and Henning Meyerhenke. "Engineering parallel algorithms for community detection in massive networks." IEEE Transactions on Parallel and Distributed Systems 27.1 (2016): 171-184.

[3] Lu, Hao, Mahantesh Halappanavar, and Ananth Kalyanaraman. "Parallel heuristics for scalable community detection." Parallel Computing 47 (2015): 19-37.

Specifications

```
tg_louvain_parallel (SET<STRING> v_type, SET<STRING> e_type, STRING wt_at
  INT iter1=10, INT iter2=10, INT iter3=10, INT split=10, BOOL print_accu
  STRING result_attr = "", STRING file_path = "", BOOL comm_by_size = TRU
```

Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value. The JSON output lists every vertex with its community ID value. It also lists community id values, sorted by community size.
	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `STRING wt_attr`: Name of edge weight attribute (must be FLOAT) • `INT iter1`: Max number of iterations for the first phase. Default value is 10

Input Parameters	• `INT iter2`: Max number of iterations for the second phase. Default value is 10 • `INT iter3`: Max number of iterations for the refinement phase. Default value is 10 • `INT split`: Number of splits in phase 1. Increase the number to save memory, at the expense of having a longer running time. Default value is 10. • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store community id values (INT) to this attribute • `STRING file_path`**:** If not empty, write output to this file. • `BOOL comm_by_size`: If True, and if `print_accum` is True, output the membership of each community, with communities arranged by size.
Result Size	V = number of vertices
Time Complexity	O(V^2*L), V = number of vertices, L = (iter1 * iter2 + iter3) = total number of iterations
Graph Types	Undirected, weighted edges An edge weight attribute is required.

Example

If we use `louvain_parallel` for social10 graph, it will give the same result as the connected components algorithm. The social26 graph is a densely connected graph. The connected components algorithm groups all the vertices into the same community and label propagation does not consider the edge weight. On the contrary, `louvain_parallel` detects 7 communities in total, and the cluster distribution is shown below (`csize` is cluster size):

```
{
    "@@clusterDist": [
      {
        "csize": 2,
        "number": 1
      },
      {
        "csize": 3,
        "number": 2
      },
      {
        "csize": 4,
        "number": 2
      },
      {
        "csize": 5,
        "number": 2
      }
    ]
```

Speaker-listener Label Propagation (Beta)

The Speaker-listener Label Propagation Algorithm (SLPA) is a variation of the Label Propagation algorithm that is able to detect overlapping communities. The main difference between LPA and SLPA is that each node can only hold a single label in LPA while it is allowed to possess multiple labels in SLPA.

The algorithm begins with each vertex having its own unique label. It then iteratively records labels in a local accumulator based on specific speaking and listening rules. Then the post-processing of the record labels is applied. Finally, the algorithm removes the nested communities and outputs all the communities. Note that it is not guaranteed to produce the same results every time.

For more information, see [SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process ↗].

Specifications

```
CREATE QUERY tg_slpa (SET<STRING> v_type, SET<STRING> e_type, FLOAT thres
BOOL print_accum = TRUE, STRING file_path = "")
```

Characteristic	Value
Result	Assigns a list of component id (INT) to each vertex, such that members of the same component have the same id value.
Required Input Parameters	• `v_type`: vertex types to traverse • `e_type`: edge types to traverse • `threshold`: threshold to drop a label • `max_iter`: number of iterations • `print_accum`: print JSON output • `file_path`: file to write CSV output to

	• `output_limit`: max #vertices to output (-1 = all)
Result Size	V = number of vertices
Time Complexity	O(E*k), E = number of edges, k = number of iterations.
Graph Types	Undirected edges

Example

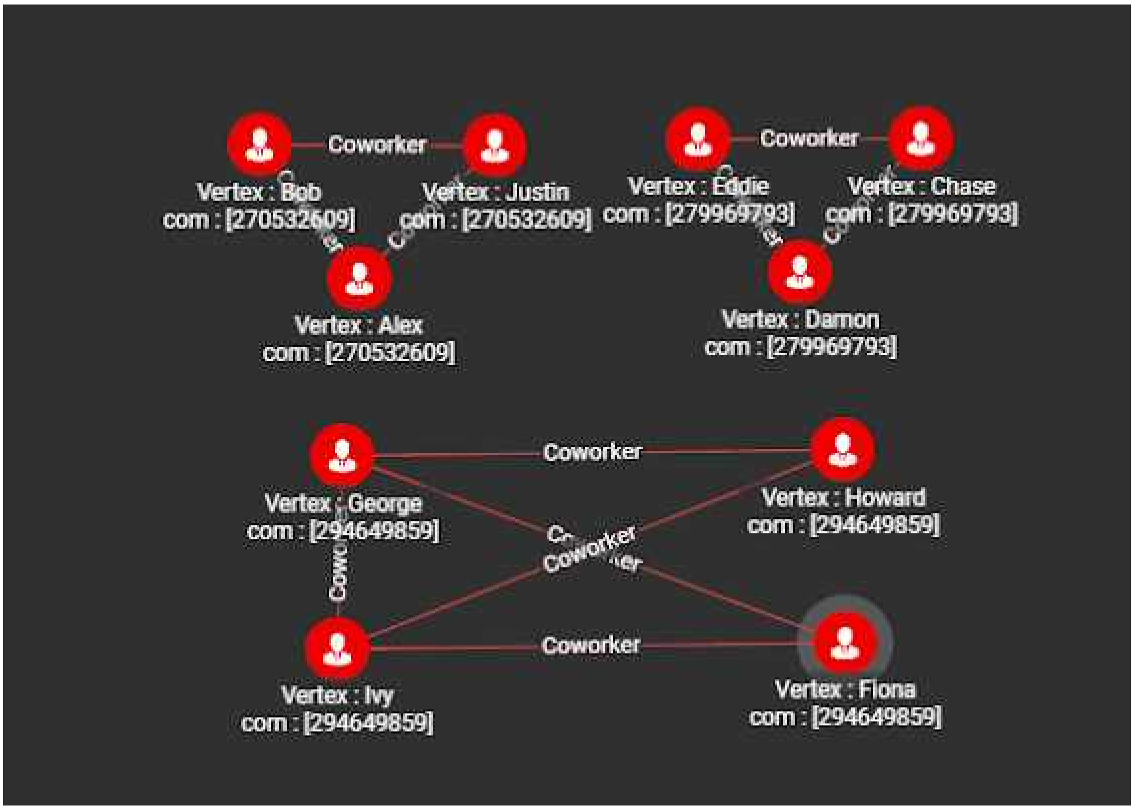
In the example below, we run the `tg_slpa` algorithm on the social10 graph. We set max_iter = 10 and threshold = 0.1.

Query

```
# Use _ for default values
RUN QUERY (["Person"], ["Coworker"], _, _, _, _, _
```

Result

```
[
  {
    "@@COM": {
      "Fiona": [294649859],
      "Alex": [270532609],
      "Damon": [279969793],
      "Justin": [270532609],
      "Eddie": [279969793],
      "Chase": [279969793],
      "Howard": [294649859],
      "George": [294649859],
      "Bob":[270532609],
      "Ivy":[294649859]
    }
  }
]
```



Visualized query result

Triangle Counting

Why triangles? Think of it in terms of a social network:

- If A knows B, and A also knows C, then we complete the triangle if B knows C. If this situation is common, it indicates a community with a lot of interaction.
- The triangle is in fact the smallest multi-edge "complete subgraph," where every vertex connects to every other vertex.

Triangle count (or density) is a measure of community and connectedness. In particular, it addresses the question of transitive relationships: If A→ B and B→C, then what is the likelihood of A→ C?

Note that it is computing a single number: How many triangles are in this graph? It is not finding communities within a graph.

It is not common to count triangles in directed graphs, though it is certainly possible. If you choose to do so, you need to be very specific about the direction of interest: In a directed graph, If A→ B and B→ C, then

- if A→C, we have a "shortcut".
- if C→A, then we have a feedback loop.

This algorithm can only be run on TigerGraph V3.1 or higher.

Specifications

We present two different algorithms for counting triangles. The first, tri_count(), is the classic edge-iterator algorithm. For each edge and its two endpoint vertices S and T, count the overlap between S's neighbors and T's neighbors.

```
tg_tri_count(STRING v_type, STRING e_type)
```

One side effect of the simple edge-iterator algorithm is that it ends up considering each of the three sides of a triangle. The count needs to be divided by 3, meaning we did 3 times more work than a smaller algorithm would have.

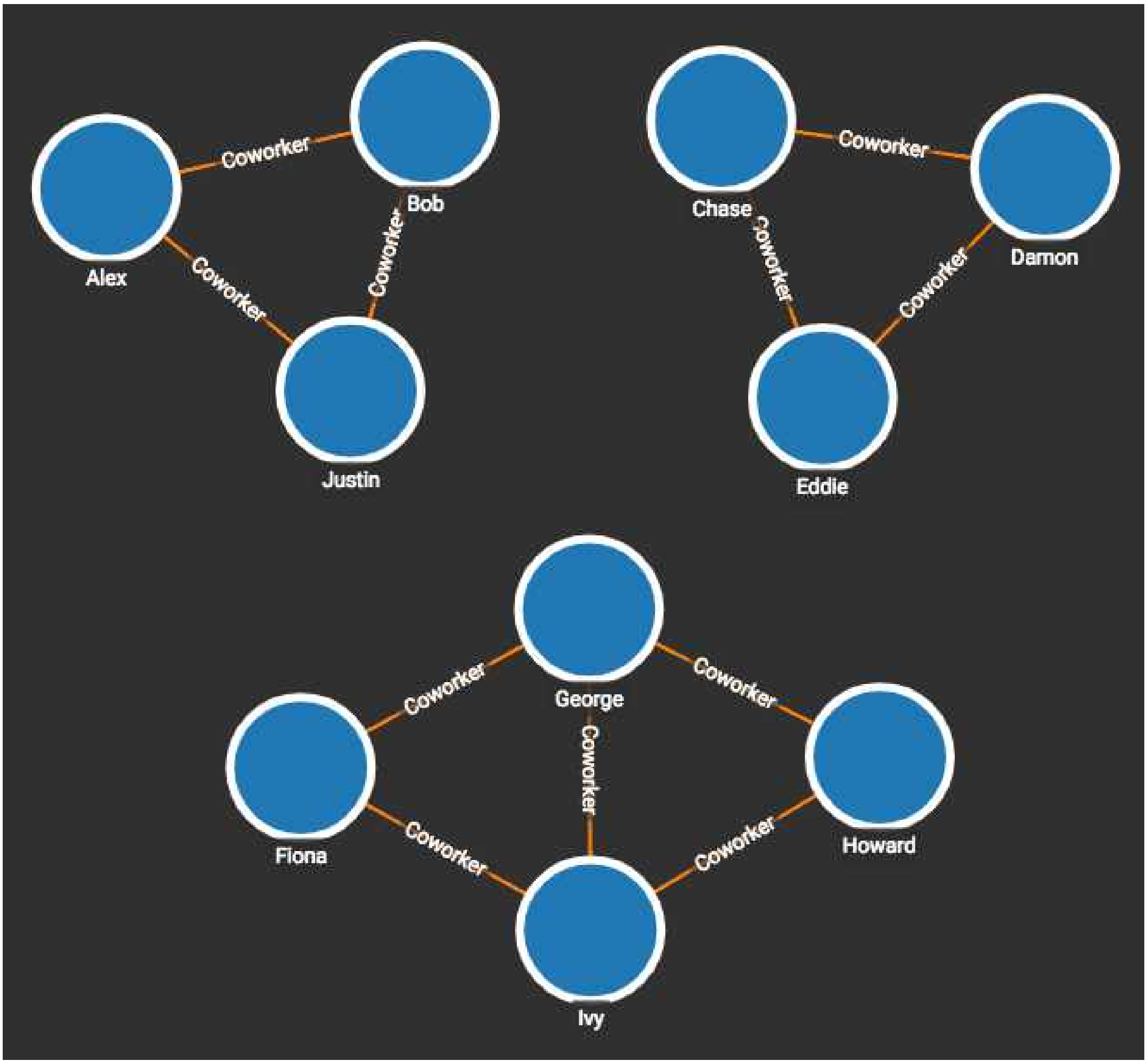
tri_count_fast() is a smarter algorithm which does two passes over the edges. In the first pass we mark which of the two endpoint vertices has fewer neighbors. In the second pass, we count the overlap only between marked vertices. The result is that we eliminate 1/3 of the neighborhood matching, the slowest 1/3, but at the cost of some additional memory.

```
tg_tri_count_fast(STRING v_type, STRING e_type)
```

Characteristic	Value
Result	Returns the number of triangles in the graph.
Input Parameters	`v_type`: Vertex type to count `e_type`: Edge type to traverse
Result Size	1 integer
Time Complexity	O(V * E), V = number of vertices, E = number of edges
Graph Types	Undirected edges

Example

In the social10 graph with Coworker edges, there are clearly 4 triangles.




```
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"num_triangles": 4}
  ]
}
```

Node Embeddings

Node2Vec

Node2Vec is a node embedding algorithm that uses random walks in the graph to create a vector representation of a node.

A random walk starts with a node, and the algorithm iteratively selects neighboring nodes to visit, and each neighboring node has an assigned probability. This transforms graph structure into a collection of linear sequences of nodes. For each node we will be left with a list of other nodes from their local or extended neighborhoods.

Once the above step is complete, the algorithm uses a variation of the word2vec model from the language modeling community to turn each node into a vector of probabilities. The probabilities represent the likelihood of visiting a given node in a random walk from each starting node.

Specification

```
tg_random_walk(INT step = 8, INT path_size = 4,
    STRING filepath = "/home/tigergraph/path.csv", SET<STRING> edge_types
    INT sample_num)

tg_node2vec_query(STRING filepath = "/home/tigergraph/path.csv",
    STRING output_file = "/home/tigergraph/embedding.csv",
    INT dimension)
```

Installing this query requires installing a UDF, which can be found in the Github repository of the query ↗. If you are running the query on a cluster, you need to manually install the UDF on every node of the cluster.

Parameters

Parameter	Description	Data type

`step`	Number of random walks per node	`INT`
`path_size`	Number of hops per walk	`INT`
`filepath`	File path to output results to	`STRING`
`edge_types`	Edge types to traverse	`SET<STRING>`
	Number of nodes to be	

Fast Random Projection (FastRP)

Fast Random Projection (FastRP) is a scalable and performant node-embedding algorithm. It generates node embeddings (vectors) of low dimensionality through random projections from the graph's adjacency matrix (a high-dimensional matrix) to a low-dimensional matrix, significantly reducing the computing power required to process the data. The algorithm is theoretically backed by the Johnsson-Lindenstrauss lemma ↗, which states that a set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved ↗.

This allows us to aggressively reduce the dimensionality of a dataset while preserving the distance information. By running the algorithm on a graph, we can preserve the similarity between nodes and their neighbors. Nodes that have similar neighborhoods are mapped to similar embedding vectors, while nodes that are not similar are mapped to dissimilar vectors.

The algorithm starts by assigning random vectors using *very sparse random projection.* The algorithm continues to average the random vectors, and then iteratively average the neighboring vectors from the previous iteration to construct intermediate embeddings. In each iteration, the algorithm also normalizes the intermediate embedding using a standard Euclidean norm.

In the end, the embedding for each node is a weighted sum of the intermediate embeddings, where the weights are provided as a parameter of the algorithm.

This algorithm is especially useful in the following situations:

- Your data is in such high dimension that it is too expensive to perform calculations.
- The dimensionality is very high and the number of samples is too sparse to calculate covariances.
- You don't have access to the entire dataset, such as when working with real-time data.

Specifications

```
CREATE QUERY tg_fastRP_query(
INT num_edges, INT num_nodes, INT k, INT sampling_constant,
INT reduced_dimension, DOUBLE normalization_strength, STRING input_weight
STRING index_attr)
```

This algorithm requires an index attribute on the schema. Before running the algorithm, create an attribute on all vertex types to hold the index. You can use our provided schema change job script in the algorithm library.

After the attribute has been created, run the tg_fastRP_preprocessing.gsql ↗ query to index all vertices on the graph.

Parameters

Parameter	Description	Data type
`num_edges`	The total number of edges in the graph.	`INT`
`num_nodes`	The total number of vertices in the graph.	`INT`
`sampling_constant`	The sampling constant. A high `sampling_constant` increases the speed of the algorithm.	`INT`
`reduced_dimension`	The length of the produced vectors. A greater dimension offers greater precision, but is more costly to operate over.	`INT`
`k`	The number of iterations for the algorithm to construct intermediate embeddings.	`INT`

`normalization_strength`	The normalization strength determines how node degrees influence the embedding. Using a negative value will downplay the importance of high degree neighbors, while a positive value will instead increase their importance.	`DOUBLE`
`input_weights`	A list of floats that determines the weight of the intermediate embeddings constructed in each iteration.	`STRING`
`index_attr`	The name of the index attribute created during	`STRING`

Parameter tuning

The optimal values for the following parameters are dependent on your dataset. In order to obtain the best quality embeddings for your graph, it is a good idea to tune these parameters.

`reduced_dimension`

The reduced dimension (`reduced_dimension`) is the length of the produced vectors. A greater dimension offers greater precision, but is more costly to operate over.

`input_weights` and `k`

The algorithm interactively constructs intermediate embeddings by averaging either neighboring intermediate embeddings from the previous iteration, or the generated random vectors during the first iteration. The final embeddings are weighted sums of the intermediate embeddings from each iteration.

`k` is the number of iterations and the `input_weights` parameter determine how much each set of intermediate embeddings from each iteration weighs. You should make sure that you supply a value of `k` that is in accordance with the number of weights you provide.

With each iteration, the algorithm will consider neighbors that are one step farther away.

`normalization_strength`

The normalization strength determines how node degrees influence the embedding. Using a negative value will downplay the importance of high degree neighbors, while a positive value will instead increase their importance.

`sampling_constant`

FastRP uses *very sparse random projection* to reduce the dimensionality of the data from an n*m matrix* to an n*d matrix where d <= m by multiplying the original matrix with an m*d matrix. *The* m*d matrix is made up of independently and identically distributed data sampled from:

$$\mathbf{R}_{ij} = \begin{cases} \sqrt{s} \text{ with probability } \frac{1}{2s} \\ 0 \text{ with probability } 1 - \frac{1}{s} \\ -\sqrt{s} \text{ with probability } \frac{1}{2s} \end{cases}$$

img

Where *s* is the sampling constant (`sampling_constant`). The higher the constant, the higher the number of zeros in the resulting matrix, making it more sparse, but also speeds up the algorithm.

Pathfinding Algorithms

A* (Beta)

A∗ (pronounced "A-star") is a graph traversal and path search algorithm, which achieves better performance by using heuristics to guide its search.

The algorithm starts from a source node, and at each iteration of its main loop, it selects the path that minimizes $f(n) = g(n) + h(n)$ where n is the next node on the path, g(n) is the cost of the path from the source node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the target node.

The algorithm terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the target, the algorithm is guaranteed to return a least-cost path from source to target.

Specifications

```
CREATE QUERY tg_astar(VERTEX source_vertex, VERTEX target_vertex, SET<STR
STRING wt_attr, BOOL display = False)
```

Characteristic	Value
Result	Computes a shortest distance (INT) and shortest path (STRING) from vertex source to target vertex.
Required Input Parameters	• **source_vertex**: start vertex • **target_vertex**: target vertex • **e_type**: edge types to traverse • **wt_type**: weight data type (INT,FLOAT,DOUBLE) • **wt_attr**: attribute for edge weights • **display**: output edges for visualization

Result Size	1
Time Complexity	O(V^2), V = number of vertices.
Graph Types	Directed or Undirected edges, Weighted edges

Example

In the example below, we run the `tg_astar` algorithm to find the shortest path between the source vertex "Kings Cross" and the target vertex "Kentish Town" on a graph which is a transport network of stations. Each station has its geometric attibutes(i.e.latitude and longitude) and the edge weight represents the distance between stations in kilometers. The heuristic function used to guide the search is the Haversine Formula ↗, which computes the distance between two points on a sphere given their longitudes and latitudes

```
# Use _ for default values
RUN QUERY astar(("Kings Cross","Station"), ("Kentish Town","Station"),["C
```

Below is the visualized result of the query:

Breadth-First Search (Beta)

Breadth-first Search (BFS) is an algorithm used to explore the vertexes of a graph layer by layer. It starts at the given vertex and explores all vertices at the present depth prior to moving on to the vertices at the next depth level.

Specifications

```
CREATE QUERY tg_bfs(SET<STRING> v_type, SET<STRING> e_type,INT max_hops=1
    VERTEX v_start, BOOL print_accum = True, STRING result_attr = "",
    STRING file_path = "", BOOL display_edges = True)
```

Characteristic	Value
Result	Returns all the nodes that are accessible from the source vertex
Required Input Parameters	• `v_type`: vertex types to traverse • `e_type`: edge types to traverse • `max_hops`: look only this far from each vertex • `v_start`: source vertex for traverse • `print_accum`: print JSON output • `result_attr`: INT attr to store results to • `file_path`: file to write CSV output to • `display_edges`:output edges for visualization
Result Size	V = number of vertices
Time Complexity	O(E+V), E = number of edges, V = number of vertices.since every vertex and every edge will be explored in the worst case.
Graph Types	Directed or Undirected edges, Weighted or Unweighted edges

Example

In the example below, we run `tg_bfs` algorithm from the source vertex `alex` on the social10 graph.

```
# Use _ for default values
RUN QUERY tg_bfs(["Person"], ["Friend"], _, ("Alex","Person"), _, _, _, _
```

Below is the visualized result of the query:

Single-source Shortest Path (Unweighted)

If a graph has unweighted edges, then finding the shortest path from one vertex to another is the same as finding the path with the fewest hops. Think of Six Degrees of Separation and Friend of a Friend. Unweighted Shortest Path answers the question "How are you two related?" The two entities do not have to be persons. Shortest Path is useful in a host of applications, from estimating influences or knowledge transfer, to criminal investigation.

When the graph is unweighted, we can use a "greedy" approach to find the shortest path. In computer science, a greedy algorithm makes intermediate choices based on the data being considered at the moment, and then does not revisit those choices later on. In this case, once the algorithm finds any path to a vertex T, it is certain that that is a shortest path.

> This algorithm finds an unweighted shortest path from one source vertex to each possible destination vertex in the graph. That is, it finds n paths.
>
> If your graph has weighted edges, see the next algorithm. With weighted edges, it is necessary to search the whole graph, whether you want the path for just one destination or for all destinations.

Specifications

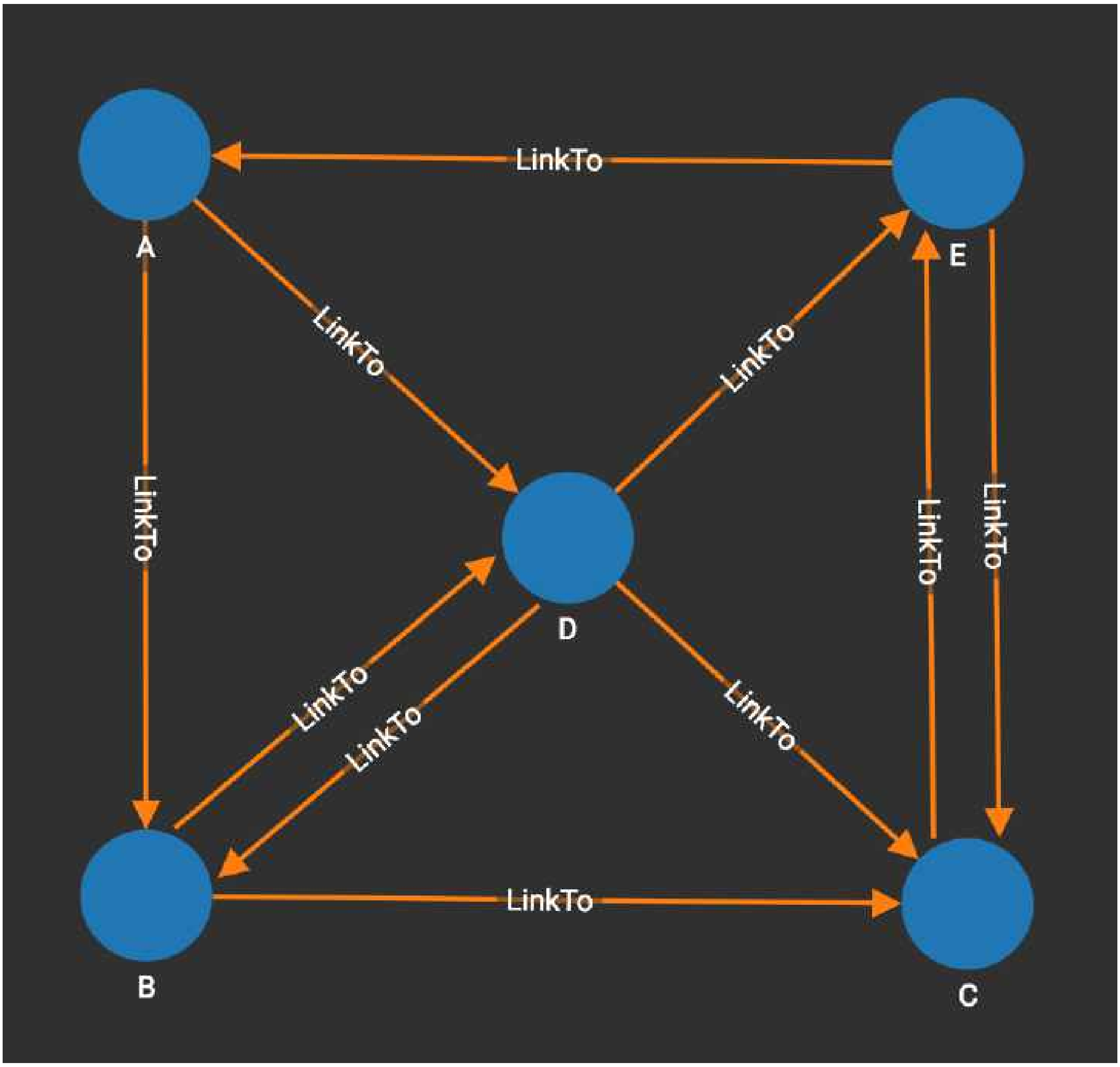
```
CREATE QUERY tg_shortest_ss_no_wt (VERTEX source, SET<STRING> v_type,
  SET<STRING> e_type, INT output_limit = -1, BOOL print_accum =TRUE,
  STRING result_attr ="", STRING file_path ="", BOOL display_edges =FALSE
```

Characteristic	Value
Result	Computes a shortest distance (INT) and shortest path (STRING) from vertex *source* to each other vertex.

Input Parameters	• `VERTEX source`: ID of the source vertex • `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `INT output_limit`: If >=0, max number of vertices to output to JSON. • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store distance values (INT) to this attribute • `STRING file_path`: If not empty, write output to this file. • `BOOL display_edges`: If true, include the graph's edges in the JSON output, so that the full graph can be displayed.
Result Size	V = number of vertices
Time Complexity	O(E), E = number of edges

Example

In the below graph, we do not consider the weight on edges. Using vertex A as the source vertex, the algorithm discovers that the shortest path from A to B is A-B, and the shortest path from A to C is A-D-C, etc.



Generic graph with unweighted edges

```
[
  {
    "ResultSet": [
      {
        "v_id": "B",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 1,
          "ResultSet.@path": [
            "A",
            "B"
          ]
        }
      },
      {
        "v_id": "A",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 0,
          "ResultSet.@path": [
            "A"
          ]
        }
      },
      {
        "v_id": "C",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 2,
          "ResultSet.@path": [
            "A",
            "D",
            "C"
          ]
        }
      },
      {
        "v_id": "E",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 2,
          "ResultSet.@path": [
            "A",
            "D",
            "E"
          ]
        }
      },
```

```
      },
      {
        "v_id": "D",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 1,
          "ResultSet.@path": [
            "A",
            "D"
          ]
        }
      }
    ]
  }
]
```

Single-source Shortest Path (Weighted)

Finding shortest paths in a graph with weighted edges is algorithmically harder than in an unweighted graph because even after you find a path to a vertex T, you cannot be certain that it is a shortest path. If edge weights are always positive, then you must keep trying until you have considered every in-edge to T. If edge weights can be negative, then it's even harder. You must consider all possible paths.

A classic application for weighted shortest path is finding the shortest travel route to get from A to B. (Think of route planning "GPS" apps.) In general, any application where you are looking for the cheapest route is a possible fit.

Specifications

The shortest path algorithm can be optimized if we know all the weights are nonnegative. If there can be negative weights, then sometimes a longer path will have a lower cumulative weight. Therefore, we have two versions of this algorithm

```
tg_shortest_ss_pos_wt (VERTEX source, SET<STRING> v_type, SET<STRING> e_t
 STRING wt_attr, STRING wt_type, INT output_limit = -1, BOOL print_accum
 STRING result_attr = "", STRING file_path = "", BOOL display_edges = FAL
```

```
tg_shortest_ss_any_wt (VERTEX source, SET<STRING> v_type, SET<STRING> e_t
 STRING wt_attr, STRING wt_type, INT output_limit = -1, BOOL print_accum
 STRING result_attr = "", STRING file_path = "", BOOL display_edges = FAL
```

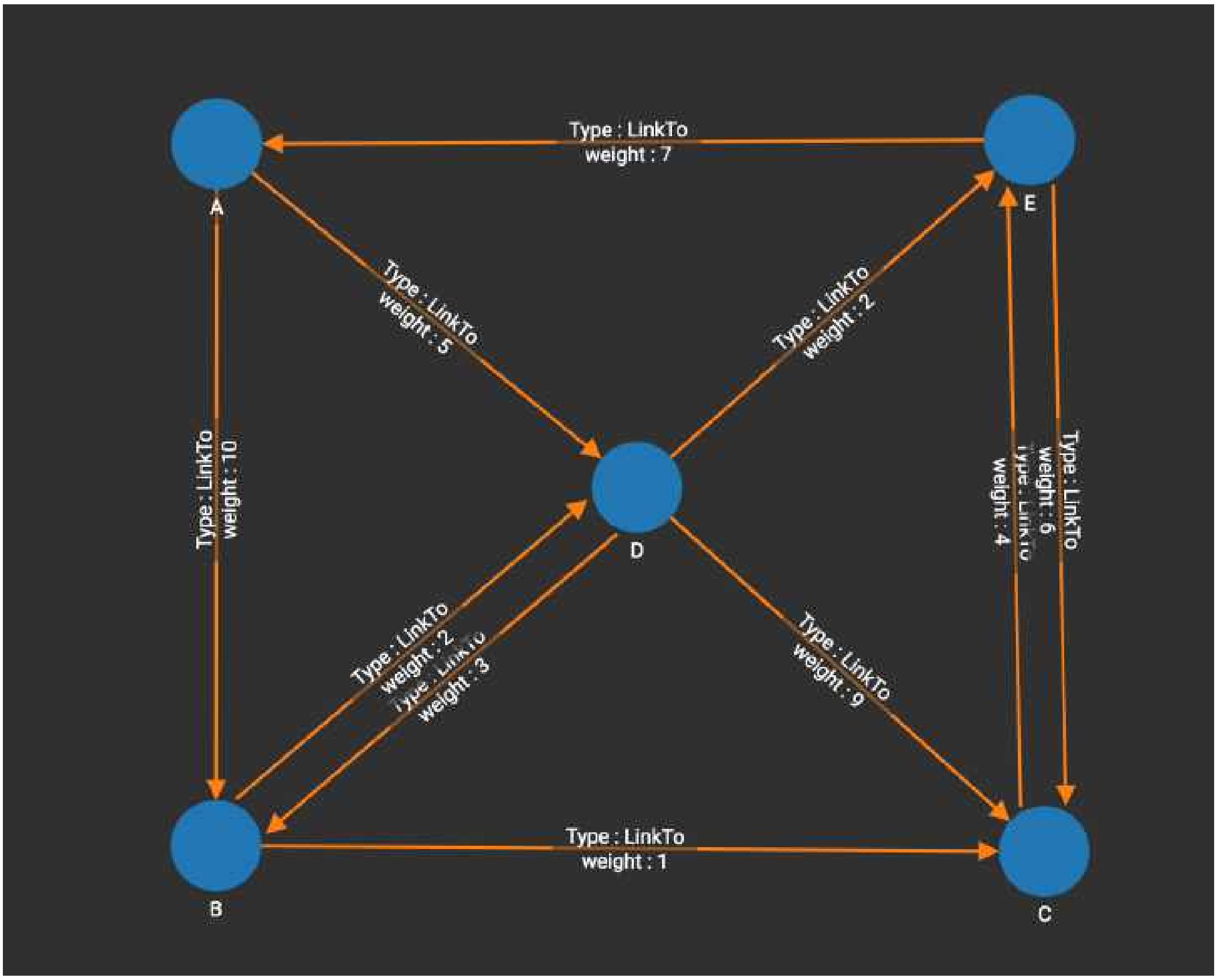
Characteristic	Value
Result	Computes a shortest distance (INT) and shortest path (STRING) from vertex *source* to each other vertex.
	• **`VERTEX source`**: Id of the source vertex

Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `STRING wt_attr`: Name of edge weight attribute • `STRING wt_type`: Data type of edge weight attribute: "INT", "FLOAT", or "DOUBLE" • `INT output_limit`: If >=0, max number of vertices to output to JSON. • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store distance values (INT) to this attribute • `STRING file_path`: If not empty, write output to this file. • `BOOL display_edges`: If true, include the graph's edges in the JSON output, so that the full graph can be displayed.
Result Size	V = number of vertices
Time Complexity	O(V*E), V = number of vertices, E = number of edges
Graph Types	Directed or Undirected edges, Weighted edges

> The shortest_path_any_wt query is an implementation of the Bellman-Ford algorithm. If there is more than one path with the same total weight, the algorithm returns one of them.
>
> Currently, shortest_path_pos_wt also uses Bellman-Ford. The well-known Dijsktra's algorithm is designed for serial computation and cannot work with GSQL's parallel processing.

Example

The graph below has only positive edge weights. Using vertex A as the source vertex, the algorithm discovers that the shortest weighted path from A to B is A-D-B, with distance 8. The shortest weighted path from A to C is A-D-B-C with distance 9.

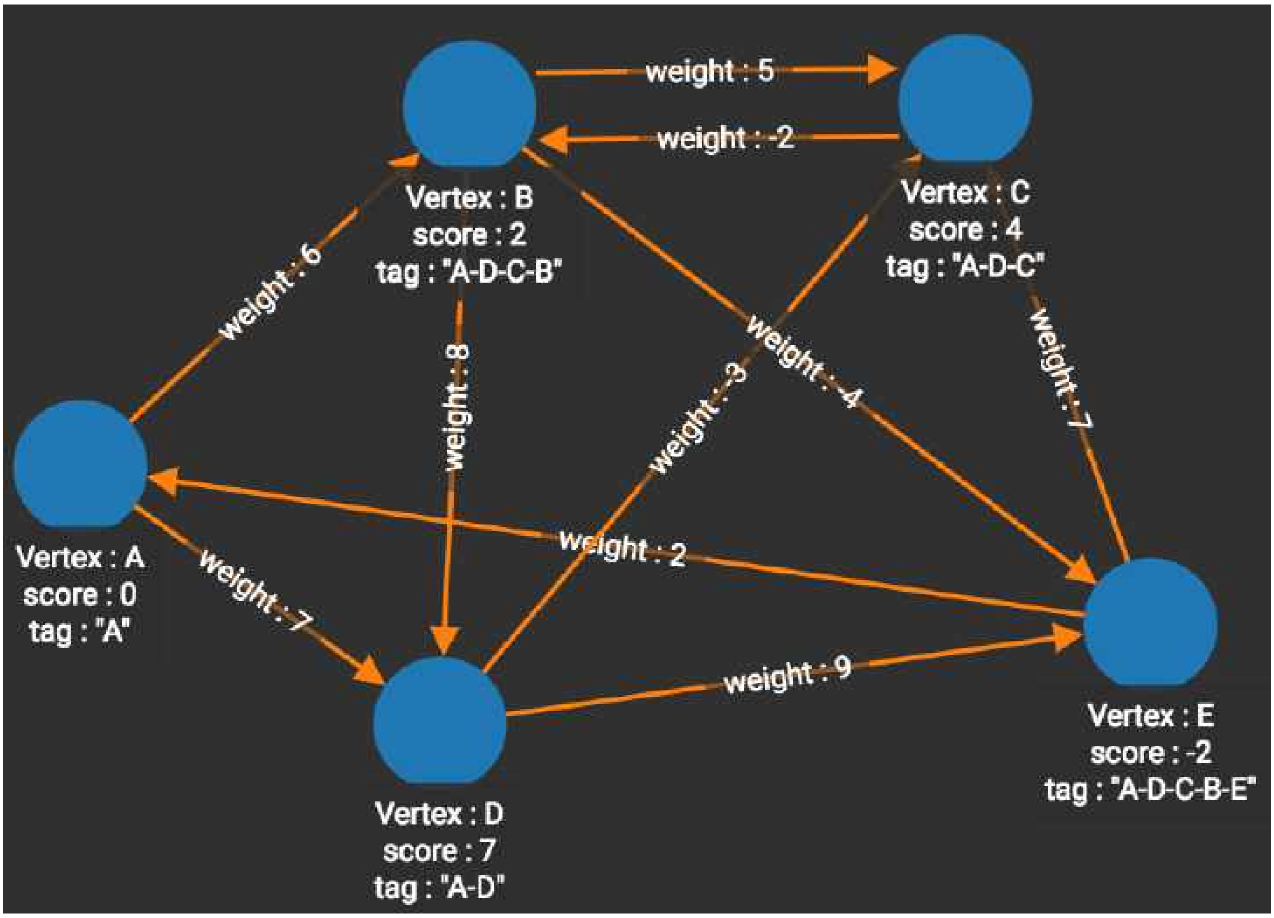


Generic graph with only positive weights

```
[
  {
    "ResultSet": [
      {
        "v_id": "B",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 8,
          "ResultSet.@path": [
            "D",
            "B"
          ]
        }
      },
      {
        "v_id": "A",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 0,
          "ResultSet.@path": []
        }
      },
      {
        "v_id": "C",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 9,
          "ResultSet.@path": [
            "D",
            "B",
            "C"
          ]
        }
      },
      {
        "v_id": "E",
        "v_type": "Node",
        "attributes": {
          "ResultSet.@dis": 7,
          "ResultSet.@path": [
            "D",
            "E"
          ]
        }
      },
      {
        "v_id": "D",
        "v_type": "Node",
```

```
          "attributes": {
            "ResultSet.@dis": 5,
            "ResultSet.@path": [
              "D"
            ]
          }
        }
      ]
    }
  ]
```

The graph below has both positive and negative edge weights. Using vertex A as the source vertex, the algorithm discovers that the shortest weighted path from A to E is A-D-C-B-E, with a cumulative score of 7 - 3 - 2 - 4 = -2.



Example results on a graph with negative weights on edges

Single-Pair Shortest Path

The Single-Pair Shortest Path task seeks the shortest path between a source vertex S and a target vertex T. If the edges are unweighted, then use the query in our tutorial document GSQL Demo Examples.

If the edges are weighted, then use the Single-Source Shortest Path algorithm. In the worst case, it takes the same computational effort to find the shortest path for one pair as to find the shortest paths for all pairs from the same source S. The reason is that you cannot know whether you have found the shortest (least weight) path until you have explored the full graph. If the weights are always positive, however, then a more efficient algorithm is possible. You can stop searching when you have found paths that use each of the in-edges to T.

All-Pairs Shortest Path

The All-Pairs Shortest Path algorithm is costly for large graphs because the computation time is O(V^3) and the output size is O(V^2). Be cautious about running this on very large graphs.

The All-Pairs Shortest Path (APSP) task seeks to find the shortest paths between every pair of vertices in the entire graph. In principle, this task can be handled by running the Single-Source Shortest Path (SSSP) algorithm for each input vertex, e.g.,

```
CREATE QUERY tg_all_pairs_shortest(SET<STRING> v_type, SET<STRING> e_type
 STRING wt_attr, STRING wt_type, STRING result_attr = "", STRING file_pat
{
  Start = {v_type};
  Result = SELECT s FROM Start:s
         POST-ACCUM
           shortest_ss_any_wt(s, v_type, e_type, wt_attr, wt_type,
           result_attr, file_path+s);
}
```

This example highlights one of the strengths of GSQL: treating queries as stored procedures that can be called from within other queries. We only show the result_attr and file_path options, because subqueries cannot send their JSON output.

For large graphs (with millions of vertices or more), however, this is an enormous task. While the massively parallel processing of the TigerGraph platform can speed up the computation by 10x or 100x, consider what it takes just to store or report the results. If there are 1 million vertices, then there are nearly 1 trillion output values.

There are more efficient methods than calling the single-source shortest path algorithm n times, such as the Floyd-Warshall algorithm, which computes APSP in O(V^3) time.

Our recommendation:

- If you have a smaller graph (perhaps thousands or tens of thousands of vertices), the APSP task may be tractable.
- If you have a large graph, avoid using APSP.

Random Walk

Random Walk is an algorithm that generate random paths in a graph. A random walk starts at every vertex that has an outgoing edge, and moves to another vertex at random. The random walk algorithm will print all possible paths of the specified size from the walks that were performed onto a file in the CSV format.

Specifications

```
CREATE QUERY tg_random_walk(
INT step = 8, INT path_size = 4, STRING filepath = "/home/tigergraph/path
SET<STRING> edge_types, INT sample_num)
```

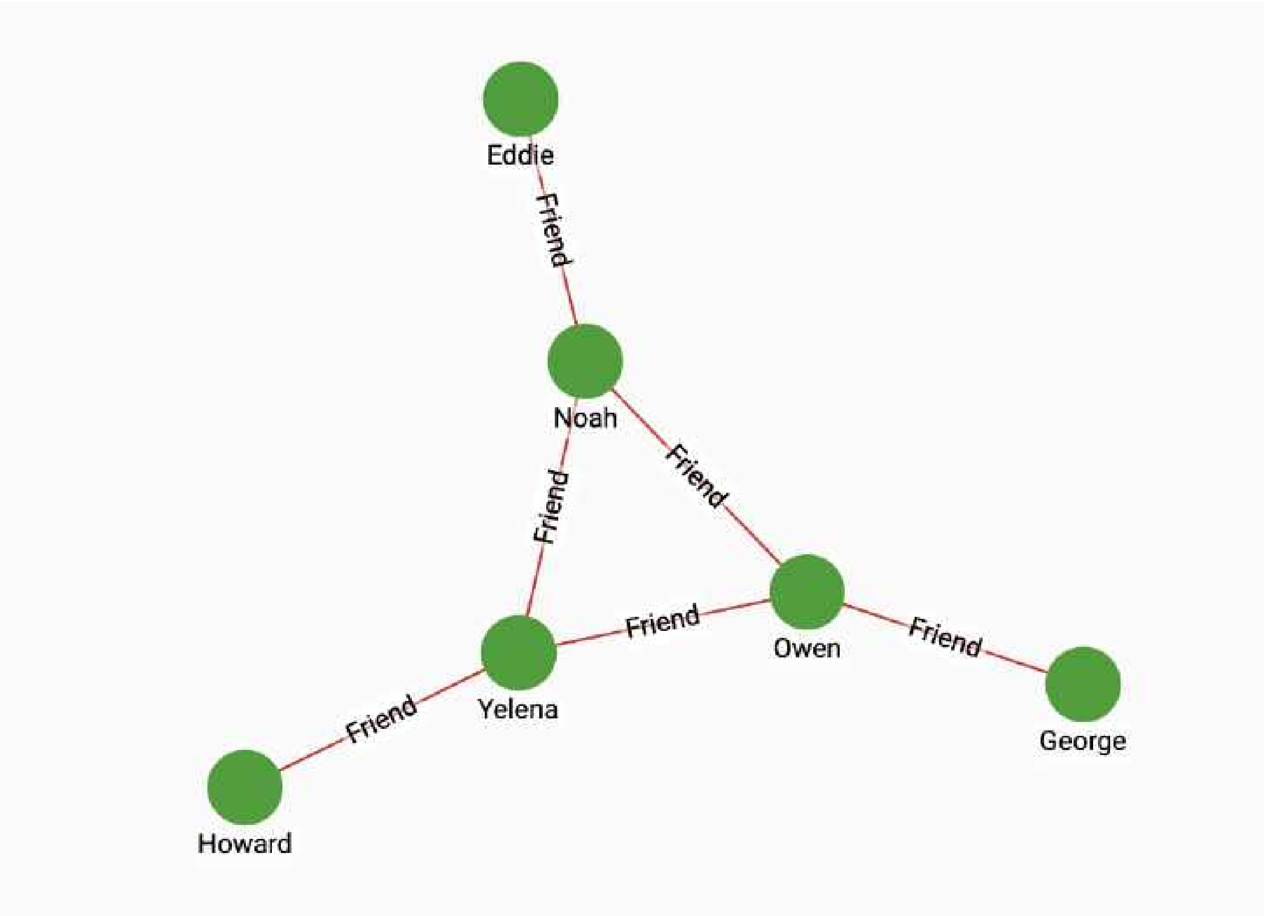
Parameters

Name	Description	Data type
`step`	The number of hops performed in each random walk.	`INT`
`path_size`	The length of the paths to output. The length refers to the number of vertices in a path. For example, if a walk has two steps: A → B → C, then there are paths of both lengths 2 and lengths 3 that can be output from this walk. If a path size of 2 is supplied, then the algorithm outputs two paths: A → B and B → C. If the path size is 3, then there is one path: A → B → C.	`INT`

`filepath`	The filepath to output the paths to.	`STRING`
`edge_types`	The edge types that the random walk will traverse.	`SET<STRING>`
`sample_num`	At each possible step, the number of sample walks to perform	`INT`

Example

Suppose we have the following social graph:



Example social graph

If we run the random walk algorithm on the graph and supply a path size of 3 and a step of 2, and a sample number of 1. Then from each vertex there is a two-step random walk, and a total of 6 three-vertex paths:

path.csv

```
Howard Yelena Noah
Noah Owen Noah
George Owen Noah
Eddie Noah Yelena
Yelena Howard Yelena
Owen George Owen
```

We can also perform a 3 step walk and still ask for three-vertex paths, in which case the number of paths would double, because in each 3 step walk, there are two possible three-vertex paths:

path.csv

```
Howard Yelena Noah
Owen Yelena Noah
Noah Owen Noah
George Owen Noah
Yelena Owen Noah
Eddie Noah Owen
Howard Yelena Noah
Noah Owen Noah
George Owen Noah
Eddie Noah Howard
Yelena Noah Howard
Owen Noah Howard
```

Minimum Spanning Tree

Given an undirected and connected graph, a minimum spanning tree is a set of edges that can connect all the vertices in the graph with the minimal sum of edge weights. The library implements a parallel version of Prim's algorithm ↗:

1. Start with a set A = { a single vertex *seed* }
2. For all vertices in A, select a vertex y such that
   a. y is not in A, and
   b. There is an edge from y to a vertex x in A, and
   c. The weight of the edge e(x,y) is the smallest among all eligible pairs (x,y).
3. Add y to A, and add the edge (x,y) to MST.
4. Repeat steps 2 and 3 until A has all vertices in the graph.

If the user specifies a source vertex, this will be used as the *seed*. Otherwise, the algorithm will select a random *seed* vertex.

If the graph contains multiple components (i.e., some vertices are disconnected from the rest of the graph, then the algorithm will span only the component of the seed vertex.

If you do not have a preferred vertex, and the graph might have more than one component, then you should use the Minimum Spanning Forest (MDF) algorithm instead.

Specifications

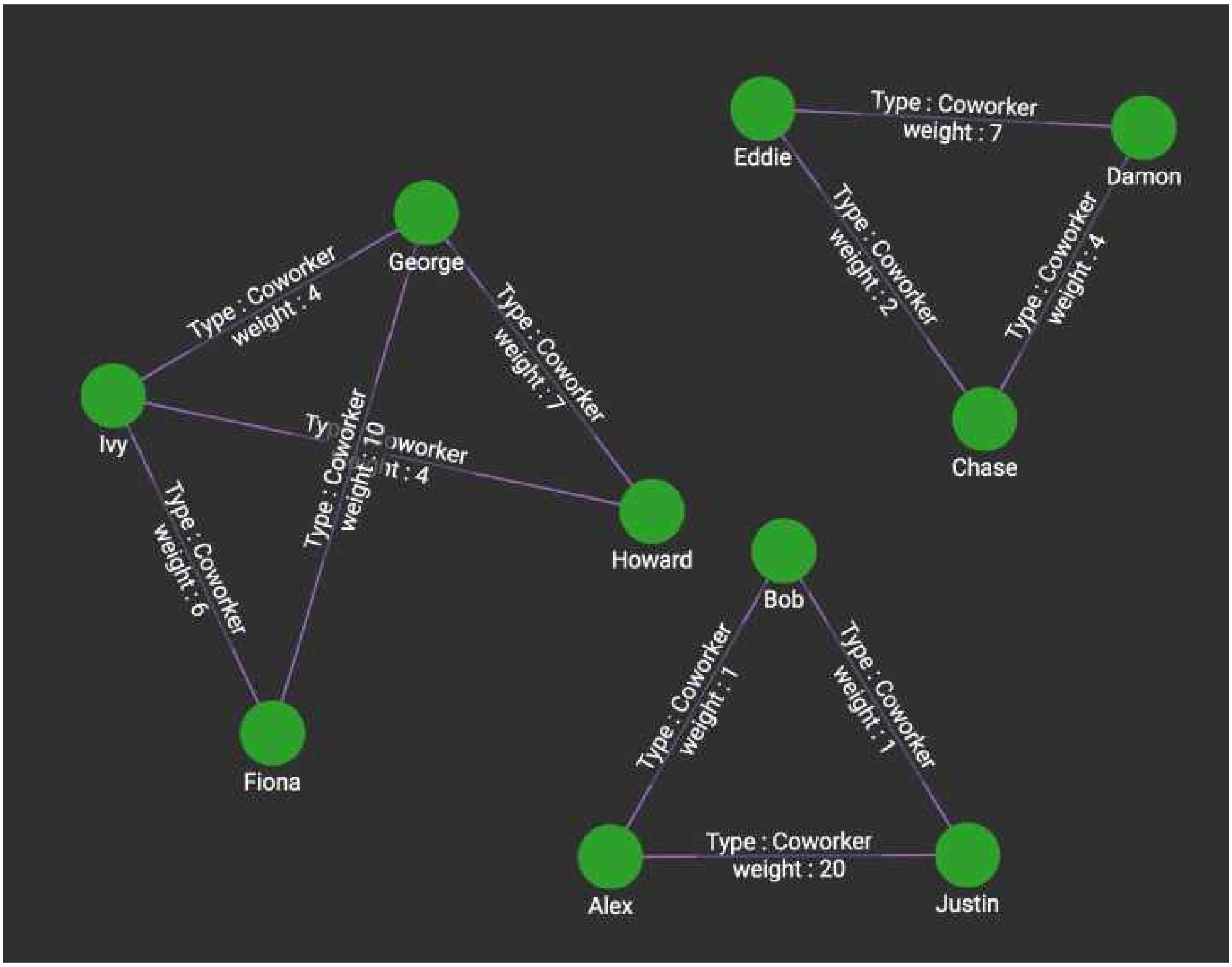
```
tg_mst (VERTEX opt_source, SET<STRING> v_type, SET<STRING> e_type,
  STRIN wt_attr, STRING wt_type, INT max_iter = -1,
  BOOL print_accum = TRUE, STRING result_attr = "", STRING file_path = ""
```

Characteristic	Value
	Computes a minimum spanning tree. If the JSON or file output selected, the output is

Result	the set of edges that form the MST. If the result_attr option is selected, the edges which are part of the MST are tagged True; other edges are tagged False.
Input Parameters	• `VERTEX opt_source` : ID of a source vertex (optional) • `SET<STRING> v_type` : Names of vertex types to use • `SET<STRING> e_type` : Names of edge types to use • `STRING wt_attr` : Name of edge weight attribute • `STRING wt_type` : Data type of edge weight attribute: "INT", "FLOAT", or "DOUBLE" • `INT max_iter` : Maximum of edges to include. If less than (V-1), then the result is not a true spanning tree. • `BOOL print_accum` : If True, output JSON to standard output • `STRING result_attr` : If not empty, store result values (BOOL) to this attribute • `STRING file_path` : If not empty, write output to this file.
Result Size	V - 1 = number of vertices - 1
Time Complexity	O(V^2)
Graph Types	Undirected edges and connected

Example

In the graph `social10`, we consider only the undirected Coworker edges.



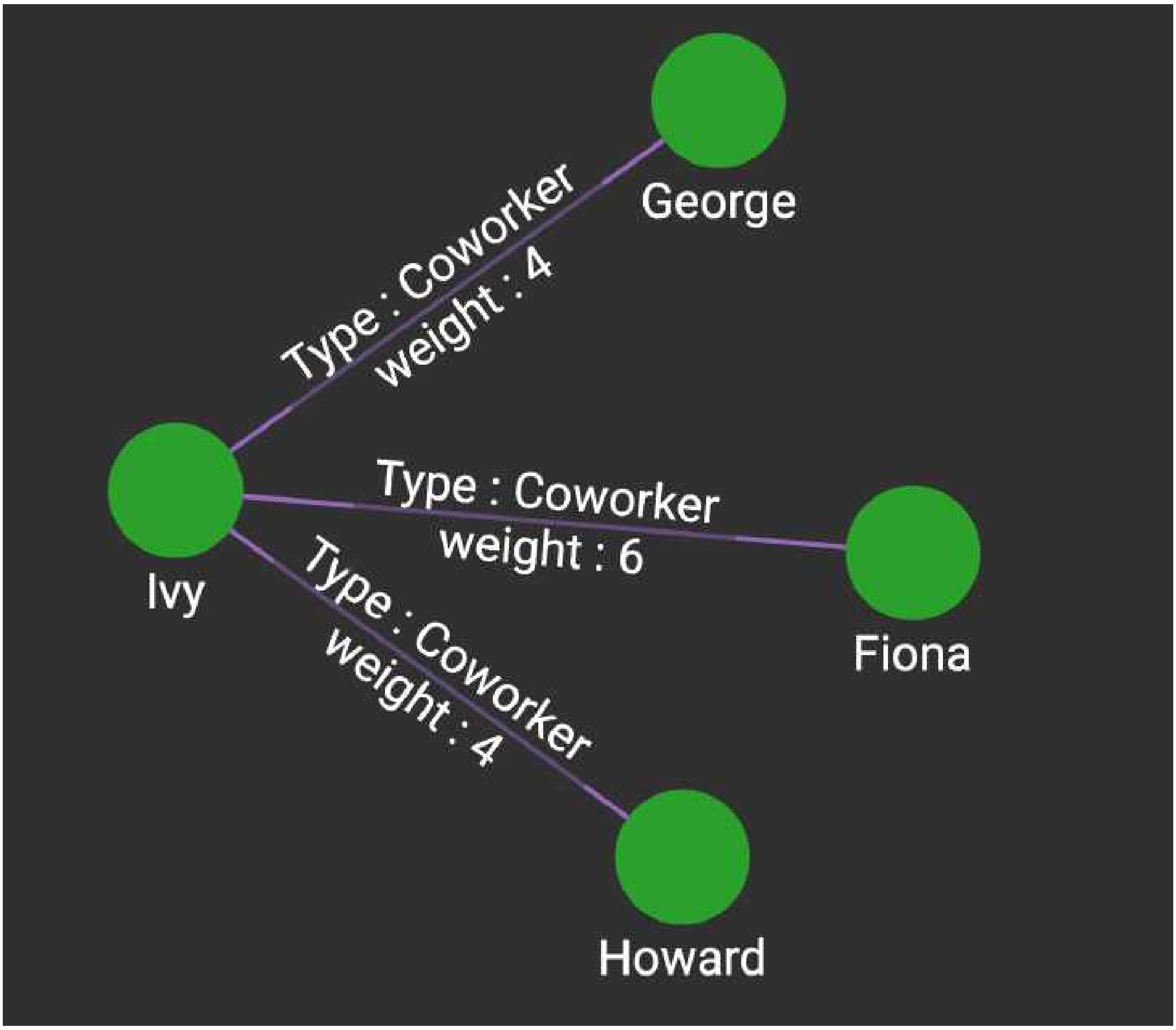
Visualized results of example graph social10 graph with Coworker edges

This graph has 3 components. Minimum Spanning Tree finds a tree for one component, so which component it will work on depends on what vertex we give as the starting point. If we select Fiona, George, Howard, or Ivy as the start vertex, then it works on the 4-vertex component on the left. You can start from any vertex in the component and get the same or an equivalent MST result.

The figure below shows the result of

```
# Use _ for default values
RUN QUERY mst(("Ivy", "Person"), ["Person"], ["Coworker"] "weight", "INT"
_, _, _, _)
```

Note that the value for the one vertex is `("Ivy", "Person")`. In GSQL, this 2-tuple format which explicitly gives the vertex type is used when the query is written to accept a vertex of any type.

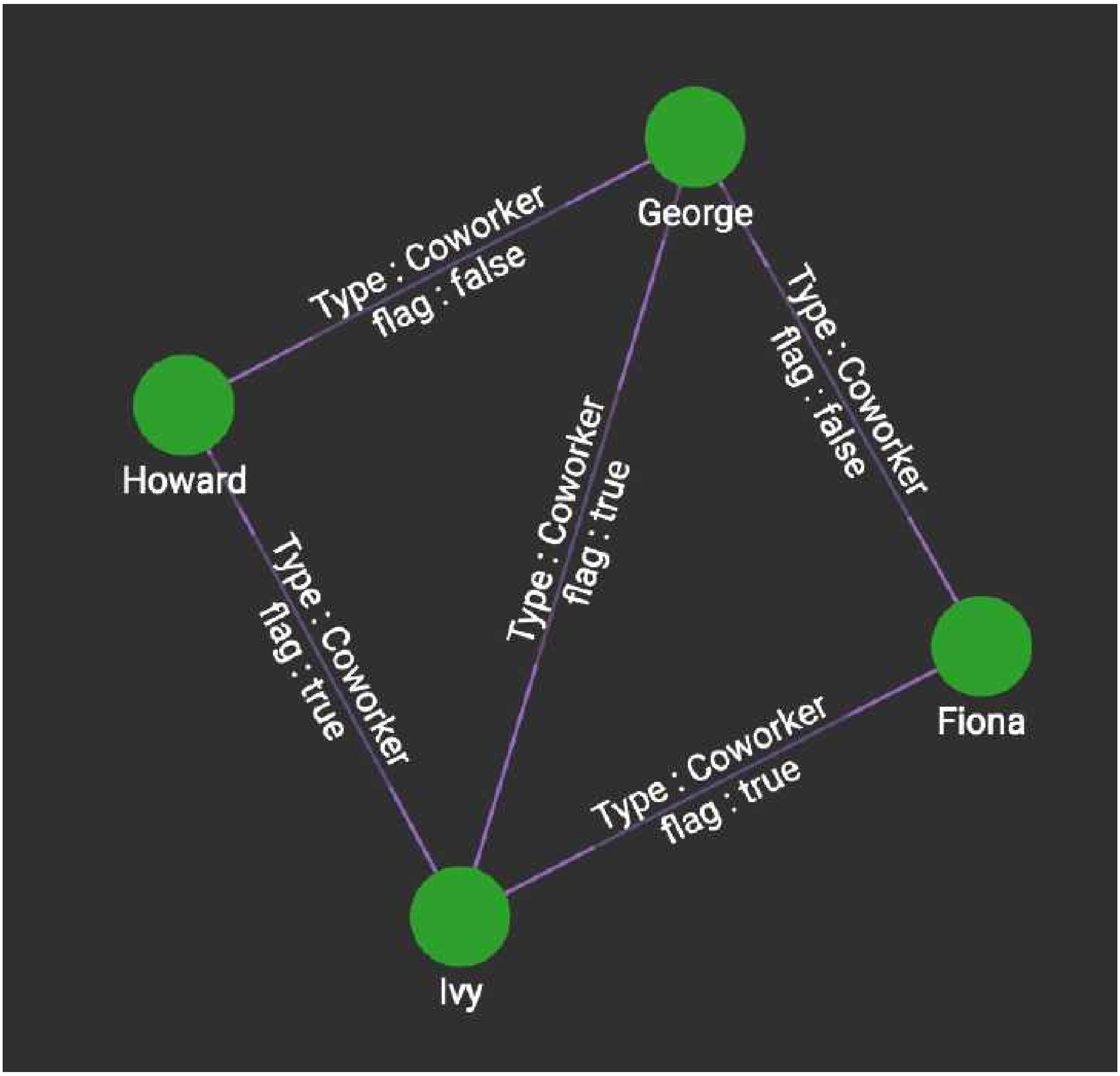


Visualized results of example query on social10 graph

File output:

```
From,To,Weight
Ivy,Fiona,6
Ivy,Howard,4
Ivy,George,4
```

The attribute version requires a boolean attribute on the edge, and it will assign the attribute to "true" if that edge is selected in the MST:



Visualized results of example query on social10 graph, with Coworker edges & edge attribute "flag"

Minimum Spanning Forest

Given an undirected graph with one or more connected components, a minimum spanning forest is a set of minimum spanning trees, one for each component. The library implements the algorithm in section 6.2 of Qin et al. 2014: http://www-std1.se.cuhk.edu.hk/~hcheng/paper/SIGMOD2014qin.pdf.

Specifications

```
tg_msf (SET<STRING> v_type, SET<STRING> e_type, STRING wt_attr, STRING wt
BOOL print_accum = TRUE, STRING result_attr = "", STRING file_path = "")
```

Characteristic	Value
Result	Computes a minimum spanning forest. If the JSON or file output selected, the output is the set of edges that form the MSF. If the result_attr option is selected, the edges which are part of the MSF are tagged True; other edges are tagged False.
Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `STRING wt_attr`: Name of edge weight attribute • `STRING wt_type`: Data type of edge weight attribute: "INT", "FLOAT", or "DOUBLE" • `BOOL print_accum`: If True, output JSON to standard output • `STRING result_attr`: If not empty, store result values (BOOL) to this attribute • `STRING file_path`: If not empty, write output to this file.

Result Size	V - c, V = number of vertices, c = number of components
Time Complexity	O((V+E) * logV)
Graph Types	Undirected edges

Example

Refer to the example for the MST algorithm. This graph has 3 components. MSF will find an MST for each of the three components.

Maximal Independent Set

An **independent set of vertices** does not contain any pair of vertices that are neighbors, i.e., ones which have an edge between them. A [maximal independent set](#) ↗ is the largest independent set that contains those vertices; you cannot improve upon it unless you start over with a different independent set. However, the search for the largest possible independent set (the **maximum independent set**, as opposed to the maximal independent set) is an NP-hard problem: there is no known algorithm that can find that answer in polynomial time. So we settle for the maximal independent set.

This algorithm finds use in applications wanting to find the most efficient configuration which "covers" all the necessary cases. For example, it has been used to optimize delivery or transit routes, where each vertex is one transit segment and each edge connects two segments that can NOT be covered by the same vehicle.

Specifications

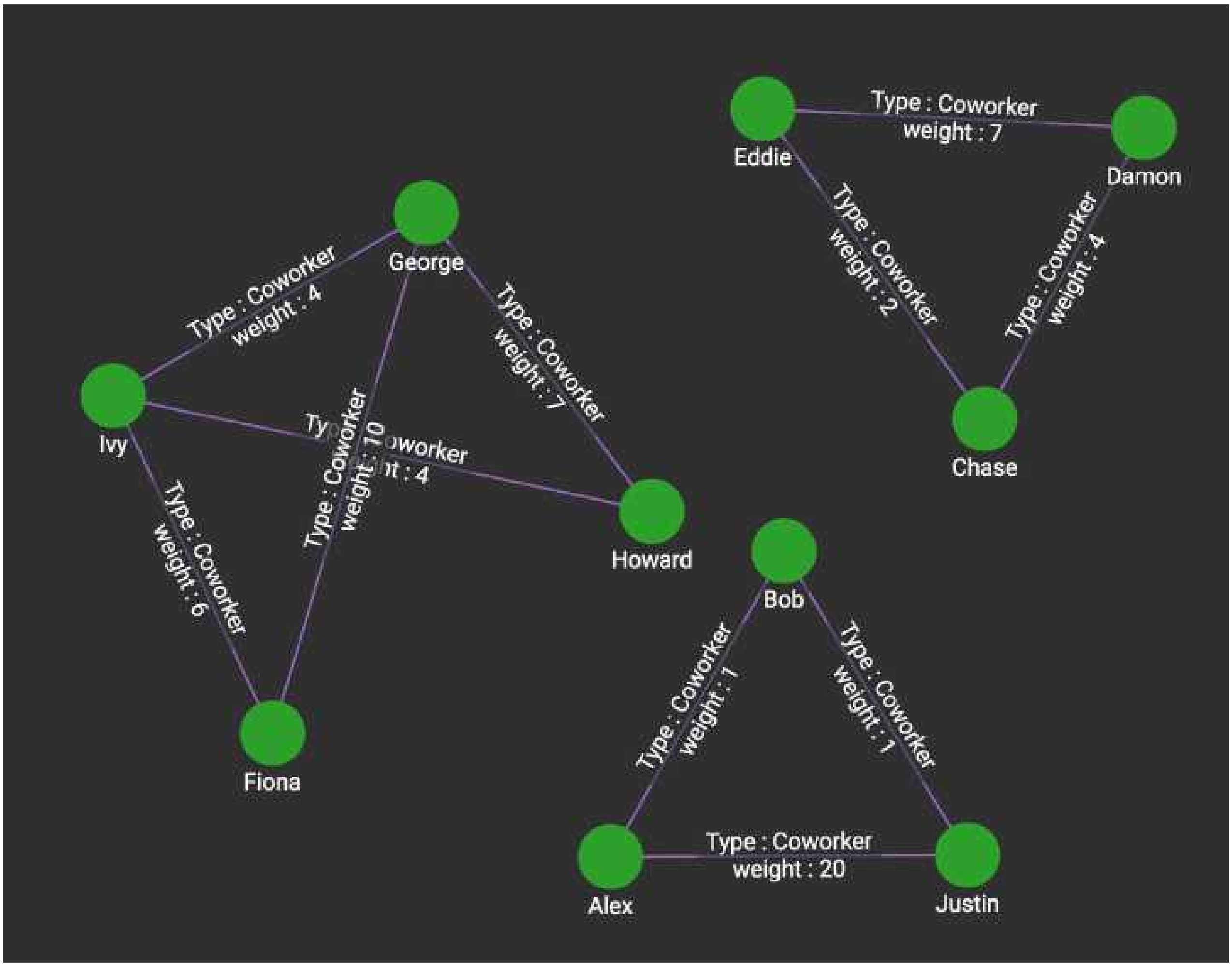
```
tg_maximal_indep_set(STRING v_type, STRING e_type,
INT max_iter = 100, BOOL print_accum = TRUE, STRING file_path = "")
```

Characteristic	Value
Result	A set of vertices that form a maximal independent set.
Input Parameters	• `STRING v_type`: Name of vertex type to use • `STRING e_type`: Name of edge type to use • `INT max_iter`: maximum number of iterations for the search • `BOOL print_accum`: If True, output JSON to standard output

	• `STRING file_path` : If not empty, write output to this file.
Result Size	Size of the MIS: unknown. Worst case: If the graph is a set of N unconnected vertices, then the MIS is all N vertices.
Time Complexity	O(E), E = number of edges
Graph Types	Undirected edges

Example

Consider our social10 graph, with three components.



It is clear that for each of the two triangles -- (Alex, Bob, Justin) and (Chase, Damon, Eddie) -- we can select one vertex from each triangle to be part of the MIS. For the 4-vertex component (Fiona, George, Howard, Ivy), it is less clear what will happen. If the algorithm selects either George or Ivy, then no other independent vertices remain in the component. However, the algorithm could select both Fiona and Howard; they are independent of one another.

This demonstrates the uncertainty of the Maximal Independent Set algorithm and how it differs from Maximum Independent Set. A *maximum* independent set algorithm would *always* select Fiona and Howard, plus 2 others, for a total of 4 vertices. The *maximal* independent set algorithm relies on chance. It could return either 3 or 4 vertices.

Cycle Detection

The Cycle Detection problem seeks to find all the cycles (loops) in a graph. We apply the usual restriction that the cycles must be "simple cycles", that is, they are paths that start and end at the same vertex but otherwise never visit any vertex twice.

There are two versions of the task: for directed graphs and undirected graphs. The GSQL algorithm library currently supports only directed cycle detection. The Rocha–Thatte algorithm ↗ is an efficient distributed algorithm, which detects all the cycles in a directed graph. The algorithm will self-terminate, but it is also possible to stop at k iterations, which finds all the cycles having lengths up to k edges.

The basic idea of the algorithm is to (potentially) traverse every edge in parallel, again and again, forming all possible paths. At each step, if a path forms a cycle, it records it and stops extending it. More specifically:

Initialization
For each vertex, record one path consisting of its own id. Mark the vertex as Active.

Iteration steps:
For each Active vertex v:

1. Send its list of paths to each of its out-neighbors.
2. Inspect each path P in the list of the paths received:
    - If the first id in P is also id(v), a cycle has been found:
        - Remove P from its list.
        - If id(v) is the least id of any id in P, then add P to the Cycle List. (The purpose is to count each cycle only once.)
    - Else, if id(v) is somewhere else in the path, then remove P from the path list (because this cycle must have been counted already).
    - Else, append id(v) to the end of each of the remaining paths in its list.

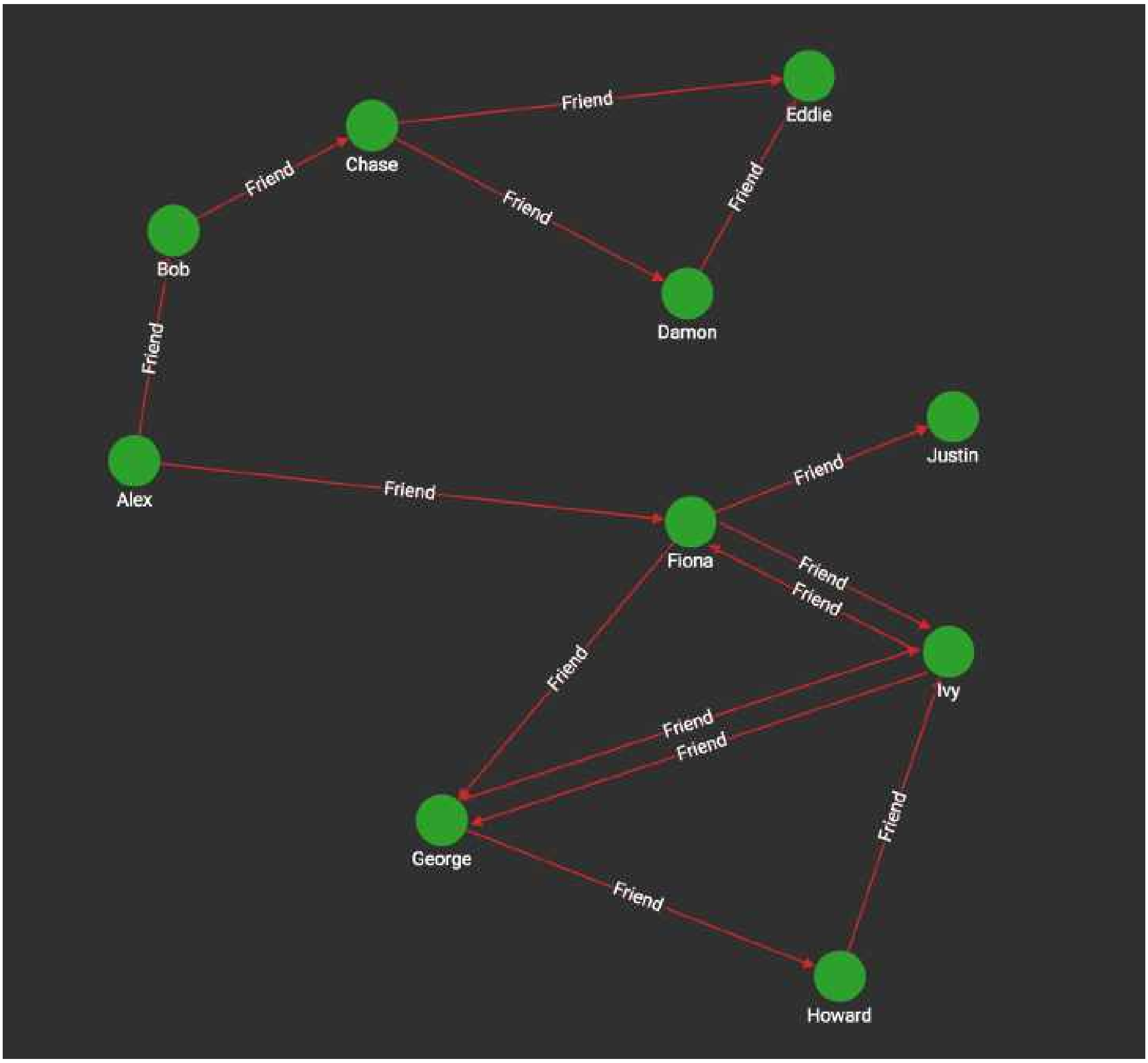
Specifications

```
tg_cycle_detection (SET<STRING> v_type, SET<STRING> e_type, INT depth,
  BOOL print_accum = TRUE, STRING file_path = "")
```

Characteristic	Value
Result	Computes a list of vertex id lists, each of which is a cycle. The result is available in 2 forms: • streamed out in JSON format • written to a file in tabular format
Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `INT depth`: Maximum cycle length to search for = maximum number of iterations • `BOOL print_accum`: If True, output JSON to standard output • `STRING file_path`: If not empty, write output to this file.
Result Size	Number of cycles * average cycle length Both of these measures are not known in advance.
Time Complexity	O(E *k), E = number of edges. k = min(max. cycle length, depth parameter)
Graph Types	Directed

Example

In the social10 graph, there are 5 cycles, all with the Fiona-George-Howard-Ivy cluster.



Visualized results of cycle_detection("Person", "Friend", 10) on social10 graph

```
[
  {
    "@@cycles": [
      [
        "Fiona",
        "Ivy"
      ],
      [
        "George",
        "Ivy"
      ],
      [
        "Fiona",
        "George",
        "Ivy"
      ],
      [
        "George",
        "Howard",
        "Ivy"
      ],
      [
        "Fiona",
        "George",
        "Howard",
        "Ivy"
      ]
    ]
  }
]
```

Estimated Diameter

The diameter of a graph is the worst-case length of a shortest path between any pair of vertices in a graph. It is the farthest distance to travel, to get from one vertex to another, if you always take the shortest path. Finding the diameter requires calculating (the lengths of) all shortest paths, which can be quite slow.

This algorithm uses a simple heuristic to estimate the diameter. rather than calculating the distance from each vertex to every other vertex, it selects K vertices randomly, where K is a user-provided parameter. It calculates the distances from each of these K vertices to all other vertices. So, instead of calculating V*(V-1) distances, this algorithm only calculates K*(V-1) distances. The higher the value of K, the greater the likelihood of hitting the actual longest shortest path.

The current version only computes unweighted distances.

This algorithm query employs a subquery called **max_BFS_depth**. Both queries are needed to run the algorithm.

Specifications

```
tg_estimate_diameter (SET<STRING> v_type, SET<STRING> e_type, INT seed_se
  BOOL print_accum = TRUE, STRING file_path = "", BOOL display = FALSE)
```

Characteristic	Value
Result	Returns the estimated value for the diameter of the graph
Input Parameters	• `SET<STRING> v_type`: Names of vertex types to use • `SET<STRING> e_type`: Names of edge types to use • `INT seed_set_length`: The number (K) of random seed vertices to use • `BOOL print_accum`: If True, output JSON to standard output

	• `STRING file_path`: If not empty, write output to this file.
Result Size	one integer
Time Complexity	O(k*E), E = number of edges, k = number of seed vertices
Graph Types	Directed

Similarity Algorithms

Euclidean Distance (Beta)

Euclidean distance measures the straight line distance between two points in n-dimensional space. The algorithm takes two vectors denoted by `ListAccum` and return the Euclidean distance between them.

> This algorithm is implemented as a user-defined function. You need to follow the steps in Add a User-Defined Function to add the function to GSQL. After adding the function, you can call it in any GSQL query in the same way as a built-in GSQL function.

Specifications

```
tg_euclidean_distance_accum(A, B)
```

Parameters

Name	Description	Data type
`A`	An n-dimensional vector denoted by a `ListAccum` of length `n`	`ListAccum<INT/UINT/FLOAT/DOUBLE>`
`B`	An n-dimensional vector denoted by a `ListAccum` of length `n`	`ListAccum<INT/UINT/FLOAT/DOUBLE>`

Return value

The Euclidean distance between the two vectors.

Example

Query

```
CREATE QUERY euclidean_example() FOR GRAPH social {
  ListAccum<INT> @@a = [1, 2, 3];
  ListAccum<INT> @@b = [4, 5, 6];
  double distance = tg_euclidean_distance_accum(@@a, @@b);
  PRINT distance;
}
```

Result

```
  {
    "distance": 5.19615
  }
```

Overlap Similarity (Beta)

The **overlap coefficient**, or **Szymkiewicz–Simpson coefficient**, is a similarity measure ↗ that measures the overlap between two finite sets.

$$overlap(X,Y) = \frac{|X \cap Y|}{\min(|X|,|Y|)}$$

The algorithm takes two vectors denoted by `ListAccum` and returns the overlap coefficient between them.

> This algorithm is implemented as a user-defined function. You need to follow the steps in Add a User-Defined Function to add the function to GSQL. After adding the function, you can call it in any GSQL query in the same way as a built-in GSQL function.

Specification

```
tg_overlap_similarity_accum( A, B )
```

Parameters

Name	Description	Data type
`A`	An n-dimensional vector denoted by a `ListAccum` of length `n`	`ListAccum<INT/UINT/FLOAT/DOUBLE>`
`B`	An n-dimensional vector denoted by a `ListAccum` of length `n`	`ListAccum<INT/UINT/FLOAT/DOUBLE>`

Return value

The overlap coefficient between the two vectors.

Example

Query

```
CREATE QUERY overlap_example(/* Parameters here */) FOR GRAPH social
  ListAccum<INT> @@a = [1, 2, 3];
  ListAccum<INT> @@b = [2, 2, 3];
  double overlap_similarity = tg_overlap_similarity_accum(@@a, @@b);
  PRINT overlap_similarity;
}
```

Result

```
[
  {
    "overlap_similarity": 0.66667
  }
]
```

Pearson Similarity (Beta)

The **Pearson correlation coefficient** is a measure of linear ↗ correlation ↗ between two sets of data. It is the ratio between the covariance ↗ of two variables and the product of their standard deviations ↗.

The formula for calculating the Pearson correlation coefficient is as follows:

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

This algorithm is implemented as a user-defined function. You need to follow the steps in Add a User-Defined Function to add the function to GSQL. After adding the function, you can call it in any GSQL query in the same way as a built-in GSQL function.

Specifications

```
tg_pearson_similarity_accum(A, B)
```

Parameters

Name	Description	Data type
`A`	An n-dimensional vector denoted by a `ListAccum` of length `n`	`ListAccum<INT/UINT/FLOAT/DOUBLE>`
`B`	An n-dimensional vector denoted by a `ListAccum` of length `n`	`ListAccum<INT/UINT/FLOAT/DOUBLE>`

Return value

The Pearson correlation coefficient between the two vectors.

Example

Query

```
CREATE QUERY pearson_example() FOR GRAPH social {
  ListAccum<INT> @@a = [1, 2, 3];
  ListAccum<INT> @@b = [2, 2, 3];
  double pearson_similarity = tg_pearson_similarity_accum(@@a, @@b);
  PRINT pearson_similarity;
}
```

Result

```
  {
    "pearson_similarity": 0.86603
  }
```

Cosine Similarity of Neighborhoods (Single-Source)

To compare two vertices by cosine similarity ↗, the selected properties of each vertex are first represented as a vector. For example, a property vector for a Person vertex could have the elements age, height, and weight. Then the cosine function is applied to the two vectors.

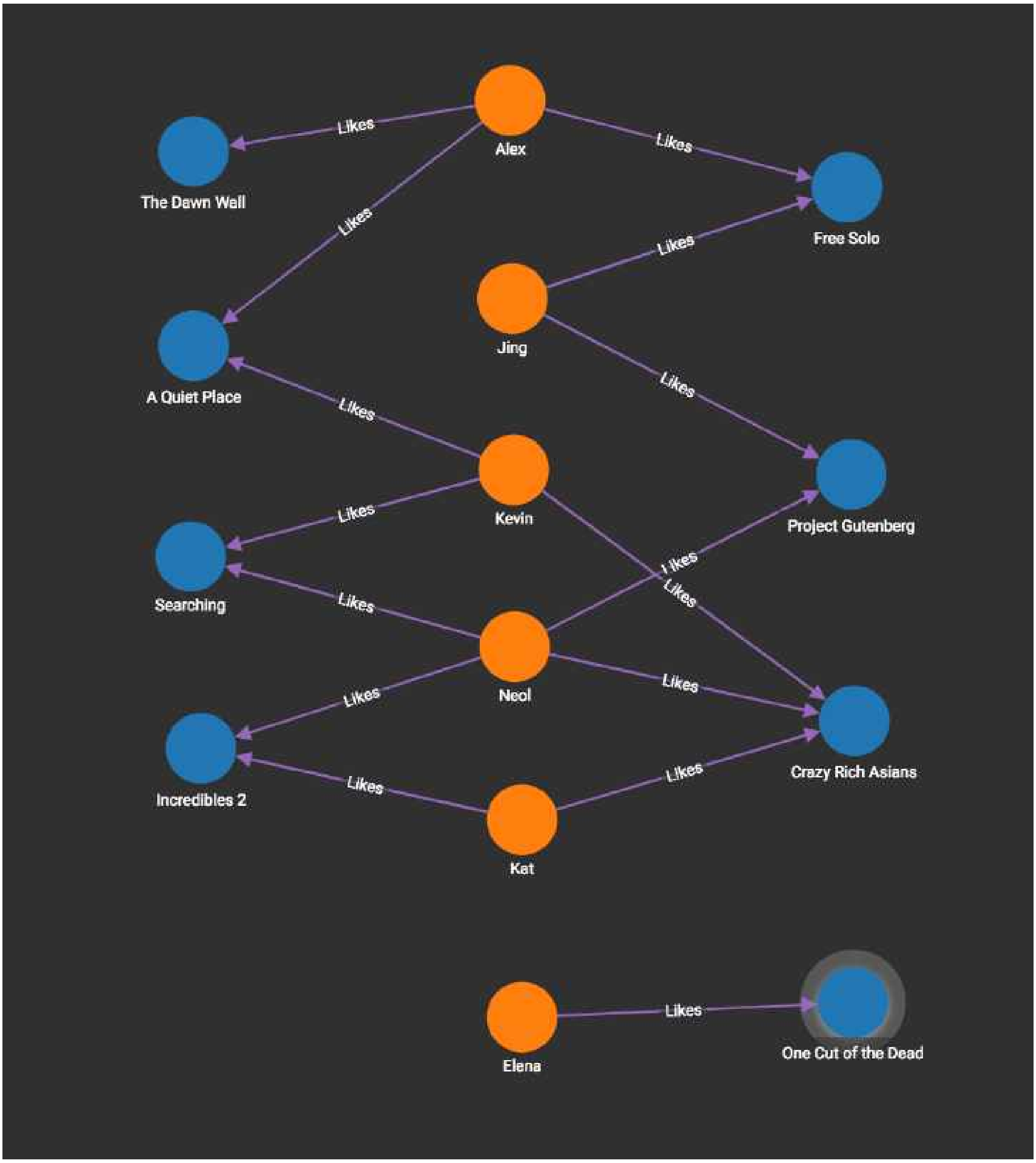
The cosine similarity of two vectors A and B is defined as follows:

$$cos(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_i A_i B_i}{\sqrt{\sum_i A_i^2}\sqrt{\sum_i B_i^2}}$$

If A and B are identical, then cos(A, B) = 1. As expected for a cosine function, the value can also be negative or zero. In fact, cosine similarity is closely related to the Pearson correlation coefficient.

For this library function, the feature vector is the set of edge weights between the two vertices and their neighbors.

In the movie graph shown in the figure below, there are Person vertices and Movie vertices. Every person may give a rating to some of the movies. The rating score is stored on the Likes edge using the weight attribute. For example, in the graph below, Alex gives a rating of 10 to the movie "Free Solo".



movie graph

Specifications

```
tg_cosine_nbor_ss (VERTEX source, SET<STRING> e_type, SET<STRING> re_type
  STRING weight, INT top_k, INT output_limit, BOOL print_accum = TRUE,
  STRING file_path = "", STRING similarity_edge = "")
RETURNS (MapAccum<VERTEX, FLOAT>)
```

Characteristic	Value
Result	The top K vertices in the graph that have the highest similarity scores, along with their scores. The result is available in three forms: • streamed out in JSON format • written to a file in tabular format • stored as a vertex attribute value
Input Parameters	• `VERTEX source`: Source vertex • `SET<STRING> e_type`: Edge type to traverse • `SET<STRING> re_type`: Reverse edge type to traverse • `STRING weight:` The edge attribute to use as the weight of the edge • `INT top_k`: Number of vertices • `INT output_limit:` • `BOOL print_accum`: If `true`, output JSON to standard output. • `STRING filepath`: If provided, the output will be written to this file path in CSV format. • `STRING similarity_edge`: If provided, the similarity score will be saved to this edge.
Result Size	`top_k`
Time Complexity	O(D^2), D = outdegree of vertex v
Graph Types	Undirected or directed edges, weighted edges

The output size is always K (if K <= N), so the algorithm may arbitrarily choose to output one vertex over another if there are tied similarity scores.

Example

Given one person's name, this algorithm calculates the cosine similarity between this person and each other person where there is at one movie they have both rated.

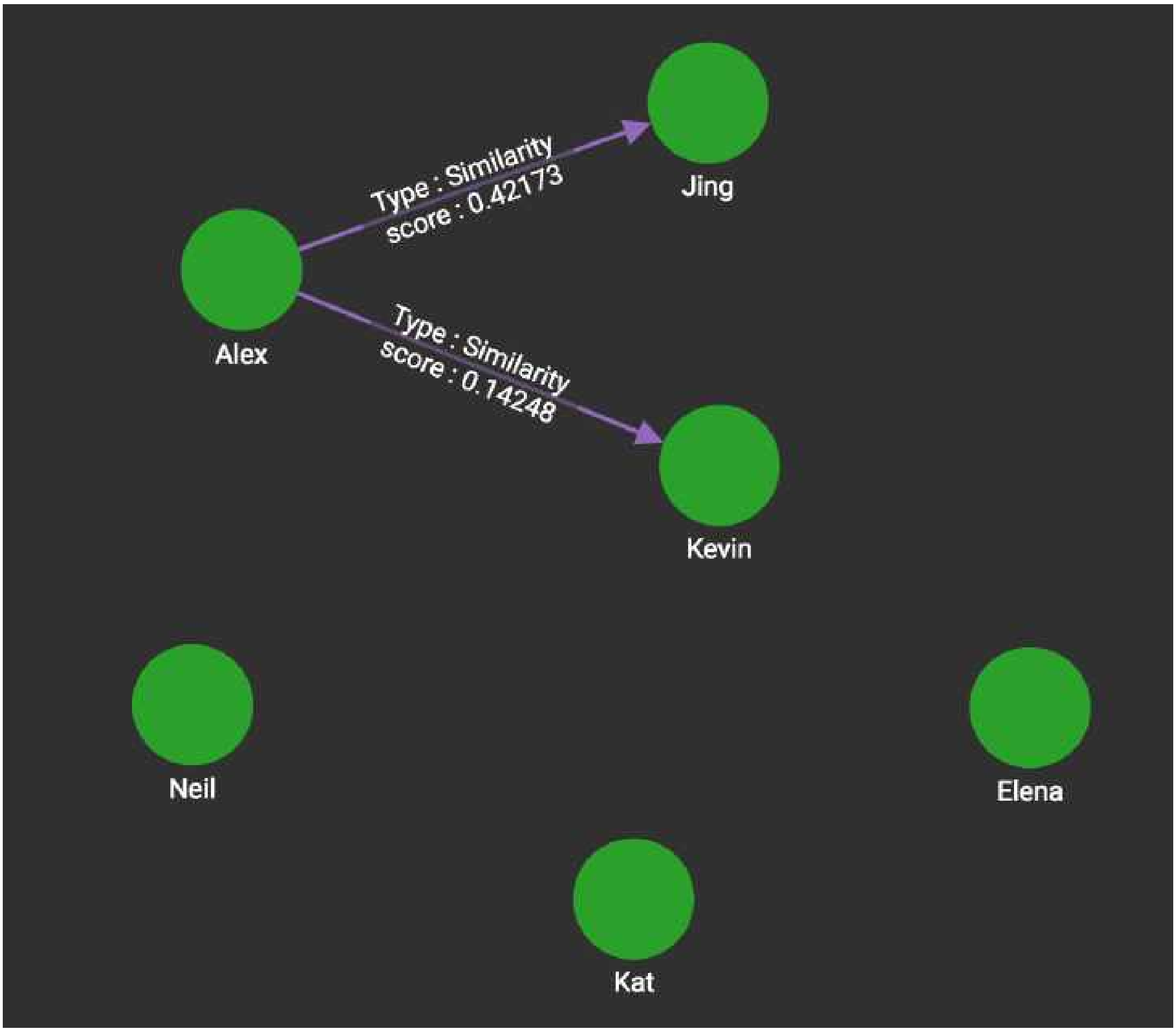
In the previous example, if the source vertex is Alex, and `top_k` is set to 5, then we calculate the cosine similarity between him and two other persons, Jing and Kevin. The JSON output shows the top 5 similar vertices and their similarity score in descending order. The output limit is 5 persons, but we have only 2 qualified persons:

```
[
  {
    "@@result_topk": [
      {
        "vertex1": "Alex",
        "vertex2": "Jing",
        "score": 0.42173
      },
      {
        "vertex1": "Alex",
        "vertex2": "Kevin",
        "score": 0.14248
      }
    ]
  }
]
```

The FILE version output is not necessarily in descending order. It looks like the following:

```
Vertex1,Vertex2,Similarity
Alex,Kevin,0.142484
Alex,Jing,0.421731
```

The ATTR version inserts an edge into the graph with the similarity score as an edge attribute whenever the score is larger than zero. The result looks like this:


Type : Similarity
score : 0.42173
Jing
Alex
Type : Similarity
score : 0.14248
Kevin
Neil
Elena
Kat

Cosine Similarity of Neighborhoods (All Pairs)

This algorithm computes the same similarity scores as the cosine similarity of neighborhoods, single-source algorithm (`cosine_nbor_ss`), except that it considers ALL pairs of vertices in the graph (for the vertex and edge types selected by the user). Naturally, this algorithm will take longer to run. For very large and very dense graphs, this may not be a practical choice.

Specifications

```
tg_cosine_nbor_ap (SET<STRING> v_type, SET<STRING> e_type, SET<STRING> re
  STRING weight, INT top_k, INT output_limit, BOOL print_accum = TRUE,
  STRING similarity_edge = "", STRING file_path = "")
```

Characteristic	Value
Result	The top *k* vertex pairs in the graph which have the highest similarity scores, along with their scores. The result is available in three forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• `SET<STRING> v_type`: Vertex types to calculate cosine similarity score for • `SET<STRING> e_type`: Edge types to traverse • `SET<STRING> re_type`: Reverse edge types to traverse • `STRING weight`: Edge attribute to use as weight • `INT top_k`: the number of vertex pairs with the highest similarity scores to

	return • `INT output_limit`: If >=0, max number of vertices to output to JSON. • `BOOL print_accum`: If `true, output JSON to standard output.` • `STRING similarity_edge`: If provided, the similarity score will be saved to this edge • `filepath` (for file output only): the path to the output file
Result Size	`top_k`
Time Complexity	O(E), E = number of edges
Graph Types	Undirected or directed edges, weighted edges

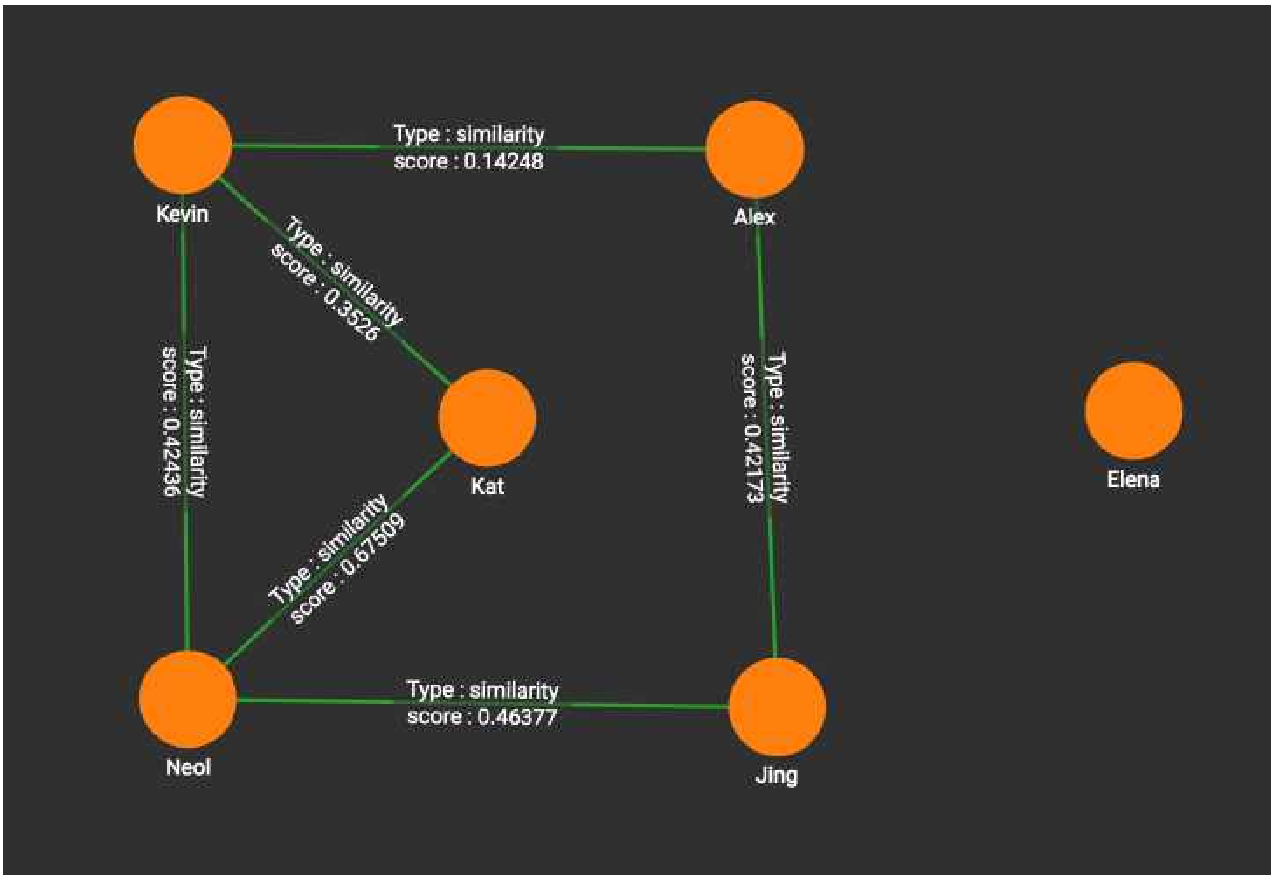
Example

Using the movie graph, calculate the cosine similarity between all pairs and show the top 5 pairs. This is the JSON result:

```
[
  {
    "@@total_result": [
      {
        "vertex1": "Kat",
        "vertex2": "Neil",
        "score": 0.67509
      },
      {
        "vertex1": "Jing",
        "vertex2": "Neil",
        "score": 0.46377
      },
      {
        "vertex1": "Kevin",
        "vertex2": "Neil",
        "score": 0.42436
      },
      {
        "vertex1": "Jing",
        "vertex2": "Alex",
        "score": 0.42173
      },
      {
        "vertex1": "Kat",
        "vertex2": "Kevin",
        "score": 0.3526
      }
    ]
  }
]
```


The FILE output is similar to the output of cosine_nbor_file.

The ATTR version will create *k* edges:


Type : similarity
score : 0.14248
Kevin
Alex
Type : similarity
score : 0.3526
Type : similarity
score : 0.42436
Kat
Type : similarity
score : 0.42173
Elena
Type : similarity
score : 0.67509
Type : similarity
score : 0.46377
Neol
Jing

Cosine Similarity of Neighborhoods (Batch)

This algorithm computes the same similarity scores as the Cosine similarity of neighborhoods, all pairs algorithm, except that it starts from all of the vertices as the source vertex and computes its similarity scores with its neighbors for all the vertices in parallel. Since this is a memory-intensive operation, it is split into batches to reduce peak memory usage. The user can specify how many batches it is to be split into. Compared with the Cosine similarity of neighborhoods, all pairs algorithm, this algorithm allows you to split the workload into multiple batches and reduces the burden on memory.

This algorithm has a time complexity of *O(E)*, where E is the number of edges, and runs on graphs with weighted edges (directed or undirected).

Specifications

```
tg_cosine_batch(STRING vertex_type, STRING edge_type, STRING edge_attribu
INT topK, BOOL print_accum = true, STRING file_path,
STRING similarity_edge, INT num_of_batches=1)
```

Parameters

Name	Description
`v_type`	Vertex type to calculate similarity for
`e_type`	Directed edge type to traverse
`edge_attribute`	Name of the attribute on the edge type to use as the weight
`topK`	Number of top scores to report for each vertex

`print_accum`	If `true`, output JSON to standard output.
`similarity_edge`	If provided, the similarity score will be saved to this edge.
`file_path`	If not empty, write output to this file in CSV.

Result

The result of this algorithm is the top *k* cosine similarity scores and their corresponding pair for each vertex. The score is only included if it is greater than 0.

The result can be output in JSON format, in CSV to a file, or saved as a similarity edge in the graph itself.

Example

Using the `social10` graph, we can calculate the cosine similarity of every person to every other person connected by the `Friend` edge, and print out the top *k* most similar pairs for each vertex.

```
GSQL > RUN QUERY tg_cosine_batch("Person", "Friend", "weight", 5, true, "

// Every vertex and their most similar pairs ranked by their Cosine
// Similarity score.
[
  {
    "start": [
      {
        "attributes": {
          "start.@heap": [
            {
              "val": 0.49903,
              "ver": "Howard"
            },
            {
              "val": 0.43938,
              "ver": "George"
            },
            {
              "val": 0.05918,
              "ver": "Alex"
            },
            {
              "val": 0.05579,
              "ver": "Ivy"
            }
          ]
        },
        "v_id": "Fiona",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@heap": []
        },
        "v_id": "Justin",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@heap": []
        },
        "v_id": "Bob",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@heap": [
```

```
            {
              "val": 0.22361,
              "ver": "Bob"
            },
            {
              "val": 0.21213,
              "ver": "Alex"
            }
          ]
        },
        "v_id": "Chase",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@heap": [
            {
              "val": 0.57143,
              "ver": "Bob"
            },
            {
              "val": 0.12778,
              "ver": "Chase"
            }
          ]
        },
        "v_id": "Damon",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@heap": []
        },
        "v_id": "Alex",
        "v_type": "Person"
      },
      {
        "attributes": {
          "start.@heap": [
            {
              "val": 0.64253,
              "ver": "Alex"
            },
            {
              "val": 0.63607,
              "ver": "Ivy"
            },
            {
```

```
            "val": 0.27091,
            "ver": "Howard"
          },
          {
            "val": 0.14364,
            "ver": "Fiona"
          }
        ]
      },
      "v_id": "George",
      "v_type": "Person"
    },
    {
      "attributes": {
        "start.@heap": []
      },
      "v_id": "Eddie",
      "v_type": "Person"
    },
    {
      "attributes": {
        "start.@heap": [
          {
            "val": 0.94848,
            "ver": "Fiona"
          },
          {
            "val": 0.6364,
            "ver": "Alex"
          },
          {
            "val": 0.31046,
            "ver": "George"
          },
          {
            "val": 0.1118,
            "ver": "Howard"
          }
        ]
      },
      "v_id": "Ivy",
      "v_type": "Person"
    },
    {
      "attributes": {
        "start.@heap": [
          {
            "val": 1.09162,
```


$$Jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Specifications

```
tg_jaccard_nbor_ss (VERTEX source, STRING e_type, STRING rev_e_type,  INT
  BOOL print_accum = TRUE, STRING similarity_edge_type = "",STRING file_p
```

```
            "val": 0.78262,
            "ver": "Ivy"
          },
          {
            "val": 0.11852,
            "ver": "George"
          }
        ]
      },
      "v_id": "Howard",
      "v_type": "Person"
    }
  ]
 }
]
```

Characteristic	Value
Result	The top *k* vertices in the graph that have the highest similarity scores, along with their scores. The result is available in three forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• `SET<STRING> v_type`: Vertex type to calculate similarity score for • `SET<STRING> e_type`: Edge type to traverse • `SET<STRING> re_type`: Reverse edge type to traverse • `INT top_k`: the number of vertex pairs with the highest similarity scores to return • `BOOL print_accum`: Boolean value that decides whether to output to console • `STRING similarity_edge`: If provided, the similarity score will be saved to this edge • `STRING filepath:` If provided, the algorithm will output to the file path in CSV format
Result Size	`top_k`
Time Complexity	O(D^2), D = outdegree of vertex v
Graph Types	Undirected or directed edges, unweighted edges

The algorithm will not output more than K vertices, so the algorithm may arbitrarily choose to output one vertex over another if there are tied similarity scores.

Example

Using the movie graph, we run `jaccard_nbor_ss("Neil", 5)`:

```
[
  {
    "@@result_topK": [
      {
        "vertex1": "Neil",
        "vertex2": "Kat",
        "score": 0.5
      },
      {
        "vertex1": "Neil",
        "vertex2": "Kevin",
        "score": 0.4
      },
      {
        "vertex1": "Neil",
        "vertex2": "Jing",
        "score": 0.2
      }
    ]
  }
]
```

If the source vertex (person) doesn't have any common neighbors (movies) with any other vertex (person), such as Elena in our example, the result will be an empty list:

```
[
  {
    "@@result_topK": []
  }
]
```

Jaccard Similarity of Neighborhoods (All Pairs)

This algorithm computes the same similarity scores as the Jaccard similarity of neighborhoods, single-source algorithm, except that it considers ALL pairs of vertices in the graph (for the vertex and edge types selected by the user). Naturally, this algorithm will take longer to run. For very large and very dense graphs, this algorithm may not be a practical choice.

Specifications

```
tg_jaccard_nbor_ap(STRING v_type, STRING e_type, STRING re_type, INT top_
  BOOL print_accum = TRUE, STRING similarity_edge = "", STRING file_path
```

Characteristic	Value
Result	The top *k* vertex pairs in the graph that have the highest similarity scores, along with their scores. The result is available in three forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• `SET<STRING> v_type`: Vertex types to calculate similarity score for • `SET<STRING> e_type`: Edge types to traverse • `SET<STRING> re_type`: Reverse edge types to traverse • `INT top_k`: the number of vertex pairs with the highest similarity scores to return • `BOOL print_accum`: Boolean value that decides whether to output to console

	• `STRING similarity_edge`: If provided, the similarity score will be saved to this edge • `STRING file_path`: If provided, the algorithm will output to the file path in CSV format
Result Size	`top_k`
Time Complexity	O(E^2 / V), V = number of vertices, E = number of edges
Graph Types	Undirected or directed edges, unweighted edges

The algorithm will not output more than K vertex pairs, so the algorithm may arbitrarily chose to output one vertex pair over another, if there are tied similarity scores.

Example

For the movie graph, calculate the Jaccard similarity between all pairs and show the 5 most similar pairs: jaccard_nbor_ap(5). This is the JSON output :

```
[
  {
    "@@total_result": [
      {
        "vertex1": "Kat",
        "vertex2": "Neil",
        "score": 0.5
      },
      {
        "vertex1": "Kevin",
        "vertex2": "Neil",
        "score": 0.4
      },
      {
        "vertex1": "Jing",
        "vertex2": "Alex",
        "score": 0.25
      },
      {
        "vertex1": "Kat",
        "vertex2": "Kevin",
        "score": 0.25
      },
      {
        "vertex1": "Jing",
        "vertex2": "Neil",
        "score": 0.2
      }
    ]
  }
]
```

Jaccard Similarity of Neighborhoods (Batch)

This algorithm computes the same similarity scores as the Jaccard similarity of neighborhoods, all pairs, except that it starts from all of the vertices as the source vertex and computes its similarity scores with its neighbors for all the vertices in parallel. Since this is a memory-intensive operation, it is split into batches to reduce peak memory usage. The user can specify how many batches it is to be split into. Compared with the Jaccard similarity of neighborhoods, all pairs, this algorithm allows you to split the workload into multiple batches and reduces the burden on memory.

This algorithm has a time complexity of *O(E)*, where E is the number of edges, and runs on graphs with unweighted edges (directed or undirected).

Specifications

```
tg_jaccard_batch (STRING v_type, STRING e_type, STRING re_type, INT topK,
BOOL print_accum = true, STRING similarity_edge, STRING file_path,
INT num_of_batches = 1)
```

Parameters

Name	Description
`v_type`	Vertex type to calculate similarity for
`e_type`	Directed edge type to traverse
`re_type`	Reverse edge type to traverse
`topK`	Number of top scores to report for each vertex
`print_accum`	If `true`, output JSON to standard output.

`similarity_edge`	If provided, the similarity scores will be saved to this edge type.
`file_path`	If a file path is provided, the algorithm will output to a file specified by the file path in CSV format
`num_of_batches`	Number of batches to divide the query into

Result

The result contains the top *k* Jaccard **similarity** scores for each vertex and its corresponding pair. A pair is only included if its similarity is greater than 0, meaning there is at least one common neighbor between the pair. The result is available in JSON format, or can be output to a file in CSV, or it can be saved as an edge on the graph itself. A JSON formatted result could look like this:

```
// Run jaccard_batch on social10 graph traversing through Friend edges
[
  {
    "Start": [
      {
        "attributes": {
          "Start.@heap": [
            {
              "val": 0.33333,
              "ver": "Howard"
            },
            {
              "val": 0.25,
              "ver": "Ivy"
            },
            {
              "val": 0.25,
              "ver": "George"
            }
          ]
        },
        "v_id": "Fiona",
        "v_type": "Person"
      },
      {
        "attributes": {
          "Start.@heap": []
        },
        "v_id": "Justin",
        "v_type": "Person"
      },
      {
        "attributes": {
          "Start.@heap": []
        },
        "v_id": "Bob",
        "v_type": "Person"
      },
      {
        "attributes": {
          "Start.@heap": [
            {
              "val": 0.5,
              "ver": "Damon"
            }
          ]
        },
        "v_id": "Chase",
```

```
      "v_type": "Person"
    },
    {
      "attributes": {
        "Start.@heap": [
          {
            "val": 0.5,
            "ver": "Chase"
          }
        ]
      },
      "v_id": "Damon",
      "v_type": "Person"
    },
    {
      "attributes": {
        "Start.@heap": [
          {
            "val": 0.33333,
            "ver": "Ivy"
          }
        ]
      },
      "v_id": "Alex",
      "v_type": "Person"
    },
    {
      "attributes": {
        "Start.@heap": [
          {
            "val": 0.5,
            "ver": "Howard"
          },
          {
            "val": 0.25,
            "ver": "Fiona"
          }
        ]
      },
      "v_id": "George",
      "v_type": "Person"
    },
    {
      "attributes": {
        "Start.@heap": []
      },
      "v_id": "Eddie",
      "v_type": "Person"
```

```
      },
      {
        "attributes": {
          "Start.@heap": [
            {
              "val": 0.33333,
              "ver": "Alex"
            },
            {
              "val": 0.25,
              "ver": "Fiona"
            }
          ]
        },
        "v_id": "Ivy",
        "v_type": "Person"
      },
      {
        "attributes": {
          "Start.@heap": [
            {
            },
            {
              "val": 0.33333,
              "ver": "Fiona"
            }
          ]
        },
        "v_id": "Howard",
        "v_type": "Person"
      }
    ]
  }
]
```


$$A(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{\log |N(u)|}$$

Specification

```
CREATE QUERY tg_adamic_adair(VERTEX a, VERTEX b, SET<STRING> e_type)
```

Parameters

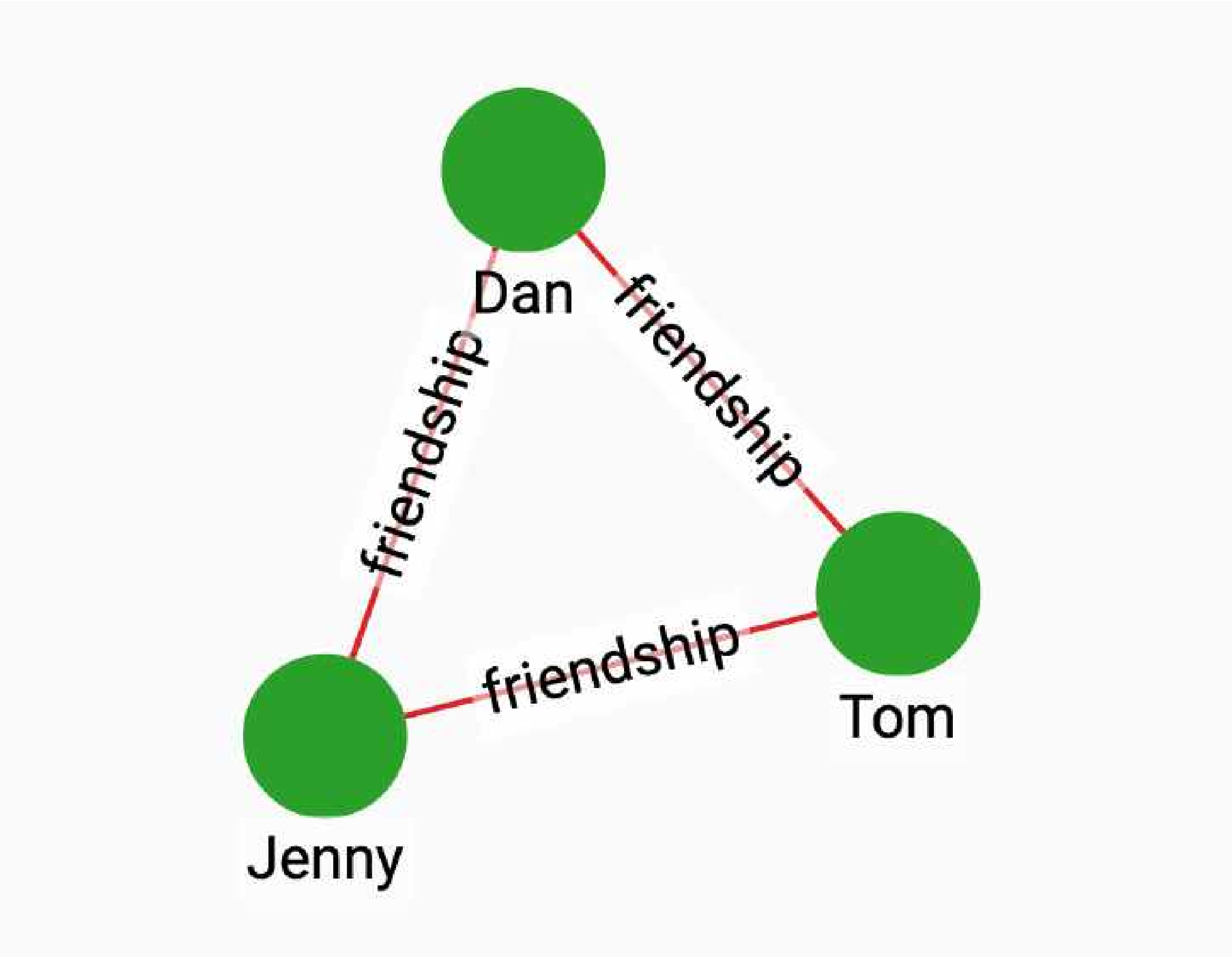
Name	Description	Data type
`a`	A vertex.	`VERTEX`
`b`	A vertex.	`VERTEX`
`e_type`	Edge types to traverse.	`SET<STRING>`

Return value

The Adamic Adar index between two vertices. If the two vertices do not have common neighbors, the algorithm will return a division by 0 error.

Example

Suppose we have the graph below:



Running the algorithm between Jenny and Dan will give us a result of 1/log(2)=3.321931.

Query

```
RUN QUERY tg_adamic_adar (("Jenny", "person"), ("Dan", "person"), ["f
```

Result

```
{
    "error": false,
    "message": "",
    "version": {
    "schema": 1,
    "edition": "enterprise",
    "api": "v2"
    },
    "results": [{"@@closeness": 3.32193}]
}
```

Common Neighbors (Beta)

The Common Neighbors algorithm calculates the number of common neighbors between two vertices.

Specification

```
CREATE QUERY tg_common_neighbors(VERTEX a, VERTEX b, SET<STRING> e_type)
```

Parameters

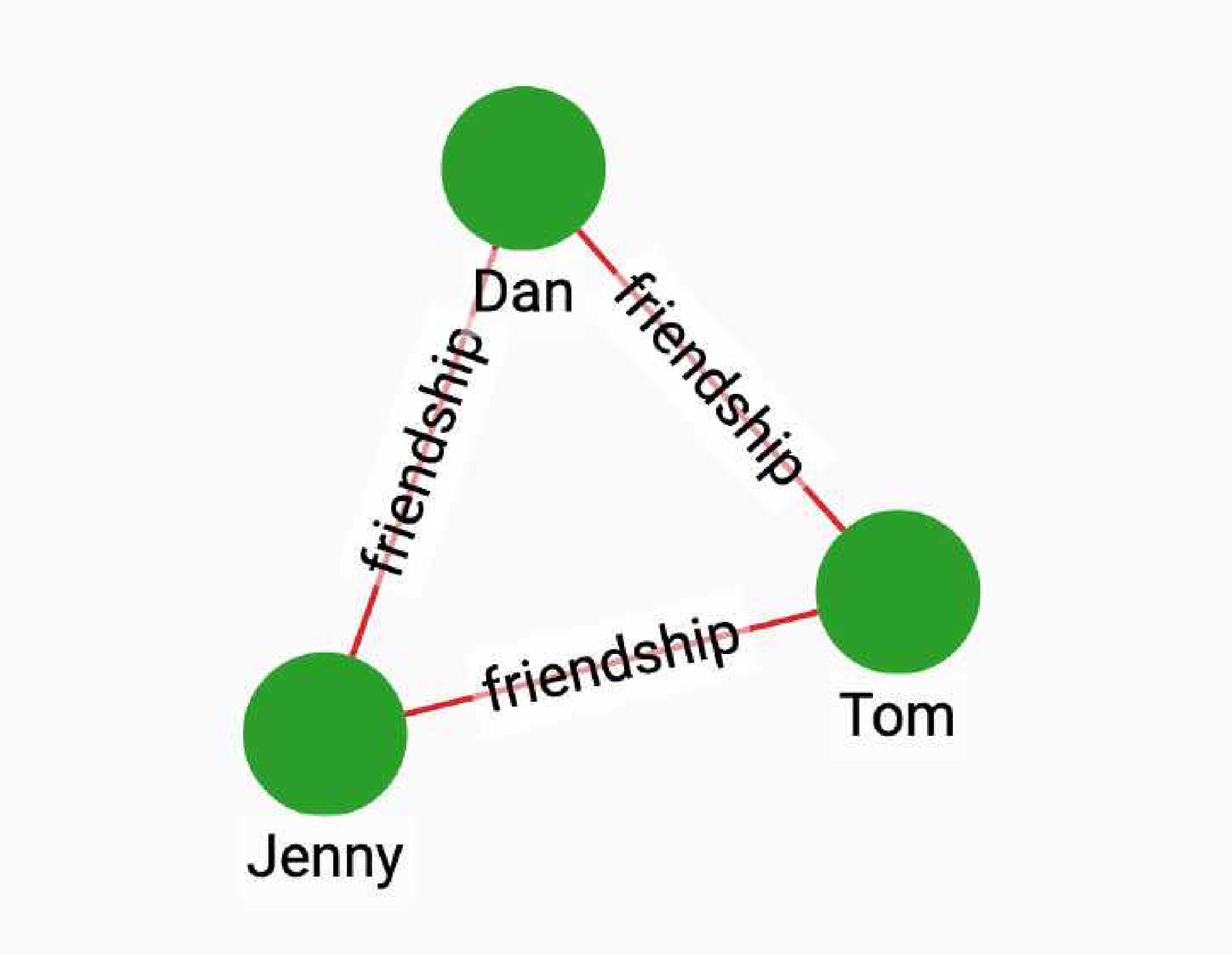
Name	Description	Data type
`a`	A vertex.	`VERTEX`
`b`	A vertex.	`VERTEX`
`e_type`	Edge types to traverse.	`SET<STRING>`

Return value

The number of common neighbors between two vertices.

Example

Suppose to have the following graph:



Running the algorithm between Dan and Jenny will show that they have 1 common neighbor:

Query

```
RUN QUERY common_neighbors (("Jenny", "person"), ("Dan", "person"),
    ["friendship"])
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 1,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"closeness": 1}]
}
```

Preferential Attachment (Beta)

Preferential Attachment is a measure to compute the closeness of vertices, based on the number of their neighbors. The algorithm returns the product two vertices' number of neighbors.

For more information, see Preferential Attachment ↗.

Specification

```
CREATE QUERY tg_preferential_attachment(VERTEX a, VERTEX b,
    SET<STRING> e_type)
```

Parameters

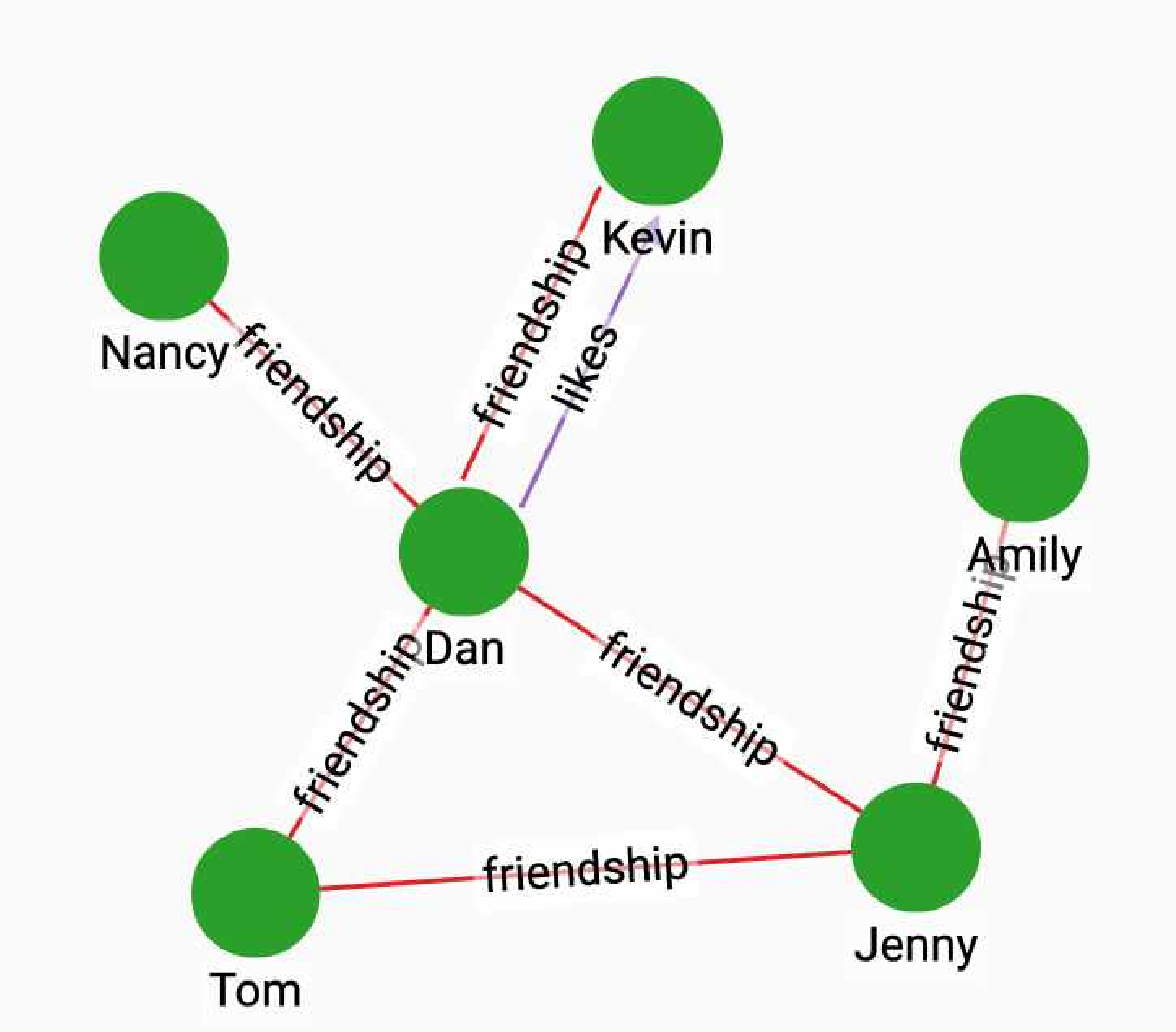
Name	Description	Data type
`a`	A vertex.	`VERTEX`
`b`	A vertex.	`VERTEX`
`e_type`	Edge types to traverse.	`SET<STRING>`

Return value

The product of the number of neighbors of the two vertices.

Example

Suppose we have the following graph:



Since Dan has four neighbors, while Jenny has three, the return value of the algorithm is 3∗4=12:

Query

```
RUN QUERY preferential_attachment (("Jenny", "person"), ("Dan", "pers
    ["friendship"])
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 1,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"closeness": 12}]
}
```

Resource Allocation (Beta)

Resource Allocation is used to compute the closeness of nodes based on their shared neighbors. It is computed by the following formula:

$$RA(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|N(u)|}$$

Where $N(u)$ is the set of nodes adjacent to *u*.

Specification

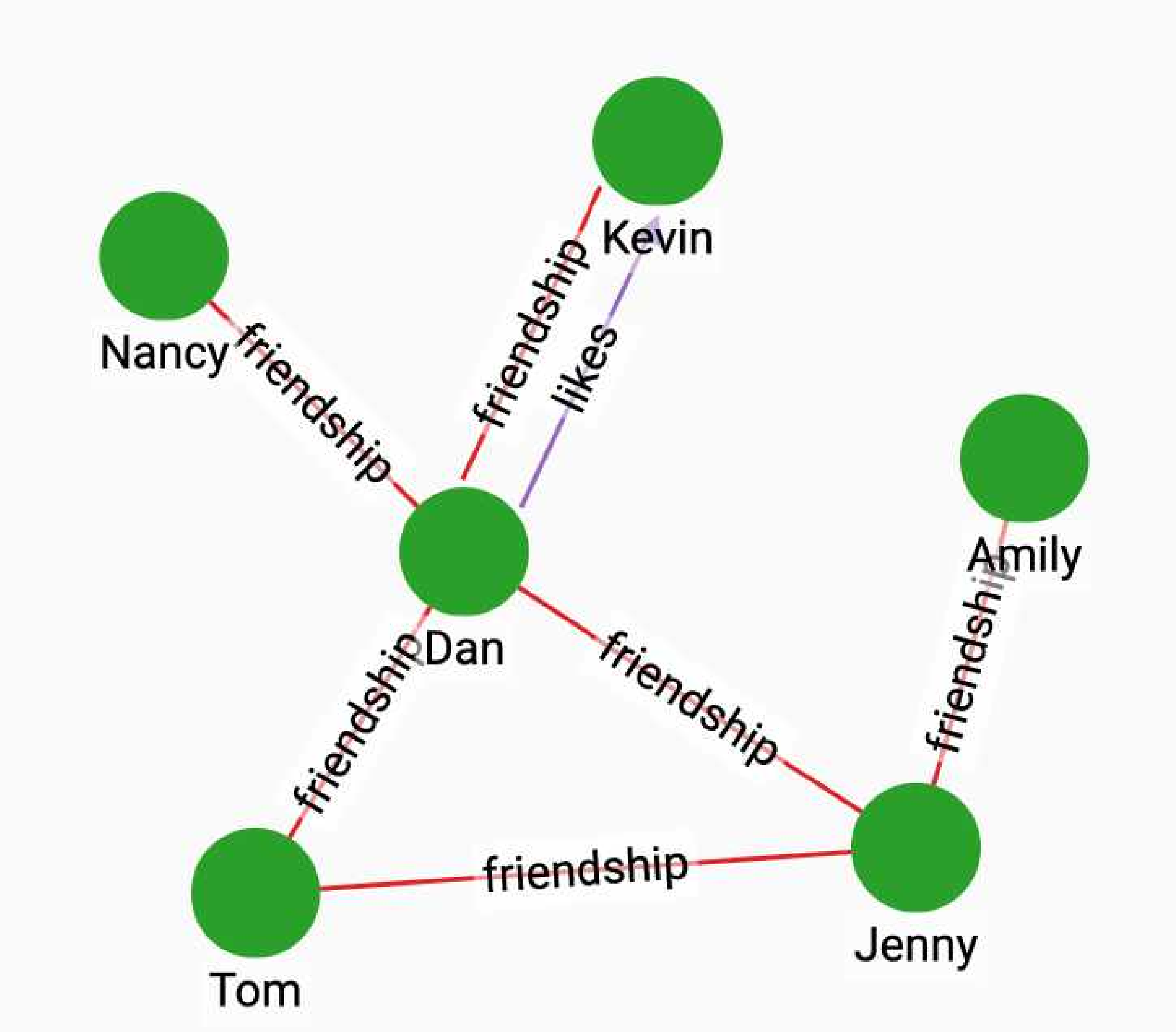
```
CREATE QUERY tg_resource_allocation(VERTEX a, VERTEX b, SET<STRING> e_typ
```

Parameters

Name	Description	Data type
`a`	A vertex.	`VERTEX`
`b`	A vertex.	`VERTEX`
`e_type`	Edge types to traverse.	`SET<STRING>`

Example

Suppose we have the following graph:



Since Dan and Jenny has one shared neighbor `Tom`, who has two neighbors, running the algorithm between Dan and Jenny with friendship edges would give us a result of 0.5.

Query

```
RUN QUERY resource_allocation (("Jenny", "person"), ("Dan", "person")
    ["friendship"])
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 1,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{
    "@@closeness": 0.5
  }]
}
```

Same Community (Beta)

This algorithm takes two vertices, and returns 1 if they two vertices are in the same community, and returns 0 if they are not in the same community.

> The algorithm assumes that community detection has already completed and that a community ID is stored in an integer attribute on each vertex.

Specifications

```
CREATE QUERY tg_same_community(VERTEX a, VERTEX b, STRING
    communityAttribute)
```

Parameters

Name	Description	Data type
`a`	A vertex.	`VERTEX`
`b`	A vertex.	`VERTEX`
`communityAttribute`	The community attribute used to store a vertex's community ID.	`STRING`

Return value

- Returns 1 if the two vertices are in the same community.
- Returns 0 if the two vertices are not in the same community.

Example

Suppose we have the following vertices:

Their community IDs were generated by running the weakly connected component algorithm on the graph. If we run the algorithm between Kevin and Jenny, we get 1 because the two vertices are in the same community as indicated by their `community` attribute:

Query

```
RUN QUERY tg_same_community (("Jenny", "person"), ("Kevin", "person")
    "community")
```

Result

```json
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"1": 1}]
}
```

Total Neighbors (Beta)

The algorithm calculates the total number of neighbors of two vertices.

Specifications

```
CREATE QUERY tg_total_neighbors(VERTEX a, VERTEX b,
    SET<STRING> e_type)
```

Parameters

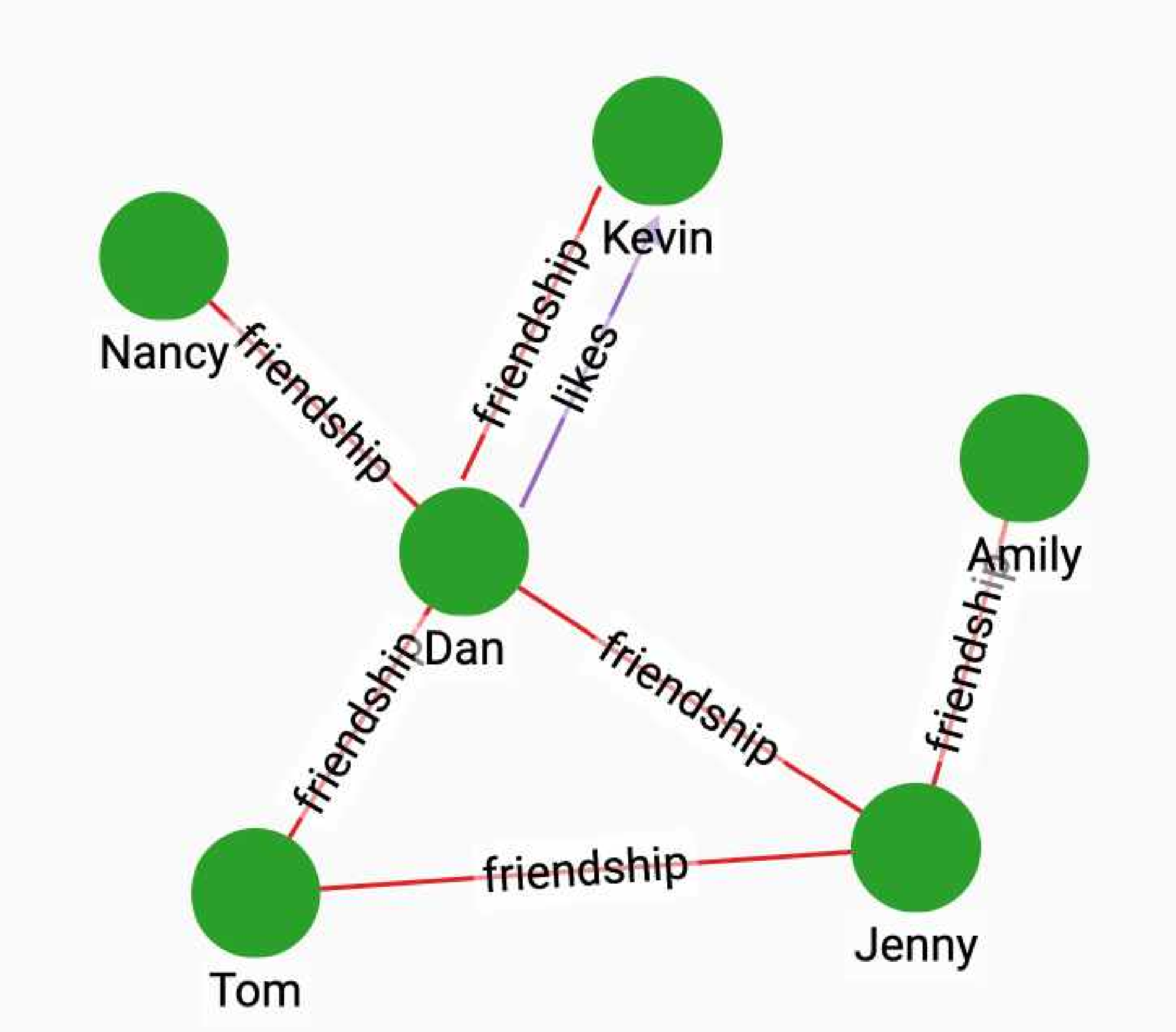
Name	Description	Data type
`a`	A vertex.	`VERTEX`
`b`	A vertex.	`VERTEX`
`e_type`	Edge types to traverse.	`SET<STRING>`

Return value

The total number of neighbors of two vertices.

Example

Suppose we have the following graph.



Dan and Jenny together have 6 neighbors in total. Running the algorithm between Dan and Jenny would give us a result of 6. Note that since Jenny and Dan are neighbors themselves, the union of their neighbors includes both Jenny and Dan:

Query

```
RUN QUERY tg_total_neighbors (("Jenny", "person"), ("Dan", "person"),
    ["friendship"])
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"closeness": 6}]
}
```

Release Notes, FAQs and Troubleshooting

Release Notes - TigerGraph Server 3.2

Release Date: Sep 30th, 2021

Release Notes for Previous Versions:

- Release Notes - TigerGraph 3.1 ↗
- Release Notes - TigerGraph 3.0 ↗
- Release Notes - TigerGraph 2.6 ↗
- Release Notes - TigerGraph 2.5 ↗
- Release Notes - TigerGraph 2.4 ↗
- Release Notes - TigerGraph 2.3 ↗
- Release Notes - TigerGraph 2.2 ↗
- For v2.1 and older, contact TigerGraph Support

For the running log of bug fixes, see the Change Log.

Compatibility with TigerGraph 3.1

The following changes were made to the built-in roles in TigerGraph's Role-based Access Control

- The built-in role `queryreader` can no longer run queries that include updates to the database.
  - To emulate the old `queryreader` role, create a role with all `queryreader` privileges, and also grant the `WRITE_DATA` privilege to the new role to allow users with the role to run queries that update the graph.
- The built-in role `admin` can no longer create users
  - To emulate the old `admin` role, create a global role with all `admin` privileges, and also grant the `WRITE_USER` privilege to the new role to allow users with the role to create users.

Compatibility with TigerGraph 2

Major revisions (e.g., from TigerGraph 2 to TigerGraph 3) are the opportunity to deliver significant improvements. While we make every effort to maintain backward compatibility, in selected cases APIs have changed or deprecated features have been dropped, in order to advance the overall product.

Data migration: A tool is available to migrate the data in TigerGraph 2.6 to TigerGraph 3.0. Please contact TigerGraph Support for assistance.

Query and API compatibility:

- Some gadmin syntax has changed. Notably. `gadmin set config` is now `gadmin config set`. Please see Managing with gadmin.
- Some features which were previously deprecated have been dropped. Please see V3.0 Removal of Previously Deprecated Features for a detailed list.

New Features

User-defined Roles

Privileges are introduced as the atomic unit for managing database user access. Database administrators can now define their own roles with customizable collections of privileges.

For details on this feature, see User Access Management.

Cross-region replication

TigerGraph 3.2 provides complete native support for all data and metadata cross-region replication including automated schema changes, user and query management.

For details on this feature, see Cross-Region Replication.

Containerization of services

TigerGraph 3.2 allows users to deploy TigerGraph single servers and clusters on Kubernetes.
Running applications in containers on Kubernetes provides rapid spin-up and repeatability across environments.

For details on this feature, see Kubernetes.

Cluster resizing

TigerGraph 3.2 provides built-in cluster management features that allow users to expand, shrink, and repartition their TigerGraph clusters.

For details on this feature, see Cluster resizing.

Workload Management

TigerGraph 3.2 provides the `GSQL-REPLICA` and `GSQL-THREAD-LIMIT` header to specify the replica for a query to run on and a thread limit that a query is allowed to use.

For details on this feature, see Run a query.

File output policy

Starting with TigerGraph 3.2, GSQL has a file output policy that contains a whitelist and a blacklist. GSQL queries will only write to the whitelist and are forbidden from writing to the blacklist.

Log viewing with Elasticsearch, Kibana, and Filebeat

Starting with TigerGraph 3.2, TigerGraph provides a gadmin command that can generate Filebeat configuration files for a TigerGraph cluster. Read our step-by-step guide to set up Elasticsearch, Kibana, and Filebeat to view TigerGraph logs.

General Improvements

Expanded library of built-in GSQL functions

TigerGraph 3.2 added over 30 built-in functions to the GSQL query language.

Relaxed variable declaration rules

Starting in TigerGraph 3.2, base type variables and accumulators can be declared anywhere in a query and are block-scoped.

Starting in TigerGraph 3.2, variables and accumulators can be initialized with expressions.

Parameter passing with JSON

Starting in TigerGraph 3.2, users can pass in parameters to a GSQL query with a JSON object.

Query installation improvements

TigerGraph 3.2 implemented the following improvements for query installation:

- Updating a subquery will no longer require reinstalling all dependent queries
- Schema change will no longer trigger reinstalling all queries of the graph
- Concurrent query installation between graphs is now supported

- When installing queries on a cluster, TigerGraph will now utilize the computing power of multiple nodes to compile the queries, greatly improving installation performance

WCAG-compliant accessibility features

The user interfaces of GraphStudio and Admin Portal - TigerGraph’s GUI are improved to meet WCAG accessibility criteria. More users across a wider range of physical abilities will now be able to work effectively with GraphStudio and the visual Admin Portal.

Edit mode for graph exploration in GraphStudio

Starting with TigerGraph 3.2, users need to enter Edit Mode in the Graph Exploration Panel in order to write to the graph.

Enhanced cluster service status for AdminPortal

Cluster service status is moved from the footer of the Admin Portal page to the bottom of the navigation menu.

Improved GSQL auto-complete and syntax highlighting in GraphStudio

We made improvements to GraphStudio's auto-complete and syntax highlighting features so users have a better experience writing and editing queries in GraphStudio.

Substring search within graph exploration result in GraphStudio

Starting with TigerGraph 3.2, users can search substring to locate the vertices in addition to using exact match.

Anonymous tuples as return type in subqueries

Starting in TigerGraph 3.2, subqueries in GSQL can return an anonymous tuple or a collection of anonymous tuples.

Function overloading

Starting in TigerGraph 3.2, function overloading is now available. Query UDFs with the same name but different signatures can be defined in the UDF library.

Known Issues

Applications

- **GraphStudio**
  - Known Issues for GraphStudio
  - The No-Code Data Migration feature is in Alpha release. Your feedback would be appreciated.
  - The No-Code Visual Query Builder is in Beta release. Your feedback would be appreciated.
- **AdminPortal**
  - Known Issues for AdminPortal

GSQL

- **Multiple (Conjunctive) Path Patterns**:
  - There are no known functional problems, but the performance has not be optimized. Your feedback would be appreciated.

- **DML type check error in V2 Syntax:**
  - GSQL will report a wrong type check error for Query block with multiple POST-ACCUM clauses and Delete/Update attribute operation.
- **Turn on GSQL HA manually when upgrading from 3.0.x**
  - Users who are upgrading from 3.0.X need to manually start GSQL HA service. Please reach out to support for help with the process documented in: https://tigergraph.freshdesk.com/a/solutions/articles/5000865072 ↗
- **Stale data visible after Deletes using index**
  - Queries that use secondary index may still see the vertices being deleted until after the snapshots are fully rebuilt.

Change Log

This page will document all the changes to TigerGraph product including New Features and Bug Fixes.

> Distributed Graph support and certain other enterprise-level features are available in the Enterprise Edition only. They do not pertain to the Developer Edition.

TigerGraph 3.2.1

Changes

- Increased limit for graph catalog size
- Added retention for Metadata topic in Kafka
- Improved error handling when retrieving patterns from server in Visual Query Builder

Fixes

- Fixed an issue during upgrade related to initiating Kafka
- Fixed a bug that in rare cases caused a catalog size limit issue
- Fixed an issue that slowed queries that write to files in 3.2.0
- Fixed a bug that in rare cases caused query compilation issues

TigerGraph 3.2

Release Date: 2021-09-30

Features

Check release notes: 3.2.0 Release notes ↗

Changes:

GSQL:

- ADD Edge Pair commands as part of Schema Change operations are now allowed.
- Query-calling-query limitation: Distributed main query cannot call a distributed sub-query.
- Default logging level for GSQL logs has been changed from DEBUG to INFO

Enhancements

Database Server:

- Core: Improve Abort transaction if the transaction is too long
- Core: GPE hung under high number of concurrent queries
- Core: Turn on transaction for RestPP Post for atomicity
- Core: Workload management: Specify replica for a query to run in a distributed cluster
- Core: Standardize the correct http response code for query requests
- GSQL: Query Installation Performance Improvements
    - Support longer reload time for Query installation on 1000's of queries
    - Don't drop parent queries when subquery is installed
- GSQL: Support for new built-in functions:
    - Math-related functions: round(), reverse(), repeat(), insert(), cot(), degrees(), radians, square, truncate, log2
    - String-related functions instr(), length(), substr(), PI(), rand(), lpad(), rpad(), replace(), ascii(), chr(), soundex(), difference(), translate(), space(), ltrim(), rtrim(), find_in_set(), left(), right();
- GSQL: Support FROM/TO vertex type change for edge type metadata

- GSQL: Support VLAC tags in Import and Export operations
- GSQL: Allow variable declaration anywhere in query body
- GSQL: Support initialization from an expression;
- GSQL: Support revoking superuser role from default user
- GSQL: Improve the error message displayed when connecting to LDAP server
- Platform: Upgrade to Java 11
- Platform: Add support for ubuntu20
- Platform: Show executor status and updated status of other services
- Platform: Run upgrade locally without ssh if user is local with only a single node
- Platform: Start/stop local executor will no longer need ssh,
- Platform: Increase Backup/Restore S3 upload Partition Size
- Platform: Make Backup/Restore Heartbeat timeout configurable to allow media with slower speeds.

GraphStudio:

- WCAG compliance changes
- Support overwriting exploration result
- Support duplicate file-edge mappings and fix setSelection error;
- Add graph information and variable names to auto-complete list;

Admin Portal:

- WCAG changes
- Support Privilege based management
- Improve unauthorized access warning popup message
- Display secrets table for each graph

Fixed

Database Server:

- Core: Kafka loader should exit gracefully
- Core: GPE crash if the request specifies an invalid replica
- Core: Health check for 1 mins in RESTPP startup
- Core: Fixed file loading failed due to OOM
- Core: Fixed no error message when edge does not exist
- Core: Fixed issue with deleted_vertex_check API after dropping vertex type;
- GSQL: LDAP user privilege parsing missed authorization checks
- GSQL: Fixed rhs check issue for direct interpret query;
- GSQL: Fixed print Vset issue with vertex accum declaration order;
- GSQL: Added semantic checker for rhs with the same name;
- GSQL: Export fails due to mismatching token of an unexpected graph
- GSQL: Fixed wrong name when looking up variable from global
- GSQL: Fix datetime_format function not working for v2 syntax
- GSQL: The result of printing string differs in interpret mode and installed mode
- GSQL: Fixed issue with Order by for interpret query
- GSQL: Fix to handle abort while adding queries if a concurrent delete fails
- Platform: Service status for KAFKA is down when one zookeeper server offline
- Platform: Fix for Admin log rotation time issue

GraphStudio:

- Addressed Schema change logic for reversed edge
- Fix for privilege based access control issue
- Fix for loading job information migration failure
- Remove loading job log on export;
- Remove graphName from loading job information interface;
- Use authorization token in header instead of logging in;
- Send heartbeat to keep client connection alive

TigerGraph 3.1.6

Release Date: 2021-08-09

Fixed

Application

- Configuration for light or dark mode in GraphStudio/Admin Portal
- Multiple maps from a single file to an edge are indistinguishable
- GraphStudio: Implement responsive design for all sizes of screens
- GraphStudio: Rearrange elements to avoid overlay in small screen
- GraphStudio: Support toolbar button announcement for screen readers
- GraphStudio: Support keyboard shortcut for focusing elements within working panels

TigerGraph 3.1.5

Release Date: 2021-07-23

Fixed

Database Server

- Core: GPE on DR cluster stuck in warm up state after failover due to invalid requests
- GSQL: Prevent QueryReader role to run any graph updates query
- GSQL: Validation script to check schema consistency issue
- Platform: Increase in proxy request buffer size for NGINX
- Platform: Change in GRPC maximum message size for GBAR backup of catalog data

Application

- GraphStudio: Reuse controller connections to avoid running out of used ports
- GraphStudio: Remove "change layout" button in toolbar in Visual Editor

TigerGraph 3.1.4

Release Date: 2021-07-01

Enhancements

- GSQL: `\requesttoken` API can be used to create authorization tokens using User name/password in addition to secret.
- GSQL: Secrets created without alias will be assigned a system-generated alias so that they can be dropped
- Platform: Nginx upgrade from 1.18.0 to 1.21.0
- Platform: Backup/Restore configuration improvements to allow use of slower HDD media for storage
- GraphStudio: UI enhancements to support WCAG compliance

Fixed

Database Server

- Core: GPE need to verify catalog updates after new schema changes are applied
- Core: Running Louvain algorithm as a distributed query crashed GPE due to unnecessary vertex activation
- Core: Backup failed with WaitForDeltaToBeProcessed timeout
- Core: Updated log messages to reference /deleted_vertex_check endpoint in RESTPP correctly
- GSQL: Fix schema consistency issues due to duplicate Vertex/Edge type names
- GSQL: Fix for schema consistency issue due to GPE referencing a dropped Vertex

- GSQL: Additional semantic check for local schema change job to prevent schema inconsistency
- GSQL: Error when making schema changes using UI/ Install all queries fails
- GSQL: Inconsistency between GSQL and GPE catalog data after 'Drop graph' fails
- GSQL: 'From' clause missing from delete loading jobs when Export Graph command is run
- GSQL: Query installation will fail due to wrong order of arguments in PRINT statement
- GSQL: *"Incompatible argument types for function/tuple evaluate"* error when using evaluate without second argument on v2 syntax
- GSQL: Designer Role unable to run a query in Interpret Mode
- Platform: Updates to Nginx templates for security updates
- Platform: Change in default value for UI request timeout to 3600

Application

- GraphStudio: Vertex and Edge statistics generation optimization to avoid Cluster CPU usage spike
- GraphStudio: Unexpected error when dropping edge with reversed edge
- GraphStudio: Fix for failure to migrate loading job info from 3.0.x to 3.1.2+

TigerGraph 3.1.3

Release Date: 2021-06-05

Enhancements

GraphStudio

- Theme color adjustment to meet Web Content Accessibility Guidelines(WCAG).

- Support responsive page layout for "Home" page, "Load Data" page and "Write Queries" page.
- Add information transcripts for visualization areas in each page.
- Add keyboard navigation in graph charts.
- Improve tabbing capability and tabbing order.
- Improve element status announcement.
- Add headings for the entire application.
- Add aria-labels for the entire application to meet WCAG compliance.
- Add captions for all table elements.

AdminPortal

- Theme color adjustment to meet WCAG compliance.

TigerGraph 3.1.2

Release Date: 2021-05-20

Features

- **SQL to GSQL translation** for Enterprise BI tools like Tableau and Power BI
  - This enriches data visualization tools with graph-enabled dashboards

Enhancements

- Core: Increase the maximum allowed size of Vertex/Edge delta files to allow larger number of updates for write-heavy applications.
- GSQL: Support for more than 10K elements in a Set<> of a query parameter
- GSQL: Support VertexAccessControl Tags in DBImportExport

Fixed

Database Server

- Core: Pick the latest version of GPE data for backup
- GSQL: datetime attribute type in a schema-level user-defined tuple translated as int32_t
- GSQL: NullPointerException when handle VSet variable in nested if statement
- GSQL: NullPointerException when using multiple POST-ACCUM clauses
- GSQL: INSERT statement with non-existent edge does not report error in V1 syntax
- GSQL: GSQL does not produce type error when inserting non-existent edge with vertices from query parameters
- GSQL: NoSuchElementException when using a non-existent edge on INSERT statement
- GSQL: Lexical error when a newline is followed by an exclamation mark (!) in a string
- GSQL: Printing string with newline fails compilation
- GSQL: Refresh RESTPP Token: output and default lifetime is not correct
- GSQL: Multiplicity propagation ACCUM clauses should reset only if the block is within a loop
- GSQL: Create user don't allow an empty password
- GSQL: Pattern match - propagation accumulator values not cleared
- GSQL: Push-down error reported for non-alias expressions
- GSQL: Support TAGS in DBImportExport
- GSQL: Fix TokenBank compilation slowdown
- Platform: Graceful handling of port used by Executor component
- Platform: Got failed to authenticate with GSQL server error when login with SSO on tg3.1.1
- Platform: Remove gsql password printing

GraphStudio

- The loading data status is incorrect while import a solution
- Imported solution with no modification, should not ask user to publish Data mapping.
- Failed to overwrite datafile in Map Data to Graph

AdminPortal

- Display of secrets on AdminPortal - User management should be paginated.

TigerGraph 3.1.1

Release Date: 2021-04-02

Changes:

- Change **BY(OR|OVERWRITE)** syntax to **BY OR|OVERWRITE** for explicit tag creation
- Changed name of 'dbsanitycheck' endpoint to 'deleted_vertex_check'

Enhancements

Database Server

- Core: Improved throttling mechanism for Updates when memory usage has hit critical threshold
- Core: Improved reliability of transferring in-memory data to on-disk within GSE
- Core: Logging improvements to support both time-based and size-based configuration for all the component logs
- Fixes/Enhancements for Vertex Level Access Control feature
  - GSQL: Performance improvement for tag creation only operations
  - GSQL: Make tag description optional

  - GSQL: Block altering taggable value of global vertex if being used in tag based graph
  - GSQL: Show tag expression of tag graphs in base graph “ls” command
  - GSQL: Allow vertex taggable property to be updated even if it is currently being used in a tag-based graph
- GSQL: Support for accumulators in table-style SELECT clause expression lists
- GSQL Query syntax extensions for table support
- GADMIN: Allow script to be used to configure LDAP TrustStore Path
- Platform: Security enhancement to allow HTTPS traffic only access securely through dedicated interfaces when SSLis enabled.
- Platform: Upgrade grpc to 1.33.0

GraphStudio

- Add a * in the label of a data source if the loading job is changed
- Return detailed error messages when install queries failed
- Enable only one column header to be editable at the same time
- Enable closing popup with Escape
- Add a max validator for timeout field for configuration
- Query name conflict check uses all available type names from GSQL

Fixed

Database Server

- Core: Retry logic for adding data to GSE in the DR cluster
- Core: Fix for GPE crash due to potential race condition between queries and updates.
- Core: Partial result output in extreme cases before a running query has finished
- Core: restpp crashed when missing parameter name
- Core: Fixed file loading job failures due to OOM
- GSQL: Fix for catalog access issue due to concurrent schema change requests

- GSQL: GPE crash due to incorrect catalog update issued by GSQL
- GSQL: LDAP password visible in GSQL logs
- GSQL: Exit code from GSQL CLI needs to return non-zero code if there is an error
- GSQL: Unable to run global schema change on global vertex if local vertex with same name exists
- GSQL: Query created through GSQL shell, but returns error through GraphStudio
- GSQL: Add check for GPE readiness for create/drop vertex/edge operations for global schema changes
- GSQL: GSQL v2 syntax - vertex-attached containers cannot be read in WHERE/ACCUM clauses
- GSQL: Enhance Export/Import by pre-creating necessary directories
- GSQL: Fix calling subquery without RETURNS clause
- GSQL: Code generation error for multiple dynamic expressions with the same parameter
- GSQL: Wrong result for the output of datetime_format function
- GSQL: SET<VERTEX> Not Working in Query Parameter
- GSQL: GLE error message uses incorrect terminology: 'batch mode' should say 'distributed query mode'
- GSQL: Printing vertex set variable with parentheses causes wrong printing for attributes
- GSQL: GSQL pattern match - incorrect WHERE condition parsing
- GSQL: GSQL query doesn't work on HA cluster when RESTPP#1 is down
- GSQL: Fix for Catalog backup file cleaning failure
- GSQL: Empty gsql password should not be allowed.
- GSQL: NullPointerException on creating a query with a body-level DML delete statement
- GSQL: Query cannot be dropped after its caller queries have been dropped
- Platform: Remove user authentication information after installation
- Platform: GSQL user defined functions are not backed up
- Platform: Residual GPE/GSE processes are not terminated before restore

- Platform: GBAR gracefully exit after ctrl-c
- Platform: guninstall does not take into account the password login
- Platform: gbar restore failed with message: Failed to import key-value store
- Platform: Single node 3.1 installation in in VMware private cloud environment in VMWare Private Cloud Environment
- Platform: Restore failure from S3 didn’t update the replicas correctly
- Platform: Check to prevent migration tool running twice
- Platform: GBAR restore fails with invalid checksums
- Platform: User didn’t receive correct feedback when incorrect password entered during 3.1 upgrade

GraphStudio

- Query goes back to a previous version after schema change in query editor
- Remove the use of regex for GSQL CLI and rely on exit code instead
- Progress bar hangs if query installation fails
- datetime's default value field does not support rfc3339 nor iso8601 format
- Export solution is only available for superuser
- Unexpected error when changing the schema (Fix from GSQL side)
- Update global schema after a local schema is dropped
- Uploading progress bar hangs after choosing unsupported file type
- Query editor does not display full text if line cannot break
- Undo button should clear the expand list
- JSON result of "write query" is not updated in error mode
- Not possible to unset/cancel custom radius in Graph Exploration
- Syntax highlighting is incomplete
- Link to License page from GST is wrong
- Long messages in Design Schema overlap vertex properties editor's ✓ button
- The loading progress bar is stuck if import fails
- The data mapping will disappear after change the global vertex's attribute
- Address Export/Import solution migration issues

Admin Portal

- Validate input on config management
- Ignore blank spaces in log search

TigerGraph 3.1.0

Release Date: 2020-12-02

Features

New features are described in 3.1.0 Release notes ↗.

Changes:

- GSQL: STRING COMPRESS data type will no longer be allowed for new data objects. However, existing objects with STRING COMPRESS data type will continue to work.
- GSQL: Changes to ADD/DROP Edge Pair commands
  - ADD edge pair in schema change will not be allowed
  - Drop vertex will be disallowed if it is currently being used in edge pair.
- Platform: *tigergraph* user id included with default installation will be allowed to be dropped
- Platform: Root user will now be disallowed to do an upgrade using installer -U option

Enhancements

Database Server

- Engine: License enforcement check improvements

- Engine: Restpp memory footprint reduction by recycling memory periodically
- GSQL: Support JSON Payload Method for Calling GSQL Built-In Dynamic Endpoints
- GSQL: Support Async query execution with query status/result functionality
- GSQL: Enhanced Interpreted Query support:
  - Support graph update for interpreted query
  - Support Where filter in PRINT statement for interpreted query
- GSQL: Logging for /requesttoken API endpoint
- GSQL: Reset function for vertex attached accumulators
- GSQL: Make token expiration maximum limit configurable
- Platform: Enterprise Free Package improvement to make pre-installed license work in both interactive and non-interactive modes
- Platform: Allow users to set hard coded timeout for Backup jobs
- Platform: Allow configurable minimum and maximum memory limits for Kafka, Kafka Connect and Kafka Stream
- Platform: Software upgrades for the following packages:
  - etcd, Kafka plugins, Jsoncpp library

GraphStudio

- Add new application server framework to offer continuous availability in GraphStudio and Admin Portal
- Update APIs for the new application server
- Support solution export/import without graph metadata
- Integrate GraphStudio with the new application server
- Increase unit test timeout

Admin Portal

- Add log management for viewing, searching and downloading
- Add configuration management settings
- Add Restpp setting: Default query timeout

- Add Nginx setting: SSL setting and whitelist IP setting
- Add application server setting: Query return size
- Add security management settings: LDAP, SSO
- Integrate Admin Portal with the new application server
- Change SSO authorization request URL
- Handle SAML ACS for SSO
- Disable authorization check for SSO metadata

Fixed

Database Server:

- Engine: Correct HTTP response code will be returned when query times out
- Engine: GPE status reporting is delayed due to backlog of large number of Kafka messages in the queue.
- Engine: GPE crash in Sub-query print statement
- Engine: Infinite loop in refresh index when some attributes are disabled
- Engine: RESTPP memory consumption increase caused by timed out queries
- Engine: Query using index will not fully utilize compute resources.
- Engine: When query times out, JSON may not be well formed
- Engine: Failed to post data when id is int and primary_id_as_attribute is true
- Engine: Avoid converting string compress index hint in remote topology edge action
- Engine: GPE not responding to SIGTERM
- GSQL: Refactor memory usage in query installation to reduce the memory footprint when there is a large number of queries
- GSQL: When creating the edge pairs, allow use of new vertex types that will be added from the current schema change job
- Platform: Backup/Restore fails to backup GUI related data
- Platform: Installer will print progress message during package install to avoid ssh timeout

TigerGraph 3.0.6

Release Date: 2020-11-11

Enhancements

Database Server

- Audit Logging Enhancements
  - User information for all requests.
  - Request Status (request succeeded or failed) for all requests irrespective of access mode
- Remove Hard timeout limit for Backup/Restore operations

Fixed

Database Server

- Platform: Resolve the issues where Kafka start-up will hang in certain OS and shell environment.
- Platform: Backup/Restore hangs if there are too many files
- Platform: Backup/Restore list error when backup files on S3 are corrupted
- Engine: Builtin query running background blocks schema change
- GSQL: Fix for SSL certificate exception

TigerGraph 3.0.5

Release Date: 2020-09-05

Features

New features and described in [3.0.5 Release notes](#).

Enhancements

Database Server

- Longer timeout for retrieving enum maps when using STRING COMPRESS
- Socket timeout adjustment to improve RESTPP stability
- Implement SetAccum<vertex> as bitset
- Semantic check for println of File object for compiled query
- Installer improvements
-
    - Enhancement to change the user and group separately.
    - Check permission of parent dir of App/Temp/Data/Log Roots
- TigerGraph 2.x to 3.x Migration tool enhancements
    - Support for copying UDFs and other functions during migration
- Enhanced license support for Cloud deployments
- Enhanced upgrade version checking
- Zookeeper client connection retry mechanism to avoid Zookeeper operation failures

Changes

Installer Configuration JSON format

- Install Configuration is separated into basic configuration and advanced configuration sections
- Support for allowing replication factor to be set during installation as opposed to limited HA on/off setting previously

Fixed

Database Server

- Core: GPE down during Backup for large number of files
- Core: GPE will crash if the data comes from a machine without relevant metadata.
- Core: Query failure due to string overflow
- Core: Query with large UDF job didn't stop for configured time out setting
- Platform: Kafka loading bug when number of loaders exceeds 10
- Platform: Backup hangs when there are very large number of files in Graph Store
- Platform: Backup reports successful operation even if it's actually incomplete
- Platform: gadmin reset does not reset all files
- GSQL: V2 syntax removes edge type that is excluded by Accum clause.
- GSQL: Force query install should regenerate the endpoints
- GSQL: Loading Job failed with SSL enabled
- GSQL: Query installation performance issue for V2 syntax
- GSQL: ArrayAccum value is not accessible in the ACCUM block when query is installed in distributed mode.
- GSQL: Dictionary Fails when Tokens are too many
- GSQL: Query installation fails due to schema change
- GSQL: gsql_client strips out newlines when writing gsql queries by pasting into gsql shell

GraphStudio

- Apply previous visualization result should handle empty saved schema
- Displaying attribute for raw type in visualization should not use JSON stringify
- Remove clear text user password in error log for migration from RDBMS to Graph

TigerGraph 3.0

Release Date: 2020-06-30

Features

New and modified features and described in the [TigerGraph 3.0 Release Notes ↗](TigerGraph 3.0 Release Notes).

Enhancements

Database Server

- Support for reload libudf command
- Schema validation before apply settings
- Relax Developer Edition restrictions
- YAML parsing support for edge pairs
- Support SPLIT for UDT loading, Load From/To Type from File
- Data generator 2.0
- Change log level by SIGUSER1, avoid unnecessary error log
- Restpp self-report status
- Allow users to remove data for reinstallation
- Upgrade kafka to 2.3.0
- Path pattern optimization with pattern flipping and PER clause
- Combine service status and processState into one log event
- Support validation of entry value during gadmin config set command
- Add strong check for symlinks
- Support to_datetime builtin function in expressions
- Support string set filter for edge and target vertex
- Support local vertex and edge with same name in multiple graphs
- Index hint for interpret mode
- Support string compress attributes in built-in Query filters
- Enable jemalloc profiling

- Utility function to get disk free percentage
- Allow concurrent user query access during Query Installation
-

GraphStudio:

- Support multiple-pair edge type
- Schema change job for add/drop attribute index
- Improved clear graph warning
- New layout for logo and multiple graphs
- Allow user edit header for sample data
- Support multiple files upload
- Cancel autofit for adding vertex and double click actions
- Cancel auto login if user has logged out
- Save JSON format of query result to local storage
- Create Edge Type from Multiple Vertex Types to Multiple Vertex Types

Fixed

Database Server

- Add on-demand heap profiling for jemalloc
- Delete legacy ids data
- Periodically force Jemalloc release memory to OS / on demand profiling
- Change debug log in convertids into verbose
- Print warning but no assert in ZMQ
- Wrong JSON format for tempTables
- Fix wrong check for loading job completion
- Allow interpret query to recognize html encoded string constant
- Handle logical type in json converter
- Corrected URL decode for whitespace character

- Add time before delete edges command to ensure rebuild has enough time to complete
- Fix remove session bug for the aborted handler after 'ctrl + c'
- Synchronize concurrent install queries
- Change logic to check service status for cluster mode
- Support the ‘=‘ operator SumAccum;
- Drop vertex/edge/graph when there are local and global vertex/edge have the same name;
- Support removing a SetAccum from another SetAccum;
- Remove the reversed edge too when removing an edge;
- Cannot create query due to the overflow of the size of the HeapAccum;
- Query referred as subquery from interpreted mode query can not be dropped;
- Index out of bound when ignoring the parameter checking for interpret query
- Output error message for invalid job id
- Fix codegen to insert a vertex/edge without attributes
- Support file regexp in checking header of filename
- Support the true value of key word header and transaction in the loading data job to be case-insensitive
- Dedupe proxy user's own roles from groups
- Make schema change metadata modification a transaction
- Fix builtin k_step expansion query bug
- Check disk space before exporting each vertex/edge type
- Allowed non-English string constants in interpreted queries
- Edge variable prints attribute by default
- Print developer information only in gadmin status
- Restrict symlinks and check their existence

GraphStudio:

- Fix error message for new secret creation
- Refactor keywords

- Do not emit explorer config if saved exploration doesn't have it
- Check for Valid date time
- Extend wait time for progress bar finish
- Add right border for side navigation
- Upgrade color-picker
- Fix check accumulator format
- Fix percentage of performing schema change
- Run interpreted query through websocket

TigerGraph 2.6.6

Release Date: 2021-03-23

Fixed

Database Server

- Core: Fix concurrent access of abort messages
- Core: Fix for GPE crash due to wrong license
- Core: Fixes to gcollect utility:
-
    - Improvements to work in clustered environments
    - Accidental removal of directory with old data collection run
- GSQL: Fix for catalog access issue due to concurrent schema change requests
- GSQL: Increase timeout for download upload catalog, make it configurable
- Platform: Upgrade of gRPC version to 1.33.0
- Platform: Remove user authentication information after installation

TigerGraph 2.6.5

Release Date: 2021-01-15

Enhancements:

Database Server

- GSE/GPE segment consistency check utility
- Integration with GSE/GPE consistency check utility with Backup/Restore

Changes

- Increase in refresh timeout for RESTPP from 20 to 60 seconds;

Fixed

Database Server

- GSE replica synchronization for Zookeeper errors
- Explicitly check replica follower status before automatic promotion to leader is allowed
- RESTPP fix - memory leaks caused by timed out queries
- Backup/Restore: Ensure GPE and GSE snapshots are done in correct order

TigerGraph 2.6.4

Release Date: 2020-11-02

Enhancements

Database Server

- Allow RESTPP to manage log files based on timestamp

- Upgrade NGINX to 1.18 version
- Correct status code to indicate GSQL operation result
- Remove Hard timeout limit for Backup/Restore operations
- Token Management Improvements:
  - Improve GSQL stability by setting a limit on number of tokens allowed
  - Logging improvement to indicate new and refreshed tokens separately

Fixed

Database Server

- Core: GSE follower replicas lag leader replica on the data updates
- Core: Shuffle abort causing GPE crash
- Core: Handle un-released lock gracefully during json print command failure
- Core: Incremental Snapshot triggers creation of all segments causing delays
- Core: Kafka loading fails when number of loaders exceed 10
- GSQL: Query Install fails for batch installs
- Backup/Restore hangs if there are too many files

TigerGraph 2.6.3

Release Date 2020-08-21

Enhancements

- Improved handling of query time outs for distributed queries.

Fixed

- Longer timeout for retrieving large memory map for attributes of STRING COMPRESS data type with large number of distinct values.

- Backup jobs report incorrect successful runs
- Incorrect type check logic for trim function;

TigerGraph 2.6.2

Release Date 2020-08-14

Enhancements

- Improvements to GSE Upsert performance
- Add User Id information to RESTPP logs for all user initiated calls
- Improvements to Query Installation performance time
- Provide warning message when revoking a role from proxy user if needed

Fixed

- Core: GPE crash on unknown vertex / segment
- Core: PostWriter needs to skip vertices if the internal vertex id is invalid one.
- Core: Handle exception in ResponseThread of RemoteTopology
- Core: Query re-installation issue caused by non-deterministic transformation
- Core: Address Data Loading speed for hub loading
- Core: Inconsistent result with and without using local accumulators
- Core: RestPP payload scale issue due to 3rd party FCGI library
- GSQL: GSQL pattern match - translation error when vertex type is the keyword "ANY"
- GSQL: Issue with reduce function with Bitwise OR operator in the LOAD functions
- GSQL: *gsql_client* strips out newlines when writing gsql queries by pasting into gsql shell
- GSQL: Secrets and token associated with a graph and not removed during graph delete

- GraphStudio: Displaying attribute for raw type in visualization should not use JSON *stringify* method

TigerGraph 2.6.1

Release Date 2020-06-12

Enhancements

- Allow concurrent user query access during Query Installation
- GPE & GSE Data Sync Check Utility
- Use of POST for /requesttoken API so that user password is not exposed
- Write Performance improvements
- Error handling and reporting improvements for Query Timeout and Failures
- UX improvement for ‘Clear Graph’ command in GraphStudio

Fixed

- Ensure cleanup and compaction of delta records in a large transaction even in the event of TigerGraph service restart
- Performance improvement to make Graph Updates faster by parallelizing and sharing transaction
- Fix for the leftover Shuffle threads after Query Abort/Timeout
- Change in the error message of AbortQuery request inside the Shuffle Operator
- Bug fixes for GSE compaction feature to address exporting with mixed segments of data and load data from the database in worker mode
- Fix for GSE crash triggered by schema change
- Enable background thread on JEMALLOC for memory cleanup even when system is idle
- /showprocesslist and /abortquery APIs do not list the running queries of old worker if RESTPP is refreshed

- S3 loader header check doesn't apply file filter regex
- GSQL V2 syntax does not handle ACCUM operator correctly
- Fix for RESTPP timeout error

TigerGraph 2.6.0

Changes

Release Date 2020-04-24

New and modified features and described in the TigerGraph 2.6 Release Notes. ↗

Enhancements

- Remove SSH connection use dependency for GSQL Install Query command
- New 'force' parameter to RebuildNow so that engine to start the rebuild.

Fixed

- Core: GSE crash in HA setup when CPU usage is extremely high
- Core: Out Of Memory handling improvements to prevent GPE crash due bad memory allocation call
- GLE: fix builtin query crash in worker due to graph id missing
- Core: Skewed CPU usage for high-query throughput scenarios
- Fixes in Rebuild to address broken edge count
- Fix for 2.5.2 bug - Inconsistent query results when running non-distributed query on a cluster
- Unable to find local vertex and edge with same name in multiple graphs
- RESTPP memory leak due to yaml file
- Reverse edge id is wrong when two local edges with reverse edge are created with same name

TigerGraph 2.5.4

Release Date: 2020-04-24

Enhancements

- New 'force' parameter to RebuildNow so that engine to start the rebuild.
- Improved version of /abortquery so that query can be aborted more quickly

Fixed

- Fixes in Rebuild to address broken edge count
- RESTPP memory leak due to yaml file
- Builtin query crashed due to missing Graph Id
- RESTPP crash for same vertex name in the global graph
- Resolved the distributed query hanging issue which could block rebuild and schema change
- Core: Skewed CPU usage for high-query throughput scenarios

TigerGraph 2.5.3

Release Date: 2020-02-26

Fixed

- Ensure catalog data backed up before schema change
- Support creation of two local edges with same name with one being a reverse edge
- Support Local vertex and edge type with same name in multiple graphs in

- Support for multi-lingual string constant in Interpret query mode
- Upgrade to Release 2.5.2 leads to inconsistent query results
- Compute resource usage spikes on particular node in cluster
- GCleanUp failed to cleanup all pointers when adjusting thread

TigerGraph 2.5.2

Release Date: 2020-01-27

> **TigerGraph 2.5.2 is not compatible with versions prior to 2.5.1. Customers who are using Pre-2.5.1 version and intending to migrate to 2.5.2 are advised to take backup of their existing version before upgrading to 2.5.2. This will enable them to downgrade back to the original Pre-2.5.1 version if nee**

New Features

- GPE: Increase MemoryCheck frequency based on Resource Usage
- GPE: Abort Query if Memory usage crosses critical threshold
- GSE: Support Log compaction as part of startup for GSE
- GraphStudio: Support Multi-edge pair in design schema.
- Core: Support OS RHEL 8.0 in Installer

Enhancements

- REST: Increase the RESTPP reload timeout
- GSQL: Change error message to specify user when default *tigergraph* user is dropped
- GSQL: Make user *tigergraph* droppable
- GraphStudio: Do not change layout when adding/updating/deleting vertex and edge

Fixed

- Core: GPE crashed running distributed LDBC query
- GST: Incorrect vertex count in TigerGraph GraphStudio
- Core: Shuffle deadlock causing full system memory use
- Core: Replace GASSERT with GWARN in GDataBox
- Core: BATCH_SIZE of Kafka loader set from GSQL console doesn't work
- GPE: Schema Change failed due to Query Install OOM
- GSQL: Quote in string key is not escaped
- GraphStudio: Reverse edge filter doesn't work
- Core: Don't display LDAP password in IUM

TigerGraph 2.5.1

Release Date: 2019-11-25

Fixed

- Core: Distributed delete affects data consistency after GPE restart
- Core: Shuffle hangs when sendingQueue is full
- Core: Longevity test failing due to change in memory allocator (TCMalloc)
- GPE: Crash after upgrade from 2.4.1 to 2.5
- GPE: Serialization error when reading from input stream
- GPE: Query state can result in race condition inside ReadOneDelta;
- GPE: GPE crashes when a query calls a sub-query with a write operation
- GSE: Script to resolve delete inconsistency between GSE and GPE
- GSE: Multiple Kafka loading jobs fail
- GSQL: Built-in function names in GSQL are case sensitive
- GSQL: Interpret query doesn't work when authentication is on

- GSQL: Deadlock when graph store is being cleared and authentication is on
- GSQL: Token authentication returning null during Global schema change
- GSQL: SSO login failure due to missing org.apache.santuario:xmlsec library
- GraphStudio: Vertex to edge expansion settings are not retained
- GBAR Backup: Backup failure if loading jobs are in progress

TigerGraph 2.5.0

Release Date 2019-09-18

Changes

New and modified features and described in the TigerGraph 2.5 Release Notes ↗.

Fixed

- Improvements to fix possible crash, deadlock, overflow, and memory leak situations
- Improve query performance stability
- Fix some query string passing and parsing issues
- Correct some inconsistencies between the documented specification and actual behavior
- Improve robustness of Kakfa and S3 Loaders
- Clean up files and graph properly after certain failed operations
- Fix some installation issues

TigerGraph 2.4.1

Release Date 2019-07-23

Changes

- To select pattern matching support in a query, the syntax is now
  `CREATE QUERY ... SYNTAX v2`
  instead of
  `CREATE QUERY ... SYNTAX("v2")`

Fixed

- GPE: Fix uint32 overflow
- Loader: Allow temp_table to be used without flatten function
- IDS: Disable empty UID
- ZMQ: Fix crash on ill-formed message
- Util: Fix Unix domain socket file not generated correctly in cron job
- Util: Extend data size for GoutputStreamBuffer beyond 4GB
- Connector: Fix first line is not ignored with has_header enabled
- Connector: Fix failures on retrieving connector status
- GSQL: Fix syntax version setting inconsistency issues
- GSQL: Fix schema change with USING primary_id_as_attribute
- GSQL: Fix JSON output format of requesttoken API
- Admin Portal: Display correct counts of physical vertices and edges on each machine

TigerGraph 2.4.0

Release Date 2019-06-25

New Features

See Release Notes - TigerGraph 2.4 ↗

Fixed

- GSQL: The built-in count() function gives the correct value in all cases.
- GPE: startup hang
- GSQL server start/stop command not working
- LDAP config truncated by space
- GSE: boolean values are not displayed correctly
- Security issue CVE-2013-7459 caused by unused python crypto library
- IUM status is displayed incorrectly in some cases;

TigerGraph 2.3.2

Release Date 2019-04-01

Issues

- GSQL: The built-in count() function may give the incorrect value for clustered systems after some vertices have been deleted.

Fixed

- GraphStudio: Send query pre-install dependency analysis result through WebSocket
- GraphStudio: filter out improper attributes in when building filter expressions
- GPE: fix wrong enumerator id issue
- GPE: avoid using /tmp
- GPE: handle exceptions for LIKE <expr>
- GPE: Fix crash due to writing wrong size of STRING_LIST
- GPE: Fix global schema change error which added local vertex twice
- GSE (Developer Edition): Keep one copy of segment

TigerGraph 2.3.1

Release Date 2019-02-19

New Features

See Release Notes-TigerGraph 2.3 ↗

Issues

- GSQL: The built-in count() function may give the incorrect value for clustered systems after some vertices have been deleted.

Fixed

- Install: The IP list fetched by the installer could be incomplete.
- Loading: Speed up batch-delta loading.
- GraphStudio: Disable Install Query button for queryreader users.
- GraphStudio: Re-initialize the database after import.
- GraphStudio: Could not drop query with non-default username/password.
- AdminPortal: Queries-Per-Second display didn't work if RESTPP authorization was enabled.
- Schema change: Improve schema change stability by reducing schema change history and increasing gRPC max message limit.
- GPE: Improve query HA stability.
- GPE: Fix crash under certain conditions.
- Core: Memory leak due to yamlcpp.
- Core: compatibility issue between libc and ssh utility.
- IUM: Fix exceptions due to legacy config entries.

TigerGraph 2.2.4

Release Date: 2018-12-13

Fixed

- Distributed System: Fix possible deadlock and race conditions
- GSE Storage Engine: Fix disk seek overflow
- RESTPP: Optimize the memory consumption when system is idle
- RESTPP: Optimize config reload time
- GSQL: Fix query installation error with option -optimize
- GSQL: Fix a code generation bug related to static variable
- GSQL: Fix a compilation error when a statement is in nested if statement
- GraphStudio: Security update for npm-run-all
- GraphStudio: Change Help button to point to new docs.tigergraph.com site
- Gadmin: Fix gadmin/ts3 restart and status error after changing port of TS3

TigerGraph 2.2.3

Release Date: 2018-11-30

Fixed

- GraphStudio: Fix schema change bug (Note: In 2.2, GraphStudio now does not drop all data when making a schema change.)
- GraphStudio: Fix display issue in Graph Explore when switch to a new graph
- GraphStudio: Improve password security
- GraphStudio: Modify URL to AdminPortal for better universal support
- IUM: Fix kafka-loader configuration after cluster expansion

- IUM: Resolve python module name conflict
- IUM: Fix ssh_port is always 1 under bash interactive mode
- GSE Storage Engine: Reduce memory consumption
- RESTPP: Improve logging messages

TigerGraph 2.2

Release Date: 2018-11-05

New Features

See Release Notes-TigerGraph 2.2 ↗

Fixed

- GraphStudio: When both a query draft and an installed query exist, Export Solution will keep the installed query code instead of the query draft
- Admin Portal: Number of nodes in the cluster is reported as 0 when no graph yet exists

TigerGraph 2.1.8

Release Date: 2018-11-05

Issues

- GBAR Backup fails if HA is enabled
- GSE status shows unknown with HA enabled
- TS3 fails to collect QPS when RESTPP Authentication is enabled (Admin Portal QPS monitor will be unavailable in this case).

- GraphStudio: When both a query draft and an installed query exist, Export Solution will keep the installed query code instead of the query draft.
- Admin Portal: Number of cluster nodes is reported as 0 when no graph exists.

Fixed

- GSQL server error if schema is too large
- In a cluster, not all servers may be aware of deleted vertices.
- PAM limit set-up issue in installer
- In MultiGraph, a local (FROM *, TO *) local edge has global side effects.
- RESTPP's default API version is not set after installation
- An engine bug which occasionally causes crash

Added

- SSH port configuration in installer.
- Installation script checks that the machine meets the minimum RAM (8GB) and CPU (2-core) requirements.
- For Ubuntu 16.04/18.04, support logon with systemd service.

TigerGraph 2.1.7

Release Date: 2018-08-20

Issues

- GBAR backup fails if HA is enabled.
- TS3 fails to collect QPS when RESTPP Authentication is enabled (Admin Portal QPS monitor will be unavailable in this case).
- GraphStudio: When both a query draft and an installed query exist, Export Solution will keep the installed query code instead of the query draft.

- Admin Portal: Number of cluster nodes is reported as 0 when no graph exists.

Fixed

- Cluster configuration with HA enabled is wrong if the number of nodes is odd (3, 5, 7, 9...).
- GraphStudio and GSQL inconsistent checking for some keywords
- GBAR backup and restore fail if special character is in tag name

TigerGraph 2.1.6

Release Date: 2018-08-15

Issues

- Cluster configuration with HA enabled is wrong if the number of nodes is odd (3, 5, 7, 9...).
- GraphStudio: When both a query draft and an installed query exist, Export Solution will keep the installed query code instead of the query draft.
- TS3 fails to collect QPS when RESTPP Authentication is enabled (Admin Portal QPS monitor will be unavailable in this case).
- Admin Portal: Number of cluster nodes is reported as 0 when no graph exists.

Fixed

- GSQL null pointer exception during schema change if a directed edge is dropped but its partner reverse edge is kept.
- Some complex attribute types cannot be correctly posted via /graph endpoint.
- In some cases, tuple on reverse edge crashes GPE.
- GraphStudio throws an authentication error if RESTPP authentication is enabled.

Added

- License level control of MultiGraph functionality.

Tigergraph 2.1.5

Release Date: 2018-07-24

Known Issues

- GSQL null pointer exception during schema change if a directed edge is dropped but its partner reverse edge is kept.
- Some complex attribute types cannot be correctly posted via /graph endpoint.
- In some cases, tuple on reverse edge crashes GPE.

Fixed

- GraphStudio Export package is occasionally incomplete.
- GSE status is always "not ready" if schema is too large.
- Cannot modify RESTPP port configuration.
- IUM error in a cluster when not running on node m1

V3.0 Removal of Previously Deprecated Features

TigerGraph 2.x contained some features which were labeled as deprecated. These features are no longer necessary because they have been superseded already by improved approaches for using the TigerGraph platform. The new approaches were developed because they use more consistent grammar, are more extensible, or offer higher performance. Therefore, TigerGraph 3.0 has streamlined the product by removing support for some of these deprecated features, listed below:

Data Types

See [Data Types in GSQL Language Reference]

Deprecated Type	Alternate Approach
REAL	Use FLOAT or DOUBLE
INT_SET	Use SET<INT>
INT_LIST	Use LIST<INT>
STRING_SET_COMPRESS	Use SET<STRING COMPRESS>
STRING_LIST_CONPRESS	Use LIST<STRING COMPRESS>
UINT_SET	Use SET<INT>
UINT32_UINT32_KV_LIST	Use MAP<UINT, UINT>
INT32_INT32_KV_LIST	Use MAP<INT, INT>
UINT32_UDT_KV_LIST	Use MAP<UINT, UDT_type>, where UDT_type is a user-defined tuple type
INT32_UDT_KV_LIST	Use MAP<INT, UDT_type>, where UDT_type is a user-defined tuple type

Syntax for Control Flow Statements

See Control Flow Statements in GSQL Language Reference

Deprecated Statement	Alternate Statement
FOREACH ... DO ... DONE	FOREACH... DO... END

FOREACH (condition) {
body
}
	FOREACH condition DO
	body
	END

IF (condition) {
body1
} else {
body2
}
	IF condition THEN
	body1
	ELSE
	body2
	END

WHILE (condition) {
body
}
	WHILE condition DO
	body
	END

Vertex Set Variable Declaration

See Vertex Set Variable Declaration and Assignment

If a vertex type is specified, the vertex type must be within parentheses.

Deprecated Statement	Alternate Statement
MySet Person = ...	MySet (Person) = ...

Query, Job, and Token Management

These are documented in several places throughout the GSQL Language Reference:

- CREATE QUERY Statement

- Creating a Loading Job and Running a Loading Job
- RUN SCHEMA_CHANGE JOB, and RUN GLOBAL SCHEMA_CHANGE JOB
- CREATE / SHOW / DROP / REFRESH Token
- offline2online in 'Creating a Loading Job'

Deprecated Operation	Alternate Operation
CREATE JOB [loading job definition]	CREATE LOADING JOB [loading job definition]
RUN JOB [for loading and schema change jobs]	Specify the job type: RUN LOADING JOB RUN SCHEMA_CHANGE JOB RUN GLOBAL SCHEMA_CHANGE JOB
CREATE / SHOW/ REFRESH TOKEN	To create a token, use the REST endpoint GET /requesttoken
offline2online	The offline loading job mode was discontinued in v2.0. Do not write loading jobs using this syntax.

Output

See PRINT Statement in 'Output Statements and File Objects'

Deprecated Syntax	Alternate Syntax
JSON API v1	v2 has been the default JSON format since TigerGraph 1.1. No alternate JSON version will be available.
PRINT ... TO_CSV [filepath]	Define a file object, then PRINT ... TO_CSV [file_object]

Built-in Queries

See Run Built-in Queries in 'GSQL 101'

Deprecated Statement	Alternate Statement

SELECT count() FROM ...
// count may be out of date
	SELECT approx_count(*) FROM ...
	// same behavior as count(); may not include all the latest data updates
	SELECT count(*) FROM ...
	// exact, but slower than approx_count(*)

Knowledge Base and FAQs

If you have a problem with the procedure described in the ***TigerGraph Platform Installation Guide***, please contact support@tigergraph.com ↗and summarize your issue in the email subject.

Getting Started and Basics

I need help installing the system.

What version of the TigerGraph platform am I running?

Use the following command:

```
$ gsql --version
```

To see the version numbers of individual components of the platform:

```
$ gadmin version
```

How do I upgrade from an earlier version?

Each release comes with documentation addressing how to perform an upgrade. Upgrade instructions are documented in Installation guide. Please contact support@tigergraph.com ↗for help in your specific situation.

I'm not sure how to run the TigerGraph system.

If you correctly installed the system and are now logged in as the TigerGraph system user, you should be able to enter the GSQL shell by typing the `gsql` command from an operating system prompt. If this command has never worked, then probably the installation was not successful. If it works but you are not sure what to do next, please see the ***GSQL Demo Examples*** guide.

The system does not seem to be running correctly.

If you believe you have installed the system correctly (e.g., you followed the ***TigerGraph Platform Installation Guide*** and received no errors, and the `gsql` and `gadmin` commands are now recognized), then please contact support@tigergraph.com and summarize your issue in the email subject.

Do I need to start the TigerGraph servers (e.g., GPE, GSE) to run the system?

Different servers are needed for different purposes, but the TigerGraph should automatically turn services on and off as needed. Please be sure that the Dictionary (dict) server is on when using the TigerGraph system:

To check the status of servers:

```
$ gadmin status
```

Does the TigerGraph system have in-tool help?

Yes. For the GSQL shell and language, first enter the shell (type `gsql` from an operating system prompt). Then type the `help` command, e.g.,

```
HELP
```

This gives you a short list of commands. Note that "help" itself is one of the listed commands; there are help options to get more details about `BASIC`, `QUERY` commands. For example,

```
HELP QUERY
```

lists the command syntax for queries. See the "System Basics" section of the ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data.*** The `gadmin` administration tool also has a help menu and a manual page:

```
$ gadmin help
```

Is the GSQL language case sensitive?

User-defined identifiers are case-sensitive. For example, the names `User` and `user` are different. The GSQL language keywords (e.g., CREATE, LOAD, VERTEX) are not case-sensitive, but in our documentation examples, we generally show keywords in ALL CAPS to make them easy to distinguish.

What are the rules for naming identifiers?

An identifier consists of letters, digits, and the underscore. Identifiers may not begin with a digit. Identifiers are case sensitive. Special naming rules apply to accumulators (see the Query section).

When are quotation marks required? Single or double quotes?

The general rule is that string literals within the GSQL language are enclosed in double quotation marks. For data that is to be imported (not yet in the GSQL data store), the GSQL loading language lets the user specify how data fields are delimited within your input files. The loading language has an option to specify whether single quotes or double quotes are used to mark strings. For more help on loading, see the "Loading Data" section of this document or of the ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data .***

Can I run GSQL Shell commands in batch command?

Yes. You can create a text file containing a sequence of GSQL commands and then execute that file. To execute from outside the shell:

```
$ gsql filename
```

To execute the command file from within the shell:

```
@filename
```

See also the "Language Basics" and "System Basics" sections of the *GSQL Language Reference, Part 1: Defining Graphs and Loading Data* document.

I have a long command line. Can I split it into multiple lines?

Yes. Normally, an end-of-line character triggers execution of a line. You can use the `BEGIN` and `END` keywords to mark off a multi-line block of text that should not be executed until `END` is encountered.

This is an example of a loading statement split into multiple lines using BEGIN and END:

```
BEGIN
CREATE ONLINE_POST JOB load1 FOR GRAPH LaborForce {
  LOAD
    TO VERTEX user VALUES ($0, _, _, _),
    TO VERTEX occupation VALUES ($0, _),
    TO EDGE user_occupation VALUES ($0, $1);
}
END
```

What is Limited Capacity Mode?

When a license limit has been reached, your system will be placed in a read-only mode - incapable of loading anymore data. You will still be able to delete data and view the graph.

Defining a Graph Schema

What are the components of a graph schema?

A TigerGraph graph schema consists of (A) one or more vertex types, (B) one or more edge types, and (C) a graph type. Each edge type is defined to be either DIRECTED or UNDIRECTED. The graph type is simply the list of vertex types and edges types which may exist in the graph. For more: See the section "Defining a Graph Schema" in the ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data*** . Below is an example of a graph schema containing two vertex types, one edge type, and one graph type:

```
CREATE VERTEX user (PRIMARY_ID user_id UINT, age UINT, gender STRING, pos
CREATE VERTEX occupation (PRIMARY_ID occ_id STRING, occ_name STRING)
CREATE UNDIRECTED EDGE user_occupation (FROM user, TO occupation)
CREATE GRAPH LaborForce (user, occupation, user_occupation)
```

Alternately, a generic CREATE GRAPH statement can be used:

```
CREATE GRAPH LaborForce (*)
```

Should I model this data field as an attribute or as a vertex type?

Property graphs can model data fields ("properties") as either a property of a vertex or edge or as a vertex linked to other vertices. If your property relates to an edge, it should be an attribute of that edge (for example, a Date attribute of a CustomerBoughtProduct edge). If your property relates to a vertex, you have a choice. The optimal choice depends on how you will typically use this attribute in your application. If you will frequently search or filter based on that data, we suggest your treat it as a separate vertex type. Otherwise, we recommend modeling this data as an attribute of the principal vertex.

What data types do you support for vertex and edge attributes?

Each attribute of a vertex or edge has an assigned data type. v0.8 of the TigerGraph adds support for many more attribute types.: DATETIME, UDT, and container types LIST, SET, and MAP. The following is an abbreviated list. For a complete list and description, see the section "Attribute Data Types" of the ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data .***

Primitive Types	Advanced Types	Complex Types
INT		
UINT		User-Defined Tuple (UDT)
FLOAT	STRING COMPRESS	LIST
DOUBLE	DATETIME	SET
BOOL		MAP
STRING		

Discontinued Feature

The UINT_SET and STRING_SET COMPRESS types have been discontinued since there is now equivalent functionality from the more general SET and SET types.

Can I define and load multiple graph schemas?

The TigerGraph MultiGraph service, an add-on option, supports logical partitions of one unified global graph. Each partition is treated as an independent local graph, with its own set of user privileges. Local graphs can overlap, to create a shared data space.

How many vertex and edge types can I include in a graph?

For performance reasons, we recommend to keep the number of different vertex and edge types under 5,000. The upper limit for the number of different vertex and edge types is approximately 10,000, depending on the complexity of the types.

How do I check the definition of the current schema?

From within the GSQL Shell, the `ls` command lists the *catalog* : the vertex type, edge type, and graph type definitions, job definitions, query definitions, and some system configuration settings. If you have not set your active graph, then `ls` will show only item which have global scope. To see graph-specific items (including loading jobs and queries), you must define an active graph.

How do I modify my graph schema?

The GSQL language includes ADD, ALTER, and DROP commands. See the section "Update Your Data" in the ***GSQL Demo Examples*** or the section "Modifying a Graph Schema" in the ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data*** for details. Note that altering the graph schema will invalidate your old data loading and query jobs. You should create and install new loading and query jobs.

How do I delete my entire graph schema?

To delete your entire catalog, containing not just your vertex, edge, and graph type definitions, but also your loading job and query definitions, use the following command:

`GSQL>` **`DROP ALL`**

To delete just your graph schema, use the DROP GRAPH command:

`GSQL>` **`DROP GRAPH g1`**

UPDATE Deleting the graph schema also erases the contents of the graph store. To erase the graph store without deleting the graph schema, use the following command:

```
GSQL> CLEAR GRAPH STORE
```

See also " **How do I erase all data?** "

Loading Data

How do I load data?

To load structured data stored in files, you write a loading job and then execute it. See ***GSQL 101*** and the ***GSQL Demo Examples*** for introductory examples. Loading jobs can include instructions for parsing and processing the data, in order to perform many ETL tasks. See Creating a Loading Job for the complete specifications. To load streaming data or data coming from other data stores, see Data Loader User Guides.

In v2.0, the TigerGraph introduced a more powerful and comprehensive syntax which has several advantages:

- The TigerGraph platform can handle concurrent loading jobs, which can greatly increase throughput.
- The data file locations can be specified at compile time or at run time. Run-time settings override compile-time settings.
- A loading job definition can include several input files. When running the job, the user can choose to run only part of the job by specifying only some of the input files.
- Loading jobs can be monitored, aborted, and restarted.

What types of data can be read?

The GSQL data loader reads text files organized in tabular or JSON format . Each field may represent numeric, boolean, string, or binary data. Each data field may contain a single value or a list of values (see *How do I split a data field containing a list of values into separate vertices and edges?*).

Additional data formats are continually being added. See Data Loader User Guides and the TigerGraph Ecosystem Github Repository's etl folder https://github.com/tigergraph/ecosys/tree/master/tools/etl ↗

What is the format of a tabular input data file?

Each tabular input data file should be structured as a table, in which each line represents a row, and each row is a sequence of data fields, or columns. A data field can contain string or numeric data. To represent boolean values, 0 or 1 is expected. A header line may be included, to associate a name with each column. A designated character separates columns. For example, if the designated separator character is the comma, this format is commonly called CSV, for Comma-Separated Values. Below is an example of a CSV file with a header. The `uid` column is int type, `name` is string type, `avg_score` is float type, and `is_member` is boolean type. See simple examples in *Real-Life Data Loading and Querying Examples* and a complete specification in the section "Creating a Loading Job" in ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data .***

```
uid,name,avg_score,is_member
100,"Lee, Tom",48.5,1
101,"Wu, Ming",33.9,0
102,"Gables, Anne", 72.2,1
```

The loader does not filter out extra white space (spaces or tabs). The user should filter out extra white space from the files before loading into the TigerGraph system.

How should data fields be separated?

The data field (or *token*) separator can be any single ASCII character, including one of the non-printing characters. The separator is specified with the SEPARATOR

phrase in the USING clause. For example, to specify the semicolon as the separator:
`USING SEPARATOR=";"`

To specify the tab character, use `\t`. To specify any ASCII character, use `\nn` where `nn` is the character's ASCII code, in decimal. For example, to specify ASCII 30, the Record Separator (RS):
`USING SEPARATOR="\30"`

Should fields be enclosed in quotation marks?

TigerGraph does not require fields to be enclosed in quotation marks, but is it recommended for string fields. If the QUOTE option is enabled, and if the loader finds a pair of quotation marks, then the loader treats the text within the quotation marks as one value, regardless of any separation characters that may occur in the value. The user must specify whether strings are marked by single quotation marks or double quotation marks.
`USING QUOTE="single"`
or
`USING QUOTE="double"`

For example, if `SEPARATOR=","` and `QUOTE="double"` are set, then when the following data are read,

```
uid,name,avg_score,is_member
100,"Lee, Tom",48.5,1
101,"Wu, Ming",33.9,0
102,"Gables, Anne,"72.2,1
```

"Lee, Tom" will be read as a single field. The comma between Lee and Tom will not separate the field.

Does the GSQL Loader automatically interpret quotation marks as enclosing strings?

No. You must specify either `QUOTE="single"` or `QUOTE="double"`.

What are the parameters (in the USING clause) for a loading job?

The following three parameters should be considered for every loading job from a tabular input file:

Parameter	Meaning of value	Allowed values	Comments
SEPARATOR	specifies the special character that separates tokens (columns) in the data file	any single ASCII character	Required. `"\t"` for tab `"\nn"` for ASCII decimal code nn
HEADER	whether the data file's first line is a header line which assigns names to the columns. In offline loading, the Loader reads the header line to obtain mnemonic names for the columns. In online loading, the Loader just skips the header line.	"true", "false"	Default = "false"
QUOTE	specifies whether strings are enclosed in single quotation marks: `'a string'` or double quotation marks: `"a string"`	`"single"`, `"double"`	Optional; no default value.

The next two parameters, FILENAME and EOL are required if the job is an ONLINE_POST job:

Parameter	Meaning of value	Allowed values	Comments
FILENAME	name of input data file	any valid path to a data file	Required for online loading. Not allowed for offloading loading
EOL	the end-of-line character	any ASCII sequence	Default = `"\n"` (system-defined newline character or character sequence)

All of the these five parameters are combined into one USING clause with a list of parameter/value pairs. The parameters may appear in any order.

```
USING parameter1="value1", parameter2="value2",... , parameterN="valueN"
```

The location of the USING clause depends on whether the job is an offline loading job or an online loading job. For offline loading, the USING clause appears at the end of the LOAD statement. For example:

```
CREATE LOADING JOB load1 FOR GRAPH LaborForce{
  LOAD "jobs.csv" TO VERTEX occupation VALUES ($0, $1) USING HEADER="true
}
```

For online loading, the USING clause appears at the end of the RUN statement

```
CREATE ONLINE_POST JOB load2 FOR GRAPH LaborForce{
  LOAD TO VERTEX occupation VALUES ($0, $1);
}
RUN JOB load2 USING FILENAME="./jobs.csv", HEADER="true", SEPARATOR="|",
```

My data file doesn't have a header but I still want to name the columns.

You can define a header line (a sequence of column names) within a loading job using a DEFINE HEADER statement, such as the following:

```
DEFINE HEADER head1 = "jobId", "jobName";
```

This statement must appear before the LOAD statement that wishes to use the header definition. Then, the LOAD statement must set the USER_DEFINED_HEADER parameter in the USING clause. A brief example is shown below:

```
CREATE ONLINE_POST JOB load2 FOR GRAPH LaborForce{
  DEFINE HEADER head1 = "jobId", "jobName";
  LOAD TO VERTEX occupation VALUES ($"jobId", $"jobName") USING USER_DEFI
}
```

How do I identify and refer to the input data fields?

Input data fields can always be referenced by position. They can also be referenced by name, if a header has been defined.

-
    - Position-based reference: The leftmost field is `$0`, the next one is `$1`, and so on.
    - Name-based reference: `$"name"`, where *`name`* is one of the header column names.

For example, if the header is
`abc,def,ghi`

then the third field can be referred to as either `$2` or `$"ghi"`.

How do I split (flatten) a data field containing a list of values into separate vertices and edges?

First, to clarify the task, consider a graph schema with two vertex types, Book and Genre, and one edge type, book_genre:

create_book_schema.gsql

```
CREATE VERTEX Book  (PRIMARY_ID bookcode STRING, title STRING)
CREATE VERTEX Genre (PRIMARY_ID genre_id STRING, genre_name STRING)
CREATE UNDIRECTED EDGE book_genre (FROM Book, TO Genre)
CREATE GRAPH book_rating (Book, Genre, book_genre)
```

Further, each row of the input data file contains three fields: bookcode , title , and genres , where genres is a list of strings associated with the book. For example, the first few lines of the data file could be the following:

book.dat

```
bookcode|title|genres
101|"Harry Potter and the Philosopher's Stone"|fiction,fantasy,young
102|"The Three-Body Problem"|fiction,science fiction,Chinese
```

The data line for bookcode 101 should generate one Book instance ("Harry Potter and the Philosopher's Stone"), four Genre instances ("fiction", "adventure", "fantasy", "young adult"), and four Book_Genre instances, connecting the Book instance to each of the Genre instances. This process of creating multiple instances from a list field (e.g., the genres field) is called flattening .

To flatten the data, we use a two-step load. The first LOAD statement uses the flatten() function to split the multi-value field and stores the results in a TEMP_TABLE. The second LOAD statement takes the TEMP_TABLE contents and writes them to the final edge type.

load_books.gsql

```
CREATE ONLINE_POST JOB load_books FOR GRAPH book_rating {
  LOAD
      TO VERTEX Book VALUES ($0, $1),
      TO TEMP_TABLE t1(bookcode,genre) VALUES ($0, flatten($2,",",1))

  LOAD TEMP_TABLE t1
      TO VERTEX Genre VALUES($"genre", $"genre"),
      TO EDGE book_genre VALUES($"bookcode", $"genre");
}
RUN JOB load_books USING FILENAME="book.dat", SEPARATOR="|", HEADER="
```

The flatten function has three arguments: (field_to_split, separator, number_of_parts_in_one_field). In this example, we want to split $2 (genres), the separator is the comma, and each field has only 1 part. So, the flatten function is called with the following arguments: `flatten($2, ",",1)`. Using the example of data file , TEMP_TABLE t1 will then contain the following:

bookcode	genre
101	fiction
101	adventure
101	fantasy
101	young adult
102	fiction
102	science fiction
102	Chinese

The second LOAD statement uses the TEMP_TABLE t1 to generates Genre vertex instances and book_genre_instances. While there are 7 rows shown in the sample TEMP_TABLE, only 6 Genre vertices will be generated, because there are only 6 unique values; `"Fiction"` appears twice. Seven book_genre edges will be generated, one for each row in the TEMP_TABLE.

There is another version of the flatten function which has four arguments and which supports a two-level grouping. That is, the field contains a list of groups, each group composed of N subfields. The arguments are (field_to_split, group_separator, sub_field_separator, number_of_parts_in_one_group). For example, suppose the data line were organized this way instead:

book2.dat

```
bookcode|title|genres
101|"Harry Potter and the Philosopher's Stone"|FIC:fiction,ADV:advent
102|"The Three-Body Problem"|FIC:fiction,SF:science fiction,CHN:Chine
```

Then the following loading statements would be appropriate:

load_books2.gsql

```
CREATE ONLINE_POST JOB load_books2 FOR GRAPH book_rating {
  LOAD
      TO VERTEX Book VALUES ($0, $1),
      TO TEMP_TABLE t1(bookcode,genre_id,genre_name) VALUES ($0, flat

  LOAD TEMP_TABLE t1
      TO VERTEX Genre VALUES($"genre_id", $"genre_name"),
      TO EDGE book_genre VALUES($"bookcode", $"genre_id");
}
RUN JOB load_books2 USING FILENAME="book2.dat", SEPARATOR="|", EOL="\
```

Can the TigerGraph system load data from a streaming source?

Yes. Two approaches are to use our Kafka Loader or to periodically read from one or more files. A loading job lets you define a general loading process without naming the data source. Every time you call an online loading job, you name the source file. It can be a different file each time, or it can be the same file, if the contents of the file are changing over time. Also, if it happens that the loader re-reads a data line that it has encountered before, it will just reload the data (except for container attributes, e.g., a LIST attribute, using a reduce() loading function. In that case, there is an accumulative effect for re-reading a data line).

I want to compute an attribute value. What built-in functions are available?

The GSQL Loading includes some built-in token functions (a token is one column or field of a data input line.) A user can also define custom token functions. Please see the section "Built-In Loader Token Functions" in the [***GSQL Language Reference, Part 1: Defining Graphs and Loading Data***].

Do I need a one-to-one correspondence between input files and vertex types and edge types?

No. One of the advantages of the TigerGraph loading system is the flexible relationship between input files and resulting vertex and edge instances. In general, there is a many-to-many relationship: one input file can generate many vertex and edge types.

From the LOAD statement perspective for a online loading job:

```
LOAD
  TO VERTEX vertex_type VALUES (attr_expr...) [WHERE conditions],
  ...,
  TO VERTEX vertex_typeN VALUES (attr_expr...) [WHERE conditions],
  TO EDGE  edge_type VALUES (attr_expr...) [WHERE conditions] [OPTION (op
  ...,
  TO EDGE edge_typeN VALUES (attr_expr...) [WHERE conditions] [OPTION (op
  [Parsing_Conditions];
```

- Each LOAD statement refers to one input file.
- Each LOAD statement can have one or more resulting vertex types and one or more resulting edge types.
- Hence, one LOAD statement can potentially describe the one-to-many mapping from one input file to many resulting vertex and edge types.
- It is not necessary for every input line to always generate the same set of vertex types and edge types. The WHERE clause in each TO VERTEX | TO EDGE

clause can be used to selectively choose and filter which input lines generate which resulting types.

My input data includes multiple edge instances between a pair of vertices. Why is there only one in the graph?

This not an error. There can only be one instance of a certain edge type between any given pair of vertices, so the most recently loaded edge data will be the edge that you will see in the graph.

Updating and Modifying Data

How can I insert / load more data?

If there is already data in the graph store and you wish to insert more data, you have a few options. First, if you have bulk data stored in a file (local disk, remote or distributed storage), you can us e Online Loading .

Second, if you have a few specific insertions, you can use the Upsert da ta command in the ***RESTPP API User Guide .*** For Upsert, the data must be formatted in JSON format.

Third, you can write a query containing INSERT statements. The syntax is similar to SQL INSERT. (See ***GSQL Language Reference Part 2 - Querying .***) The advantage of query-based INSERT is that the details (id values and attribute values) can be determined at run time and even can be based on an exploration and analysis of the existing graph. The disadvantage is that the query-insert job must be compiled first and data values must either be hardcoded or supposed as input parameters.

How can I modify the graph schema?

You can modify the schema in several ways:

- Add new vertex or edge types
- Drop existing vertex or edge types
- Add or drop attributes from an existing vertex or edge type

⚠ Any schema change can invalidate existing loading jobs and queries.

See the section "Modifying a Graph Schema" in ***GSQL Language Reference Part 1 - Defining Graphs and Loading Data .***

How do I modify data?

To make a known modification of a known vertex or edge:
Option 1) Make a RESTPP endpoint request, to the POST /graph or DELETE /graph endpoint. See the ***RESTPP API User Guide*** .

Option 2) The Loading language includes an upsert command. The UPSERT statement performs a combined modify-or-add operation, depending on whether the indicated vertex or edge already exists. Examples of UPSERT are described in the ***GSQL Demo Examples*** document. The ***GSQL Language Reference Part 1 - Defining Graphs and Loading Data*** provides a full specification .

Option 3) The query language now includes an UPDATE statement which enables sophisticated selection of which vertices and edges to update and how to update them. Likewise, there is an INSERT statement in the query language. See the ***GSQL Language Reference Part 2 - Querying .***

How do I selectively delete data?

You can write a query which selects vertices or edges to be deleted. See the DELETE subsections of the "Data Modification Statements" section in ***GSQL Language Reference Part 2 - Querying .***

How do I erase all the data?

If you wish to completely clear all the data in the graph store, use the `CLEAR GRAPH STORE -HARD` command. Be very careful using this command; deleted data cannot be restored (except from a Backup). Note that clearing the data does not erase the catalog definitions of vertex, edge, and graph types. See also " **How do I delete my entire graph schema?** "

-HARD must be in all capital letters.

Querying

Is there more than one TigerGraph query language?

Yes. The GSQL Query Language is a full-featured graph query-and-data-computation language. In addition, there is a small lightweight set of built-in query commands that can inspect the set of stored vertices and edges, but these built-in commands do not support graph traversal (moving from one vertex to another via edges). We refer to this as the Standard Data Manipulation API or the Built-in Query Language (described in RESTPP API User Guide and the GSQL Demo Examples)

What is the basic syntax for the TigerGraph query language?

For a first-time user: See the documents ***GSQL Demo Examples*** and then ***GSQL Language Reference Part 2 - Querying .***
For users with some experience, a reference card is now available: *GSQL Query Language Reference Card.*

Is GSQL a query language or a programming language?

The GSQL Query Language supports powerful graph querying, but it is also designed to perform powerful computations. GSQL is Turing-complete, so it can be considered a programming language. It can be used for simple SQL-like queries, but it also features control flow (IF, WHILE, FOREACH), procedural calls, local and global variables, complex data types, and *accumulators* to enable much more sophisticated use.

What types of accumulators are available?

Three new types were introduced in v0.8: GroupByAccum, BitwiseAndAccum, and BitwiseOrAccum. Version 0.8.1. added ArrayAccum. This is a quick summary. For a more detailed explanation, see the "Accumulator Types" section of ***GSQL Language Reference Part 2 - Querying .***

In the following table, baseType means any of the following: INT, UINT, FLOAT, DOUBLE, STRING, BOOL, VERTEX, EDGE, JSONARRAY, JSONOBJECT, DATETIME

Accumulators	data types
SumAccum	INT, UINT, FLOAT, DOUBLE, STRING
MaxAccum, MinAccum	INT, UINT, FLOAT, DOUBLE, VERTEX
AvgAccum	INT, UINT, FLOAT, DOUBLE (output is DOUBLE)
AndAccum, OrAccum	BOOL
BitwiseAndAccum, BitwiseOrAccum	INT (acting as a sequence of bits)
ListAccum, SetAccum, BagAccum	baseType, TUPLE, STRING COMPRESS
ArrayAccum	accumulator, other than MapAccum, HeapAccum, or GroupByAccum
MapAccum	key: baseType, TUPLE, STRING COMPRESS value: baseType, TUPLE, STRING COMPRESS, ListAccum, SetAccum, BagAccum, MapAccum, HeapAccum

HeapAccum< *tuple_type* >(heapSize, sortKey [, sortKey_i]*)	TUPLE
	key: baseType, TUPLE, STRING COMPRESS

How do I use accumulators?

See the section "Accumulators" in the ***GSQL Language Reference Part 2 - Querying*** document.

How do I reference the ID fields of a vertex or edge in a built-in query?

Vertex and edge IDs (i.e., the unique identifier for each vertex or edge) are treated differently than user-defined attributes. Special keywords must be used to refer to the PRIMARY_ID, FROM, or TO id fields.

Vertices :

In a CREATE VERTEX statement, the PRIMARY_ID is required and is always listed first. User-defined attributes are optional and come after the required ID fields.

```
CREATE VERTEX Book  (PRIMARY_ID bookcode STRING, title STRING)
CREATE VERTEX Genre (PRIMARY_ID genre_id STRING, genre_name STRING)
CREATE UNDIRECTED EDGE book_genre (FROM Book, TO Genre)
CREATE GRAPH book_rating (Book, Genre, book_genre)
```

In a built-in query, if you wish to select vertices by specifying an attribute value, you use the attribute name (e.g., title):

```
SELECT * FROM Book WHERE title=="The Three-Body Problem"
```

In contrast, if you wish to reference vertices by the id value, the lowercase keyword `primary_id` must be used. Note that that query does not use the id name `pid`.

```
SELECT * FROM Book WHERE primary_id=="101"
```

Edges :

In a CREATE EDGE statement, the FROM and TO vertex identifiers are required and are always listed first. The FROM and TO values should match the PRIMARY_ID values of a source vertex and a target vertex. In the example below, `rating` and `date_time` are user-defined optional attributes.

```
CREATE UNDIRECTED EDGE book_genre (FROM Book, TO Genre, rating uint, date
```

In a query, if you wish to select edges by specifying their FROM or TO vertex values, you must use the lowercase keywords from_id or to_id .

```
SELECT * FROM Book-(book_genre)->Genre WHERE from_id=="101"
```

What is the format of data returned by a query?

The data are in JSON format. See the section "Output Statements" in the ***GSQL Language Reference Part 2 - Querying .***

Is there an output size limit for a data query?

Yes. The maximum output size for a query is 2GB. If the result of a query would be larger than 2GB, the system may return no data. No error message is returned.

Also, for built-in queries (using the Standard Data Manipulation REST API), queries return at most 10240 vertices or edges.

How and when do I use INSTALL QUERY and INSTALL QUERY -OPTIMIZE?

INSTALL QUERY *query_name* is required for each GSQL query, after its initial CREATE QUERY *query_name* statement and before using RUN QUERY *query_name* . After INSTALL query has been executed, RUN QUERY can now be used.

Anytime after INSTALL QUERY, another statement, INSTALL QUERY -OPTIMIZE can be executed once. This operation optimizes all previously installed queries, reducing their run times by about 20%.

Should I run INSTALL QUERY -OPTIMIZE?

Optimize a query if query run time is more important to you than query installation time.

The initial INSTALL QUERY operation runs quickly. This is good for the development phase.

The optional additional operation INSTALL QUERY -OPTIMIZE will take more time, but it will speed up query run time. This makes sense for *production* systems.

Legal:

```
CREATE QUERY query1...
INSTALL QUERY query1
RUN QUERY query1(...)
...
INSTALL QUERY -OPTIMIZE    # (optional) optimizes run time performance fo
RUN QUERY query1(...)      # runs faster than before
```

Illegal:

```
INSTALL QUERY -OPTIMIZE query_name
```

Can multiple users install queries at the same time?

In short, yes. They will not be executed at the same time, but the installations will be queued by the order in which they were received.

Can I make a 2-dimensional (or multi-dimensional) array?

Yes. A ListAccum is like an array, a 1-dimensional array. If you nest ListAccums as the elements within an outer ListAccum, you have effectively made a 2-dimensional array. Please read Section "Nested Accumulators" in the ***GSQL Language Reference Part 2 - Querying*** for more details. Here is an example:

```
CREATE QUERY nestedAccumEx() FOR GRAPH anyGraph {
  ListAccum<ListAccum<INT>> @@_2d_list;
  ListAccum<ListAccum<ListAccum<INT>>> @@_3d_list;
  ListAccum<INT> @@_1d_list;
  SumAccum <INT> @@sum = 4;

  @@_1d_list += 1;
  @@_1d_list += 2;
  // add 1D-list to 2D-list as element
  @@_2d_list += @@_1d_list;

  // add 1D-enum-list to 2D-list as element
  @@_2d_list += [@@sum, 5, 6];
  // combine 2D-enum-list and 2d-list
  @@_2d_list += [[7, 8, 9], [10, 11], [12]];

  // add an empty 1D-list
  @@_1d_list.clear();
  @@_2d_list += @@_1d_list;

  // combine two 2D-list
  @@_2d_list += @@_2d_list;

  PRINT @@_2d_list;

  // test 3D-list
  @@_3d_list += @@_2d_list;
  @@_3d_list += [[7, 8, 9], [10, 11], [12]];
  PRINT @@_3d_list;
}
```

Can I make nested container Accumulators?

Yes, please read Section "Nested Accumulators" in the ***GSQL Language Reference Part 2 - Querying*** for more details. There are seven types of container accumulators: ListAccum, SetAccum, BagAccum, MapAccum, ArrayAccum HeapAccum, and GroupByAccum. Here the allowed combinations:

- ListAccum can contain ListAccum.
- MapAccum and GroupByAccum can contain any container accumulator except HeapAccum.
- ArrayAccum is always nested.

Here is an example:

```
CREATE QUERY nestedMap() FOR GRAPH anyGraph
{
  MapAccum<String, MapAccum<int, String>> @@testMap;

  @@testMap += ("m1" -> (0 -> "value1"));
  @@testMap += ("m1" -> (1 -> "value2"));
  @@testMap += ("m2" -> (2 -> "value3"));

  IF @@testMap.containsKey("m1") THEN
    PRINT @@testMap.get("m1");
  END;
   //for map, we can get it's value, and then, get the value's key.
  PRINT @@testMap.get("m1").get(0);
}
```

Testing and Debugging

How can I validate a loading job?

To write a loading job, you must know the format of the input data files, so that you can describe to GSQL how to parse each data line and convert it into vertex and edge attributes. To validate a loading job, that is, to check that the actual input data meet your expectations, and that they produce the expected vertices and edges, you can use two features of the RUN JOB command: the -DRYRUN option and loading a specified range of data lines.

The full syntax for an (offline) loading job is the following:

`RUN JOB [-DRYRUN] [-n [` *`first_line_num`* `,]` *`last_line_num`* `]` *`job_name`*

The **`-DRYRUN`** option will read input files and process data as instructed by the job, but it does not store data in the graph store.

The **`-n`** option limits the loading job to processing only a range of lines of each input data file. The selected data will be stored in the graph store, so the user can check the results. The -n flag accepts one or two arguments. For example,

`-n 50` means read lines 1 to 50.
`-n 10,50` means read lines 10 to 50.
The special symbol $ is interpreted as "last line", so **`-n 10,$`** means reads from line 10 to the end.

Where are the logs?

The following command lists the log locations of the log files:

```
gadmin log
```

If the platform has been installed with default file locations, so that <TigerGraph_root_dir> = /home/tigergraph/tigergraph, then the output would be the following:

```
GPE : /home/tigergraph/tigergraph/logs/gpe/gpe1.out
GPE : /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO
GSE : /home/tigergraph/tigergraph/logs/gse/gse1.out
GSE : /home/tigergraph/tigergraph/logs/GSE_1_1/log.INFO
RESTPP : /home/tigergraph/tigergraph/logs/restpp/restpp1.out
RESTPP : /home/tigergraph/tigergraph/logs/RESTPP_1_1/log.INFO
RESTPP : /home/tigergraph/tigergraph/logs/RESTPP-LOADER_1_1/log.INFO
GSQL : /home/tigergraph/tigergraph/logs/gsql_server_log/GSQL_LOG
```

ⓘ As of v2.4, the GSQL log files have been moved in order to keep all logs in a standard directory.

GPE: general system performance logs.
GSE: Graph services logs.
RESTPP: REST API call logs.
GSQL: General GSQL logs.

Where are the log files of loading runs?

Each loading run creates a log file, stored in the folder `<TigerGraph_rootdir>/app/<VERSION_NUM>/dev/gdk/gsql/output`. The filename load_output.log is a link to the most recent log file. This file contains summary statistics on the number of lines read, the vertices created, and various types of errors encountered. Or, you can type a shell command to find log paths "`gadmin log`".

What are in the log files?

The log files record detailed internal operations and state information in response to user actions. They provide vital information for diagnosing and debugging your system. All log files can be found in the /home/tigergraph/tigergraph/logs directory. Through typing the command `gadmin log`, you will be given all the file paths of the most commonly used log files.

GPE Logs - Graph Processing Engine Logs
GSE Logs - Graph Storage Engine Logs
GSQL Logs - System & Query Logs
RESTPP Logs - API call Logs
NGINX Logs - HTTP Request Logs
VIS Logs - GraphStudio Logs

I can’t seem to load any more data. What’s the matter?

One possible explanation is that you have reached a capacity limit controlled by your product license. To check if this is the case, run the command gadmin status. If the limit has been reached, there will be a warning message, such as the following:

```
[Warning] License limit exceeded. The system is running in limited capaci
```

In Limited Capacity mode, additional data may not be inserted. Data may be queried and deleted.

Troubleshooting Guide

This troubleshooting guide is only up to date for v2.6 and below.
Additional guidance for v3.0+ is in development.

Introduction

The Troubleshooting Guide teaches you how to monitor the status of your TigerGraph system, and when needed, find the log files in order to get a better understanding of why certain errors are occurring. This section covers log file debugging for data loading and querying.

General

Before any deeper investigation, always run these general system checks :

```
$ gadmin status          (Make sure all TigerGraph services are UP.)

$ df -lh                 (Make sure all servers are getting enough disk spa

$ free -g                (Make sure all servers have enough memory.)

$ tsar                   (Make sure there is no irregular memory usage on t

$ dmesg -T | tail        (Make sure there are no Out of Memory, or any othe

$ grun all "date"        (Make sure the time across all nodes are synchroni
                          with time difference under 2 seconds. )
```

Location of Log Files

The following command reveals the location of the log files :

```
gadmin log
```

You will be presented with a list of log files. The left side of the resulting file paths is the component for which the respective log file is logging information. The majority of the time, these files will contain what you are looking for. You may notice that there are multiple files for each TigerGraph component.

> The .out file extension is for errors.
> The .INFO file extension is for normal behaviors.

In order to diagnose an issue for a given component, you'll want to check the .out log file extension for that component.

```
tigergraph@ubuntu:~/tigergraph/logs/nginx$ gadmin log
GPE : /home/tigergraph/tigergraph/logs/gpe/gpe_1_1.out
GPE : /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO
GSE : /home/tigergraph/tigergraph/logs/gse/gse_1_1.out
GSE : /home/tigergraph/tigergraph/logs/GSE_1_1/log.INFO
RESTPP : /home/tigergraph/tigergraph/logs/restpp/restpp_1_1.out
RESTPP : /home/tigergraph/tigergraph/logs/RESTPP-LOADER_1_1/log.INFO
RESTPP : /home/tigergraph/tigergraph/logs/RESTPP_1_1/log.INFO
GSQL : /home/tigergraph/tigergraph/dev/gdk/gsql/logs/GSQL_LOG
NGINX : /home/tigergraph/tigergraph/logs/nginx/nginx1.out
NGINX : /home/tigergraph/tigergraph/logs/nginx/nginx_1.error.log
NGINX : /home/tigergraph/tigergraph/logs/nginx/nginx_1.access.log
VIS : /home/tigergraph/tigergraph/logs/gui/gui_ADMIN.log
VIS : /home/tigergraph/tigergraph/logs/gui/gui_INFO.log
```

Other log files that are not listed by the **`gadmin log`** command are those for Zookeeper and Kafka, which can be found here:

```
zookeeper : ~/tigergraph/zk/zookeeper.out.*
kafka     : ~/tigergraph/kafka/kafka.out
```

Synchronize time across nodes in a cluster

TigerGraph will experience a variety of issues if clocks across different nodes in a cluster are out of sync. If running `grun all "date"` shows that the clocks are out of sync, it is highly recommended that you install NTP implementations such as `chrony` ↗ or `timesyncd` ↗ to keep them in sync.

Installation Error Debugging

Missing Dependencies

The installation will quit if there are any missing dependency packages, and output a message. Please run `bash install_tools.sh` to install all missing packages. You will need an internet connection to install the missing dependencies.

Pinpoint The Failed Step

Using the -x flag during installation will show you the detailed shell commands being run during installation.
`bash -x install.sh`

Disk Space Errors

- The `/home` directory requires at least 200MB of space, or the installation will fail with an out of disk message. This is temporary only during installation and will be moved to the root directory once installation is complete.
- The `/tmp` directory requires at least 1GB of space, or the installation will fail with an out of disk message
- The directory in which you choose to install TigerGraph requires at least 20GB of space, or the installation will report the error and exit.

Firewall Errors

If your firewall blocks all ports not defined for use, we recommend opening up internal ports 1000-50000.

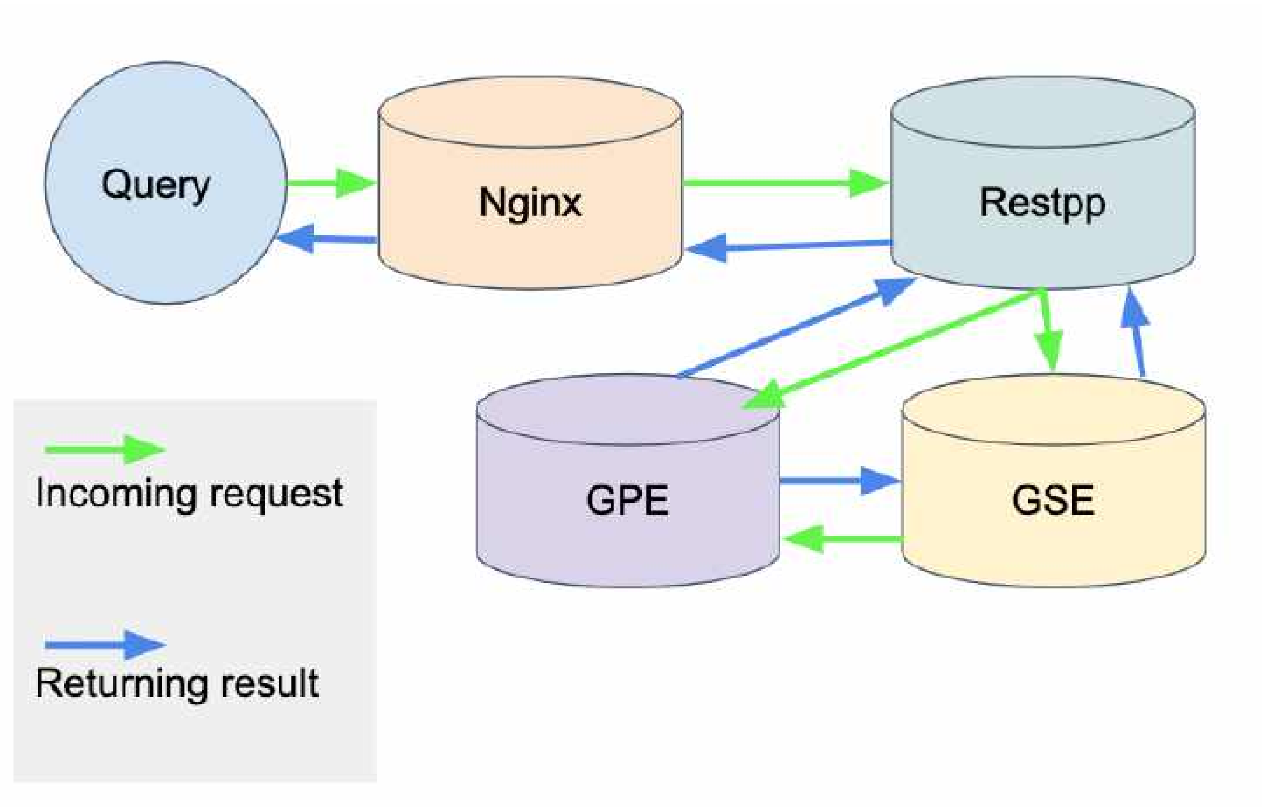
If you are using a cloud instance, you will need to configure the firewall rules through the respective consoles.
e.g. Amazon AWS or Microsoft Azure
If you are managing a local machine, you can manage your open ports using the `iptables` command. Please refer to the example below to help with your firewall configuration.

```
# iptables help page
$ sudo iptables -h

# This will list your firewall rules
$ sudo iptables -L

# Allow incoming SSH connections to port 22 from the 192.168.0.0 subnet
$ sudo iptables -A INPUT -p tcp --dport 22 -s 192.168.0.0/24 -j ACCEPT
$ sudo iptables -A INPUT -p tcp --dport 22 -s 127.0.0.0/8 -j ACCEPT
$ sudo iptables -A INPUT -p tcp --dport 22 -j DROP
```

Query Debugging

Checking the Logs - Flow of a query in the system

To better help you understand the flow of a query within the TigerGraph system, we've provided the diagram below with arrows showing the direction of information flow. We'll walk through the execution of a typical query to show you how to observe the information flow as recorded in the log files.



From calling a query to returning the result, here is how the information flows:

1. Nginx receives the request.

```
grep <QUERY_NAME> /home/tigergraph/tigergraph/logs/nginx/ngingx_1.access.
```

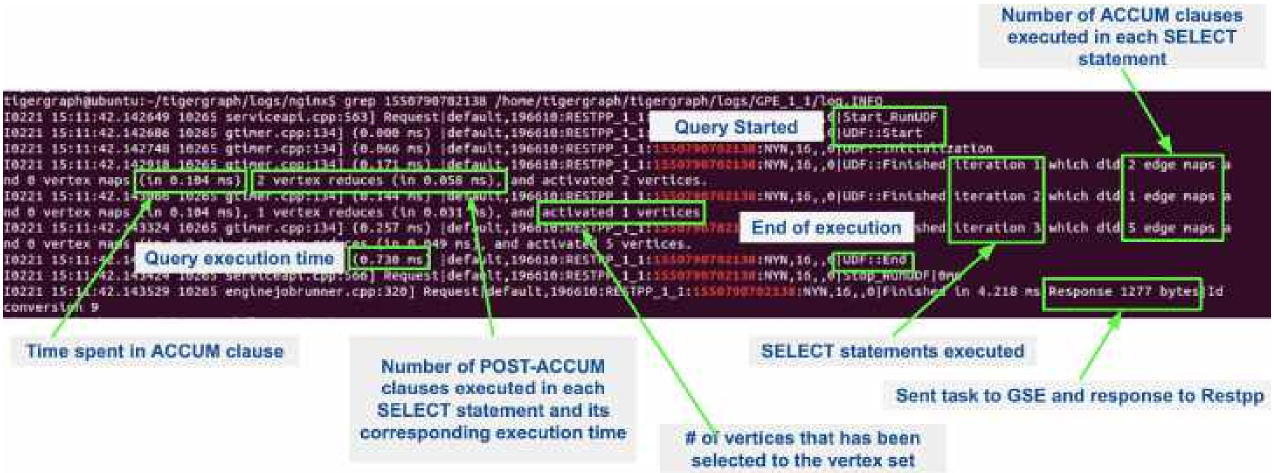
ⓘ You can click on the image below to expand it.

2. Nginx sends the request to Restpp.

```
grep <QUERY_NAME> /home/tigergraph/tigergraph/logs/RESTPP_1_1/log.INFO
```

3. Restpp sends an ID translation task to GSE and a query request to GPE.
4. GSE sends the translated ID to GPE, and the GPE starts to process the query.
5. GPE sends the query result to Restpp, and sends a translation task to GSE, which then sends the translation result to Restpp.

```
grep <REQUEST_ID> /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO
```




```
grep <REQUEST_ID> /home/tigergraph/tigergraph/logs/GSE_1_1/log.INFO
```

6. Restpp sends the result back to Nginx.

```
grep <REQUEST_ID> /home/tigergraph/tigergraph/logs/RESTPP_1_1/log.INFO
```

7. Nginx sends the response.

```
grep <QUERY_NAME> /home/tigergraph/tigergraph/logs/nginx/nginx_1.access.l
```

Other Useful Commands for Query Debugging

```
Check recently executed query:
$ grep UDF:: /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO | tail -n

Get the number of queries executed recently:
$ grep UDF::End /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO | wc -l

Grep distributed query log:
$ grep “Action done” /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO |


Grep logs from all servers:
$ grun all “grep UDF:: /home/tigergraph/tigergraph/logs/GPE_*/log.INFO |
```

Slow Query Performance

Multiple situations can lead to slower than expected query performance:

- **Insufficient Memory**
  When a query begins to use too much memory, the engine will start to put data onto the disk, and memory swapping will also kick in. Use the Linux command: `free -g` to check available memory and swap status, or you can also monitor the memory usage of specific queries through GPE logs. To avoid running into insufficient memory problems, optimize the data structure used within the query or increase the physical memory size on the machine.
- **GSQL Logic**
  Usually, a single server machine can process up to 20 million edges per second. If the actual number of vertices or edges is much much lower, most of the time it can be due to inefficient query logic. That is, the query logic is now following the natural execution of GSQL. You will need to optimize your query to tune the performance.
- **Disk IO**
  When the query writes the result to the local disk, the disk IO may be the bottleneck for the query's performance. Disk performance can be checked with

this Linux command : `sar 1 10`.
If you are writing (PRINT) one line at a time and there are many lines, storing the data in one data structure before printing may improve the query performance.

- **Huge JSON Response**
If the JSON response size of a query is too massive, it may take longer to compose and transfer the JSON result than to actually traverse the graph. To see if this is the cause, check the GPE log.INFO file. If the query execution is already completed in GPE but has not been returned, and CPU usage is at about 200%, this is the most probable cause. If possible, please reduce the size of the JSON being printed.
- **Memory Leak**
This is a very rare issue. The query will progressively become slower and slower, while GPE's memory usage increases over time. If you experience these symptoms on your system, please report this to the TigerGraph team.
- **Network Issues**
When there are network issues during communication between servers, the query can be slowed down drastically. To identify that this is the issue, you can check the CPU usage of your system along with the GPE log.INFO file. If the CPU usage stays at a very low level and GPE keeps printing `???` , it means network IO is very high.
- **Frequent Data Ingestion in Small Batches**
Small batches of data can increase the data loading overhead and query processing workload. Please increase the batch size to prevent this issue.

Query Hangs

When a query hangs or seems to run forever, it can be attributed to these possibilities :

- **Services are down**
Please check that TigerGraph services are online and running. Run `gadmin status` and possibly check the logs for any issues that you find from the status check.
- **Query is in an infinite loop**
To verify this is the issue, check the GPE log.INFO file to see if graph iteration log lines are continuing to be produced. If they are, and the edgeMaps log the same number of edges every few iterations, you have an infinite loop in your

query.
If this is the case, please restart GPE to stop the query : `gadmin restart gpe -y`.
Proceed to refine your query and make sure your loops within the query are able to break out of the loop.

- **Query is simply slow**
  If you have a very large graph, please be patient. Ensure that there is no infinite loop in your query, and refer to the slow query performance section for possible causes.
- **GraphStudio Error**
  If you are running the query from GraphStudio, the loading bar may continue spinning as if the query has not finished running. You can right-click the page and select *inspect→console* (in the Google Chrome browser) and try to find any suspicious errors there.

Query Returns No Result

If a query runs and does not return a result, it could be due to two reasons:
1. Data is not loaded.
From the Load Data page on GraphStudio, you are able to check the number of loaded vertices and edges, as well as the number of each vertex or edge type. Please ensure that all the vertices and edges needed for the query are loaded.

2. Properties are not loaded.
The number of vertices and edges traversed can be observed in the GPE log.INFO file. If for one of the iterations you see **activated 0 vertices**, this means no target vertex satisfied your searching condition. For example, the query can fail to pass a WHERE clause or a HAVING clause.

If you see **0 vertex reduces** while the edge map number is not 0, that means that all edges have been filtered out by the `WHERE` clause, and that no vertices have entered into the `POST-ACCUM` phase. If you see more than 0 vertex reduces, but **activated 0 vertices**, this means all the vertices were filtered out by the `HAVING` clause.

To confirm the reasoning within the log file, use GraphStudio to pick a few vertices or edges that should have satisfied the conditions and check their attributes for any unexpected errors.

Query Installation Failed

Query Installation may fail for a handful of reasons. If a query fails to install, please check the GSQL log file. The default location for the GSQL log is here :

```
/home/tigergraph/tigergraph/logs/gsql_server_log/GSQL_LOG
```

Go down to the last error and it will point you to the error. This will show you any query errors that could be causing the failed installation. If you have created a user-defined function, you could potentially have a c++ compilation error.

> If you have a c++ user-defined function error, your query will fail to install, even if it does not utilize the UDF.

Debugging Memory-related Failures

How to monitor memory usage by query

GPE records memory usage by query at different stages of the query and saves it to `$(gadmin config get System.LogRoot)/gpe/log.INFO`. You can monitor how much memory a query is using by searching the log file for the request ID and filter for lines that contain `"QueryMem"`:

```
grep -i <request_id> $(gadmin config get System.LogRoot)/gpe/log.INFO |
    grep -i "querymem"
```

You can also run a query first, and then run the following command immediately after to retrieve the most recent query logs and filter for `"QueryMem"`:

```
tail -n 50 $(gadmin config get System.LogRoot)/gpe/log.INFO |
    grep -i "querymem"
```

You will get results that look like the following, which shows memory usage by the query at different stages of its execution. The number at the end of each line indicates the number of bytes of memory utilized by the query:

```
0415 01:33:40.885433  6553 gpr.cpp:195] Engine_MemoryStats|     \
ldbc_snb::,196612.RESTPP_1_1.1618450420870.N,NNN,15,0,0|         \
MONITORING Step(1) BeforeRun[GPR][QueryMem]: 116656

I0415 01:33:42.716199  6553 gpr.cpp:241] Engine_MemoryStats|     \
ldbc_snb::,196612.RESTPP_1_1.1618450420870.N,NNN,15,0,0|         \
MONITORING Step(1) AfterRun[GPR][QueryMem]: 117000
```

How to check system free memory percentage

You can check how much free memory your system has as a percentage of its total memory by running the following command:

```
tail -n 50 $(gadmin config get System.LogRoot)/gpe/log.INFO | grep -i 'fr
```

The number following `"FreePct"` indicates the percentage of the system free memory. The following example shows the system free memory is 69%:

```
I0520 23:40:09.845811  7828 gsystem.cpp:622]
System_GSystem|GSystemWatcher|Health|ProcMaxGB|0|ProcAlertGB|0|
CurrentGB|1|SysMinFreePct|10|SysAlertFreePct|30|FreePct|69
```

When free memory drops below 10 percent (`SysMinFreePct`), all queries are aborted. This threshold is adjustable through [`gadmin config`].

How to retrieve information on queries aborted due to memory usage

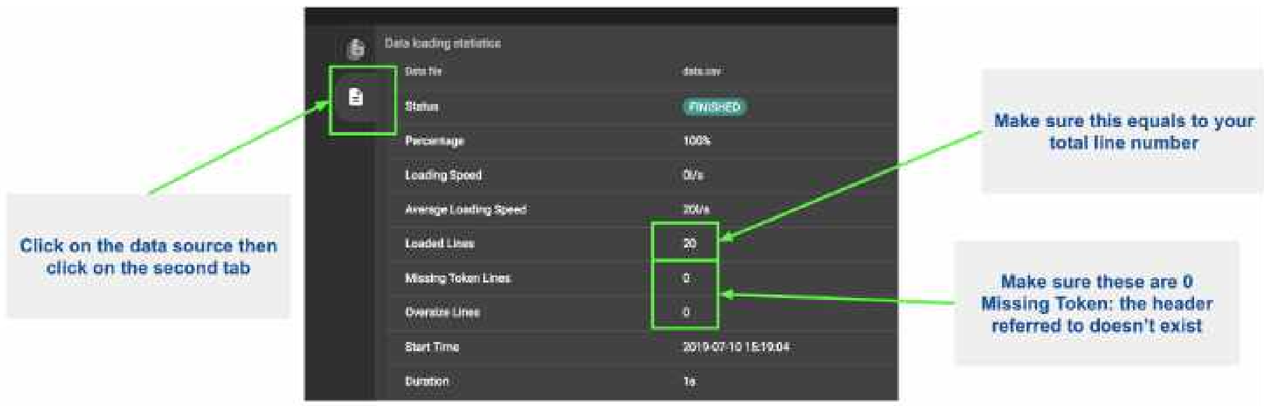
```
 log:W0312 02:10:57.839139 15171 scheduler.cpp:116] System Memory in Crit
```

Data Loading Debugging

Checking the Logs

GraphStudio

Using GraphStudio, you are able to see, from a high-level, a number of errors that may have occurred during the loading. This is accessible from the Load Data page. Click on one of your data sources, then click on the second tab of the graph statistics chart. There, you will be able to see the status of the data source loading, number of loaded lines, number of lines missing, and lines that may have an incorrect number of columns. (Refer to picture below.)

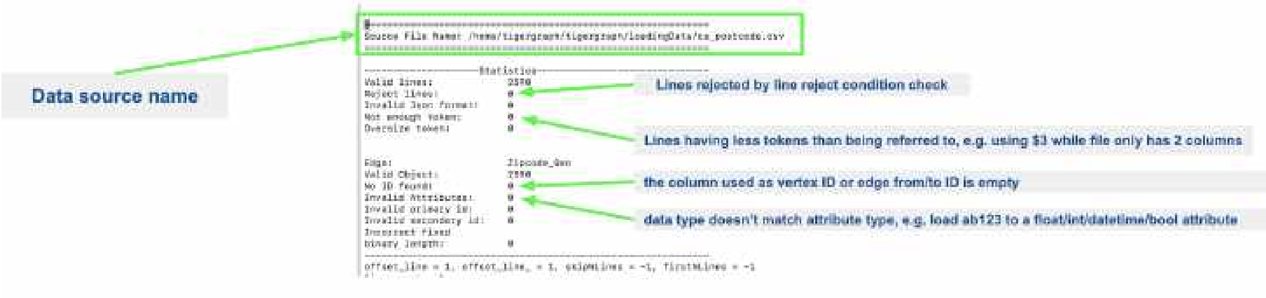


Command Line

If you see there are a number of issues from the GraphStudio Load Data page, you can dive deeper to find the cause of the issue by examining the log files. Check the loading log located here:

```
/home/tigergraph/tigergraph/logs/restpp/restpp_loader_logs/<GRAPH_NAME>/
```

Open up the latest **.log** file and you will be able to see details about each data source. The picture below is an example of a correctly loaded data file.



Here is an example of a loading job with errors :

From this log entry, you are able to see the errors being marked as lines with invalid attributes. The log will provide you the line number from the data source which contains the loading error, along with the attribute it was attempting to load to.

Slow Loading

Normally, a single server running TigerGraph will be able to load from 100k to 1000k lines per second, or 100GB to 200GB of data per hour. This can be impacted by any of the following factors:

- **Loading Logic**
  How many vertices/edges are generated from each line loaded?
- **Data Format**
  Is the data formatted as JSON or CSV? Are multi-level delimiters in use? Does the loading job intensively use temp_tables?

- **Hardware Configuration**
  Is the machine set up with HDD or SSD? How many CPU cores are available on this machine?
- **Network Issue**
  Is this machine doing local loading or remote POST loading? Any network connectivity issues?
- **Size of Files**
  How large are the files being loaded? Many small files may decrease the performance of the loading job.
- **High Cardinality Values Being Loaded to String Compress Attribute Type**
  How diverse is the set of data being loaded to the String Compress attribute?

To combat the issue of slow loading, there are also multiple methods:

- If the computer has many cores, consider increasing the number of Restpp load handlers.

```
$ gadmin --config handler
increase the number of handlers
save
$ gadmin --config apply
```

- Separate `~/tigergraph/kafka` from `~/tigergraph/gstore` and store them on separate disks.
- Do distributed loading.
- Do offline batch loading.
- Combine many small files into one larger file.

Loading Hangs

When a loading job seems to be stuck, here are things to check for :

- **GPE is DOWN**
  You can check the status of GPE with this command : `gadmin status gpe`
  If GPE is down, you can find the logs necessary with this command : `gadmin log -v gpe`

- **Memory is full**
  Run this command to check memory usage on the system : `free -g`
- **Disk is full**
  Check disk usage on the system : `df -lh`
- **Kafka is DOWN**
  You can check the status of Kafka with this command : `gadmin status kafka`
  If it is down, take a look at the log with this command : `vim ~/tigergraph/kafka/kafka.out`
- **Multiple Loading Jobs**
  By default, the Kafka loader is configured to allow a single loading job. If you execute multiple loading jobs at once, they will run sequentially.

Data Not Loaded

If the loading job completes, but data is not loaded, there may be issues with the data source or your loading job. Here are things to check for:

- Any invalid lines in the data source file. Check the log file for any errors. If an input value does not match the vertex or edge type, the corresponding vertex or edge will not be created.
- Using quotes in the data file may cause interference with the tokenization of elements in the data file. Please check the GSQL Language Reference section under Other Optional LOAD Clauses. Look for the QUOTE parameter to see how you should set up your loading job.
- Your loading job loads edges in the incorrect order. When you defined the graph schema, the **from** and **to** vertex order will affect the way you write the loading job. If you wrote the loading job in reversed order, the edges will not be created, possibly also affecting the population of vertices.

Loading is Incorrect

If you know what data you expect to see (number of vertices and edges, and attribute values), but the loaded data does not mean your expectations, there are a number of possible causes to investigate:

1. First, check the logs for important clues.
2. Are you reaching and reading all the data sources (paths and permissions)?
3. Is the data mapping correct?
4. Are your data fields correct? In particular, check data types. For strings, check for unwanted extra strings. Leading spaces are not removed unless you apply an optional token function to trim the extra spaces.
5. Do you have duplicate ids, resulting in the same vertex or edge being loading more than once. Is this intended or unintended? TigerGraph's default loading semantics is UPSERT. Check the loading documentation to maker sure you understand the semantics in detail:

   https://docs.tigergraph.com/dev/gsql-ref/ddl-and-loading/creating-a-loading-job#cumulative-loading ↗

Loading Failure

Possible causes of a loading job failure are:

- **Loading job timed out**
  If a loading job hangs for 600 seconds, it will automatically time out.
- **Port Occupied**
  Loading jobs require port 8500. Please ensure that this port is open.

Schema Change Debugging

This section will only cover the debugging schema change jobs, for more information about schema changes, please read the Modifying a Graph Schema page.

Understanding what happens behind the scenes during a schema change.

1. **DSC (Dynamic Schema Change) Drain** - Stops the flow of traffic to RESTPP and GPE
   If GPE receives a DRAIN command, it will wait 1 minute for existing running

queries to finish up. If the queries do not finish within this time, the DRAIN step will fail, causing the schema change to fail.

2. **DSC Validation** - Verification that no queries are still running.
3. **DSC Apply** - Actual step where the schema is being changed.
4. **DSC Resume** - Traffic resumes after schema change is completed. Resume will automatically happen if a schema change fails. RESTPP comes back online. All buffered query requests will go through after RESTPP resumes, and will use the new updated schema.

> Schema changes are not recommended for production environments.
> Even if attributes are deleted, TigerGraph's engine will still scan all previous attributes.
> We recommend limiting schema changes to dev environments.
>
> Schema changes are all or nothing. If a schema change fails in the middle, changes will not be made to the schema.

Signs of Schema Change Failure

- Failure when creating a graph
- Global Schema Change Failure
- Local Schema Change Failure
- Dropping a graph fails
- If GPE or RESTPP fail to start due to YAML error, please report this to TigerGraph.

If you encounter a failure, please take a look at the GSQL log file : `gadmin log gsql`. Please look for these error codes:

- **Error code 8** - The engine is not ready for the snapshot. Either the pre-check failed or snapshot was stopped. The system is in critical non-auto recoverable error state. Manual resolution is required. Please contact TigerGraph support.
- **Error code 310** - Schema change job failed and the proposed change has not taken effect. This is the normal failure error code. Please see next section for failure reasons.

Reasons For Dynamic Schema Change Failure

- **Another schema change or a loading job is running.** This will cause the schema change to fail right away.
- **GPE is busy.** Potential reasons include :
  - Long running query.
  - Loading job is running.
  - Rebuild process is taking a long time.
- **Service is down.** (RESTPP/GPE/GSE)
- **Cluster system clocks are not in sync.** Schema change job will think the request is stale, causing this partition's schema change to fail.
- **Config Error.** If the system is shrunk manually, schema change will fail.

Log Files

You will need to check the logs in this order : GSQL log, admin_server log, service log.
Admin_server log files can be found here : `~/tigergraph/logs/admin_server/` You will want to take a look at the INFO file.
The service log is each of the services respectively. `gadmin log <service_name>` will show you the location of these log files.

Example of a successful schema change job. (admin_server log)

```
$ grep DSC ~/tigergraph/logs/admin_server/INFO.20181011-101419.98774

I1015 12:04:14.707512 116664 gsql_service.cpp:534] Notify RESTPP DSCDrain
I1015 12:04:15.765108 116664 gsql_service.cpp:534] Notify GPE DSCDrain su
I1015 12:04:16.788666 116664 gsql_service.cpp:534] Notify GPE DSCValidati
I1015 12:04:17.805620 116664 gsql_service.cpp:534] Notify GSE DSCValidati
I1015 12:04:18.832386 116664 gsql_service.cpp:534] Notify GPE DSCApply su
I1015 12:04:21.270011 116664 gsql_service.cpp:534] Notify RESTPP DSCApply
I1015 12:04:21.692147 116664 gsql_service.cpp:534] Notify GSE DSCApply su
```

Example of DSC fail

```
E1107 14:13:03.625350 98794 gsql_service.cpp:529] Failed to notify RESTPP
E1107 14:13:03.625562 98794 gsql_service.cpp:344] DSC failed at Drain sta
E1107 14:14:03.814132 98794 gsql_service.cpp:513] Failed to notify RESTPP
```

In this case, we see that RESTPP failed at the DRAIN stage. We need to first look at whether RESTPP services are all up. Then, verify that the time of each machine is the same. If all these are fine, we need to look at RESTPP log to see why it fails. Again, use the "DSC" keyword to navigate the log.

GSE Error Debugging

To check the status of GSE, and all other processes, run `gadmin status` to show the status of key TigerGraph processes. As with all other processes, you are able to find the log file locations for GSE by the `gadmin log` command. Refer to the Location of Log Files for more information about which files to check.

```
$ gadmin log gse
[Warning] License will expire in 5 days
GSE : /home/tigergraph/tigergraph/logs/gse/gse_1_1.out
GSE : /home/tigergraph/tigergraph/logs/GSE_1_1/log.INFO
```

GSE Process Fails To Start

If the GSE process fails to start, it is usually attributed to a license issue, please check these factors :

- **License Expiration**
  `gadmin status license` This command will show you the expiration date of your license.
- **Single Node License on a Cluster**
  If you are on a TigerGraph cluster, but using a license key intended for a single machine, this will cause issues. Please check with your point of contact to see which license type you have.

- **Graph Size Exceeds License Limit**
  Two cases may apply for this reason. The first reason is you have multiple graphs but your license only allows for a single graph. The second reason is that your graph size exceeds the memory size that was agreed upon for the license. Please check with your point of contact to verify this information.

GSE status is "not_ready"

Usually in this state, GSE is warming up. This process can take quite some time depending on the size of your graph.

<INCLUDE PROCESS NAME SHOWING CPU USAGE TO VERIFY THE "WARMING UP" STATE>

> Very rarely, this will be a ZEROMQ issue. Restarting TigerGraph should resolve this issue
>
> `gadmin restart -y`

GSE crash

GSE crashes are likely due to and Out Of Memory issue. Use the `dmesg -T` command to check any errors.

> If GSE crashes, and there are no reports of OOM, please reach out to TigerGraph support.

GSE High Memory Consumption

If your system has unexpectedly high memory usage, here are possible causes :

- **Length of ID strings is too long**
  GSE will automatically deny IDs with a length longer than 16k. Memory issues

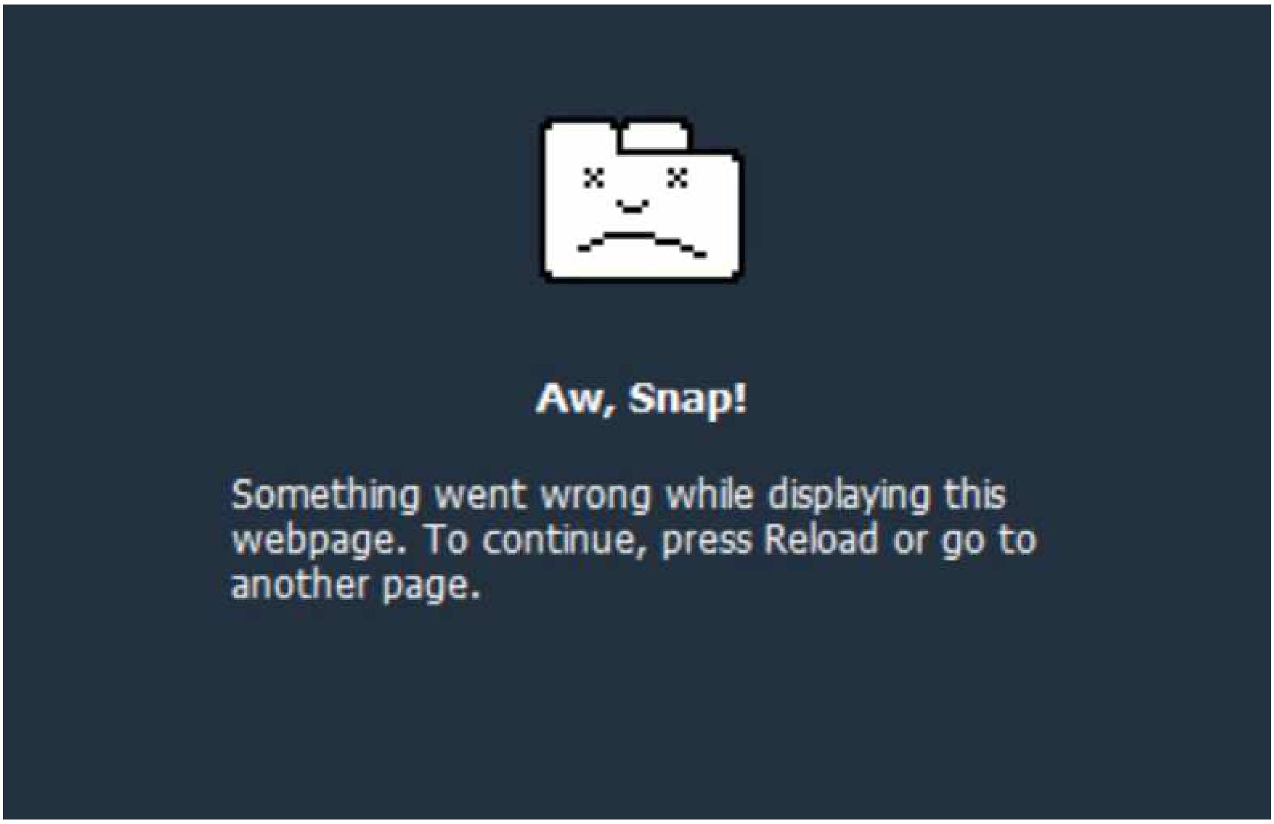
could also arise if an ID string is too long (> 500). One proposed solution to this is to hash the string.

- **Too Many Vertex Types**
  Check the number of unique vertex types in your graph schema. If your graph schema requires more than 200 unique vertex types, please contact TigerGraph support.

GraphStudio Debugging

Browser Crash / Freeze

If your browser crashes or freezes (shown below), please refresh your browser.



GraphStudio Crash

If you suspect GraphStudio has crashed, first run `gadmin status` to verify all the components are in good shape. Two known causes of GraphStudio crashes are :

- **Huge JSON response**
  User-written queries can return very large JSON responses. If GraphStudio often crashes on large query responses, you can try reducing the size limit for JSON responses by changing the `GUI.RESTPPResponseMaxSizeBytes` configuration using `gadmin config`. The default limit is 33554432 bytes.

```
$ gadmin config entry GUI.RESTPPResponseMaxSizeBytes
New: 33554431
[   Info] Configuration has been changed. Please use 'gadmin config apply
$ gadmin config apply
```

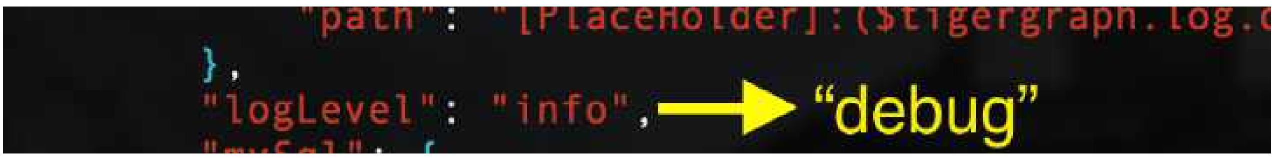
- **Very Dense Graph Visualization**
  On the Explore Graph page, the "Show All Paths" query on a very dense graph is known to cause a crash.

DEBUG mode

To find the location of GraphStudio log files, use this command : `gadmin log gui`

```
$ gadmin log vis
[Warning] License will expire in 5 days
VIS : /home/tigergraph/tigergraph/logs/gui/gui_ADMIN.log
VIS : /home/tigergraph/tigergraph/logs/gui/gui_INFO.log
```

Allowing GraphStudio DEBUG mode will print out more information to the log files. To allow DEBUG mode, please edit the following file :
`/home/tigergraph/tigergraph/visualization/server/src/config/local.json`



After editing the file, run `gadmin restart gui -y` to restart the GraphStudio service. Follow along the log file to see what is happening : `tail -f /home/tigergraph/tigergraph/logs/gui/gui_INFO.log`

Repeat the error-inducing operations in GraphStudio and view the logs.

Known Issues

There is a list of known GraphStudio issues here.

Further Debugging

If after taking these actions you cannot solve the issue, please reach out to support@tigergraph.com to request assistance.

Log Files

TigerGraph Database captures key information on activities occurring across its different components through log functions that output to log files. These log files are not only helpful in troubleshooting but also serve as an auditory resource. This document gives a high-level overview of TigerGraph's logging structure and lists some common information one might need to monitor their database services and where to obtain them in the logs.

Overall Logging Structure

Logs in TigerGraph are stored at `<tigergraph_root_dir>/log/`. TigerGraph's logs are divided into different folders by the different internal components and each folder corresponds to a different component. Log formats also vary across the different components. In folders where logs are checked often, such as `restpp`, `gsql`, and `admin`, there are three symbolic links that help you quickly get to the most recent log file of that category:

- `log.INFO`
  - Contains regular output and errors
- `log.ERROR`
  - Contains errors only
- `<component_name>.out`
  - Contains all output from the component process
- `log.WARNING` or `log.DEBUG`
  - `log.WARNING` contains warnings
  - In the `gsql` folder, `log.DEBUG` contains very specific information you only need when certain errors happen

Knowing where certain activities are recorded allows one to use tools such as the Linux `grep` command to easily obtain critical information from your database.

Log locations on a cluster

In a TigerGraph cluster, each node will only keep logs of activities that took place on the node itself. For example, the GSQL logs on the m1 node will only record events for m1 and are not replicated across the cluster.

For GSQL specifically, the cluster will elect a leader to which all GSQL requests will be forwarded. To check which node is the leader, start by checking the GSQL logs of the m1 node. Check the most recent lines of `log.INFO` and look for lines containing information about leader switch. For example, the logs below recorded a GSQL leader switch from m2 to m1:

```
I@20210709 13:56:52.214  (GsqlHAHandler.java:231) GSQL leader switches fr
E@20210709 13:56:52.215  (GsqlHAHandler.java:246) GSQL HA leader switches
If you want to lower the chance of leader switch by increasing timeout, p
I@20210709 13:56:52.219  (SessionManager.java:197) Abort and clear all se
I@20210709 13:56:52.220  (SessionManager.java:204) All sessions aborted.
I@20210709 13:56:52.224  (GsqlHAHandler.java:283) switched to new leader
```

Monitor request history

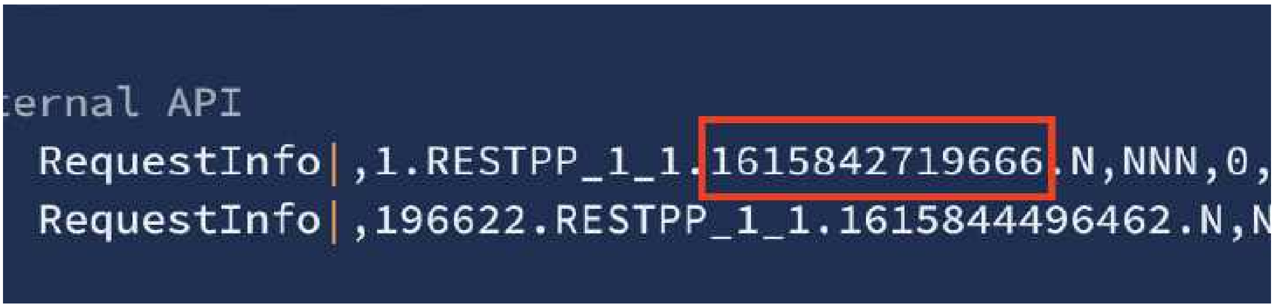
All requests made to TigerGraph's REST endpoints are recorded by the RESTPP logs and Nginx logs. Information available in the logs includes:

- Timestamp of the request
- API request parameters
- Request Status
- User information (when RESTPP authentication is turned on)

RESTPP is responsible for many tasks in the TigerGraph internal architecture and records many internal API calls, which can be hard to distinguish from manual requests. When RESTPP authentication is on, the RESTPP log will record the user information and mark a call if it is made by an internal API. Therefore, you can use the command below to filter for manual requests:

```
# In the restpp log directory
$ grep -i "requestinfo" log.INFO | grep -v "__INTERNAL_API__"

# All requests exluding the ones made by internal API
I0315 21:11:59.666318 14535 handler.cpp:351] RequestInfo|,1.RESTPP_1_1.16
I0315 21:41:36.462616 14541 handler.cpp:351] RequestInfo|,196622.RESTPP_1
```

`RequestInfo` contains the ID of the request, which you can use to look up more information on the request :



Request ID

Here is an example of using a request ID to look up a request in the restpp log:

```
$ grep "1615842719666" log.INFO

# Returns all information about the specific request
# RawRequest log is captured at the entry point of a query
I0315 21:11:59.666026 14535 handler.cpp:285] RawRequest|,1.RESTPP_1_1.161
# RequestInfo log is captured after the request has been parsed,
# and contains information such as username and the function or UDF to ru
I0315 21:11:59.666318 14535 handler.cpp:351] RequestInfo|,1.RESTPP_1_1.16
# ReturnResult is captured when the request has been processed
I0315 21:11:59.666509 14535 requestrecord.cpp:325] ReturnResult|0|0ms|RES
```

Monitor user management tasks

User management activities, such as logins, role and privilege changes are recorded in the GSQL logs in the folder `gsql`.

To view recent activities, use the symlink `log.INFO`. There is a lot of information in the logs - to filter for information that you need, you can use Linux commands such as `grep` ↗ and `tail` ↗ For example, to view recent changes in roles, you can run the following command in the `gsql` log directory:

```
$ grep -i "role" log.INFO

# Returns all lines containing the word "role"
#                        username     source IP
I@20210312 22:41:16.167 tigergraph|127.0.0.1:45854|00000000077 (BaseHandl
I@20210312 22:41:16.863 tigergraph|127.0.0.1:45854|00000000077 (BaseHandl
```

To view login activities, search `log.INFO` for `"login"` instead.

```
$ grep -i "login" log.INFO

# Returns all lines containing the world "login"
I@20210315 21:08:42.047 tigergraph|127.0.0.1:53960|00000000001 (BaseHandl
I@20210315 21:08:42.061 tigergraph|127.0.0.1:53960|00000000001 (LoginHand
is /home/tigergraph/tigergraph/log/restpp
I@20210315 21:08:42.072 tigergraph|127.0.0.1:53960|00000000001 (LoginHand
I@20210315 21:08:42.080 tigergraph|127.0.0.1:53960|00000000001 (BaseHandl
```

Set up Log Viewing with Elasticsearch, Kibana and Filebeat

The TigerGraph system produces extensive and detailed logs about each of its components. Starting with TigerGraph 3.2, TigerGraph provides a `gadmin` utility that allows users to easily view log files through an Elasticsearch, Kibana, and Filebeat setup. This page offers a step-by-step guide to set up log viewing for all components in a TigerGraph cluster with Elastic search, Kibana, and Filebeat.

Before you begin

- Install Elasticsearch ↗ on a machine that is running TigerGraph.
  - If you have a TigerGraph cluster, you only need to install Elasticsearch on one node.
- Install Kibana ↗ on the same machine where you installed Elasticsearch.
- Install Filebeat ↗.
  - If you have a TigerGraph cluster, you need to install Filebeat on all nodes in the cluster.

Procedure

1. Configure Elasticsearch for remote access

The default Elasticsearch settings only allow the Elasticsearch service to be accessed from the same machine it starts from. In order to allow Elasticsearch to receive log files from other servers in the cluster, we have to make the following edits to the file at `/etc/elasticsearch/elasticsearch.yml`

```
network.host: "<server_private_ip>"
discovery.seed_hosts: ["<server_private_ip>"]
# server_private_ip refers to the private ip address of the machine where
# elasticsearch is installed
cluster.initial_master_nodes: [ "node-1" ]
# "node-1" is the default name of the Elasticsearch node. If you changed
# the defualt name, you would use the name you chose instead
```

After editing the configurations, restart the Elasticsearch service.

Elasticsearch is a memory-intensive service. For more information on memory management for Elasticsearch, see Managing and Troubleshooting Elasticsearch Memory ↗.

2. Configure Kibana with Elasticsearch and enable remote access

You need to make the following changes to the file at `/etc/kibana/kibana.yml:`

- To allow remote access, change the value of `server.host` to the IP address or DNS name of the Kibana server. Since the Kibana server is on the same machine as Elasticsearch, this value should be the same private IP that you specified as Elasticsearch's `network.host`.
- Additionally, you need to provide the address of the Elasticsearch server in the `elasticsearch.hosts` setting. By default, Elasticsearch is on port 9200, so the value for this setting should be `["server_private_ip:9200"]`

After editing the configurations, restart the Kibana service.

3. Configure Filebeat

Finally, we need to configure Filebeat to have each component on each node send its logs to the Elasticsearch server. To do so, run the following gadmin command:

```
gadmin log build filebeat <--host={ip_address_1}[,{ip_address_n}]*>
```

The command outputs a Filebeat configuration file `filebeat.yml` . The following options are available:

Option	Description
`--host=<ip_list>`	Required. The list of IP addresses of the nodes whose logs you want to send to the Elasticsearch server. Example: `--host=10.128.0.97,10.128.0.99,10.128.0.100`
`--from-beginning`	Optional. If this flag is provided, Filebeat will harvest all log files including the oldest. If not included, Filebeat will only harvest the logs since the most recent time each service started.
`--path=<path_to_file>`	Optional. The path to output the configuration file. By default, the command outputs the configuration file `filebeat.yml` to the current directory.
`--service=<service_list>`	Optional. The services you want Filebeat to monitor. By default, all services are included. Example: --service=

After generating the `filebeat.yml` file, copy it to the directory `/etc/filebeat` on every node, and restart the Filebeat service on each node.

After the service restarts, you should be able to view the logs through Kibana's user interface in your browser at `server_ip:5601`.

Error Codes

The reference page for status codes on the TigerGraph platform.

This page documents the *status codes* and *exit codes* on the TigerGraph platform. Each status code follows the format: `<component>-<code>` , while exit codes are numeric values between 0 - 255.

If the system was handling a user request, the status code and message will be in the JSON response (see GSQL Query Output Format). For internal errors, the error information may be in a log file.

The GSQL Client will exit with a non-zero code if there's an error while handling a user request. To view the exit code of the GSQL client, run the command `echo $?` and the exit code of the most recent command will be printed to the terminal.

Return codes

REST

This section covers return codes from the REST++ server.

0000 - 0999

Codes in this range are success codes. When the conditions for multiple codes are true, the lowest code is returned.

Code	Description
`REST-0000`	General successful completion
`REST-0001`	Successful vertex insertion or update
`REST-0002`	Successful edge insertion or update

`REST-0003`	Successful vertex and edge insertion or update

1000 - 1999

RESTPP endpoint errors.

Code	Description
`REST-1000`	The endpoint does not exist.
`REST-1003`	The query could not run because there is no graph schema.
`REST-1004`	The graph cannot be found.

2000 - 2999

Payload errors.

Code	Description
`REST-2001`	The JSON payload is invalid.
`REST-2004`	The payload contains vertices of an invalid type.
`REST-2005`	The payload contains edges of an invalid type.

3000 - 3999

RESTPP general errors.

Code	Description
`REST-3000`	The query timed out from the timeout limit set by the request header.

`REST-3001`	The query timed out from the timeout limit set by the built-in endpoint.
	The query timed out from the timeout limit set by the

10000+

Other RESTPP errors.

Code	Description
`REST-10001`	The endpoint has been removed.
`REST-10004`	There was an exception when starting a scheduler.
`REST-10005`	There was an exception when processing results.
`REST-10015`	The request is in an orphan state due to malfunction in user-defined schedulers.
`REST-10016`	Access denied due to an invalid token.
`REST-12000`	RESTPP failed to get a response from GSE or GPE in time.
`REST-12001`	No running GSE or GPE instance detected.
`REST-12002`	The query was aborted by the user.
`REST-12003`	The query failed due to insufficient disk space.

GSQL

0000 - 0999

Codes in this range are success codes. GSQL will return the smallest code when the conditions are met for multiple codes.

Code	Description
`GSQL-0000`	The query is successful.
`GSQL-0001`	Vertices and edges were updated or inserted successfully.
`GSQL-0002`	Vertices were updated or inserted successfully.
`GSQL-0003`	Edges were updated or inserted successfully.

1000 - 1999

Query parameter errors.

Code	Description
`GSQL-1001`	The query contains a parameter with a `NULL` value.
`GSQL-1002`	General parameter error.
`GSQL-1020`	Invalid parameters.

2000 - 2499

JSON string related errors.

Code	Description
`GSQL-2010`	JSON object format error.
`GSQL-2011`	JSON array format error.

2500 - 2999

Operator errors.

Code	Description
`GSQL-2500`	Division by zero error.
`GSQL-2501`	The query contains incorrectly formatted `DATETIME` strings.
`GSQL-2502`	The query contains illegal patterns.
`GSQL-2503`	The query contains invalid operators.
`GSQL-2601`	The parameter provided is of the wrong primitive type.
`GSQL-2620`	The query contains an invalid array index.
`GSQL-2621`	The query contains an out-of-bounds array index.

3000 - 3999

Dynamic expression errors and expression function errors.

Code	Description
`GSQL-3000`	The query references non-existent attributes.
`GSQL-3001`	The query references non-existent vertex types.

4500 - 4999

Vertex type, edge type, and ID translation errors

Code	Description
`GSQL-4500`	The query references a vertex type that does not exist.
`GSQL-4501`	The query references an edge type that does not exist.

GSQL-4502	The query references an invalid vertex ID.
GSQL-4511	The query contains an invalid vertex attribute.
GSQL-4521	The query contains an invalid edge attribute.
GSQL-4522	The number of edge attributes is invalid.
GSQL-4523	An edge points from an invalid source vertex.
GSQL-4524	An edge points to an invalid target vertex.
GSQL-4525	An edge has both invalid source and target vertices

5000 - 5499

Print errors.

Code	Description
GSQL-5101	A file referenced in the query cannot be opened.
GSQL-5101	A file referenced in the query does not exist.
GSQL-5105	GSQL was denied access to a file referenced in the query.
GSQL-5111	A file referenced in the query cannot be read.

6000 - 6499

Errors related to updating the graph.

Code	Description

`GSQL-6000`	The query inserts an edge with an unknown vertex.

6500 - 6999

Code	Description
`GSQL-6500`	The query was aborted by the user.

7000 - 7999

Built-in query errors.

Code	Description
`GSQL-7002`	The query timed out.

8000 - 8999

Unexpected exceptions (C++).

Code	Description
`GSQL-8001`	Boost library exception
`GSQL-8002`	Runtime exception
`GSQL-8003`	`std` exception
`GSQL-8999`	All other exceptions

40000+

User-defined exception errors.

SYS

This section covers engine-related errors.

Code	Description
`SYS-0001`	The engine is not available.
`SYS-0002`	The query was rejected because the memory limit has been reached.
`SYS-0003`	The query is aborted.
`SYS-0004`	The endpoint has been removed.

GSQL client exit codes

The GSQL client will exit with non-zero code if there's an error while handling a user request. To check the exit code, run the Linux command `echo $?` and the exit code of the most recent command will be printed in the console.

Exit Code	**Description**
0	No error
41	Login or authentication error. The GSQL will also exit with this code if a graph with the supplied graph name cannot be found.
201	Invalid argument error
202	Connection error
203	Compatibility error
204	Session timeout
211*	Syntax error
212	Runtime error
213*	No graph in use error
255	Unknown error

*: The exit codes marked with a star (*) are only applicable when a GSQL script is given as an argument.

System Administration

TigerGraph Administrators Guide

Hardware and Software Requirements

This section provides an overview of the system requirements for running TigerGraph in a production or development environment.

TigerGraph can be used for different scopes and the system requirements largely depend on the use of the software. This page provides a good reference, but actual hardware requirements will vary based on your data size and workload.

Hardware Requirements

Component	Minimum	Recommended
CPU∗	4 cores for <500MB data, 8 cores for >500MB data (64-bit processor)	16+ cores (64-bit processor)
Memory*	8 GB	≥ 64GB
Storage*	20 GB	≥ 1TB, RAID10 volumes for better I/O throughput. SSD storage is recommended.
Network	1 Gigabit Ethernet adapter	10Gigabit Ethernet adapter for inter-node communication

∗Actual needs (CPU, memory, storage) depend on data size and application requirements. Consult our solution architects for an estimate of memory and storage needs.

Comments:

- The TigerGraph system is optimized to take advantage of multiple cores.
- Performance is optimal when the memory is large enough to store the full graph and to perform computations.

Certified Operating Systems

The TigerGraph Software Suite is built on 64-bit Linux. It can run on a variety of Linux 64-bit distributions. The software has been tested on the operating systems listed below:

	On-Premises hosting	Java JDK version	GCC version (C/C++)
RedHat 7.0 to 7.8 (x64)	Yes	1.8.0_141	4.8.2
RedHat 8.0 to 8.3 (x64)	Yes	1.8.0_141	4.8.2
Centos 7.0 to 7.4 (x64)	Yes	1.8.0_141	4.8.2
Centos 8.0 to 8.2 (x64)	Yes	1.8.0_141	4.8.2
Ubuntu 16.04 LTS Ubuntu 18.04 LTS Ubuntu 20.04 LTS (x64)	Yes	1.8.0_141	4.8.4
Debian 8 (jessie)	Yes	1.8.0_141	4.8.4

When a range of versions is given, it means that the software has been tested on the oldest and newest versions. We continually evaluate the operating systems on the market and work to update our set of supported operating systems as needed. The TigerGraph installer will install its own copies of Java JDK and GCC, accessible only to the TigerGraph user account, to avoid interfering with any other applications on the same server.

Prerequisite Software

Shell

Please use a bash shell for the installation process.

Utilities

Before offline installation, the TigerGraph system needs a few basic software packages to be present:

- `tar`
- `curl`
- `crontab`
- `ip`
- `ssh` / `sshd` (Only required for cluster installation)
- `more`
- `netstat`
- `sshpass`
  - if you intend to use password login method (P method) instead of ssh key login method (K method) to install the TigerGraph platform.

If they are not present, contact your system administrator to have them installed on your target system. For example, they can be installed with one of the following commands.

```
# Centos or RedHat:
sudo yum install tar curl cronie iproute util-linux-ng net-tools coreutil

# Ubuntu or Debian (Except Ver 18.04):
sudo apt install tar curl cron iproute util-linux net-tools coreutils ope

# Ubuntu or Debian (Ver. 18.04):
sudo apt install tar curl cron iproute2 util-linux net-tools coreutils op
```

NTP

If you are running TigerGraph on a multi-node cluster, you **must** install, configure and run the NTP (Network Time Protocol) daemon service. This service will synchronize system time among all cluster nodes.

Firewall

If you are running TigerGraph on a multi-node cluster, you **must** configure the iptables/firewall rules to make all TCP ports open among all cluster nodes.

Browser

In an on-premises installation, the system is fully functional without a web browser. To run the optional browser-based TigerGraph GraphStudio User Interface or Admin Portal, you need an appropriate browser:

Browser	Chrome	Safari	Firefox	Opera	Edge	Inte Exp
Supported version	54.0+	11.1+	59.0+	52.0+	80.0+	10+

Installation and Configuration

Installation, Cluster Configuration and Scale-out, License Activation

Installation Guide

Installing Single-machine and Multi-machine systems

This guide describes how to install the TigerGraph platform either as a single node or as a multi-node cluster, interactively or non-interactively.

> If you signed up for the Enterprise Free license, you also have access to the TigerGraph platform as a Docker image or a virtual machine (VirtualBox) image. Follow the instructions in Getting started to start up TigerGraph in a Docker container or with VirtualBox.

Preparation

Before you can install the TigerGraph system, you need the following:

- One or more machines that meet the minimum *Hardware and Software Requirements*.
- A sudo user with the same username and login credential on every machine.
  - If sudo privilege is not available, please contact TigerGraph support for workarounds.
- A license key provided by TigerGraph (not applicable to Enterprise Free)
- A TigerGraph system package
  - If you do not yet have a TigerGraph system package, you can request one at the following address: https://www.tigergraph.com/get-tigergraph ↗
- If you are installing a cluster, ensure that every machine has the same SSH port and the port stays open during installation

Installation

TigerGraph's installation script support both single-node and cluster installation, and the user can choose to install either interactively and non-interactively.

Interactive installation

The following describes the procedure to install TigerGraph on Linux interactively. The filename of your package may vary, depending on the product edition and version. For the examples here, we use the filename `tigergraph-<version>.tar.gz`, which should be replaced by the actual filename of your package.

Step 1: Extract the package

Extract the package by running the following command. A folder named `tigergraph-<version>-offline` will be created.

Example: extract for <version> = x.y.z

```
$ tar -xzf tigergraph-<version>.tar.gz
```

Step 2: Run installation script

Navigate to the `tigergraph-<version>-offline` folder and run script `install.sh` with the following commands:

Example: Default installation for <version> = 3.1.0

```
$ cd tigergraph-<version>-offline

# to install enterprise license
$ sudo ./install.sh

# For non-interactive installations, please see the non-interactive m
```

The installer will ask for the following information, for which you may choose to hit Enter to skip and use the system default or enter a new value:

- Your agreement to the License Terms and Conditions
- Your license key (not applicable for Enterprise Free)
- Username for the Linux user who will own and manage the TigerGraph platform
  - The installer creates a Linux user who is the only authorized user that can run `gadmin` commands to manage the TigerGraph Platform, and for whom this username is for
- Password for the Linux user who will own and manage the TigerGraph platform
- Path to where the installation folder will be
- Path to where the data folder will be
- Path to where the log folder will be
- Path to where the temp folder will be
- The SSH port for your machine

To see what the default settings are, read the Installation options section below.

> Since license keys are long – over 100 characters long. If you copy-and-paste the license key, be careful not to accidentally include an end-of-line character.

Step 3: Configure cluster settings

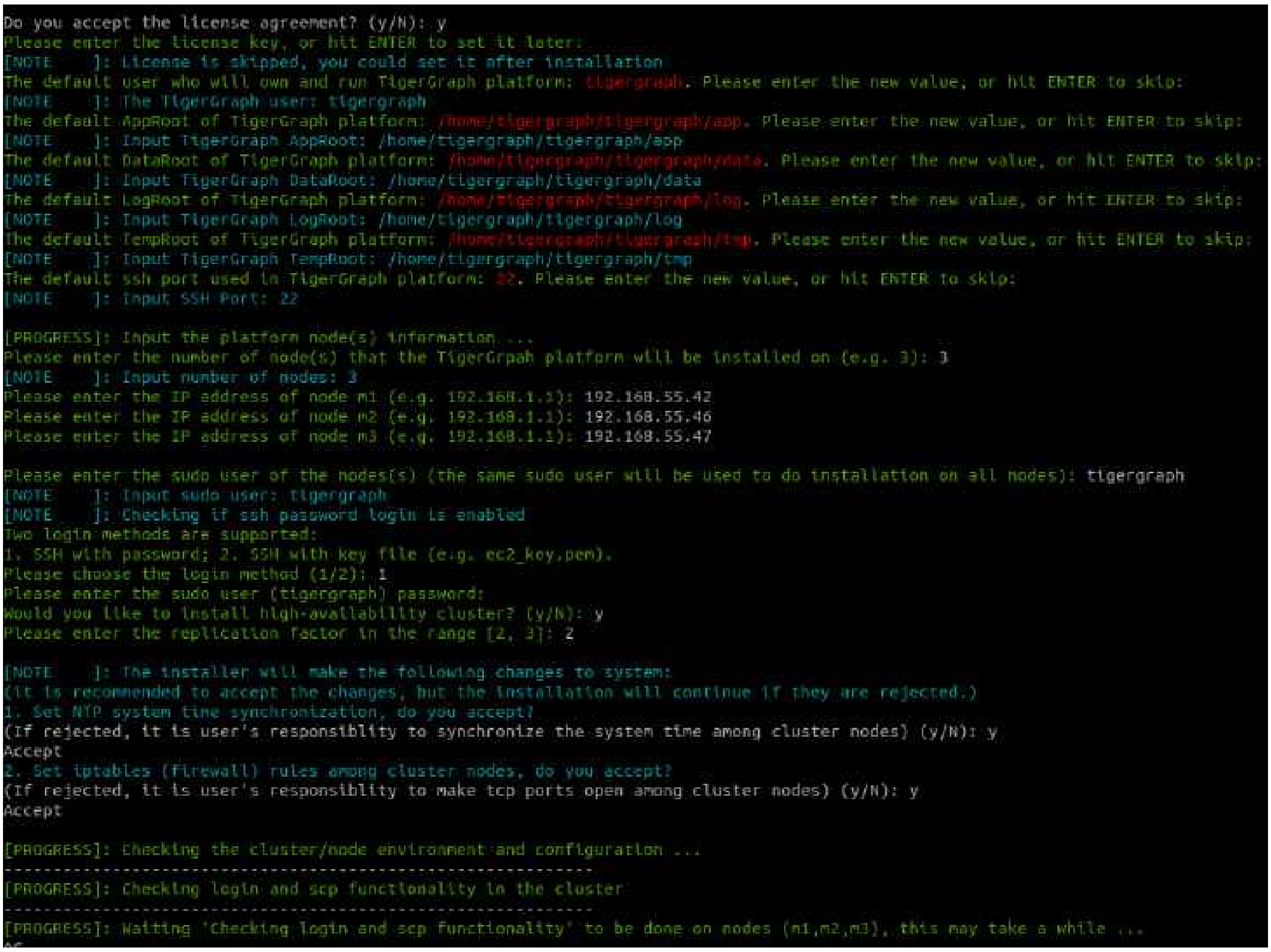
TigerGraph cluster configuration enables the graph database to be partitioned and distributed across multiple server nodes in a local network. After you have answered the questions described in the previous step, the installation script will ask for the following to complete cluster configuration:

- The number of nodes in your cluster. Each node will be given an alias following the input (`m1`, `m2`, `m3`, etc.)
  - If this is a single-node installation, enter `1`
- The IP address of each node
- Username and credentials information of the sudo user
  - Every machine in the cluster must have a sudo user with the same username and password or SSH key.
- Permission to set NPT time synchronization

- Permission to set firewall rules among the cluster nodes

> In TigerGraph 3.x, the installation machine can be within or outside the cluster. If outside the cluster, the installation machine still needs to be a Linux machine.

After all the questions are answered, the script will proceed to installation. A screenshot of the interactive installation is shown below:



Step 4: Verify installation

After installation is complete, you can switch to the Linux user who owns the platform (created in Step 2) with the following command :

```
$ su <username>  # default username: tigergraph
```

At the prompt, enter the password that was set in Step 2.

After switching to the correct user, you now have access to `gadmin` commands. Confirm successful installation by running `gadmin status`. If the system is installed correctly and the license is activated, the command should report that all services are up and ready. Since there is no graph data loaded yet, GSE and GPE will show `"Warmup"`.

```
[tigergraph@ip-172-31-18-9 ~]$ gadmin status
+--------------------+-------------------------+-------------------------+
|    Service Name    |     Service Status      |      Process State      |
+--------------------+-------------------------+-------------------------+
|       ADMIN        |         Online          |         Running         |
|        CTRL        |         Online          |         Running         |
|        DICT        |         Online          |         Running         |
|        ETCD        |         Online          |         Running         |
|        GPE         |         Warmup          |         Running         |
|        GSE         |         Warmup          |         Running         |
|        GSQL        |         Online          |         Running         |
|        GUI         |         Online          |         Running         |
|        IFM         |         Online          |         Running         |
|       KAFKA        |         Online          |         Running         |
|     KAFKACONN      |         Online          |         Running         |
|    KAFKASTRM-LL    |         Online          |         Running         |
|       NGINX        |         Online          |         Running         |
|       RESTPP       |         Online          |         Running         |
|        TS3         |         Online          |         Running         |
|      TS3SERV       |         Online          |         Running         |
|         ZK         |         Online          |         Running         |
+--------------------+-------------------------+-------------------------+
[tigergraph@ip-172-31-18-9 ~]$
```

Status of all components upon successful installation

Non-interactive installation

The following describes the procedure to install TigerGraph on Linux non-interactively.

Step 1: Extract the package

Extract the package by running the following command. A folder named `tigergraph-<version>-offline` will be created.

Example: extract for <version> = x.y.z

```
$ tar -xzf tigergraph-<version>.tar.gz
```

Step 2: Configure installation settings

Navigate to the `tigergraph-<version>-offline` folder. Inside the folder, there is a file named `install_conf.json`. For non-interactive mode installation, the user must review and modify all the settings in the file `install_conf.json` before running the installer.

Below is an example of the `install_conf.json` file:

```
{
  "BasicConfig": {
    "TigerGraph": {
      "Username": "tigergraph",
      "Password": "tigergraph",
      "SSHPort": 22,
      "PrivateKeyFile": "",
      "PublicKeyFile": ""
    },
    "RootDir": {
      "AppRoot": "/home/tigergraph/tigergraph/app",
      "DataRoot": "/home/tigergraph/tigergraph/data",
      "LogRoot": "/home/tigergraph/tigergraph/log",
      "TempRoot": "/home/tigergraph/tigergraph/tmp"
    },
    "License": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJJc3N1ZXIiOiJUaWdl
    "NodeList": [
      "m1: 127.0.0.1"
    ]
  },
  "AdvancedConfig": {
    "ClusterConfig": {
      "LoginConfig": {
        "SudoUser": "sudoUserName",
        "Method": "P[or K]",
        "P": "<sudo_user_password>",
        "K": "</path/to/my_key.pem_rsa>"
      },
      "ReplicationFactor": 1
    }
  }
}
```

Here is a description of all the fields in the config file:

- `"BasicConfig"`
  - `"TigerGraph"` : Information about the Linux user that will be created by the installer who owns and manages the TigerGraph platform.
    - `"Username"`: Username of the Linux user.
    - `"Password"`: Password of the Linux user.
    - `"SSHPort"`: Port number used to establish SSH connections.

        - `"PrivateKeyFile"` (optional): Absolute path to a valid private key file. If left empty, TigerGraph will generate one named `tigergraph.rsa` automatically.
        - `"PublicKeyFile"` (optional): Absolute path to a valid public key file. If left empty, TigerGraph will generate one named `tigergraph.pub` automatically.
    - `"RootDir"`
        - `"AppRoot"`: Absolute path to where application folder will be.
        - `"DataRoot"`": Absolute path to where the data folder will be.
        - `"LogRoot"`: Absolute path to where the log folder will be.
        - `"TempRoot"`: Absolute path to where the temp folder will be.
    - `"License"`: Your TigerGraph license string.
    - `"Node List"`: A JSON array of the nodes in the cluster. Each machine in the cluster is defined as a key-value pair, where the key is a machine alias (m1, m2, m3, etc) and the value is the IP address of the node.
- `"AdvancedConfig"`
    - `"ClusterConfig"`: Cluster configurations
        - `"LoginConfig"`: Login configurations
            - `"SudoUser"`: Username of the sudo user who will be used to execute the installation on all nodes.
            - `"Method"`: Authentication method for SSH. Enter `"P"` to use password authentication and `"K"` to use key-based authentication.
            - `"P"`: Password of the sudo user.
            - `"K"`: Absolute path to the SSH key to be used to authenticate the sudo user.
        - `"ReplicationFactor"`: [Replication factor]() of the cluster.
            - If you would like to enable the High Availability (HA) feature, please make sure you have at least 3 nodes in the cluster and set the replication factor to be greater than 1. For example, if your cluster has 6 nodes, you could set the replication factor to be 2 or 3. If you set the replication factor to be 2, then the partitioning factor will be 6 / 2 = 3. Therefore, 3 nodes will be used for one copy of the data, and the other 3 nodes will be used as a replica copy of the data.

- Ensure that the total number of nodes can be fully divided by the replication factor. Otherwise, some nodes may not be utilized as parts of the HA cluster.

Step 3: Run install script with `-n` option

Start the non-interactive installation process by running the `install.sh` script with the `-n` option:

```
$ sudo ./install.sh -n
```

Installation options

The following default settings will be applied if no parameters are specified:

- The installer will create a Linux user with username `tigergraph` and with password `tigergraph`. This user will be the only user authorized to run `gadmin` commands to manage the TigerGraph platform and services.
  - If there is already a user named `tigergraph`, this user will be designated as the platform owner and no other user will be created.
- The default root directory for the installation would be `/home/tigergraph/tigergraph` with the App/Data/Log/Temp files within it :
  App Path : `/home/tigergraph/tigergraph/app`
  Data Path : `/home/tigergraph/tigergraph/data`
  Log Path : `/home/tigergraph/tigergraph/log`
  Temp Path : `/home/tigergraph/tigergraph/tmp`
- The root directory for the installation (referred to as `<tigerGraph_root_dir>`) is a folder called `tigergraph` located in the tigergraph user's home directory, i.e., `/home/tigergraph/tigergraph`.

The installation can be customized by running command-line options with the `install.sh` script:

Command-line options

```
# Command-line options for install.sh
Usage:
./install.sh [-n] [-u <user>] [-p <password>] [-r <tigergraph_root_dir>]
./install.sh -h
Options:
  -h  --  Show help information
  -u  --  Username of Linux user to create [default: tigergraph]
  -p  --  Password of Linux user to create [default: tigergraph]
  -l  --  TigerGraph license key
  -n  --  Non-interactive option: suppress prompts, and install
          using default config
  -F  --  Set iptables (firewall) rules to open TCP ports among cluster n
  -N  --  Set NTP system time synchronization among cluster nodes

[NOTE    ]: Using option '-n' will non-interactively install the platform
            on single node or cluster with all configurations from
            config file "install_config.json". In this case, the config
            file should be modified before installation.

[WARNING ]: Installer will fail if any option (except -F and -N)
            is provided with option '-n' at the same time.
```

Upgrading an Existing Installation

This page describes the steps to upgrade an existing installation of TigerGraph to TigerGraph 3.2.

This page lists all the versions you can upgrade from through self-service. If you are trying to upgrade a production system to 3.2 from a different version than those listed on this page, please contact TigerGraph Support ↗ first.

Upgrading from v3.1.5/v3.1.6

TigerGraph 3.2.1 includes a critical patch to 3.2.0. Please upgrade to 3.2.1 instead of 3.2.0.

Please only upgrade to 3.2 from 3.1.5 or 3.1.6. If you are upgrading from another version, please upgrade to 3.1.6 first, and then upgrade from 3.1.6 to 3.3. For instructions on how to upgrade to 3.1.5/3.1.6 versions, see Advisory for upgrading to 3.1.5+ ↗.

Before you begin

- Use GBAR to perform a backup **and restore** on your existing installation. This step is necessary because TigerGraph 3.2 made major changes to the Graph Storage Engine and a restore is needed to remove certain files that would make the upgrade operation fail.
- After restoring the backup, ensure that there are no vertex inserts or deletions before starting the upgrade.

Procedure

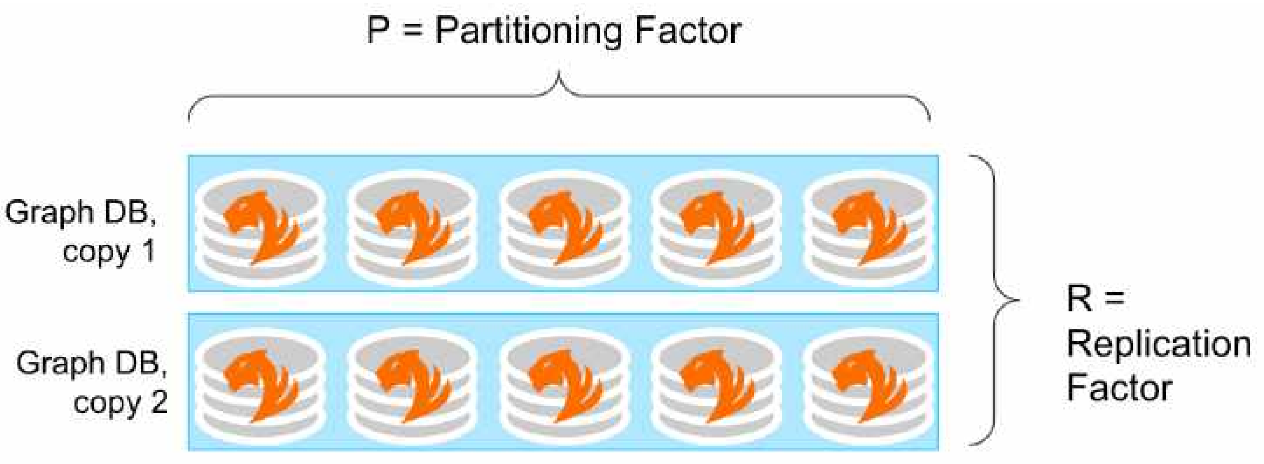
1. Download the 3.2.1 version of TigerGraph to your system.
2. Extract the tarball.

3. Run the install script under the Linux user created during installation with the upgrade flag (`-U`) that was extracted from the tarball:

```
$ ./install.sh -U
```

HA Cluster Configuration

A TigerGraph system with High Availability (HA) is a cluster of server machines which uses replication to provide continuous service when one or more servers are not available or when some service components fail. TigerGraph HA service provides load balancing when all components are operational, as well as automatic failover in the event of a service disruption. The **replication factor** is the number of copies of data. In contrast, the **partitioning factor** is the number of machines across which one copy of the database is distribution.



Replication factor vs. partitioning factor

If the replication factor is 2, a fully-functioning system maintains two copies of the data, stored on separate machines. Users can choose a higher replication factor for greater query throughput and greater system resiliency.

System Requirements

- The total cluster size should be (partitioning factor) X (replication factor).
- The smallest possible distributed database with HA is 2 × 2 = 4 machines.
- The smallest possible non-distributed database with HA is 1 × 3 = 3 machines.
- There is no upper limit for either partitioning factor or replication factor.
- The same version of the TigerGraph software package is installed on each machine.

Configuring HA

Starting from version 3.0, configuring a HA cluster is part of platform installation. See the Installation Guide page for details.

> HA configuration can only be done at the time of system installation and before deploying the system for database use. HA configuration change after installation is not supported. Converting a non-HA system to an HA cluster would require reinstalling all the TigerGraph components and rebuilding the database from the start.

During TigerGraph platform installation, specify the replication factor. The default value for replication factor is 1, which means there is no HA setup for the cluster. The user does not explicitly set the partitioning factor. Instead, the TigerGraph system will set

`partitioning factor = (number of machines / replication factor)`

If the division does not produce an integer, some machines will be left unused.

Example: If you install a 7-node cluster with replication factor = 2, the resulting configuration will be 2-way HA for a database with with a partitioning factor of 3. One machine will be unused.

Uninstallation

To uninstall TigerGraph, open the command line of the Linux server and switch to the TigerGraph user, which is created during installation:

```
$ su - <username>       # Default username is tigergraph
```

Then as the TigerGraph user, run the following Linux command:

```
$ guninstall
```

If you have the TigerGraph platform installed on a multi-node cluster, when running the `guninstall` command on a single node in the cluster, you will have the option to uninstall TigerGraph from all of the nodes in the cluster or just a single node.

High Availability Support for GSQL Server

Introduction:

By Design, TigerGraph has built-in HA for all the internal critical components from the beginning. This includes GPE, GSE, REST API Servers, etc. However, the user-facing applications (GSQL and GraphStudio) were designed to be set up by customers based on their High Availability (HA) needs. This included building solutions using non-TigerGraph components. With 3.1 release, TigerGraph will support native HA functionality for user-facing applications as well. This simplifies and streamlines HA deployment for users completely. For Operations personnel, this will reduce the operational overhead while enhancing the availability for end users.

Overview of the design:

Before we elaborate the design, we need to understand the topology of how TigerGraph services are deployed in a cluster. TigerGraph nodes in a cluster are organized as ‘m1’, ‘m2’, and so on. Although all nodes in the cluster serve the same function - store data and participate in query execution, m1 is a special node. GSQL server runs on this node to address critical services such as storing client metadata as well as managing connections between client and server. With this feature, m1 will no longer serve as the only node for GSQL server connections. In the new design, other nodes will run standby GSQL servers to provide high availability for client connections.

Role of Primary GSQL server

In the 3.1 release, primary GSQL server will continue to perform all the tasks handled by GSQL server prior to 3.1 release. This includes:

1. Process client connections
2. Querying requests from GSQL clients
3. User management requests including token management

In addition to these, when Primary fails, a standby server will switch to become the Primary server, and when the old Primary server is back to normal function, it will become a GSQL Standby server.

Role of Standby GSQL Servers

1. Redirect requests to Primary Server
2. Help Primary server to check for source data file existence and parse file header (if ANY is chosen)

Role of GSQL Client

There is no change in how GSQL Client works.

User Impact and Changes:

User Source Code Maintenance

Users store the following data on m1 node that is needed for query execution:

- GSQL loader's Token functions
- ExprFunctions
- ExprUtil

This is part of the user source code that TigerGraph system uses to compile. Prior to 3.1 release, this information is available to GSQL server only on m1 node. Typically, users can modify these files directly on the machine. But with HA, the Primary GSQL may not be in m1, and can be switched to any other machine at any time. Users have to make sure all the machines have the same content whenever there are updates to the files. This is a new requirement for users.

GSQL server will retrieve the User source code files in the following priority order when it needs them:

- Via github/github enterprise (if configuration is set),
- Files uploaded via PUT,
- Default files that are shipped with the product

User source code in github code repository

This requires public network access, or github enterprise server access. User need to provide the following gadmin configuration:

```
GSQL.GithubUserAcessToken # the credential, or "anonymous", empty means n
GSQL.GithubRepository     # e.g. tigergraph/ecosys
GSQL.GithubBranch         # optional, o/w use "master" branch, e.g. demo_
GSQL.GithubPath           # path to the directory in the github that has
GSQL.GithubUrl            # optional, used for github enterprise, e.g. ht
```

Example:

```
gadmin config set GSQL.GithubUserAcessToken anonymous
gadmin config set GSQL.GithubRepository tigergraph/ecosys
gadmin config set GSQL.GithubBranch demo_github
gadmin config set GSQL.GithubPath sample_code/src
gadmin config apply
```

When GSQL server needs to compile the files, it will retrieve them from github if the GitHub access is configured as above. It will retry 3 times, with timeout=5s for each time. If the connection fails, it will go to the next priority level method, i.e. file uploaded via PUT.

User Source code maintenance for local files in the cluster:

We are introducing new GSQL commands to address this need. These commands will allow users to upload and download the user source files.

Upload source code

```
PUT TokenBank FROM "path/to/a/file"
PUT ExprFunctions FROM "path/to/a/file"
PUT ExprUtil FROM "path/to/a/file"
```

Example:

```
temp_TokenBank=$tempDir/tmp_TokenBank.cpp
temp_ExprFunctions=$tempDir/tmp_ExprFunctions.hpp
temp_ExprUtil=$tempDir/tmp_ExprUtil.hpp

eval gsql 'PUT TokenBank FROM \"$temp_TokenBank\"'
eval gsql 'PUT ExprFunctions FROM \"$temp_ExprFunctions\"'
eval gsql 'PUT ExprUtil FROM \"$temp_ExprUtil\"'
```

Download source code

```
GET TokenBank TO "path/to/a/file"
GET ExprFunctions TO "path/to/a/file"
GET ExprUtil TO "path/to/a/file"
```

Example:

```
temp_TokenBank2=$tempDir/tmp_TokenBank_2.cpp
temp_ExprFunctions2=$tempDir/tmp_ExprFunctions_2.hpp
temp_ExprUtil2=$tempDir/tmp_ExprUtil_2.hpp

echo "GET TokenBank.cpp, ExprFunctions.hpp and ExprUtil.hpp to current no

eval gsql 'GET TokenBank TO \"$temp_TokenBank2\"'
eval gsql 'GET ExprFunctions TO \"$temp_ExprFunctions2\"'
eval gsql 'GET ExprUtil TO \"$temp_ExprUtil2\"'
```

The uploaded files will be saved to all nodes. Users will need to have either ‘superuser’ and ‘global_designer’ roles to have the sufficient privileges to run the PUT/GET commands.

When calling GET command, the user can download the corresponding file from the Primary node, to a local directory at the current cluster node.

When calling PUT command, the local file will be copied to all of the cluster nodes, including itself.

Example usage scenario to update of the files is as follows:

```
// Download the current file via GET, or create a new file from draft;
GET TokenBank TO "/myFolder/file.cpp"
// Upload the file via PUT
PUT TokenBank FROM "/myFolder/file.cpp"
```

For each cluster node, TokenBank.cpp is stored at:

```
 $(gadmin config get System.DataRoot)/gsql/tokenbank/
```

ExprFunctions.hpp and ExprUtil.hpp files are stored at:

```
 $(gadmin config get System.DataRoot)/gsql/udf/
```

Full path should be provided including the file name for PUT/GET, eg:

```
put ExprFunctions from "/home/path/tmp/ExprFunc.hpp"
get TokenBank to "doc/path/tmp/myTB.cpp"
```

Notice that in the first command, we use absolute path, while in the second command, we use relative path. Both are supported. But “~” is not supported (eg: “~/tmp/x.hpp”).

Additionally, users can also use the commands in the following manner as well:

- Use a folder name, and automatically default name will be added. For example:

```
put ExprFunctions from "/home/path/tmp"
```

It will use ExprFunctions.hpp under the directory "/home/path/tmp" for PUT.

```
get TokenBank to "home/path/tmp/"
```

It will create/overwrite the file "home/path/tmp/TokenBank.cpp".

If the file name is given in the path, its file extension must be consistent with the corresponding file. For example:

```
put ExprFunctions from "/home/path/tmp/test1.gsql"
```

is not allowed, since PUT/GET ExprFunctions must use ".hpp" as file extension.

Default file shipped with TigerGraph package

If the corresponding file is not found, the GSQL Primary server will use the default file in the package. These default files are at:

```
$(gadmin config get System.AppRoot)/dev/gdk/gsql/src/TokenBank/TokenBank.
$(gadmin config get System.AppRoot)/dev/gdk/gsql/src/QueryUdf/ExprUtil.hp
$(gadmin config get System.AppRoot)/dev/gdk/gsql/src/QueryUdf/ExprFunctio
```

File Path Configuration

In Pre-3.1 release design, the file path used in loading jobs refers to the file in m1, unless the user specifies machine name before the path (ALL, ANY, m1, m2,...). In the new HA design, the Primary server can be running on any machine, and can be switched. This means GSQL server may or may not find the file. To be back-compatible we prefix a machine name if the client is in TigerGraph cluster.

Users can specify the node ID before the path using: ALL, ANY, m1, m2 and so forth. Declaring ALL or ANY as host ID will load files from every cluster node.

User can use form like "m1|m3|m4" to declare the combination of several nodes.

If the hosts are not specified, it will look for the host ID of the current node that is running the loading job, (through searching the nodes in $(gadmin config get GSQL.BasicConfig.Nodes)). If not found, it will use node “m1” by default.

```
# current refers to /path/to/csv in m1
LOAD "/path/to/csv" TO VERTEX vt VALUES($0)
LOAD "ALL:/path/to/csv" TO VERTEX vt VALUES($0)
LOAD "m1|m2:/path/to/csv" TO VERTEX vt VALUES($0)
```

Data source can be created and used with a file path or a JSON string, same as above.

```
create data_source kafka k1 for graph poc_graph
set k1 = "/tmp/kafka_config.json"
create data_source kafka k2 = "/tmp/kafka_config.json"

CREATE LOADING JOB load_kafka FOR GRAPH poc_graph {
  DEFINE FILENAME f1 = "$k1:/tmp/topic_partition_config.json";
  LOAD f1
      TO VERTEX MyNode VALUES ($2)
      USING SEPARATOR="|";
}
```

GSQL Client connection setup:

GSQL client can connect to GSQL server in the different ways with the following priority order:

Using IP address:

Users can specify the ip and port when calling GSQL client using “gsql -i” or “gsql -ip”. For example:

```
gsql -ip 192.168.11.32:14240,192.168.11.34:14240,192.168.11.36
```

GSQL clients will try these ips and ports one by one. Notice the port is optional, it will use 14240 by default, which is the default port for GSQL server.

Using GSQL IP Configuration:

If “gsql -i” or “gsql -ip” are not used, GSQL client will search the file gsql_server_ip_config where the user runs the GSQL client. The file gsql_server_ip_config should be a one-line file such as shown below. GSQL client will traverse the ips and ports in the file in its order.

```
172.18.0.101,172.18.0.102:14240,172.18.0.103:14240
```

Similarly, the port number is also optional, using 14240 by default.

Using default local server:

If “gsql -i” or “gsql -ip” are not used, and the file gsql_server_ip_config does not exist where “gsql” is called, GSQL client will try to connect to the local server (127.0.0.1:8123).

Setting GSQL HA Configuration

Use gadmin config to get/set the following configurations related to GSQL High Availability.

The first is the heartbeat interval in milliseconds. The second (“max misses”) is the total timeout for switching to the Primary server which will measure the number of heartbeat intervals. It must be at least 2 to allow 1 heartbeat miss.

```
Controller.LeaderElectionHeartBeatIntervalMS = 2000
Controller.LeaderElectionHeartBeatMaxMiss = 4
```

For example, if we use “IntervalMS = 2000” and “max misses = 4” as shown above, then the total timeout is 2s×4 = 8 seconds. So the current Primary server will be switched if its heartbeat has stopped for more than 8 seconds.

High Availability Support for Application Server

Introduction

TigerGraph supports native HA functionality for its application server, which serves the APIs for TigerGraph's GUI - GraphStudio and Admin Portal. The application server follows the active-active architecture, in which **the server is always on m1 and all replicas of m1**. If one server falls offline, you can use the other servers without any loss of functionality.

When you deploy TigerGraph in a cluster with multiple replicas, it is ideal to set up load balancing to distribute network traffic evenly across the different servers. This page discusses what to do when a server dies when you haven't set up load balancing, and the steps needed to set up load balancing for the application server.

When a server dies

When a server dies, users can proceed to the next available server within the cluster to resume the operations. For example, assuming the TigerGraph cluster has Application Server on m1 and m2. If the server on m1 dies, users can access m2 to use GraphStudio and Admin Portal.

To find out which node hosts the application server, run the `gssh` command ↗ in the bash terminal of any active node in the cluster. The output will show you which nodes are hosting a GUI server.

Keep in mind that any long-running operation that is currently in process when the server dies will be lost.

Load Balancing

When you deploy TigerGraph in a cluster with multiple replicas, it is ideal to set up load balancing to distribute network traffic evenly across the different servers.

Set up load balancing with Nginx

One possible choice for setting up load balancing is through the use of Nginx.

Here is an example Nginx configuration for the upstream and server directives:

```
upstream flask_pool {
    ip_hash;
    zone flask_pool 64k;
    server 172.31.86.19:14240;
    server 172.31.88.70:14240;
    server 172.31.94.90:14240;

    keepalive 32;
}

server {
    listen      8000;
    server_name localhost;

    location / {
            root html;
            index index.html index.htm;
            proxy_pass http://flask_pool;
            proxy_read_timeout 3600;
            proxy_set_header Connection "";
            proxy_http_version 1.1;
            chunked_transfer_encoding off;
            proxy_buffering off;
            proxy_cache off;
    }
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
            root html;
    }
}
```

The server directives should specify your nodes' addresses which you want to load balance. Since TigerGraph requires session persistence, the load balancing methods will be limited to *ip_hash* or *hash*, unless you have access to Nginx Plus, which then means any load balancing method may be used with session persistence setup: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/#sticky ↗

An active health check can be set on the following endpoint if using Nginx Plus:

`/api/ping`

Otherwise, only a passive health check is available. See Nginx documentation for more information: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-health-check/ ↗

Set up AWS Elastic Load Balancer

If your applications are provisioned on AWS, another choice for load balancing is through the use of an Application Load Balancer ↗.

To create an application load balancer, follow AWS's guide to create an application load balancer ↗. The following configurations apply as you follow the guide:

Configure a security group

When creating or using an existing security group in Step 3, make sure it allows requests from the load balancer to port 14240 of the instances in the target group.

Health check URL

In Step 4, set the health check URL to `/api/ping`

Configure targets for the target group

In Step 5, enter 14240 for the port of your instances.

Enable sticky sessions

After following the steps and creating your load balancer, enable sticky sessions ↗ in your target group.

After successfully creating your load balancer, you should now be able to access GraphStudio through the load balancer's DNS name. The DNS name can be found under the "Description" tab of your load balancer in the Amazon EC2 console.

Set up Azure Application Gateway

If your instances are provisioned on Azure, you can set up an Application Gateway ↗.

Follow the steps for setting up an Application Gateway outlined here: Quickstart: Direct web traffic using the portal - Azure Application Gateway ↗

Some different TigerGraph specific settings are required during Application Gateway setup:

- Under the section “Configuration Tab”
  - For step 5, where it states to use port 80 for the backend port, use port 14240 instead.
  - In the same window, enable “Cookie-based affinity”.

Create a custom probe for Application Gateway

After the Application Gateway is complete, we need to create a custom health probe in order to check the health/status of our Application Servers. You can follow the following steps outlined here: Create a custom probe using the portal - Azure Application Gateway ↗

When filling out the health probe information, the fields below should have the following values:

Pick port from backend HTTP settings: yes

Path: `/api/ping`

HTTP Settings: The HTTP settings associated with the backend pool create during the Application Gateway setup

After successfully creating the Application Gateway, you should now be able to access GraphStudio from the frontend IP associated with the Application Gateway.

Set up GCP External HTTP(s) Load Balancer

If your instances are provisioned on Google Cloud, you can set up an External HTTP(s) Load Balancer ↗:

You can follow Google's provided steps in their documentation for setup here: Setting up an external HTTPS load balancer | Identity-Aware Proxy ↗

When creating the instance group ↗:

- Click "Specify port name mapping", and use 14240 for the port

When setting up the health check ↗:

- For the port, use 14240.
- For the path, use `/api/ping`.

Lastly, we need to set up session affinity for our load balancer. This is outlined in GCP documentation here: External HTTP(S) Load Balancing overview | Google Cloud ↗

After successfully creating the load balancer, you should now be able to access GraphStudio from the frontend IP associated with the load balancer.

Advanced License Issues

This guide covers two advanced license issues:

1. Activating a System-Specific License
2. Usage limits enforced by certain license keys

System-Specific License Activation

This section provides step-to-step instructions for activating or renewing a TigerGraph license, by generating and installing a license key unique to that TigerGraph system. This document applies to both non-distributed and distributed systems. In this document, a cluster acting cooperatively as one TigerGraph database is considered one system.

A valid license key activates the TigerGraph system for normal operation. A license key has a built-in expiration date and is valid on only one system. Some license keys may apply other restrictions, depending on your contract. Without a valid license key, a TigerGraph system can perform certain administrative functions, but database operations will not work.

To activate a new license, a user first configures their TigerGraph system. The user then collects the fingerprint of the TigerGraph system (license seed) using a TigerGraph-provided utility program. Then the collected materials are sent to TigerGraph or an authorized agent via email or web form. TigerGraph certifies the license based on the collected materials and sends a license key back to the user. The user then installs the license key on their system using another TigerGraph command. A new license key (e.g., one with a later expiration) can be installed on a live system that already has a valid license; the installation process does not disrupt database operations.

> ⓘ If your system is currently using an older string-based license key that does not use a license seed, please contact support@tigergraph.com ↗ for the procedure to upgrade to the new system-specific license type.

Step-by-Step Guide

> Before beginning the license activation process, the TigerGraph package must be installed on each server, and the TigerGraph system must be configured with gadmin.

1. Collect the fingerprint of the whole TigerGraph system using the command `gadmin license seed <host_signature_type>` , which can be executed on any machine in the system. The command packs all the collected data to generate the license seed and writes it into a file. When the command has completed successfully, it outputs the path of the file to the console.

   Depending on the host machine, the user needs to choose the appropriate type of host signature for `gadmin` to collect. The options are: `aws` . `azure` , `gcp` , `hardware,` and `node-id` . If you are generating the seed on a cloud instance, choose the corresponding cloud provider for the host signature type. If you are generating the seed on your own machines, choose either `hardware` or `node-id` . Signatures generated with the `hardware` parameter will use unique hardware information that persists through software changes, while signatures generated with `node-id` will use a unique [machine ID](#) that may change during an OS reinstall. Most users installing their own instances should use the `hardware` option.

   Collect Fingerprint of TigerGraph System

   ```
   $ gadmin license seed hardware
   [   Info] seed generated at: ./tigergraph_v3_seed, please send it to T
   ```

2. Send the license seed file to TigerGraph, either through our license activation web portal (preferred) or by email to [license@tigergraph.com.](mailto:license@tigergraph.com) If using email, please include the following information:
   a. Company/Organization name
   b. Contract number. If you do not know your contract number, please contact your sales representative or [sales@tigergraph.com.](mailto:sales@tigergraph.com)
3. A new license key file will be certificated and sent back to you.
4. Copy the license key file to a directory on the TigerGraph system where the TigerGraph Linux user has read permission.
5. Run the following three commands to install the license key:

```
$ gadmin license set <new_license_key>
[   Info] Configuration has been changed.
Please use 'gadmin config apply' to persist the changes.

$ gadmin config apply
[Warning] No difference from staging config, config apply is skipped.
[   Info] Successfully applied configuration change. Please restart se

$ gadmin restart
[   Note] Restart the service(s)? (y/N)y
[   Info] Stopping DICT ADMIN GSE NGINX GPE RESTPP KAFKASTRM-LL KAFKAC
[   Info] Starting ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKA
```

If the installation completes successfully, the message "install license successfully" will be displayed in the console.

Checking License Information

After a license key has been installed successfully on a TigerGraph system, the information of the installed license is available via either the CLI command `gadmin license status` or via the following REST API:

Get License Information

```
$ curl -X GET "localhost:9000/showlicenseinfo"
 {
   "message": "",
   "error": false,
   "version": {
     "edition": "developer",
     "schema": 0,
     "api": "v2",
   }
   "code": "",
   "results": [
     {
       "Days remaining": 10160,
       "Expiration date": "Mon Oct  2 04:00:00 2045\n"
     }
   ]
 }
```

Usage Limits Controlled by License Key

Some license keys include a limit on the graph size, or on the number and size of machines which may be used, or restrict the use of certain optional features. In the case of a memory usage or graph size limit, when a TigerGraph system reaches its license's limit, additional data will not be loaded into the graph. You may still query the graph and delete data. To check whether or not you have exceeded your license limits, use the command **gstatusgraph** and collect the VertexCount, EdgeCount, and Partition Size. Compare this information to the limits established for your license.

Checking graph limits

```
~/tigergraph/loadingData$ gstatusgraph
=== graph ===
[GRAPH  ] Graph was loaded (/home/tigergraph/tigergraph/data/gstore/0/par
[m1     ] Partition size: 23KiB, IDS size: 47KiB, Vertex count: 5, Edge c
[WARN   ] Above vertex and edge counts are for internal use which show ap
```

The output may include a warning message such such as the following:

```
[Warning] License limit exceeded. The system is running in limited capaci
```

Kubernetes

Users can deploy TigerGraph single servers and clusters in Kubernetes on Docker Desktop locally, or on any cloud provider of their choice.

> Known issue: TigerGraph 3.2 deployments on Kubernetes do not support backup or restore.

- Quickstart with Docker Desktop
- Quickstart with GKE
- Quickstart with EKS
- Quickstart with AKS

Quickstart with Docker Desktop

This guide provides instructions to deploy a single-server TigerGraph instance to a local Kubernetes cluster via Docker Desktop.

Before you begin

- Install Docker Desktop ↗ on your computer.
- Install the Kubernetes CLI `kubectl` ↗.
- Enable the Docker Desktop Kubernetes engine. For more information, see Docker official documentation ↗.
- Verify that you have sufficient CPU, RAM and DISK for your TigerGraph deployment.

Procedure

1. Download Kubernetes manifest

Download `tigergraph-docker.yaml` ↗. This is the file that describes your Kubernetes deployment.

2. Create deployment

Make sure that you have configured `kubectl` to use the correct context. In this case, the name of the context should be `docker-desktop`. Then run `kubectl apply` to create the deployment. Replace `<path_to_tigergraph.yaml>` with the path to the `tigergraph-docker.yaml` file you downloaded in the step before.

```
$ kubectl apply -f <path_to_tigergraph.yaml>
```

3. Verify deployment

Verify that the pods were created successfully:

```
$ kubectl get pods -l app=tigergraph -w
```

4. Test deployment

The deployment maps port 9000 of the pod to port 30090 of `localhost`. We can test that the REST API is up and running by making a curl call to port 30090:

```
$ curl http://localhost:30090/echo
{"error":false, "message":"Hello GSQL"}%
```

Port 14240 of the pod is mapped to port 30240 of `localhost`. Visit `http://localhost:30240` to access GraphStudio.

Cleanup

Run the following command to delete all cluster resources:

```
$ kubectl delete -f <path_to_tigergraph.yaml> &&  kubectl delete pvc -l a
```

Quickstart with GKE

This is a quick start guide for deploying TigerGraph on Kubernetes with Google Kubernetes Engine (GKE).

- Single-server deployment
- Cluster-deployment

Prerequisites

- The `gcloud` command-line interface (CLI) is installed ↗ on your machine.
- The `kubectl` Kubernetes client command-line tool ↗ is installed on your machine.
- A running GKE cluster with nodes that meet the minimum hardware and software requirements for running TigerGraph.
- You have configured cluster access for `kubectl` ↗.

Single-server deployment

This section describes the steps to deploy, verify, and remove a single-server deployment of TigerGraph on GKE.

Deploy single server

Step 1: Generate deployment manifest. Clone the TigerGraph ecosystems repository ↗ and change into the `k8s` directory. You can edit the `kustimization.yaml` file in the `gke` folder to change the namespace and image name for your deployment. The default namespace is `default`. No need to edit the files if no changes are needed.

Next, run the `./tg` script in the `k8s` directory to generate the deployment manifest for a single-server deployment. A `deploy` directory will be created automatically and you should find the manifest named `tigergraph-gke.yaml` in the directory.

```
$ ./tg gke kustomize -s 1
```

Step 2: Deploy manifest. Run `kubectl apply` to create the deployment using the manifest you generated in Step1.

```
$ kubectl apply -f deploy/tigergraph-gke.yaml
```

Verify single server

Run `kubectl get pods` to confirm that the pods were created successfully:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
installer-zsnb4     1/1     Running   0          4m11s
tigergraph-0        1/1     Running   0          4m10s
```

Run `kubectl describe service/tg-external-service` to find the IP address of the load balancer. You can then make curl calls to port 9000 to make sure that RESTPP is running:

```
$ curl <load_balancer_ip>:9000/echo | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
                                 Dload  Upload   Total   Spent    Left  S
100    39  100    39    0     0    120      0 --:--:-- --:--:-- --:--:--
{
  "error": false,
  "message": "Hello GSQL"
}
```

You can also copy the IP address into your browser and visit port 14240 to make sure that GraphStudio is working.

Connect to single server

You can use `kubectl` to get a shell to the container or log in via `ssh`

```
# Via kubectl
kubectl exec -it tigergraph-0 -- /bin/bash

# Via ssh
ip_m1=$(kubectl get pod -o wide |grep tigergraph-0| awk '{print $6}')
ssh tigergraph@ip_m1
```

To use GraphStudio, copy the IP address into your browser and visit port 14240.

Remove single server resources

Run the command below to delete all cluster resources:

```
$ kubectl delete -f deploy/tigergraph-gke.yaml && kubectl delete pvc -l a
```

Cluster deployment

Once your GKE cluster is ready, you can start following the below steps to deploy a TigerGraph cluster on Kubernetes.

Deploy TigerGraph cluster

1. Generate Kubernetes manifest

Clone the TigerGraph ecosystem repository ↗ and change into the `k8s` directory:

```
$ git clone https://github.com/tigergraph/ecosys.git
$ cd ecosys/k8s
```

You can customize your deployment by editing the `kustomize.yaml` file in the `gke` directory. The `tg` script in the `k8s` folder offers a convenient way to make common customizations such as namespace, TigerGraph version, as well as cluster size. Run `./tg -h` to view the help text on how to use the script.

Use the `tg` script in the `k8s` directory of the repo to create a Kubernetes manifest. Use `-s` or `--size` to indicate the number of nodes in the cluster. Use the `--ha` option to indicate the replication factor of the cluster, and the partitioning factor will be the number of nodes divided by the replication factor.

For example, the following command will create a manifest that will deploy a 3*2 cluster with a replication factor of 2 and a partitioning factor of 3. The `--version` flag specifies the version of TigerGraph to use for the deployment.

```
$ ./tg gke kustomize -s 6 --ha 2 --version 3.2.0
```

The command will create a directory named `deploy` with the manifest inside.

2. Deploy the cluster

Run `kubectl apply` to create the deployment

```
$ kubectl apply -f ./deploy/tigergraph-gke.yaml
```

Verify cluster

Run `kubectl get pods` to verify the pods were created successfully:

```
$ kubectl get pods
NAME              READY   STATUS    RESTARTS   AGE
installer-zsnb4   1/1     Running   0          4m11s
tigergraph-0      1/1     Running   0          4m10s
tigergraph-1      1/1     Running   0          75s
```

Run `kubectl describe service/tg-external-service` to find the IP address of the load balancer for your GKE cluster. You can make a curl call to port 9000 to make sure that RESTPP is working:

```
$ curl <load_balancer_ip>:9000/echo | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
                                 Dload  Upload   Total   Spent    Left  S
100    39  100    39    0     0    120      0 --:--:-- --:--:-- --:--:--
{
  "error": false,
  "message": "Hello GSQL"
}
```

You can also copy the IP address into your browser and visit port 14240 to make sure that GraphStudio is working.

Connect to instances

You can use `kubectl` to get a shell to the container or log in via `ssh`

```
# Via kubectl
kubectl exec -it tigergraph-0 -- /bin/bash

# Via ssh
ip_m1=$(kubectl get pod -o wide |grep tigergraph-0| awk '{print $6}')
ssh tigergraph@ip_m1
```

To use GraphStudio, copy the IP address into your browser and visit port 14240.

Delete cluster resources

Run the command below to delete all cluster resources:

```
$ kubectl delete -f deploy/tigergraph-gke.yaml && kubectl delete pvc -l a
```

Quickstart with EKS

This is a quickstart guide for deploying TigerGraph in Kubernetes on AWS's Elastic Kubernetes Service (EKS).

- Single-server deployment
- Cluster deployment

Before you begin

- Provision Kubernetes cluster on EKS with nodes that meet the hardware and software requirements to run TigerGraph.
- Install `kubectl` on your machine, ↗ and make sure your local `kubectl` version is within one minor version's difference from the `kubectl` version on your cluster.
- Configure `kubectl` for EKS ↗ cluster access.

Single-server deployment

This section describes the steps to deploy, verify, and remove a single-server deployment of TigerGraph on EKS.

Deploy single server

Step 1: Generate deployment manifest. Clone the TigerGraph ecosystems repository ↗ and change into the `k8s` directory. You can edit the `kustimization.yaml` file in the `eks` folder to change the namespace and image name for your deployment. The default namespace is `default`. No need to edit the files if no changes are needed.

Next, run the `./tg` script in the `k8s` directory to generate the deployment manifest for a single-server deployment. A `deploy` directory will be created automatically and you should find the manifest named `tigergraph-eks.yaml` in the directory.

```
./tg eks kustomize -s 1
```

Step 2: Deploy manifest. Run `kubectl apply` to create the deployment using the manifest you generated in step1.

```
kubectl apply -f deploy/tigergraph-eks.yaml
```

Verify single server

After you create the deployment, run `kubectl get pods` to verify that the pods were created successfully.

```
$ kubectl get pods
NAME              READY   STATUS    RESTARTS   AGE
installer-zsnb4   1/1     Running   0          4m11s
tigergraph-0      1/1     Running   0          4m10s
```

Run `kubectl describe service/tg-external-service` to find the IP address of the load balancer. You can then make curl calls to port 9000 to make sure that RESTPP is running:

```
$ curl <load_balancer_ip>:9000/echo | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
                                 Dload  Upload   Total   Spent    Left  S
100    39  100    39    0     0    120      0 --:--:-- --:--:-- --:--:--
{
  "error": false,
  "message": "Hello GSQL"
}
```

You can also copy the IP address into your browser and visit port 14240 to make sure that GraphStudio is working.

Connect to single server

You can use `kubectl` to get a shell to the container or log in via `ssh`

```
# Via kubectl
kubectl exec -it tigergraph-0 -- /bin/bash

# Via ssh
ip_m1=$(kubectl get pod -o wide |grep tigergraph-0| awk '{print $6}')
ssh tigergraph@ip_m1
```

To use GraphStudio, copy the IP address into your browser and visit port 14240.

Remove single server resources

Run the command below to delete all cluster resources:

```
$ kubectl delete -f deploy/tigergraph-eks.yaml && kubectl delete pvc -l a
```

Cluster deployment

This section describes the steps to deploy, verify, and remove a TigerGraph cluster in Kubernetes on EKS.

Deploy TigerGraph cluster

1. Generate Kubernetes manifest

Clone the TigerGraph ecosystem repository and change into the `k8s` directory:

```
$ git clone https://github.com/tigergraph/ecosys.git
$ cd ecosys/k8s
```

You can customize your deployment by editing the `kustomize.yaml` file in the `aks` directory. The `tg` script in the `k8s` folder offers a convenient way to make common customizations such as namespace, TigerGraph version, as well as cluster size. Run `./tg -h` to view the help text on how to use the script.

Use the `tg` script at in the `k8s` directory of the repo to create a Kubernetes manifest. Use `-s` or `--size` to indicate the number of nodes in the cluster. Use the `--ha` option to indicate the replication factor of the cluster; the partitioning factor will be the number of nodes divided by the replication factor.

For example, the following command will create a manifest that will deploy a 3*2 cluster with a replication factor of 2 and a partitioning factor of 3. The `--version` flag specifies the version of TigerGraph to use for the deployment.

```
$ ./tg eks kustomize -s 6 --ha 2 --version 3.2.0
```

The command will create a directory named `deploy` with the manifest inside.

2. Deploy the cluster

Run `kubectl apply` to create the deployment

```
$ kubectl apply -f ./deploy/tigergraph-eks.yaml
```

Verify cluster

Run `kubectl get pods` to verify the pods were created successfully:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
installer-zsnb4     1/1     Running   0          4m11s
tigergraph-0        1/1     Running   0          4m10s
tigergraph-1        1/1     Running   0          75s
```

Run `kubectl describe service/tg-external-service` to find the IP address of the load balancer for your EKS cluster. You can make a curl call to port 9000 to make sure that RESTPP is working:

```
$ curl <load_balancer_ip>:9000/echo | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
                                 Dload  Upload   Total   Spent    Left  S
100    39  100    39    0     0    120      0 --:--:-- --:--:-- --:--:--
{
  "error": false,
  "message": "Hello GSQL"
}
```

You can also copy the IP address into your browser and visit port 14240 to make sure that GraphStudio is working.

Connect to instances

You can use `kubectl` to get a shell to the container or log in via `ssh`

```
# Via kubectl
kubectl exec -it tigergraph-0 -- /bin/bash

# Via ssh
ip_m1=$(kubectl get pod -o wide |grep tigergraph-0| awk '{print $6}')
ssh tigergraph@ip_m1
```

To use GraphStudio, copy the IP address into your browser and visit port 14240.

Delete cluster resources

Run the command below to delete all cluster resources:

```
$ kubectl delete -f deploy/tigergraph-eks.yaml && kubectl delete pvc -l a
```

Quickstart with AKS

This is a quickstart guide for deploying TigerGraph single servers and clusters in Kubernetes on Azure Kubernetes Service(AKS).

- Single-server deployment
- Cluster-deployment

Before you begin

- Provision Kubernetes cluster on AKS with nodes that meet the hardware and software requirements to run TigerGraph.
- Install `kubectl` on your machine, ↗ and make sure your local `kubectl` version is within one minor version's difference from the `kubectl` version on your cluster.
- Configure `kubectl` for AKS cluster access ↗.

Single-server deployment

This section describes the steps to deploy, verify, and remove a single-server deployment of TigerGraph on AKS.

Deploy single server

Step 1: Generate deployment manifest. Clone the TigerGraph ecosystems repository ↗ and change into the `k8s` directory. You can edit the `kustimization.yaml` file in the `aks` folder to change the namespace and image name for your deployment. The default namespace is `default`, and the default image is the official docker image for TigerGraph 3.2.0. No need to edit the files if no changes are needed.

Next, run the `./tg` script in the `k8s` directory to generate the deployment manifest for a single-server deployment. A `deploy` directory will be created automatically and you should find the manifest named `tigergraph-aks.yaml` in the directory.

```
./tg aks kustomize -s 1
```

Step 2: Deploy manifest. Run `kubectl apply` to create the deployment using the manifest you generated in step 1.

```
kubectl apply -f deploy/tigergraph-aks.yaml
```

Verify single server

After you create the deployment, run `kubectl get pods` to verify that the pods were created successfully.

```
$ kubectl get pods
NAME                 READY    STATUS     RESTARTS    AGE
installer-zsnb4      1/1      Running    0           4m11s
tigergraph-0         1/1      Running    0           4m10s
```

Run `kubectl describe service/tg-external-service` to find the IP address of the load balancer. You can then make curl calls to port 9000 to make sure that RESTPP is running:

```
$ curl <load_balancer_ip>:9000/echo | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
                                 Dload  Upload   Total   Spent    Left  S
100    39  100    39    0     0    120      0 --:--:-- --:--:-- --:--:--
{
  "error": false,
  "message": "Hello GSQL"
}
```

You can also copy the IP address into your browser and visit port 14240 to make sure that GraphStudio is working.

Connect to single server

You can use `kubectl` to get a shell to the container or log in via `ssh`

```
# Via kubectl
kubectl exec -it tigergraph-0 -- /bin/bash

# Via ssh
ip_m1=$(kubectl get pod -o wide |grep tigergraph-0| awk '{print $6}')
ssh tigergraph@ip_m1
```

To use GraphStudio, copy the IP address into your browser and visit port 14240.

Remove single server resources

Run the command below to delete all cluster resources:

```
$ kubectl delete -f deploy/tigergraph-aks.yaml && kubectl delete pvc -l a
```

Cluster deployment

This section describes the steps to deploy, verify, and delete a TigerGraph cluster in Kubernetes on AKS.

Deploy TigerGraph cluster

1. Generate Kubernetes manifest

Clone the [TigerGraph ecosystem repository](https://github.com/tigergraph/ecosys.git) ↗ and change into the `k8s` directory:

```
$ git clone https://github.com/tigergraph/ecosys.git
$ cd ecosys/k8s
```

You can customize your deployment by editing the `kustomize.yaml` file in the `aks` directory. The `tg` script in the `k8s` folder offers a convenient way to make common customizations such as namespace, TigerGraph version, as well as cluster size. Run `./tg -h` to view the help text on how to use the script.

Use the `tg` script in the `k8s` directory of the repo to create a Kubernetes manifest. Use `-s` or `--size` to indicate the number of nodes in the cluster. Use the `--ha` option to indicate the replication factor of the cluster, and the partitioning factor will be the number of nodes divided by the replication factor.

For example, the following command will create a manifest that will deploy a 3*2 cluster with a replication factor of 2 and a partitioning factor of 3. The `--version` flag specifies the version of TigerGraph to use for the deployment.

```
$ ./tg aks kustomize -s 6 --ha 2 --version 3.2.0
```

The command will create a directory named `deploy` with the manifest inside.

2. Deploy the cluster

Run `kubectl apply` to create the deployment

```
$ kubectl apply -f ./deploy/tigergraph-aks.yaml
```

Verify cluster

Run `kubectl get pods` to verify the pods were created successfully:

```
kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
installer-zsnb4     1/1     Running   0          4m11s
tigergraph-0        1/1     Running   0          4m10s
tigergraph-1        1/1     Running   0          75s
```

Run `kubectl describe service/tg-external-service` to find the IP address of the load balancer for your AKS cluster. You can make a curl call to port 9000 to make sure that RESTPP is working:

```
$ curl <load_balancer_ip>:9000/echo | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
                                 Dload  Upload   Total   Spent    Left  S
100    39  100    39    0     0    120      0 --:--:-- --:--:-- --:--:--
{
  "error": false,
  "message": "Hello GSQL"
}
```

You can also copy the IP address into your browser and visit port 14240 to make sure that GraphStudio is working.

Connect to instances

You can use `kubectl` to get a shell to the container or log in via `ssh`

```
# Via kubectl
kubectl exec -it tigergraph-0 -- /bin/bash

# Via ssh
ip_m1=$(kubectl get pod -o wide |grep tigergraph-0| awk '{print $6}')
ssh tigergraph@ip_m1
```

To use GraphStudio, copy the IP address into your browser and visit port 14240.

Delete cluster resources

Run the command below to delete all cluster resources:

```
$ kubectl delete -f deploy/tigergraph-aks.yaml && kubectl delete pvc -l a
```

User Access Management

User Privileges and Authentication, LDAP, Single Sign-on

TigerGraph's user access management features are based on the Role-based Access Control model, on top of which there are additional features that give users finer-grain data access control. Some key features include:

- Authentication
- User management
- Role management
- Vertex-level access control (VLAC)
- Lightweight Directory Access Protocol (LDAP)
- Single sign-on

Users and Credentials

A TigerGraph user is a database-level security principal on the TigerGraph platform. When user authentication is enabled, only clients who can provide credentials that identify themselves as a user can interact with the TigerGraph database.

The TigerGraph platform offers two options for credentials:

1. A username-password pair
2. A token - a unique 32-character string used for REST++ requests, with an expiration date.

Enabling User Authentication

Enabling user authentication on TigerGraph enforces access control, requiring users to identify themselves and ensuring that users can only perform actions allowed by their roles.

When the TigerGraph platform is first installed, user authentication is disabled. The installation process creates a gsql superuser who has the name `tigergraph` and password `tigergraph`. As long as this user's password is `tigergraph`, GSQL authentication remains disabled.

Because there are two ways to access the TigerGraph system, either through the GSQL shell or through REST++ requests, there are two steps needed to secure your system with authentication enabled for both points of entry:

- Enable user authentication for the GSQL Shell
- Enable OAuth 2 authentication for REST++

Enable GSQL authentication

To enable user authentication for GSQL, change the password of the default user whose username `tigergraph` to something other than `tigergraph`.

Procedure

1. Log in to the GSQL shell as the default user `tigergraph`. Since authentication is not enabled, entering `gsql` into the Linux terminal under the TigerGraph Linux user will log you in as user `tigergraph` automatically.
2. Run the following command to change the password, and enter the new password as prompted:
```
GSQL > ALTER PASSWORD
```
3. User authentication has been enabled. Exit the GSQL shell and try to reenter, and confirm that GSQL will now prompt for your password.

```
$ gsql
Password for tigergraph : ********
```

4. To log in as a different user, use the `-u` option when you enter the GSQL shell. You can also supply the password in the same command with the `-p` option.

```
$ gsql -u newuser -p mypassword
```

Enable RESTPP authentication

To enable RESTPP authentication, set the `RESTPP.Factory.EnableAuth` parameter to `true`.

Procedure

1. As the TigerGraph Linux user, run the following command:

Enabling REST++ OAuth Authentication

```
$ gadmin config set RESTPP.Factory.EnableAuth true
```

2. Run the following commands to save the configuration and restart system services:

Enabling REST++ OAuth Authentication

```
$ gadmin config apply
$ gadmin restart restpp nginx gui gsql -y
```

Managing Credentials

When user authentication is enabled, the TigerGraph system will execute a requested operation only if the requester provides credentials for a user who has the privilege to perform the requested operation.

The TigerGraph system offers two options for credentials.

- Username and password pair.
- API token: A unique 32-character string that can be used for REST++ requests. A token expires 1 month from the date of creation by default. Users can use their secrets or their username and password pair to generate a token.

The following set of commands are used to create and manage passwords and secrets.

GSQL Commands for Managing Credentials

```
ALTER PASSWORD [user1]
CREATE SECRET [alias1]
SHOW SECRET
DROP SECRET <secret1>
```

Like any other GSQL commands, the user must supply credentials to run these commands. In order to create a secret, the user must supply their password.

Passwords

Users can change their own passwords with the `ALTER PASSWORD` command. If the user has the `WRITE_USER` privilege, they can change the password of another user.

```
ALTER PASSWORD [<user1>]
```

If a username is not provided, the command changes the password of the current user. To change the password of another user, specify the username of the user whose password you wish to change:

Example: User changing his/her own password

```
herminone:GSQL > ALTER PASSWORD
Password: *******
New Password : ************
Re-enter Password : ************
Password has been changed.
```

Example: Admin changing another user's password

```
tigergraph:GSQL > ALTER PASSWORD hermione
Password: *******
New Password : ************
Re-enter Password : ************
Password has been changed.
```

Secrets

Secrets are unique strings that serve as a user's credentials in certain circumstances. A user can have multiple secret strings. Each secret is associated with one user and their role for one graph. If the role is revoked, the secret also becomes invalid.

Create a secret

```
CREATE SECRET [<alias>]
```

Use the `CREATE SECRET` command to generate a secret for the current user and graph. It is optional to provide an alias for the secret.

Beginning with TigerGraph 3.1.4, the system will generate a random alias for the secret if the user does not provide an alias for that secret. Randomly generated aliases begin with `AUTO_GENERATED_ALIAS_` and include a random 7-character string.

List secrets

```
SHOW SECRET
```

Use `SHOW SECRET` to list all secrets of the current user. The secrets will be masked and only the first and last three characters of the secrets will be shown. The alias of the secret and the graph that the secret is associated with will also be listed:

```
GSQL > SHOW SECRET
    - Secret: s7s****3k5
      - Alias: HH
      - GraphName: Hogwarts
    - Secret: 75j****9i2
      - Alias: LL
      - GraphName: London
```

Drop a secret

```
DROP SECRET <secret>
```

Use the `DROP SECRET` command to drop a secret. Since a user can have multiple secrets, the secret to drop must be specified in the command. You can specify a secret either by the secret string itself or by its alias.

Roles and Privileges

TigerGraph uses Role-based Access Control (RBAC) to manage authorization. On every graph, privileges to perform actions are assigned to roles, and roles are granted to users. Outside of the permissions granted by their roles, a user has no access to the system.

Privileges

Privilege is permission to perform an action in a given scope. When a privilege is assigned to a role, it allows users with the role to perform the specified action in the specified scope. For example, the privilege `READ_SCHEMA` on graph `social` gives a user read permission to the schema of the graph `social`. This allows the user to run commands such as `ls` and `SHOW VERTEX` on the graph.

To view a complete list of privileges available in TigerGraph and the commands they enable a user to run, see List of Privileges.

Global vs local privileges

The scope of a privilege can be global or local. Global privileges apply to all graphs and global objects. Local privileges only apply on the graph they belong to.

For example, a role with `WRITE_QUERY` on graph `social` can only create queries on graph `social`, but not on other graphs. In contrast, a role with `WRITE_QUERY` on the global scope can create queries on all graphs.

Local roles can only be granted local privileges, while global roles can be granted both local and global privileges.

Global-only privileges

Some privileges can only be global by nature. For example, since users are global objects, any user-related privileges are global only. To see which privileges are global-only, see List of Privileges ↗.

Roles

A role is a collection of privileges you can assign to users to grant them permission to perform actions on specific resources.

Global vs local roles

Roles can be global or local. Local roles can only be granted local privileges, while global roles can be granted both local and global privileges.

For example, if a user creates a role `manager` on the graph `social`:

```
GSQL > CREATE ROLE manager ON GRAPH social
Successfully created roles: [manager].
```

This role can only be granted privileges on the graph `social`. It cannot be granted global privileges.

Built-in roles

GSQL offers 5 built-in local roles and 2 built-in global roles. The built-in roles cannot be dropped. Below is a table of the built-in roles and their corresponding set of privileges.

Name	Global or Local	Privilege List
`observer`	Local	`READ_SCHEMA`, `READ_LOADINGJOB`, `READ_QUERY`
`queryreader`	Local	`READ_SCHEMA`, `READ_LOADINGJOB`, `READ_QU`

`querywriter`	Local	`ERY`, `EXECUTE_LOADINGJOB`, `READ_SCHEMA`, `READ_LOADINGJOB`, `READ_QUERY`, `EXECUTE_LOADINGJOB`, `READ_DATA`, `WRITE_QUERY`, `WRITE_DATA`
`designer`	Local	`READ_SCHEMA`, `READ_LOADINGJOB`, `READ_QUERY`, `EXECUTE_LOADINGJOB`, `READ_DATA`, `WRITE_QUERY`, `WRITE_DATA`, `WRITE_SCHEMA`, `WRITE_LOADINGJOB`, `ACCESS_TAG`
`admin`	Local	`READ_SCHEMA`, `READ_LOADINGJOB`, `READ_QUERY`, `EXECUTE_LOADINGJOB`, `READ_DATA`, `WRITE_QUERY`, `WRITE_DATA`, `WRITE_SCHEMA`, `WRITE_LOADINGJOB`, `ACCESS_TAG` `WRITE_ROLE`, `WRITE_DATASOURCE`, `READ_ROLE`, `READ_USER`, `READ_PROXYGROUP`
`globaldesigner`	Global	Designer's privileges on global + drop graph created by the same user
`superuser`	Global	All supported privileges

User-defined roles

Users can define roles with their own list of privileges they want to grant to the role. To learn how to create/drop user-defined roles and manage privileges for the roles, see Role Management.

Role Management

This page explains the procedures for various role management tasks under TigerGraph's authorization model.

Create a local role

Syntax

```
CREATE ROLE <role_name> (, <role_name>)* [ON GRAPH <graph_name>]
```

Required privilege

`WRITE_ROLE`

Procedure

1. To create a local role, run the `CREATE ROLE` command like below. If you choose not to specify a graph in the command, the current scope will be used as the scope of the role:

   ```
   GSQL > USE GRAPH example_graph
   GSQL > CREATE ROLE role1, role2
   ```

This will create two roles named `role1` and `role2` on graph `example_graph`. By default, these two roles do not have any privilege:

```
Successfully created local roles for graph 'example_graph': [role1, role2
```

Create a global role

Syntax

```
CREATE ROLE <role_name> (, <role_name>)* ON GLOBAL
```

Required privilege

`WRITE_ROLE` on the global scope

Procedure

1. To create a global role, run the `CREATE ROLE` command like below. Replace `role1` with the name of the role you are creating.

```
CREATE ROLE role1 ON GLOBAL
```

This will create a role named `role1` on the global scope. By default, this role has no privileges:

```
Successfully created global roles: [role1].
```

View privileges of a role

Syntax

```
SHOW PRIVILEGE ON ROLE <role_name> (, <role_name2>)*
```

Required privilege

`READ_ROLE`

Procedure

1. To view the privileges of a role, run the `SHOW PRIVILEGE ON ROLE` command, and replace `role1, role2` with the names of the roles whose privileges you want to view:

```
GSQL > SHOW PRIVILEGE ON ROLE role1 , role2
```

This will show the privileges of the role `role1` and `role2:`

```
Role: "role1"
  - Graph 'tpc_graph' Privileges:
    WRITE_QUERY

Role: "role2"
This role has no privilege.
```

List all existing roles

Syntax

```
SHOW ROLE
```

Required privilege

`READ_ROLE`

Procedure

1. To list all existing roles, first ensure that you are in the correct scope. Run `USE <graph_name>` or `USE GLOBAL` to switch to your desired scope.
2. Run the `SHOW ROLE` command:

   ```
   GSQL > SHOW ROLE
   ```

This will show all the roles in your current scope:

```
    - Builtin Roles:
      observer
      queryreader
      querywriter
      designer
      admin
      globaldesigner
      superuser

    - User Defined Roles:
      - Graph 'tpc_graph' Roles:
        role1
        role2
```

Grant privileges to a role

Syntax

```
GRANT PRIVILEGE <privilege_name1> (, privilege_name2)*
        [ON GRAPH <graph_name>] TO <role_name1> (, <role_name2>)*
```

Require privilege

`WRITE_ROLE`

Procedure

1. To grant privileges to a role, run the `GRANT PRIVILEGE` command from the GSQL shell:

   ```
   GSQL > GRANT PRIVILEGE WRITE_QUERY, WRITE_ROLE
           ON GRAPH example_graph TO role1 , role2
   ```

This will allow users with the roles `role1` and `role2` to edit and install queries, as well as modify roles on the graph `example_graph`. To see a full list of privileges and the command they allow users to run, see List of Privileges.

Revoke privileges from a role

Syntax

```
REVOKE PRIVILEGE <privilege_name1> (, privilege_name2)*
        [ON GRAPH <graph_name>] FROM <role_name1> (, <role_name2>)*
```

Required privilege

`WRITE_ROLE`

Procedure

1. To revoke privileges from a role, run the `REVOKE PRIVILEGE` command from the GSQL shell:

```
GSQL > REVOKE PRIVILEGE WRITE_QUERY
        ON GRAPH example_graph FROM role1
```

This will revoke the `WRITE_QUERY` privilege from the role `role1` on graph `example_graph.`

Drop a role

Syntax

```
DROP ROLE <role_name> (, <role_name2>)*
```

Required privilege

`WRITE_ROLE`

Procedure

1. To drop a role, run the `DROP ROLE` command from the GSQL shell:

```
GSQL > DROP ROLE role1 , role2
```

This will drop the roles `role1` and `role2`. This will also revoke the roles from users who have been granted these roles.

User Management

This page explains the procedures for various user management tasks under TigerGraph's authorization model.

Create a user

Syntax

```
CREATE USER
```

Required privilege

`WRITE_USER`

Procedure

1. From the GSQL shell, run the `CREATE USER` command:

   ```
   GSQL > CREATE USER
   ```

2. Enter the user information in the prompts that follow:

   Example: Create user

   ```
   User Name : frank
   New Password : ************
   Re-enter Password : ************
   The user "frank" is created.
   ```

View roles of a user

Syntax

```
SHOW USER
```

Required privilege

`READ_USER` for displaying roles of other users

Procedure

1. From the GSQL shell, run the `SHOW USER` command:

```
GSQL > SHOW USER
```

If the user running the command has the `READ_USER` privilege, role information on all users will be displayed. Otherwise, only the current user's roles will be displayed.

View privileges of a user

Syntax

```
SHOW PRIVILEGE ON USER <username> (, <username>)*
```

Required privilege

`READ_USER`

Procedure

1. From the GSQL shell, run the `SHOW PRIVILEGE ON USER` command :

```
GSQL > SHOW PRIVILEGE ON USER tigergraph
```

The above command will show the privileges of user `tigergraph`:

```
User: "tigergraph"
  - Global Privileges:
    READ_SCHEMA
    WRITE_SCHEMA
    READ_LOADINGJOB
    EXECUTE_LOADINGJOB
    WRITE_LOADINGJOB
    READ_QUERY
    WRITE_QUERY
    READ_DATA
    WRITE_DATA
    WRITE_DATASOURCE
    READ_ROLE
    WRITE_ROLE
    READ_USER
    WRITE_USER
    READ_PROXYGROUP
    WRITE_PROXYGROUP
    READ_FILE
    WRITE_FILE
    DROP_GRAPH
    EXPORT_GRAPH
    CLEAR_GRAPHSTORE
    DROP_ALL
    ACCESS_TAG
```

Grant a role to a user

Syntax

```
GRANT ROLE <role_name1> (, role_name2)* [ON GRAPH <graph_name>]
        TO <username1> (, <username2>)*
```

Required privilege

`WRITE_ROLE`

Procedure

1. Start the GSQL shell and make sure you are using the correct graph

```
$ gsql
GSQL > USE GRAPH example_graph
```

2. From the GSQL shell, run the `GRANT ROLE` command. You can grant multiple roles to multiple users:

```
GSQL > GRANT ROLE role1 , role2 ON GRAPH example_graph
       TO user1, use2
```

The above command will grant roles `role1` and `role2` on graph `example_graph` to users `user1` and `user2`.

Revoke a role from a user

Syntax

```
REVOKE ROLE <roleName1> (, <roleName2)* [ON GRAPH <graphName>]
       FROM <userName1> (, <userName2>)*
```

Required privilege

`WRITE_ROLE`

Procedure

1. Start the GSQL shell and make sure you are using the correct graph

```
$ gsql
GSQL > USE GRAPH example_graph
```

2. From the GSQL shell, run the `REVOKE_ROLE` command. You can revoke multiple roles from multiple users at the same time:

```
GSQL > REVOKE ROLE role1, role2 ON GRAPH example_graph
       FROM user1, user2
```

The above command will revoke roles `role1` and `role2` on graph `example_graph` from users `user1` and `user2`.

Change a user's password

Syntax

```
ALTER PASSWORD <username>
```

Required privilege

`WRITE_USER` for changing the password of a user other than the current user

Procedure

1. From the GSQL shell, run the following command. Replace `username` with the user whose password you want to change

   ```
   GSQL > ALTER PASSWORD username
   ```

2. Enter the new password in the prompt that follows.

Drop a user

Syntax

```
DROP USER <user1> (,<user2>)*
```

Required privilege

`WRITE_USER`

Procedure

1. From the GSQL shell, run the `DROP USER` command. You can drop multiple users in the same command.

   ```
   GSQL > DROP USER user1, user2
   ```

2. GSQL will confirm that the users you entered have been dropped

Vertex-Level Access Control (Beta)

Vertex-level Access Control (VLAC) allows database administrators to control data access on the vertex level by attaching tags to individual vertices on a graph (the base graph) and creating *tag-based graphs*. Tag-based graphs share the underlying data with the base graph but have their own sets of roles and privileges, which allows administrators to exercise fine-grained data access control without the vertex type boundary.

Figure 1 below illustrates two tag-based graphs built upon a base graph. The base graph contains vertices of `person` type, and vertices of `post` type. Two tags (A and B) are used to tag them. For example, vertex 1 and vertex 9 both have tag A. Vertex 3 and vertex 11 both have tag A and B. A tag-based graph named `tagA` will only present to its users those base graph vertices that have tag A (the bottom-left graph). The other tag-based graph named `tagB` will only present to its users those base graph vertices that have tag B (the bottom-right graph).

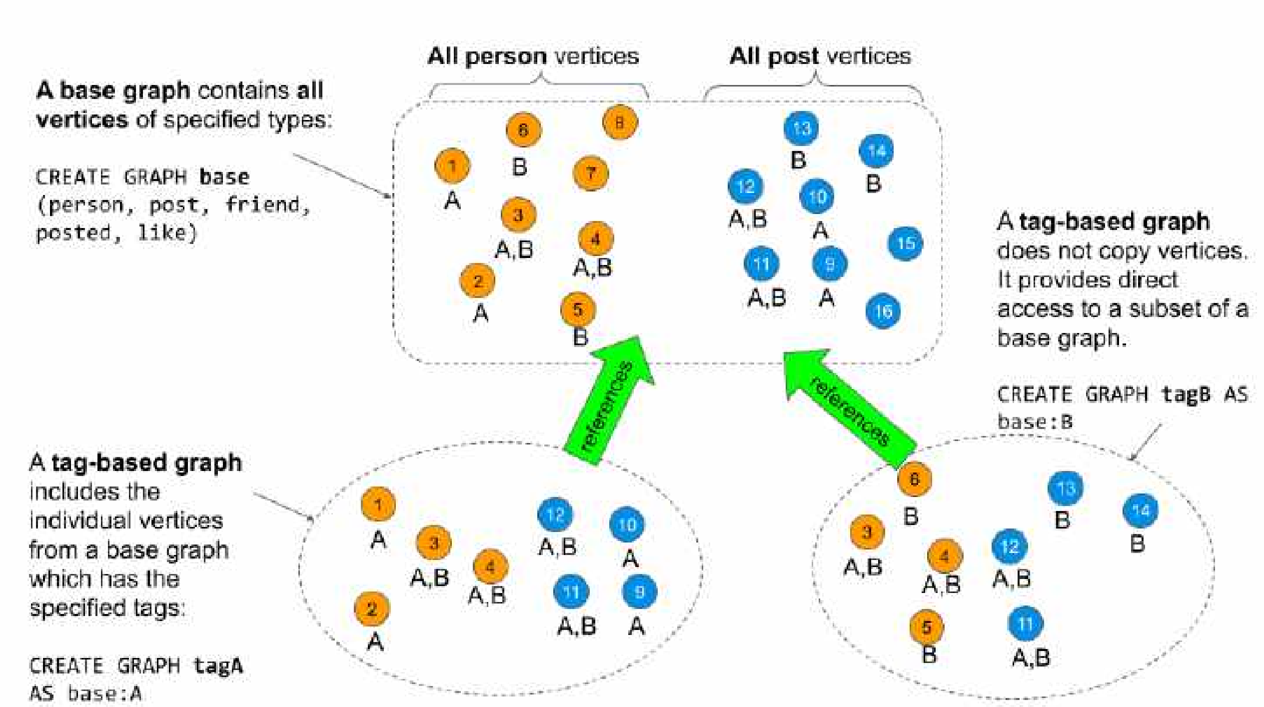


Figure 1. Tag-based graphs as subgraphs of a Base graph

For users operating on a tag-based graph, tags are an invisible aspect that silently filters how they load and query data. A tag-based graph defines its view, and all data outside its view is invisible to it.

Prerequisites

Sufficient privileges on the base graph or on the global scope are required to perform the steps described on this page.

- `ACCESS_TAG`
  - Any operation involving tags
- `WRITE_SCHEMA`
  - Create/run schema change jobs
  - Create tag-based graphs
- `READ_DATA, WRITE_DATA`
  - Run queries that modify the graph
- `WRITE_LOADINGJOB`, `EXECUTE_LOADINGJOB`
  - Create/run loading jobs

Workflow - Using Vertex Level Access Control

Below is the basic workflow on using the VLAC to control data access for a database administrator:

1. Create a tag-based graph

1. Define tags.
2. Mark vertex types as taggable.
3. Create a tag-based graph.

2. Manage users of the tag-based graph

- Grant built-in roles to users on a tag-based graph

- Define roles and grant them to users on a tag-based graph

3. Load and tag data

There are three main options for tagging vertices.

1. Explicitly add/modify tags on existing data: A user with the privilege `ACCESS_TAG` and `WRITE_DATA` privileges on the base graph can create and run a DML query that sets tags on selected individual vertices.
2. Explicitly set tags when loading data into to a base graph: A user with `EXECUTE_LOADINGJOB` , `WRITE_LOADINGJOB` and `ACCESS_TAG` privileges on the base graph can create and run a loading job that explicitly sets tags on the newly loaded base graph vertices.
3. Implicitly set tags when loading into a tag-based graph: A user with tag-based graph loading or insert privilege (e.g., a `designer` or `querywriter`) can create an ordinary loading or upsert job which inserts new vertices. The new vertices will be automatically tagged according to the tag-based graph's schema definition.

4. Query and update Data

Users with data read and write privileges (e.g., the `querywriter` and `queryreader` built-in roles) can query and update the tag-based graph as they would do any other graph. The data filtering for querying or data tagging for insertion is applied automatically.

The rest of this tutorial will first describe tag management: creating and dropping tags, making vertex type taggable, and using tags to define tag-based graphs. Next, the three ways to tag vertices are described and illustrated. We summarize the privilege scheme of tag-based graphs in terms of GSQL's predefined roles. Finally, we give some use cases that can be solved by VLAC.

> Features not yet supported:
>
> - DDL tag operations can only be done in GSQL. They are not yet supported in GraphStudio. This includes create/drop tags, create/alter vertices that are

taggable, define a tag-based graph.

- The privilege control for DDL operations (only admin and designer users should be able to explicitly manage tags) is not fully functional.

In summary, all necessary operations to set up VLAC graphs and users are supported in GSQL. Due to a known bug, standard users (with `querywriter` and `queryreader` roles) can run some DDL operations which they should not be able to.

We'll use the graph socialNet ↗ as an example in the following sections.

Tag Management

A tag is a special attribute of a vertex, which appears as a string for input and output purposes. If a vertex type is declared to be ***taggable***, then each vertex of that type can have one or more tags. The maximum number of different tags in a global graph is 64. All operations involving tags requires the user to have the `ACCESS_TAG` privilege.

Define a tag

A tag name has to be defined via `ADD TAG` before it can be used. Each base graph defines its own set of tags. However, there is a global maximum number of different tags, currently set at 64.

Syntax for ADD TAG

```
ADD TAG <tag_name> [DESCRIPTION <tag_description>]
```

`ADD TAG` can only be used inside a `SCHEMA_CHANGE JOB`. An example is below:

```
USE GRAPH socialNet

CREATE SCHEMA_CHANGE JOB add_tags {
  ADD TAG public DESCRIPTION "Open for public";
  ADD TAG tech DESCRIPTION "All about technology";
  ADD TAG vip DESCRIPTION "Very Important Person";
  ADD TAG dummy DESCRIPTION "Yeah, just a dummy";
}
RUN SCHEMA_CHANGE JOB add_tags
```

List tags

Run `ls` to see a list of defined tags:

```
...

Tags:
- TAG public DESCRIPTION "Open for public"
- TAG tech DESCRIPTION "All about technology"
- TAG vip DESCRIPTION "Very Important Person"
- TAG dummy DESCRIPTION "Yeah, just a dummy"
```

Drop a tag

The `DROP TAG` command not only removes the given tag(s) from the catalog of available tags, but also deletes them from each vertex to which it is attached. You can drop multiple tags in one statement.

Syntax for DROP TAG

```
DROP TAG <tag_name> ["," <tag_name>]*
```

Like `ADD TAG`, `DROP TAG` also needs to be inside a `SCHEMA_CHANGE JOB`:

```
USE GRAPH socialNet

CREATE SCHEMA_CHANGE JOB drop_dummy_tag {
  DROP TAG dummy;
}
RUN SCHEMA_CHANGE JOB drop_dummy_tag
```

1. You cannot drop a tag if it is used in the definition of a tag-based graph. You must drop the graph first.
2. When `DROP TAG` is executed, the specified tags will be made invalid, and then the foreground process will complete. A background process will continue to run to remove the tags from all data. In the meantime, each dropped tag still takes up one of the 64 slots for tags. The slot(s) will become available once the background process finishes.

Create a tag-based graph

A tag-based graph is a filtered view of a base graph, where a base graph is a simple collection of vertex types and edge types, without any tag specifiers. A tag-based graph must include at least one ***taggable vertex type*** from the base graph.

Mark vertex types as taggable

A vertex type has to be ***taggable*** to accept tags. `TAGGABLE` is a boolean property of a vertex type that can be set with `CREATE VERTEX` initially or with `ALTER VERTEX` in a schema change job:

```
USE GRAPH socialNet

# in general, this would be a local schema change job, but in socialNet,
# vertex types are global, so this needs to be a global schema change job
CREATE GLOBAL SCHEMA_CHANGE JOB make_taggable {
  ALTER VERTEX person WITH TAGGABLE="true";
  ALTER VERTEX post WITH TAGGABLE="true";
}

RUN GLOBAL SCHEMA_CHANGE JOB make_taggable
```

The property TAGGABLE is false by default. To change this default, use the `WITH` clause below when creating a vertex type:

```
CREATE VERTEX v2(PRIMARY_ID id UINT, name STRING) WITH TAGGABLE="true"
```

1. To change a vertex type from taggable to untaggable, use `WITH TAGGABLE="false".`
2. You cannot make a vertex type untaggable if it is used in the definition of a tag-based graph.
3. Edge types are never tagged. See the next section to see how we determine which edges to include in the tag-based graph.

Create a Tag-Based Graph

After a tag set and taggable vertex types have been created, we can use the tags to define a tag-based graph. For each vertex type we want to include, we may also specify a tag expression which must be satisfied for an individual vertex to be included.

Examples

Here is an example of creating a tag-based graph from the base graph `socialNet`.

```
USE GRAPH socialNet
CREATE GRAPH vipNet AS socialNet(person:vip, post, friend, posted, liked)
```

The interpretation is "Starting from the `socialNet` graph, create a tag-based graph called `vipNet` which includes `person` vertices which are tagged '`vip`'. Also include all `post` vertices and all `friend`, `posted` and `liked` edges."

Edges do not have tag expressions. An edge will be included when both of its vertex endpoints are included (and its edge type is included in the tag graph schema).

To describe a combination of tags, use the `&` operator to combine the tags:

```
USE GRAPH socialNet
CREATE GRAPH mixedNet AS socialNet(person:public&vip, post:public&tech&du
```

The graph `mixedNet` will only include the `person` vertices having both the `public` and `vip` tags, and posts having all three of the `public`, `tech` and `dummy` tags.

Same tag for all vertex types

If the desired tag-based graph is "anything in the base graph that has these tags", there is a convenient shortcut:

```
USE GRAPH socialNet
CREATE GRAPH publicNet2 AS socialNet:public
```

is the same as

```
USE GRAPH socialNet
CREATE GRAPH publicNet1 AS socialNet(person:public, post:public, friend,
```

General Syntax

The formal syntax for both the general form and the simplified form of creating a tag-based graph is shown below:

Syntax for CREATE GRAPH for a tag-based graph

```
<create_tag_graph> :=
    CREATE GRAPH <tag_graph_name> AS <base_graph_name>
    ( "(" <tagged_element_name> ("," <tagged_element_name>)* ")" | ":" <t

<tagged_element_name> := <tagged_vertex_name> | <edge_name>

<tagged_vertex_name> := <vertex_name> [":" <tag_expr>]

<tag_expr> := <tag> ("&" <tag_expr>)*
```

How To Tag Vertices

There are three main options for tagging vertices in the base graph.

- **Add tags on existing data with DML queries.** For existing data, a user with base graph tagging privilege (e.g., an `admin` or `designer`) can create and run a DML query that sets tags on selected individual vertices.
- **Explicitly set tags when loading/inserting to a base graph**. For new data, a user with base graph loading and tagging privilege (e.g., an `admin` or `designer`) can create and run a loading job that explicitly sets tags on the newly loaded vertices.
- **Implicitly set tags when loading/inserting into a tag-based graph**. For new data, a user with tag-based graph loading or insert privilege (e.g., a `designer` or `querywriter`) can create an ordinary Loading or Upsert Job which inserts new vertices. The new vertices will be automatically tagged according to the tag-based graph's schema definition.

Add tags on existing data

In GSQL, special vertex methods are provided to access and modify the tags of a vertex in a DML query (full list available on page Vertex Functions). These functions are only available for vertex aliases (defined in the `FROM` clause of a `SELECT` statement); they cannot be applied to vertex variables in other contexts.

There are 8 DML-level tag-access functions in the vertex-query block or edge-query block. Use the `v.addTags()` ↗ function to tag a vertex.

Required privilege

`READ_DATA`, `WRITE_DATA`, `WRITE_QUERY` , `ACCESS_TAG`

> To add or modify tags, you should work at the base graph level.

Examples

`addTags()` is shown below. This query will add tags to person vertices to achieve the same effect as a base graph loading job example in the previous section.

```
CREATE QUERY addTagsToPerson() {
  Seed = { any };
  # person1 ~ person5 will be tagged as public.
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id IN ("person1","person2","person3","person4","person5"
         ACCUM s.addTags("public");

  # person6 and person7 will be tagged as public and vip.
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id IN ("person6","person7")
         ACCUM s.addTags("vip", "public");

  # person8 will be tagged as vip
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id == "person8"
         ACCUM s.addTags("vip");
}
```

Use `removeTags()` and `removeAllTags()` to remove tags from vertices:

```
// remove tag “vip” and “public” from all person vertices.
CREATE QUERY removetagsFromPerson() {
  vSet = { person.* };
  # remove tag vip and public from all person vertices
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.removeTags("vip", "public");
}

// remove all tags from all person vertices.
CREATE QUERY removealltagsFromPerson() {
  vSet = { person.* };
  # remove all tags from all person vertices
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.removeAllTags();
}
```

Set tags explicitly with `TAGS` clause

Tags can be added to vertices at their loading time using a base graph loading job.

The `LOAD` statement has an optional clause for explicit tagging of loaded data. The tagging clause has two keywords, `TAGS` and `BY:`

- `TAGS(<tag_list>)` specifies the tags to be set.
- `BY` specifies how to merge tags if the targeted vertex exists in the graph
  - `BY OR:` Add the given tags to the existing set of tags.
  - `BY OVERWRITE:` Replace the existing tags with the given ones.

Required privilege

`WRITE_LOADINGJOB` , `EXECUTE_LOADINGJOB`, `ACCESS_TAG`

Example 1

Suppose we want to put the tags `vip` and `public` on the `person` vertex data coming from a certain file. We have three files: `persons1`, `persons2`, `persons3`.

```
$ cat persons1
person1,Male
person2,Female
person3,Male
person4,Female
person5,Female

$ cat persons2
person6,Male
person7,Male

$ cat persons3
id,gender,label
person8,Male,vip
```

Create and run three loading jobs:

```
USE GRAPH socialNet

# person1 - person5 will be tagged as public.
CREATE LOADING JOB loadPersonPublic {
  DEFINE filename f;
  LOAD f TO VERTEX person VALUES($0, $0, $1) TAGS("public") BY OR;
}
RUN LOADING JOB loadPersonPublic USING f="./persons1"

# person6 and person7 will be tagged as public and vip.
CREATE LOADING JOB loadPersonPublicVip {
  DEFINE filename f;
  LOAD f TO VERTEX person VALUES($0, $0, $1) TAGS("public", "vip") BY OR;
}
RUN LOADING JOB loadPersonPublicVip USING f="./persons2"

# person8 will be tagged as vip which is derived from the file.
CREATE LOADING JOB loadPerson {
  DEFINE filename f;
  LOAD f TO VERTEX person VALUES($0, $0, $1) TAGS($2) BY OR USING HEADER=
}
RUN LOADING JOB loadPerson USING f="./persons3"
```

Note that the `TAGS` clause can specify a tag with a string literal (`"vip"`) so every vertex gets the same tag, or with a token reference by position (`$2`) or by name (`$"label"`) from the source file, so each vertex gets a data-dependent tag. If the tag clause refers to a non-existent tag, the loading job will still run, but the data will not be loaded at runtime. The loading job log will report these non-loaded vertices.

Example 2

We have three post files: `posts1`, `posts2`, and `posts3`.

```
$ cat posts1
3,cats,2011-02-05 01:02:44
8,cats,2011-02-03 17:05:52
9,cats,2011-02-05 23:12:42
10,cats,2011-02-04 03:02:31
11,cats,2011-02-03 01:02:21

$ cat posts2
4,coffee,2011-02-07 05:02:51

$ cat posts3
0,Graphs,2010-01-12 11:22:05
1,tigergraph,2011-03-03 23:02:00
2,query languages,2011-02-03 01:02:42
5,tigergraph,2011-02-06 01:02:02
6,tigergraph,2011-02-05 02:02:05
7,Graphs,2011-02-04 17:02:41
```

We create and run the following loading jobs:

```
USE GRAPH socialNet

# posts 3, 8, 9, 10, and 11 will be tagged as public.
CREATE LOADING JOB loadPostPublic {
  DEFINE filename f;
  LOAD f TO VERTEX post VALUES($0, $1, $2) TAGS("public") BY OR ;
}
RUN LOADING JOB loadPostPublic USING f="./posts1"

# posts 0, 1, 2, 5, 6, and 7 will be tagged as both public and tech.
CREATE LOADING JOB loadPostPublicTech {
  DEFINE filename f;
  LOAD f TO VERTEX post VALUES($0, $1, $2) TAGS("public", "tech") BY OR;
}
RUN LOADING JOB loadPostPublicTech USING f="./posts3"

# post 4 will remain untagged.
```

Set tags implicitly by inserting into a tag-based graph

Loading data to a tag-based graph automatically tags each vertex with the tags specified in the graph's definition. For example, when loading to `vipNet`, the `person` vertices will automatically be tagged with `vip`.

If you load data into a tag-based graph, these vertices are actually being added to the parent base graph. If two tag-based graphs have overlapping views (e.g. if the graph `vipNet2` also includes `person:vip`), then when one adds a vertex via the tag-based graph, the other tag-based graph may also see it.

```
USE GRAPH vipNet

CREATE LOADING JOB loadMember {
  DEFINE filename f;
  // TAGS("vip") BY OR will be applied implicitly
  // since vipNet is defined based on person:vip
  LOAD f TO VERTEX person VALUES($0, $0, $1);
}
RUN LOADING JOB loadMember USING f="./persons3"
```

- **Portability and Reusability**: The same loading job works for `socialNet` or any graph derived from `socialNet` which contains `person`. The difference is in the effect: running it with `vipNet` will apply the `vip` tag. Running it with a different tag-based graph would apply different tags. *Users of a given tag-based graph automatically insert and query data for that tag-based graph.*
- **Tagging Shared Data**: The default behavior of GSQL loading is upsert: if you attempt to **insert** a vertex or edge which already exists (e.g., uses an existing ID), you will instead **update** the existing element with the new attribute values. If the attribute is a list or set, the new values will be added to the existing list/set. This applies to tags. If you attempt to load an existing vertex, the new tag(s) will be added to any existing tags. *Loading a vertex that already exists extends the tag set with the guidance of the tag-graph schema.*

Query a Tag-based Graph

The graph `vipNet` only includes vertices with the tag `vip`. We can verify this by running a simple query to return all person vertices in `vipNet`:

```
USE GRAPH vipNet

CREATE QUERY findAll() {
  seed = {person.*};
  result =
    SELECT v
    FROM seed:v
    ORDER BY v.id;

  PRINT result;
}
INSTALL QUERY findAll
RUN QUERY findAll()
```

The output of the query would be:

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"res": [
    {
      "v_id": "person6",
      "attributes": {
        "gender": "Male",
        "id": "person6"
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "id": "person7"
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {
        "gender": "Male",
        "id": "person8"
      },
      "v_type": "person"
    }
  ]}]
}
```

Access Control

Users with global `WRITE_SCHEMA` and `ACCESS_TAG` privileges can create, modify and drop tags, as well as create tag-based graphs for all graphs.

On the base graph

Users with roles on the base graph that have the `ACCESS_TAG` privilege (e.g. `admin` and `designer` roles) can create/drop tags, and tag vertices. Users that have both the `ACCESS_TAG` privilege and `WRITE_SCHEMA` privilege (e.g. `admin` and `designer` roles) can create/drop tag-based graphs of the base graph.

Users with roles that don't have the `ACCESS_TAG` privilege on the base graph are able to access the base graph as their roles allow, but they do not have access to the tags on the base graph. They cannot see whether any vertex type on the graph is taggable or if there are tag-based graphs of the base graph.

Users with roles on the tag-based graphs of the base graph cannot access the base graph if they don't have a role with privileges for the base graph.

On tag-based graphs

When a new tag-based graph is created, users with `admin` or `designer` roles will inherit their base graph role on the tag-based graph. Additionally, the creator of the tag-based graph becomes an admin of the tag-based graph.

Users who are given roles on a tag-based graph have the privileges on the tag-based graph that correspond to their roles, except they are not allowed to edit the tag-based graph's graph schema.

Sample Use Cases

Scenario I

Problem

A user with `admin` role on a graph wants to grant a group of users access to a selective set of vertices.

Solution

The base graph admin can do the following security setup.

1. **Define a tag**. In a schema change job, declare a tag ↗ `T` for this application.
2. **Mark vertex types as taggable**. Identify the vertex types you want to give selective access for, and mark those vertex types as taggable ↗ in a schema change job.
3. **Define a tag-based graph**. Define a tag-based graph ↗ `B` with the taggable vertex types, with `T` as their tag expression.
4. **Tag vertices**. Write a DML query on the base graph and use the tag functions in the query to tag the vertices ↗ you want to include in the tag-based graph, and run the query.
5. **Grant users permission to the tag-based graph**. On the tag-based graph B, grant roles that have the appropriate privileges for graph `B` to the target users.

Scenario II

Problem

You have a source file containing class annotations (tags) on vertex data. You want to grant users access to the vertices that have the annotation `T1`. In the future, you also want the ability to give other users access to vertices based on the vertex class.

Solution

The base graph `admin` user can do the following setup.

1. **Define tags**. Declare tags ↗ `T1, T2, … Tn` for all the classes in your source file in a schema change job.
2. **Mark vertex types as taggable**. Identify the vertex types of the vertices in your source file that have class annotations, and mark those vertex types as taggable ↗ in a schema change job.
3. **Define a tag-based graph**. Define a tag-based graph ↗ `B` with `T1` as the tag expression.
4. **Explicitly tag vertices during data loading**. Write a base graph loading job, and in the loading job, use a `TAGS() BY` clause to explicitly add tags ↗ to the ingested vertices.

5. **Grant roles on the tag-based graph**. On the tag-based graph `B`, grant roles that have the appropriate privileges for the graph `B` to target users.

Scenario III

Problem

An `admin` user on a graph wants to give a group of users read/write access for a specific class of vertices. The users would be able to insert new vertices into the graph and query the data, and all the data they insert into the graph are tagged as the same class.

Solution

The base graph admin can do the following setup.

- **Define a tag**. Declare a tag ↗ `T` for this application in a schema change job.
- **Mark vertex types as taggable**. Identify the vertex types to give selective access to, and mark the relevant vertex types as taggable ↗ in a schema change job.
- **Define a tag-based graph**. Define a tag-based graph ↗ `B` with `T` as the tag expression.
- **Grant roles on the tag-based graph**. On the tag-based graph, grant roles with the appropriate privileges to target users.

These group users operate (including delete/update/insert) on graph `B` as if it is a normal graph. They can ingest new data, as well as operate on those vertices from the base graph that have the tag `T`.

Lightweight Directory Access Protocol (LDAP)

The Lightweight Directory Access Protocol (LDAP) is an industry-standard protocol for accessing and maintaining directory information services across a network. Typically, LDAP servers provide centralized user authentication services. The Tigergraph system supports LDAP authentication by allowing a TigerGraph user to log in using an LDAP username and credentials. During the authentication process, the GSQL server connects to the LDAP server and requests the LDAP server to authenticate the user.

Supported Features

GSQL LDAP authentication supports any LDAP server that follows the LDAPv3 protocol. StartTLS/SSL connection is also supported.

SASL authentication is not yet supported. Some LDAP servers are configured to require a client certificate upon connection. Client certificates are not yet supported in GSQL LDAP authentication.

Mapping Users From LDAP to GSQL

In order to manage the user roles and privileges, the TigerGraph GSQL server employs two concepts—proxy user and proxy group.

Proxy User

A proxy user is a GSQL user created to correspond to an external LDAP user. When operating within GSQL, the external LDAP user's roles and privileges are determined by the proxy user.

Proxy Group

A proxy group is a GSQL user group that is used to manage a group of proxy users who share similar properties/attributes in LDAP.

An existing LDAP user can log in to GSQL only when the user matches at least one of the existing proxy groups' criteria. Once the criteria are satisfied, a proxy user will be created for the LDAP user. The roles and privileges of the proxy user are at least as permissive as the proxy group(s) he belongs to. It is also possible to change the roles of a specific proxy user independently. When the roles and privileges of a proxy group changes, the roles and privileges of all the proxy users belonging to this proxy group change accordingly.

Configure GSQL LDAP Authentication

To configure a TigerGraph system to use LDAP, there are two main configuration steps:

1. Configure the LDAP Connection.
2. Configure GSQL Proxy Groups and Users.

In order to choose and specify your LDAP configuration settings, you must understand some basic LDAP concepts. One reference for LDAP concepts is https://www.ldap.com/basic-ldap-concepts ↗.

Prerequisites

Sufficient privileges are required to configure LDAP Authentication:

- `WRITE_ROLE`
- `WRITE_PROXYGROUP`

Step 1 - Configure the LDAP Connection

To enable and configure LDAP, run three commands.

1. Configure LDAP:

```
gadmin config entry ldap
```

The gadmin program will then prompt the user for the settings for several LDAP configuration parameters.

2. Apply the configuration:

```
gadmin config apply -y
```

3. Restart the gsql server:

```
gadmin restart gsql -y
```

An example configuration is shown below.

```
Example of gadmin config entry ldap

$ gadmin config entry ldap

Security.LDAP.Enable [ false ]: Enable LDAP authentication: default false
New: true

Security.LDAP.Hostname [ localhost ]: Configure LDAP server hostname: def
New: ldap.tigergraph.com

Security.LDAP.Port [ 389 ]: Configure LDAP server port: default 389
New: 389

Security.LDAP.BaseDN [  ]: Configure LDAP search base DN, the root node t
New: dc=tigergraph,dc=com

Security.LDAP.SearchFilter [ (objectClass=*) ]: Configure LDAP search bas
New: (objectClass=*)

Security.LDAP.UsernameAttribute [ uid ]: Configure the username attribute
New: uid

Security.LDAP.AdminDN [  ]: Configure the DN of LDAP user who has read ac
New: cn=Manager,dc=tigergraph,dc=com

Security.LDAP.AdminPassword [ secret ]: Configure the password of the adm
New: secret

Security.LDAP.Secure.Protocol [ none ]: Enable SSL/StartTLS for LDAP conn
New: none

Security.LDAP.Secure.TruststorePath [  ]: Configure the truststore absolu
New: /tmp/ca_server.pkcs12

Security.LDAP.Secure.TruststoreFormat [ JKS ]: Configure the truststore f
New: PKCS12

Security.LDAP.Secure.TruststorePassword [ changeit ]: Configure the trust
New: test

Security.LDAP.Secure.TrustAll [ false ]: Configure to trust all LDAP serv
New: false
[   Info] Configuration has been changed. Please use 'gadmin config apply
```

Below is an explanation of each configuration parameter.

Name	Description	Example
Security.LDAP.AdminDN	Configure the DN of LDAP user who has read access to the base DN specified above. Empty if everyone has read access to LDAP data: default empty	`nan`
Security.LDAP.AdminPassword	Configure the password of the admin DN specified above. Needed only when admin_dn is specified: default empty. If the value provided is a path to a script, the parameter will be set to the output of the script.	`secret`
Security.LDAP.BaseDN	Configure LDAP search base DN, the root node to start the LDAP search for user authentication: must specify	`nan`
Security.LDAP.Enable	Enable LDAP authentication: default false	`false`
Security.LDAP.Hostname	Configure LDAP server hostname: default localhost	`localhost`
Security.LDAP.Port	Configure LDAP server port: default 389	`389`
Security.LDAP.SearchFilter	Configure LDAP search base DN, the root node to start the LDAP search for user authentication.	`(objectClass=*)`
Security.LDAP.Secure.Protocol	Enable SSL/StartTLS for LDAP connection [none/ssl/starttls]: default none	`none`

Security.LDAP.Secure.TrustAll	Configure to trust all LDAP servers (unsafe): default false	`false`
Security.LDAP.Secure.TruststoreFormat	Configure the truststore format [JKS/PKCS12]: default JKS	`JKS`
Security.LDAP.Secure.TruststorePassword	Configure the truststore password: default changeit	`changeit`
Security.LDAP.Secure.TruststorePath	Configure the truststore absolute path for the certificates used in SSL: default empty. If the value provided is a path to a script, the parameter will be set to the output of the script.	`nan`

Step 2 - Configure GSQL Proxy Groups and Users

This section explains how to configure a GSQL proxy group in order to allow LDAP user authentication.

Configure Proxy Group

A GSQL proxy group is created by the CREATE GROUP command with a given proxy rule. For example, assume there is an attribute called "role" in the LDAP directory, and "engineering" is one of the "role" attribute values. We can create a proxy group with the proxy rule "role=engineering". Different roles can then be assigned to the proxy group. An example is shown below. When a user logins, the GSQL server searches for the user's entry in the LDAP directory. If the user's LDAP entry matches the proxy rule of an existing proxy group, a proxy user is created to which the user will log in.

CREATE GROUP command

```
# create a proxy group
CREATE GROUP developers PROXY "role=engineering" // Any user in LDAP with

# grant role to proxy group
GRANT ROLE querywriter ON GRAPH computerNet TO developers
```

The SHOW GROUP command will display information about a group. The DROP GROUP command deletes the definition of a group.

SHOW GROUP and DROP GROUP commands

```
# show the current groups
SHOW GROUP

# delete a proxy group
DROP GROUP developers
```

Proxy User

Nothing needs to be configured for a proxy user. As long as the proxy rule matches, the proxy user will be automatically created upon login. A proxy user is very similar to a normal user. The minor differences are that a proxy user cannot change their password in GSQL and that a proxy user comes with default roles inherited from the proxy group that they belong to.

Frequently Asked Questions

What is security.ldap.admin_dn?

Admin_dn is the "distinguished name" of an LDAP entry. In LDAP, "distinguished name" is often abbreviated as dn. When configuring this field, a dn entry with read permission on the ldap directory is expected. Configuring a dn with no read permission will result in an error. Not configuring this field will likely result in an error since the LDAP server is typically not publicly readable. Please note that only the dn field will be accepted for this entry. All other entries will result in an authentication error. The corresponding password for the configured dn should also be set correctly in the configured entry "security.ldap.admin_password ".

What protocol should I use for security.ldap.secure.protocol?

It depends on what type of protocol your LDAP server uses. SSL/TLS is very common in enterprise use today. When SSL is used, the port is typically 636 instead of default port 389.

Should I configure the truststore and how?

You need to configure the truststore when SSL/TLS is used in the LDAP server. The truststore's path, password, and format need to be configured accordingly. We support two formats—JKS and PKCS12. The JKS is Java KeyStore. The corresponding certificates for the LDAP server need to be imported to the JKS for successful authentication. Different truststore formats are typically interchangeable.

What if I just want to test the LDAP login without any certificate?

This might be the case if SSL/TLS is enabled from the LDAP server side but you don't have a certificate. You can set "security.ldap.secure.trust_all" to true to bypass the SSL/TLS certificate checking.

What does it mean when I try to login but got "parameter error"? Can I see a more detailed error message?

"Parameter error" means some of the LDAP configurations are not set properly. Most often it is because admin_dn, admin_password, or the login username and password are not set correctly. Unfortunately, we cannot know exactly what field is wrong because the LDAP server side does not respond back with such detail.

What does it mean when I see error "User does not match any proxy rule"?

Congratulations! This means the LDAP is working. However, TigerGraph cannot find a matching rule for the login user. Please create a proxy group for the user. See documents for creating a proxy group here.

Single Sign-On

The Single Sign-On (SSO) feature in TigerGraph enables you to use your organization's identity provider (IDP) to authenticate users to access TigerGraph GraphStudio and Admin Portal UI.

Currently we have verified following the identity providers which support SAML 2.0 protocol:

- Okta ↗
- Auth0 ↗

For supporting additional IDPs, please inquire sales@tigergraph.com and submit a feature request.

In order to use Single Sign-On , you need to perform four steps :

1. Configure your identity provider to create a TigerGraph application.
2. Provide information from your identity provider to enable TigerGraph Single Sign-On .
3. Create user groups with proxy rules to authorize Single Sign-On users.
4. Change the password of the tigergraph user to be other than the default, if you haven't done so already.

We assume you already have TigerGraph up and running , and you can access GraphStudio UI through a web browser using the URL: http://tigergraph-machine-honestname:14240

If you enabled SSL connection, change `http` to `https`. If you changed the Nginx port of the TigerGraph system, replace 14240 with the port you have set.

Configure Identity Provider

Here we provide detailed instructions for identity providers that we have verified. Please consult your IT or security department for how to configure the identity provider for your organization if it is not listed here.

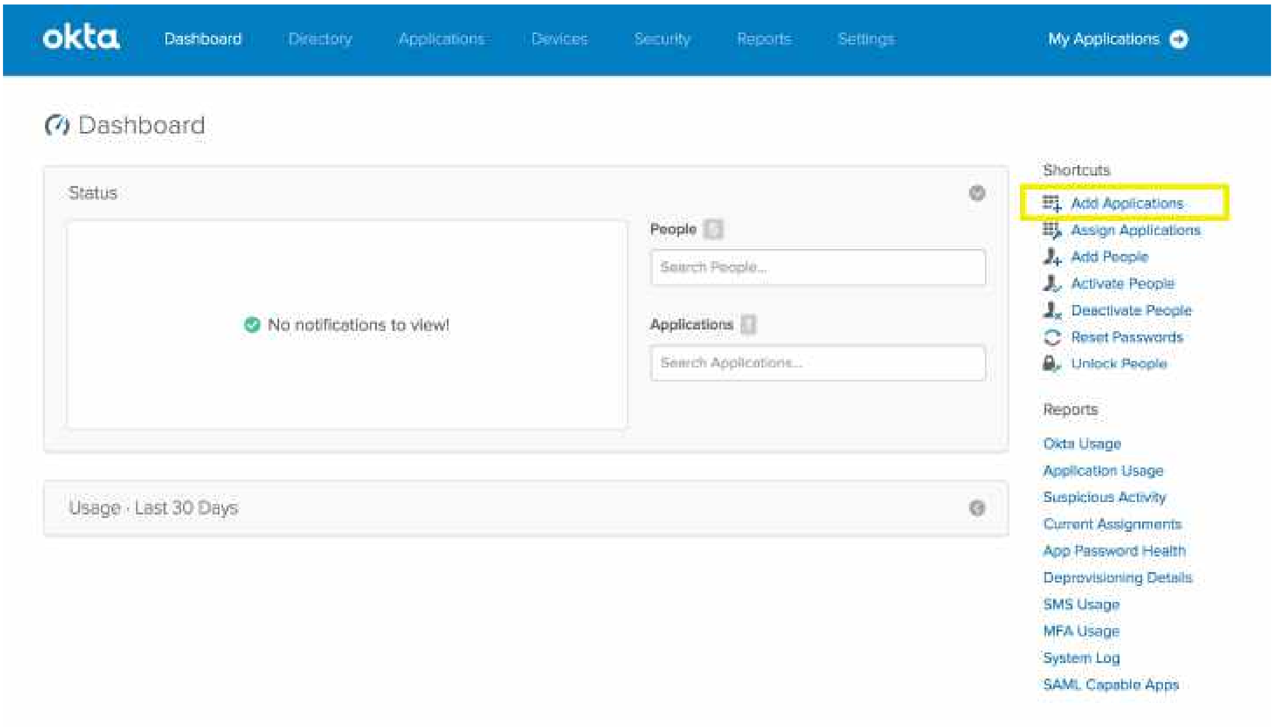
After you finish configuring your identity provider, you will get the following:

- An Identity Provider Single Sign-On URL
- An Identity Provider Entity ID
- an X.509 certificate file `idp.cert`. You need these 3 things to configure TigerGraph next.

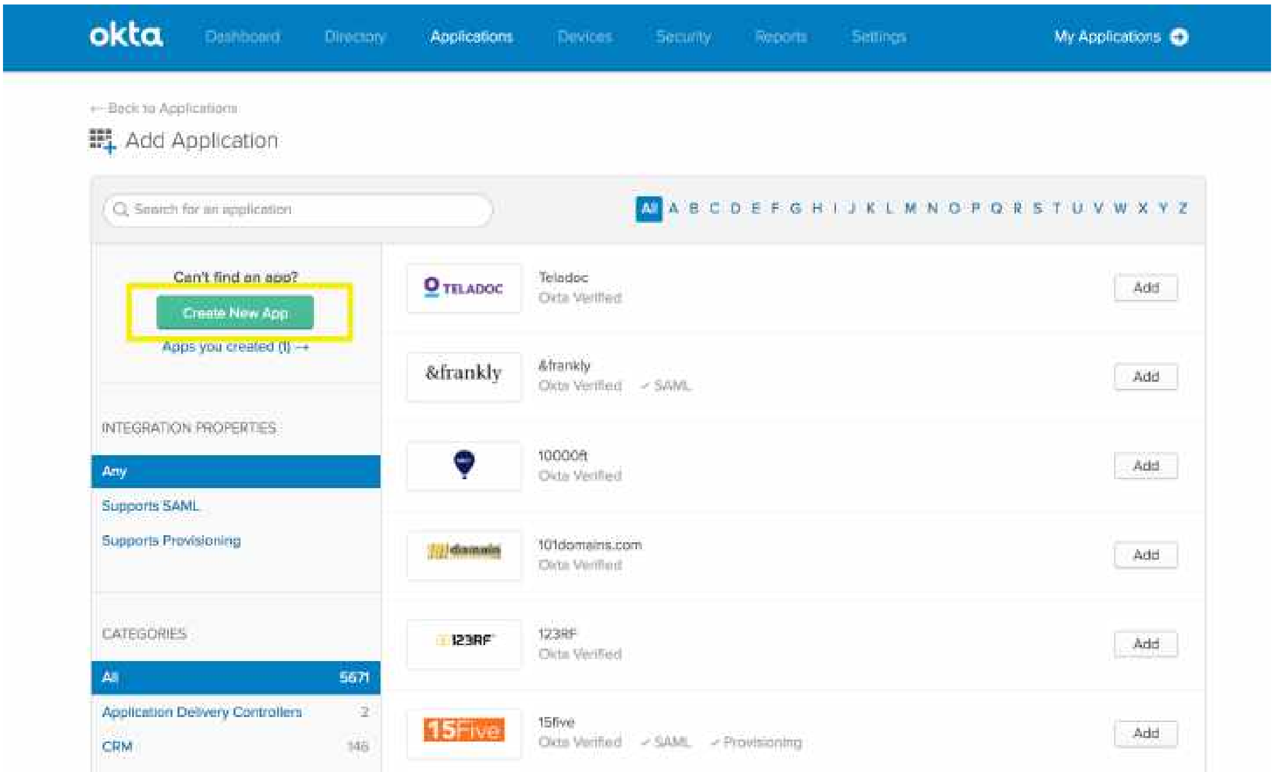
Okta

After logging into Okta as the admin user, click Admin button at the top-right corner.

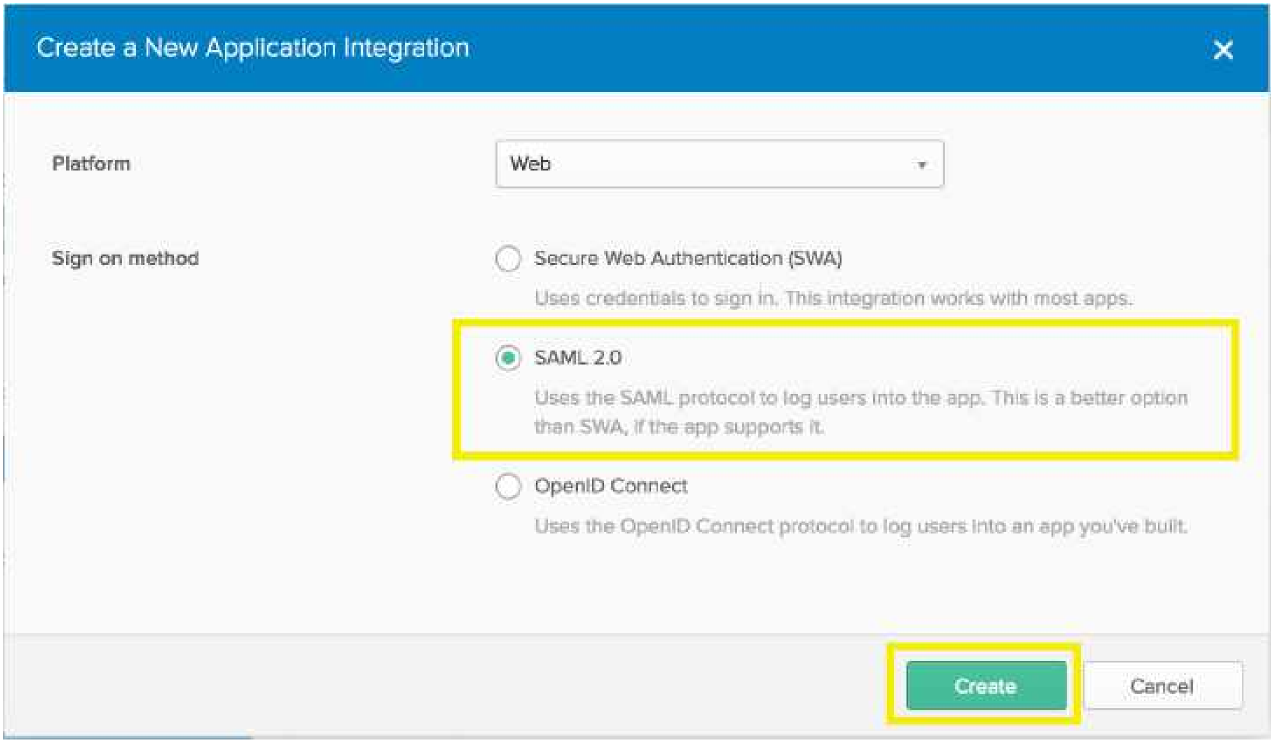
Click Add Applications in the right menu.



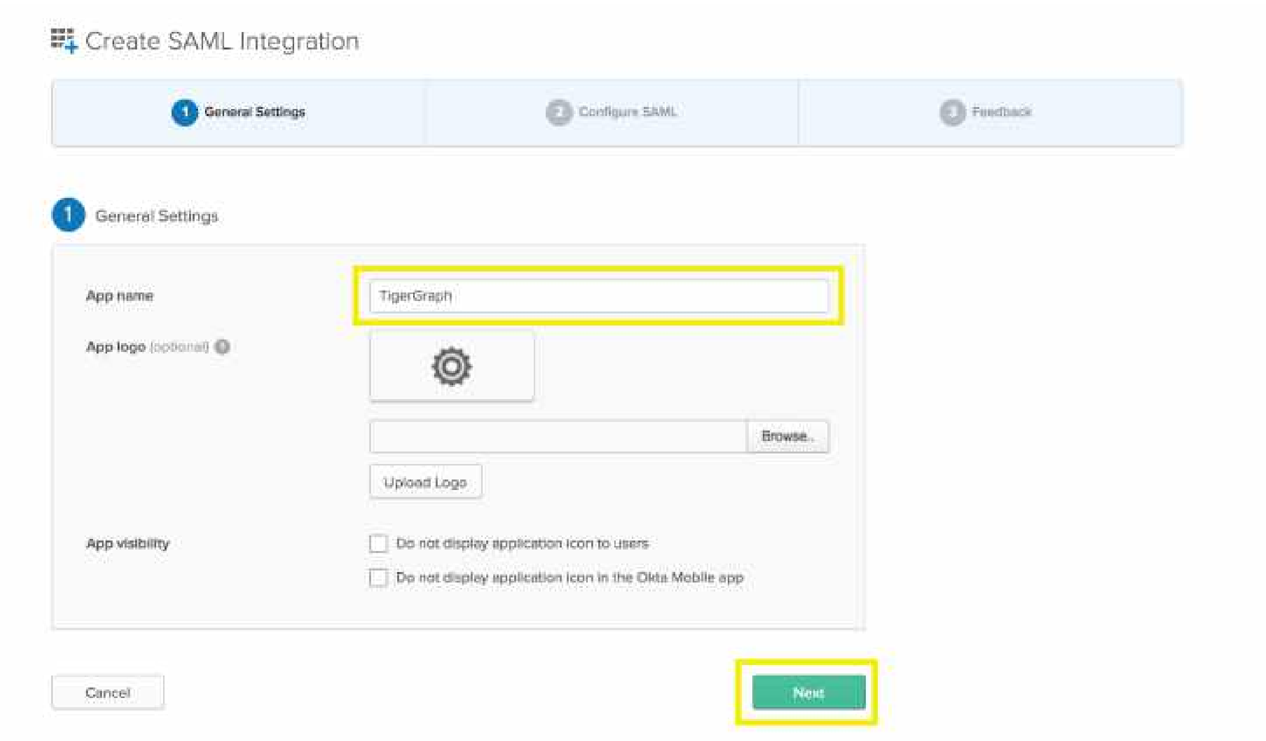
Click Create New App button in the left toolbar.



In the pop up window, choose SAML 2.0 and click Create .



Input TigerGraph (or whatever application name you want to use) in App Name , and click Next . Upload a logo if you like.



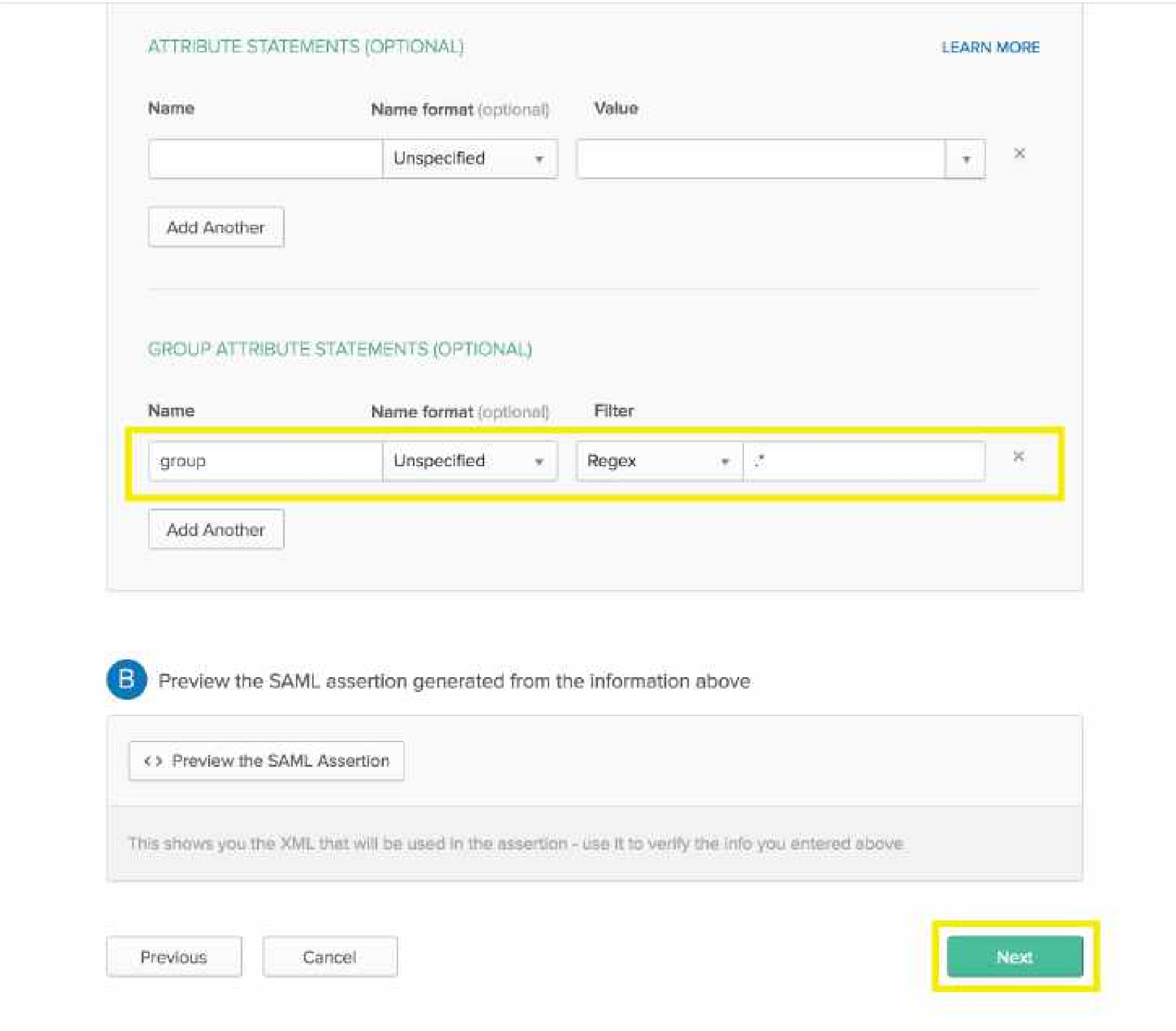
Enter the Assertion Consumer Service URL / Single sign on URL , and SP Entity ID .

Both are URLs in our case. You need to know the hostname of the TigerGraph machine. If you can visit GraphStudio UI through a browser, the URL contains the hostname. It can be either an IP or a domain name.

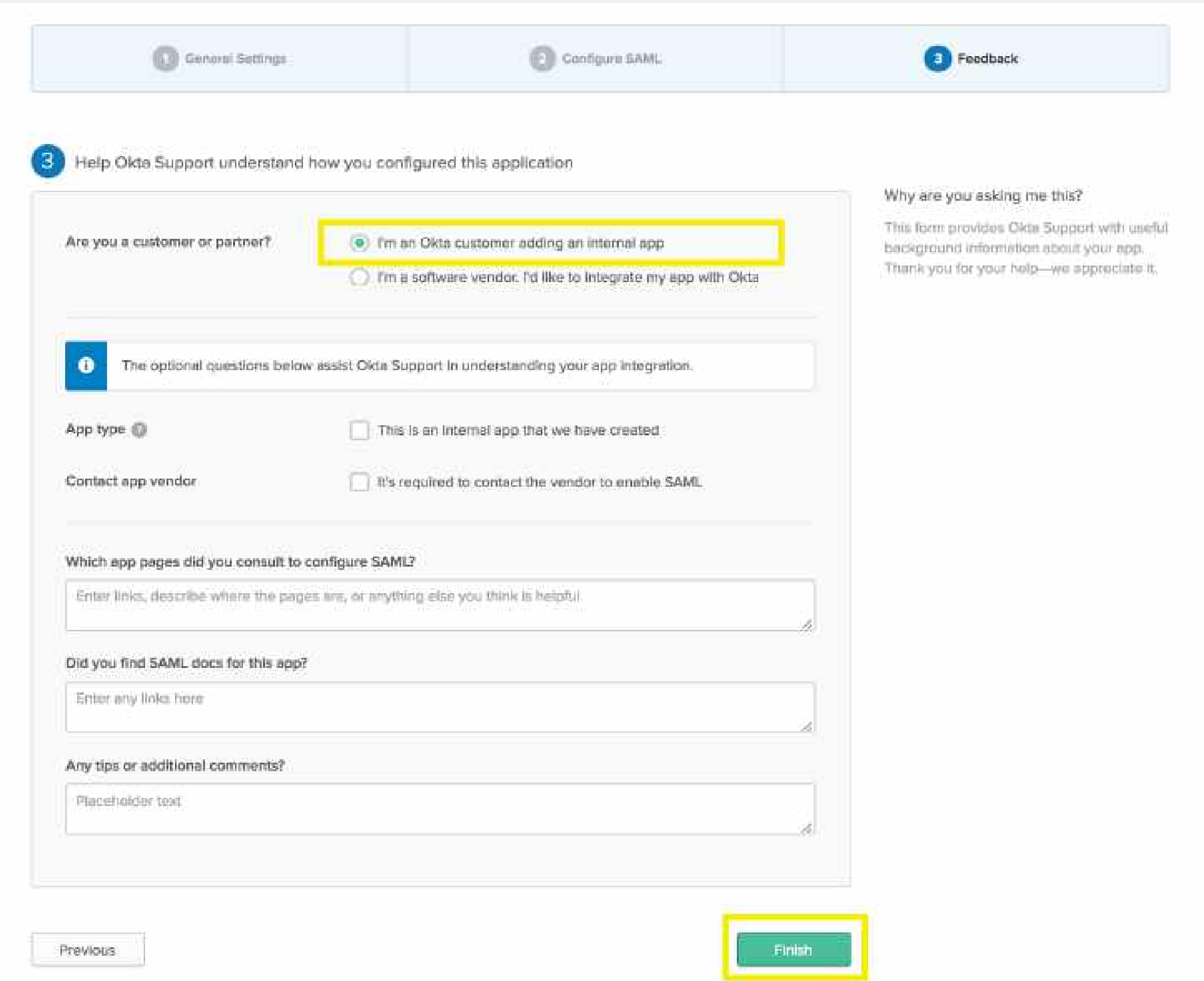
The Assertion Consumer Service URL, or Single sign on URL, is

- Assertion Consumer Service URL, or Single sign on URL: http://tigergraph-machine-hostname:14240/api/auth/saml/acs
- The SP entity id URL: http://tigergraph-machine-hostname:14240/gsqlserver/gsql/saml/meta

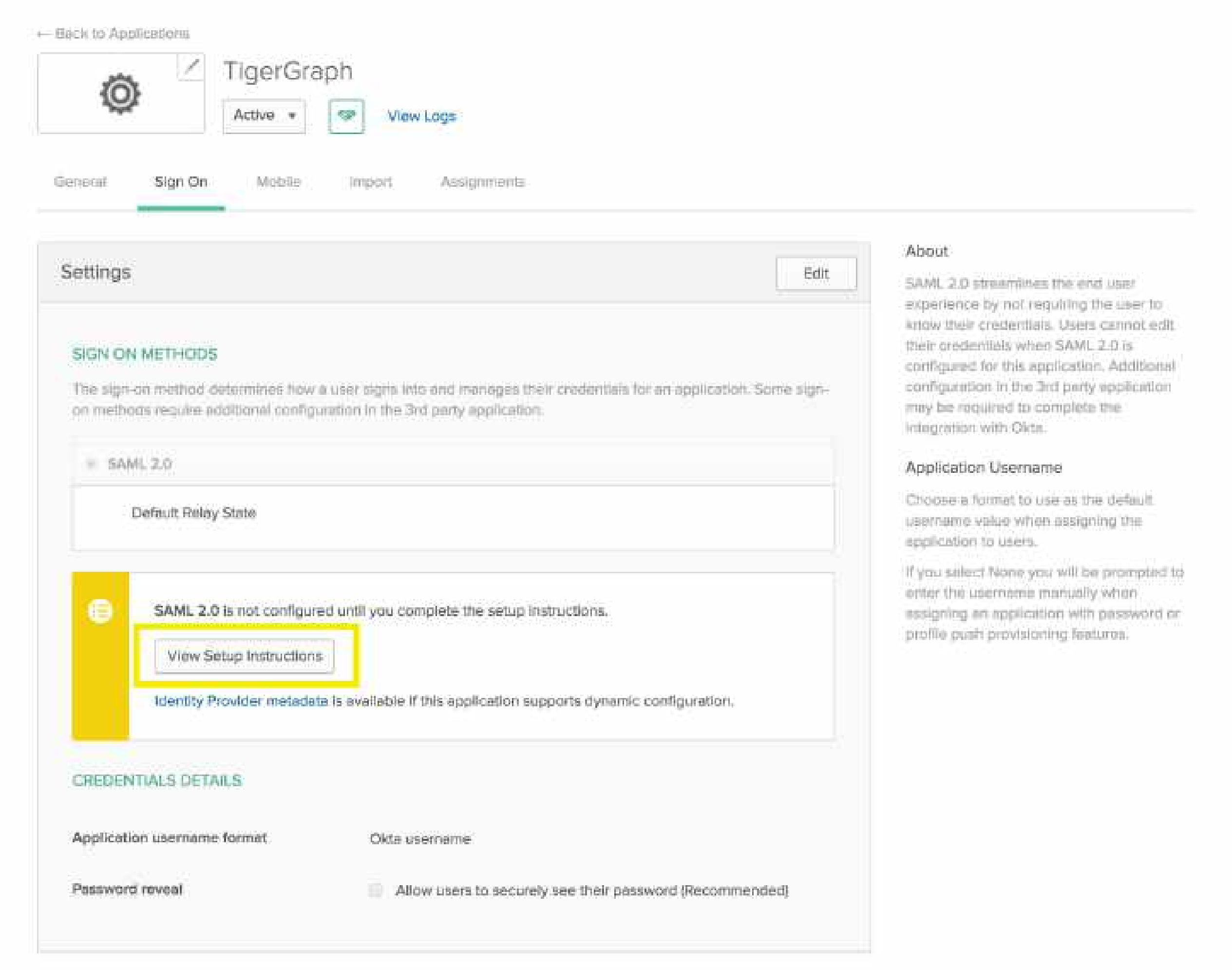
Scroll to the bottom for Group Attribute Statements. Usually you want to grant roles to users based on their user group. You can give a name to your attribute statement; here we use group . For filter, we want to return all group attribute values of all users, so we use Regex .* as the filter. Click Next after you set up everything.



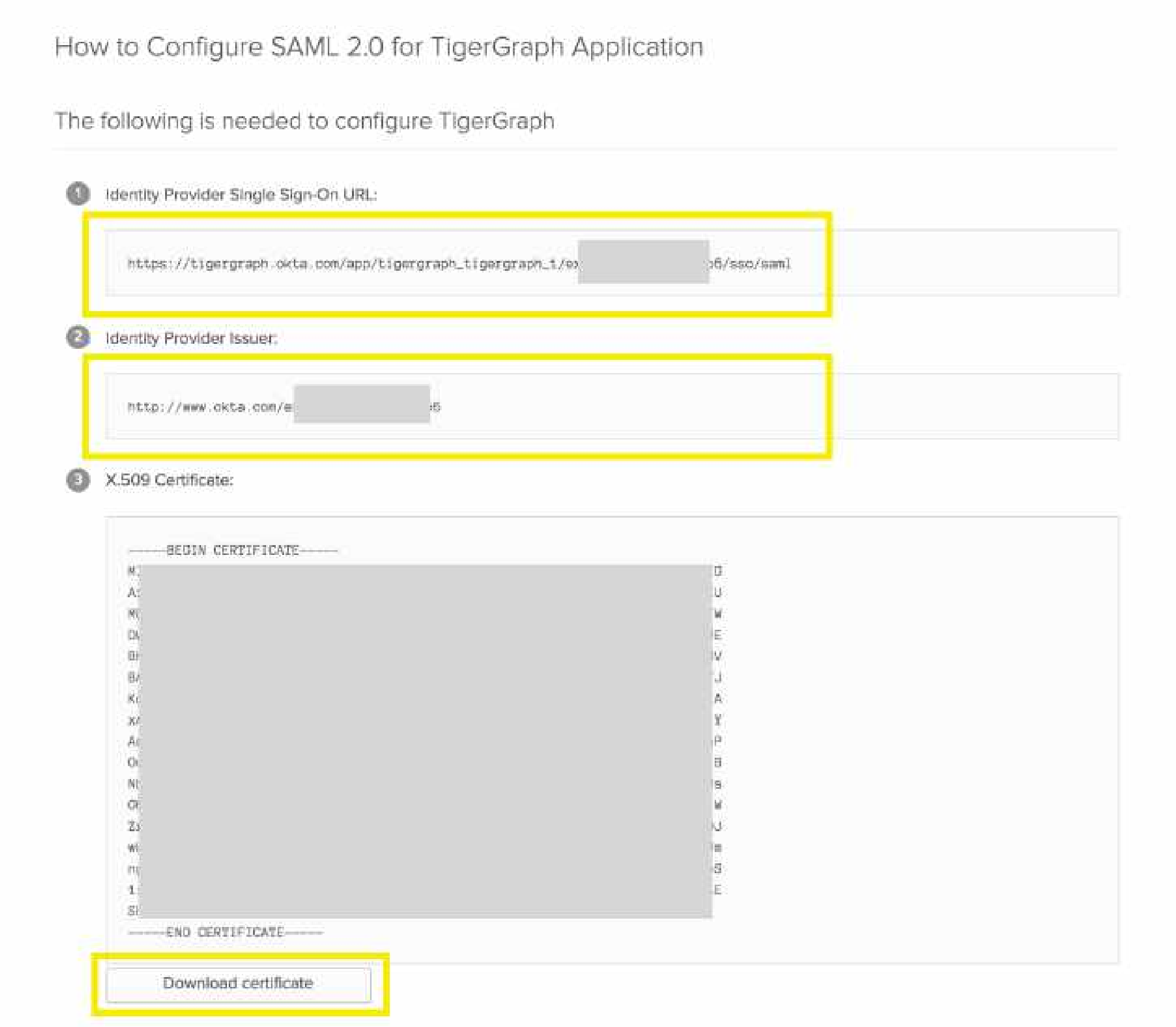
In the final step, choose whether you want to integrate your app with Okta or not. Then click Finish .



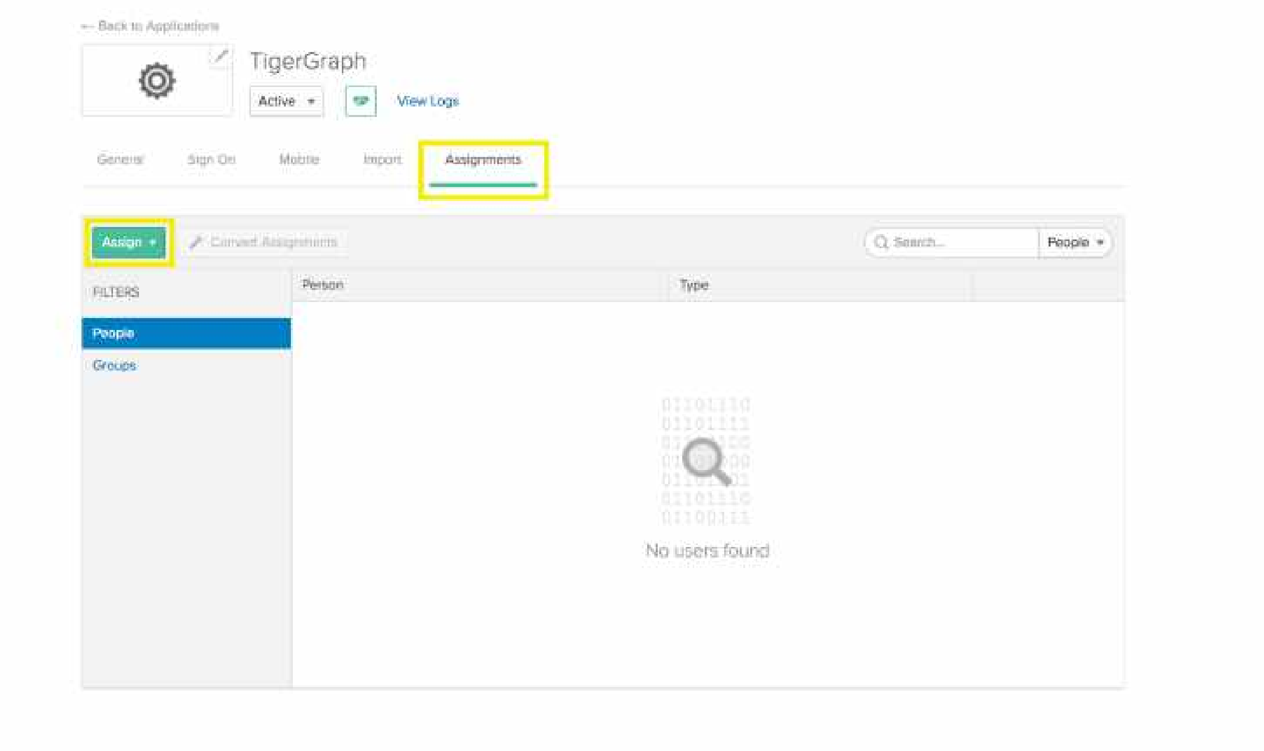
Now your Okta identity provider settings are finished. Click View Setup Instructions button to gather information you will need to setup TigerGraph Single Sign-On.



Here you want to save Identity Provider Single Sign-On URL and Identity Provider Issuer (usually known as Identity Provider Entity Id). Download the certificate file as okta.cert, rename it as idp.cert , and put it somewhere on the TigerGraph machine. Let's assume you put it under your home folder: /home/tigergraph/idp.cert. If you installed TigerGraph in a cluster, you should put it on the machine where the GSQL server is installed (usually it's the machine whose alias is m1).

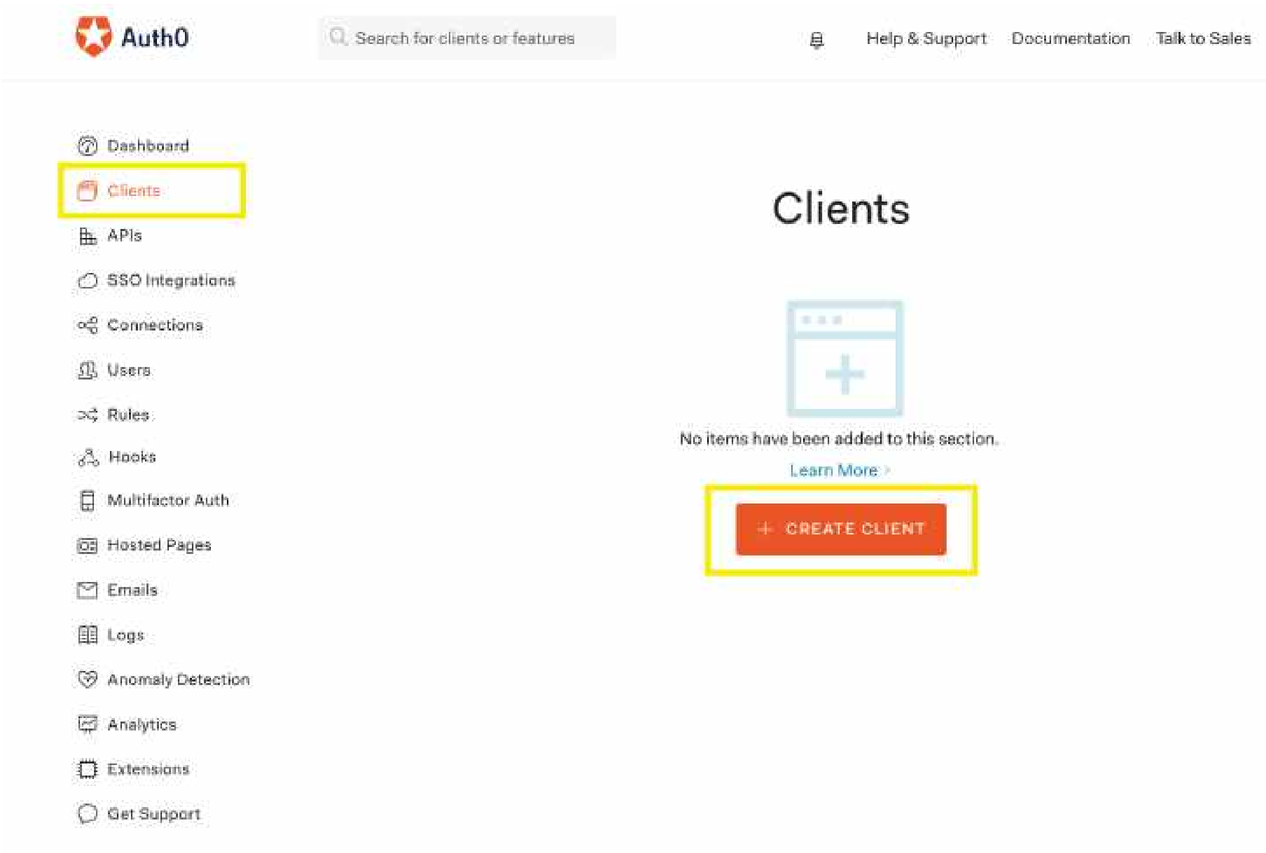


Finally, return to previous page, go to the Assignments tab, click the Assign button, and assign people or groups in your organization to access this application.

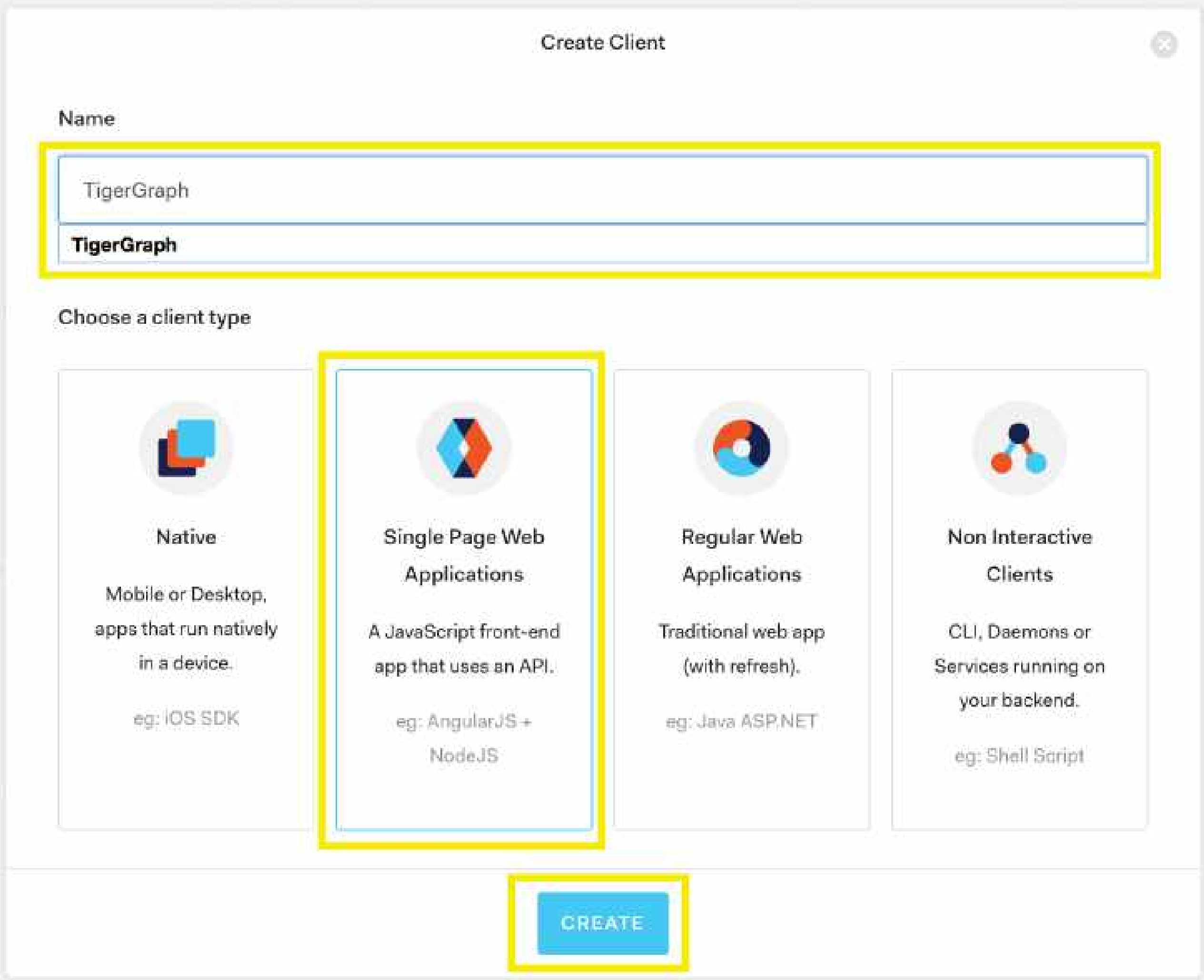


Auth0

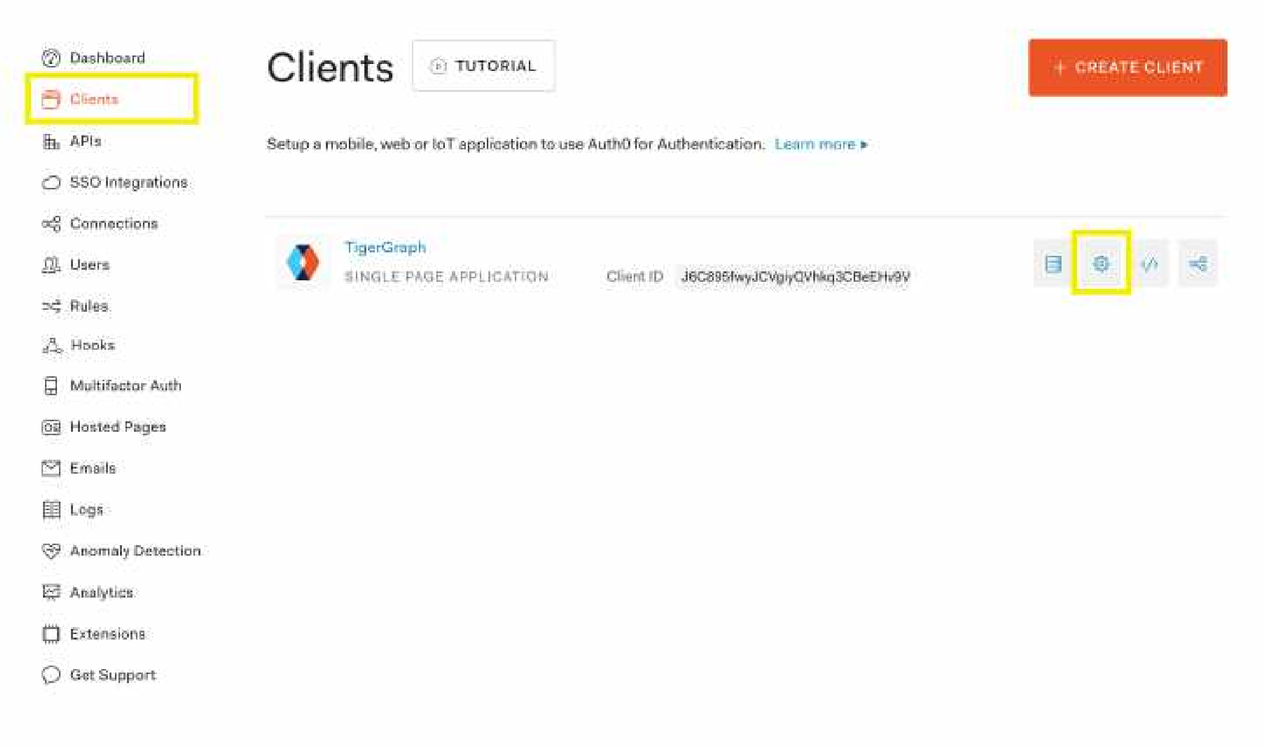
After logging into Auth0, click Clients in the left navigation bar, and then click CREATE CLIENT button.



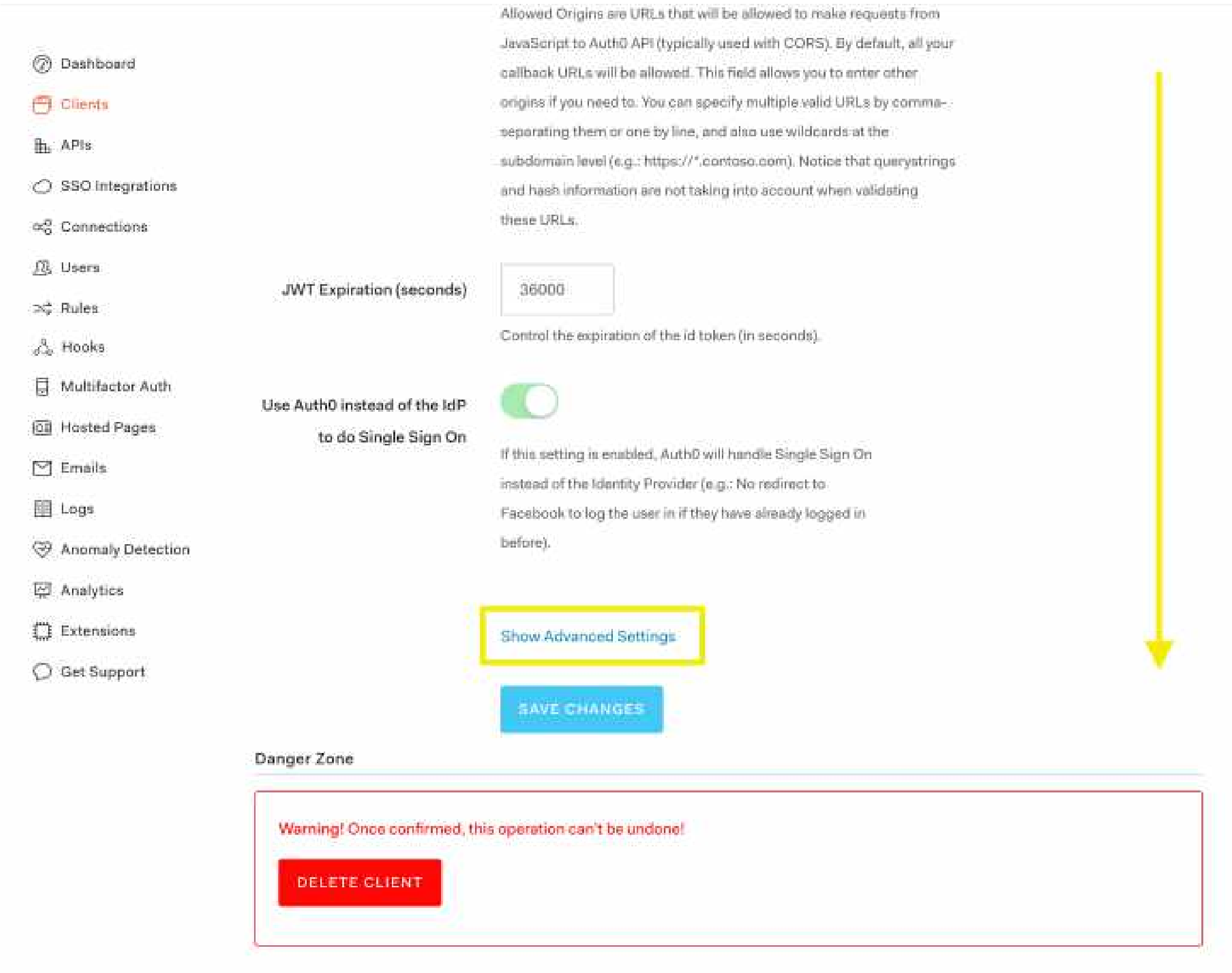
In the pop-up window, enter TigerGraph (or whatever application name you want to use) in the Name input box. Choose Single Page Web Application , and then click the CREATE button.



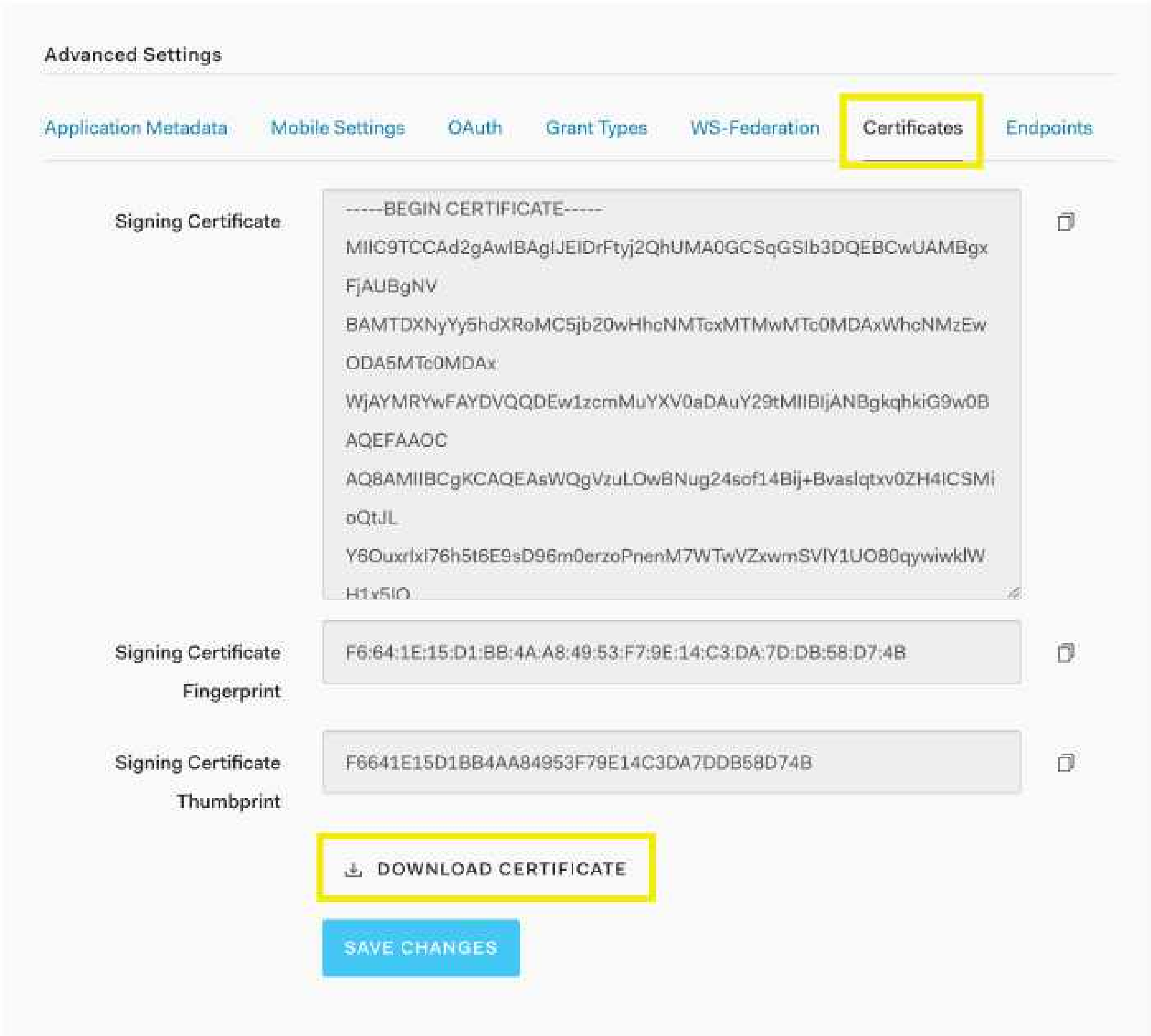
Click Clients again. In the Shown Clients list, click the settings icon of your newly created TigerGraph client.



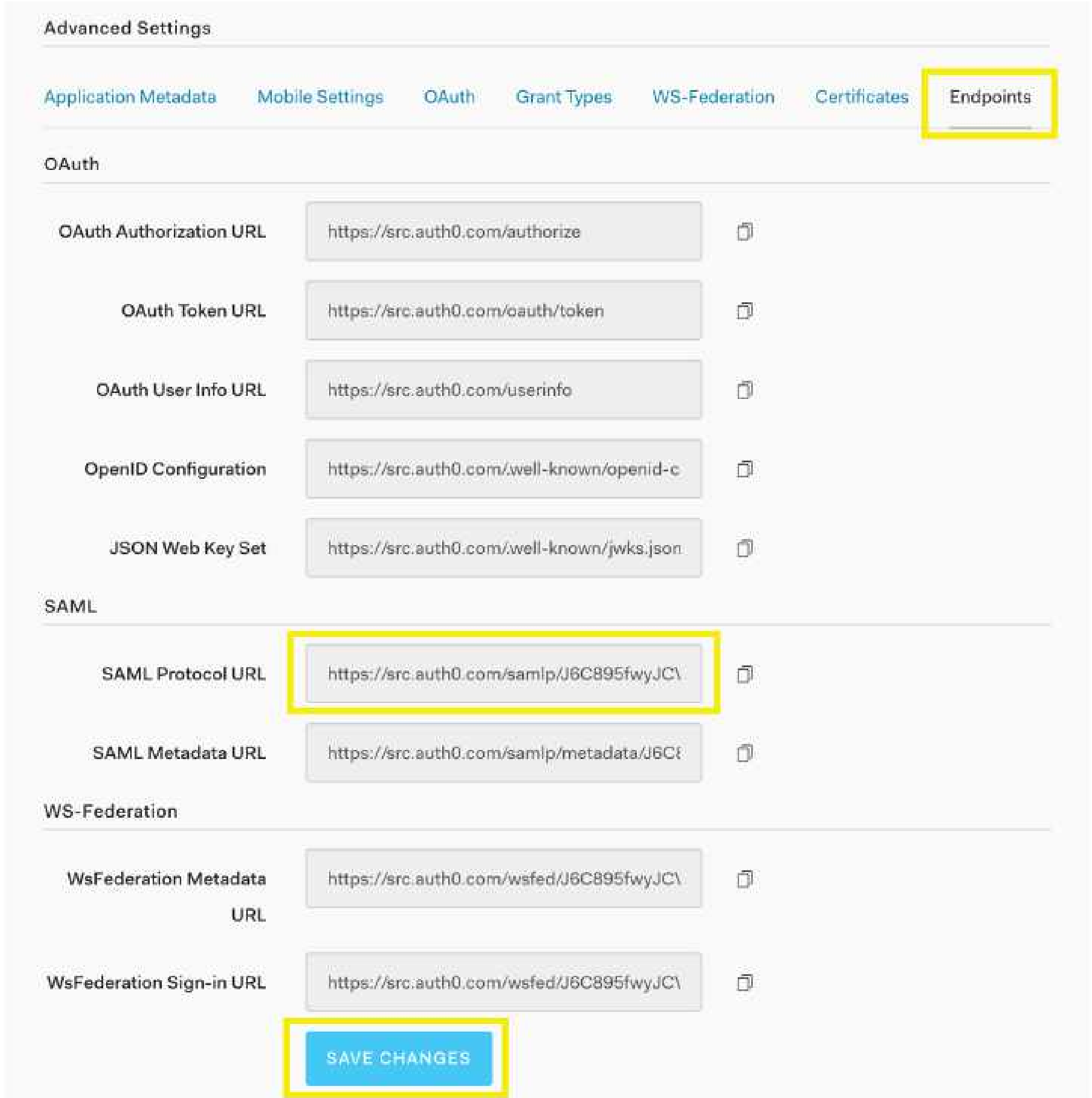
Scroll down to the bottom of the settings section, and click Show Advanced Settings .



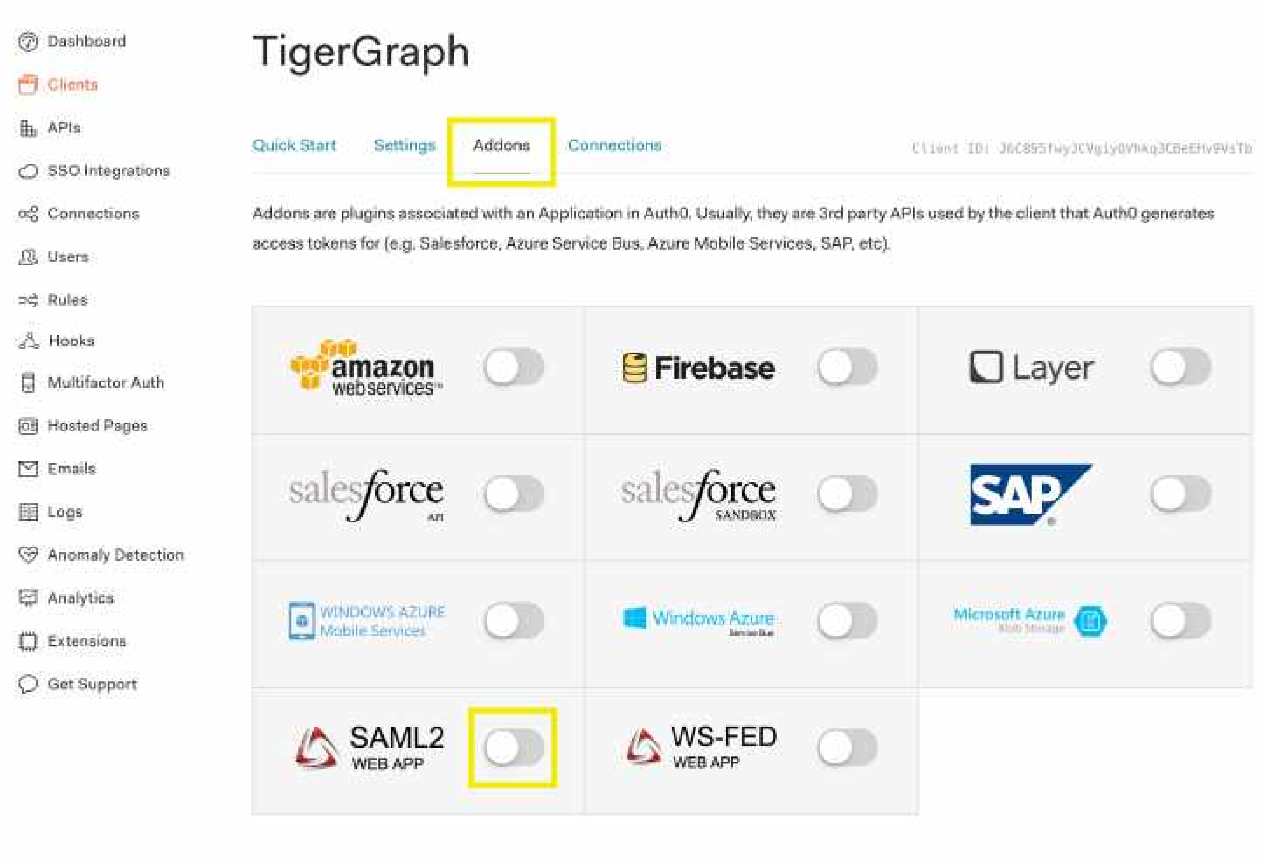
Click the Certificates tab and then click DOWNLOAD CERTIFICATE. In the chooser list, choose CER. Rename the downloaded file as idp.cert , and put it somewhere on the TigerGraph machine. Let's assume you put it under your home folder: /home/tigergraph/idp.cert. If you installed TigerGraph in a cluster, you should put it on the machine where the GSQL server is installed (usually it's the machine whose alias is m1).



Click the Endpoints tab, and copy the text in the SAML Protocol URL text box. This is the Identity Provider Single Sign-On URL that will be used to configure TigerGraph in an upcoming step.

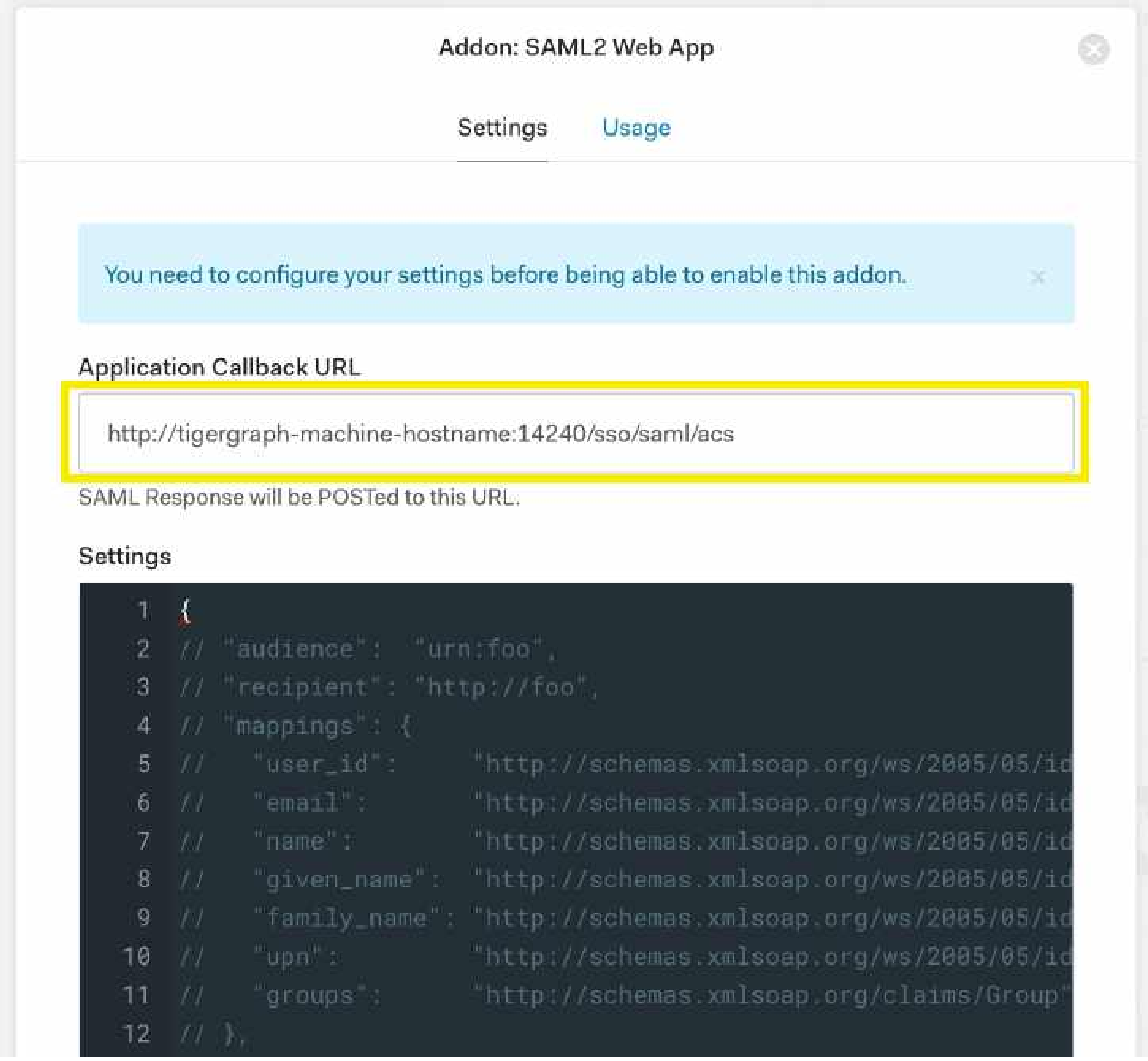


Scroll up to the top of the page, click the Addons tab, and switch on the toggle at the right side of the SAML2 card.

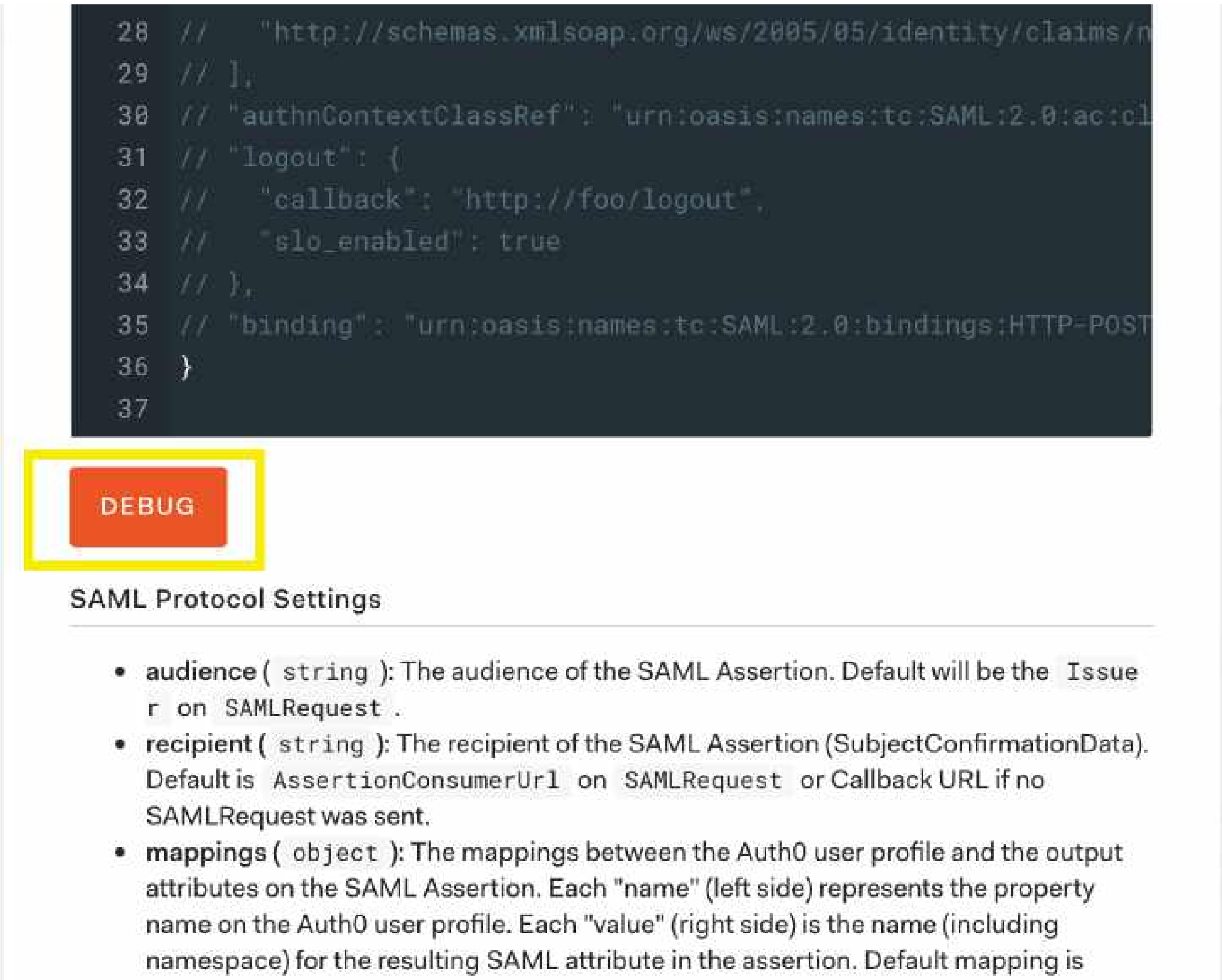


In the pop-up window, enter the Assertion Consumer Service URL in the Application Callback URL input box:

> http://tigergraph-machine-hostname:14240/api/auth/saml/acs



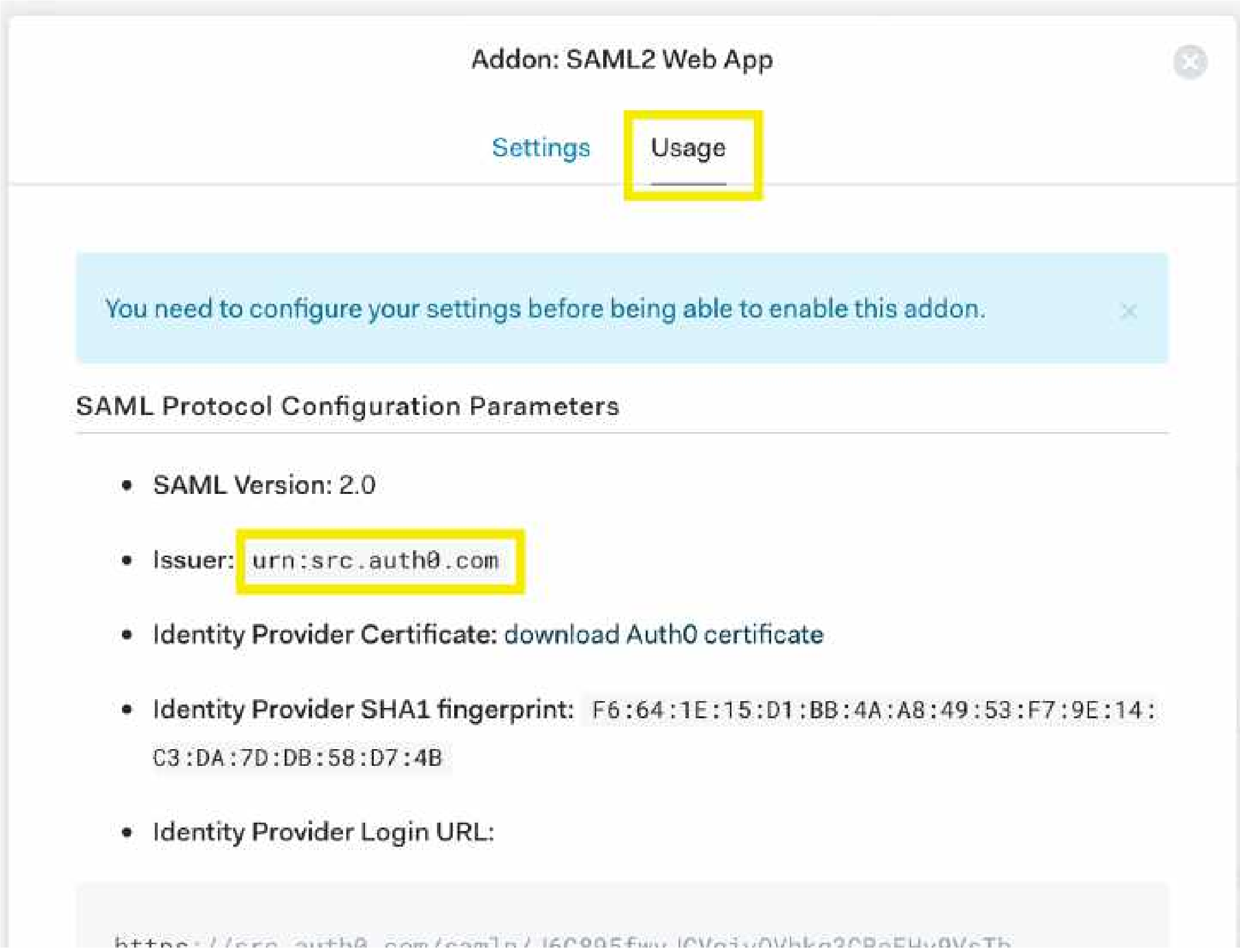
Scroll down to the end of the settings JSON code, click the DEBUG button, and log in as any existing user in your organization in the pop-up login page.



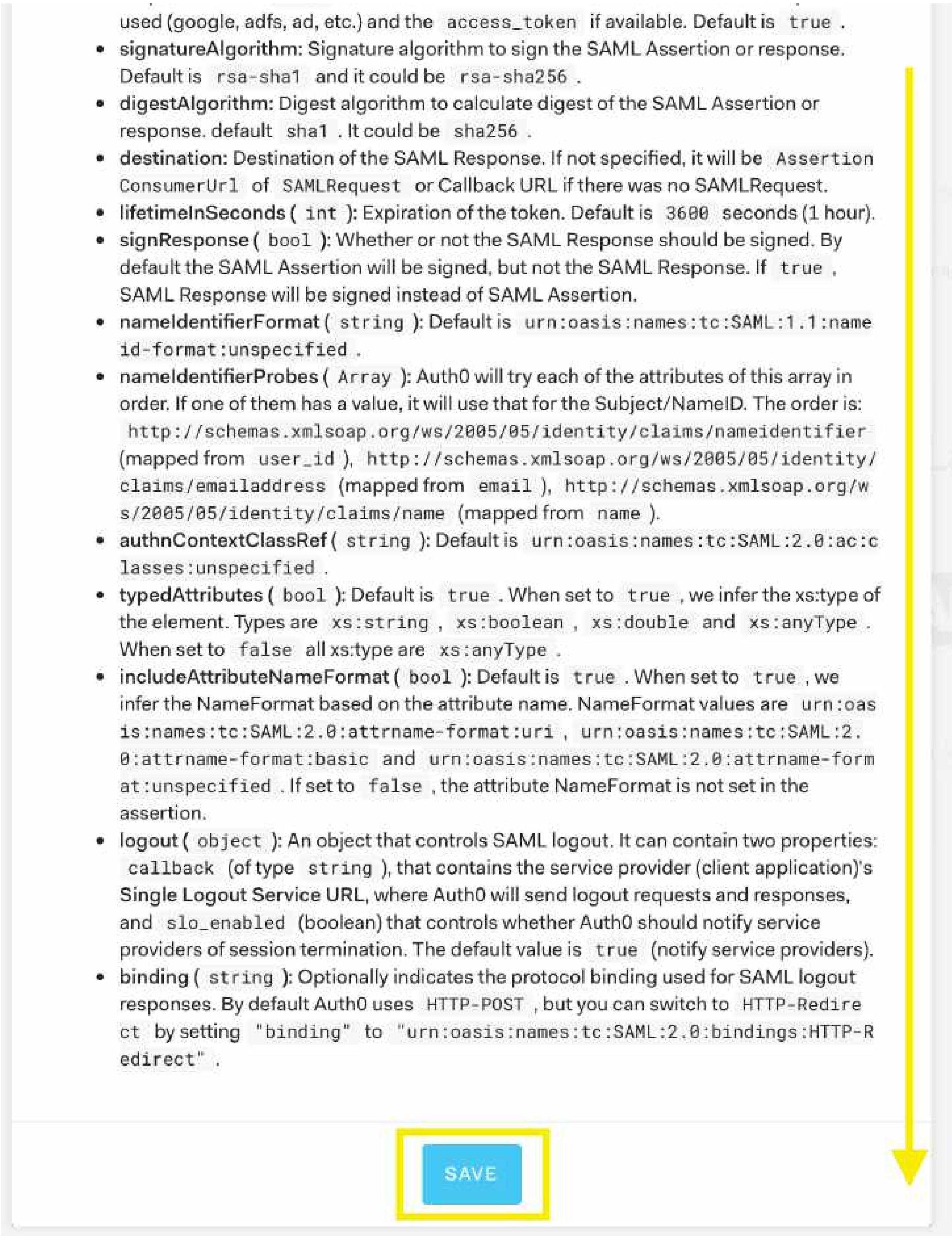
If login in successfully, the SAML response will be shown in decoded XML format. Scroll down to the attributes section. Here you will see some attribute names, which you will use to set proxy rules when creating groups in an upcoming configuration step.

```xml
  </saml:AuthnStatement>
  <saml:AttributeStatement xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/X
    <saml:Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier" NameFormat
      <saml:AttributeValue xsi:type="xs:string">auth0|5a[redacted]702</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress" NameFormat="
      <saml:AttributeValue xsi:type="xs:string">[redacted]@tigergraph.com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name" NameFormat="urn:oasi
      <saml:AttributeValue xsi:type="xs:string">[redacted]@tigergraph.com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn" NameFormat="urn:oasis
      <saml:AttributeValue xsi:type="xs:string">[redacted]@tigergraph.com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.auth0.com/identities/default/provider" NameFormat="urn:oasis:name
      <saml:AttributeValue xsi:type="xs:string">auth0</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.auth0.com/identities/default/connection" NameFormat="urn:oasis:na
      <saml:AttributeValue xsi:type="xs:string">Username-Password-Authentication</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.auth0.com/identities/default/isSocial" NameFormat="urn:oasis:name
      <saml:AttributeValue xsi:type="xs:boolean">false</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.auth0.com/email_verified" NameFormat="urn:oasis:names:tc:SAML:2.0
      <saml:AttributeValue xsi:type="xs:boolean">true</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.auth0.com/clientID" NameFormat="urn:oasis:names:tc:SAML:2.0:attrn
      <saml:AttributeValue xsi:type="xs:string">J[redacted]b</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute Name="http://schemas.auth0.com/updated_at" NameFormat="urn:oasis:names:tc:SAML:2.0:att
      <saml:AttributeValue xsi:type="xs:anyType">Fri Mar 16 2018 21:16:36 GMT+0000 (UTC)</saml:AttributeVa
    </saml:Attribute>
```


Return to the previous pop-up window and click the Usage tab. Copy the Issuer value. This is the Identity Provider Entity Id that will be used to configure TigerGraph in an upcoming step.



Click the Settings tab, scroll to the bottom of the pop-up window, and click the SAVE button. Close the pop-up window.



Enable Single Sign-On in TigerGraph

Prepare certificate and private key on TigerGraph machine

According to the SAML standard trust model, a self-signed certificate is considered fine. This is different from configuring a SSL connection, where a CA-authorized certificate is considered mandatory if the system goes to production.

There are multiple ways to create a self-signed certificate. One example is shown below.

First, use the following command to generate a private key in PKCS#1 format and a X.509 certificate file. In the example below, the Common Name value should be your server hostname (IP or domain name).

Self-Signed Certificate generation example using openssl

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /home/tiger

Generating a 2048 bit RSA private key
.....................................................................
........+++
writing new private key to '/home/tigergraph/sp-pkcs1.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a D
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:Redwood City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TigerGraph Inc
Organizational Unit Name (eg, section) []:GLE
Common Name (e.g. server FQDN or YOUR name) []: tigergraph-machine-hostna
Email Address []:support@tigergraph.com
```

Second, convert your private key from PKCS#1 format to PKCS#8 format:

```
openssl pkcs8 -topk8 -inform pem -nocrypt -in /home/tigergraph/sp-pkcs1.k
```

Enable and configure Single Sign-On Using Gadmin

From a TigerGraph machine, run the following command: gadmin config entry Security.SSO.SAML

Answering the questions is straightforward; an example is shown below.

> Since v2.3, the requirements for the Security.SSO.SAML.SP.Hostname parameter changed. The url must be a full url, starting with protocol (such as http) and ending with port number.

configure sso.saml example

```
$ gadmin config entry Security.SSO.SAML

Security.SSO.SAML.Enable [ false ]: Enable SAML2-based SSO: default false
New: true

Security.SSO.SAML.AuthnRequestSigned [ true ]: Sign AuthnRequests before
New: true

Security.SSO.SAML.AssertionSigned [ true ]: Require Identity Provider to
New: true

Security.SSO.SAML.ResponseSigned [ true ]: Require Identity Provider to s
New: false

Security.SSO.SAML.MetadataSigned [ true ]: Sign Metadata: default true
New: true

Security.SSO.SAML.SignatureAlgorithm [ rsa-sha256 ]: Signiture algorithm
New: rsa-sha256

Security.SSO.SAML.BuiltinUser [ __GSQL__saml ]: The builtin user for SAML
New: __GSQL__saml

Security.SSO.SAML.RequestedAuthnContext [  ]: Authentication context (com
New: urn:oasis:names:tc:SAML:2.0:ac:classes:Password

Security.SSO.SAML.SP.Hostname [ http://127.0.0.1:14240 ]: TigerGraph Serv
New: http://localhost:14240

Security.SSO.SAML.SP.X509Cert [  ]: Content of the x509 Certificate: defa
New: <x509 certificate>

Security.SSO.SAML.SP.PrivateKey [  ]: Content of the host machine's priva
New: <private key>

Security.SSO.SAML.IDP.EntityId [ http://idp.example.com ]: Identity Provi
New: http://idp.example.com

Security.SSO.SAML.IDP.SSOUrl [ http://idp.example.com/sso/saml ]: Single
New: http://idp.example.com/sso/saml

Security.SSO.SAML.IDP.X509Cert [  ]: Identity Provider's x509 Certificate
New: /home/tigergraph/idp.cert
```

The reason we change Security.SSO.SAML.ResponseSigned to false is because some identity providers (e.g., Auth0) don't support signed assertion and response at the same time. If your identity provider supports signing both, we strongly suggest you leave it as true.

After making the configuration settings, apply the config changes, and restart gsql.

```
$ gadmin config apply -y
$ gadmin restart gsql -y
```

Create user groups with proxy rules to authorize Single Sign-On users

In order to authorize Single Sign-On users, you need create user groups in GSQL with proxy rules and grant roles on graphs for the user groups.

In TigerGraph Single Sign-On, we support two types of proxy rules:

- nameid equations
- attribute equations

Attribute equations are more commonly used because usually user group information is transferred as attributes to your identity provider SAML assertions. In the Okta identity provider configuration example, it is transferred by the attribute statement named group. By granting roles to a user group, all users matching the proxy rule will be granted all the privileges of that role. In some cases if you want to grant one specific Single Sign-On user some privilege, you can use a nameid equation to do so.

Required privilege

- `WRITE_PROXYGROUP` for creating proxy groups.
- `WRITE_ROLE` for granting roles to users

Single User Proxy

For example, if you want to create a user group SuperUserGroup that contains the user with nameid admin@your.company.com only, and grant superuser role to that user, you can do so with the following command:

```
GSQL > CREATE GROUP SuperUserGroup PROXY "nameid=admin@your.company.com"
GSQL > GRANT ROLE superuser TO SuperUserGroup
Role "superuser" is successfully granted to user(s): SuperUserGroup
```

User Group Proxy

Suppose you want to create a user group HrDepartment which corresponds to the identity provider Single Sign-On users having the group attribute value "hr-department", and want to grant the queryreader role to that group on the graph HrGraph:

```
GSQL > CREATE GROUP HrDepartment PROXY "group=hr-department"
GSQL > GRANT ROLE queryreader ON GRAPH HrGraph TO HrDepartment
Role "queryreader" is successfully granted to user(s): HrDepartment
```

Change Password Of Default User

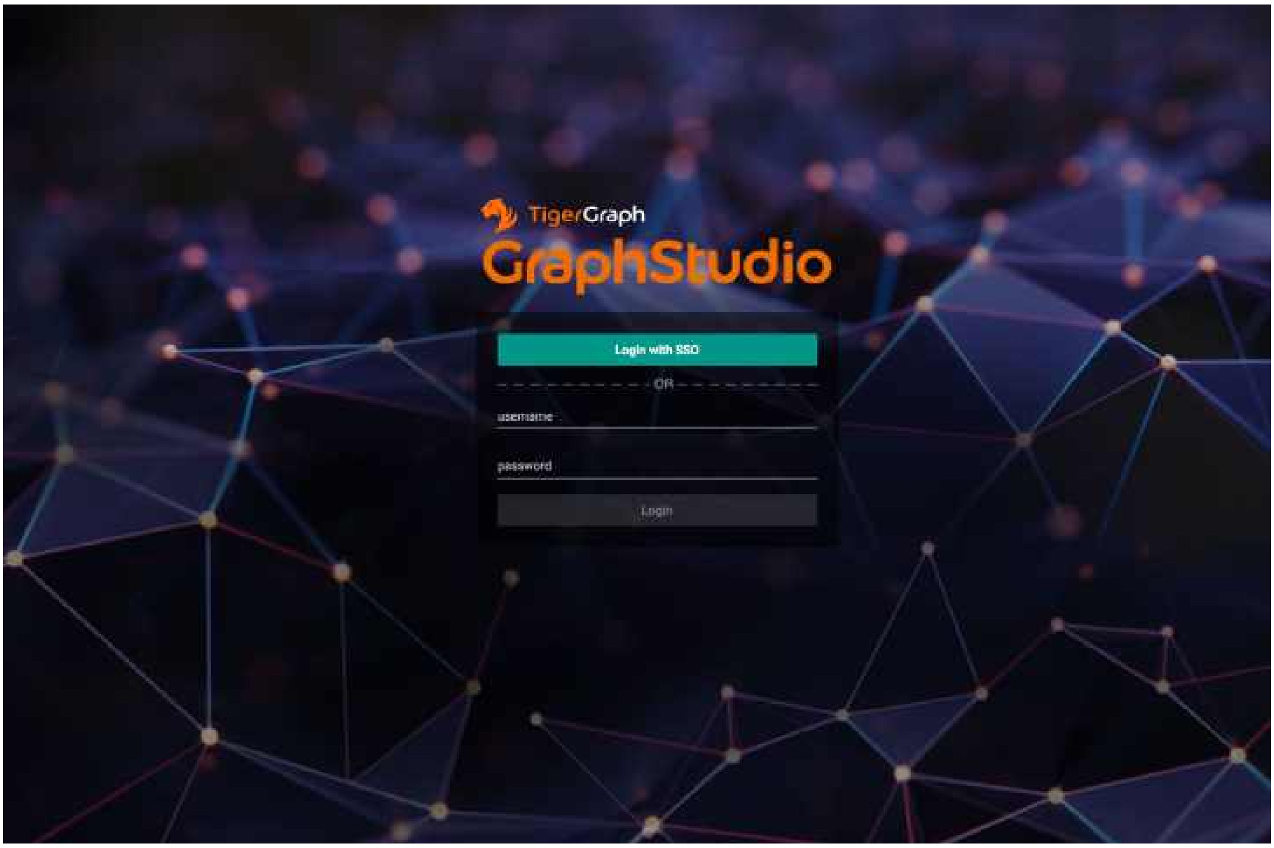
Don't forget to enable User Authorization in TigerGraph by changing the password of the default superuser tigergraph to other than its default value. If you do not change the password, then every time you visit the GraphStudio UI, you will automatically log in as the superuser tigergraph.

```
GSQL > change password
New Password : ********
Re-enter Password : ********
Password has been changed.
GSQL > exit
```

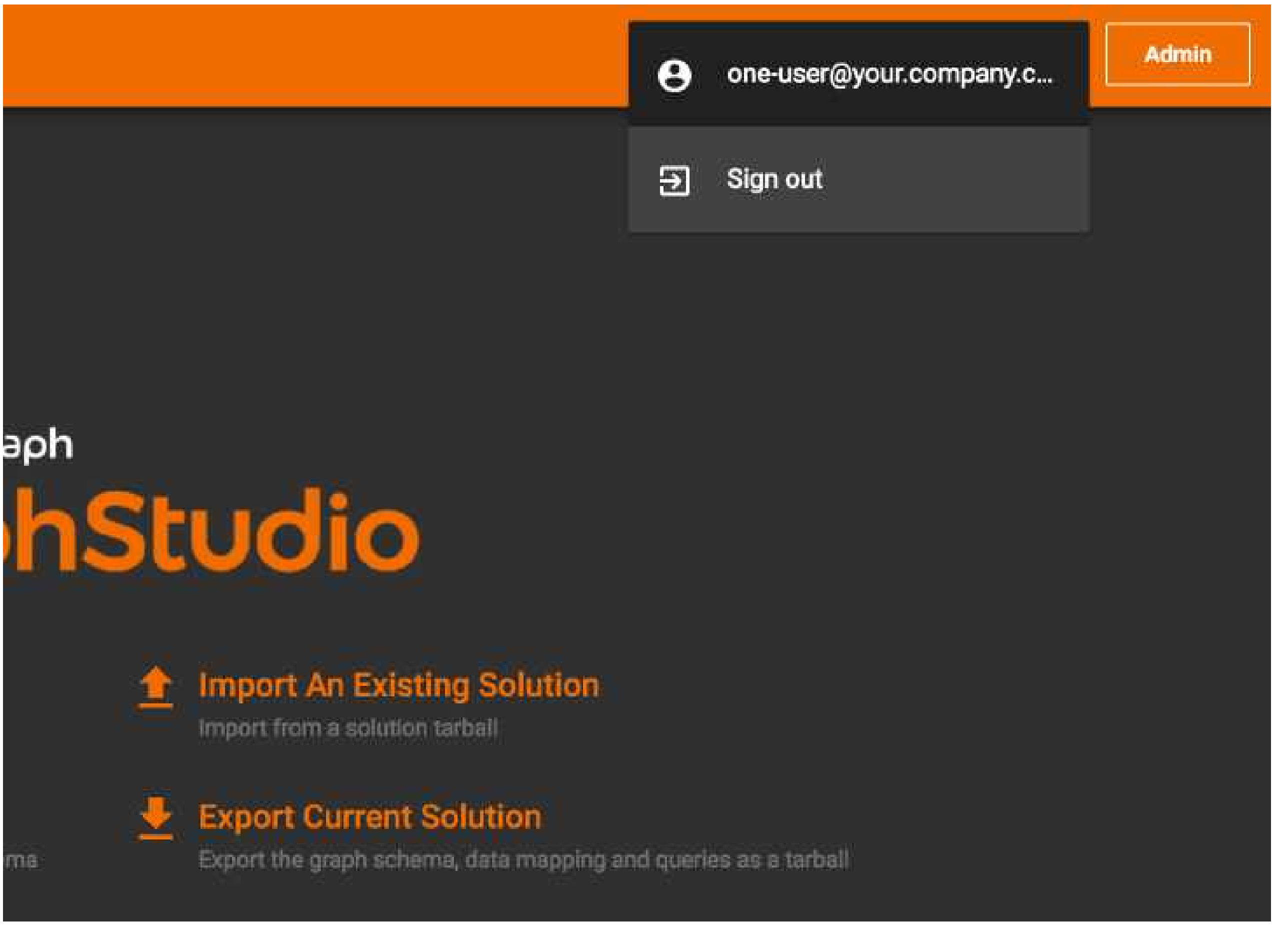
Testing Single Sign-On

Now you have finished all configurations for Single Sign-On. Let's test it.

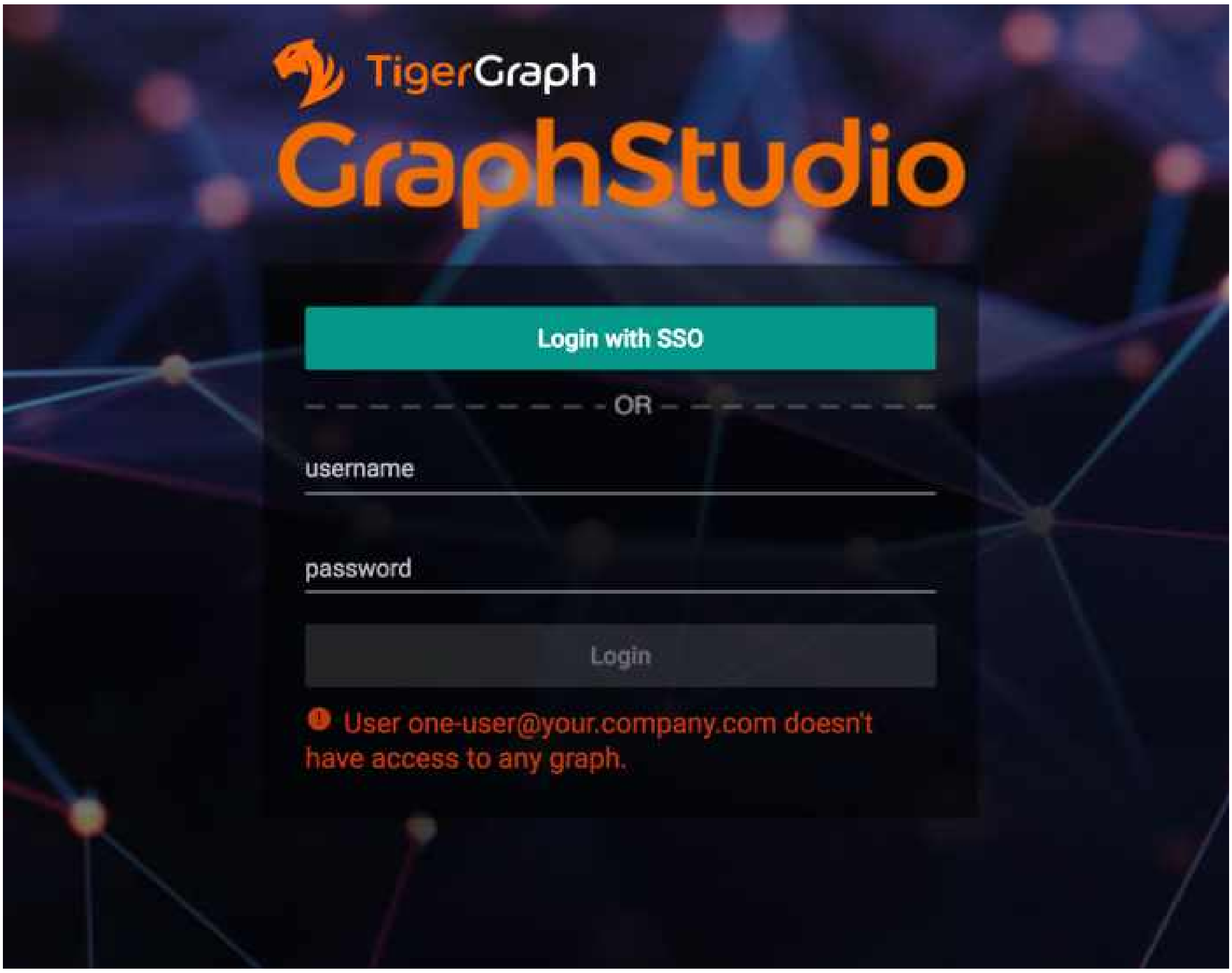
Visit the GraphStudio UI in your browser. You should see a Login with SSO button appear on top of the login panel:



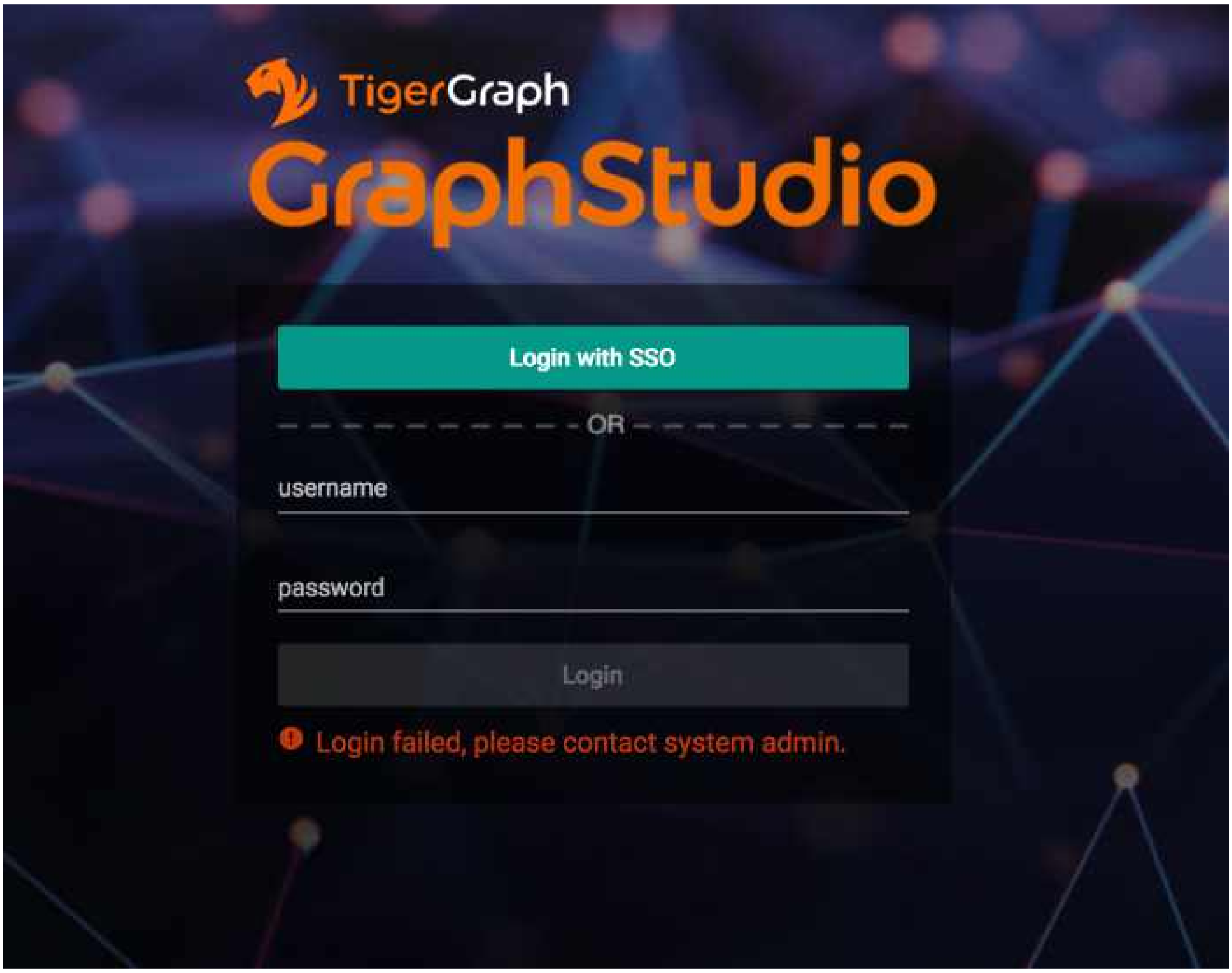
Clicking the button will navigate to your identity provider's login portal. If you have already logged in there, you will be redirected back to GraphStudio immediately. After about 10 seconds, the verification should finish, and you are authorized to use GraphStudio. If you haven't login at your identity provider yet, you will need to log in there. After logging in successfully, you will see your Single Sign-On username when you click the User icon at the upper right of the GraphStudio UI.



If after redirecting back to GraphStudio, you return to the login page with the error message shown below, that means the Single Sign-On user doesn't have access to any graph. Please double check your user group proxy rules, and roles you have granted to the groups.



If your Single Sign-On fails with error message show below, that means either some configuration is inconsistent between TigerGraph and your identity provider, or something unexpected happened.



You can check your GSQL log to investigate. First, find your GSQL log file with the following:

```
$ gadmin log gsql
GSQL   : /home/tigergraph/tigergraph/log/gsql/log.INFO
```

Then, grep the SAML authentication-related logs:

```
cat /home/tigergraph/tigergraph/log/gsql/log.INFO | grep SAMLAuth
```

Focus on the latest errors. Usually the text is self-descriptive. Follow the error message and try to fix TigerGraph or your identity provider's configuration. If you encounter any errors that are not clear, please contact support@tigergraph.com ↗.

Cluster Resizing

As your workload changes, it makes sense to resize your cluster to meet your production needs. The section describes the steps to perform the following cluster resizing operations:

- Expand a cluster
- Downsize a cluster
- Repartition a cluster

Cluster Expansion

As your workload changes, you can expand your cluster to improve its query performance, system availability, and fault tolerance. Expanding a cluster adds more nodes to the cluster. During an expansion, you can also change the replication factor of your cluster.

Before you begin

- TigerGraph must already be installed on the new nodes in **exactly the same version** as the cluster.
- No loading jobs, queries, or REST requests are running on the new node or the cluster.
- Obtain a few key measures for the state of your data before the expansion, such as vertex counts/edge counts or certain query results. This will be useful in verifying data integrity after the expansion completes.
- If the original cluster is a single node installation, make sure the IP used is not a local loopback address ↗ such as 127.0.0.1.

Procedure

Step 1: Identify new cluster replication and partition

Before running any commands to expand a cluster, make sure you have a clear idea of how the new cluster should be distributed. You should have the following information:

- The new replication factor of the cluster
- The new partitioning factor of the cluster
- The IP addresses of the new nodes to be added to the cluster

Step 2: Expand the cluster

To expand the cluster, run the `gadmin cluster expand` command like below. If the expansion involves changing the replication factor, use the `--ha` option to indicate the new replication factor:

```
$ gadmin cluster expand node_ip_list [--ha <replication_factor>]
```

`node_ip_list` is the machine aliases of the nodes you are adding to the cluster mapped to their IP addresses with a colon(`:`), and separated by a comma. Below is an example:

```
$ gadmin cluster expand m3:10.128.0.81,m4:10.128.0.82 --ha 1
```

We suggest naming the new nodes following the convention of `m<serial>`, such as `m1`, `m2`, and `m3` for a 3-node cluster. If you are adding a fourth node, then the fourth node would be named `m4`. If you decide to name them differently, make sure that all names are unique within the cluster.

Supply a staging location

Extra disk space is required during cluster expansion. If more space is not available on the same disk, you can supply a staging location on a different disk to hold temporary data:

```
$ gadmin cluster expand m3:192.168.1.3,m4:192.168.1.4 --stagingPath /tmp/
```

If you choose to supply a staging location, make sure that the TigerGraph Linux user has write permission to the path you provide. The overall amount of space required for expansion on each node is `(1 + Math.ceil(oldPartition/newPartition) ) * dataRootSize`. `oldPartition` and `newPartition` each stand for the partitioning factor of the cluster before and after expansion; `dataRootSize` stands for the size of the data root folder on the node.

For example, if you are expanding from a 6 node cluster with a replication factor of 2 and a partitioning factor of 3, to a 10-node cluster with a replication factor of 2 and a partitioning factor of 5, and the size of the data root folder on a node is 50GB. Then you would need more than `(1 + Math.ceil(3/5)) * 50) = 100 GB` of free space on the staging path.

Step 3: Verify success and delete temporary files

When the expansion completes, you should see a message confirming the completion of the cluster change. The message will also include the location of the temporary files created during the expansion.

Verify data integrity by comparing vertex/edge counts or query results. After confirming a successful expansion, delete the temporary files to free up disk space.

Cluster Shrinking

Shrinking a cluster can make sense for a few reasons:

- Your workload has changed and you can operate a cluster with fewer resources.
- Your data volume is lower than projected.

Shrinking a cluster removes nodes from the cluster. The data stored on those nodes will be redistributed to the remaining nodes.

Before you begin

- No loading jobs, queries, or REST requests are running on the new node or the cluster.
- Obtain a few key measures for the state of your data before shrinking, such as vertex counts/edge counts or certain query results. This will be useful in verifying data integrity after shrinking completes.

Procedure

Step 1: Identify new cluster replication and partition

Before running any commands to shrink a cluster, make sure you have a clear idea of how the new cluster should be distributed. You should have the following information:

- The new replication factor of the cluster
- The new partitioning factor of the cluster
- The names and IP addresses of the nodes to be removed from the cluster

Step 2: Shrink the cluster

To shrink the cluster, run the `gadmin cluster shrink` command like below. If the shrinking involves changing the replication factor, use the `--ha <replicattion_factor>` option to indicate the new replication factor:

```
$ gadmin cluster shrink node_ip_list [--ha <replication_factor>]
```

`node_ip_list` is the list of nodes you are removing from the cluster mapped to their IP addresses with a colon(`:`), and separated by a comma. Below is an example:

```
$ gadmin cluster shrink m3:10.128.0.81,m4:10.128.0.82 --ha 1
```

Supply a staging location

Extra disk space is required during cluster shrinking. If more space is not available on the same disk, you can supply a staging location on a different disk to hold temporary data:

```
$ gadmin cluster shrink m3:192.168.1.3,m4:192.168.1.4 --stagingPath /tmp/
```

If you choose to supply a staging location, make sure that the TigerGraph Linux user has write permission to the path you provide. The overall amount of space required for cluster shrinking on each node is `(1 + Math.ceil(oldPartition/newPartition) ) * dataRootSize`. `oldPartition` and `newPartition` each stand for the partitioning factor of the cluster before and after shrinking; `dataRootSize` stands for the size of the data root folder on the node.

For example, if you are shrinking from a 10-node cluster with a replication factor of 2 and a partitioning factor of 5, to a 6-node cluster with a replication factor of 2 and a partitioning factor of 3, and the size of the data root folder on a node is 50GB. Then you would need more than `(1 + Math.ceil(5/3)) * 50) = 150 GB` of free space on the staging path.

Step 3: Verify success and delete temporary files

When shrinking completes, you should see a message confirming the completion of the cluster change. The message will also include the location of the temporary files created during the operation.

Verify data integrity by comparing vertex/edge counts or query results to what they were before the shrinking. After confirming a successful cluster contraction, delete the temporary files to free up disk space.

Step 4: Uninstall TigerGraph on the removed nodes

Uninstall TigerGraph from the removed nodes by removing all root directories (`app`, `data` and `tmp`) except the log directory of TigerGraph.

> For security reasons, `guninstall` is disabled on the removed nodes. Therefore, you need to delete TigerGraph by manually deleting the root directories.

Cluster Repartition

You can repartition a cluster to meet the changing needs of your workload. Repartitioning a cluster changes the replication factor and the partitioning factor of your cluster without adding or removing nodes from the cluster.

You can also change the replication factor when you expand or shrink a cluster. See Expand a Cluster and Shrink a Cluster.

Before you begin

- No loading jobs, queries, or REST requests are running on the cluster.
- Obtain a few key measures for the state of your data before the repartition, such as vertex counts/edge counts or certain query results. This will be useful in verifying data integrity after the repartition completes.

Procedure

Step 1: Calculate new replication factor

To repartition a cluster, calculate the new replication factor for your cluster. You can calculate the replication factor by dividing the total number of nodes in the cluster by the number of partitions you'd like your cluster to have.

Step 2: Repartition the cluster

To repartition a cluster, use the `gadmin cluster expand` command like below. Use the `--ha` option to indicate the new replication factor of the cluster. For example, the command below will change the replication factor to 2:

```
$ gadmin cluster expand --ha 2
```

The partitioning factor of your cluster will change automatically based on your specified replication factor. Its updated value will be the total number of nodes divided by the replication factor.

If the total number of nodes cannot be fully divided by the replication factor, the remainder nodes will be left idle. For example, if you currently have a 5-node cluster with a replication factor of 1 and a partitioning factor of 5. Changing the replication factor to 2 without adding new nodes will change the distribution of your cluster to be 2*2, with one node being left idle. To avoid nodes being left idle, ensure that you pick a replication factor that can fully divide the total number of nodes you have in the cluster.

Supply a staging location

Extra disk space is required during cluster repartition. If more space is not available on the same disk, you can supply a staging location on a different disk to hold temporary data:

```
$ gadmin cluster expand --stagingPath /tmp/
```

If you choose to supply a staging location, make sure that the TigerGraph Linux user has write permission to the path you provide. The overall amount of space required for cluster repartition on each node is `(1 + Math.ceil(oldPartition/newPartition) ) * dataRootSize`. `oldPartition` and `newPartition` each stand for the partitioning factor of the cluster before and after repartition; `dataRootSize` stands for the size of the data root folder on the node.

For example, if you are repartitioning from a 6-node cluster with a replication factor of 2 and a partitioning factor of 3, to a 6-node cluster with a replication factor of 3 and a partitioning factor of 2, and the size of the data root folder on a node is 50GB. Then you would need more than `(1 + Math.ceil(3/2)) * 50) = 150 GB` of free space on the staging path.

Step 3: Verify success and delete temporary files

When the repartition completes, you should see a message confirming the completion of the cluster change. The message will also include the location of the temporary files created during the repartition.

Verify data integrity by comparing vertex/edge counts or query results. After confirming a successful repartition, delete the temporary files to free up disk space.

Data Encryption

Encryption for Data at Rest and Data in Motion

Encrypting Connections

TigerGraph supports secure data-in-flight communication, using SSL/TLS encryption protocol. This applies to any outward-facing channel, including GSQL clients, RESTPP endpoints, and the GraphStudio web interface. When SSL/TLS is enabled, HTTPS takes the place of HTTP for RESTPP and GraphStudio connections.

Prerequisites

You should have basic knowledge about how SSL works:

1. What the SSL certificate and key are used for
2. That an SSL certificate is bound to a domain
3. How an SSL certificate chain works

A good primer on SSL is available to https://httpd.apache.org/docs/2.4/ssl/ssl_intro.html

Nginx-Based

TigerGraph uses the Nginx web server, so SSL configuration makes use of some built-in support in Nginx.

http://nginx.org/en/docs/http/configuring_https_servers.html

Step 1. Obtain an SSL Certificate

The two main options for obtaining an SSL Certificate are to generate your own self-signed certificate or to purchase a certificate from a trusted Certificate Authority. Regardless of which method you choose, your certificate should be chained to a trusted root certificate embedded in your browser. The options and details for producing a trusted SSL certificate are beyond the scope of this document. The focus of this document is how to configure your TigerGraph system to use the certificate to enable SSL.

Option 1: Using a Certificate From A Trusted Agent

First, obtain an SSL certificate from a trusted agent of your choice. Certificate vendors will provide clear instructions for ordering a certificate and then for installing it on your system.

Then you can configure the certificate with `gadmin config entry ssl`

Option 2: Create a Self-Signed Certificate

There are multiple ways to create a self-signed certificate. One example is shown below.

For simplicity, the method below will use the root certificate directly as the HTTPS server certificate. This method is satisfactory for testing but should not be used for a production system.

In the example below, the Common Name value should be your server hostname, since HTTPS certificates are bound to domain names.

Self-Signed Certificate generation example using openssl

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ~/nginx-sel

Generating a 2048 bit RSA private key
..........................................................................
........+++
writing new private key to '/home/tigergraph/nginx-selfsigned.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a D
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:Redwood City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TigerGraph
Organizational Unit Name (eg, section) []:GLE
Common Name (e.g. server FQDN or YOUR name) []: my.ip.addr.num
Email Address []:engineer@tigergraph.com
```

Change the Certificate Permission

For security reasons, the certificates can only be used with permission 600 or less.

```
$ chmod 600 ~/nginx-selfsigned.*
```

Step 2: Configure SSL with `gadmin`

With the self-signed certificate successfully generated, you can configure it with `gadmin`, so that all the HTTP traffic will be protected with SSL.

> TigerGraph's SSL only accepts PEM-encoded certificates. If you have a certificate encoded in other formats (e.g. DER), you need to convert it to a PEM-encoded certificate ↗ first.

```
$ gadmin config entry ssl

Nginx.SSL.Enable [ false ]: Enable SSL connection for all HTTP requests
New: true

Nginx.SSL.Key [ <masked> ]: Private key for SSL
New: @privateKey_file_path

Nginx.SSL.Cert [ <masked> ]: Public certificate for SSL
New: @ssl_cert_path

# This config (Nginx.ProxySSLVerify) was removed in v3.1.1
Nginx.ProxySSLVerify [ false ]: Enable verification of the proxied HTTPS
New: true
```

After saving the settings, apply the configuration settings.

```
gadmin config apply -y

[ Info] Successfully applied configuration change. Please restart service
```

Then restart the following services: `gsql`, `nginx`, `ts3`, and `gui`.

```
$ gadmin restart gsql nginx gui ts3 -y
```

Testing Your SSL Connection

Now you may test the connection.

A direct curl request to the server will fail due to certificate verification failure:

```
$ curl https://localhost:14240

curl: (60) server certificate verification failed. CAfile: /etc/ssl/certs
More details here: http://curl.haxx.se/docs/sslcerts.html

curl performs SSL certificate verification by default, using a "bundle"
of Certificate Authority (CA) public keys (CA certs). If the default
bundle file isn't adequate, you can specify an alternate file
using the --cacert option.

If this HTTPS server uses a certificate signed by a CA represented in
the bundle, the certificate verification probably failed due to a
problem with the certificate (it might be expired, or the name might
not match the domain name in the URL).

If you'd like to turn off curl's verification of the certificate, use
the -k (or --insecure) option.
```

In v1.2, the default TCP/IP port for Nginx has changed from 44240 to 14240, to avoid possible port conflicts with Zookeeper.

You may use the -k option to turn off the verification, but it is unsafe and not recommended.

To successfully make requests with curl, you will need to specify the certificate by using the `--cacert` parameter:

```
$ curl --cacert /home/tigergraph/nginx-selfsigned.crt https://localhost:1

<!doctype html><html lang="en"><head><meta charset="utf-8"><title>GraphSt
```

Encrypting Data At Rest

Encryption Levels

The TigerGraph graph data store uses a proprietary encoding scheme which both compresses the data and obscures the data unless the user knows the encoding/decoding scheme. In addition, the TigerGraph system supports integration with industry-standard methods for encrypting data when stored in disk ("data at rest").

Data at rest encryption can be applied at many different levels. A user can choose to use one or more level.

Encryption Level	Description	TigerGraph Support
Hardware	Use specialized hard disks which perform automatic encryption on write and decryption on read (by authorized OS users)	Invisible to TigerGraph
Kernel-level file system	Use Linux built-in utilities to encrypt data. Root privilege required.	Invisible to TigerGraph
User-level file system	Use Linux built-in utilities and customized libraries to encrypt data. Root privilege is not required.	Invisible to TigerGraph

Kernel-level Encryption

File system encryption employs advanced encryption algorithms. Some tools allow the user to select from a menu of encryption algorithms. It can be done either in kernel mode or user mode. To run in kernel mode, superuser permission is required.

Since Linux 2.6, device-mapper has been an infrastructure, which provides a generic way to create virtual layers of block devices with transparent encryption blocks using the kernel crypto API.

In Ubuntu, full-disk encryption is an option during the OS installation process. For other Linux distributions, the disk can be encrypted with dm-encrypt ↗.

A commonly used utility is eCryptfs ↗, which is licensed under GPL, and it is built into some kernels, such as Ubuntu.

User-Level Encryption

If root privilege is not available, a workaround is to use FUSE (Filesystem in User Space) to create a user-level filesystem running on top of the host operating system. While the performance may not be as good as running in kernel mode, there are more options available for customization and tuning.

Example 1: Kernel-mode file system encryption with dm-crypt

In this example, we use dm-crypt to provide kernel-mode file system encryption. The dm-crypt utility is widely available and offers a choice of encryption algorithms. It also can be set to encrypt various units of storage – full disk, partitions, logical volumes, or files.

The basic idea of this solution is to create a file, map an encrypted file system to it, and mount it as a storage directory for TigerGraph with R/W permission only to authorized users.

Prerequisites

Before you start, you will need a Linux machine on which

- you have root permission,
- the TigerGraph system has not yet been installed,
- and you have sufficient disk space for the TigerGraph data you wish to encrypt. This may be on your local disk or on a separate disk you have mounted.

Instructions

- Install cryptsetup (cryptsetup is included with Ubuntu, but other OS users may need to install it with yum).
- Install the TigerGraph system.
- Grant sudo privilege to the TigerGraph OS user.
- Stop all TigerGraph services with the following commands:
  gadmin stop all -y
  gadmin stop admin -y
- Acting as the tigergraph OS user, run the following export commands to set variables. Replace the placeholders enclosed in angle brackets <...> with the values of your choice:

```
# The username for TigerGraph Database System, for example: tigergraph
export db_user='<username>'

# The path of encrypted file to be created for TigerGraph storage, for ex
export encrypted_file_path='<path-to-encrypted-file>'

# The size of encrypted file to be created (used by dd command), for exam
export encrypted_file_size=<storage-size>

# The password for the encrypted file, for example: DataAtRe5tPa55w0rd
export encryption_password='<password>'

# The root directory for tigergraph, for example: $HOME/tigergraph
export tigergraph_data_root="<tigergraph-data-root>"

# Set the first available loop device for encrypted file mapping
export loop_device=$(losetup -f)
```

- Create a file for TigerGraph data storage.

```
dd of=$encrypted_file_path bs=$encrypted_file_size count=0 seek=1
```

- Change the permission of the file so that only the owner of the file (that is, only the tigergraph user who created the file in the previous step) will be able to access it:

```
chmod 600 $encrypted_file_path
```

- Associate a loopback device with the file:

```
sudo losetup $loop_device $encrypted_file_path
```

- Encrypt storage in the device. cryptsetup will use the Linux device mapper to create, in this case, $encrypted_file_path . Initialize the volume and set a password interactively with the password you set to $encryption_password :

```
sudo cryptsetup -y luksFormat $loop_device
```

If you are trying to automate the process with a script running with root TTY session , you may use the following command:

```
echo "$encryption_password" | cryptsetup -y luksFormat $loop_device
```

- Open the partition, and create a mapping to $encrypted_file_path :

```
sudo cryptsetup luksOpen $loop_device tigergraph_gstore
```

If you are trying to automate the process with a script running with root TTY session , you may use the following command:

```
echo "$encryption_password" | cryptsetup luksOpen $loop_device tigergraph
```

- Clear the password from bash variables and bash history.

The following commands may clear your previous bash histories as well. Instead, you may edit ~/.bash_history to selectively delete the related entries.

```
unset encryption_password
history -c
history -w
```

- Create a file system and verify its status:

```
sudo mke2fs -j -O dir_index /dev/mapper/tigergraph_gstore
```

- Mount the new file system to /mnt/secretfs:

```
sudo mkdir -p /mnt/secretfs
sudo mount /dev/mapper/tigergraph_gstore /mnt/secretfs
```

- Change the permission to 700 so that only $db_user has access to the file system:

```
sudo chmod -R 700 /mnt/secretfs
sudo chown -R $db_user:$db_user /mnt/secretfs
```

- Move the original TigerGraph files to the encrypted filesystem and make a symbolic link. If you wish to encrypt only the TigerGraph data store (called gstore), use the following commands:

```
mv $tigergraph_data_root/gstore /mnt/secretfs/gstore
ln -s /mnt/secretfs/gstore $tigergraph_data_root/gstore
```

There are other TigerGraph files which you might also consider to be sensitive and wish to encrypt. These include the dictionary, kafka data files, and log files. You could selectively identify files to protect or you could encrypt the entire TigerGraph folder(App/Data/Log/TempRoot). In this case, simply move $tigergraph_data_root instead of $tigergraph_data_root/gstore.

```
mv $tigergraph_data_root /mnt/secretfs/tigergraph
ln -s /mnt/secretfs/tigergraph $tigergraph_data_root
```

The data of TigerGraph data is now stored in an encrypted filesystem. It will be automated decrypted when the tigergraph user (and only this user) accesses it.

To automatically deploy this encryption solution, you may

1. Chain all the steps as a bash script
2. Remove all "sudo" since the script will be running as root.
3. Run the script as root user after TigerGraph Installation.

> The setup scripts contain your encryption password. To follow good security procedures, do not leave your password in plaintext format in any files on your disk. Either remove the setup scripts or edit out the password.

Performance Evaluation

Encryption is usually CPU-bound rather than I/O-bound. If CPU usage reamains below 100%, encryption should not cause much performance slowdown. A performance test using both small and large queries supports this prediction: for small (~1 sec) and large (~100 sec) queries, there is a ~5% slowdown due to filesystem encryption.

	GSE Cold Start (read)	**Load Data (write)**
original	45s	809s
encrypted	47s	854s
% slowdown	4.4%	5.8%

We used the TPC-H dataset with scale factor 10 (http://www.tpc.org/tpch/ ↗). The data size is 23GB after loading into TigerGraph..The write test (data loading) was done by running a loading job and then killing the GPE with SIGTERM (to exit gracefully) to ensure that all kafka data is consumed.The read test (GSE cold start) measures the time from "gadmin start gse" until "online" appears in "gadmin status gse".

Example 2: Encrypting Data on Amazon EC2

Major cloud service providers often provide their own methodologies for encrypting data at rest. For Amazon EC2, we recommend users start by reading the AWS Security Blog: How to Protect Data at Rest with Amazon EC2 Instance Store Encryption ↗.

In this section, we provide a simple example for configuring file system encryption for a TigerGraph running on Amazon EC2. The steps are based on those given in How to Protect Data at Rest with Amazon EC2 Instance Store Encryption ↗, with some additions and modifications.

The basic idea of this solution is to create a file, map an encrypted file system to it, and mount it as a storage directory for TigerGraph with permission only to authorized users.

> ⚠ Angle brackets <...> are used to mark placeholders which you should replace with your own values (without the angle brackets).

Prerequisites

Make sure you have installed and configured AWS CLI ↗with keys locally.

Create an S3 Bucket

from Amazon Data-at-Rest blog

```
Sign in to the S3 console and choose Create Bucket .
In the Bucket Name box, type your bucket name and then choose Create .
You should see the details about your new bucket in the right pane.
```

Configure IAM roles and permission for the S3 bucket

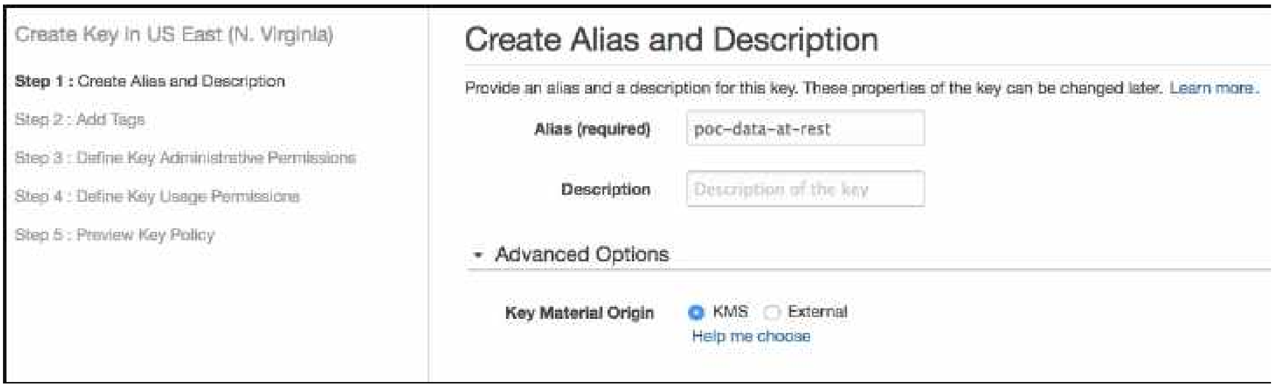
from Amazon Data-at-Rest blog

```
1.Sign in to the AWS Management Console and navigate to the IAM console .

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::<your-bucket-name>/LuksInternalStor
        }
    ]
}
The preceding policy grants read access to the bucket where the encrypted
(The following instructions have been updated since the original blog pos

2."Select type of trusted entity: Choose AWS service .
3."Select the service that will use this role": Choose EC2 then choose Ne
4.Choose the policy you created in Step 1 and then choose Next: Review.
5.On the Create role page, type your role name , a Role description, and
6.The newly created IAM role is now ready. You will use it when launching
```

Create a KMS Key (optional)

If you don't have a KMS key, you can create it first:

1. From the IAM console ↗, choose Encryption keys from the navigation pane.
2. Select Create Key , and type in <your-key-alias>
3. For Step 2 and Step 3 , see https://docs.aws.amazon.com/kms/latest/developerguide/create-keys.html ↗ for advice.
4. In Step 4 : Define Key Usage Permissions , select <your-role-name>
5. The role now has permission to use the key.



Step 2. Create Key

Encrypt a secret password with KMS and store it in the S3 bucket

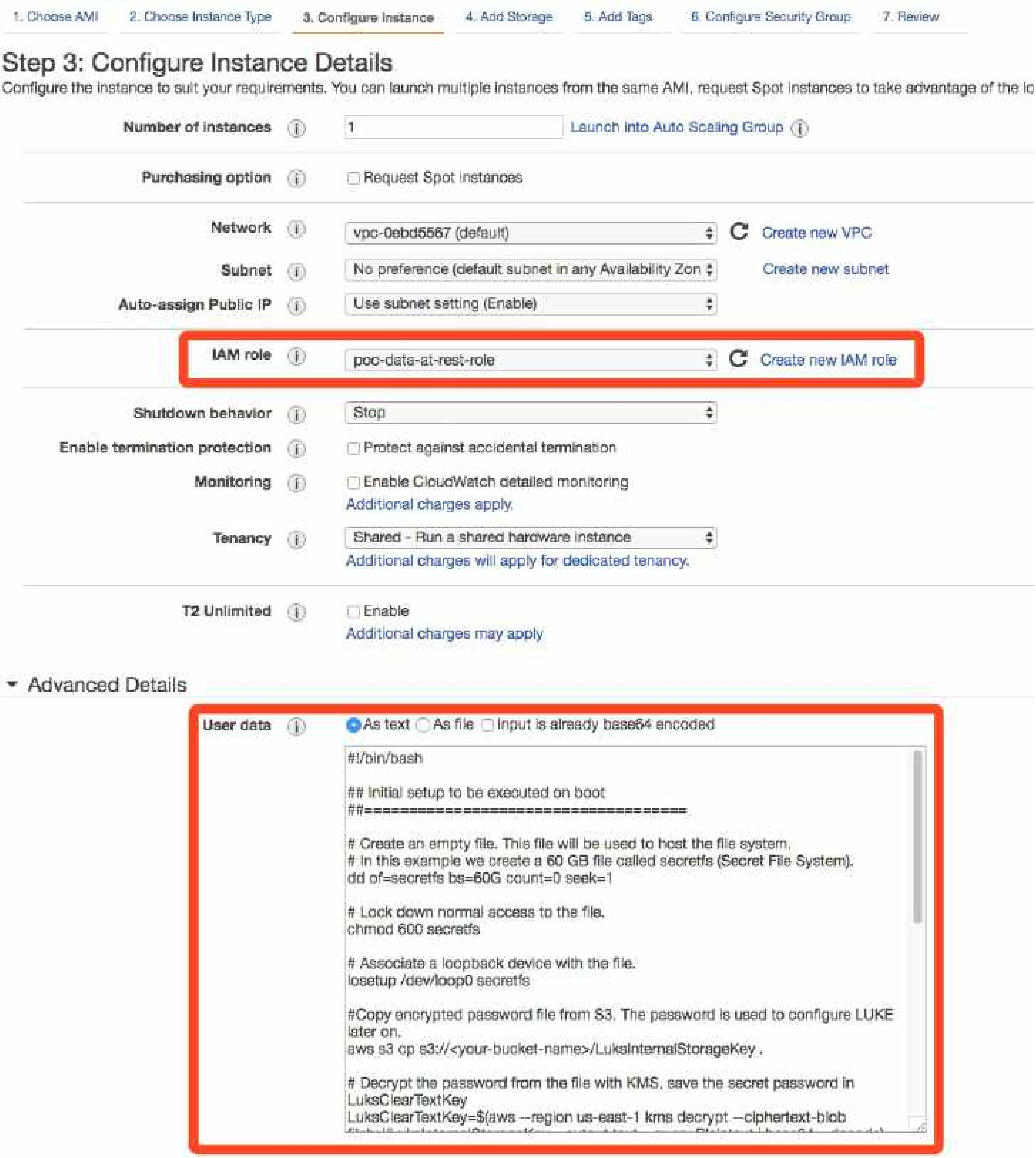
from Amazon Data-at-Rest blog

```
Next, use KMS to encrypt a secret password. To encrypt text by using KMS,

To encrypt a secret password with KMS and store it in the S3 bucket:

From the AWS CLI, type the following command to encrypt a secret password
aws --region <your-region> kms encrypt --key-id 'alias/<your-key-alias>'

aws s3 cp LuksInternalStorageKey s3://<your-bucket-name>/LuksInternalStor
The preceding commands encrypt the password (Base64 is used to decode the
```

Configure EC2 with role and launch configurations

In this section, you launch a new EC2 instance with the new IAM role and a bootstrap script that executes the steps to encrypt the file system.

⚠ The script in this section requires root permission, and it cannot be run manually through an ssh tunnel or by an unprivileged user.



1. In the [EC2 console ↗](), launch a new instance (see [this tutorial ↗]()for more details). Amazon Linux AMI 2017.09.1 (HVM), SSD Volume Type (If NOT using Amazon Linux AMI, a script the installs python, pip and AWS CLI needs to be added in the beginning).
2. In Step 3: Configure Instance Details
   a. In IAM role , choose <your-role-name>
   b. In User Data , paste the following code block after replacing the placeholders with your values and appending TigerGraph installation script

Encryption bootstrap script

```
#!/bin/bash

db_user=tigergraph

## Initial setup to be executed on boot
##====================================
# Create an empty file. This file will be used to host the file system.
# In this example we create a <disk-size> (for example: 60G) file at <pat
dd of=<path-to-encrypted-file> bs=<disk-size> count=0 seek=1

# Lock down normal access to the file.
chmod 600 <path-to-encrypted-file>

# Associate a loopback device with the file.
losetup /dev/loop0 <path-to-encrypted-file>

#Copy encrypted password file from S3. The password is used to configure
aws s3 cp s3://<your-bucket-name>/LuksInternalStorageKey .

# Decrypt the password from the file with KMS, save the secret password i
LuksClearTextKey=$(aws --region <your-region> kms decrypt --ciphertext-bl

# Encrypt storage in the device. cryptsetup will use the Linux
# device mapper to create, in this case, /dev/mapper/tigergraph_gstore.
# Initialize the volume and set an initial key.
echo "$LuksClearTextKey" | cryptsetup -y luksFormat /dev/loop0

# Open the partition, and create a mapping to /dev/mapper/tigergraph_gsto
echo "$LuksClearTextKey" | cryptsetup luksOpen /dev/loop0 tigergraph_gsto

# Clear the LuksClearTextKey variable because we don't need it anymore.
unset LuksClearTextKey

# Create a file system and verify its status.
mke2fs -j -O dir_index /dev/mapper/tigergraph_gstore

# Mount the new file system to /mnt/secretfs.
mkdir -p /mnt/secretfs
mount /dev/mapper/tigergraph_gstore /mnt/secretfs

# create user tigergraph
adduser $db_user

# Change the permission so that only tigergraph has access to the file sy
chmod -R 700 /mnt/secretfs
chown -R $db_user:$db_user /mnt/secretfs

# Install TigerGraph
```

```
# Run the one-command installation script with TigerGraphh root path unde
```

It may take a few minutes for the script to complete after system launch.

Then, you should be able to launch one or more EC2 machines with an encrypted folder under /mnt/secretfs that only OS user *tigergraph* can access.

Performance

Encryption is usually CPU-bound rather than I/O bound. If CPU usage is below 100%, TigerGraph tests show no significant performance downgrade.

System Management

Admin Portal, gamin utility, GBAR backup and restore

Managing with gadmin

Managing TigerGraph Servers with gadmin

Introduction

TigerGraph Graph Administrator (gadmin) is a tool for managing TigerGraph servers. It has a self-contained help function and a man page, whose output is shown below for reference. If you are unfamiliar with the TigerGraph servers, please see *GET STARTED with TigerGraph.*

To see a listing of all the options or commands available for gadmin, run any of the following commands:

```
$ gadmin -h
$ gadmin --help
```

After changing a configuration setting, it is generally necessary to run **`gadmin config apply`**. Some commands invoke config apply automatically. If you are not certain, just run
`gadmin config apply`

List of commands

Below is the man page for gadmin. Most of the commands are self-explanatory. Common examples are provided with each command.

NOTE: Some commands have changed in v3.0. In particular,
`gadmin set <config | license>`
has changed to
`gadmin <config | license> set`

```
GADMIN(1)                          User Commands

NAME
       gadmin - manual page for TigerGraph Administrator.

SYNOPSIS
       gadmin [flags]
       gadmin [command]

DESCRIPTION
       gadmin is a tool for managing TigerGraph servers

OPTIONS
       Available Commands:
         autocomplete Generate autocomplete script
         config       Manage the configuration for the TigerGraph system
         help         Help about any command
         init         Init the whole cluster or given service
         license      Manage TigerGraph license
         log          List log files of the given services or all service
         reset        Reset the whole init or given service with its data
         restart      Restart services by service id
         start        Start services by service id
         status       Show current status of service
         stop         Stop services by service id
         version      Show the version information

Flags:
      --debug   enable debug log output to stdout
  -h, --help    help for gadmin

Use "gadmin [command] --help" for more information about a command.
GADMIN(1)
```

gadmin autocomplete

Gadmin autocomplete is more of a feature than a command. It is an auto-complete feature that allows you to see all possible entries of a specific configuration. You can press tab when typing a command to either print out all possible entries, or auto-complete the entry you are currently typing.

```
$ gadmin autocomplete -h

Generate autocomplete script

Usage:
  gadmin autocomplete <bash|zsh> [flags]

Description:
  If you want to make this automatic, add ". <(gadmin autocomplete bash)"
  .bashrc file.

Flags:
  -h, --help   help for autocomplete

Global Flags:
      --debug   enable debug log output to stdout
```

The example below shows an example of the autocomplete for the command `gadmin status`.

```
tigergraph@ubuntu:~$ gadmin status
admin          exe            ifm            nginx
all            gpe            infra          restpp
ctrl           gse            kafka          ts3
dict           gsql           kafkaconn      ts3serv
etcd           gui            kafkastrm-ll   zk
```

gadmin config

`gadmin config` commands are used to manage the configuration for the TigerGraph system. To get a complete list of configuration parameters that are available, see Configuration Parameters.

`gadmin config` has many sub-entries as well, they will be listed below.

```
$ gadmin config -h

Manage the configuration for the TigerGraph system

Usage:
  gadmin config [flags]
  gadmin config [command]

Description:
  You can specify local config file to turn on file mode, which no remote
  connection will be made and the configs are read/write from/to the file
  all config commands work on file mode.

Available Commands:
  apply       Apply the config changes in staging state
  diff        Show the differences between staging and applied configs
  discard     Discard the staging config changes
  dump        Dump the staging system config in json format
  entry       Configure the entries with given substring patterns interac
  get         Get the config value of given entry name non-interactivly
  group       Configure the entries of given groups interactively
  init        Initialize configuration
  list        List the available config entries or groups
  set         Configure the entry of given config entry name in a non-int

Flags:
      --file string   specify config file path
  -h, --help          help for config

Global Flags:
      --debug   enable debug log output to stdout

Use "gadmin config [command] --help" for more information about a command
```

Example: Change the retention size of the kafka queue to 10GB:

```
$ gadmin config set Kafka.RetentionSizeGB 10
```

gadmin config apply

```
$ gadmin config apply -h
Apply the config changes in staging state

Usage:
  gadmin config apply [flags]

Flags:
  -y, --confirm              confirm to apply
  -f, --force                force components to apply new config
  -h, --help                 help for apply
      --initial              config apply with the initial configuration
      --with-config string   the input config file used to config apply,

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config diff

Show what configuration changes were made.

```
$ gadmin config diff -h
Show the differences between staging and applied configs

Usage:
  gadmin config diff [flags]

Flags:
  -h, --help   help for diff

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config discard

Discard the configuration changes without applying them.

```
$ gadmin config discard -h
Discard the staging config changes

Usage:
  gadmin config discard [flags]

Flags:
  -h, --help   help for discard

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config dump

Display all configuration entries.

```
$ gadmin config dump -h
Dump the staging system config in json format

Usage:
  gadmin config dump [flags]

Flags:
  -h, --help   help for dump

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config entry

Change a configuration entry.

```
$ gadmin config entry -h
Configure the entries with given substring patterns interactively

Usage:
  gadmin config entry [EntryName] [flags]

Description:
  You may use `config entry system` to go through all the system related

Flags:
  -a, --all     configure all entries
      --basic   configure basic entries only
  -h, --help    help for entry

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config get

Get the value of a specific configuration entry.

```
$ gadmin config get -h
Get the config value of given entry name non-interactivly

Usage:
  gadmin config get [EntryName] [flags]

Flags:
  -h, --help   help for get

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config group

Configure entries for a specific service group.
e.g. KAFKA, GPE, ZK

```
$ gadmin config group -h
Configure the entries of given groups interactively

Usage:
  gadmin config group [GroupName] [flags]

Description:
  You may use `gadmin config list group` to see all the groups.

Flags:
  -h, --help   help for group

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config init

Initialize your configuration.

```
$ gadmin config init -h
Initialize configuration

Usage:
  gadmin config init [flags]

Flags:
      --all            display every configurable entry
      --expert         display node assignment entries
      --ha             enable HA for init
  -h, --help           help for init
  -i, --input string   provide an input file name and init the configurat
      --template       show the template for init initialization

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config list

List all configurable entries or entry groups.

```
$ gadmin config list -h
List the available config entries or groups

Usage:
  gadmin config list <group|entry> [flags]

Description:
  List prints out the available config groups or config entries, which ca
  in entry/group commands.

Flags:
      --basic   list basic entries only
  -h, --help    help for list

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin config set

```
$ gadmin config set -h
Configure the entry of given config entry name in a non-interactive manne

Usage:
  gadmin config set [EntryName] [EntryValue] [flags]

Description:
  [EntryName] [EntryValue] must be provided in pairs, and use space to se
  each pair.

Flags:
  -h, --help   help for set

Global Flags:
      --debug         enable debug log output to stdout
      --file string   specify config file path
```

gadmin init

```
$ gadmin init -h

Init the whole cluster or given service

Usage:
  gadmin init [flags]
  gadmin init [command]

Description:
  Init command initializes the cluster/kafka. When "cluster" is specified
  a config path is required.

Available Commands:
  cluster     Init the whole cluster
  kafka       Init the KAFKA

Flags:
  -h, --help   help for init

Global Flags:
      --debug   enable debug log output to stdout

Use "gadmin init [command] --help" for more information about a command.
```

gadmin license

Options for configuring your license.

```
$ gadmin license -h

Manage TigerGraph license

Usage:
  gadmin license [flags]
  gadmin license [command]

Available Commands:
  seed        Collects host signature and generates seed file for issuing
  set         Set new license
  status      Display license status and info

Flags:
  -h, --help   help for license

Global Flags:
      --debug   enable debug log output to stdout

Use "gadmin license [command] --help" for more information about a comman
```

Generating a license seed

To generate a license seed, use the following command:

```
$ gadmin license seed <host_signature_type>
# host_signature_type: [aws|azure|gcp|hardware|node-id]
```

Depending on your host machine, you need to choose the appropriate host signature type. If you are generating the seed from a cloud instance, choose the corresponding cloud provider as your signature type.

If you are generating the seed from your own machine, choose either `node-id` or `hardware`.

- The `hardware` option tells `gadmin` to collect information from your machine's hardware as the host signature to generate the license seed. A signature produced by using this parameter will not be altered by software changes on the machine, including OS reinstalls. This is the usual choice.

- `node-id` refers to the machine ID in the `machine-id` file located at `/etc/machine-id` and is a unique signature for the OS that identifies your machine. A reinstall of the OS may change the machine ID.

Applying a new license key

Example flow for applying a new license (which may be replacing an existing license key):

```
$ gadmin license set <new_license_key>
[   Info] Configuration has been changed.
Please use 'gadmin config apply' to persist the changes.

$ gadmin config apply
[Warning] No difference from staging config, config apply is skipped.
[   Info] Successfully applied configuration change. Please restart servi

$ gadmin restart
[   Note] Restart the service(s)? (y/N)y
[   Info] Stopping DICT ADMIN GSE NGINX GPE RESTPP KAFKASTRM-LL KAFKACONN
[   Info] Starting ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKASTR
```

Once the license has been set and config has been applied, you can run `gadmin license status` to view the details of your license, including the expiration date and time.

```
$ gadmin license status
[Warning] License will expire in 6 days

   Issuer: TigerGraph Inc.
 Audience: tigergraph user
IssueTime: 2020-06-12 17:45:10 +0000 UTC
  EndTime: 2020-06-30 17:45:10 +0000 UTC
  Edition: Enterprise

Host:
    MaxCPUCore: 1024
    MaxPhysicalMemorySize: 1073741824.00MB
    MaxClusterNodeNumber: 1024

Topology:
    MaxVertexNumber: 9.007199254740991e+15
    MaxEdgeNumber: 9.007199254740991e+15
    MaxGraphNumber: 1000
    MaxTopologySize: 8.00MB

RuntimeMemory:
    MaxUserResidentSetSize: 1073741824.00MB
```

gadmin log

The `gadmin log` command will reveal the location of all commonly checked log files for the TigerGraph system.

```
$ gadmin log -h

List log files of the given services or all services

Usage:
  gadmin log [service name...] [flags]

Description:
  Service name should be a valid TigerGraph service name, for example, GS
  or GPE.

Flags:
  -h, --help   help for log

Global Flags:
      --debug   enable debug log output to stdout
```

```
$ gadmin log
ADMIN  : /home/tigergraph/tigergraph/log/admin/ADMIN#1.out
ADMIN  : /home/tigergraph/tigergraph/log/admin/ADMIN.INFO
CTRL   : /home/tigergraph/tigergraph/log/controller/CTRL#1.log
CTRL   : /home/tigergraph/tigergraph/log/controller/CTRL#1.out
DICT   : /home/tigergraph/tigergraph/log/dict/DICT#1.out
DICT   : /home/tigergraph/tigergraph/log/dict/DICT.INFO
ETCD   : /home/tigergraph/tigergraph/log/etcd/ETCD#1.out
EXE    : /home/tigergraph/tigergraph/log/executor/EXE_1.log
EXE    : /home/tigergraph/tigergraph/log/executor/EXE_1.out
GPE    : /home/tigergraph/tigergraph/log/gpe/GPE_1#1.out
GSE    : /home/tigergraph/tigergraph/log/gse/GSE_1#1.out
GSE    : /home/tigergraph/tigergraph/log/gse/log.INFO
GSQL   : /home/tigergraph/tigergraph/log/gsql/GSQL#1.out
GSQL   : /home/tigergraph/tigergraph/log/gsql/log.INFO
GUI    : /home/tigergraph/tigergraph/log/gui/GUI#1.out
IFM    : /home/tigergraph/tigergraph/log/informant/IFM#1.log
IFM    : /home/tigergraph/tigergraph/log/informant/IFM#1.out
KAFKA  : /home/tigergraph/tigergraph/log/kafka/controller.log
KAFKA  : /home/tigergraph/tigergraph/log/kafka/kafka-request.log
KAFKA  : /home/tigergraph/tigergraph/log/kafka/kafka.log
KAFKA  : /home/tigergraph/tigergraph/log/kafka/server.log
KAFKA  : /home/tigergraph/tigergraph/log/kafka/state-change.log
KAFKACONN: /home/tigergraph/tigergraph/log/kafkaconn/KAFKACONN#1.out
KAFKACONN: /home/tigergraph/tigergraph/log/kafkaconn/kafkaconn.log
KAFKASTRM-LL: /home/tigergraph/tigergraph/log/kafkastrm-ll/KAFKASTRM-LL_1
KAFKASTRM-LL: /home/tigergraph/tigergraph/log/kafkastrm-ll/kafkastrm-ll.l
NGINX  : /home/tigergraph/tigergraph/log/nginx/logs/NGINX#1.out
NGINX  : /home/tigergraph/tigergraph/log/nginx/logs/error.log
NGINX  : /home/tigergraph/tigergraph/log/nginx/logs/nginx.access.log
NGINX  : /home/tigergraph/tigergraph/log/nginx/logs/nginx.error.log
RESTPP : /home/tigergraph/tigergraph/log/restpp/RESTPP#1.out
RESTPP : /home/tigergraph/tigergraph/log/restpp/log.INFO
TS3    : /home/tigergraph/tigergraph/log/ts3/TS3_1.log
TS3    : /home/tigergraph/tigergraph/log/ts3/TS3_1.out
TS3SERV: /home/tigergraph/tigergraph/log/ts3serv/TS3SERV#1.out
ZK     : /home/tigergraph/tigergraph/log/zk/ZK#1.out
ZK     : /home/tigergraph/tigergraph/log/zk/zookeeper.log
```

gadmin reset

```
$ gadmin reset -h

Reset the whole init or given service with its data

Usage:
  gadmin reset [service name...] [flags]

Description:
  Service name should be a valid TigerGraph service name, for example, GS
  or GPE.

Flags:
  -y, --confirm   confirm to reset service
  -h, --help      help for reset

Global Flags:
      --debug   enable debug log output to stdout
```

gadmin restart

The `gadmin restart` command is used to restart one, many, or all TigerGraph services. You will need to confirm the restarting of services by either entering y (yes) or n (no). To bypass this prompt, you can use the -y flag to force confirmation.

```
$ gadmin restart -h

Restart services by service id

Usage:
  gadmin restart [serviceID...] [flags]

Description:
  ServiceID should be [serviceName][_partition][#replica], e.g., GSE_1#3.
  replica field empty(e.g. GSE_1) to either refer to all replicas of give
  partition, or if the service has no replicas(e.g. EXE_1). Same for pari

Flags:
  -y, --confirm   confirm to restart service
  -h, --help      help for restart
      --no-dep    restart service without dependency

Global Flags:
      --debug   enable debug log output to stdout
```

```
$ gadmin restart all -y
[   Info] Stopping ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKASTR
[   Info] Stopping CTRL
[   Info] Stopping EXE
[   Info] Starting EXE
[   Info] Starting CTRL
[   Info] Starting ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKASTR
```

gadmin start

The `gadmin start` command can be used to start one, many, or all services.

```
$ gadmin start -h

Start services by service id

Usage:
  gadmin start [serviceID...] [flags]

Description:
  ServiceID should be [serviceName][_partition][#replica], e.g., GSE_1#3.
  replica field empty(e.g. GSE_1) to either refer to all replicas of give
  partition, or if the service has no replicas(e.g. EXE_1). Same for pari
  If no serviceID is specified, it only starts services excluding the
  infrastructure. Use 'gadmin start all' to start all services.

Flags:
      --dry-run              dry run and output command to start the serv
  -h, --help                 help for start
      --no-dep               start service without dependency
      --with-config string   start with given config file and dump it to
      --ignore-error         starting services with ignore-error will sta

Global Flags:
      --debug   enable debug log output to stdout
```

```
$ gadmin start all
[   Info] Starting EXE
[   Info] Starting CTRL
[   Info] Starting ZK ETCD DICT KAFKA ADMIN GSE NGINX GPE RESTPP KAFKASTR
```

gadmin status

Check the status of TigerGraph component servers:

```
$ gadmin status -h

Show current status of service

Usage:
  gadmin status [serviceID...] [flags]

Description:
  ServiceID should be [serviceName][_partition][#replica], e.g., GSE_1#3.
  replica field empty(e.g. GSE_1) to either refer to all replicas of give
  partition, or if the service has no replicas(e.g. EXE_1). Same for pari
  If no serviceID is specified, it will show all service status

Flags:
  -h, --help      help for status
  -v, --verbose   report service status in detail

Global Flags:
      --debug   enable debug log output to stdout
```

Use **`gadmin status`** to report whether each of the main component servers is running (up) or stopped (off). The example below shows the normal status when the graph store is empty and a graph schema has not been defined:

```
$ gadmin status

+--------------------+-------------------------+-------------------------
|    Service Name    |      Service Status     |       Process State
+--------------------+-------------------------+-------------------------
|       ADMIN        |         Online          |         Running
|        CTRL        |         Online          |         Running
|        DICT        |         Online          |         Running
|        ETCD        |         Online          |         Running
|        GPE         |         Online          |         Running
|        GSE         |         Online          |         Running
|        GSQL        |         Online          |         Running
|        GUI         |         Online          |         Running
|        IFM         |         Online          |         Running
|       KAFKA        |         Online          |         Running
|     KAFKACONN      |         Online          |         Running
|    KAFKASTRM-LL    |         Online          |         Running
|       NGINX        |         Online          |         Running
|       RESTPP       |         Online          |         Running
|        TS3         |         Online          |         Running
|      TS3SERV       |         Online          |         Running
|         ZK         |         Online          |         Running
+--------------------+-------------------------+-------------------------
```

You can also check the status of each instance using the verbose flag : `gadmin status -v` or `gadmin status --verbose`. This will show each machine's status. See example below

```
$ gadmin status -v GPE

+--------------------+-------------------------+-------------------------
|    Service Name    |      Service Status     |       Process State
+--------------------+-------------------------+-------------------------
|      GPE_1#1       |         Warmup          |         Running
|      GPE_1#2       |         Warmup          |         Running
+--------------------+-------------------------+-------------------------
```

Here are the most common service and process status states you might see from running the `gadmin status` command :

Service Status Definitions

- **Online** - The service is online and ready.
- **Warmup** - The service is processing the graph information and will be online soon.
- **Stopping** - The service has received a stop command and will be down soon.
- **Offline** - The service is not available.
- **Down** - The service has been stopped or crashed.
- **StatusUnknown** - The valid status of the service is not tracked.

Process State Status Definitions

- **Init** - Process is initializing and will be in the running state soon.
- **Running** - The process is running and available.
- **Zombie** - There is a leftover process from a previous instance.
- **Stopped** - The process has been stopped or crashed.
- **StatusUnknown** - The valid status of the process is not tracked.

gadmin stop

The gadmin stop command can be used to stop one, many, or all TigerGraph services. You will need to confirm the restarting of services by either entering y (yes) or n (no). To bypass this prompt, you can use the -y flag to force confirmation.

```
$ gadmin stop -h

Stop services by service id

Usage:
  gadmin stop [serviceID...] [flags]

Description:
  ServiceID should be [serviceName][_partition][#replica], e.g., GSE_1#3.
  replica field empty(e.g. GSE_1) to either refer to all replicas of give
  partition, or if the service has no replicas(e.g. EXE_1). Same for pari
  If no serviceID is specified, it only stops services excluding the
  infrastructure. Use 'gadmin stop all' to stop all services.

Flags:
  -y, --confirm   confirm to stop service
  -h, --help      help for stop
  --ignore-error  stopping services with ignore-error will stop all possi

Global Flags:
      --debug   enable debug log output to stdout
```

```
$ gadmin stop gsql
[   Note] Stop the service(s)? (y/N)y
[   Info] Stopping GSQL
```

gadmin version

```
$ gadmin version -h

Show the version information

Usage:
  gadmin version [flags]

Description:
  Show version information of all TigerGraph components, including repo n
  version, git commit number, git commit datetime.

Flags:
  -h, --help   help for version

Global Flags:
      --debug   enable debug log output to stdout
```

More examples

Configuring memory use thresholds

TigerGraph offers two levels of memory thresholds using the following configuration settings:

SysAlertFreePct and **SysMinFreePct**

SysAlertFreePct setting indicates that the memory usage has crossed a threshold where the system will start throttling Queries to allow long-running queries to finish and release the memory.

SysMinFreePct setting indicates that the memory usage has crossed a critical threshold and the Queries will start aborting automatically to prevent GPE crash and system stability.

By default, SysMinFreePct is set at 10%, at which point Queries will be aborted.

```
$ gadmin config entry GPE.BasicConfig.Env

GPE.BasicConfig.Env [ LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY
✔ New: LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY_PATH;

Add this line to the existing config :
SysMinFreePct=20;SysAlertFreePct=70;

Your config line should now look like this :

GPE.BasicConfig.Env [ LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY
✔ New: LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY_PATH;SysMinFre

$ gadmin restart gpe -y
```

Example:

SysAlertFreePct=30 means when the system memory consumption is over **70%** of the memory, the system will enter an alert state, and Graph updates will start to slow down.

SysMinFreePct=20 means 20% of the memory is required to be free. When memory consumption enters a critical state **(over 80% memory consumption)** queries will be aborted. automatically.

Configuring Nginx configuration template

Follow the steps documented in this support article ↗ to update the Nginx configurations of your TigerGraph instance.

File Output Policy

GSQL restricts where a query can produce output to files through a file output policy. The policy consists of a *whitelist* and a *blacklist*.

- GSQL queries must only output to the directories and their descendants or the files indicated by paths in the whitelist.
- GSQL queries *cannot* output to the directories and their descendants or the files indicated by paths in the blacklist. The blacklist takes precedence over the whitelist.

By default, the file output policy allows outputs to all files.

`GSQL.FileOutputPolicy`

The file output policy is implemented through the system configuration parameter `GSQL.FileOutputPolicy`, which is a JSON array of strings that represents a list of paths. If there is an exclamation mark (`!`) preceding a path, the path is on the blacklist. If there is no exclamation mark preceding a path, the path is on the whitelist.

Example

For example, if the value for `GSQL.FileOutputPolicy` is `["/home/tigergraph", "!/home/tigergraph/documents", "!/home/tigergraph/desktop"]`, then below are the paths on the white list and on the black list:

- Whitelist: `/home/tigergraph` and all its descendants
- Blacklist: `/home/tigergraph/documents, /home/tigergraph/desktop` and all their descendants.

Since the blacklist takes precedence, GSQL will allow queries to write to all files and directories under `/home/tigergraph` except the `documents` and `destktop` folders.

Edit the file output policy

1. To edit the file policy, ensure that you are logged in as the TigerGraph Linux user, and run the following command:

```
$ gadmin config entry GSQL.FileOutputPolicy
```

2. In the prompt, enter the new value for the parameter:

```
GSQL.FileOutputPolicy [ ["/"] ]: The policy to control file outputs ir
New: ["/home/tigergraph", "!/home/tigergraph/app"]
# Whitelist: /home/tigergraph and all its descendants
# Blacklist: /home/tigergraph/app and all its decendants
# Effect: GSQL can output to /home/tigergraph and all its decendants e
```

3. Apply the new configurations and restart GSQL

```
$ gadmin config apply
$ gadmin restart gsql
```

After implementing the file output policy, queries that write to paths that are not on the whitelist are forbidden:

```
GSQL > BEGIN
GSQL > CREATE QUERY fileOutput() FOR GRAPH tpc_graph {
GSQL >   FILE f ("/home/documents/data.txt");
GSQL > }
GSQL > END

Semantic Check Error in query fileOutput (SEM-2502): line 2, col 7
The path '/home/documents/data.txt' is not allowed by the file output pol
For more info, please check log at node 'm2': /home/tigergraph/tigergraph
Failed to create queries: [fileOutput].
```

If a `FILE` object is defined with an empty string, it is regarded as a null file. The file output policy will not block the definition of the `FILE` object, but writing to a null file would cause a runtime error.

Additionally, queries that write to paths on the whitelist, but also on the blacklist are also forbidden:

```
GSQL > BEGIN
GSQL > CREATE QUERY fileOutput() FOR GRAPH tpc_graph {
GSQL >   FILE f ("/home/tigergraph/app/data.txt");
GSQL > }
GSQL > END

Semantic Check Error in query fileOutput (SEM-2502): line 3, col 7
The path '/home/tigergraph/app/data.txt' is not allowed by the file outpu
policy.
For more info, please check log at node 'm2': /home/tigergraph/tigergraph
Failed to create queries: [fileOutput].
```

Backup and Restore

GBAR - Graph Backup and Restore

Graph Backup And Restore (GBAR), is an integrated tool for backing up and restoring the data and data dictionary (schema, loading jobs, and queries) of a TigerGraph instance or cluster.

The backup feature packs TigerGraph data and configuration information into a directory on the local disk or a remote AWS S3 bucket. Multiple backup files can be archived. Later, you can use the restore feature to roll back the system to any backup point. This tool can also be integrated easily with Linux cron to perform periodic backup jobs.

Syntax

The current version of GBAR is intended for restoring the same machine that was backed up. For help with cloning a database (i.e., backing up machine A and restoring the database to machine B), please contact support@tigergraph.com.

Synopsis

```
Usage: gbar backup [options] -t <backup_tag>
              gbar restore [options] <backup_tag>
              gbar list [backup_tag] [-j]
              gbar remove|rm <backup_tag>
              gbar cleanup
              gbar expand [-a] <new_nodes>
                   New nodes must be written in <name>:<host> pairs separ
                   Example:
                       m1:192.168.1.2,m2:192.168.1.3,m3:192.168.1.4

        Options:
         -h, --help         Show this help message and exit
         -v                 Run with debug info dumped
         -vv                Run with verbose debug info dumped
         -y                 Run without prompt
         -j                 Print gbar list as JSON
         -t BACKUP_TAG      Tag for backup file, required on backup
         -a, --advanced    Enable advanced mode for node expansion
```

The `-y` option forces GBAR to skip interactive prompt questions by selecting the default answer. There is currently one interactive question:

- At the start of restore, GBAR will always ask if it is okay to stop and reset the TigerGraph services: (y/N)? The default answer is yes.

Configure GBAR

Before using the backup or the restore feature, GBAR must be configured.

1. Run `gadmin config entry system.backup`. At each prompt, enter the appropriate values for each config parameter.

```
$ gadmin config entry system.backup

System.Backup.TimeoutSec [ 18000 ]: The backup timeout in seconds
New: 18000

System.Backup.CompressProcessNumber [ 8 ]: The number of concurrent pr
New: 8

System.Backup.Local.Enable [ true ]: Backup data to local path
New: true

System.Backup.Local.Path [ /tmp/backup ]: The path to store the backup
New: /data/backup

System.Backup.S3.Enable [ false ]: Backup data to S3 path
New: false

System.Backup.S3.AWSAccessKeyID [ <masked> ]: The path to store the ba
New:

System.Backup.S3.AWSSecretAccessKey [ <masked> ]: The path to store th
New:

System.Backup.S3.BucketName [  ]: The path to store the backup files
New:
```

2. After entering the configuration values, run the following command to apply the new configurations

```
gadmin config apply -y
```

Note:

- You can specify the number of parallel processes for backup and restore.
- You must provide username and password using GSQL_USERNAME and GSQL_PASSWORD environment variables.

  ```
  $ GSQL_USERNAME=tigergraph GSQL_PASSWORD=tigergraph gbar backup -t dai
  ```

Perform a backup

To perform a backup, run the following command as the TigerGraph Linux user:

```
gbar backup -t <backup_tag>
```

Depending on your configuration settings, your backup archive will be output to your local backup path and/or your AWS S3 bucket. If you are running a cluster, there will be a backup archive on every node in the same path.

A backup archive is stored as several files in a folder, rather than as a single file. The backup tag acts like a filename prefix for the archive filename. The full name of the backup archive will be `<backup_tag>-<timestamp>`, which is a subfolder of the backup repository.

- If `System.Backup.Local.Enable` is set to `true`, the folder is a local folder on every node in a cluster, to avoid massive data moving across nodes in a cluster.
- If `System.Backup.S3.Enable` is set to `true`, every node will upload data located on the node to the s3 repository. Therefore, every node in a cluster needs access to Amazon S3.

GBAR Backup performs a live backup, meaning that normal operations may continue while the backup is in progress. When GBAR backup starts, GBAR will check if there are running loading jobs. If there are, it will pause loading for 1 minute to generate a snapshot and then continue the backup process. You can specify the loading pausing interval by the environment variable `PAUSE_LOADING`.

GBAR then sends a request to the admin server, which then requests the GPE and GSE to create snapshots of their data. Per the request, the GPE and GSE store their data under GBAR's own working directory. GBAR also directly contacts the Dictionary and obtains a dump of its system configuration information. In addition, GBAR gathers the TigerGraph system version and customized information including user-defined functions, token functions, schema layouts and user-uploaded icons. Then, GBAR compresses each of these data and configuration information files in tgz format and stores them in the `<backup_tag>-<timestamp>` subfolder on each node. As the last step, GBAR copies that file to local storage or AWS S3, according to the Config settings, and removes all temporary files generated during backup.

The current version of GBAR Backup takes snapshots quickly to make it very likely that all the components (GPE, GSE, and Dictionary) are in a consistent state, but it does not fully guarantee consistency.

⚠ Backup does not save input message queues for REST++ or Kafka.

List Backup Files

```
gbar list
```

This command lists all generated backup files in the storage place configured by the user. For each file, it shows the file's full tag, its size in human-readable format, and its creation time.

Restore from a backup archive

Before restoring a backup, you should ensure that the backup you are restoring from is in the **same exact version** as your current version of TigerGraph.

To restore a backup, run the following command:

```
gbar restore <archive_name>
```

If GBAR can verify that the backup archive exists and that the backup's system version is compatible with the current system version, GBAR will shut down the TigerGraph servers temporarily as it restores the backup. After completing the restore, GBAR will restart the TigerGraph servers. If you are running a cluster, and you have copied the backup files to each individual node in the cluster, running `gbar restore` on any node will restore the entire cluster.

Restore is an offline operation, requiring the data services to be temporarily shut down. The user must specify the full archive name (`<backup_tag>-<timestamp>`) to be restored. When GBAR restore begins, it first searches for a backup archive exactly matching the archive name supplied in the command line. Then it decompresses the backup files to a working directory. Next, GBAR will compare the TigerGraph system version in the backup archive with the current system's version, to make sure that the backup archive is compatible with that current system. It will then shut down the TigerGraph servers (GSE, RESTPP, etc.) temporarily. Then, GBAR makes a copy of the current graph data, as a precaution. Next, GBAR copies the backup graph data into the GPE and GSE and notifies the Dictionary to load the configuration data. Also, GBAR will notify the GST to load backup user data and copy the backup user-defined token/functions to the right location. When these actions are all done, GBAR will restart the TigerGraph servers.

Note: GBAR restore does not estimate the uncompressed data size and check whether there is sufficient disk space.

> The primary purpose of GBAR is to save snapshots of the data configuration of a TigerGraph system, so that in the future the same system can be rolled back (restored) to one of the saved states. A key assumption is that Backup and Restore are performed on the same machine, and that the file structure of the TigerGraph software has not changed.

> Restore needs enough free space to accommodate both the old gstore and the gstore to be restored.

Remove a backup

To remove a backup, run the `gbar remove` command:

```
$ gbar remove <backup_tag>
```

The command removes a backup from the backup storage path. To retrieve the tag of a backup, you can use the `gbar list` command.

Clean up temporary files

Run `gbar cleanup` to delete the temporary files created during backup or restore operations:

```
$ gbar cleanup
```

GBAR Detailed Example

The following example describes a real example, to show the actual commands, the expected output, and the amount of time and disk space used, for a given set of graph data. For this example, an Amazon EC2 instance was used, with the following specifications:

Single instance with 32 vCPU + 244GB memory + 2TB HDD.

Naturally, backup and restore time will vary depending on the hardware used.

GBAR Backup Operational Details

To run a daily backup, we tell GBAR to backup with the tag name *daily*.

```
$ gbar backup -t daily
[23:21:46] Retrieve TigerGraph system configuration
[23:21:51] Start workgroup
[23:21:59] Snapshot GPE/GSE data
[23:33:50] Snapshot DICT data
[23:33:50] Calc checksum
[23:37:19] Compress backup data
[23:46:43] Pack backup data
[23:53:18] Put archive daily-20180607232159 to repo-local
[23:53:19] Terminate workgroup
Backup to daily-20180607232159 finished in 31m33s.
```

The total backup process took about 31 minutes, and the generated archive is about 49 GB. Dumping the GPE + GSE data to disk took 12 minutes. Compressing the files took another 20 minutes.

GBAR Restore Operational Details

To restore from a backup archive, a full archive name needs to be provided, such as *daily-20180607232159*. By default, restore will ask the user to approve to continue. If you want to pre-approve these actions, use the "-y" option. GBAR will make the default choice for you.

```
$ gbar restore daily-20180607232159
[23:57:06] Retrieve TigerGraph system configuration
GBAR restore needs to reset TigerGraph system.
Do you want to continue?(y/N):y
[23:57:13] Start workgroup
[23:57:22] Pull archive daily-20180607232159, round #1
[23:57:57] Pull archive daily-20180607232159, round #2
[00:01:00] Pull archive daily-20180607232159, round #3
[00:01:00] Unpack cluster data
[00:06:39] Decompress backup data
[00:17:32] Verify checksum
[00:18:30] gadmin stop gpe gse
[00:18:36] Snapshot DICT data
[00:18:36] Restore cluster data
[00:18:36] Restore DICT data
[00:18:36] gadmin reset
[00:19:16] gadmin start
[00:19:41] reinstall GSQL queries
[00:19:42] recompiling loading jobs
[00:20:01] Terminate workgroup
Restore from daily-20180607232159 finished in 22m55s.
Old gstore data saved under /home/tigergraph/tigergraph/gstore with suffi
```

For our test, GBAR restore took about 23 minutes. Most of the time (20 minutes) was spent decompressing the backup archive.

Note that after the restore is done, GBAR informs you were the pre-restore graph data (gstore) has been saved. After you have verified that the restore was successful, you may want to delete the old gstore files to free up disk space.

Performance Summary of Example

GStore size	Backup file size	Backup time	Restore time
219GB	49GB	31 mins	23 mins

Database Import/Export

Export/Import is a complement to Backup/Restore, not a substitute.

The GSQL `EXPORT` and `IMPORT` commands perform a logical backup and restore. A database export contains the database's data, and optionally some types of metadata, which can be subsequently imported in order to recreate the same database, in the original or in a different TigerGraph platform instance.

To import an exported database, ensure that the export files are from a database that was running **the exact same version** of TigerGraph as the database that you are importing into.

Known Issues (Updated Feb 16th):

- User-defined loading jobs containing `DELETE` statements are not exported correctly.
- If a graph contains vertex or edge types with a composite key, the graph data is exported in a nonstandard format that cannot be reimported.

EXPORT GRAPH ALL

The `EXPORT GRAPH ALL` command reads the data and metadata for all graphs in the TigerGraph system and writes the information to a zip file in the designated folder. If no options are specified, then a full backup is performed, including schema, data, template information, and user profiles.

Required privilege

`EXPORT_GRAPH`

Synopsis

```
EXPORT GRAPH ALL [<export_options>] TO "<directory_name>"

exportOptions ::=
(-S | --SCHEMA | -T | --TEMPLATE | -D | --DATA | -U | --USERS | -P | --PA

    -S, --SCHEMA        Only Schema will be exported
    -T, --TEMPLATE      Only Schema, Queries, Loading Jobs, UDFs
    -D, --DATA          Only Data Sources will be exported
    -U, --USERS         Includes Permissions, Secrets, and Tokens
    -P, --PASSWORD      Encrypt with password. User will be prompted
```

The export directory should be empty before running the command because all contents are zipped and compressed.

Parameters

Parameter	Description
`directory_name`	The path of the directory to output export files to. Must be an absolute path.

Options

Option	Description
`-S` or `--SCHEMA`	Only export graph schema.
`-T` or `--TEMPLATE`	Only export graph schema, queries, loading jobs and UDFs.
`-D` or `--DATA`	Must be used with either `-S` or `-T`. Export data in CSV in addition to graph schema or template.
`-U` or `--Users`	Must be used with either `-S` or `-T`. Export users, role assginments, secrets, and tokens.
`-P` or `--Password`	Encrypt the exported file with a password. Users will be prompted to enter a password

when using this option.

Output

The `EXPORT GRAPH` command exports all graphs in the database.

The export contains four categories of files:

1. Data files in CSV format, one file for each type of vertex and each type of edge.
2. GSQL DDL command files created by the export command. The import command uses these files to recreate the graph schema(s) and reload the data.
3. Copies of the database's queries, loading jobs, and user-defined functions.
4. GSQL command files used to recreate the users and their privileges.

The following files are created in the specified directory when exporting and are then zipped into a single file called ExportedGraphs.zip.

> If the file is password-protected, it can only be unzipped using GSQL IMPORT. The security feature prevents users from directly unzipping it.

- A **DBImportExport_<graphName>.gsql** for each graph called <graphName> in a multigraph database that contains a series of GSQL DDL statements that do the following:
  - Create the exported graph, along with its local vertex, edge, and tuple types,
  - Create the loading jobs from the exported graphs
  - Create data source file objects
  - Create queries
- A **graph_<graphName>/** folder for each graph in a multigraph database containing data for local vertex/edge types in <graphName>. For each vertex or edge type called <type>, there is one of the following two data files:
  - vertex_<type>.csv
  - edge_<type>.csv

- **global.gsql** - DDL job to create all global vertex and edge types, and data sources.
- **tuple.gsql** - DDL job to create all User Defined Tuples.
- Exported data and jobs used to restore the data:
  - **GlobalTypes/** - folder containing data for global vertex/edge types
    - vertex_name.csv
    - edge_name.csv
  - **run_loading_jobs.gsql** - DDL created by the export command which will be used during import:
    - Temporary global schema change job to add user-defined indexes. This schema job is dropped after it is has run.
    - Loading jobs to load data for global and local vertex/edges.
- Database's saved queries, loading jobs, and schema change jobs
  - **SchemaChangeJob/ -** folder containing DDL for schema change jobs. See section "Schema Change Jobs" for more information
    - Global_Schema_Change_Jobs.gsql contains all global schema change jobs
    - graphName_Schema_Change_Jobs.gsql contains schema change jobs for each graph "graphName"

    Tokenbank.cpp - copy of `<tigergraph.root.dir>/app/<VERSION_NUM>/dev/gdk/gsql/src/TokenBank/TokenBank.cpp`
  - **ExprFunctions.hpp** - copy of `<tigergraph.root.dir>/app/<VERSION_NUM>dev/gdk/gsql/src/QueryUdf/ExprFunctions.hpp`
  - **ExprUtil.hpp** - copy of `<tigergraph.root.dir>/app/<VERSION_NUM>/dev/gdk/gsql/src/QueryUdf/ExprUtil.hpp`
- Users:
  - **users.gsql** - DDL to create all exported users and import Secrets and Tokens, and grant permissions.

Example

```
GSQL > EXPORT GRAPH ALL TO "/tmp/export_graphs/"
```

Insufficient Disk Space

If not enough disk space is available for the data to be exported, the system returns an error message indicating not all data has been exported. Some data may have already been written to disk. If an insufficient disk error occurs, the files will not be zipped, due to the possibility of corrupted data which would then corrupt the zip file. The user should clear enough disk space, including deleting the partially exported data, before reattempting the export.

It is possible for all the files to be written to disk and then to run out of disk space during the zip operation. If that is the case, the system will report this error. The unzipped files will be present in the specified export directory.

Default Timeout and Session Parameter export_timeout

If the timeout limit is reached during export, the system returns an error message indicating not all data has been exported. Some data may have already been written to disk. If a timeout error occurs, the files will not be zipped. The user should delete the export files, increase the timeout limit and then rerun the export.

The timeout limit is controlled by the session parameter **export_timeout**. The default timeout is ~138 hours. To change the timeout limit, use the command:

```
GSQL > SET EXPORT_TIMEOUT = <timeout_in_ms>
```

IMPORT GRAPH ALL

The `IMPORT GRAPH ALL` command unzips the `.zip` file `ExportedGraph.zip` located in the designated folder, unzips it, and then runs the GSQL command files within to import the graph.

Required privilege

`WRITE_SCHEMA`, `WRITE_QUERY`, `WRITE_LOADINGJOB`, `EXECUTE_LOADINGJOB`, `DROP ALL`, `WRITE_USERS`

Synopsis

```
IMPORT GRAPH ALL [import_options] FROM "<filename>"

importOptions ::= [-P | --PASSWORD ] [ (-KU | -- keep-users]
    -P,  --PASSWORD     Decrypt with password. User will be prompted.
    -KU, --KEEP-USERS   Do not delete user identities before importing
```

Parameters

Parameter	Description
`filename`	The path to the zip file produced by the `EXPORT GRAPH ALL` command. Must be an absolute path.

Options

Option	Description
`-P` or `--PASSWORD`	Decrypt with password. You will be prompted to enter a password when using this option.
`-KU` or `--KEEP-USERS`	Keep the current users for during the import operation. New users from the imported graph will still be added. If there are users from the same name, GSQL will report an error when importing the users, and the current users will be kept.

Example

```
GSQL > IMPORT GRAPH ALL FROM "/tmp/export_graphs/"
```

`IMPORT GRAPH` looks for specific filenames. If either the zip file or any of its contents are renamed by the user, IMPORT GRAPH may fail.

`IMPORT GRAPH` erases the current database (equivalent to running DROP ALL). The current version does not support incremental or supplemental changes to an existing database (except for the --keep-users option)

Loading Jobs

There are two sets of loading jobs:

1. Those that were in the **catalog** of the database which was exported. These are embedded in the file DBImportExport_graphName.gsql
2. Those that are **created by EXPORT GRAPH** and are used to assist with the import process. These are embedded in the file run_loading_jobs,gsql.

The catalog loading jobs are not needed to restore the data. They are included for archival purposes.

Some special rules apply to importing loading jobs. Some catalog loading jobs will not be imported.

1. **If a catalog loading job contains** `DEFINE FILENAME F = "/path/to/file/"`, the path will be removed and the imported loading job will only contain `DEFINE FILENAME F`.

   This is to allow a loading job to still be imported even though the file may no longer exist or the path may be different due to moving to another TigerGraph instance.

2. **If a specific file path is used directly in the LOAD statement, and the file cannot be found, the loading job cannot be created and will be skipped.**

   For example, `LOAD "/path/to/file" to vertex v1` cannot be created if `/path/to/file` does not exist.
3. **Any file path using `$sys.data_root` will be skipped.**
   This is because the value of `$sys.data_root` is not retained from export. During import, `$sys.data_root` is set to the root folder of the import location.

Schema Change Jobs

There are two sets of schema change jobs:

1. Those that were in the catalog of the database which was exported. These are stored in the folder /SchemaChangeJobs.
2. Those that were created by EXPORT GRAPH and are used to assist with the import process. These are in the run_loading_jobs.gsql command file. The jobs are dropped after the import command is finished with them.

The database's schema change jobs are not executed during the import process. This is because if a schema change job had been run before the export, then the exported schema already reflects the result of the schema change job. The directory /SchemaChangeJobs contains these files:

- **Global_Schema_Change_Jobs.gsql** contains all global schema change jobs
- **<graphName>_Schema_Change_Jobs.gsql** contains schema change jobs for each graph <graphName>.

Cluster import/export

In the current version, importing and exporting clusters is not fully automated. The database can be exported and imported by following some additional steps.

Export from a Cluster

Rather than creating a single export zip file, export will create a file for each machine. Before exporting, specific folders must be created on each server using the following commands:

Run on each server before EXPORT

```
$ grun all "mkdir -p /path/to/export_directory/GlobalTypes/"
$ grun all "mkdir -p /path/to/export_directory/graph_<graphName>/"
```

Then run the export command on one server. The EXPORT command does not bundle all the files to one server, and it does not compress each server's files to one zip. Some files, including the data files, will be exported to each server, to the folders created above. Some files will be only on the local server where EXPORT GRAPH was run.

Import to a Cluster

1. Place the files on the import servers

You may only import to a cluster that has the same number and configuration of servers as the data from which the export originated. **Transfer the files from one export server to a corresponding import server.** That is, copy the files from `export_server_n:/path/to/export_directory` to `import_server_n:/path/to/import/directory`

2. Manually modify the loading jobs

On the main server, edit the run_loading_jobs.gsql files as follows.

Find the line(s) of the form:
`LOAD "sys.data_root/.../<vertex_or_edge_type>.csv"`
Close to it should be similar line that is commented out which have the "all:" data source directive:
`#LOAD "all:sys.data_root/.../<vertex_or_edge_type>.csv"`

See the example below:

```
LOAD "$sys.data_root/graph_graph1/localE.csv"
#If running on a cluster, check that the file exists on all nodes then un
#LOAD "all:$sys.data_root/graph_graph1/localE.csv"
    TO EDGE localE VALUES ($"from", $"to") USING SEPARATOR = "^]", HEADER
```

Comment out the LOAD line and uncomment the LOAD all: line. Be sure that you do this for all data source files.

3. Run the IMPORT GRAPH command from the main server (e.g., the one that corresponds to the server where EXPORT GRAPH was run).

Advanced Platform-layer Commands

This page documents a list of advanced Linux commands that simplify platform operations that are performed often during debugging, especially on high availability (HA) clusters. Only the TigerGraph platform owner - the Linux user created during installation has access to the commands on this page.

Users are advised to use these commands only at the guidance and recommendation of TigerGraph support.

Connection between nodes

Connect to another node via SSH

```
$ gssh <node_name>
```

This command allows you to connect to another node in your cluster via SSH.

Example:

```
# Originally on m1
[tigergraph@ip-172-31-88-111 ~]$ gssh m3
Last login: Fri Apr 23 18:24:27 2021
# Now connected to m3 via ssh
[[tigergraph@ip-172-31-93-187 ~]$
```

Loading data

Offline loading

```
$ gautoloading.sh (-g <graph_name> -j <loading_job_name> | path_to_config
```

With huge data volumes, data loading can be time-consuming. If you find yourself often loading huge volumes of data into an empty graph, and your data volume is so large that your loading jobs are taking hours to complete, you might consider using offline loading to speed up data loading.

In order to use offline loading, all the filename variables in the loading job must take an initial path value. After creating the loading job and ensuring that all the data files are referenced correctly in the loading job, use the options `-g` and `-j` to specify the graph and loading job to run. During offline loading, your database is focused on loading data and will not be able to handle requests and queries.

⚠ Offline loading deletes all existing graph data before it starts. Back up your data before using offline loading.

Options

- `-g <graph_name>`: Name of the graph whose loading job to run
- `-j <loading_job_name>`: Name of the loading job to run

Example

The following command runs the loading job `load_ldbc_snb` on the graph `ldbc_snb`:

```
$ gautoloading.sh -g ldbc_snb -j load_ldbc_snb
```

You can also provide the graph name and the loading job name with a config file written in Bash:

~/example_config

```
# the name of the graph for the initial loading
GRAPH_NAME="tpc_graph"

# the name of loading jobs separated by white space
LOADING_JOBS=("load_test")
```

Once you have the config file, you can run `gautoloading.sh` with the config file instead of the `-g` and `-j` options:

```
$ gautoloading.sh ~/example_config
```

File operations

Copy files on the specified nodes

```
$ gscp <all|component_name|node_list> <source_path> <target_dir>
```

This command allows you to copy files from the current node to target folders on multiple nodes at the same time. The file or directory on the current node specified by the source path will be copied into the target folder on every node. If the target folder does not exist at the path given, the target folder will be created automatically. You can also specify multiple source files or directories, in which case, the source paths need to be absolute paths, put in quotes, and separated by space.

You can specify the nodes where you want the copy operation to occur in the following ways:

- `gscp all <source_path> <target_dir>` will execute the command on all nodes
- `gscp <component_name> <source_path> <target_dir>` will execute the command on nodes where the component you specified is running
- `gscp <node_list> <source_path> <target_dir>` will execute the command on the nodes you specify in the node list

Example

Single source

```
$ gscp all /tmp/gscp_test /tmp/gscp_test_folder

### Connecting to local  server 172.31.91.54 ...

### Connecting to remote server 172.31.88.179 ...

### Connecting to remote server 172.31.91.208 ...

// A copy of gscp_test is on every node
$ grun all 'ls /tmp/gscp_text_folder'

### Connecting to local  server 172.31.91.54 ...
gscp_test

### Connecting to remote server 172.31.88.179 ...
gscp_test

### Connecting to remote server 172.31.91.208 ...
gscp_test

// Copy file to the target folder only on nodes where GPE is running
$ gscp gpe /tmp/gscp_test1 /tmp/gscp_test_folder

// Copy file to a specified list of nodes
$ gscp m1,m2 /tmp/gscp_test3 /tmp/gscp_test_folder
```

Multiple sources

```
$ gscp all "/tmp/gscp_test1 /tmp/gscp_test2" /tmp/gscp_test_folder

### Connecting to local  server 172.31.91.54 ...

### Connecting to remote server 172.31.88.179 ...

### Connecting to remote server 172.31.91.208 ...

// Copies of both files are on every node
$ grun all 'ls /tmp/gscp_text_folder'

### Connecting to local  server 172.31.91.54 ...
gscp_test1 gscp_test2

### Connecting to remote server 172.31.88.179 ...
gscp_test1 gscp_test2

### Connecting to remote server 172.31.91.208 ...
gscp_test1 gscp_test2
```

Download file from another node

```
$ gfetch <all|component_name|node_list> <source_path> <target_dir>
```

This command downloads a file or directory from every specified node to the target directory on the current node.

Example

```
$ gfetch all ~/test.txt ~/test_folder

### Connecting to local  server 172.31.91.54 ...

### Connecting to remote server 172.31.88.179 ...

### Connecting to remote server 172.31.91.208 ...
scp: /home/tigergraph/test.txt: No such file or directory

// Nothing is downloaded if the file does not exist on a node
$ ls ~/test_folder
test.txt_m1  test.txt_m2
```

Run commands on multiple nodes

Run commands sequentially

```
$ grun <all|component_name|node_list> '<command>'
```

This command allows you to run commands on a specified list of nodes in your cluster one by one, and the output from every node will be visible to the terminal. `grun` will wait for the command to finish running on one node before executing the command on the next node.

You can specify which nodes to run commands on in the following ways:

- `grun all '<command>'` will execute the command on all nodes
- `grun <component_name> '<command>'` will execute the command on nodes where the component you specified is running
- `grun <node_list> '<command>'` will execute the commands on the nodes you specify in the node list

Example

All nodes

```
grun all 'echo "hello world"'

### Connecting to local  server 172.31.91.54 ...
hello world

### Connecting to remote server 172.31.88.179 ...
hello world

### Connecting to remote server 172.31.91.208 ...
hello world
```

By component name

```
# Run 'echo "hello world"' on every node where GPE is running
grun gpe 'echo "hello world"'

### Connecting to local  server 172.31.91.54 ...
hello world

### Connecting to remote server 172.31.88.179 ...
hello world

### Connecting to remote server 172.31.91.208 ...
hello world
```

By node list

```
grun m1,m3 'echo "hello world"'

### Connecting to local  server 172.31.91.54 ...
hello world

### Connecting to remote server 172.31.91.208 ...
hello world
```

Run commands in parallel

```
$ grun_p <all|component_name|node_list> '<command>'
```

This command allows you to run commands on a specified list of nodes in your cluster in parallel, and the output will be visible to the terminal where the `grun_p` command was run. You can specify the nodes to run commands on in the following ways:

- `grun_p all '<command>'` will execute the command on all nodes
- `grun_p <component_name> '<command>'` will execute the command on nodes where the component you specified is running
- `grun_p <node_list> '<command>'` will execute the commands on the nodes you specify in the node list. The list of nodes should be separated by a comma, e.g.: `m1,m2`

All nodes

```
$ grun_p all 'echo "hello world"'

### Connecting to local  server 172.31.91.54 ...

### Connecting to remote server 172.31.88.179 ...

### Connecting to remote server 172.31.91.208 ...

### ---- (m1)_172.31.91.54 ---0--
hello world

### ---- (m2)_172.31.88.179 ---0--
hello world

### ---- (m3)_172.31.91.208 ---0--
hello world
```

By component

```
$ grun_p gpe 'echo "hello world"'

### Connecting to local  server 172.31.91.54 ...

### Connecting to remote server 172.31.88.179 ...

### Connecting to remote server 172.31.91.208 ...

### ---- (m1)_172.31.91.54 ---0--
hello world

### ---- (m2)_172.31.88.179 ---0--
hello world

### ---- (m3)_172.31.91.208 ---0--
hello world
```

By node list

```
$ grun_p m1,m3 'echo "hello world"'

### Connecting to local  server 172.31.91.54 ...

### Connecting to remote server 172.31.91.208 ...

### ---- (m1)_172.31.91.54 ---0--
hello world

### ---- (m3)_172.31.91.208 ---0--
hello world
```

Display cluster information

Show current node IP

```
$ gmyip
```

This command returns the private IP address of your current node.

Example:

```
$ gmyip
172.31.93.187 # Current node IP address
```

Show current node number and servers

```
$ ghostname
```

This command returns your current node number as well as all servers that are running on the current node.

Example

In this example, `m1` is the current node number, and `ADMIN#1`, `admin#1` etc. are all servers that are running on `m1`.

```
$ ghostname

m1 ADMIN#1 admin#1 CTRL#1 ctrl#1 DICT#1 dict#1 ETCD#1 etcd#1 EXE_1 exe_1
```

Show deployment information

```
$ gssh
```

The `gssh` command, when used without arguments, outputs information about server deployments in your cluster. The output contains the names and IP addresses of every node. For each node, the output shows the full list of servers that are running on the node. For each server, the output shows the full list of the nodes that the server is running on.

Example

```
$ gssh

Usage: gssh m1|gpe_1#1|gse1_1#1|...
Usage: ----------------Available hosts--------------
Host *
    IdentityFile /home/tigergraph/.ssh/tigergraph_rsa
    Port 22

Host m1 ADMIN#1 admin#1 CTRL#1 ctrl#1 DICT#1 dict#1 ETCD#1 etcd#1 EXE_1 e
    HostName 172.31.91.54

Host m2 ADMIN#2 admin#2 CTRL#2 ctrl#2 DICT#2 dict#2 ETCD#2 etcd#2 EXE_2 e
    HostName 172.31.88.179

Host m3 ADMIN#3 admin#3 CTRL#3 ctrl#3 DICT#3 dict#3 ETCD#3 etcd#3 EXE_3 e
    HostName 172.31.91.208

#cluster.nodes: m1:172.31.91.54,m2:172.31.88.179,m3:172.31.91.208
#admin.servers: m1,m2,m3
#ctrl.servers: m1,m2,m3
#dict.servers: m1,m2,m3
#etcd.servers: m1,m2,m3
#exe.servers: m1,m2,m3
#gpe.servers: m1,m2,m3
#gse.servers: m1,m2,m3
#gsql.servers: m1,m2,m3
#gui.servers: m1,m2,m3
#ifm.servers: m1,m2,m3
#kafka.servers: m1,m2,m3
#kafkaconn.servers: m1,m2,m3
#kafkastrm-ll.servers: m1,m2,m3
#nginx.servers: m1,m2,m3
#restpp.servers: m1,m2,m3
#ts3.servers: m1,m2,m3
#ts3serv.servers: m1
#zk.servers: m1,m2,m3
#log.root: /home/tigergraph/tigergraph/log
#app.root: /home/tigergraph/tigergraph/app/3.1.1
#data.root: /home/tigergraph/tigergraph/data
```

Show graph status

```
$ gstatusgraph
```

This command returns the size of your data, the number of existing vertices and edges, as well as the number of deleted and skipped vertices on every node in your cluster. If you are running TigerGraph on a single node, it will return the same information that one node.

Single-node example

```
$ gstatusgraph
=== graph ===
[GRAPH  ] Graph was loaded (/home/tigergraph/tigergraph/data/gstore/0/par
[m1     ] Partition size: 437MiB, IDS size: 103MiB, Vertex count: 3181724
[WARN   ] Above vertex and edge counts are for internal use which show ap
```

Cluster example

```
$ gstatusgraph
=== graph ===
[GRAPH  ] Graph was loaded (/home/tigergraph/tigergraph/data/gstore/0/par
[m1     ] Partition size: 246MiB, IDS size: 31MiB, Vertex count: 1152822,
[m2     ] Partition size: 248MiB, IDS size: 31MiB, Vertex count: 1157325,
[m3     ] Partition size: 225MiB, IDS size: 29MiB, Vertex count: 1049883,
[WARN   ] Above vertex and edge counts are for internal use which show ap
```

Cross-Region Replication

TigerGraph's Cross-Region Replication (CRR) feature allows users to keep two or more TigerGraph clusters in different data centers or regions in sync. One cluster is the primary cluster, where users would perform all normal database operations, while the other is a read-only Disaster Recovery(CR) cluster that syncs with the primary cluster. CRR includes complete native support for syncing all data and metadata including automated schema, user, and query changes.

For users of TigerGraph, cross-region replication will help deliver on the following business goals:

- **Disaster Recovery**: Support Disaster Recovery functionality with the use of a dedicated remote cluster
- **Enhanced Availability**: Enhance Inter-cluster data availability by synchronizing data using Read Replicas across two clusters
- **Enhanced Performance**: If the customer application is spread over different regions, CRR can take advantage of data locality to avoid network latency.
- **Improved System Load-balancing**: CRR allows you to distribute computation load evenly across two clusters if the same data sets are accessed in both clusters.
- **Data Residency Compliance**: Cross-Region replication allows you to replicate data between different data centers or Regions to satisfy compliance requirements. Additionally, this feature can be used to set up clusters in the same region to satisfy more stringent Data sovereignty or localization business requirements.
- Besides providing disaster recovery and enhanced business continuity, CRR also allows you to set up the clusters as part of Blue/Green deployment purposes for agile upgrades.

This page describes the procedure to set up a DR cluster, and the steps to perform a failover in the event of a disaster.

What is included

The following information is automatically synced from the primary cluster to the DR cluster:

- All data in every graph
- All graph schemas, including tag-based graphs
- All schema change jobs
- All users and roles
- All queries in every graph. Queries that are installed in the primary cluster will be automatically installed in the DR cluster.

Exclusions

The following information and commands are **not** synced to the DR cluster

- GraphStudio metadata
  - This includes graph layout data and user icons for GraphStudio.
- Loading jobs
- `gadmin` configurations
- `gsql --reset` command
- The following GSQL commands:
  - `EXPORT` and `IMPORT` commands
  - `DROP ALL` and `CLEAR GRAPH STORE`

> When the primary cluster executes an `IMPORT` , `DROP ALL`, or `CLEAR GRAPH STORE` GSQL command, or the `gsql --reset` bash command, the services on the DR cluster will stop syncing with the primary and become outdated.
>
> See Sync an outdated DR cluster on how to bring an outdated DR cluster back in sync.

Before you begin

- Install TigerGraph 3.2 or higher on both the primary cluster and the DR cluster **in the same version**.
- Make sure that your DR cluster has the same number of partitions as the primary cluster.
- Make sure that your DR cluster is in the same VPC as your primary cluster.
- Make sure TigerGraph is not installed with a local loopback IP such as 127.0.0.1.

Setup

The following setup is needed in order to perform a failover in the event of a disaster.

> ⓘ This setup assumes that you are setting up a DR cluster for an existing primary cluster. If you are setting up both the primary cluster and DR cluster from scratch, you only need to perform Step 3 after TigerGraph is installed on both clusters.

Step 1: Backup primary data

Use GBAR to create a backup of the primary cluster. See Backup and Restore on how to create a backup.

If you are setting up both the primary cluster and the DR cluster from scratch, you can skip Steps 1, 2, and 4 and only perform Step 3.

Step 2: Restore on the DR cluster

Copy the backup files from every node to every node on the new cluster. Restore the backup of the primary cluster on the DR cluster. See Backup and Restore on how to restore a backup.

Step 3: Enable CRR on the DR cluster

Run the following commands on the DR cluster to enable CRR on the DR cluster.

```
# Enable Kafka Mirrormaker
gadmin config set System.CrossRegionReplication.Enabled true

# Kafka mirrormaker primary cluster's IPs, separator by ','
gadmin config set System.CrossRegionReplication.PrimaryKafkaIPs PRIMARY_I

# Kafka mirrormaker primary cluster's KafkaPort
gadmin config set System.CrossRegionReplication.PrimaryKafkaPort 30002

# The prefix of GPE/GUI/GSQL Kafka Topic, by default is empty.
gadmin config set System.CrossRegionReplication.TopicPrefix Primary

# Apply the config changes, init Kafka, and restart
gadmin config apply -y
gadmin init kafka -y
gadmin restart all -y
```

Step 4: Force install a query on primary

Run the `INSTALL QUERY -force <query_name>` command on the primary cluster to force install any query (Replace `<query_name>` with the name of any query). After the command is finished, all other metadata operations on the primary cluster will start syncing to the DR cluster.

Restrictions on the DR cluster

After being set up, the DR cluster will be read-only and all data update operations will be blocked. This includes the following operations:

- All metadata operations
    - Schema changes
    - User access management operations
    - Query creation, installation, and dropping
    - User-defined function operations

- Data-loading operations
  - Loading jobs operations
  - RESTPP calls that modify graph data
- Queries that modify the graph

Fail over to the DR cluster

In the event of catastrophic failure that has impacted the full cluster due to Data Center or Region failure, the user can initiate the failover to the DR cluster. This is a manual process. Users will have to make the following configuration changes on the DR cluster to upgrade it to the primary cluster.

```
gadmin config set System.CrossRegionReplication.Enabled false
gadmin config set System.CrossRegionReplication.PrimaryKafkaIPs
gadmin config set System.CrossRegionReplication.PrimaryKafkaPort
gadmin config set System.CrossRegionReplication.TopicPrefix Primary
gadmin config apply -y
gadmin restart -y
```

Set up a new DR cluster after failover

After you fail over to your DR cluster, your DR cluster is now the primary cluster. You may want to set up a new DR cluster to still be able to recover your services in the event of another disaster.

To set up a new DR cluster over the upgraded primary cluster:

1. Make a backup of the upgraded primary cluster
2. Run the following command on the new cluster. The commands are the mostly same as setting up the first DR cluster, except that in the fourth command, the value for `System.CrossRegionReplication.TopicPrefix` becomes `Primary.Primary` instead of `Primary`
3. On the new DR cluster, restore from the backup of the upgraded primary cluster

```
# Enable Kafka Mirrormaker
gadmin config set System.CrossRegionReplication.Enabled true

# Kafka mirrormaker primary cluster's IPs, separator by ','
gadmin config set System.CrossRegionReplication.PrimaryKafkaIPs PRIMARY_I

# Kafka mirrormaker primary cluster's KafkaPort
gadmin config set System.CrossRegionReplication.PrimaryKafkaPort 30002

# The prefix of GPE/GUI/GSQL Kafka Topic, by default is empty.
gadmin config set System.CrossRegionReplication.TopicPrefix Primary.Prima

# Apply the config changes, init Kafka, and restart
gadmin config apply -y
gadmin init kafka -y
gadmin restart all -y
```

There is no limit on the number of times a cluster can fail over to another cluster. When designating a new DR cluster, make sure that you set the `System.CrossRegionReplication.TopicPrefix` parameter correctly by adding an additional `.Primary` .

For example, if your original cluster fails over once, and the current cluster's `TopicPrefix` is `Primary`, then the new DR cluster needs to have its `TopicPrefix` be `Primary.Primary`. If it needs to fail over again, the new DR cluster needs to have its `TopicPrefix` be set to `Primary.Primary.Primary`.

Sync an outdated DR cluster

When the primary cluster executes an `IMPORT`, `DROP ALL`, or `CLEAR GRAPH STORE` GSQL command, or the `gsql --reset` bash command, the services on the DR cluster will stop syncing with the primary and become outdated.

To bring an outdated cluster back in sync, you need to generate a fresh backup of the primary cluster, and perform the setup steps again. However, you can skip Step 3: Enable CRR on the DR cluster, because CRR will have already been enabled.

System Administration FAQs

How do I apply or update my license key?

If you have a version 1.0 string-type license key, then during initial platform installation, you can either specify your license key as an argument, for example:

```
./install.sh -l <your_license_key>
```

Or you may input it when prompted.

To apply a new license key string, use the following command:

```
gadmin license set <license_key>
or
gadmin license set @textfile
```

If you have a version 2.0 file-type license key which is linked to a specific machine or cluster:

- If this is the initial installation or you are updating a previous key file, then please see the document Activating a System-Specific License
- If you are updating from a version 1.0 key string to a version 2.0 key file, please contact support@tigergraph.com ↗for the correct procedure.

When does my license key expire?

If you have a version 1.0 string-type license key, the following command will tell you your key's expiration date:

```
gadmin status license
```

If you have a version 2.0 file-type license key which is linked to a specific machine or cluster, then run the following command:

```
curl -X GET "localhost:9000/showlicenseinfo"
```

If you are running TigerGraph v3.0+, run the following command:

```
gadmin license status
```

What are the components of the TigerGraph platform?

```
Command Line Usage Name : Official Name

"GPE" : GPE,
"GSE" : GSE,
"RESTPP" : RESTPP,
"GSQL" : GSQL,
"EXE" : EXECUTOR,
"IFM" : INFORMANT,
"GUI" : GUI,
"CTRL" : CONTROLLER,
"KAFKA" : KAFKA,
"ETCD" : ETCD,
"ZK" : ZK,
"NGINX" : NGINX,
"TS3" : TS3,
"TS3SERV" : TS3SERV,
"DICT" : DICT,
"ADMIN" : ADMIN,
"KAFKACONN" : KAFKACONN,
"KAFKASTRM-LL" : KAFKASTRMLL,
```

A description of each component is given in the Glossary section of the [TigerGraph Platform Overview](#) document.

How can I find out current status of the system?

The following command tells you the basic summary of each component:

```
gadmin status
```

If you want to know more, including process information, memory/cpu usage information of each component, use the -v option for verbose output.

```
gadmin status -v
```

How can I find out the port of a service?

To find out the port of a service, use the `gadmin config get <port_name>` command:

```
$ gadmin config get RESTPP.NginxPort
```

To list and edit all ports, use the following command:

```
gadmin config group port
```

To change the port number of one service, use the following command:

```
gadmin config set <port_name> <port_number>
```

How do I backup my data?

GBAR is the utility to do backup and restore of TigerGraph system. Before a backup, GBAR needs to be configured. Please see GBAR - Graph Backup and Restore for details.

To backup the current system:

```
gbar backup -t <tag_of_the_backup>
```

Please be advised that GBAR only backs up data and configuration. No logs or binaries will be backed up.

How do I restore a backup?

To restore an existing backup:

```
gbar restore <tag_of_the_backup>
```

Please be advised that running restore will STOP the service and ERASE existing data.

How can I find out statistics of my graph data?

You can get statistics of Graph data on TigerGraph database instance using **gstatusgraph** utility:

```
Syntax:

gstatusgraph [-s <node_name>]

using -s to do statistics for one node
```

```
$ gstatusgraph
=== graph ===
[GRAPH   ] Graph was loaded (/data/tigergraph/tigergraph3/data/gstore/0/pa
[m1      ] Partition size: 43GiB, IDS size: 16GiB, Vertex count: 262053633
[m2      ] Partition size: 40GiB, IDS size: 16GiB, Vertex count: 261996922
[m3      ] Partition size: 44GiB, IDS size: 16GiB, Vertex count: 271436710
[m4      ] Partition size: 44GiB, IDS size: 16GiB, Vertex count: 262030593
[WARN    ] Above vertex and edge counts are for internal use which show ap
```

Due to a known bug, **gstatusgraph** command will count each undirected edge as two edges. To get an accurate number of undirected edges, user should use the built-in queries instead. The message below is sent as a warning to users when **gstatusgraph** is used.

`"[WARN ] Above vertex and edge counts are for internal use which show approximate topology size of the local graph partition. Use DML to get the correct graph topology information"`

How can I find out statistics of requests?

TigerGraph provides a RESTful API to tell request statistics. Assuming REST port is 9000, use command below:

```
curl -l http://localhost:9000/statistics
```

How do I restart a service?

If you need to restart everything, use the following:

```
gadmin restart
```

If you know which component(s) you want to restart,you can list them:

```
gadmin restart <component_name(s)>
```

Multiple component names are separated by spaces.

How to I stop some or all services?

Normally it is not necessary to manually turn off any services. However if you wish to, use the stop command.

```
# stop (nearly) all services
# will stop services except for infrastructure services
gadmin stop

# stop selected services
gadmin stop <component_name(s)>
```

Why the service is down?

There are a few typical causes for a service being down:

1. Expired license key.
   Double check your license key expiration date, and contact support@tigergraph.com ↗if it is expired. After applying a new license key, your service will come back online.
   Usually, TigerGraph will reach out before your license key expires. Please act accordingly when that happens.
2. Not enough memory.
   TigerGraph is a memory intensive system. When there is not much free memory, Linux may kill a process based on memory usage. Please check your memory usage after TigerGraph starts. We suggest at least 30% free memory after TigerGraph starts up.
   To confirm if one of TigerGraph's processes is a victim, use dmesg ↗to check.
3. Not enough free disk space.
   TigerGraph writes data, logs, as well as some temporary files onto disk(s). It requires enough free space to function properly. If TigerGraph service or one of its components is down, please check whether there is enough free space on the disk using df ↗.

Where are the logs?

Use following command to figure out where are log files for each component:

```
gadmin log
```

To log at the log file for a particular component:

```
gadmin log <component>
```

Why has my request timed out?

Timeout is applied to any request coming into TigerGraph system. If a request runs longer than the Timeout value, it will be killed. The default timeout value is 16 second.

If you knows that your query will run longer than the value, configure all related timeouts to a bigger value. To do this:

```
gadmin config entry RESTPP.Factory.DefaultQueryTimeoutSec
```

Input a value you expected, the unit is in second. Then apply the config to the system and restart the service.

```
gadmin config apply
gadmin restart
```

The timeout can also be changed for each query, but only when calling the REST endpoint. You would need to use a timeout value each time you run a query, otherwise the default timeout value will be assumed.

```
curl -X <GET/POST> -H "GSQL-TIMEOUT: <timeout value in milliseconds>" '<r
```

Where are the core dump files located?

A core dump file is produced by the OS when a certain signal causes a process to terminate. The core dump is a disk file containing an image of the process's memory at the time of termination. This image can be used in a debugger (e.g., gdb) to inspect the state of the program at the time that it terminated.

The TigerGraph installation process configures the operating system to place core dump files in the TigerGraph root directory, with the name **core-%e-%s-%p.%t,** where

- %e: executable filename (without path prefix)
- %s: signal number which caused the dump
- %p: PID of dumped process
- %t: time of dump, expressed as seconds since the epoch

The coredump configuration was set by the following command:

```
echo "$coreLocation/core-%e-%s-%p.%t" > /proc/sys/kernel/core_pattern
```

If you want to alter the location or file name template, you can edit the contents of `/proc/sys/kernel/core_pattern`

Reference

Configuration Parameters

This page lists the configuration parameters that are available for `gadmin config`.

To change a parameter, use the following command:

```
$ gadmin config set <parameter> <value>
```

After updating a parameter, run `gadmin config apply` to apply the change and restart the corresponding services to make the change take effect.

Admin

Name	Description	Example
Admin.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
Admin.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
Admin.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
Admin.BasicConfig.LogConfig.LogLevel	The log level("INFO","WARN","ERROR"), default is INFO	`INFO`
Admin.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Admin.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Admin	`admin`
Admin.BasicConfig.Nodes	The node list for Admin	`[{"HostID":"m1","Partition":0,"Replica":1},`

		`{"HostID":"m2","Partition":0,"Replica":2}]`
Admin.Port	The port for Admin	`12471`

Controller

Name	Description	Example
Controller.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
Controller.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
Controller.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
Controller.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
Controller.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Controller.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Controller	`controller`
Controller.BasicConfig.Nodes	The nodes to deploy Controller	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
Controller.ConfigRepoRelativePath	The relative path (to the System.DataRoot) of config repo where the service config files are stored	`configs`

Controller.FileRepoRelativePath	The relative path (to the System.DataRoot) of the file repo for file management	`files`
Controller.FileRepoVersionNum	The maximum version of files to keep in the file repo	`3`
Controller.LeaderElectionHeartBeatIntervalMS	The maximum interval(milliseconds) at which each service should call controller leader election service to be considered alive.	`6000`
Controller.LeaderElectionHeartBeatMaxMiss	The maximum number of heartbeat can be missed before one service is considered dead by controller	`5`

Dict

Name	Description	Example
Dict.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
Dict.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
Dict.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
Dict.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Dict.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Dict	`dict`

Dict.BasicConfig.Nodes	The node list for Dict	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
Dict.Port	The port for Dict	`17797`

ETCD

Name	Description	Example
ETCD.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
ETCD.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
ETCD.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
ETCD.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
ETCD.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for ETCD	`etcd`
ETCD.BasicConfig.Nodes	The node list for ETCD	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
ETCD.ClientPort	The port of ETCD to listen for client traffic	`20000`
ETCD.DataRelativePath	The data dir of etcd under $DataRoot	`etcd`

ETCD.ElectionTimeoutMS	Time (in milliseconds) for an election to timeout	`1000`
ETCD.HeartbeatIntervalMS	Time (in milliseconds) of a heartbeat interval	`100`
ETCD.MaxRequestBytes	Maximum client request size in bytes the server will accept	`52428800`
ETCD.MaxSnapshots	Maximum number of snapshot files to retain (0 is unlimited)	`5`
ETCD.MaxTxnOps	Maximum number of operations permitted in a transaction	`8192`
ETCD.MaxWals	Maximum number of wal files to retain (0 is unlimited)	`5`
ETCD.PeerPort	The port of ETCD to listen for peer traffic	`20001`
ETCD.SnapshotCount	Number of committed [illegible]	[illegible]

Executor

Name	Description	Example
Executor.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
Executor.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
Executor.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`

Executor.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
Executor.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Executor.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Executor	`executor`
Executor.BasicConfig.Nodes	The nodes to deploy Executors	`[{"HostID":"m1","Partition":1,"Replica":0},{"HostID":"m2","Partition":2,"Replica":0}]`
Executor.DataRelativePath	The data dir of executor under $DataRoot	`executor`
Executor.FileTransferPort	The port for Executor to do file transfer	`9178`
Executor.FileVersionNum	The maximum version of files to keep	`10`
Executor.Port	The serving port for Executor	`9177`
Executor.WatchDogInterval	The process status check	

FileLoader

Name	Description	Example
FileLoader.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
FileLoader.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`

FileLoader.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
FileLoader.BasicConfig.LogConfig.LogLevel	The log level("OFF", "BRIEF", "DEBUG", "VERBOSE"), default is BRIEF	`BRIEF`
FileLoader.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
FileLoader.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for FileLoader	`fileLoader`
FileLoader.Factory.DefaultLoadingTimeoutSec	The default per request loading timeout (s) for FileLoader	`600`
FileLoader.Factory.DefaultQueryTimeoutSec	The default query timeout (s) for FileLoader	`16`
FileLoader.Factory.DynamicEndpointRelativePath	FileLoader's relative (to data root) path to store the dynamic endpoint	`fileLoader/endpoint/`
FileLoader.Factory.DynamicSchedulerRelativePath	FileLoader's relative (to data root) path to store the dynamic scheduler	`fileLoader/scheduler/`
FileLoader.Factory.EnableAuth	Enable authentication of FileLoader	`false`
FileLoader.Factory.HandlerCount	FileLoader's handler count	`4`
FileLoader.Factory.StatsIntervalSec	FileLoader's time interval to collect stats (e.g. QPS)	`60`
FileLoader.GPEResponseBasePort	The port of FileLoader to accept GPE response	`8400`
FileLoader.GSEResponseBasePort	The port of FileLoader to accept GSE response	`8500`

GPE

Name	Description	Example
GPE.BasicConfig.Env	The runtime environment variables, separated by ';'	`LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY_PATH; CPUPROFILE=/tmp/tg_cpu_profiler; CPUPROFILESIGNAL=12; MALLOC_CONF=prof:true,prof_active:false`
GPE.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
GPE.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
GPE.BasicConfig.LogConfig.LogLevel	The log level("OFF", "BRIEF", "DEBUG", "VERBOSE"), default is BRIEF	`BRIEF`
GPE.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
GPE.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for GPE	`gpe`
GPE.BasicConfig.Nodes	The node list for GPE	`[{"HostID":"m1","Partition":1,"Replica":1},{"HostID":"m2","Partition":1,"Replica":2}]`
GPE.Disk.CompressMethod	The compress method of GPE disk data	`nan`

GPE.Disk.DiskStoreRelativePath	The path(relative to temp root) to store GPE temporary disk data	`gpe/disks`
GPE.Disk.LoadThreadNumber	The number of threads to load from disk	`1`
GPE.Disk.SaveThreadNumber	The number of threads to save to disk	`1`
GPE.EdgeDataMemoryLimit	The memory limit of edge data	`-1`
GPE.GPE2GPEResponsePort	The GPE port for receiving response back from other GPEs	`7501`
GPE.GPERequestPort	The GPE port for receiving requests	`7502`
GPE.IdResponsePort	The GPE port for receiving id response from GSE	`7500`
GPE.Kafka.BatchMsgNumber	The number of messages to send in one batch when using async mode. The producer will wait until either this number of messages are ready to send or queue buffer max ms is reached.	`64`
GPE.Kafka.CompressCodec	This parameter allows you to specify the compression codec for all data generated by this producer. Valid values are none, gzip and snappy.	`none`
GPE.Kafka.FetchErrorBackoffTimeMS	How long to postpone the next fetch request for a topic+partition in case of a fetch error.	`6`
	The maximum amount of time the server will block before answering the fetch	

GPE.Kafka.FetchWaitMaxTimeMS	request if there isn't sufficient data to immediately satisfy fetch min bytes.	`10`
GPE.Kafka.MsgMaxBytes	Maximum transmit message size.	`10485760`
GPE.Kafka.QueueBufferMaxMsgNumber	The maximum number of unsent messages that can be queued up the producer when using async mode before either the producer must be blocked or data must be dropped.	`64`
GPE.Kafka.QueueBufferMaxTimeMS	Maximum time to buffer data when using async mode.	`1`
GPE.Kafka.QueueMinMsgNumber	Minimum number of messages per topic+partition in the local consumer queue.	`100000`
GPE.Kafka.RequestRequiredAcks	This field indicates how many acknowledgements the leader broker must receive from ISR brokers before responding to the request.	`1`
GPE.MemoryLimitMB	The total topology memory limit GPE	`0`
GPE.NumberOfHashBucketInBit	The number of bits used to represent hash bucket counts.	`5`
GPE.RebuildThreadNumber	The number of rebuild threads for GPE	`3`
GPE.StopTimeoutMS	Stop GPE timeout	`300000`

GSE

Name	Description	Example
GSE.BasicConfig.Env	The runtime environment variables, separated by ';'	`LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY_PATH; CPUPROFILE=/tmp/tg_cpu_profiler; CPUPROFILESIGNAL=12; MALLOC_CONF=prof:true,prof_active:false`
GSE.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
GSE.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
GSE.BasicConfig.LogConfig.LogLevel	The log level("OFF", "BRIEF", "DEBUG", "VERBOSE"), default is BRIEF	`BRIEF`
GSE.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
GSE.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for GSE	`gse`
GSE.BasicConfig.Nodes	The node list for GSE	`[{"HostID":"m1","Partition":1,"Replica":1},{"HostID":"m2","Partition":1,"Replica":2}]`
GSE.IdRequestPort	The id request serving port of GSE	`6500`
GSE.JournalTopicPrefix	Kafka Topic prefix of GSE journal storage/replication	`GSE_journal_`

GSE.LeaderElectionHeartbeatIntervalMS	Time (in milliseconds) of the GSE heartbeat interval	`60000`
GSE.LeaderElectionTTLSec	The time-to-live of a GSE election participant.A GSE will be kicked out of election if one GSE is not responsive after the TTL.	`30`
GSE.RLSPort	The serving port of GSE RLS	`8900`
GSE.StopTimeoutMS	Stop GSE timeout	`300000`

GSQL

Name	Description	Example
GSQL.BasicConfig.Env	The runtime environment variables, separated by ';'	`CPATH=$CPATH; LD_LIBRARY_PATH=$LD_LIBRARY_PATH;`
GSQL.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
GSQL.BasicConfig.LogConfig.LogLevel	GSQL log level: ERROR, INFO, DEBUG	`DEBUG`
GSQL.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
GSQL.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for GSQL	`gsql`
GSQL.BasicConfig.Nodes	The node list for GSQL	`[{"HostID":"m1","Partition":0,"Replica":1}, {"HostID":"m2","Partition":0,"Replica":2}]`
GSQL.CatalogBackupFileMaxDurationDay	The maximum number of days for catalog backup files to retain	`30`

GSQL.CatalogBackupFileMaxNumber	The maximum number of catalog backup files to retain	`20`
GSQL.DataRelativePath	The data dir of gsql under $DataRoot	`gsql`
GSQL.EnableStringCompress	Enable string compress	`false`
GSQL.FileOutputPolicy	The policy to control file outputs in GSQL queries	`["/"]`
GSQL.GithubBranch	The working branch in provided repository. Will use 'master' as the default branch	`nan`
GSQL.GithubPath	The path to the directory in the github that has TokenBank.cpp, ExprFunctions.hpp, ExprUtil.hpp, e.g. sample_code/src	`nan`
GSQL.GithubRepository	The repository name, e.g. tigergraph/ecosys	`nan`
GSQL.GithubUrl	The url that is used for github enterprise, e.g. https://api.github.com ↗	`nan`
GSQL.GithubUserAcessToken	The credential for github. Set it to 'anonymous' for public access, or empty to not use github	`nan`
GSQL.GrpcMessageMaxSizeMB	The maximum size of grpc message request of gsql	`40`
GSQL.ManageCatalogTimeoutSec	GSQL connection timeout (second) to admin server when trying to	`nan`
GSQL.MaxAuthTokenLifeTimeSec	The maximum lifetime of auth token in seconds, 0 means unlimited	`0`

GSQL.OutputTokenBufferSize	The buffer size for output token from GSQL	`16000000`
GSQL.Port	The server port for GSQL	`8123`
GSQL.QueryResponseMaxSizeByte	Maximum response size in byte	`33554432`
GSQL.RESTPPRefreshTimeoutSec	Refresh time in Seconds of Restpp	`60`
GSQL.SchemaIndexFileNumber	File number	`10`
GSQL.TokenCleaner.GraceTimeSec	The grace time (in seconds) for expired tokens to exist without being cleaned	`0`
GSQL.TokenCleaner.IntervalTimeSec	The running interval of TokenCleaner in seconds	`10800`
GSQL.UserInfoLimit.TokenSizeLimit	The max number of tokens allowed	`60000`
GSQL.UserInfoLimit.UserCatalogFileMaxSizeByte	The file size limit for user metadata in byte	`2097152`
GSQL.UserInfoLimit.UserSizeLimit	The max number of users allowed	`12000`
GSQL.WaitServiceOnlineTimeoutSec	Timeout to wait for all services online	`20`

GUI

Name	Description	Example
GUI.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
GUI.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`

GUI.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
GUI.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
GUI.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
GUI.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for GUI	`gui`
GUI.BasicConfig.Nodes	The node list for GraphStudio	`[{"HostID":"m1","Partition":0,"Replica":1}, {"HostID":"m2","Partition":0,"Replica":2}]`
GUI.ClientIdleTimeSec	The maximum idle time of client-side GraphStudio and AdminPortal before inactivity logout	`604800`
GUI.Cookie.DurationSec	GUI Cookie duration time in seconds	`86400`
GUI.Cookie.SameSite	Default mode: 1; Lax mode: 2; Strict mode: 3; None mode: 4	`3`
GUI.DataDirRelativePath	The relative path of gui data folder (to the System.DataRoot)	`gui`
GUI.EnableDarkTheme	The boolean value on whether or not GUI should enable dark theme	`true`
GUI.GraphStatCheckIntervalSec	The internval(in seconds) GraphStudio wait before checking the graph statistics	`10`

GUI.HTTPRequest.RetryMax	GUI http request max retry times	`4`
GUI.HTTPRequest.RetryWaitMaxSec	GUI HTTP request max retry waiting time in seconds	`30`
GUI.HTTPRequest.RetryWaitMinSec	GUI HTTP request minimum retry waiting time in seconds	`1`
GUI.HTTPRequest.TimeoutSec	GUI HTTP request timeout in seconds	`300`
GUI.Port	The serving port for GraphStudio Websocket communication	`14242`
GUI.RESTPPResponseMaxSizeBytes	The RESTPP response size limit bytes.	`33554432`
GUI.TempDirRelativePath	The relative path of gui temp folder (to the System.TempRoot)	`gui`

Gadmin

Name	Description	Example
Gadmin.StartServiceDefaultTimeoutMS	The start one service default timeout in milliseconds	`30000`
Gadmin.StartStopRequestTimeoutMS	The start/stop service default request timeout in milliseconds	`600000`
Gadmin.StopServiceDefaultTimeoutMS	The stop one service default request timeout in milliseconds	`30000`

Informant

Name	Description	Example
Informant.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
Informant.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
Informant.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
Informant.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
Informant.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Informant.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Informant	`informant`
Informant.BasicConfig.Nodes	The nodes to deploy Informant	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
Informant.DBRelativePath	The relative path (to the System.DataRoot) of informant database source folder	`informant/db`
Informant.GrpcPort	The grpc server port for Informant	`9166`
Informant.RestPort	The restful server port for Informant	`9167`

Informant.RetentionPeriodDay	The period in days for local database records to keep, set -1 for keeping forever	`30`

Kafka

Name	Description	Example
Kafka.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
Kafka.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
Kafka.BasicConfig.LogConfig.LogLevel	The log level for kafka ("TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL" "OFF")	`INFO`
Kafka.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Kafka.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Kafka	`kafka`
Kafka.BasicConfig.Nodes	The node list for Kafka	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
Kafka.DataRelativePath	The data dir of kafka under $DataRoot	`kafka`
Kafka.IOThreads	The number of threads for Kafka IO	`2`
Kafka.LogFlushIntervalMS	The threshold of time for flushing log (ms)	`10000`
Kafka.LogFlushIntervalMessage	The threshold of message for flushing log	`10000`

Kafka.MessageMaxSizeMB	The maximum size of a message of Kafka to be produced (megabytes)	`10`
Kafka.MinInsyncReplicas	The minimal number of insync replicas that must acknowledge, when producer sets acks to 'all'	`1`
Kafka.NetworkThreads	The number of threads for Kafka Network	`4`
Kafka.Port	The serving port for Kafka	`30002`
Kafka.RetentionHours	The minimum age of a log file of Kafka to be eligible for deletion (hours)	`168`
Kafka.RetentionSizeGB	The minimum size of a log file of Kafka to be eligible for deletion (gigabytes)	`40`
Kafka.StartTimeoutMS	Start kafka timeout	`300000`
Kafka.TopicReplicaFactor	The default replica number	`1`

KafkaConnect

Name	Description	Example
KafkaConnect.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
KafkaConnect.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
KafkaConnect.BasicConfig.LogConfig.LogLevel	The log level for kafka connect ("TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL" "OFF")	`INFO`
KafkaConnect.BasicConfig.LogConfig.LogRotationFileN	The maximum number of old log files to retain	`100`

umber		
KafkaConnect.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Kafka connect	`kafkaconn`
KafkaConnect.BasicConfig.Nodes	The node list for Kafka connect	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
KafkaConnect.MaxRequestSize	The max request size of kafka connect producer	`1048576`
KafkaConnect.OffsetFlushIntervalMS	The interval at which Kafka connect tasks' offsets are committed	`10000`
KafkaConnect.Port	The port used for kafka connect	`30003`
KafkaConnect.ReconnectBackoffMS	The amount of time to wait before attempting to reconnect to a given host	`500`
KafkaConnect.RetryBackoffMS	The amount of time to wait before attempting to retry a failed fetch request to a given topic partition	`10000`

KafkaLoader

Name	Description	Example
KafkaLoader.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
KafkaLoader.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`

KafkaLoader.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
KafkaLoader.BasicConfig.LogConfig.LogLevel	The log level("OFF", "BRIEF", "DEBUG", "VERBOSE"), default is BRIEF	`BRIEF`
KafkaLoader.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
KafkaLoader.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for KafkaLoader	`kafkaLoader`
KafkaLoader.Factory.DefaultLoadingTimeoutSec	The default per request loading timeout (s) for KafkaLoader	`600`
KafkaLoader.Factory.DefaultQueryTimeoutSec	The default query timeout (s) for KafkaLoader	`16`
KafkaLoader.Factory.DynamicEndpointRelativePath	KafkaLoader's relative (to data root) path to store the dynamic endpoint	`kafkaLoader/endpoint/`
KafkaLoader.Factory.DynamicSchedulerRelativePath	KafkaLoader's relative (to data root) path to store the dynamic scheduler	`kafkaLoader/scheduler/`
KafkaLoader.Factory.EnableAuth	Enable authentication of KafkaLoader	`false`
KafkaLoader.Factory.HandlerCount	KafkaLoader's handler count	`4`
KafkaLoader.Factory.StatsIntervalSec	KafkaLoader's time interval to collect stats (e.g. QPS)	`60`
KafkaLoader.GPEResponseBasePort	The port of KafkaLoader to accept GPE response	`9400`
KafkaLoader.GSEResponseBasePort	The port of KafkaLoader to accept GSE response	`9500`

KafkaStreamLL

Name	Description	Example
KafkaStreamLL.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
KafkaStreamLL.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
KafkaStreamLL.BasicConfig.LogConfig.LogLevel	The log level for Kafka stream LoadingLog ("TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL" "OFF")	`INFO`
KafkaStreamLL.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
KafkaStreamLL.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Kafka stream LoadingLog	`kafkastrm-ll`
KafkaStreamLL.BasicConfig.Nodes	The node list for Kafka stream LoadingLog	`[{"HostID":"m1","Partition":1,"Replica":0},{"HostID":"m2","Partition":2,"Replica":0}]`
KafkaStreamLL.Port	The port used for Kafka stream LoadingLog	`30004`
KafkaStreamLL.StateDirRelativePath	The relative folder path for Kafka stream LoadingLog state	`kafkastrm-ll`

Nginx

Name	Description	Example

Nginx.AllowedCIDRList	The whitelist of IPv4/IPv6 CIDR blocks to restrict the application access, separate in comma.	`0.0.0.0/0, ::/0`
Nginx.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
Nginx.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
Nginx.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
Nginx.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for Nginx	`nginx`
Nginx.BasicConfig.Nodes	The node list for Nginx	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
Nginx.ClientMaxBodySize	The maximum request size for Nginx in MB	`200`
		`worker_processes __WORKER_PROCESSES__;\ndaemon off;\npid __NGINX_PID_PATH__;\n\nevents {\n worker_connections 10240;\n}\n\nhttp {\n\n server_tokens off;\n\n map $request_uri $request_uri_path {\n \~^(?P\u003cpath\u003e[^?]*)(\\?.*)?$\" $path;\n }\n\n log_format combined_no_query '$remote_addr - $remote_user`

```
[$time_local] '\n
'\"$request_method
$request_uri_path
$server_protocol\"
$status $body_bytes_sent
'\n
'\"$http_referer\"
\"$http_user_agent\"';\n\
n\t#Set allowed CIDR
blocks\n__CIDR_LIST__\n
types {\n
text/html
html htm shtml;\n
text/css
css;\n              text/xml
xml;\n              image/gif
gif;\n              image/jpeg
jpeg jpg;\n
application/javascript
js;\n
application/atom+xml
atom;\n
application/rss+xml
rss;\n\n
text/mathml
mml;\n              text/plain
txt;\n
text/vnd.sun.j2me.app-
descriptor
jad;\n
text/vnd.wap.wml
wml;\n              text/x-
component
htc;\n\n              image/png
png;\n
image/svg+xml
svg svgz;\n
image/tiff
tif tiff;\n
image/vnd.wap.wbmp
wbmp;\n              image/webp
webp;\n              image/x-
icon
ico;\n              image/x-jng
jng;\n              image/x-ms-
```

`bmp`
`bmp;\n\n font/woff`
`woff;\n font/woff2`
`woff2;\n\n`
`application/java-archive`
`jar war ear;\n`
`application/json`
`json;\n`
`application/mac-binhex40`
`hqx;\n`
`application/msword`
`doc;\n`
`application/pdf`
`pdf;\n`
`application/postscript`
`ps eps ai;\n`
`application/rtf`
`rtf;\n`
`application/vnd.apple.mpe`
`gurl`
`m3u8;\n`
`application/vnd.google-`
`earth.kml+xml`
`kml;\n`
`application/vnd.google-`
`earth.kmz`
`kmz;\n`
`application/vnd.ms-excel`
`xls;\n`
`application/vnd.ms-`
`fontobject`
`eot;\n`
`application/vnd.ms-`
`powerpoint`
`ppt;\n`
`application/vnd.oasis.ope`
`ndocument.graphics`
`odg;\n`
`application/vnd.oasis.ope`
`ndocument.presentation`
`odp;\n`
`application/vnd.oasis.ope`
`ndocument.spreadsheet`
`ods;\n`
`application/vnd.oasis.ope`

`ndocument.text`
`odt;\n`
`application/vnd.openxmlformats-officedocument.presentationml.presentation\n`
`pptx;\n`
`application/vnd.openxmlformats-officedocument.spreadsheetml.sheet\n`
`xlsx;\n`
`application/vnd.openxmlformats-officedocument.wordprocessingml.document\n`
`docx;\n`
`application/vnd.wap.wmlc`
`wmlc;\n`
`application/x-7z-compressed`
`7z;\n`
`application/x-cocoa`
`cco;\n`
`application/x-java-archive-diff`
`jardiff;\n`
`application/x-java-jnlp-file`
`jnlp;\n`
`application/x-makeself`
`run;\n`
`application/x-perl`
`pl pm;\n`
`application/x-pilot`
`prc pdb;\n`
`application/x-rar-compressed`
`rar;\n`
`application/x-redhat-package-manager`
`rpm;\n`
`application/x-sea`
`sea;\n`
`application/x-shockwave-`

```
flash
swf;\n
application/x-stuffit
sit;\n
application/x-tcl
tcl tk;\n
application/x-x509-ca-
cert
der pem crt;\n
application/x-xpinstall
xpi;\n
application/xhtml+xml
xhtml;\n
application/xspf+xml
xspf;\n
application/zip
zip;\n\n
application/octet-stream
bin exe dll;\n
application/octet-stream
deb;\n
application/octet-stream
dmg;\n
application/octet-stream
iso img;\n
application/octet-stream
msi msp msm;\n\n
audio/midi
mid midi kar;\n
audio/mpeg
mp3;\n              audio/ogg
ogg;\n              audio/x-m4a
m4a;\n              audio/x-
realaudio
ra;\n\n              video/3gpp
3gpp 3gp;\n
video/mp2t
ts;\n              video/mp4
mp4;\n              video/mpeg
mpeg mpg;\n
video/quicktime
mov;\n              video/webm
webm;\n              video/x-
flv
flv;\n              video/x-m4v
```

Nginx.ConfigTemplate	The template to generate nginx config. Please use `@filepath` to parse template from file. Check the default template first at https://docs.tigergraph.com. [(Warning](https://docs.tigergraph.com): Don't modify the reserved keywords(string like **UPPER_CASE**) in template.)	`m4v;\n video/x-mng mng;\n video/x-ms-asf asx asf;\n video/x-ms-wmv wmv;\n video/x-msvideo avi;\n }\n default_type application/octet-stream;\n client_max_body_size __MAX_BODY_SIZE__;\n\n access_log __NGINX_LOG_PER_RESTPP__ combined_no_query;\n error_log __NGINX_ERR_PER_RESTPP__;\n fastcgi_temp_path __TEMP_ROOT__;\n fastcgi_buffers 256 8k;\n\n ####### [BEGIN] customized headers #######\n __HEADER_CONFIG__\n ####### [END] customized headers #######\n\n\n keepalive_timeout 900s;\n\n upstream fastcgi_backend {\n server unix:__FASTCGI_PASS__;\n keepalive 128;\n }\n\n # Use upstream derivative for listing all gsql server \n # that could be used in requesttoken proxy_pass\n __ENABLE_RESTPP_AUTH__ upstream gsql_token_server {\n __ENABLE_RESTPP_AUTH__ __GSQL_TOKEN_SERVER_LIST__\n __ENABLE_RESTPP_AUTH__`

```
}\n\n    # Keep it for
backward compatibility\n
server {\n
add_header Strict-
Transport-Security \"max-
age=63072000;
includeSubdomains;
preload\";\n
ssl_protocols TLSv1.2;\n
ssl_ciphers ECDHE-ECDSA-
AES128-GCM-SHA256:ECDHE-
RSA-AES128-GCM-
SHA256:ECDHE-ECDSA-
AES256-GCM-SHA384:ECDHE-
RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-
CHACHA20-POLY1305:ECDHE-
RSA-CHACHA20-
POLY1305:DHE-RSA-AES128-
GCM-SHA256:DHE-RSA-
AES256-GCM-SHA384;\n
ssl_prefer_server_ciphers
on;\n        listen
__PORT_PER_RESTPP__
__GUI_CONNECTION_TYPE__;\
n         server_name
localhost;\n\n
fastcgi_read_timeout
72000s;\n
fastcgi_send_timeout
72000s;\n\n
__SSL_CERT_ATTR__
__SSL_CERT_PATH__;  # if
SSL is disabled, here
should be '#'\n
__SSL_KEY_ATTR__
__SSL_KEY_PATH__;  # if
SSL is disabled, here
should be '#'\n\n\n
location / {\n
fastcgi_pass
fastcgi_backend;\n
fastcgi_keep_conn on;\n
fastcgi_param
REQUEST_METHOD
```

```
$request_method;\n
fastcgi_param
CONTENT_TYPE
$content_type;\n
fastcgi_param
CONTENT_LENGTH
$content_length;\n
fastcgi_param REQUEST_URI
$request_uri;\n
fastcgi_param GSQL_ASYNC
$http_gsql_async;\n
fastcgi_param
GSQL_TIMEOUT
$http_gsql_timeout;\n
fastcgi_param
GSQL_MEMLIMIT
$http_gsql_memlimit;\n
fastcgi_param
RESPONSE_LIMIT
$http_response_limit;\n
}\n\n        # To ensure
the performance of
RESTPP, this rule
shouldn't be enabled\n
# unless
restpp.authentication is
True.\n         # And for
performance
consideration,\n
#!!!!!!!   DO NOT USE
REGULAR EXPRESSION HERE
!!!!!!!\n
__ENABLE_RESTPP_AUTH__
location = /requesttoken
{\n
__ENABLE_RESTPP_AUTH__
proxy_ssl_verify
__PROXY_SSL_VERIFY__;\n
__ENABLE_RESTPP_AUTH__
proxy_set_header X-Real-
IP $remote_addr;\n
__ENABLE_RESTPP_AUTH__
proxy_pass
__REQUEST_TOKEN_PROTOCOL_
_://gsql_token_server;\n
```

`__ENABLE_RESTPP_AUTH__ }\n }\n\n server {\n\t\tlisten __NGINX_SERVICES_PORT__ __GUI_CONNECTION_TYPE__;\n\n\t\t__SSL_CERT_ATTR__ __SSL_CERT_PATH__; # if SSL is disabled, here should be '#'\n\t\t__SSL_KEY_ATTR__ __SSL_KEY_PATH__; # if SSL is disabled, here should be '#'\n\n\t\tlocation / {\n\t\t\t# Set whether to enable compression\n\t\t\tgzip on;\n\t\t\tgzip_types\n\t\t\t\tapplication/javascript # works significantly with javascript files in GUI\n\t\t\t;\n\t\t\troot __GST_STATIC_FOLDER__;\n\t\t\ttry_files $uri $uri/ @backend;\n\t\t}\n\n\t\tlocation /assets/img/user-uploaded-icons/ {\n\t\t\talias __GUI_DATA_FOLDER__/user_icons/;\n\t\t\ttry_files $uri $uri/ =404;\n\t\t}\n\n\t\tlocation /admin/ {\n\t\t\t# Set whether to enable compression\n\t\t\tgzip on;\n\t\t\tgzip_types\n\t\t\t\tapplication/javascript # works significantly with javascript files in GUI\n\t\t\t;\n\t\t\talias __ADMIN_PORTAL_STATIC_FOLDER__/;\n\t\t\ttry_files $uri $uri/ =`

`404;\n\t\t}\n\n\t\tlocati` `on @backend` `{\n\t\t\tproxy_read_timeo` `ut` `3600s;\n\t\t\tproxy_set_h` `eader X-Real-IP` `$remote_addr;\n\t\t\tprox` `y_pass` `http://localhost:__GUI_WE` `BSERVER_PORT__;\n\t\t}\n\` `n\t\tlocation ~ ^/ts3/(?` `\u003cts3_uri\u003e.*)` `{\n\t\t\tproxy_read_timeo` `ut 3600s;\n\t\t\trewrite` `^/ts3/(.*) /$ts3_uri` `break;\n\t\t\tproxy_set_h` `eader X-Real-IP` `$remote_addr;\n\t\t\tprox` `y_pass` `http://localhost:__TS3_RE` `ST_PORT__;\n\t\t}\n\n\t\t` `location ~` `^/gsqlserver/(?` `\u003cgsql_uri\u003e.*)` `{\t\n\t\t\trewrite` `^/gsqlserver/(.*)` `/$gsql_uri` `break;\n\t\t\tproxy_read_` `timeout` `3600s;\n\t\t\tproxy_set_h` `eader X-Real-IP` `$remote_addr;\n\t\t\tprox` `y_pass` `http://localhost:__GSQL_S` `ERVER_PORT__;\n\t\t\tprox` `y_http_version` `1.1;\n\t\t}\n\n\t\t# This` `RESTPP endpoint shares` `the same security` `configuration\n\t\tfastcg` `i_read_timeout` `72000s;\n\t\tfastcgi_send` `_timeout` `72000s;\n\n\t\t__ENABLE_R` `ESTPP_AUTH__ location ~`

		`^/restpp/(?\u003ctoken_uri\u003erequesttoken.*) {\n\t\t__ENABLE_RESTPP_AUTH__ rewrite ^/restpp/(.*) /$token_uri break;\n\t\t__ENABLE_RESTPP_AUTH__ proxy_ssl_verify __PROXY_SSL_VERIFY__;\n\t\t__ENABLE_RESTPP_AUTH__ proxy_set_header X-Real-IP $remote_addr;\n\t\t__ENABLE_RESTPP_AUTH__ proxy_pass __REQUEST_TOKEN_PROTOCOL__://gsql_token_server;\n\t\t__ENABLE_RESTPP_AUTH__ }\n\n\t\tlocation ~ ^/restpp/(.*) {\n\t\t\tfastcgi_pass fastcgi_backend;\n\t\t\tfastcgi_keep_conn on;\n\t\t\tfastcgi_param REQUEST_METHOD $request_method;\n\t\t\tfastcgi_param CONTENT_TYPE $content_type;\n\t\t\tfastcgi_param CONTENT_LENGTH $content_length;\n\t\t\tfastcgi_param REQUEST_URI $1?$query_string; # the url pattern matched above\n\t\t\tfastcgi_param GSQL_TIMEOUT [illegible]gsql_timeout;\n\t\t\tfastcgi_param [illegible]aram RESPONSE_LIMIT $http_response_limit;\t\n\t\t}\n\t}\n}\n"`
Nginx.Port	The serving port for Nginx	`14240`
Nginx.ResponseHeaders	The customized headers in HTTP Response	`[{"FieldName":"X-Frame-Options","FieldValue":"SAMEORIGIN"}]`
Nginx.SSL.Cert	Public certificate for SSL. (Could use @cert_file_path	`nan`

	to parse the certificate from file)	
Nginx.SSL.Enable	Enable SSL connection for all HTTP requests	`false`
Nginx.SSL.Key	Private key for SSL. (Could use @key_file_path to parse the key from file)	`nan`
Nginx.WorkerProcessNumber	The number of worker processes for Nginx	`4`

RESTPP

Name	Description	Example
RESTPP.BasicConfig.Env	The runtime environment variables, separated by ';'	`LD_PRELOAD=$LD_PRELOAD; LD_LIBRARY_PATH=$LD_LIBRARY_PATH;`
RESTPP.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
RESTPP.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
RESTPP.BasicConfig.LogConfig.LogLevel	The log level("OFF", "BRIEF", "DEBUG", "VERBOSE"), default is BRIEF	`BRIEF`
RESTPP.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
RESTPP.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for RESTPP	`restpp`
RESTPP.BasicConfig.Nodes	The node list for RESTPP	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`

RESTPP.FCGISocketBackLogMaxCnt	RESTPP fcgi socket backlog max length which is the listen queue depth used in the listen() call.	`36864`
RESTPP.FCGISocketFileRelativePath	The relative path of FCGI socket for RESTPP-Nginx communitation under $TempRoot	`rest/restpp-nginx.fcgi.sock`
RESTPP.Factory.DefaultLoadingTimeoutSec	The default per request loading timeout (s) for RESTPP	`600`
RESTPP.Factory.DefaultQueryTimeoutSec	The default query timeout (s) for RESTPP	`16`
RESTPP.Factory.DynamicEndpointRelativePath	RESTPP's relative (to data root) path to store the dynamic endpoint	`restpp/endpoint/`
RESTPP.Factory.DynamicSchedulerRelativePath	RESTPP's relative (to data root) path to store the dynamic scheduler	`restpp/scheduler/`
RESTPP.Factory.EnableAuth	Enable authentication of RESTPP	`false`
RESTPP.Factory.HandlerCount	RESTPP's handler count	`4`
RESTPP.Factory.QueryMemoryLimitMB	The memory limit of query runs for container on disk. The default value is -1, meaning no limit	`-1`
RESTPP.Factory.StatsIntervalSec	RESTPP's time interval to collect stats (e.g. QPS)	`60`
RESTPP.GPEResponsePort	The port of RESTPP to accept GPE response	`5400`
RESTPP.GSEResponsePort	The port of RESTPP to accept GSE response	`5500`
RESTPP.HttpServer.Enable	Enable RESTPP's http server	`false`

RESTPP.HttpServer.Port	RESTPP's http server port	`10000`
RESTPP.HttpServer.WorkerNum	RESTPP's http server worker number	`2`
RESTPP.NginxPort	The port of RESTPP to accept upstream Nginx requests	`9000`

Security

Name	Description	Example
Security.LDAP.AdminDN	Configure the DN of LDAP user who has read access to the base DN specified above. Empty if everyone has read access to LDAP data: default empty	`nan`
Security.LDAP.AdminPassword	Configure the password of the admin DN specified above. Needed only when admin_dn is specified: default empty	`secret`
Security.LDAP.BaseDN	Configure LDAP search base DN, the root node to start the LDAP search for user authentication: must specify	`nan`
Security.LDAP.Enable	Enable LDAP authentication: default false	`false`
Security.LDAP.Hostname	Configure LDAP server hostname: default localhost	`localhost`
Security.LDAP.Port	Configure LDAP server port: default 389	`389`
Security.LDAP.SearchFilter	Configure LDAP search base DN, the root node to	`(objectClass=*)`

	start the LDAP search for user authentication.	
Security.LDAP.Secure.Protocol	Enable SSL/StartTLS for LDAP connection [none/ssl/starttls]: default none	`none`
Security.LDAP.Secure.TrustAll	Configure to trust all LDAP servers (unsafe): default false	`false`
Security.LDAP.Secure.TruststoreFormat	Configure the truststore format [JKS/PKCS12]: default JKS	`JKS`
Security.LDAP.Secure.TruststorePassword	Configure the truststore password: default changeit	`changeit`
Security.LDAP.Secure.TruststorePath	Configure the truststore absolute path for the certificates used in SSL: default empty	`nan`
Security.LDAP.UsernameAttribute	Configure the username attribute name in LDAP server: default uid	`uid`
Security.SSO.SAML.AssertionSigned	Require Identity Provider to sign assertions: default true	`true`
Security.SSO.SAML.AuthnRequestSigned	Sign AuthnRequests before sending to Identity Provider: default true	`true`
Security.SSO.SAML.BuiltinUser	The builtin user for SAML	`__GSQL__saml`
Security.SSO.SAML.Enable	Enable SAML2-based SSO: default false	`false`
Security.SSO.SAML.IDP.EntityId	Identity Provider Entity ID: default http://idp.example.com ↗	`http://idp.example.com`
Security.SSO.SAML.IDP.SSOUrl	Single Sign-On URL: default http://idp.example.com/sso/saml ↗	`http://idp.example.com/sso/saml`

Security.SSO.SAML.IDP.X509Cert	Identity Provider's x509 Certificate filepath: default empty. You can use @/cert/file/path to pass the certificate from a file.	`nan`
Security.SSO.SAML.MetadataSigned	Sign Metadata: default true	`true`
Security.SSO.SAML.RequestedAuthnContext	Authentication context (comma separate multiple values)	`nan`
Security.SSO.SAML.ResponseSigned	Require Identity Provider to sign SAML responses: default true	`true`
Security.SSO.SAML.SP.Hostname	TigerGraph Service Provider URL: default http://127.0.0.1:14240 ↗	`http://127.0.0.1:14240`
Security.SSO.SAML.SP.PrivateKey	Content of the host machine's private key. Require PKCS#8 format (start with "BEGIN PRIVATE KEY"). You can use @/privatekey/file/path to pass the certificate from a file.	`nan`
Security.SSO.SAML.SP.X509Cert	Content of the x509 Certificate: default empty. You can use @/cert/file/path to pass the certificate from a file.	`nan`
Security.SSO.SAML.SignatureAlgorithm	Signiture algorithm [rsa-sha1/rsa-sha256/rsa-sha384/rsa-sha512]:	`rsa-sha256`

System

Name	Description	Example

System.AppRoot	The root directory for TigerGraph applications	`/home/tigergraph/tigergraph/app`
System.AuthToken	The authorization token for TigerGraph services	`eusMVm3CSwSbQgxVOI6yBqYEp3wUn2Gy`
System.Backup.CompressProcessNumber	The number of concurrent process for compression during backup	`8`
System.Backup.Local.Enable	Backup data to local path	`false`
System.Backup.Local.Path	The path to store the backup files	`nan`
System.Backup.S3.AWSAccessKeyID	The AWS access key ID for s3 bucket of backup	`nan`
System.Backup.S3.AWSSecretAccessKey	The secret access key for s3 bucket	`nan`
System.Backup.S3.BucketName	The S3 bucket name	`nan`
System.Backup.S3.Enable	Backup data to S3 path	`false`
System.Backup.TimeoutSec	The backup timeout in seconds	`18000`
System.CrossRegionReplication.Enabled	Enable Kafka Mirrormaker	`false`
System.CrossRegionReplication.GSQLTopic	The name of the topic shared by GSQL and GUI	`GSQL`
System.CrossRegionReplication.PrimaryKafkaIPs	Kafka mirrormaker primary cluster's IPs, separator by ','	`nan`
System.CrossRegionReplication.PrimaryKafkaPort	Kafka mirrormaker primary cluster's KafkaPort	`30002`
System.CrossRegionReplication.TopicPrefix	The prefix of GPE/GUI/GSQL Kafka Topic, by default is empty.	`nan`

System.DataRoot	The root directory for data	`/home/tigergraph/tigergraph/data`
System.Event.EventInputTopic	Kafka topic name of event input queue	`EventInputQueue`
System.Event.EventOffsetFolderRelativePath	The relative path (to the System.DataRoot) of the folder to keep track of Kafka offsets for event input/output queue	`offset`
System.Event.EventOutputTopic	Kafka topic name of event output queue	`EventOutputQueue`
System.HostList	The aliases and hostnames/IPs for nodes	`[{"ID":"m1","Hostname":"192.168.1.1","Region":"r1"},{"ID":"m2","Hostname":"192.168.1.2","Region":"r2"}]`
System.License	The license key for TigerGraph system	`nan`
System.LogRoot	The root directory for TigerGraph logs	`/home/tigergraph/tigergraph/log`
System.SSH.ConfigFileRelativePath	The relative path (to the System.DataRoot) of SSH config file	`ssh/ssh_config`
System.SSH.Port	SSH port	`22`
System.SSH.User.Password	OS User password (optional if using privatekey)	`tigergraph`
System.SSH.User.Privatekey	OS user private key path	`nan`
System.SSH.User.Username	OS Username for TigerGraph database	`tigergraph`

TS3

Name	Description	Example
TS3.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
TS3.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
TS3.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
TS3.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
TS3.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
TS3.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for TS3	`ts3`
TS3.BasicConfig.Nodes	The node list for TS3	`[{"HostID":"m1","Partition":1,"Replica":0},{"HostID":"m2","Partition":2,"Replica":0}]`
TS3.BufferSize	The buffer size of TS3	`10`
TS3.DBRelativePath	The relative path (to the System.DataRoot) of TS3 database source folder	`ts3/db`
TS3.DbTrace	Enable tracing for db operations	`false`
TS3.Metrics	The metrics TS3 will be collecting	`cpu,mem,diskspace,qps,servicestate,network,connection`
TS3.RetentionPeriodDay	The period in days for local database records to keep, set -1 for keeping forever	`30`

TS3Server

Name	Description	Example
TS3Server.BasicConfig.Env	The runtime environment variables, separated by ';'	`nan`
TS3Server.BasicConfig.LogConfig.LogFileMaxDurationDay	The maximum number of days to retain old log files based on the timestamp encoded in their filename	`90`
TS3Server.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
TS3Server.BasicConfig.LogConfig.LogLevel	The log level("DEBUG","INFO","WARN","ERROR","PANIC","FATAL"), default is INFO	`INFO`
TS3Server.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
TS3Server.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for TS3Serer	`ts3serv`
TS3Server.BasicConfig.Nodes	The node list for TS3Server(Currently only support one node)	`[{"HostID":"m1","Partition":0,"Replica":1},{"HostID":"m2","Partition":0,"Replica":2}]`
TS3Server.GrpcPort	The grpc api port for TS3Server	`19001`
TS3Server.RestPort	The restful api port for TS3Server	`19000`

ZK

Name	Description	Example
ZK.BasicConfig.Env	The runtime environment variables, separated by ';'	`ZK_SERVER_HEAP=4096;`
ZK.BasicConfig.LogConfig.LogFileMaxSizeMB	The maximum size in megabytes of the log file before it gets rotated	`100`
ZK.BasicConfig.LogConfig.LogLevel	The log level for zk ("TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL" "OFF")	`INFO`
ZK.BasicConfig.LogConfig.LogRotationFileNumber	The maximum number of old log files to retain	`100`
ZK.BasicConfig.LogDirRelativePath	The relative path (to the System.LogRoot) of log directory for ZK	`zk`
ZK.BasicConfig.Nodes	The node list for Zookeeper	`[{"HostID":"m1","Partition":0,"Replica":1}, {"HostID":"m2","Partition":0,"Replica":2}]`
ZK.DataRelativePath	The data dir of zookeeper under $DataRoot	`zk`
ZK.ElectionPort	The port for Zookeeper to do leader election	`3888`
ZK.ForceSync	The force syncronize property of zookeeper	`false`
ZK.InitLimit	The amount of time, in ticks(by default 2s for one tick), to allow followers to connect and sync to a leader. Increased this value as needed, if the amount of data managed by ZooKeeper is large	`30`
ZK.Port	The serving port for Zookeeper	`19999`

ZK.QuorumPort	The port for Zookeeper to do peer communication	`2888`

download/upload/delete catalog

Name	Description	Example
download/upload/delete catalog. Default value: 20	nan	`nan`

Vertex-level Access Control Functions

These functions are only available for vertex aliases (defined in the `FROM` clause of a `SELECT` statement); they cannot be applied to vertex variables in other contexts.

Function	Description	Return type
`v.isTaggable()`	Return true if v is of a taggable vertex type.	`BOOL`
`v.getTags()`	Return v's set of tags. If v is untaggable, it returns an empty set.	`SET<STRING>`
`v.hasTags(` `STRING tag1,... STRING tagN)`	Return true if v has every tag in the argument list of tags.	`BOOL`
`v.intersectTags(` `VERTEX v2)`	Return the set of tags that v and v2 have in common.	`SET<STRING>`
`v.differenceTags(` `VERTEX v2)`	Return the set of tags that v has but v2 does not have.	`SET<STRING>`
`v.addTags(` `STRING tag1,... STRING tagN)`	Add the given tags to v.	n/a
`v.removeTags(` `STRING tag1,...` `STRING tagN)`	Remove the given tags from v.	n/a
`v.removeAllTags()`	Remove all tags from V.	n/a

Security Roles: In order to use these functions, a user must have `admin` or `designer` roles on the base graph. These funcitons cannot be used on a tag-based graph.

`isTaggable()`

This function returns true if the vertex is taggable.

Example:

```
//count the number of taggable vertices in the graph.
CREATE QUERY countIstaggable() for graph poc_graph_tag {
  SumAccum<int> @@count;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s
         WHERE s.isTaggable()
         ACCUM @@count += 1;
  PRINT @@count;
}
```

`getTags()`

This function returns the vertex's tags as a set. If the vertex has no tags or is untaggable, it returns an empty set.

Example:

```
//print the tags of each vertices, in 2 different ways.
CREATE QUERY exampleGettags() {
  SetAccum<string> @vAcc;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.@vAcc += s.getTags();
  PRINT vSet[vSet.@vAcc];
  PRINT vSet[vSet.gettags()];
}
```

Tip: `getTags()` can be used within a `PRINT` statement:

- `PRINT R [R.getTags()];`

- or `PRINT R WITH TAGS` which is syntax sugar, except that it won’t print `"R.getTags()": []` for non-taggable vertices.

`hasTags(STRING tag1,... STRING tagN)`

This function returns `true` if the vertex has every tag provided in the argument list and returns `false` if it does not.

Example:

```
USE GRAPH socialNet

CREATE QUERY findVertexWithTag(STRING tag) {
  seed = { ANY };
  res =
    SELECT v
    FROM seed:v
    WHERE v.hasTags(tag)
    ORDER BY v.id;
  PRINT res WITH TAGS;
}

INSTALL QUERY findVertexWithTag

RUN QUERY findVertexWithTag("vip")
```

The output of the query would be:

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"res": [
    {
      "v_id": "person6",
      "attributes": {
        "gender": "Male",
        "id": "person6",
        "res.gettags()": [
          "vip",
          "public"
        ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "id": "person7",
        "res.gettags()": [
          "vip",
          "public"
        ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {
        "gender": "Male",
        "id": "person8",
        "res.gettags()": ["vip"]
      },
      "v_type": "person"
    }
  ]}]
}
```

`intersectTags(VERTEX v2)`

This function returns the common tags between the vertex and another vertex as a set.

Example:

```
//return the intersect set of tags between two vertices.
CREATE QUERY exampleIntersecttags() {
  SetAccum<string> @vAcc;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s -(_)-> :t
         WHERE t.type == "person"
         ACCUM s.@vAcc += s.intersectTags(t);
  PRINT vSet[vSet.@vAcc];
}
```

`differenceTags(VERTEX v2)`

This function returns the difference in tags between the vertex and another vertex as a set.

Example:

```
//return the difference set of tags between two vertices
CREATE QUERY exampleDifferencetags() {
  SetAccum<string> @vAcc;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s -(_)-> :t
         WHERE t.type == "person"
         ACCUM s.@vAcc += s.differenceTags(t);
  PRINT vSet[vSet.@vAcc];
}
```

`addTags(STRING tag1,... STRING tagN)`

This function adds the tags provided in the argument list to the vertex.

Example:

```
CREATE QUERY addTagsToPerson() {
  Seed = { any };
  # person1 ~ person5 will be tagged as public.
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id IN ("person1","person2","person3","person4","person5"
         ACCUM s.addTags("public");

  # person6 and person7 will be tagged as public and vip.
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id IN ("person6","person7")
         ACCUM s.addTags("vip", "public");

  # person8 will be tagged as vip
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id == "person8"
         ACCUM s.addTags("vip");
}
```

`removeTags(STRING tag1,... STRING tagN)`

This function removes the tags provided in the argument list from the vertex.

```
//remove tag “vip” and “public” from all person vertices.
CREATE QUERY removetagsFromPerson() {
  vSet = { person.* };
  # remove tag vip and public from all person vertices
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.removeTags("vip", "public");
}
```

`removeAllTags()`

This function removes all tags from the vertex.

Example:

```
//remove all tags from all person vertices.
CREATE QUERY removealltagsFromPerson() {
  vSet = { person.* };
  # remove all tags from all person vertices
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.removeAllTags();
}
```

List of Privileges

This page provides a complete list of privileges in TigerGraph's Role-based Access Control system.

- Any privilege marked “on global only” can only be granted to a global role. It cannot be granted to a local role (See Global role vs local role).
- The command `IMPORT GRAPH <gName>` needs multiple privileges, .e.g `WRITE_SCHEMA`, `WRITE_LOADING_JOB`, `WRITE_QUERY` and so on.
- To run the command `CREATE SECRET` on a graph, the user must have at least one of the access database privileges: `READ_DATA`, `WRITE_DATA` and `EXECUTE_LOADINGJOB` on that graph. Thus the built-in `queryreader` role and above can create secrets on a graph, but the observer role cannot.

Table of Privileges

Privilege Name	Commands Associated	Global Only

`READ_SCHEMA`	`ls`
	`show vertex <vName>`
	`show edge <eName>`
	`show graph <gName>`
	`show job (<schema_changeJobName>`
		No

`WRITE_SCHEMA`	`create schema_change job <scjName>`
	`run schema_change job <scjName>`
	`drop schema_change job <scjName>`
	`create vertex <vName>`
	`drop vertex <vName>`
	`create edge <eName>`
	`drop edge <eName>`
	`create graph <gName>`
		No

	`create global schema_change job <gscjName>`
	`run global schema_change job <gscjName>`
	`drop global schema_change job <gscjName>`	

`READ_LOADINGJOB`	`show job <loadingJobName>` `show data_source <dsName>`	No

`EXECUTE_LOADINGJOB`	`run loading job <ljName>`
	`show loading status <jobId>`
	`abort loading job <ljName>`
	`resume loading job <ljName>`
		No

`WRITE_LOADINGJOB`	`create loading job <ljName>` `drop loading job <ljName>`	No
`READ_QUERY`	`show query <qName>`	No

`WRITE_QUERY`	`create query <qName>`
	`install query <qName>`
	`drop query <qName>`
		No

`READ_DATA`	`run Read_Only_Query <qName>`	No
`WRITE_DATA`	`run Query_With_Update <qName>` `(including create, delete, update in its own or any sub-query)`	No

`WRITE_DATASOURCE`	`create data_source <dsName>`
	`grant data_source <dsName>`
	`revoke data_source <dsName>`
	`drop data_source <dsName>`
		No

`READ_ROLE`	`show role` `show privilege on role <rName>`	No

`WRITE_ROLE`	`create role <rName>`
	`grant role <rName>`
	`revoke role <rName>`
	`drop role <rName>`
	`grant privilege <pName> on graph <gName> to <rName>`
	`revoke privilege <pName> on graph <gName> from <rName>`
		No

`READ_USER`	`show user`
	`show privilege on user <uName>`
	`show secret`
		No

`WRITE_USER`	`create user <uName>`
	`drop user <uName>`
	`alter password`
		Yes

`READ_PROXYGROUP`	`show group`	No
`WRITE_PROXYGROUP`	`create group <pgName> proxy <rule>` `drop group <pgName>`	Yes
`READ_FILE`	`get <fileName> to <path-to-file>`	Yes

WRITE_FILE	put <fileName> from <path-to-file>	Yes
DROP_GRAPH	drop graph <gName>	Yes
EXPORT_GRAPH	export graph <gName>	Yes
CLEAR_GRAPHSTORE	clear graph store	Yes
DROP ALL	drop all	Yes
ACCESS_TAG	create/drop/run schema_change jobs involving tags create/drop/install/run queries involving tags create/drop/run loading	NO

Web UI

GraphStudio UI Guide

Known Issues

GraphStudio Patent and Third Party Notice

GraphStudio Overview

Overview

The TigerGraph GraphStudio™ UI (User Interface) provides an intuitive, browser-based interface that helps users get started quickly with graph-based application development tasks: designing a graph schema, creating a schema mapping, loading data, exploring the graph, and writing GSQL queries. This guide serves as an introduction and quick-start manual for the GraphStudio UI.

As of Jan 2020, the GraphStudio UI is certified on following browsers:

Browser	Chrome	Safari	Firefox	Opera	Edge	Inte Exp
Supported version	54.0+	11.1+	59.0+	52.0+	80.0+	10+

Not all features are guaranteed to work on other browsers.

Please make sure to enable JavaScript and cookies in your browser settings.

GraphStudio In The Cloud

If you are using GraphStudio in the TigerGraph cloud environment, you can directly access GraphStudio via a browser.

GraphStudio On-Premises

For on-premise deployment, the system by default is listening to port 14240. Any machine connected to the server can access GraphStudio from a browser with the following address:

```
http://<your_tigergraph_server_ip_address>:<your_tigergraph_server_port>
```

In v1.2, the default TCP/IP port for GraphStudio has changed from 44240 to 14240, to avoid possible port conflicts with Zookeeper.

If you cannot access GraphStudio, check your firewall rules and open 14240 port to public. For example, if your Linux OS uses firewall-cmd:

```
$ firewall-cmd -zone=public -add-port=14240/tcp -permanent
```

If the GraphStudio UI shows some errors at the login page like below, the GUI service might be off.

To verify, in a linux shell of the server, type

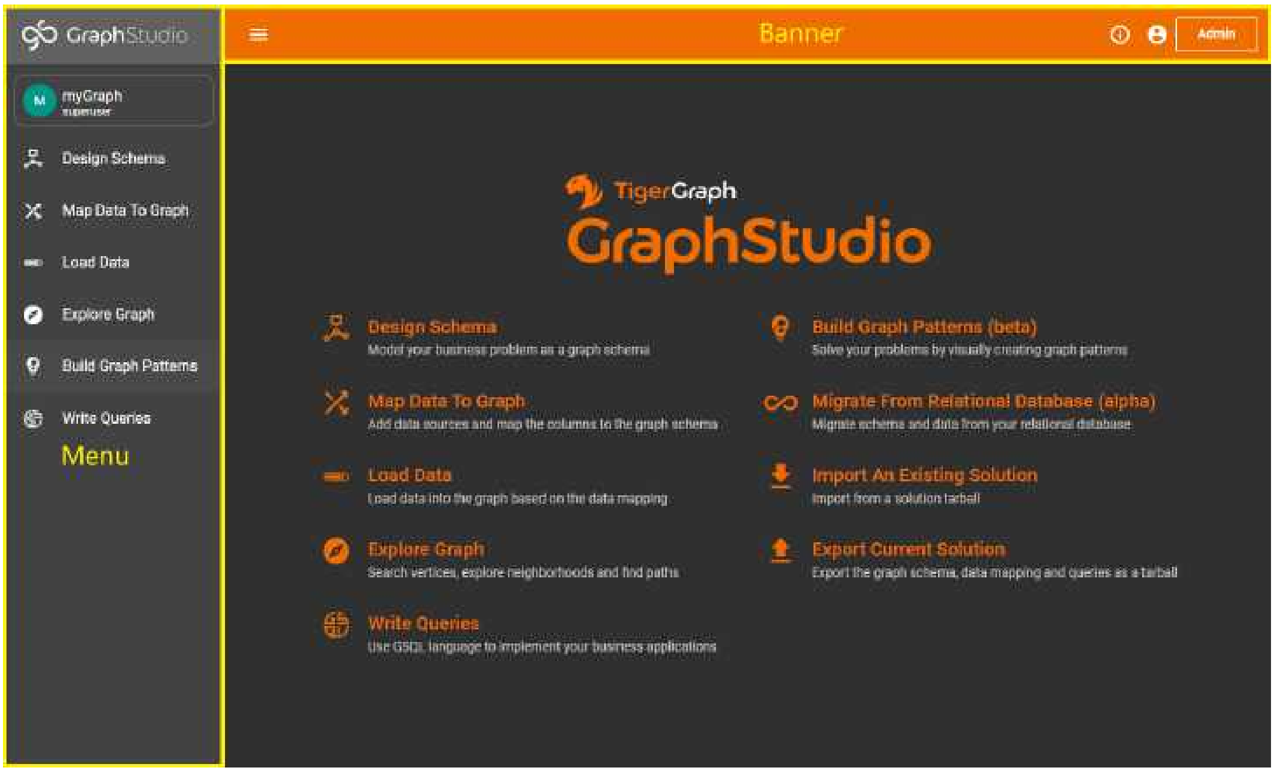
```
$ gadmin status gui
```

If it is off, turn it on:

```
$ gadmin start gui
```

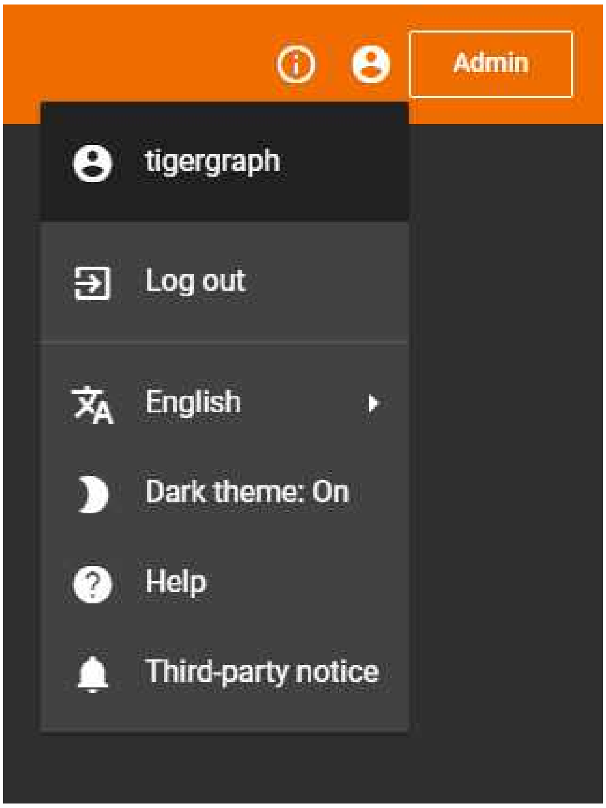
Home Page

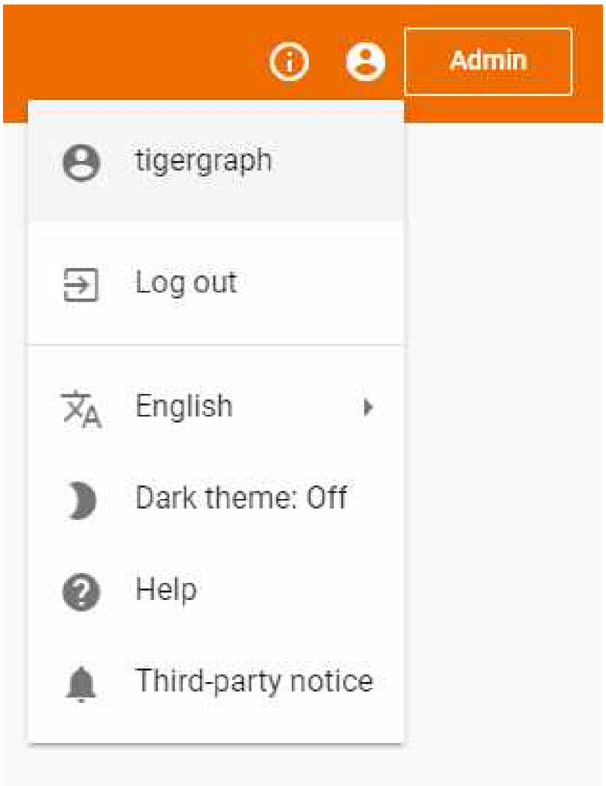
The home page of GraphStudio contains links to each of the six steps of solving a business problem: Design Schema, Map Data To Graph, Load Data, Explore Graph, Build Graph Patterns and Write Queries. Users can also navigate to each step from the buttons in the left menu bar. Each of these major steps has its own page. To hide/show the left menu bar, click the top-left menu button: . Clicking the logo on the top of the left menu will take you back to the home page. You can click to go to the Admin Portal (read more at Admin Portal UI Guide).

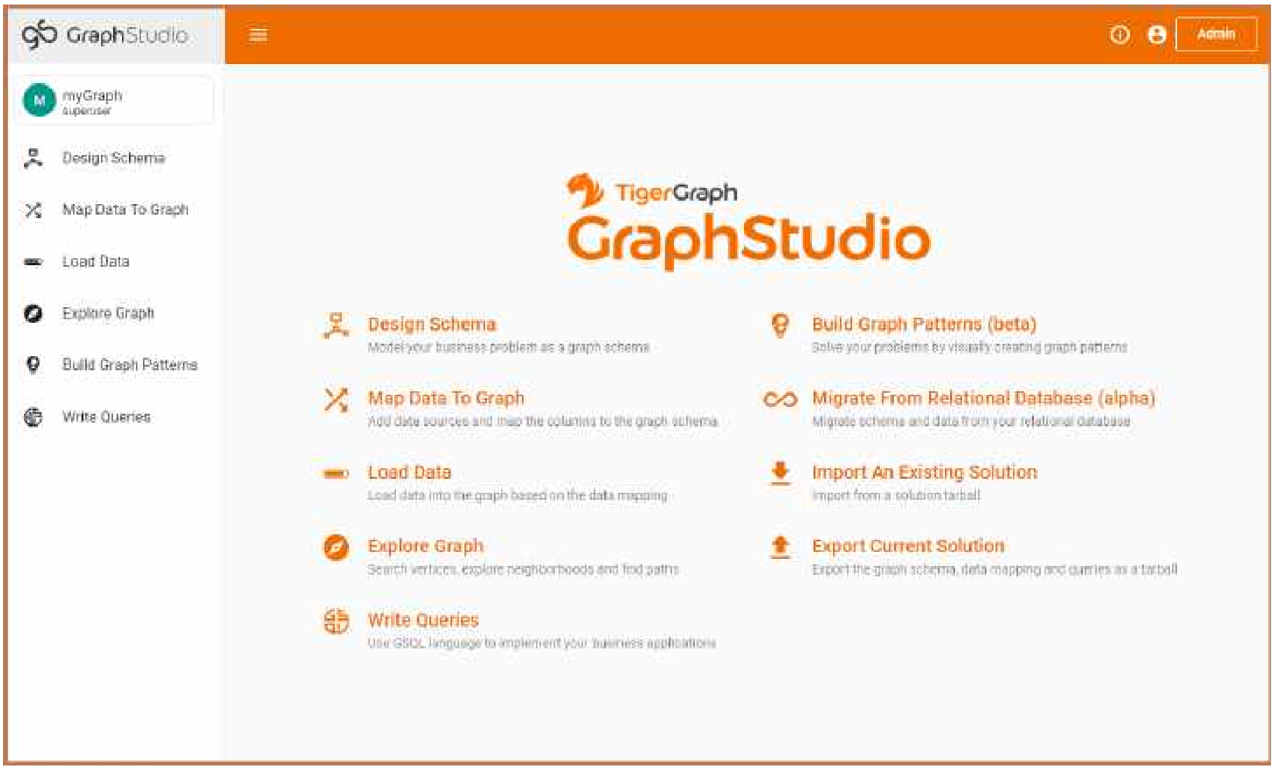


Switch Between Dark And Light Theme

GraphStudio provides two themes: dark theme and light theme. By default it uses dark theme. You can click the User icon and then toggle the Dark theme to be Off to switch to light theme:

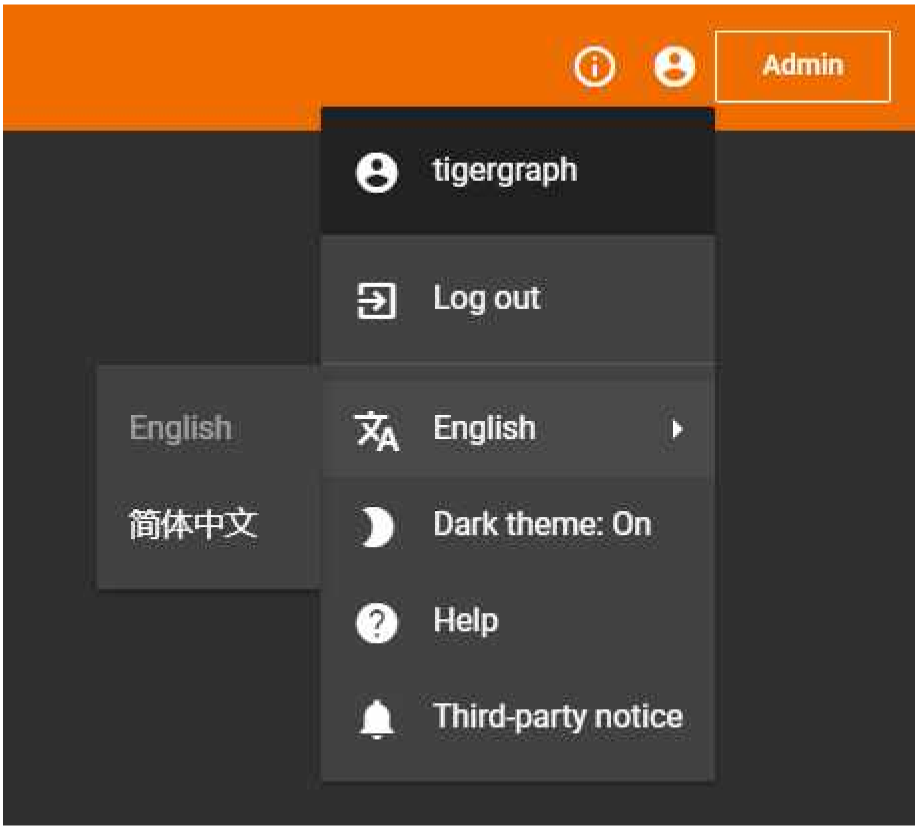




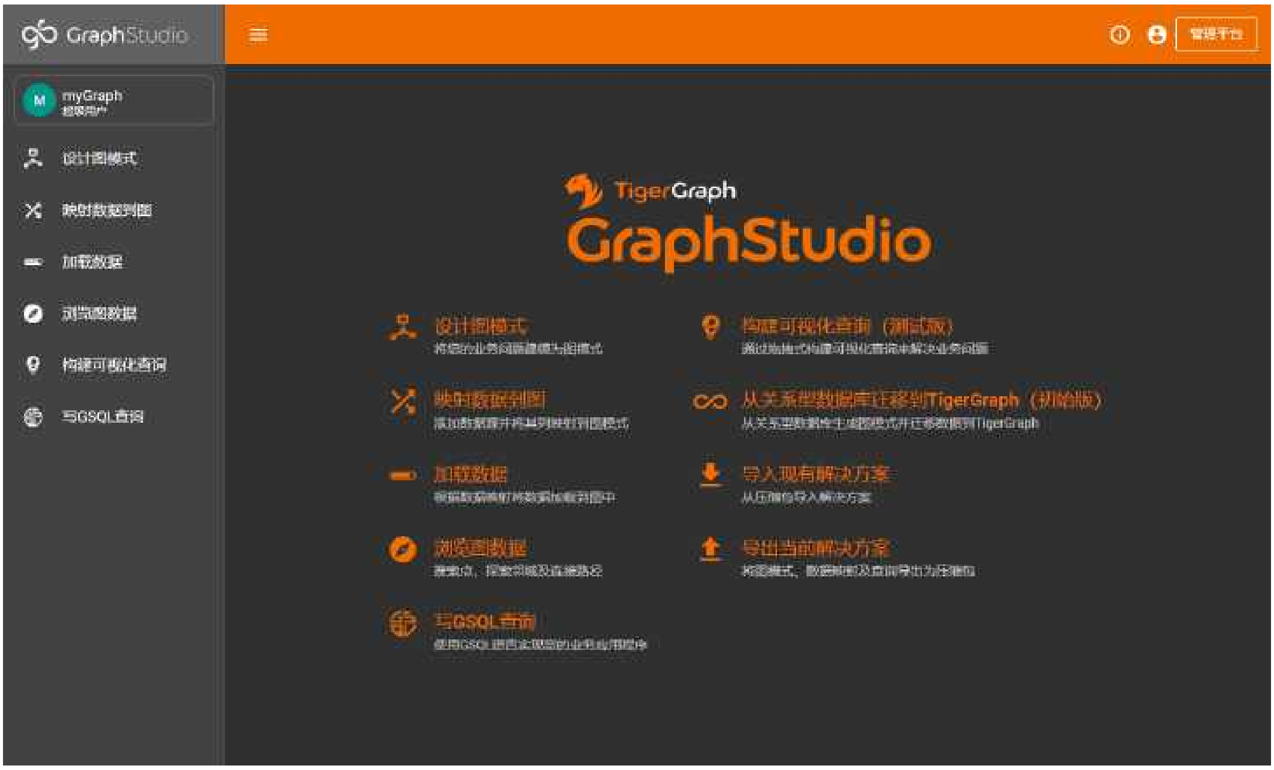


Switch Between Languages

GraphStudio supports 2 languages: English and Chinese. By default it uses English. You can click and then select a language:



This shows what Home page looks like if you choose Chinese.



GraphStudio Session Timeout

GraphStudio has a default session timeout of 1 week. If, during this time, the user has no interaction with the page, the session will expire and the user will be logged out automatically. The timeout can be configured with:

```
$ gadmin config set GUI.ClientIdleTimeSec [timeout]
```

GraphStudio Online Test Drive

Visit TigerGraph Test Drive demos at: https://testdrive.tigergraph.com/ ↗

The GraphStudio online Test Drive features several instances of the TigerGraph system, each one targeting a different use case. Each copy of TigerGraph has a GraphStudio interface and is preloaded with application-specific queries and synthetic data. These demo applications are provided in a read-only mode. Users can explore and play with pre-installed queries. Users on these demo systems cannot save changes to the graph schema, the loading job, or queries. The corresponding buttons are disabled.

GraphStudio Limitations

Some features which are available in GSQL are not available in GraphStudio.

Design Schema

- Fixed binary data types are not supported.
- PRIMARY KEY and composite key are not supported.

Map Data to Graph

- Cannot load JSON data.

Load Data

- Data loading jobs written in a GSQL console are not shown in GraphStudio.

- USING options are not available.
- Concurrent loading is not available.

Write Queries

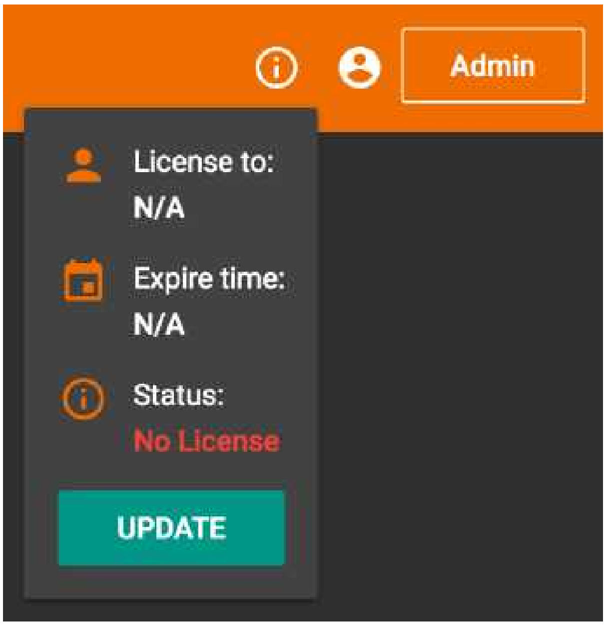
- You cannot define a user-defined function (you can use the user-defined functions created from TigerGraph server by importing the solution which contains pre-defined UDFs into GraphStudio).

License

License

In order to use GraphStudio, you need a valid TigerGraph license that enables GraphStudio access. Please contact sales@tigergraph.com for license related questions.

Click the Information icon , and the current TigerGraph license status will pop up. If a TigerGraph license key has not been added, the license status will look like this:

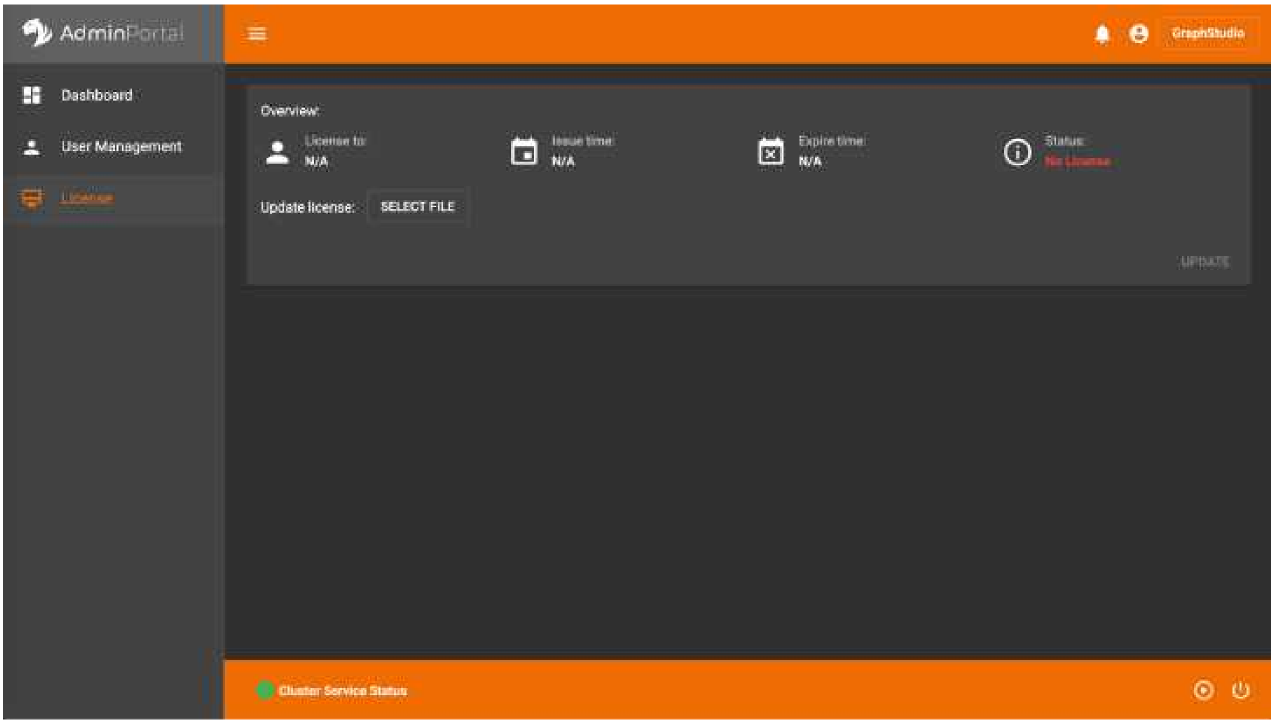


Without a valid license, it is not possible to navigate to the Design Schema, Map Data To Graph, Load Data, Explore Graph, Build Graph Patterns or Write Queries pages.

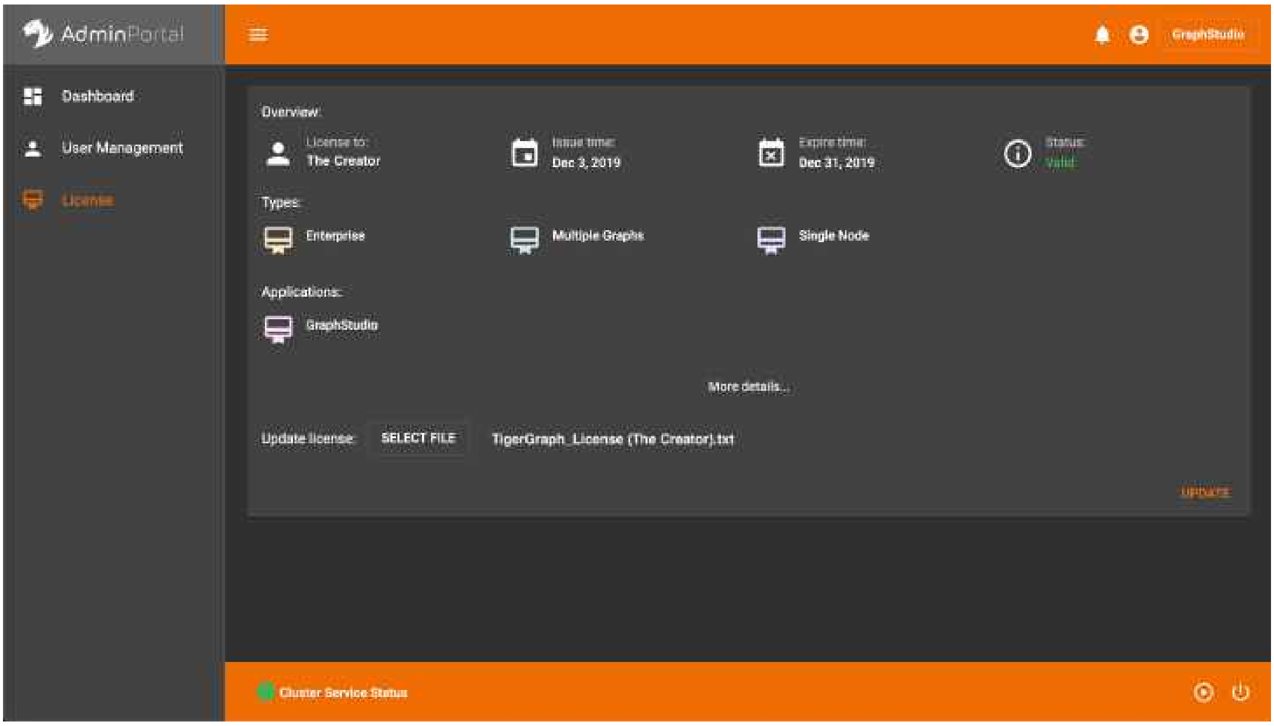
Update a TigerGraph License Key

Click the 

link on the bottom of the license status to go to Admin Portal license page to update a TigerGraph license key:



Click the SELECT FILE button and choose the license text file, then click UPDATE button. The license detail will be loaded like below:

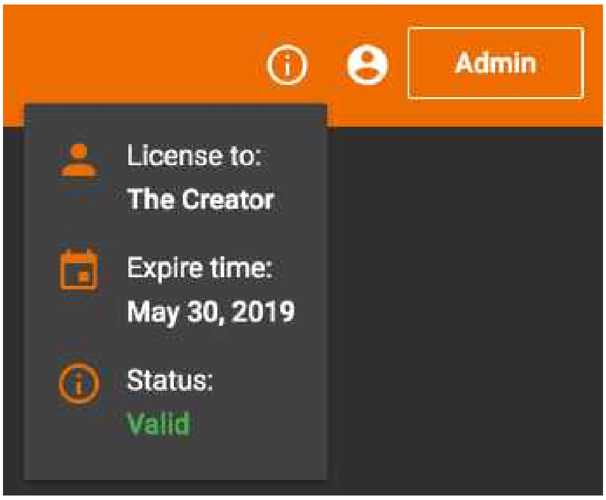


Please note that the GraphStudio item under Applications section needs to be lightened like above. If it looks like this grayed icon: 

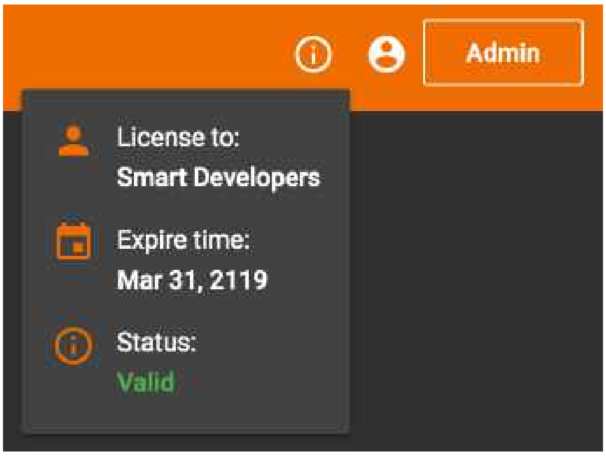
, it means GraphStudio is not enabled in the provided license, and you won't be able to use

GraphStudio in this case. You can upgrade your license to enable GraphStudio access, contact sales@tigergraph.com for more information.

Click at the top-right corner of the page to go back to GraphStudio. If you click the Info icon again, you should see the updated license. Now you can start to use GraphStudio.



The Developer Edition package includes a pre-installed license. Please note that Developer Edition may not be used for production use.



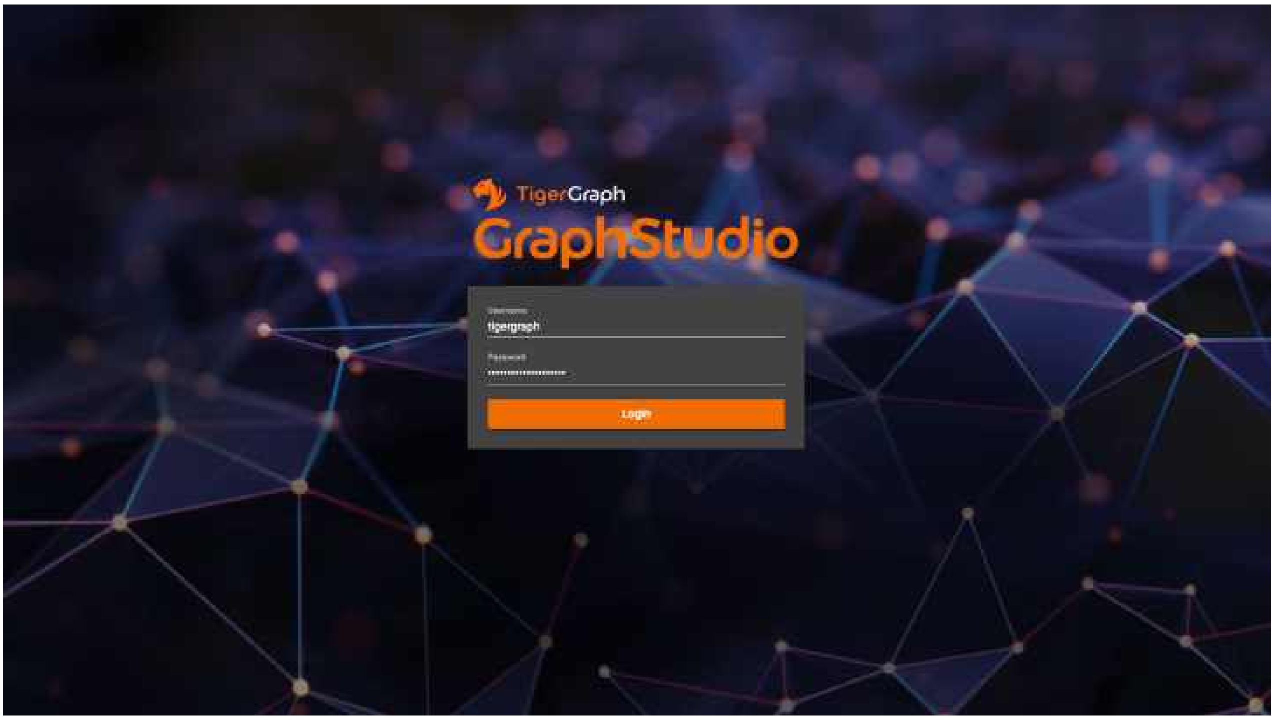
User Access Management

User Access Management

GraphStudio follows TigerGraph user authentication and role-based access control model. Read more in the document Managing User Privileges and Authentication.

Log On

If GSQL *tigergraph* superuser password hasn't been changed (default password is *tigergraph*), then no user login is needed for GraphStudio. Otherwise, users must provide credentials (e.g., username and password) to log in GraphStudio. In addition, your system administrator can integrate TigerGraph with other user access management systems (e.g., LDAP, Active Directory, or SAML-based Single Sign On). See the User Access Management for how to set up LDAP or SSO.

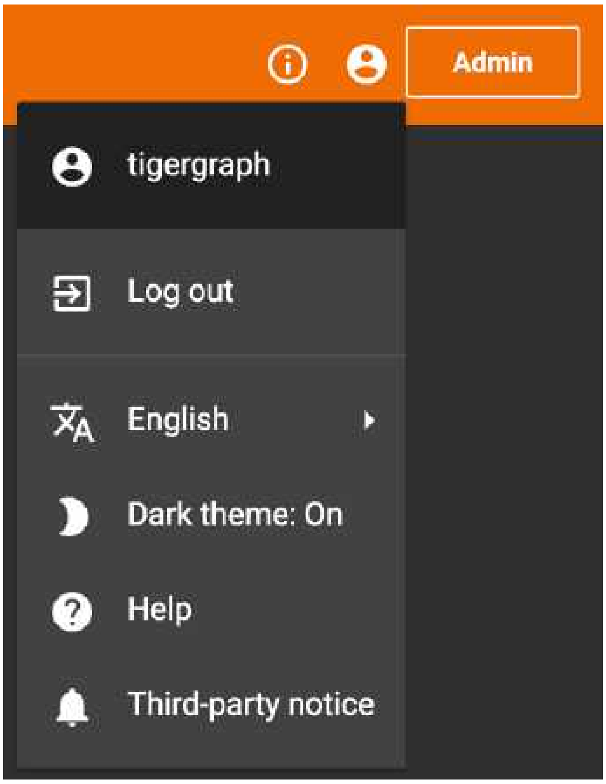


After login, the user is assigned to one of the graphs for which he has access to.

For users other than superuser or globaldesigner, if users do not have access to any graphs, they cannot enter GraphStudio. Hints will show in login page:

After login, the user is assigned to one of the graphs for which he has access to.

To log out, click the User icon  and then the Sign Out icon.



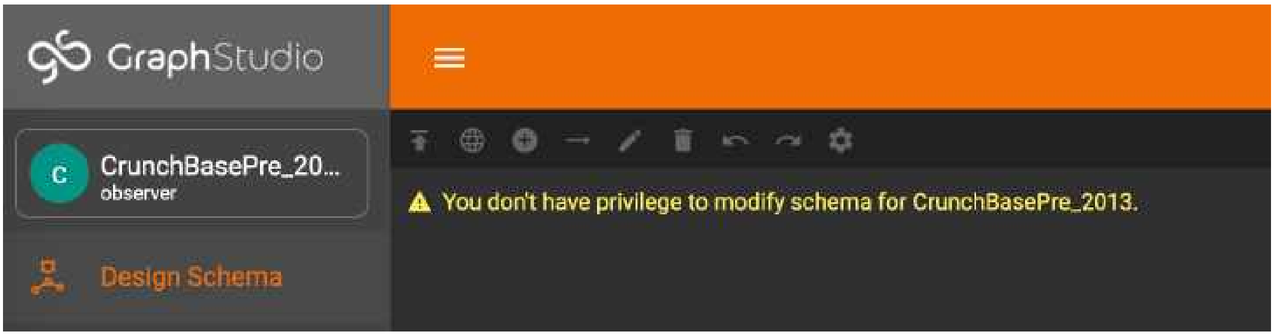
Role- and Graph- Based Access Control

TigerGraph uses role-based access control with several pre-defined roles. Each role is a logical collection of data access privileges, such as querywriter or admin. Each user is assigned one or more roles by a graph admin user or by a superuser. Roles are also graph-specific. For example, user Pat could be an admin on graph G1 but a querywriter on graph G2.

New feature available

In v3.0, Admin Portal supports user management. Read more at Admin Portal UI Guide ↗.

When a user logs in and/or selects a graph, GraphStudio will disable certain actions based on the user's role on that graph. On each working panel, a warning note will alert the user to features which are disabled. For example, in the current version of GraphStudio, users with querywriter, queryreader, or observer role will see the following warnings on the Design Schema working panel:



The table below summarizes the built-in roles and of their key privileges on GraphStudio:

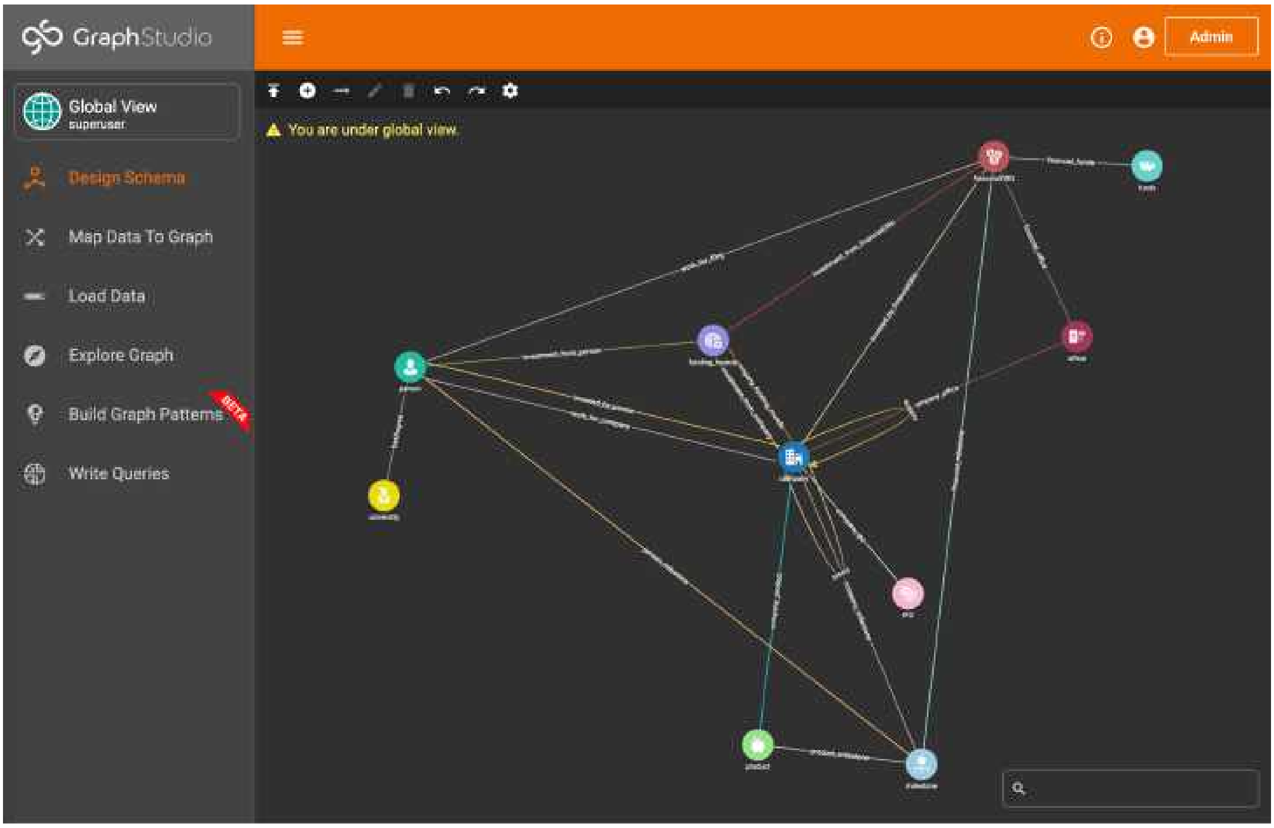
	superuser	globaldesigner	admin	designer	querywriter	que er
Create a new graph schema	YES	YES				
Drop a graph	YES	Can only drop graphs created by herself				
Modify a graph schema	YES	YES	YES	YES		
View a graph schema	YES	YES	YES	YES	YES	YE
Create a data mapping	YES	YES	YES	YES		
View a data mapping	YES	YES	YES	YES	YES	YE
Load data	YES	YES	YES	YES	YES	YE
Explore a graph	YES	YES	YES	YES	YES	YE

Write a query	YES	YES	YES	YES	YES	
Run a	YES	YES	YES	YES	YES	YES

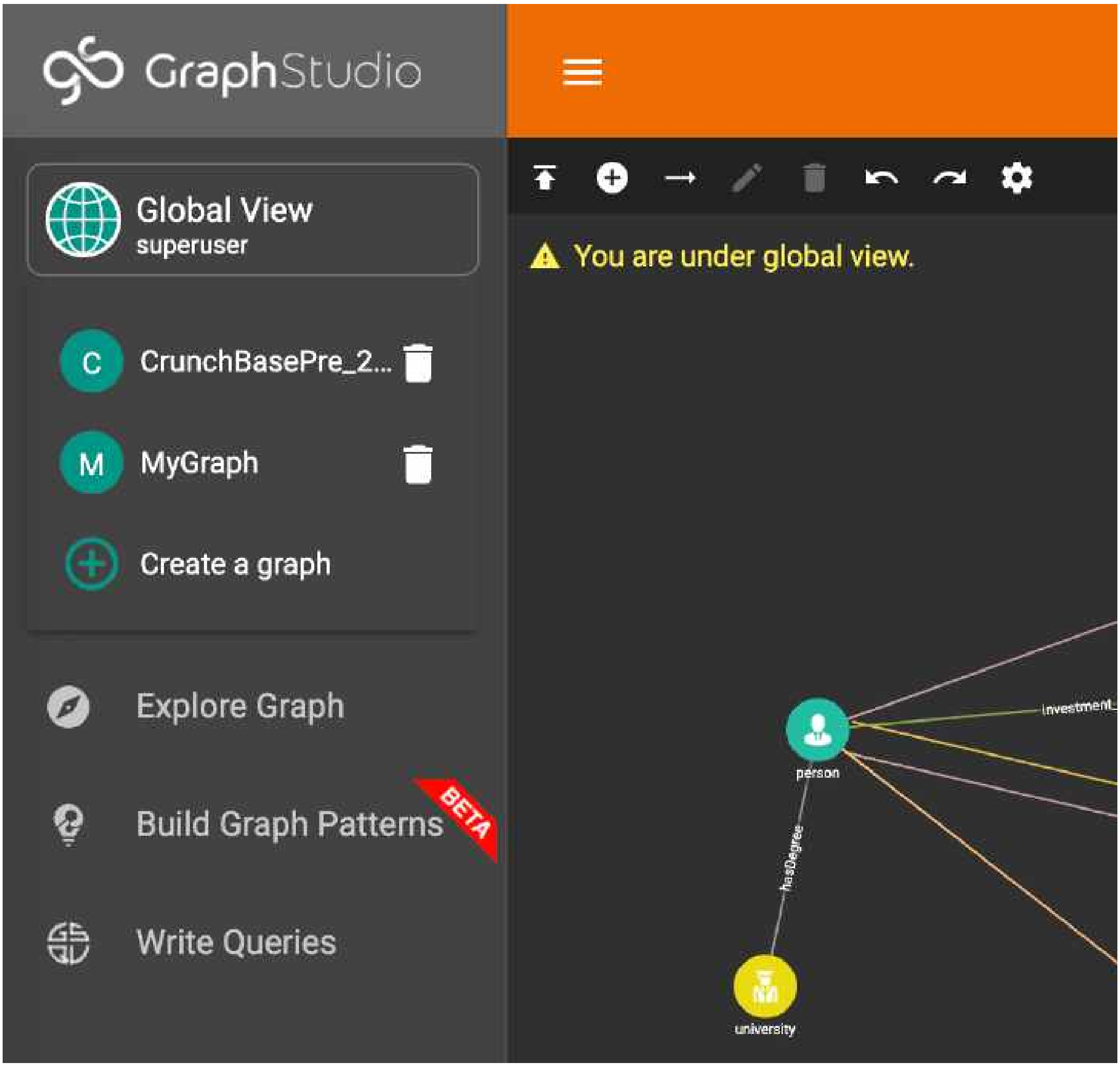
Global View and Select A Graph

Beginning with Version 1.2, the TigerGraph system can support multiple graphs within one TigerGraph instance. Read more at [MultiGraph - An Overview](#).

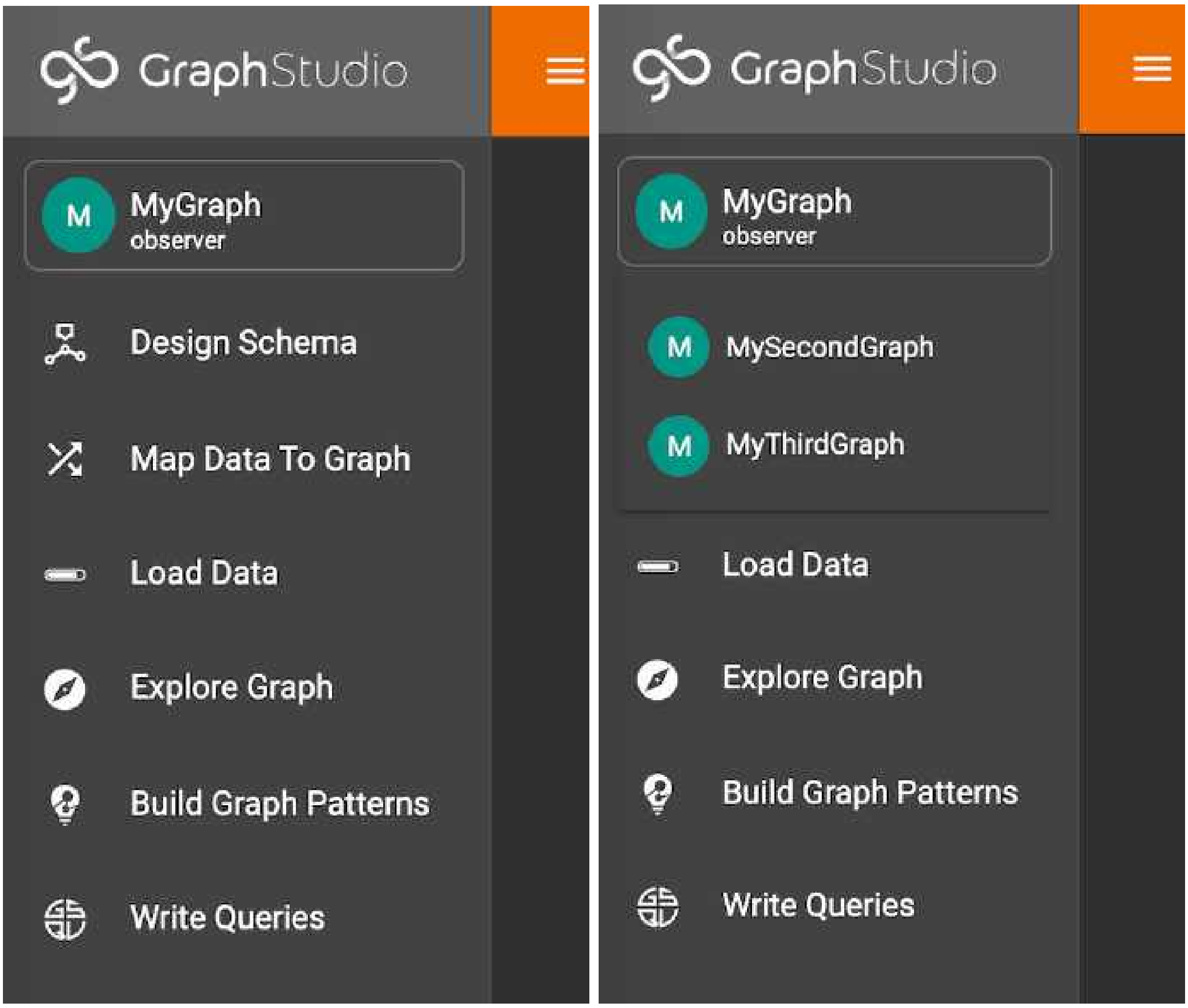
For superuser and globaldesigner users, by default they will be in global view.



Click the card showing current graph and user roles, and a dropdown menu of graphs the current user can access will appear. In addition, superuser or globaldesigner users can create and drop graphs.



For other users, there is no access to global view, and you cannot create or drop graphs.



Click the graph name to switch to another graph.

A graph superuser grants each user access to particular graphs in User Management page at Admin Portal. In addition, users with either superuser and admin role can grant other users access to particular graphs using GSQL commands.

Design Schema

Design Schema

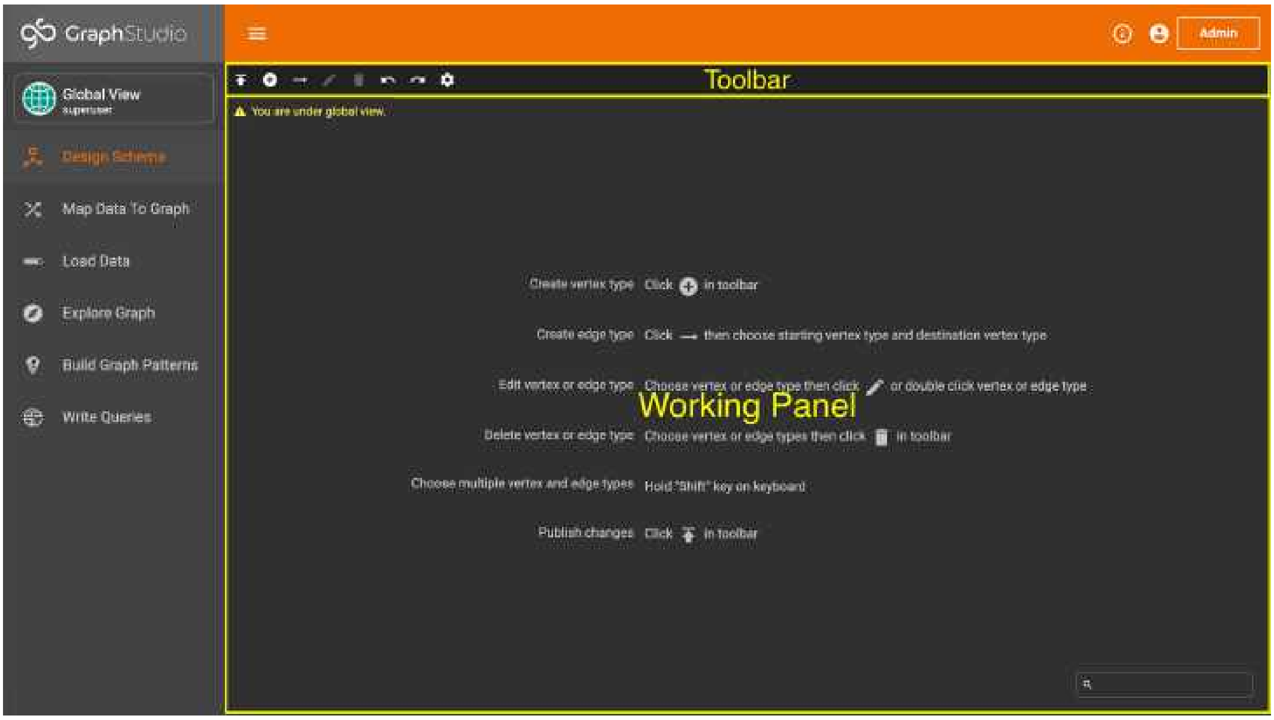
Beginning with Version 3.0, GraphStudio can support modifying multiple graphs within one TigerGraph instance. Read more about multigraph at MultiGraph - An Overview.

Designing the graph schema is the first and most important step of solving a business problem. The graph schema is the model of the problem, and all of the subsequent steps depend on the graph schema. If you are not on the Design Schema page yet, click "Design Schema" on the left side menu bar.

> Only certain roles have the privilege to modify a graph schema.
>
> 1. Only the superuser and globaldesigner roles can modify the global graph schema, including creating local graphs.
> 2. Only the superuser, globaldesigner, and designer roles can modify a local graph schema.

When there is no graph schema in the system, this page will show some hints:

1. Only the superuser and globaldesigner roles can modify the global graph schema, including creating local graphs.
2. Only the superuser, globaldesigner, and designer roles modify a local graph schema.



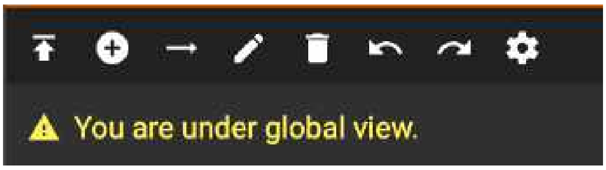
Otherwise this page will visualize the schema:

Each circle represents a vertex type, and each link represents an edge type. You can drag the circles to change their positions. There are two ways to zoom in and out. If you have a touchpad, two-finger moving up zooms in; two-finger moving down zooms out. Similarly, if your mouse has a scroll wheel, spinning forward zooms in, and spinning backward zooms out.

Note: The relationship between a vertex type and a vertex instance of a graph is like the relationship between a table and one record of a table in the relational database world. The relationship between an edge type and an edge instance is similar. In the Design Schema step, the user defines vertex types and edge types to model the data schema. After the schema has been created, the next two steps, Map Data To Graph and Load Data, are for loading data into the graph.

Global View vs. Graph View

For superuser and globaldesigner, they can modify global vertex and edge types under Global View. Under global view, the toolbar area only contains "Publish schema", "Add a vertex type", "Add an edge type", "Edit", "Delete", "Undo" and "Redo" buttons. There will be a warning message in the working panel saying user is under global view.

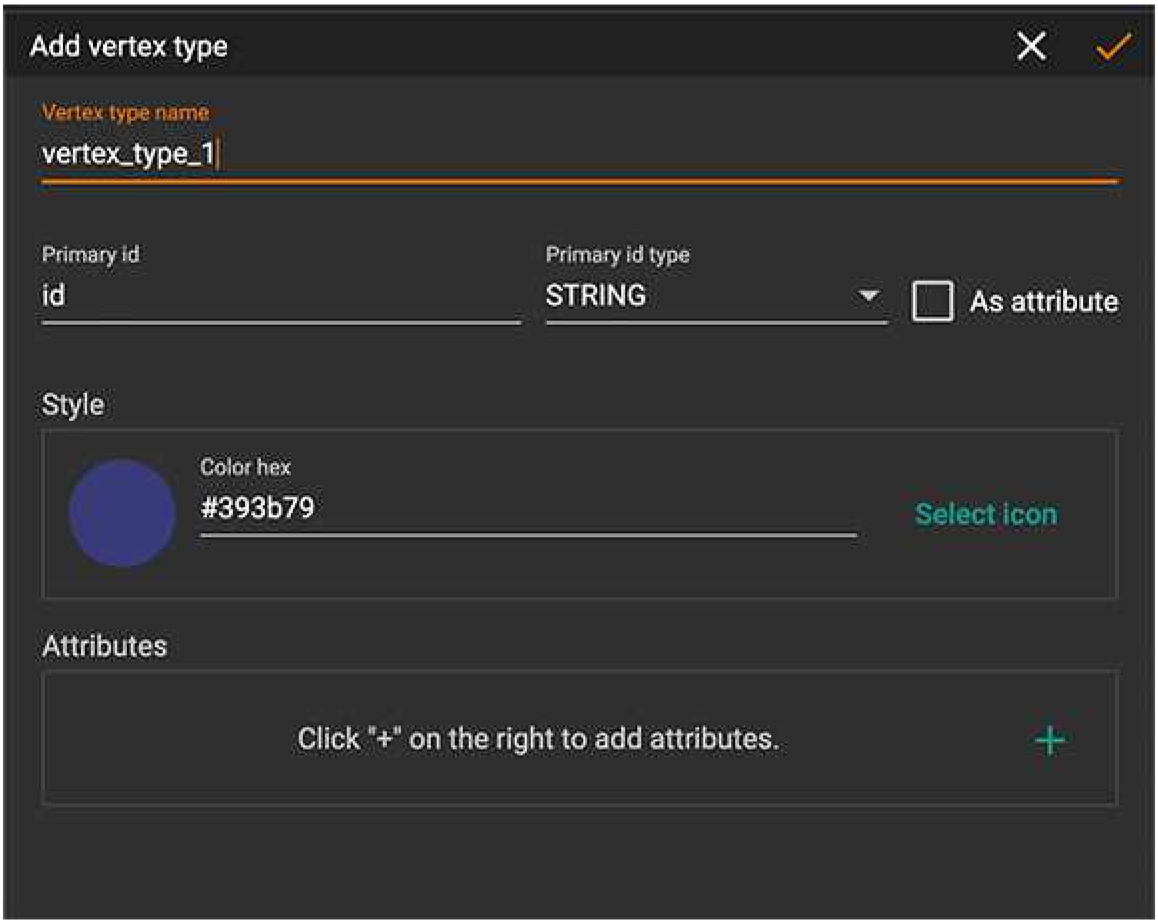


1. If under global view, other pages except Home and Design Schema will be disabled.
2. Users other than superuser and globaldesigner cannot access global view.

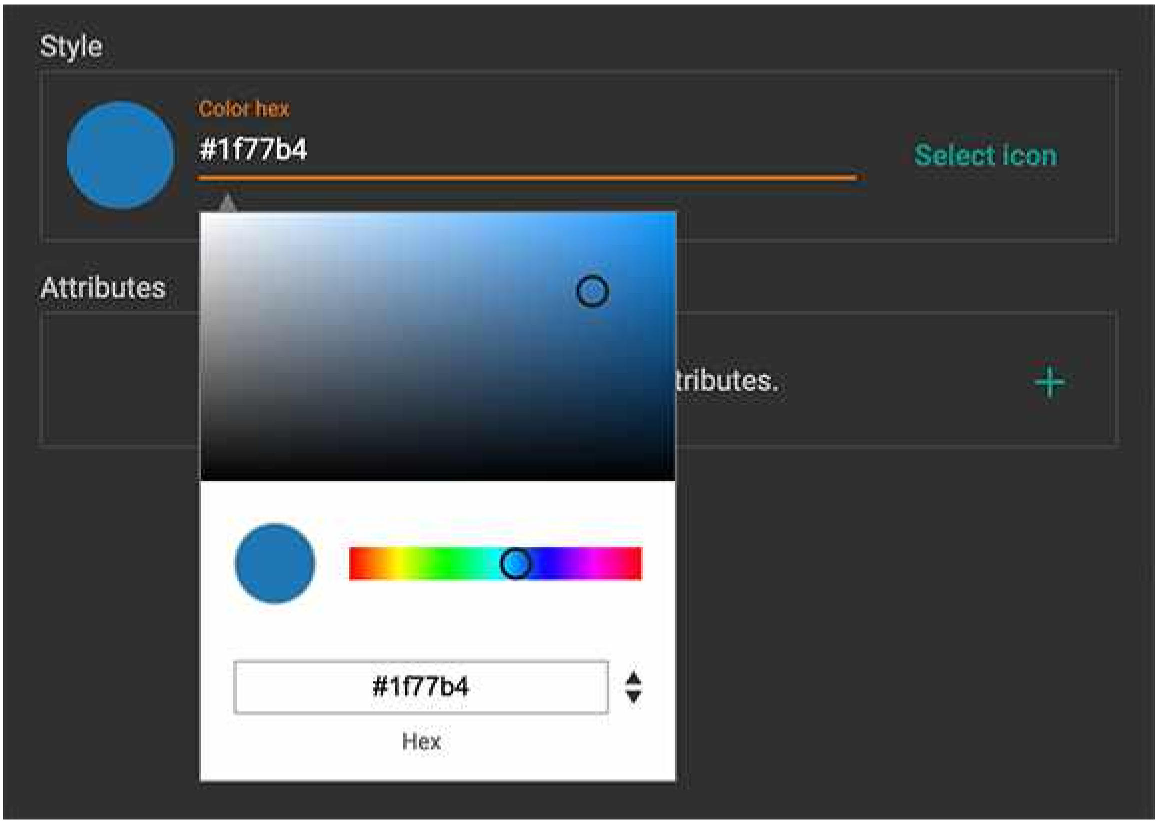
Users can modify a graph schema under graph view. In a graph, there is one more in the toolbar area - "View global vertex and edge types". And for global vertex and edge types used in a graph, a little global icon is attached to them.

Add A Vertex Type

Click the add vertex type button  to add a vertex type. The working space will enter Add Vertex mode and the button color will change to green . Click the button again to exit Add Vertex mode. The add vertex type panel in the right:



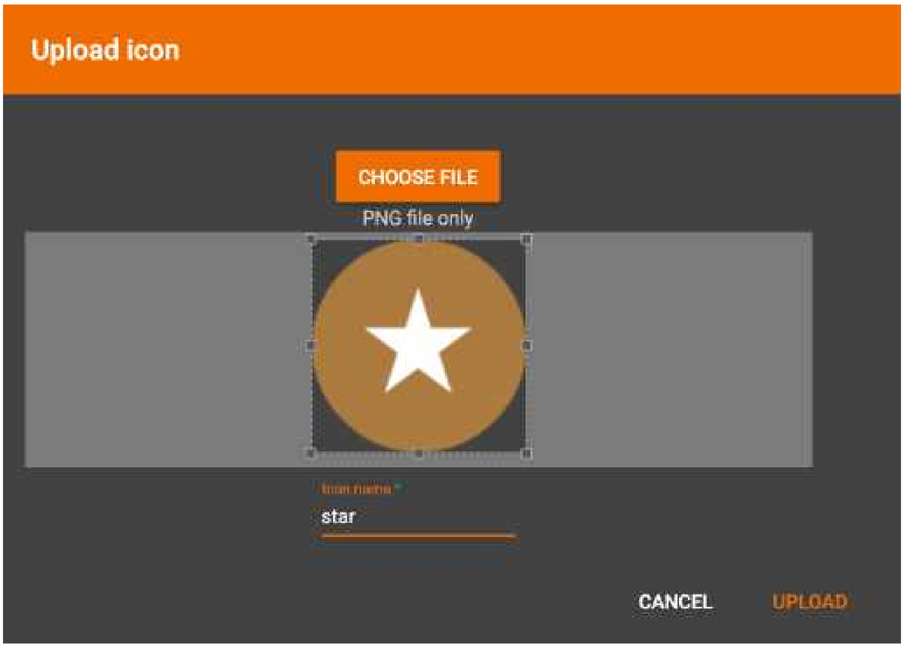
In this window you specify a vertex type name, primary id name. GraphStudio will automatically select a color for your vertex type icon. You can change the vertex type color by clicking the value under the "Color hex" label. A color palette window will pop up allowing you to choose a new color:



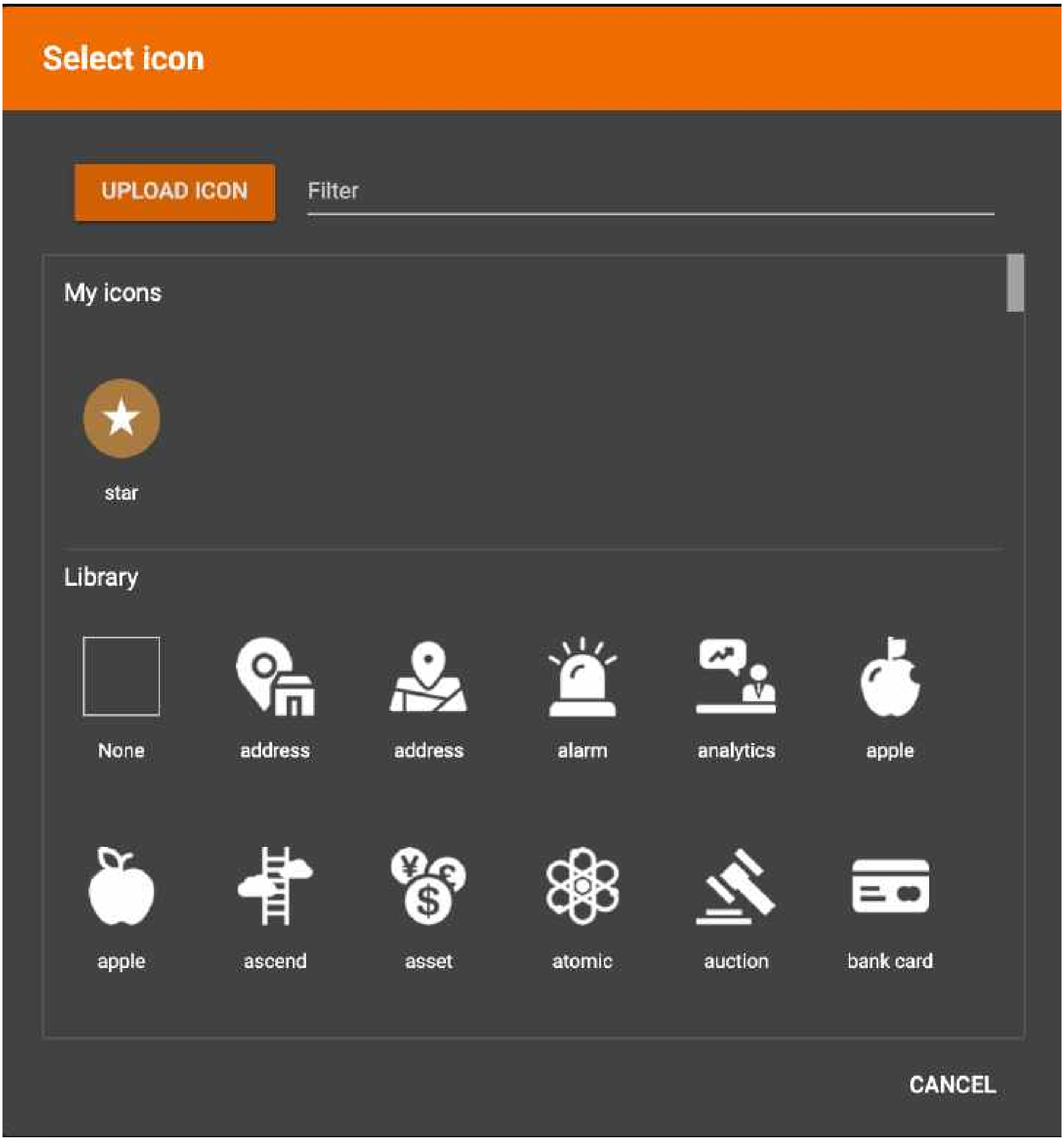
Once you are satisfied with the color, click anywhere outside of the color palette window to set the color.
You can also choose an icon for the vertex type by clicking the Select Icon button 

. Then a Select Icon window will pop up. Select an icon that fits the vertex type semantic best. You can type in keywords to help filter the icons and find the best match faster.

You can also upload your own icons by clicking the Upload Icon button, choosing a PNG image, giving a name and click Upload:

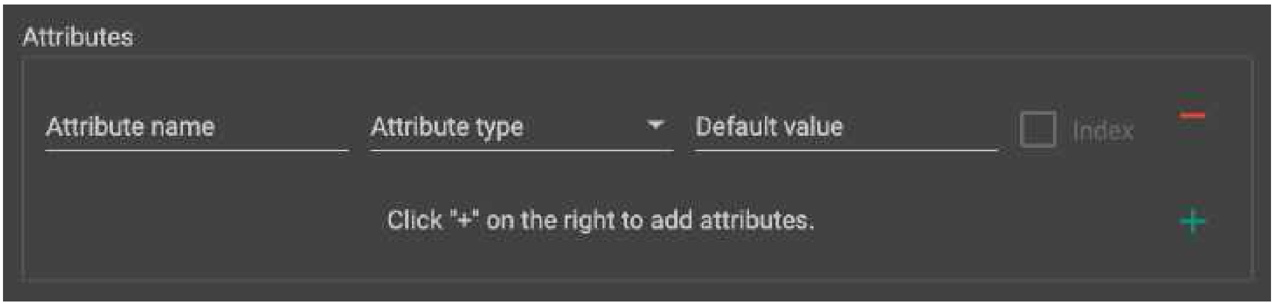


Then you can use your uploaded icons:



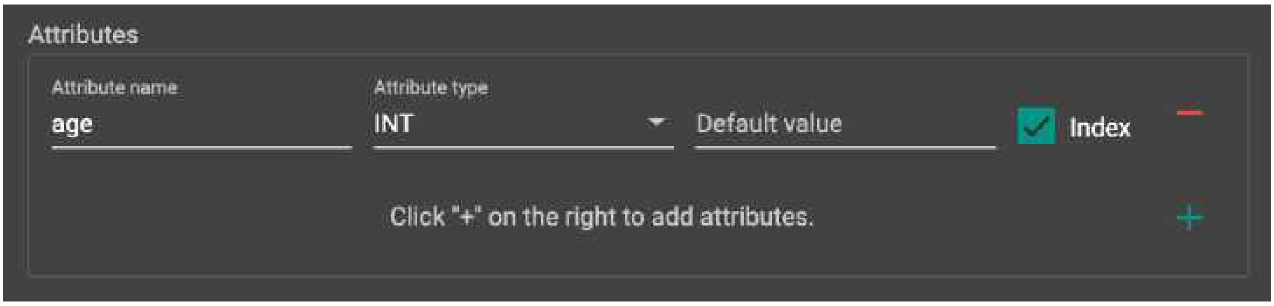
Adding and Deleting Attributes

To add an attribute, click the green plus sign at the right of the Attributes section:



Provide a name and data type for your new attribute. Optionally, you can specify a default value for the attribute. (If you do not specify, every data type has a system default value. For example, the default value for an integer is 0.)

For some types of attributes (INT, UINT, STRING, STRING COMPRESS, DATETIME), you can add an index, which will improve query performance when accessing these attributes.



Click the red minus sign to the right of the attribute to delete an existing attribute.

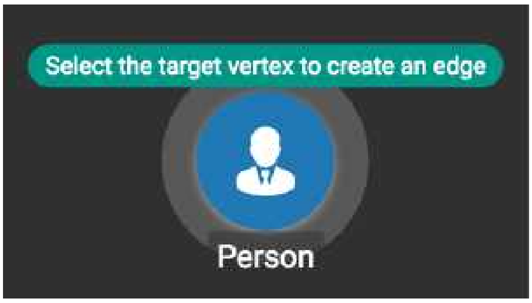
Once you are satisfied with the vertex type settings, click the Add button to add the vertex type. A new circle will appear in the working panel. You can drag the circle to any desired position.

Note: PRIMARY KEY and COMPOSITE KEY is not supported in GraphStudio. If you decide to use these features, you will only be able to use command line interface.

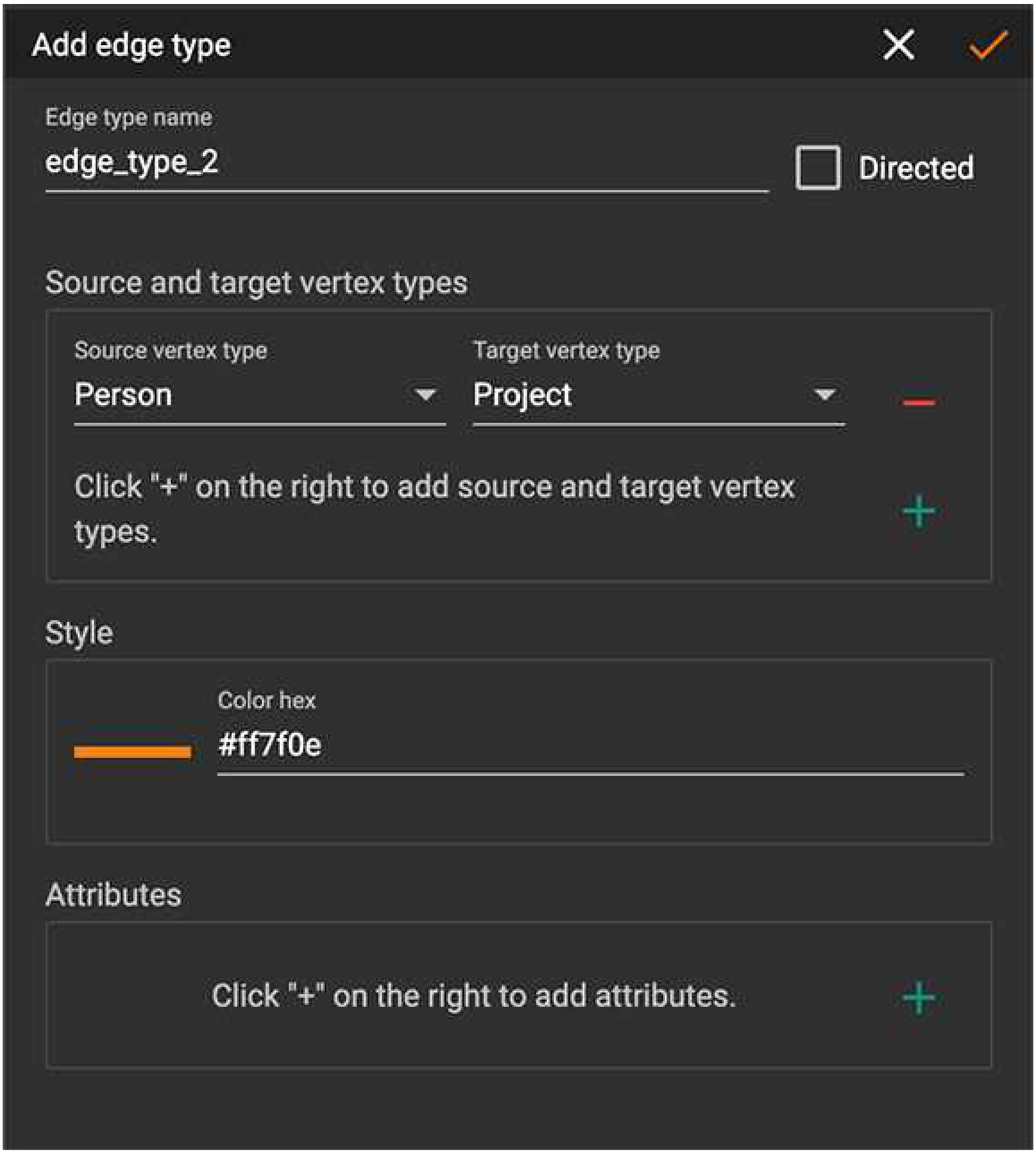
Add An Edge Type

Click the add edge type button to add an edge type. The working space will enter Add Edge mode and the button color will change to green. Click the button again to exit Add Edge mode.

Each edge type has one or many source vertex types and target vertex types. First, click the source vertex type. A hint will appear on the vertex type circle:

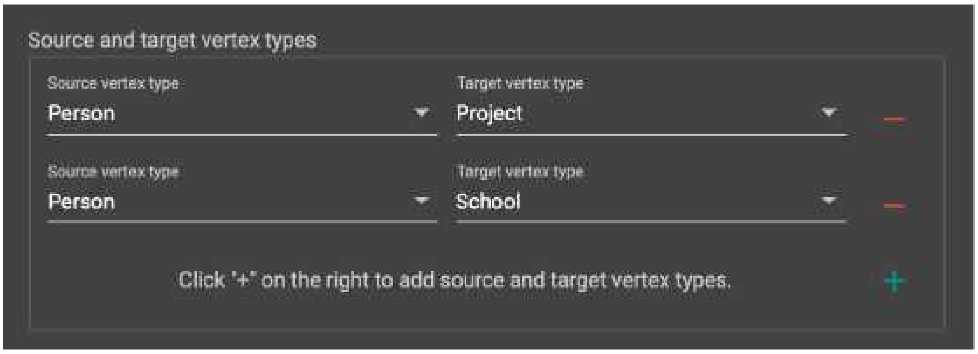


Then click the target vertex type. The add edge type panel will be in the right:

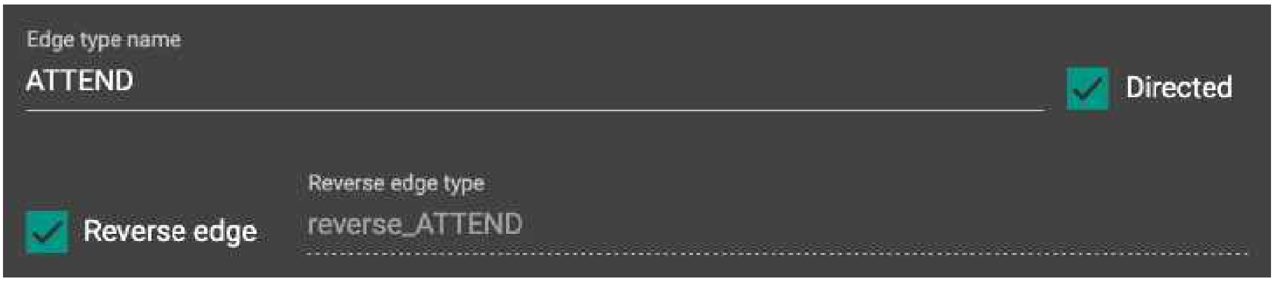


You must specify an edge type name. The source vertex type and target vertex type are selected based on your clicking action. However, you can change that by choosing another vertex type in the dropdown list.

You can also click the green plus sign at the right of the source and target vertex types section to add more source and target vertex types of the edge type.



By default, the edge type is undirected. To make the edge type directed, mark the Directed checkbox:



If Directed is checked, another checkbox will appear for you to choose whether the edge type should include reverse edges. Including reverse edges provides more flexibility when designing queries. Unselect the reverse edge checkbox ONLY IF your machine memory is very tight, because if there is no reverse edge, queries will not be able to traverse backwards along this directed edge type, from the target vertex to the source vertex.

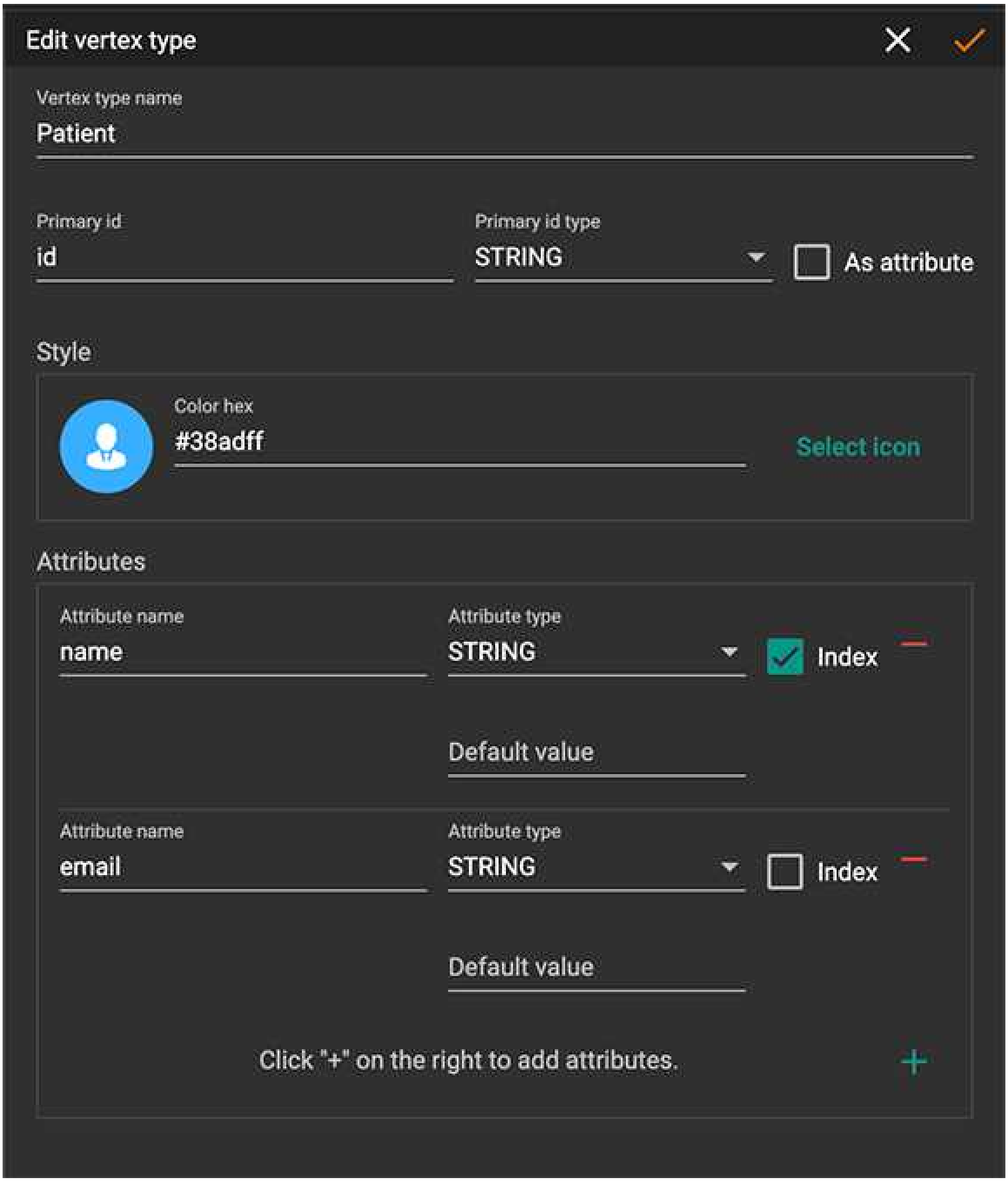
Editing edge type attributes is the same as editing vertex type attributes.

Once you are satisfied with the edge type settings, click the Add button  to add the edge type. A new link between the selected source vertex type circle(s) and target vertex type circle(s) will appear in the working panel.

You can add multiple edge types between the same source vertex type and target vertex type pair. Moreover, an edge can use the same vertex type for both its source vertex type and its target vertex type, e.g., a Friendship edge from Person vertex to Person vertex.

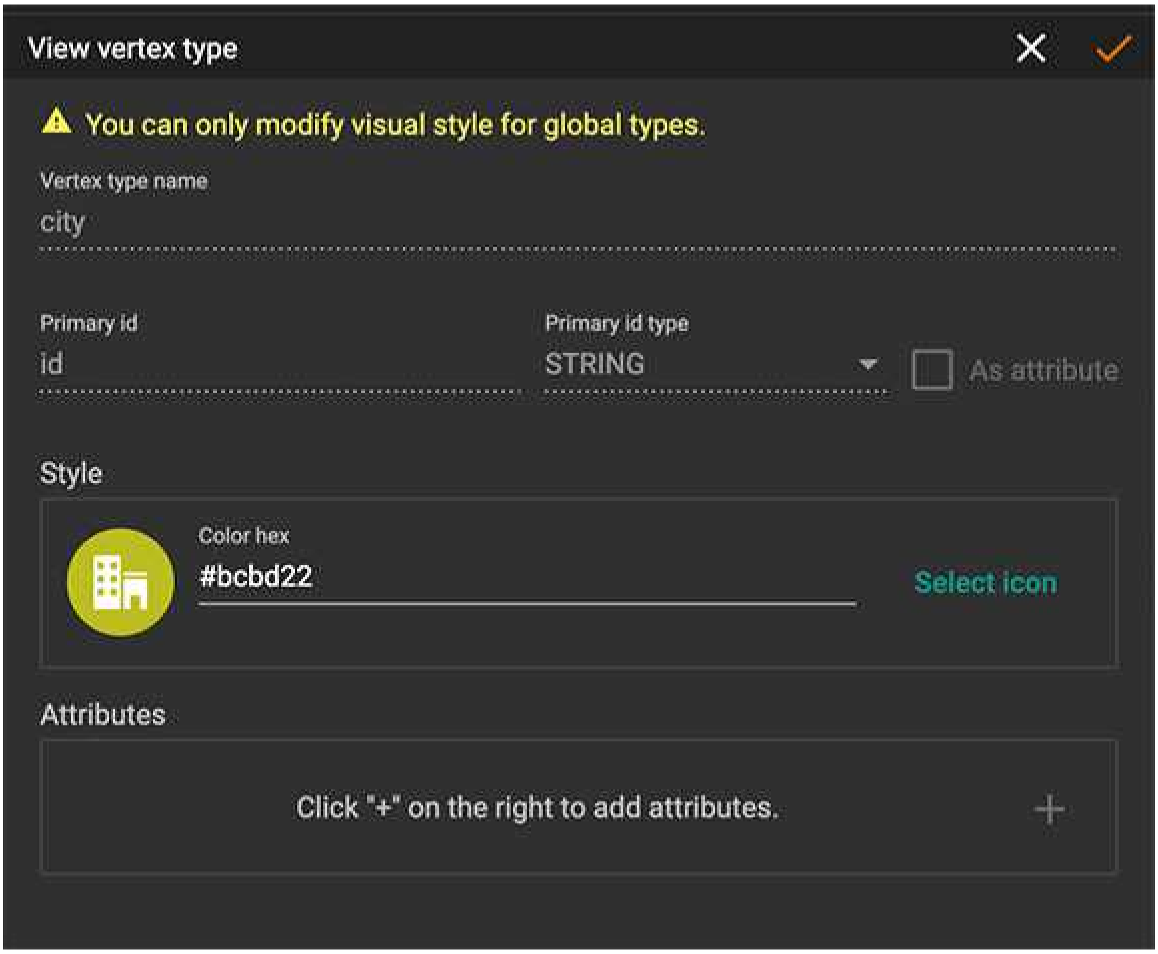
Edit Vertex Or Edge Type

You can edit the vertex types or edge types at any time after you add them. Just click one vertex type circle or one edge link, and then click the edit button  (double clicking on the selected vertex/edge will have the same effect). The working space will enter Edit mode and the button color will change to green . Click the button again to exit Edit mode. The Edit Attributes panel in the right:



Once you are satisfied with the change, click the Update button .

In graph mode, you can only edit the style of a global vertex or edge type:



Delete Vertex Or Edge Type

You can delete a vertex type or an edge type by first choosing the vertex type circles or edge type links, then clicking the delete button . In order to delete multiple vertex types and edge types, hold down the "Shift" key while you select multiple items.

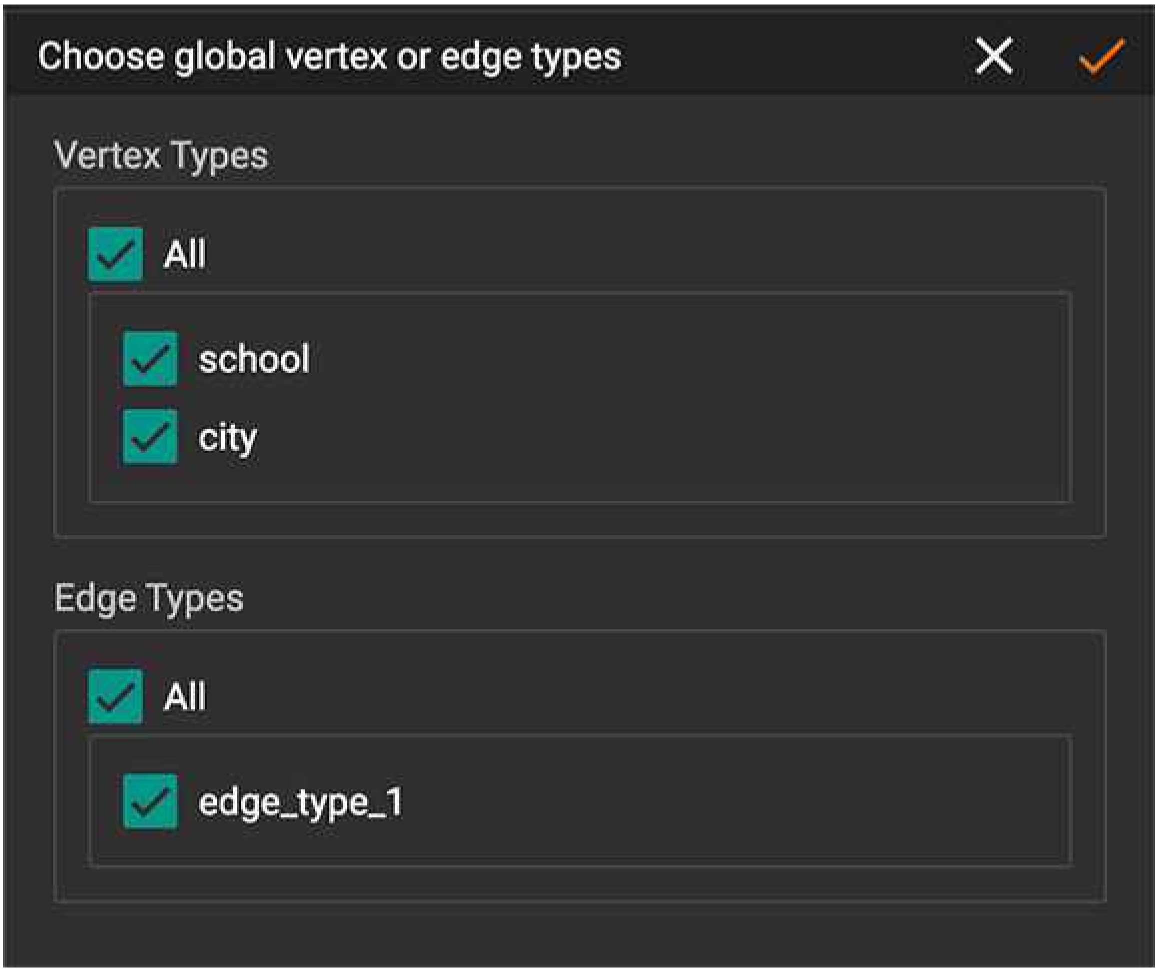
> Note that user cannot delete a global vertex or edge type using delete button in a graph.

Redo And Undo

You can redo and undo your changes by clicking the two buttons: . The whole history since the time you entered Design Schema page is recorded.

View Global Vertex And Edge Types

Click the view global vertex and edge types button to assign global vertex and edge types to a graph, or drop them from a graph. The working space will enter View Global Vertex and Edge Types mode and the button color will change to green . Click the button again to exit View Global Vertex and Edge Types mode. The add vertex type panel in the right:



> Only superuser or globaldesigner can modify global types in a graph. The view global vertex and edge types button will be disabled for other users.

Publish Schema

Once you are satisfied with the graph schema, click the publish schema button to publish the schema to the TigerGraph system. If you are publishing a brand new schema, a progress bar will show:

Note that **Publish Schema** applies to both creating a new schema as well as modifying an existing schema. If you have already loaded data into or created queries for an existing graph, please note that GraphStudio's Publish Schema is only able to retain your existing data in some circumstances. Read the following section carefully.

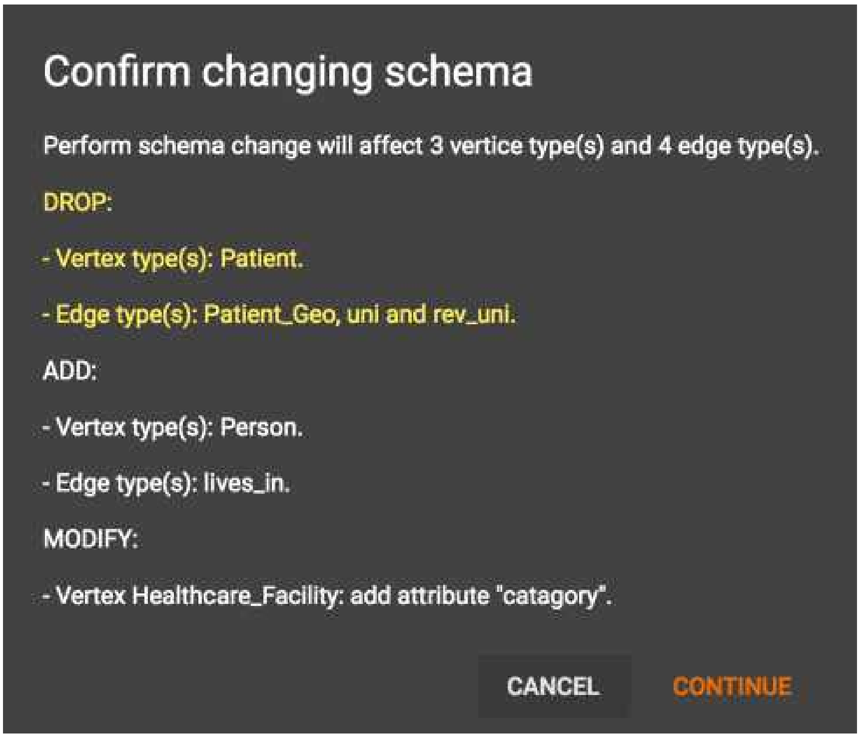
If you are editing an existing graph schema, GraphStudio will analyze your changes. If the change to a vertex or edge type is to remove some attributes and / or to add some new attributes, or add or remove index to some attributes, GraphStudio will employ a GSQL SCHEMA_CHANGE job in order to retain the graph data you already loaded.

All other types of changes, including **renaming** the vertex or edge type, changing **attribute name or data type**, changing **edge direction**, adding or removing **reverse edge** will result in removing the old vertex or edge type and then adding the new one with your desired configurations. In that case, the loaded data to that vertex or edge type will be erased. Please think twice before you do this type of changes.

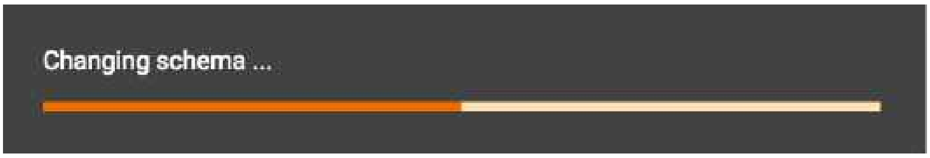
If a vertex type will be removed in order to change the schema, all edge types connected to that vertex type will also be removed.

When you are editing a graph schema, a warning message in the top-left side of the working panel will show which old vertex and edge types will be removed. Make sure to check the message periodically to make sure it is as you expect:

Finally, when you click publish schema button, a pop up window will summarize your changes to the schema. The vertex and edge types that will be removed are highlighted. Make sure you confirm the changes before continue:



Click continue button, and GraphStudio will start changing your schema:

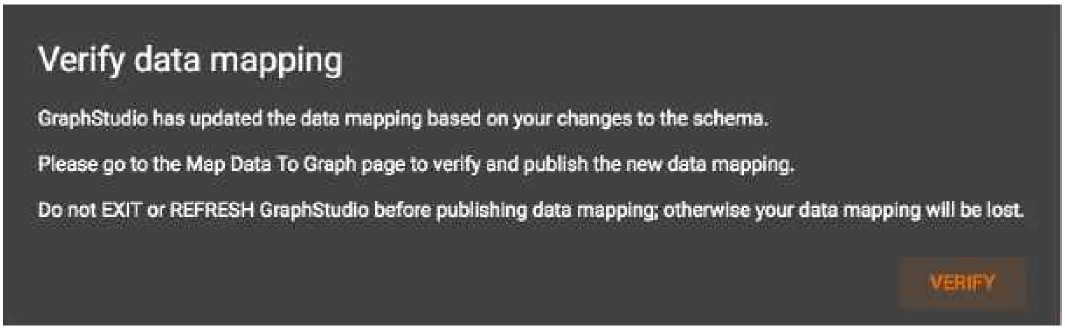


If you have already created a data mapping and written queries, GraphStudio will try its best to preserve your work when you publish your modified schema:

1. All your queries will be saved as query drafts, so you can install the queries again after you change your schema. If a query has a conflict with the new schema (e.g., referring to a vertex type that is deleted), you need to fix it before installing the query.
2. GraphStudio will migrate your data mapping based on your changes to the schema. Since GraphStudio records your whole operation history, the migration is smart enough to cover most cases. The basic migration rules are the following:
    a. Rename vertex types and edge types

b. Remove mappings to deleted vertex types and edge types.
c. Remove mappings to deleted or modified attributes.
d. New vertex types, edge types and new attributes won't be mapped.
e. After the schema is successfully published, GraphStudio will instruct you to go to the Map Data To Graph page to verify and publish the revised data mapping. If any mapping is not correct, you can fix it. You **must publish** the migrated data mapping; otherwise, it will be lost.

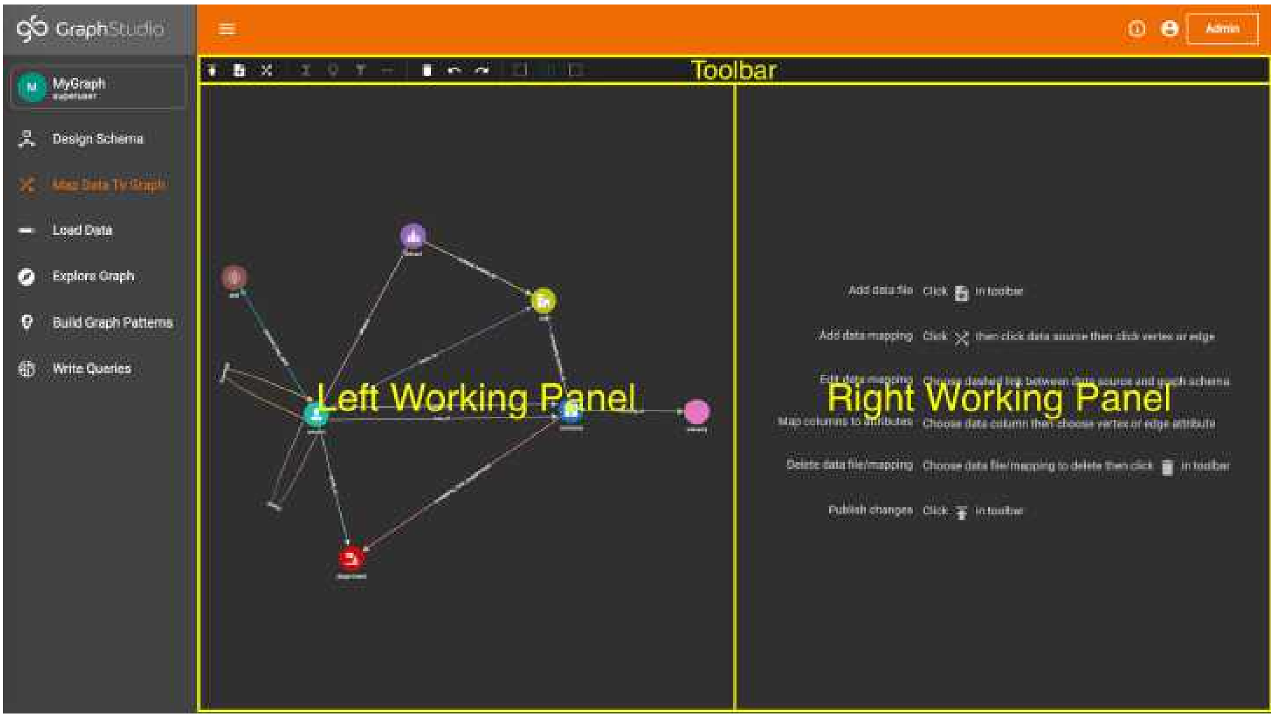
If you have published some data mapping through GraphStudio, then after schema is changed successfully, a pop up window will guide you to go to the Map Data To Graph page to confirm and publish the migrated data mapping:



Map Data To Graph

Map Data To Graph

After you have created a graph schema, the next major step is to map your data to the schema. Click "Map Data To Graph" on the left side menu bar. The working panel is split into a left panel and a right panel. Initially when there is no data mapping yet, the left panel will display only the graph schema.



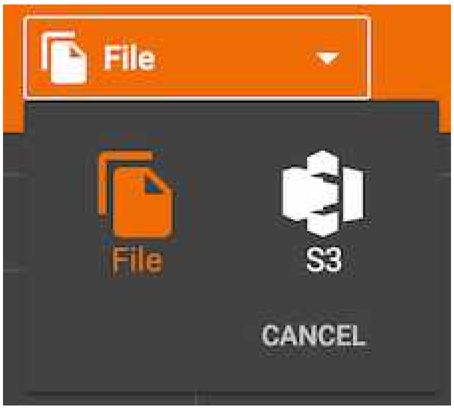
The main steps are

1. Select a data source.
2. Add data file(s)
3. Map data file(s) to vertex/edge types
4. Map data file columns to vertex/edge fields
5. Publish data mapping

1 Select a Data Source

Beginning with v2.4, GraphStudio supports loading data from either local files on the TigerGraph server or files stored on Amazon S3. In future releases, GraphStudio will support loading from other data sources.

Click the data file type selector button on the banner of Add Data File window, and choose either File or S3 from the list:



- If you select **File**, no more configuration is needed. Skip the sections for external sources and go to [Map Data To Graph](#).
- If you select **S3**, then read the section [Create S3 Data Source](#).

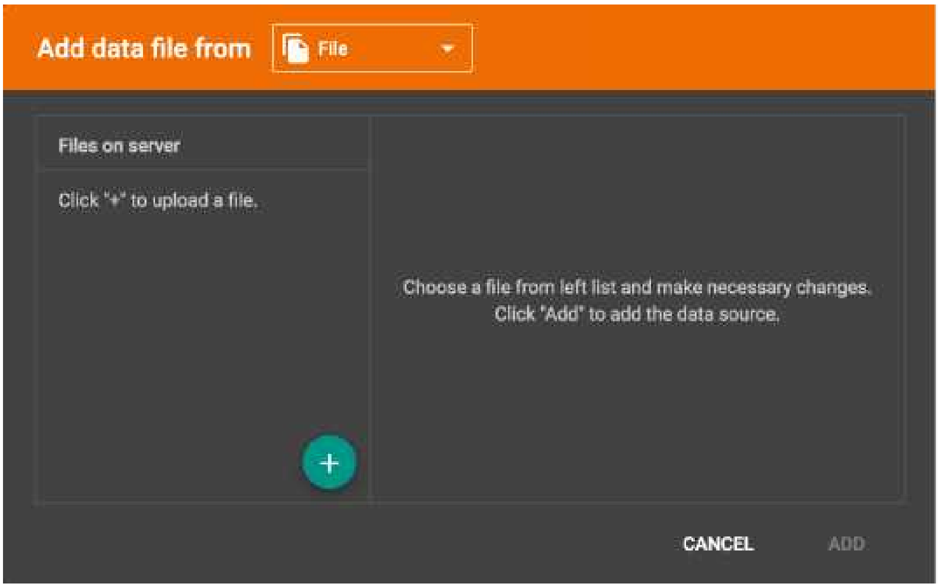
2 Add Data Files

This section contains a subsection for each of the different data sources. Read the section which pertains to your data source:

- Local File System - Add Local Data File
- AWS S3 - Create S3 File Source

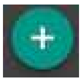
Add Local Data File

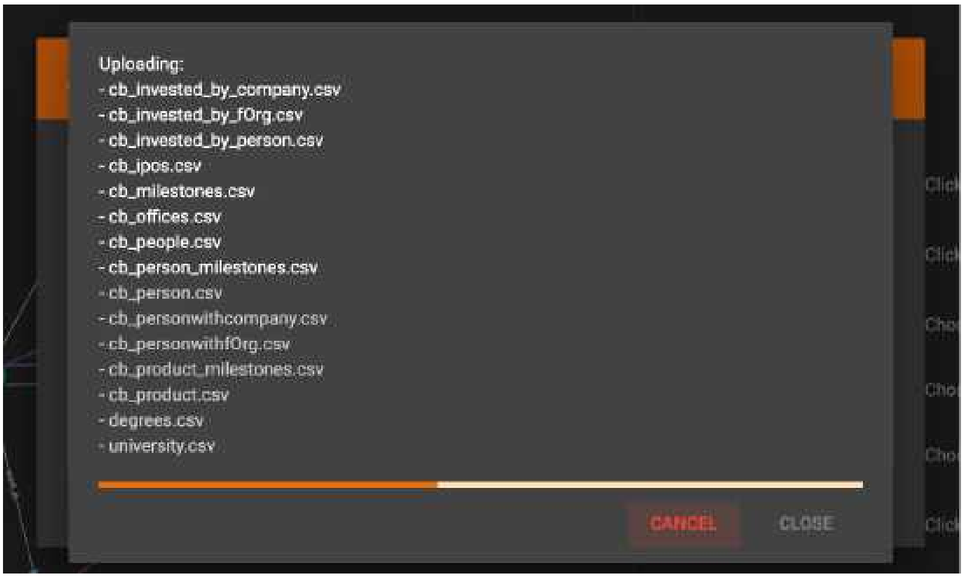
In this step, you inform GraphStudio about your data files. A data file is a file containing structured data to be loaded into the graph, creating vertex and/or edge instances. The first step for data mapping is to specify your data files. Click the Add Data File button to add data files. The Add Data File window will pop up:



Initially, there are no data files in the server data folder.

Upload File To Server

Click the Upload File button  . A file selection window will appear. Choose the data files you want to use. The files will be uploaded to the server data folder:



If all the files are uploaded successfully, the progress window will close automatically, otherwise the window will stay open to notify users of any errors.

Once the files are uploaded to the server, it will appear in the "Files on server" list on the left side of the Add Data Files window.

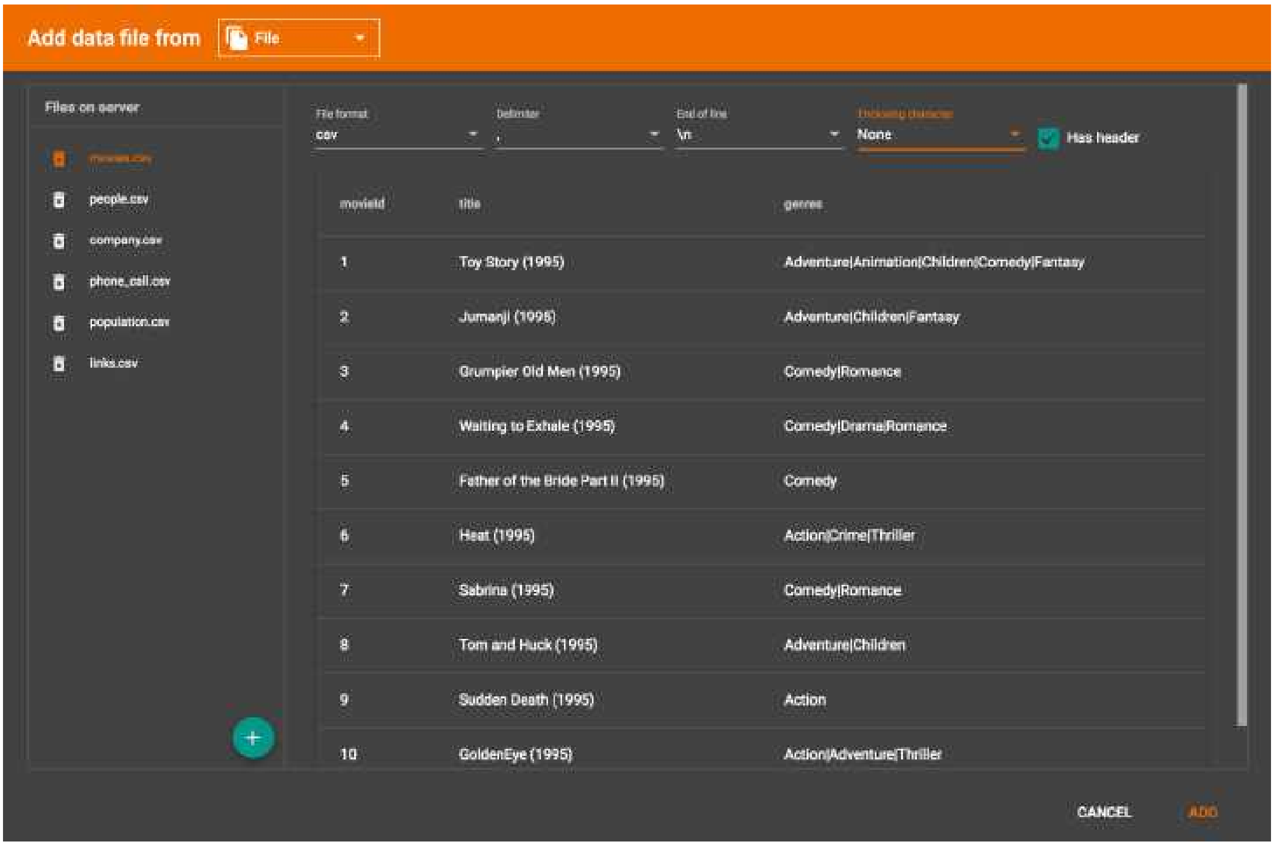
Data Files must be .csv files to allow data mapping in GraphStudio (.json file upload is allowed in TigerGraph V3.0.5+ through GraphStudio)

The Add Data File box will only upload files which end in ".csv". If you manually place files in the <TigerGraph_root_dir>/data/gui/loading_data folder, please don't put any files into subfolders because they will be ignored.

In TigerGraph V3.0.5, you can upload files which end in ".json". This allows users to upload ".json" files to the filesystems and use a remote GSQL client, or GSQL web shell in TigerGraph Cloud to map data to the graph using GSQL commands. Currently, data mapping for ".json" files through GraphStudio UI is not supported.

Configure the File Parser

In this step, you tell GraphStudio how to parse your data file. If your data file is in tabular format, the parser will split each line into a series of *tokens*. Click on one file from the file list to choose it. The parsing result for the first line of data is shown as a **preview** table on the right side:



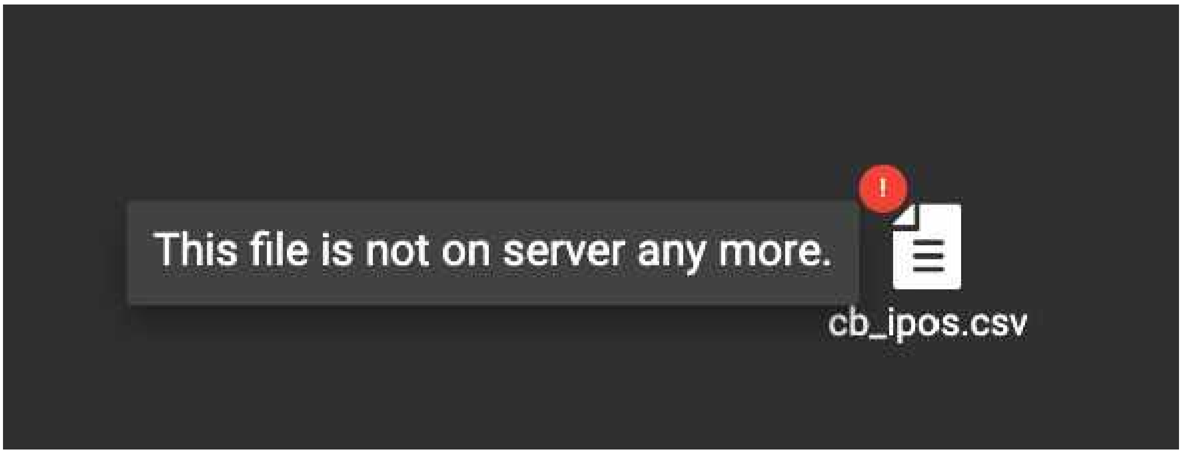
If the parsing is not correct, click on the down arrow in a table column to choose a different option for file format, delimiter, or end of line. The file will immediately be re-parsed when you change a setting. The enclosing character is used to mark the boundaries of a token, overriding the delimiter character. For example, if your delimiter is comma (,), but you have commas in some strings, then you can define either double quotes (") or single quotes (') as the enclosing character to mark the endpoints of your string tokens. It is not necessary for every token to have enclosing characters; the parser will use enclosing characters when it encounters them.

GraphStudio allows users to edit the header line of the parsing result, to make the data mapping more intuitive. This doesn't affect the data loading because the header line will be ignored.

Once you are satisfied with the file parsing configuration, click the add button 

to add the data file into left working panel. The data file will be shown as a file icon on the working panel:

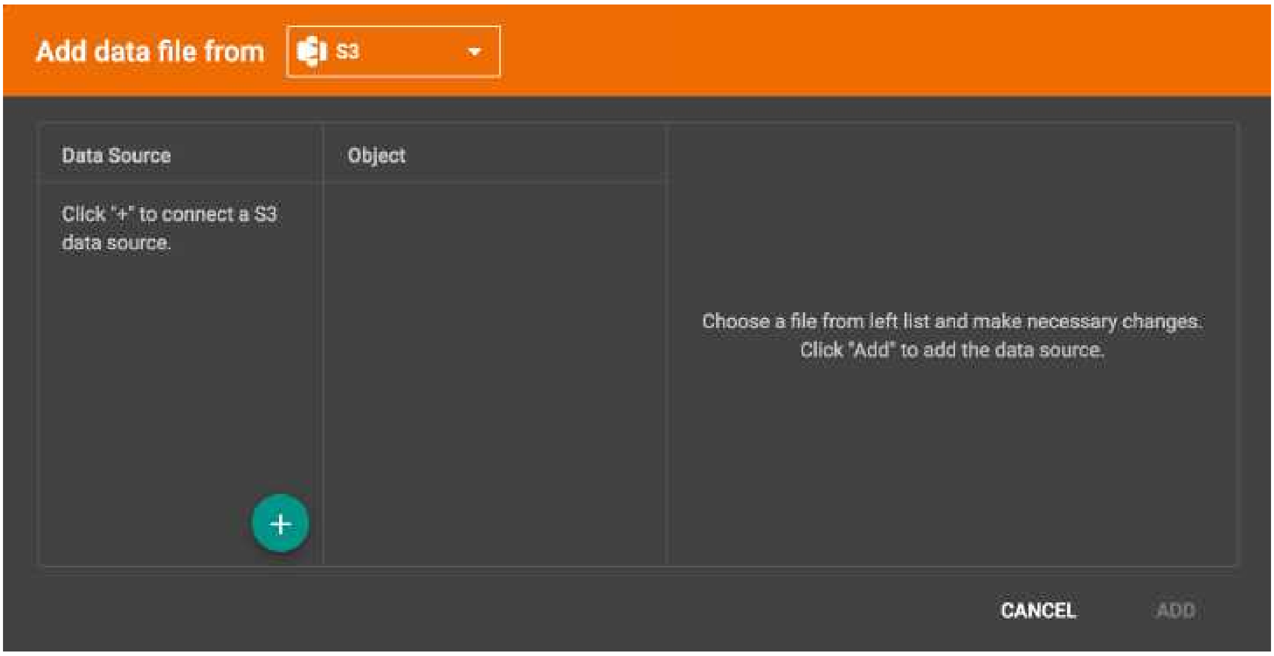
Once you think a file is no longer needed, you can remove it from server by clicking the delete button  to the left of each file. Please note that you also need to manually remove data mapping using this file as data file, otherwise when you load data later, a "file not found" error will be triggered.



After adding all your data files, continue with [Step 3 Map Data to Vertex/Edge Types]

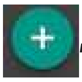
Create S3 data source

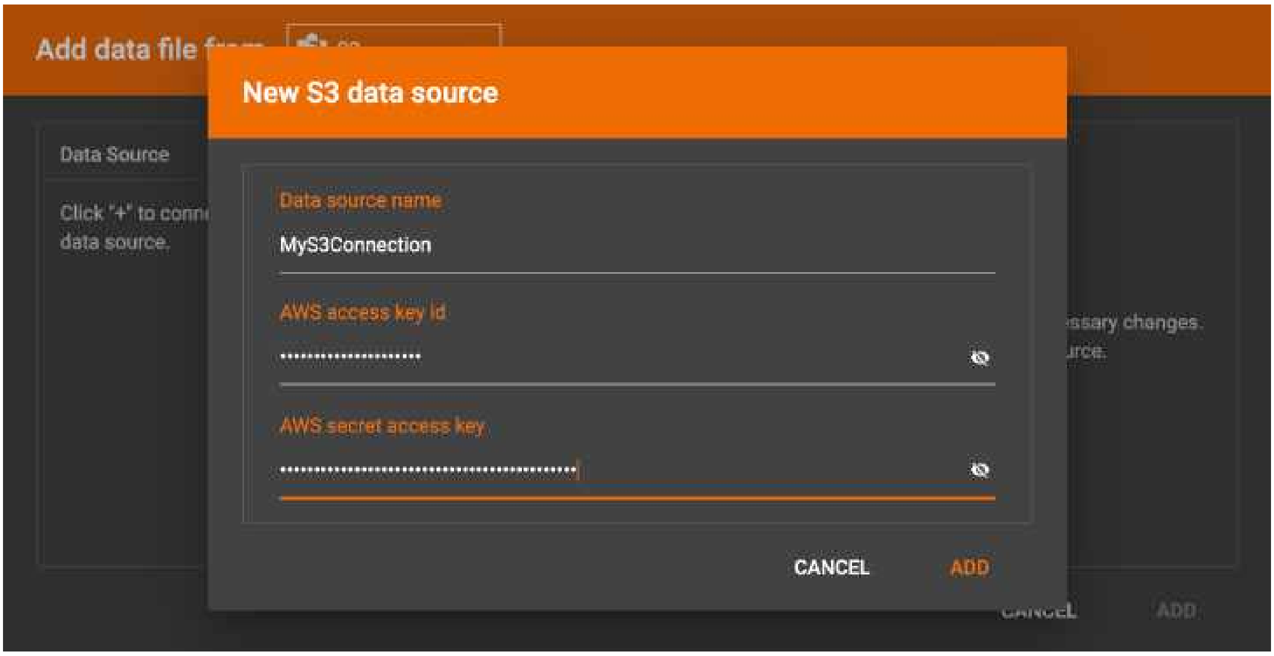
After you click the S3 data source icon, you should see the following window:



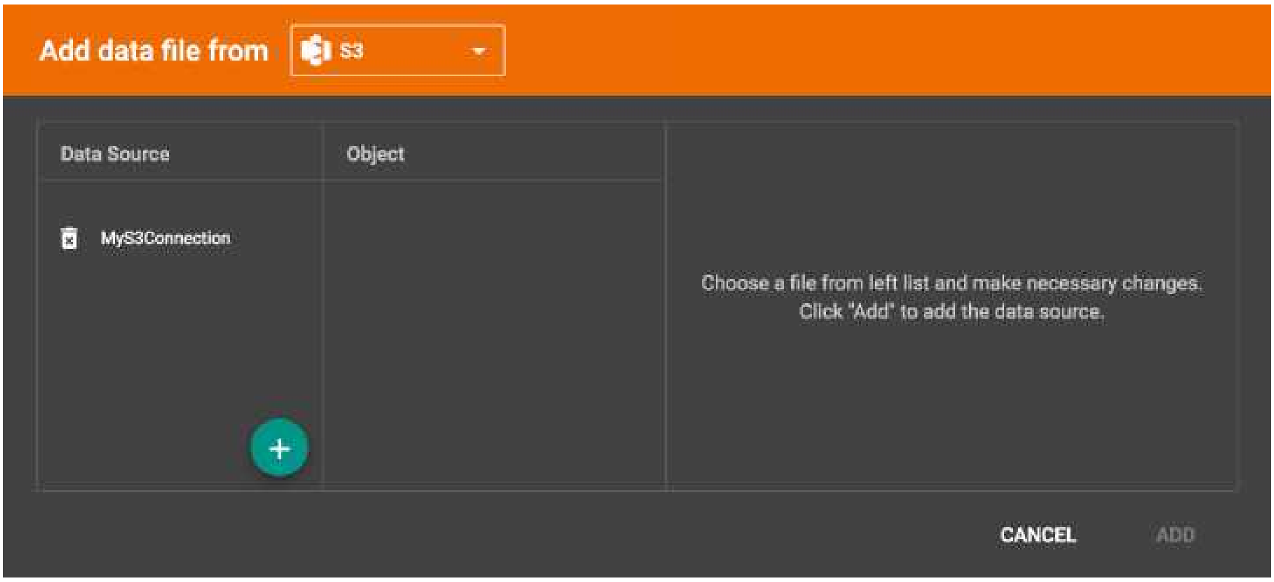
Initially, there are no S3 data sources in the system.

A data source is an appropriately configured connection to some remote source of data file(s). When the data file type is switched to S3, you can configure connection to your S3 buckets.

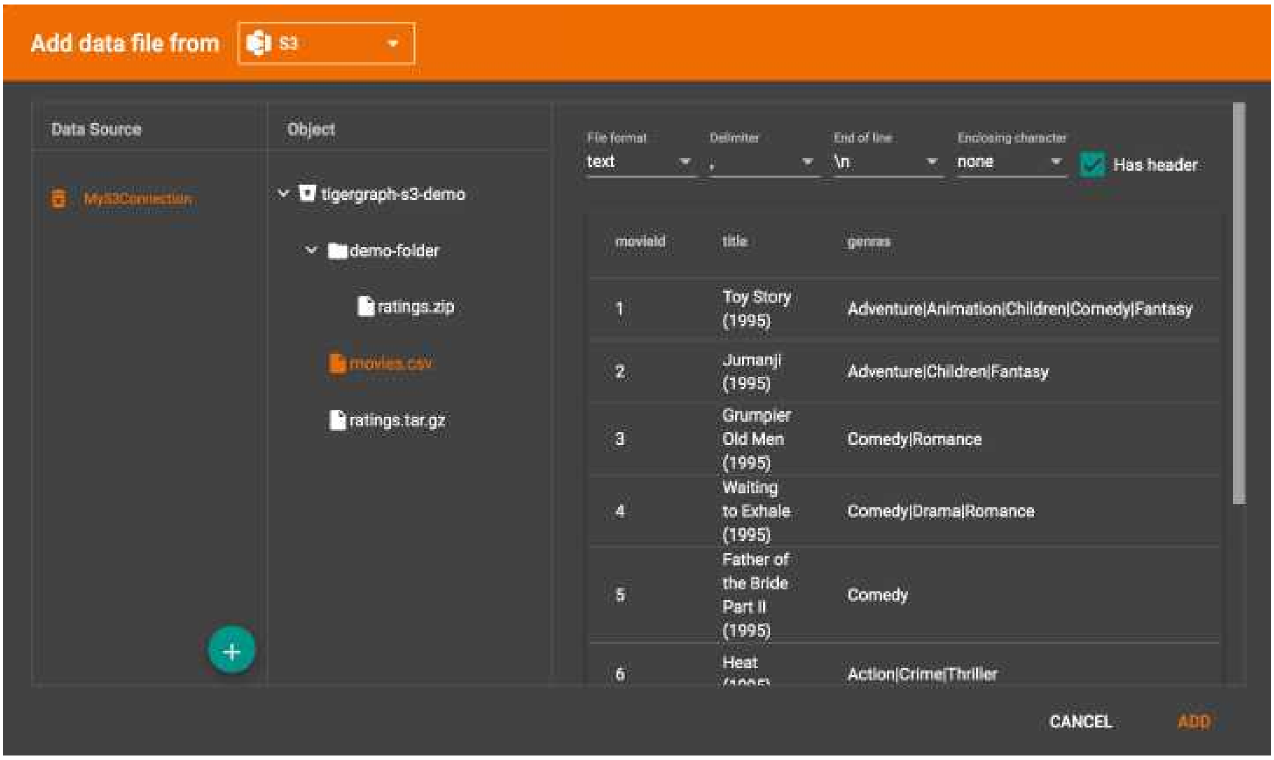
Click the Add new data source button , then the new S3 data source window will pop up. Give a name to the data source, and provide the access key id and secret access key to connect to S3. Then click the ADD button:



The data source will be created and shown in the Data Source list:



Click the data source to list all the buckets the credentials can access, and click the Expand icon to see all the buckets or folders within the buckets. The file hierarchy will be shown as a tree. Choose the file you want to add, and change the parsing options if necessary. (See [Configure the file parser.](#))



Data files, after decompression, must be in either csv or parquet format.

TigerGraph supports loading from archived and compressed S3 files directly. Currently supported file extensions includes zip, tar.gz, tgz and tar. GraphStudio detects the file extension and automatically chooses the corresponding file format. If the file is encoded with one of these formats but has a non-standard file extension, you can manually specify the File format.

After clicking the ADD button, an S3 file icon will appear on the working panel:

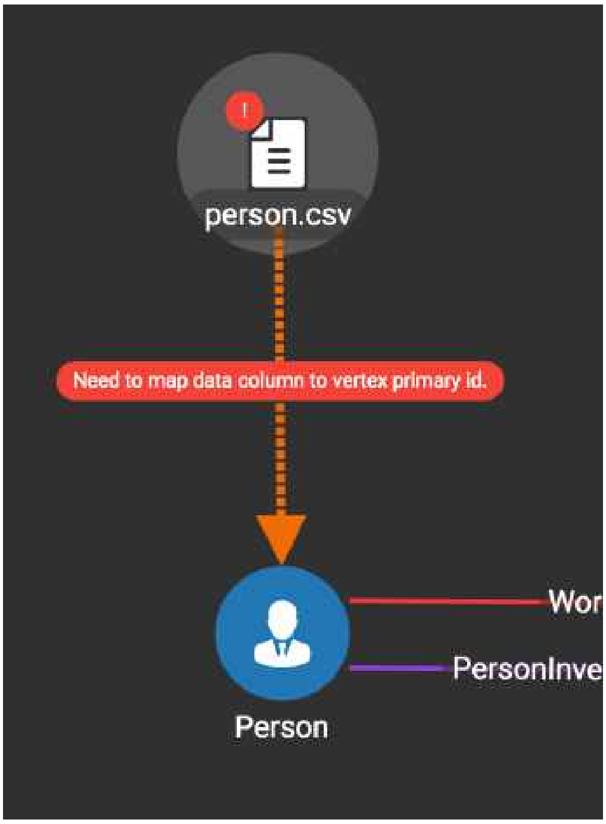
After adding all your s3 data files, continue with Step 3 Map Data Files to Vertex/Edge Types

3 Map Data Files To Vertex Type Or Edge Type

In this step, you link (map) a data file to a target vertex type or edge type. The mapping can be many-to-many, which means one data file can map to multiple vertex and / or edge types, and multiple data files can map to the same vertex or edge type. Click the map data file to vertex or edge button  to enter *map data file to vertex or edge* mode. When you finish mapping the data files, click the button again to exit this mode.

Then, click the data file icon. A hint will appear over the icon:

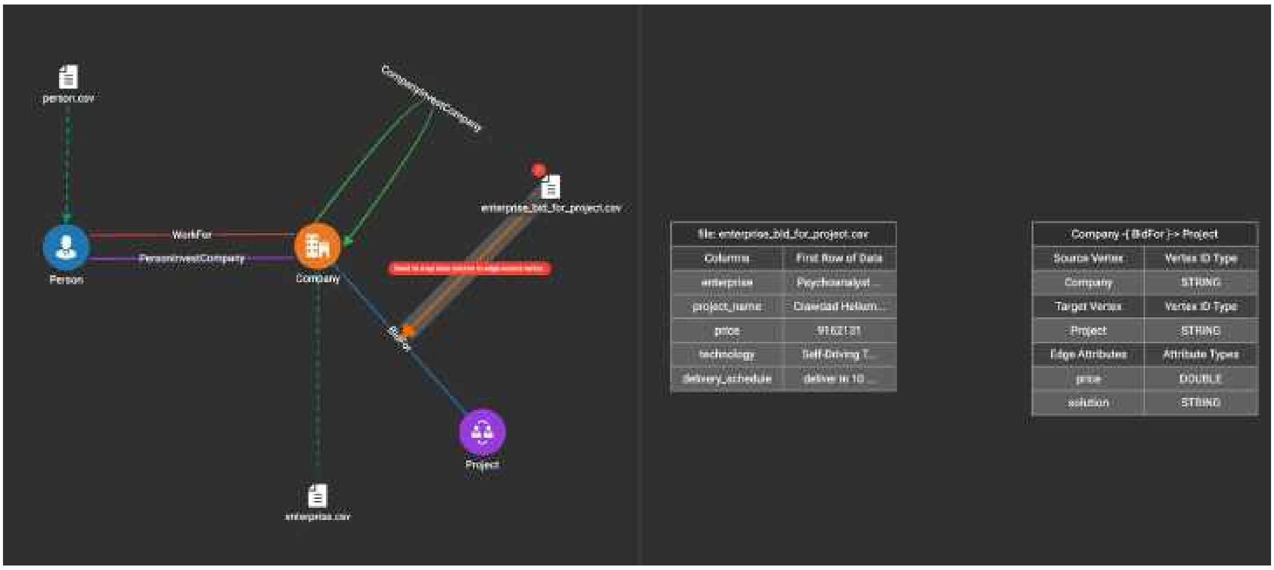
Next, click the target vertex type circle or edge type link. A dashed link will appear between the data file and the target vertex or edge type:



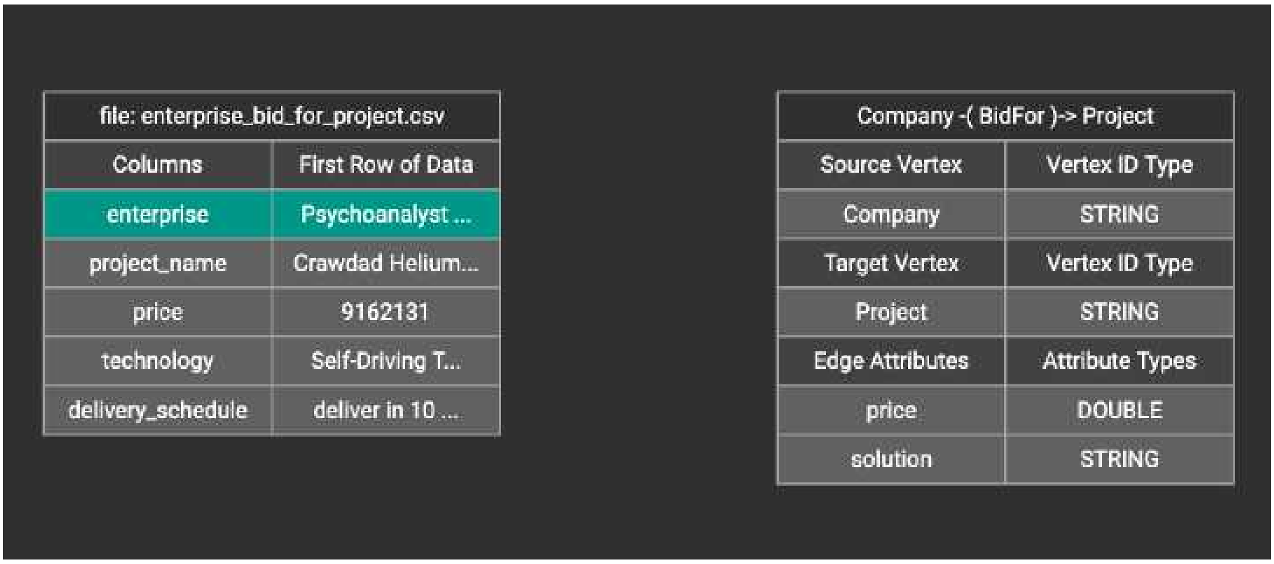
A red hint will appear if the target type has not yet received a mapping for its primary id(s).

4 Map Data Columns To Vertex Or Edge Attributes

In this step, you link particular columns of a data file to particular ids or attributes of a vertex type or edge type. First, choose one data mapping from one data file to one vertex or edge type (represented as a dashed green link on the left working panel). When selected, the dashed line becomes orange (active), and the right working panel will show two tables. The left table shows the data file columns along with the first row's tokens as sample data. The right table shows the fields of the target vertex or edge. For a vertex, its fields are primary id and attributes. For an edge, its fields are source vertex, target vertex, and attributes.



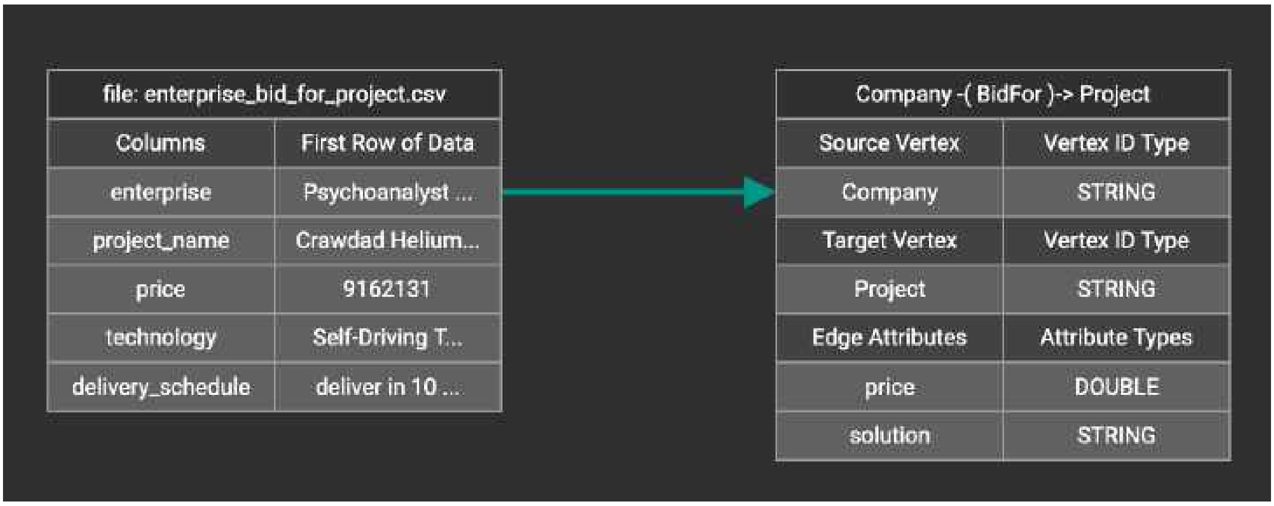
In order to a column in the data file to a vertex or edge field, first click the row representing the data column in the left side data file table:



file: enterprise_bid_for_project.csv	
Columns	First Row of Data
enterprise	Psychoanalyst ...
project_name	Crawdad Helium...
price	9162131
technology	Self-Driving T...
delivery_schedule	deliver in 10 ...

Company -(BidFor)-> Project	
Source Vertex	Vertex ID Type
Company	STRING
Target Vertex	Vertex ID Type
Project	STRING
Edge Attributes	Attribute Types
price	DOUBLE
solution	STRING

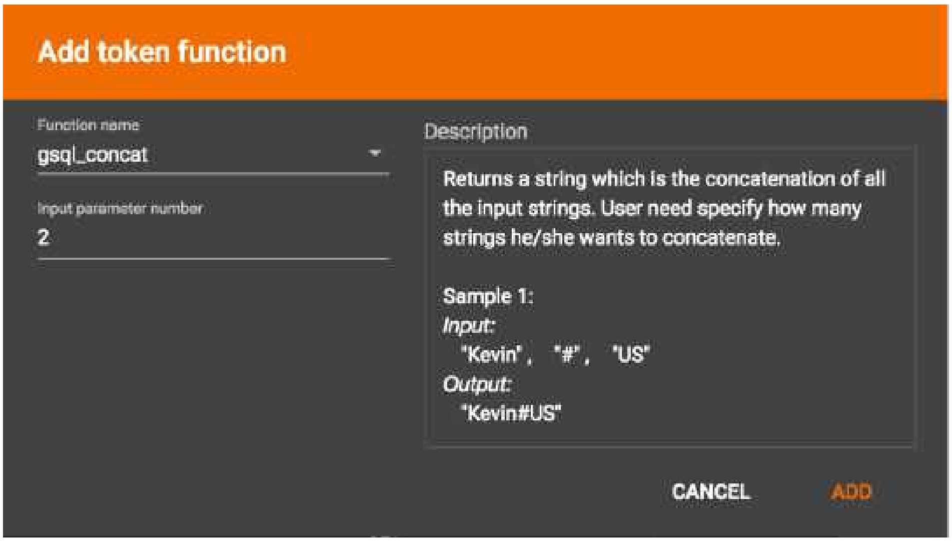
Then, click the row representing the target field in the right side table. A green arrow appears to show the mapping. Repeat as needed to create all the mappings for this table-to-vertex/edge pair. Since many-to-one mapping is allowed, it is not necessary for one table to provide a mapping for every field in the target vertex/edge.



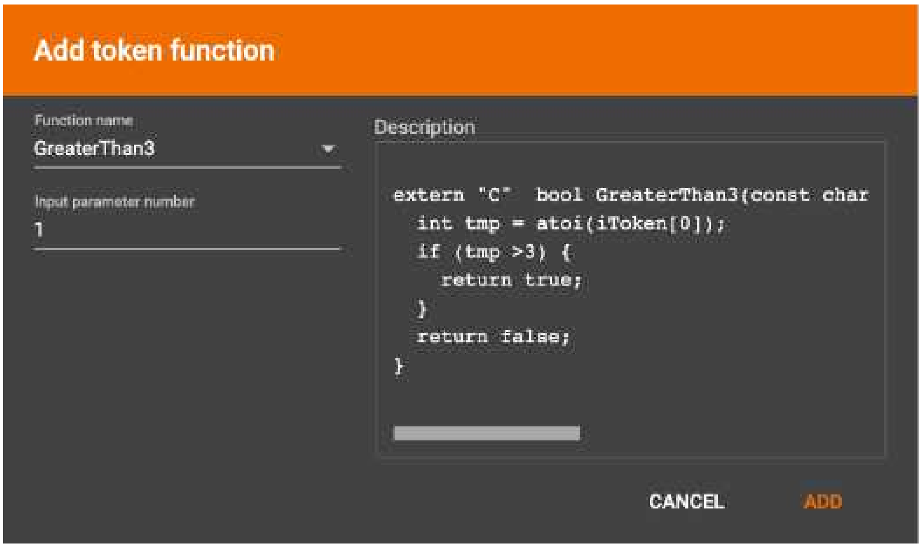
Using a Token Function

GraphStudio gives you access to both a set of built-in functions and user-defined token functions to preprocess data file tokens before loading them in to the graph. For example, you can concatenate two columns in the data file and load them as an attribute. This section describes how to use these token functions.

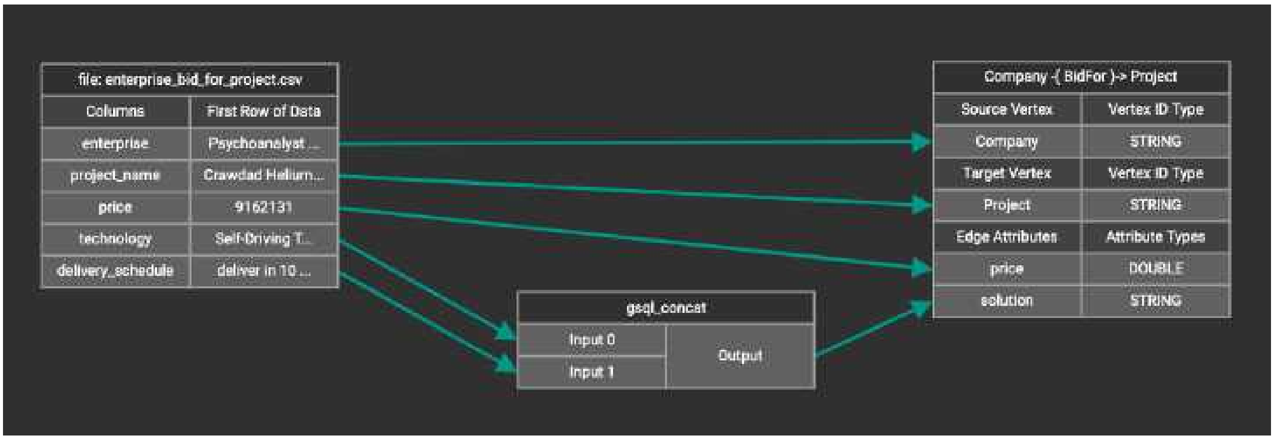
First click the add token function button Σ. The Add Token Function window will pop up. Click the down arrow to see the list of available token functions and select one. For some functions, you may also specify the number of input parameters. (Most token functions have a fixed number of input parameters; gsql_concat can accept any positive number of inputs). Click Add.



GraphStudio currently does not support creating new user-defined functions. If a user-defined function has been added via the GSQL interface, it will be listed here. To use a user-defined token function, you must manually specify the number of input parameters. The C++ code is shown in the Description section for your reference:



A token function table will be added to the attribute mapping panel. You can drag the tables to re-arrange them. Token functions act as intermediate steps in the mapping. Create mappings from the data file table to the token function table, and then from the token function table to the vertex/attribute table. The final result looks like below:



Auto Mapping

If the data file columns and the vertex/edge attributes have very similar names (only capitalization and hyphen differences), you can click the auto mapping button . All similar columns will be mapped automatically.

Map A Constant Value To An Attribute Or Token Function Input

Sometimes, a user may need to load a constant value to an id or attribute. Here we show how to do this in GraphStudio.

Loading A Constant to An Attribute

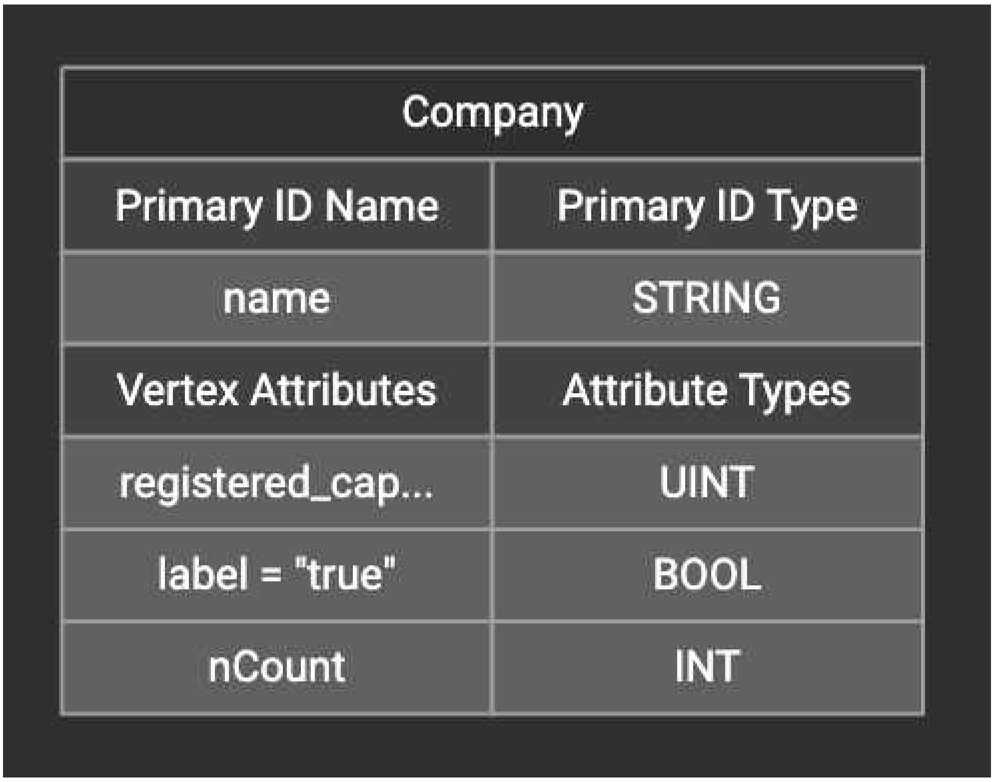
In the right working panel, double-click on the target id or attribute (in the left column of the right table). In the example below, the attribute "label" has been double-clicked:

file: sc_companies.csv	
Columns	First Row of Data
id	c1
label	1
company_name	com1

Company	
Primary ID Name	Primary ID Type
name	STRING
Vertex Attributes	Attribute Types
registered_cap...	UINT
label	BOOL
nCount	INT

This will cause the Load Constant window to pop up. Type in the constant value, and click the Add button to apply the mapping.

After adding the constant value, the attribute's label will change to **id/attribute = "(your valid input value)"** .

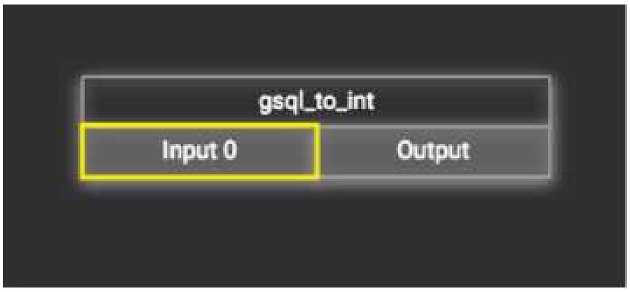


Company	
Primary ID Name	Primary ID Type
name	STRING
Vertex Attributes	Attribute Types
registered_cap...	UINT
label = "true"	BOOL
nCount	INT

To modify or remove a constant mapping, double-click the id/attribute again. In the Load Constant window, enter the new value, or erase the value if you want to remove the mapping. Click the Add button to apply.

Use A Constant Input for a Token Function

First add the token function. Then double-click on the target input (in the left column of the token function table). In the example below, "Input 0" has been double-clicked.



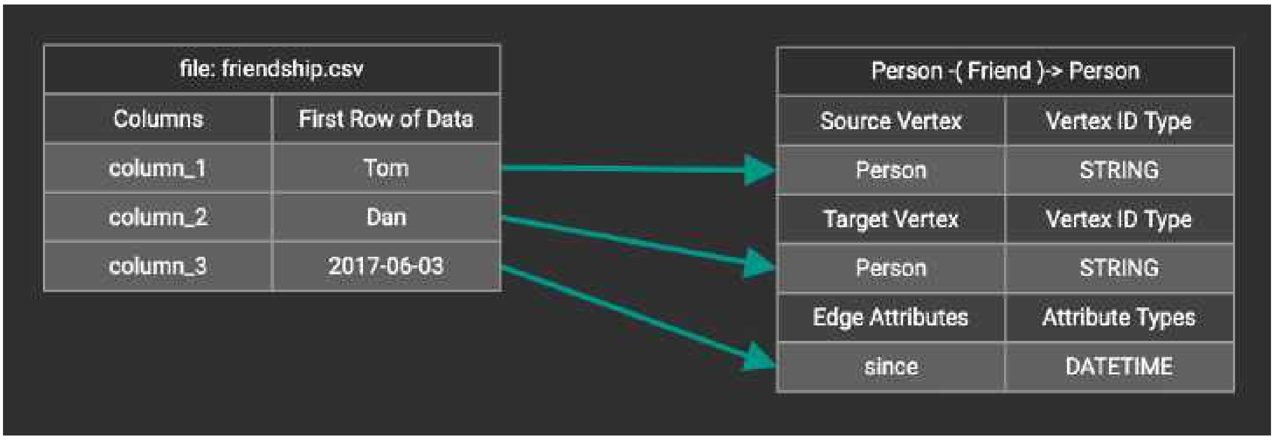
This will cause the Load Constant window to pop up. Type in the constant value and click the Add button to apply the mapping. After adding the constant value, the input's label will change to **Input = "(your input value)"** .

The constant value can be modified or removed by double-clicking the label and editing the value in the Load Constant window.

Add Data Filter

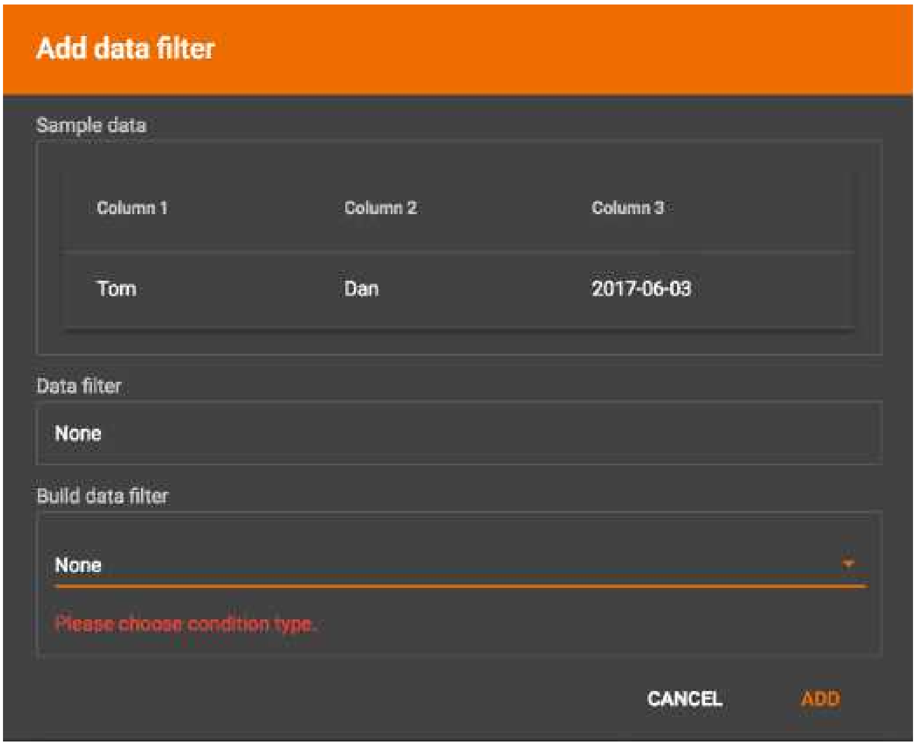
You can add a data filter to a data mapping so that only data records which meet conditions that you specify will be loaded into the graph. This is equivalent to the WHERE clause in a GSQL load statement.

You can add one data filter for each data mapping from a data file to a vertex type or edge type, and the data filter only applies to that one mapping. Consider the following data mapping:



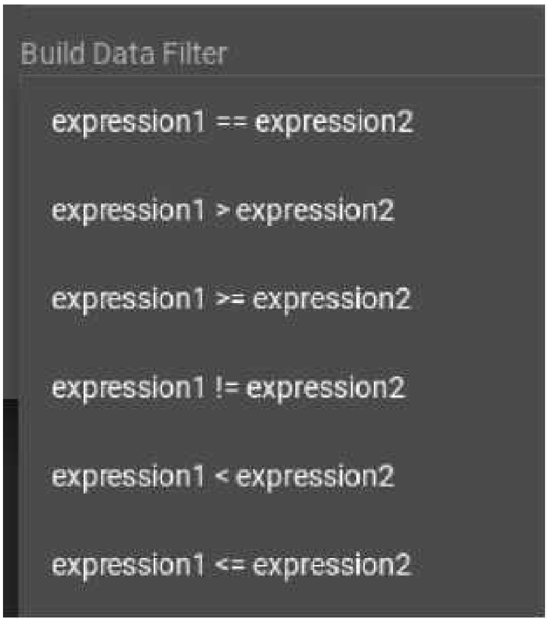
By default, there is no data filter. Click the Data Filter button  to start creating a data filter. The Add Data Filter window will appear. The window contains three parts:

1. The top section shows one row of sample data from your file, as a handy reference to the file's contents.
2. The middle sections shows what the data filter looks like when it is converted a to GSQL WHERE clause. For more details, see the **WHERE Clause** section in the [GSQL Language Reference Part 1 - Defining Graphs and Loading Data]
3. The bottom section is where you define your data filter. The data filter will be converted to a GSQL WHERE clause and shown in real time.

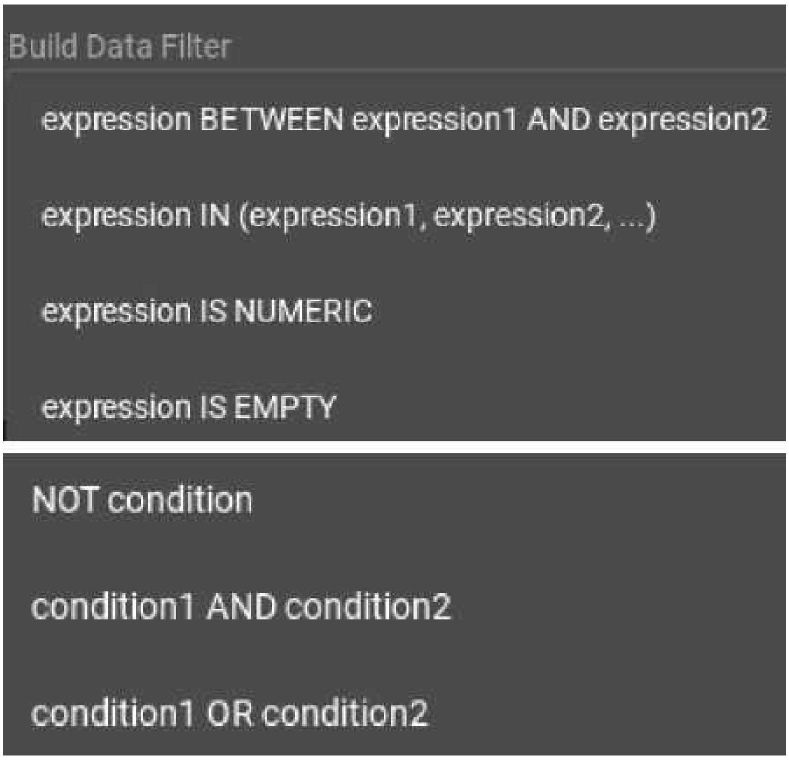


A data filter condition is a Boolean expression, which can be a nested set of conditions. TigerGraph data loader evaluates the condition for each line in your input file. If the condition evaluates to be true, then the line of data is loaded.

First, click the Build Data Filter chooser (with default value "None"). A menu will appear, with many Boolean expression templates. Choose one of the options. If you plan to build a nested condition, start with your top level. The first several options are for comparison expressions:



After this are several more options, using operators such as AND, OR, NOT, IN, BETWEEN...AND, IS NUMERIC, and IS EMPTY.

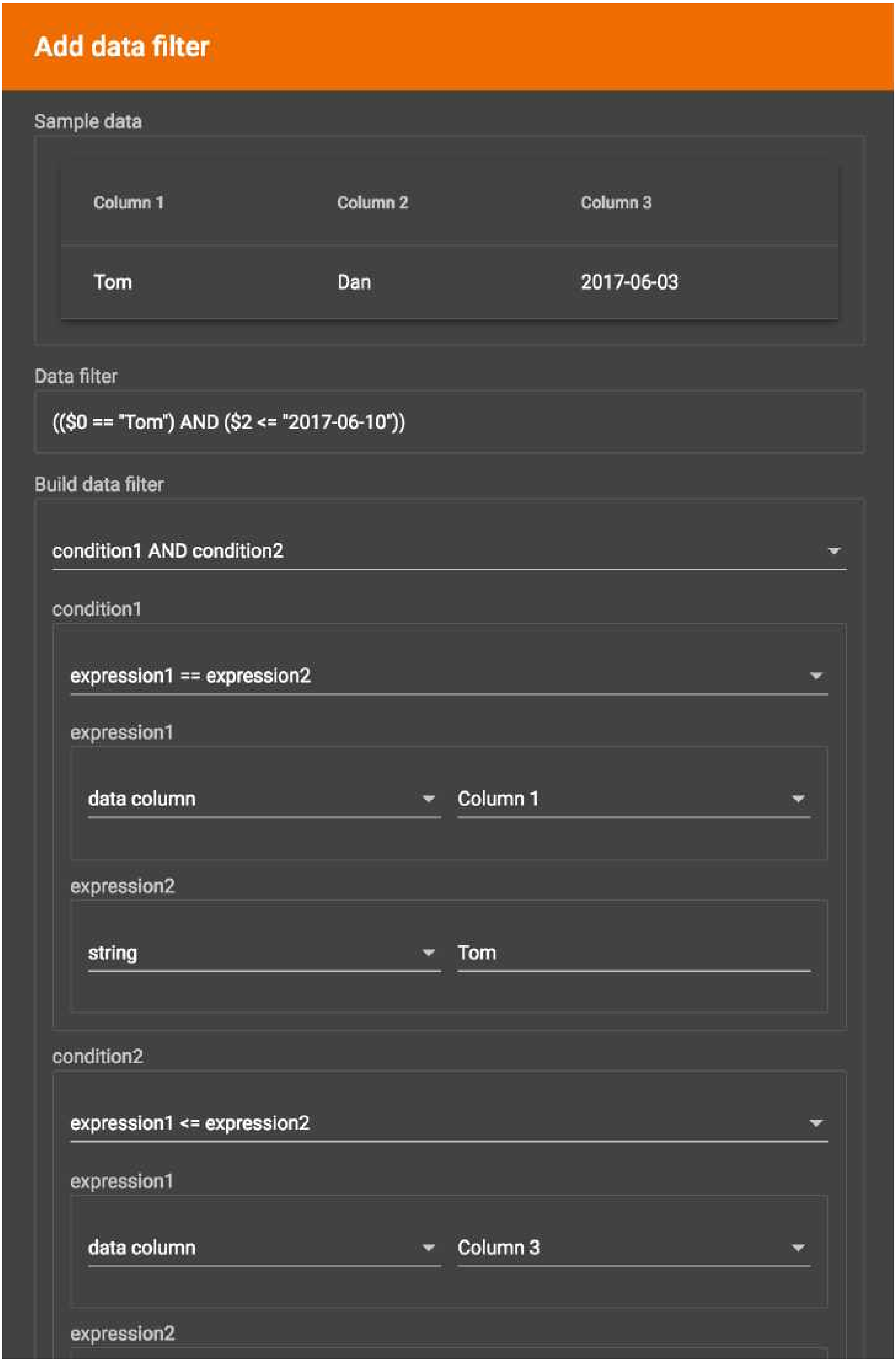


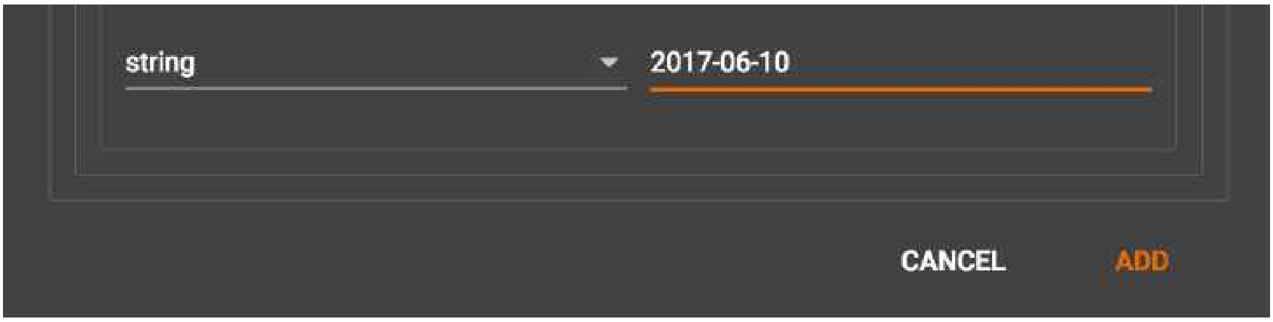
Note that each of these expressions calls for 1, 2, 3, or a list of operands, and the operands themselves can be expressions. When you select an expression, additional choosers will appear below for you to specify the operand expressions. The operand choices are context-sensitive, but typically they include

- a Data Column from the input file

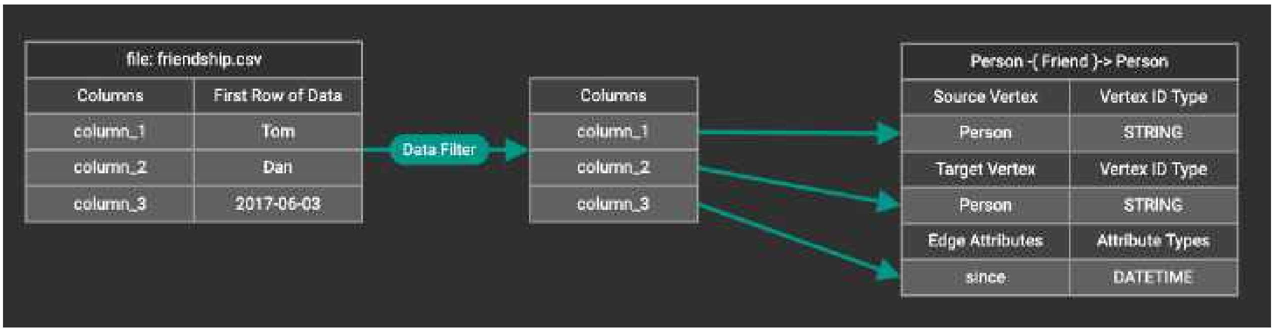
- A constant value
- If the operator is AND, OR, or NOT, then the operand can be another condition. Thus is how conditions can be nested.

Suppose you are loading friendship edges where the input data fields are (person1, person2, friendship_start_date). You want to load only the records where person1 is Tom and the friendship began on or before 2017-06-10. The data filter looks like the following:


Add data filter
Sample data
Column 1
Column 2
Column 3
Tom
Dan
2017-06-03
Data filter
(($0 == "Tom") AND ($2 <= "2017-06-10"))
Build data filter
condition1 AND condition2
condition1
expression1 == expression2
expression1
data column
Column 1
expression2
string
Tom
condition2
expression1 <= expression2
expression1
data column
Column 3
expression2



After adding the data filter, the right working panel will look like this:



Hovering the mouse over the data filter indicator 

will make the data filter condition appear. If you want to modify the data filter, click the Data Filter button  or double-click the data filter indicator. The Add Data Filter panel will appear.

To remove a data filter, select "None" at the top level dropdown of Build Data Filter section and then click Add. The data filter will be deleted.

Advanced Features

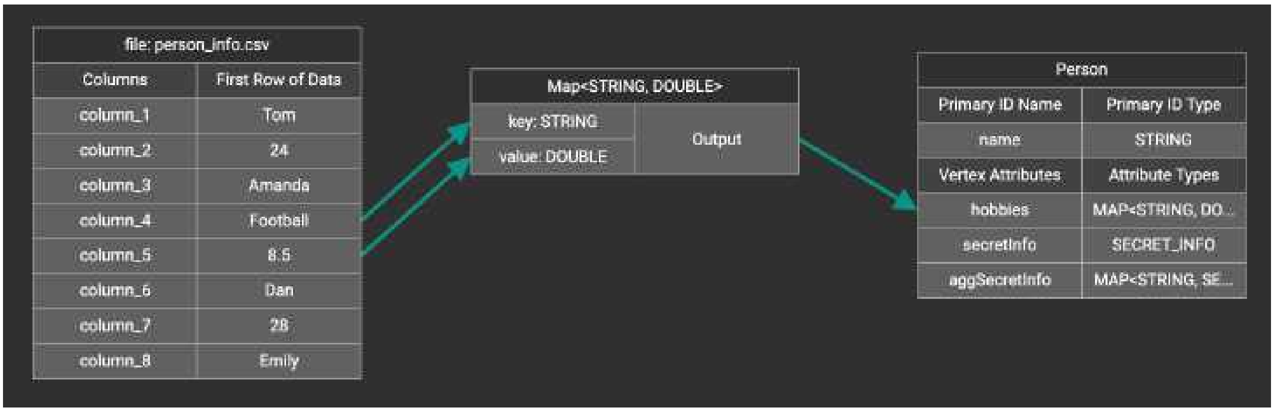
More advanced data mapping features are grouped in the dropdown list under .

Map data to a map type attribute

Click {a:b} Map widget in the dropdown list, then choose key type and value type. The types must match the key type and value type of the attribute you are mapping towards.

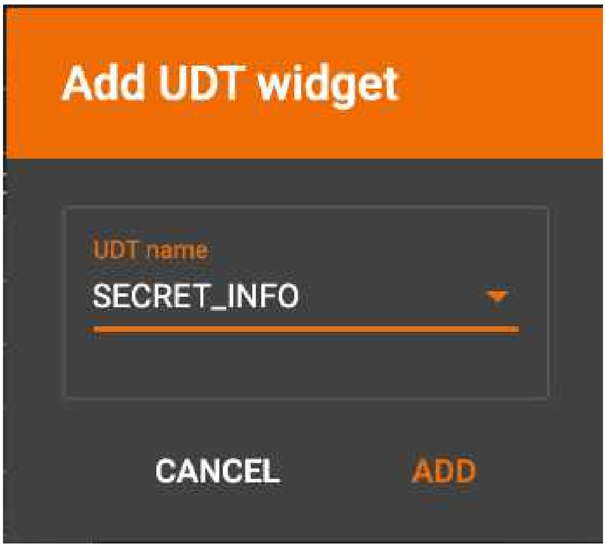
A Map widget will be added to the attribute mapping panel.

Create the mapping from the data columns to the Map widget, and from the Map widget to the attribute.



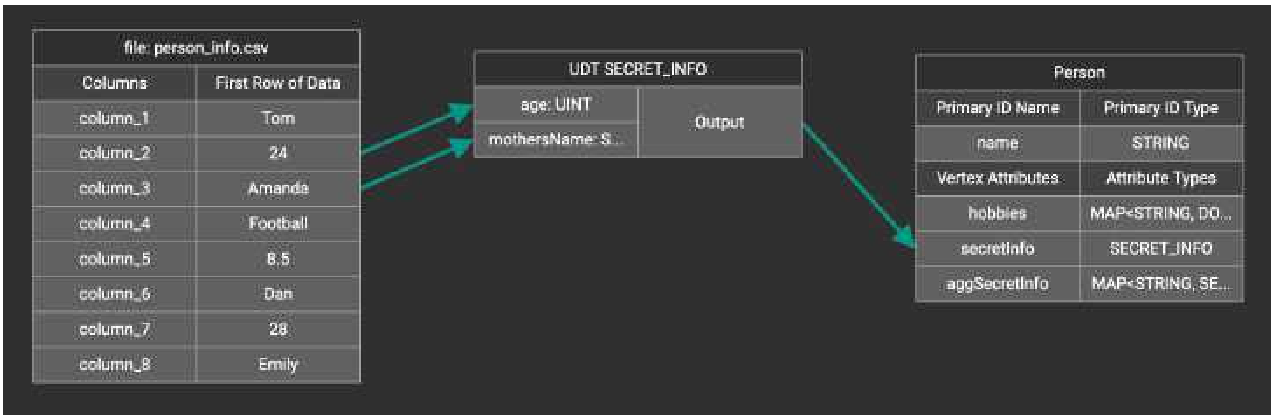
Map data to a UDT type attribute

Click UDT widget in the dropdown list, then choose UDT name. The name must match the UDT type of the attribute you are mapping towards.



A UDT widget will be added to the attribute mapping panel.

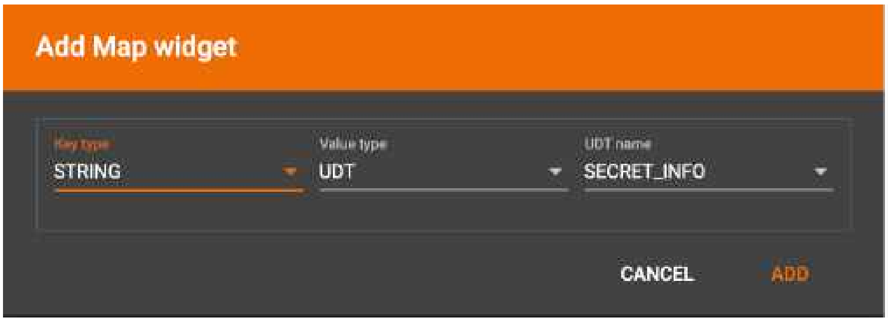
Create the mapping from the data columns to the UDT widget, and from the UDT widget to the attribute.



Map data to a map type attribute with UDT value type

If you want to map data to an attribute of map type with UDT value type, you have to combine a Map widget with a UDT widget.

Choose UDT as the value type and then choose the UDT name when adding the Map widget.



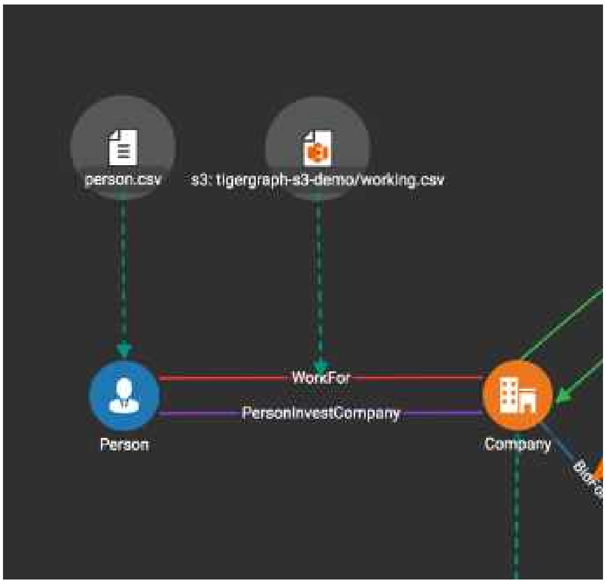
Create data mapping between data columns, the UDT widget, the Map widget, and the attribute.

Delete Options

In the Map Data To Graph page, you can delete anything that you added. Choose what you want to delete, then click the delete button . Press the "Shift" key to select multiple icons you want to delete. Note that you cannot delete vertex or edge types in this page.

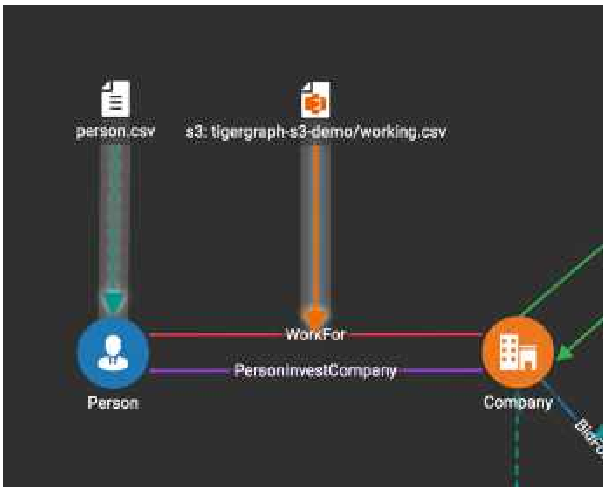
Delete Data Files

Select the data file icon(s), then click the delete button.



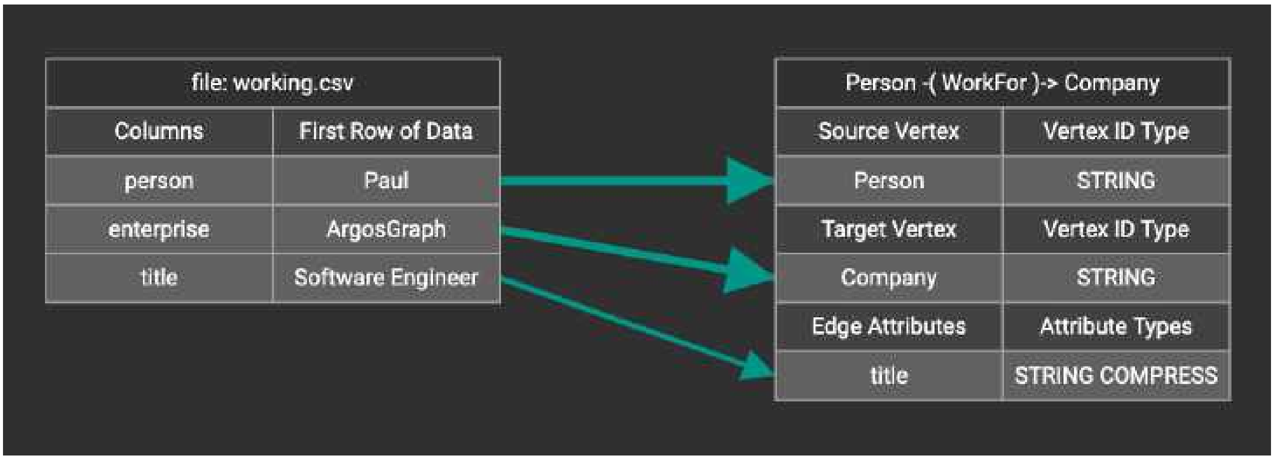
Delete Data File To Vertex Or Edge Mapping

Select the dashed green link(s) between data file and mapped vertex/edge type, then click the delete button.



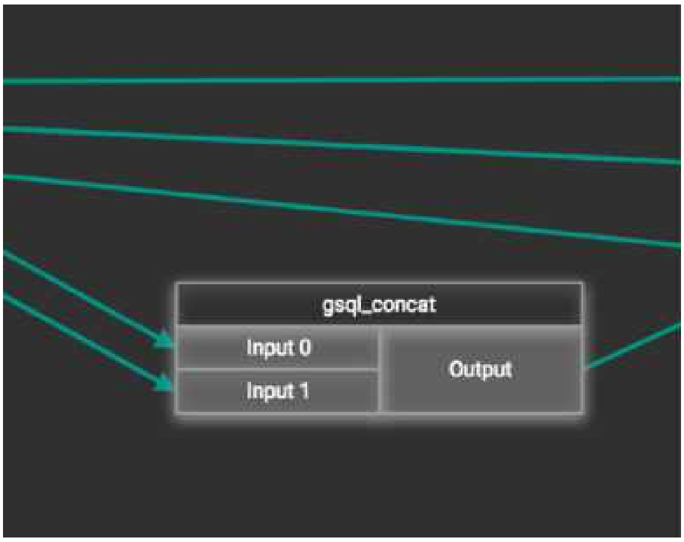
Delete Data Column To Vertex Or Edge Attribute Mapping

Select the green arrow(s) between data file table and vertex/edge attributes table, then click the delete button.



Delete Token Functions

Select the token function table(s), then click the delete button.



Undo And Redo

You can undo or redo changes by clicking the Back or Forward buttons, respectively: . The whole history since the time you entered the Map Data To Graph page is recorded.

5 Publish Data Mapping

Once you are satisfied with the data loading procedure, click the publish schema button  to publish the data loading procedure to the TigerGraph system. It takes about 2 to 3 seconds for publishing each data file mapping.

Expand Panels

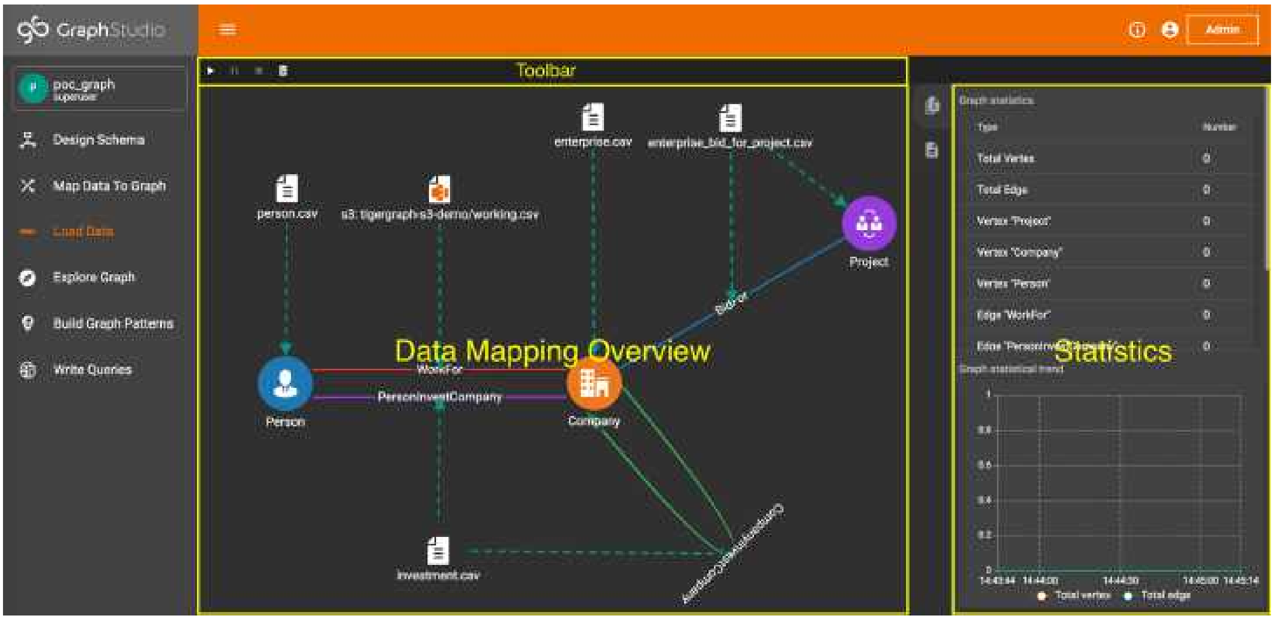
The following three buttons allow you to expand the left or right working panel: .

By default, the two panels have equal widths. Click the left button to expand the left working panel, or click the right button to expand the right working panel.

Load Data

Load Data

After mapping data files to the graph schema, you can start loading data. Click "Load Data" on the left side menu bar to go to the Load Data page.



The "Load Data" interface is separated into three parts:

- Data Mapping Overview
  - Provides a general view of the graph and the data mapping.
  - Shows the loading progress of each data file.
- Toolbar (above Data Mapping)
  - Start/pause/resume/stop data loading and clear graph data buttons.
- Statistics
  - Graph statistics: displays the numbers of vertices and edges in total and per type, with real-time loading progress.
  - Loading statistics: displays the total number of vertices and edges loader vs. time.

To display real-time graph statistics, this page checks the number of vertices and edges every 10 seconds, which adds overhead. To maximize loading performance, move to a different page after starting loading, and only come back here occasionally to check the progress.

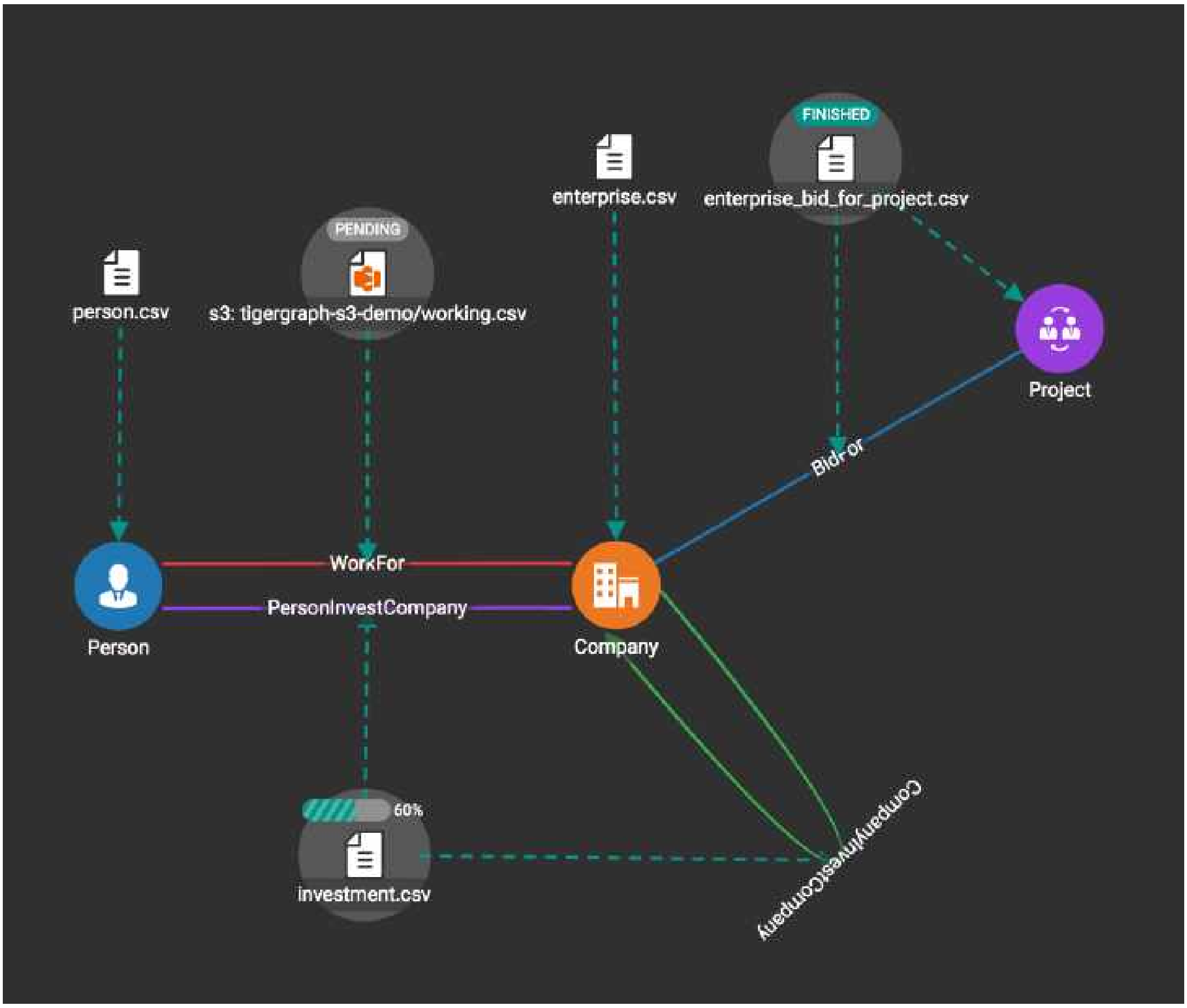
Start Loading

GraphStudio provides two types of loading:

- Partial Loading: load a subset of the data files which the user selects.
- Full Loading: load all of the data files.

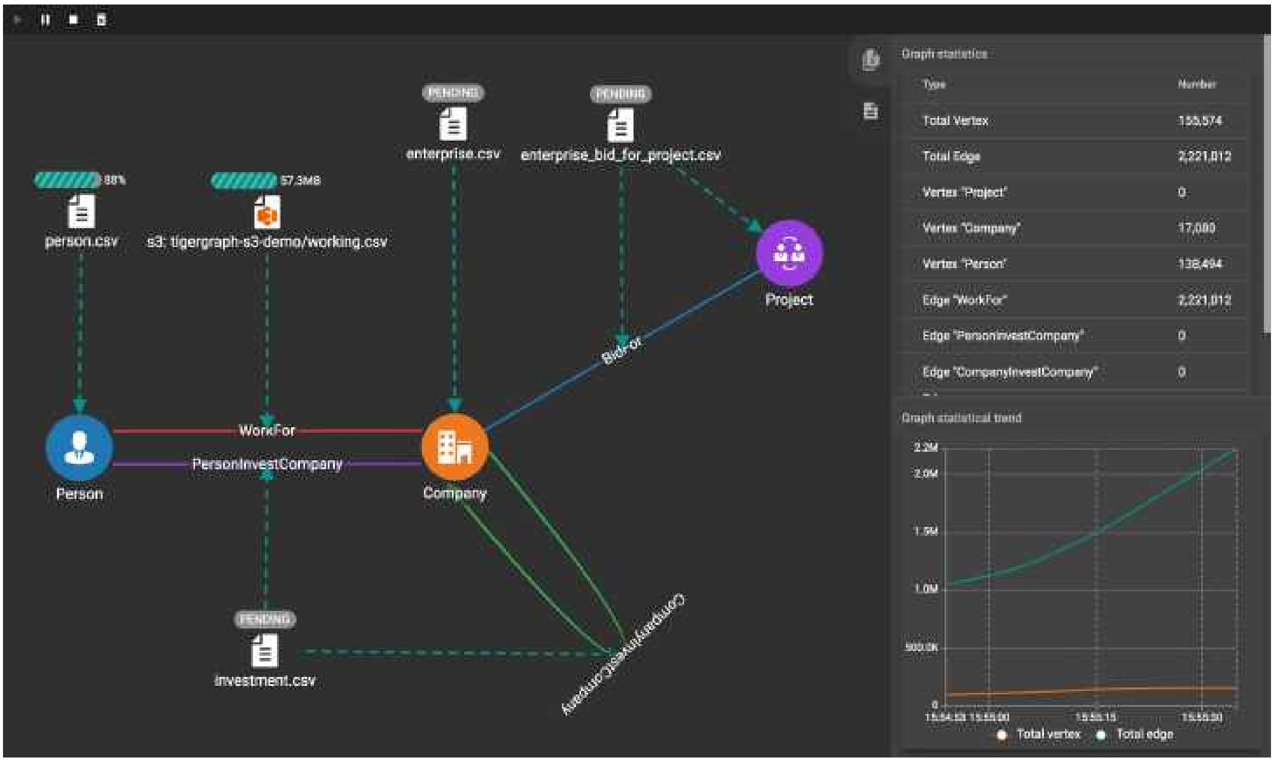
Load Some Data Files

Select one or more data files (holding down the "shift" key to select multiple data files), and click on the "start loading" button on the toolbar.



Load All Data Sources

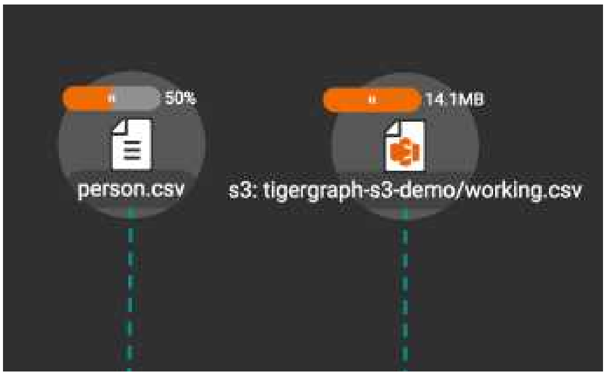
Click on a blank space in the data mapping overview panel to unselect the data sources, and click on the "start/resume loading" button ▶ on the toolbar. While loading is in progress a green hatched bar will appear over each data file to show its real time progress.



Pause Loading

Similar to Start Loading, you can pause loading some of the data files, or all loading data files.

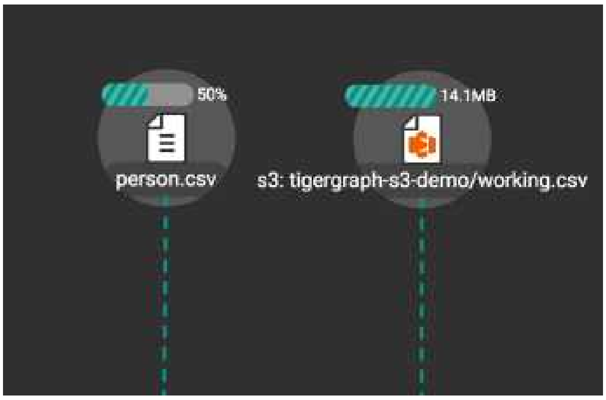
Select one or more data files (holding down the "shift" key to select multiple data files), and click on the "pause loading" button on the toolbar. In the Paused state, the progress bar will change to a solid orange color.



Resume Loading

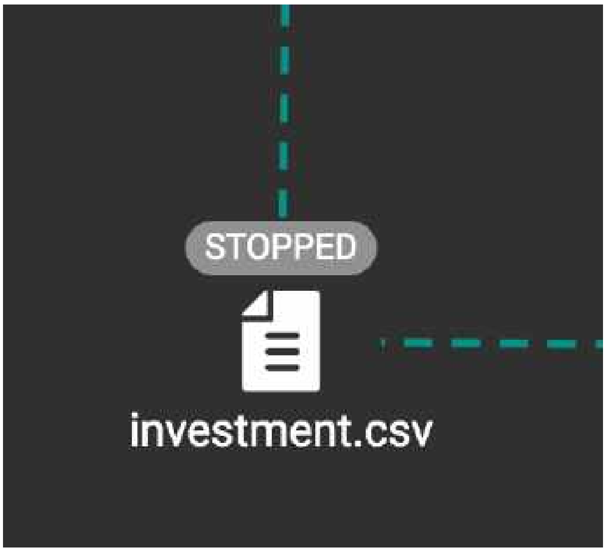
You can resume loading some or all loading data files which have been paused.

Select one or more data files (holding down the "shift" key to select multiple data files), and click on the "start/resume loading" button on the toolbar. After resuming, the data file loading will continue from where it was paused:



Stop Loading

After loading has been started or paused, you can stop loading from these data files by clicking the "stop load" button. Similar to Start Loading, you can stop loading some or all loading data files. After stopping, the loading status of the data files will become "Stopped":

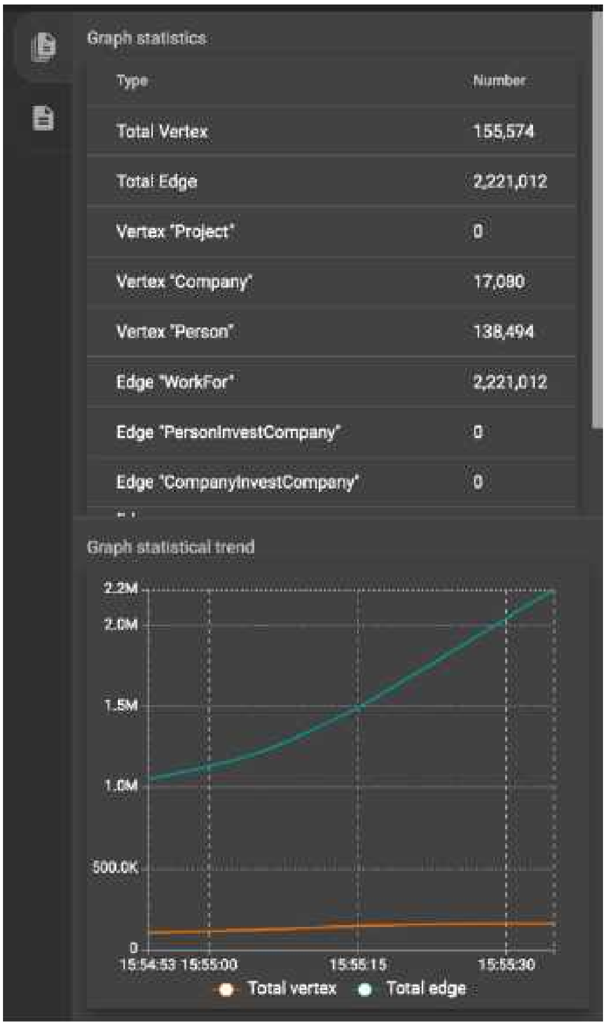


Statistics Panel

The Statistics panel contains two tabs: Graph Statistics (1st tab) and Data Loading Statistics (2nd tab).

Graph Statistics

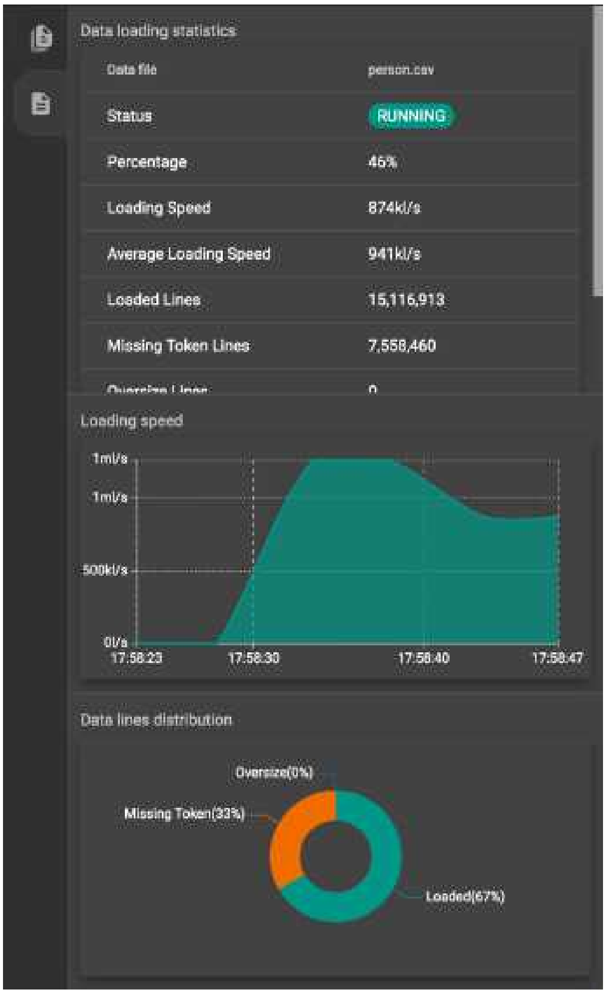
By default if no data file is selected, the Statistics panel will show Graph Statistics.



The table at the top shows the total number of vertices and edges in the current graph, and the number of each vertex type and edge type as well. The line chart at the bottom shows the number of vertices and edges over time, when loading is in progress.

Data Loading Statistics

If you click on one data file, the Statistics panel will change to show Data Loading Statistics:



The table at the top shows the detailed loading information of the selected data file, including:

- Status (RUNNING, PAUSED, STOPPED, etc)
- Loaded percentage (for files on server) or loaded size (for S3 file)
- Loading speed
- Average loading speed
- Number of loaded lines
- Number of missing token lines
- Number of oversize lines
- Loading start time
- Loading duration

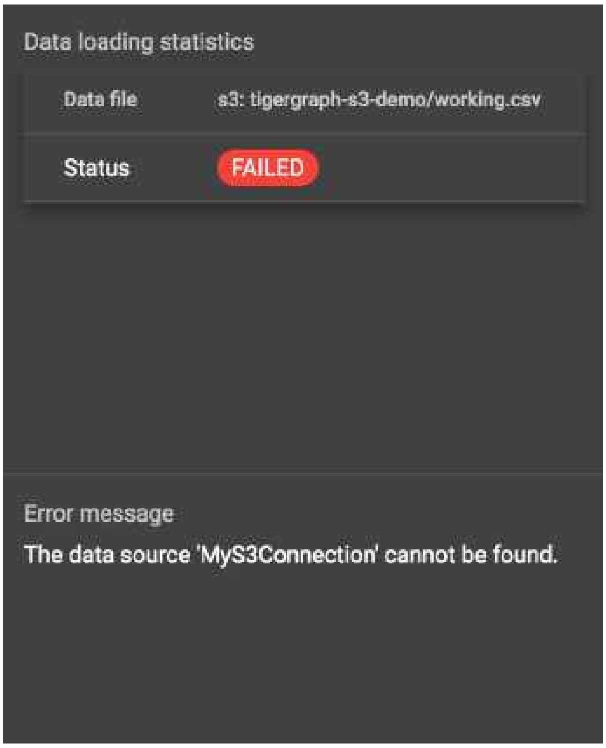
The area chart in the middle shows the real-time loading speed (lines per second) for this data file.

The pie chart at the bottom shows the distribution of data lines, among three categories:

- Loaded lines
- Missing token lines (the lines contain fewer tokens than required by the data mapping)
- Oversize lines (some tokens are too large)

> The number of loaded lines doesn't mean all these lines are successfully loaded. Some issues during Data Mapping (like mapping a non-numeric column to an integer attribute) or because of dirty data may cause some of these lines not to be loaded.

If data file loading encounters any issues and gets an error message, the error message will be shown at the bottom:



Clear Graph Data

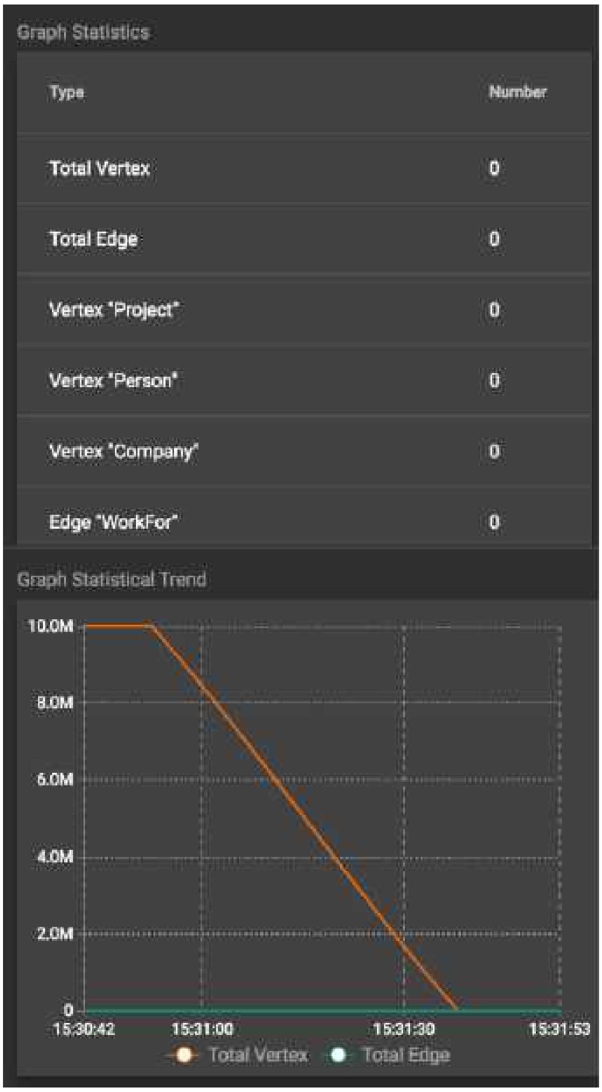
Click on the "clear graph data" button on the toolbar to clear the graph data. This operation will take approximately 1 minute or more, depending on the size of your graph and the hardware.

Caution: Clear Graph Data deletes all data from your database. The schema and queries will remain. This deletion is irreversible. Please confirm the impact before you proceed with clearing graph data operation.

Tip: Only users with superuser role can clear graph. You can consider assigning other roles to your team to avoid accidental data deletion.

Tip: If you clear graph data by accident, you can reload the data into the database by clicking on the "start/resume loading" button on the toolbar. The data files are still in the filesystem, as long as you do not deliberately delete the data files from the filesystem.

After the clear operation, the graph vertex and edge number statistics will both drop to 0.



After data has been loaded, you can go to the Explore Graph or Write Queries pages.

Explore Graph

After data has been loaded, the Explore Graph page allows you to explore the graph in the following ways:

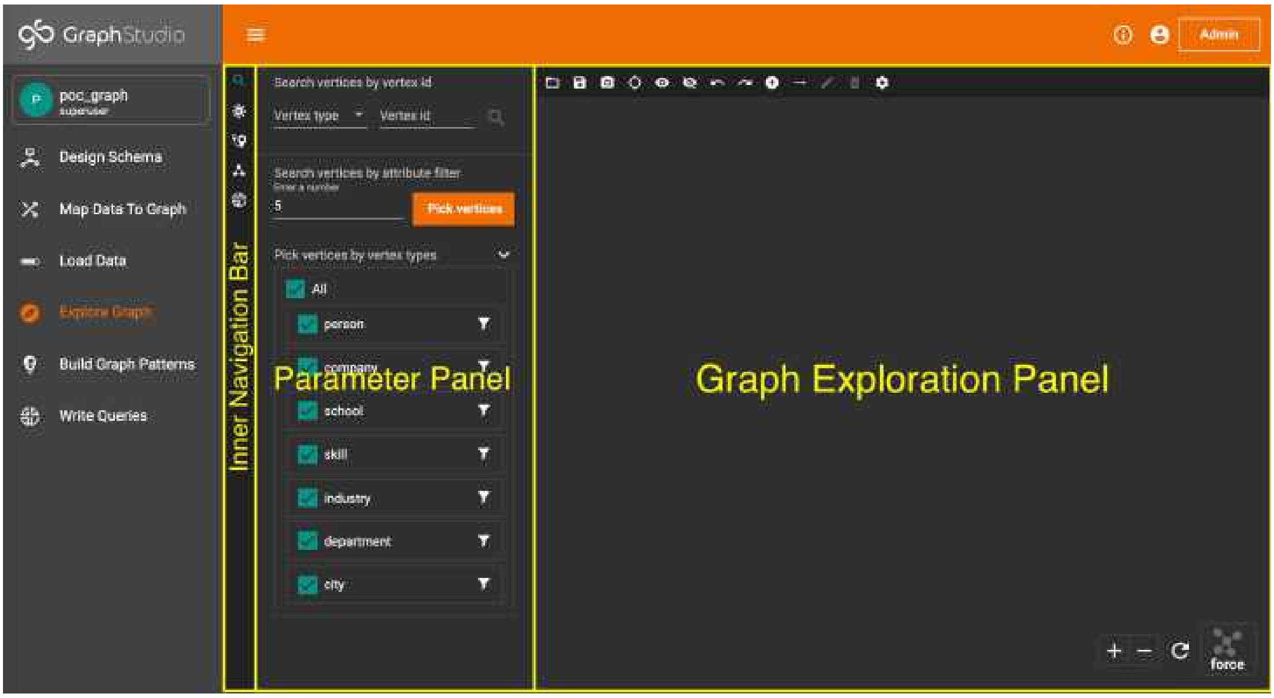
- Search for vertices in a graph
- Discover nearby vertices
- Find paths between vertices.
- Run GSQL queries

The Graph Exploration panel also provides you with many ways to modify and augment the visualization of your exploration results, as well as the ability to add vertices and edges to the graph:

- Graph Exploration Panel
- Augment Visualization Result

Page layout

The Explore Graph page is vertically divided into three parts, from left to right:



The Inner Navigation Bar

The menu options, from top to bottom, are the following:

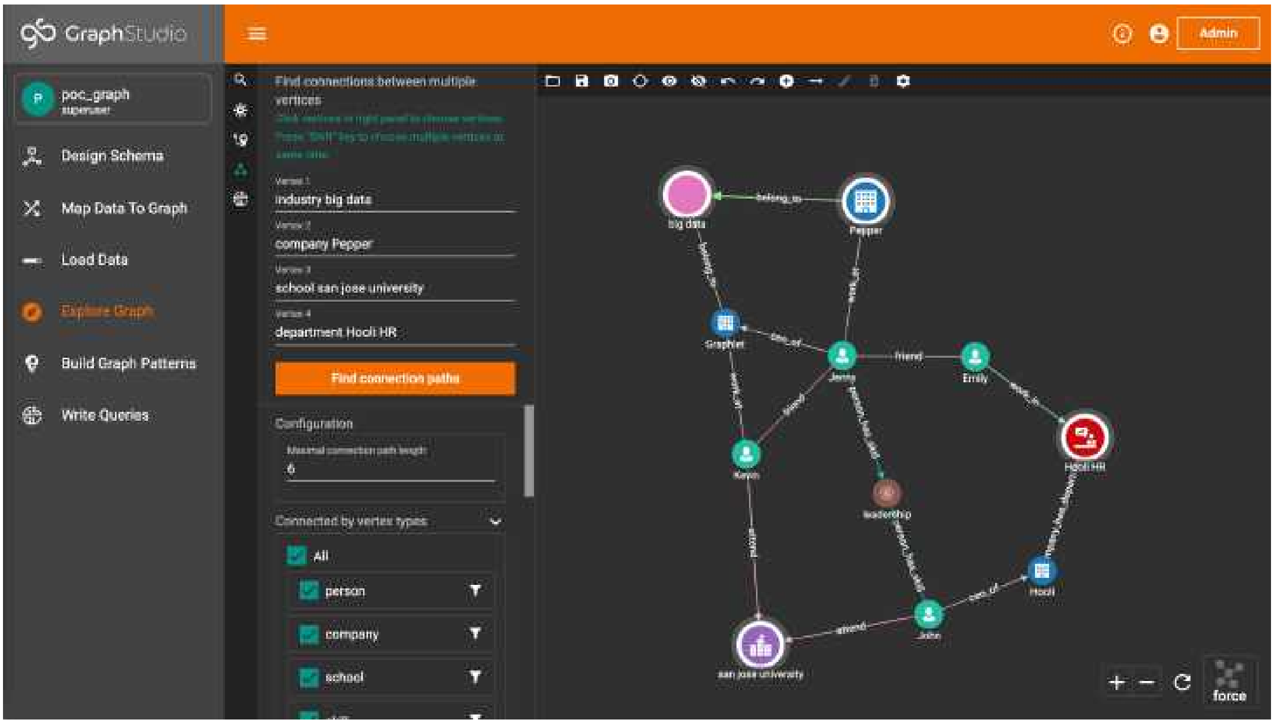
Menu option	Functionality
	Search vertices: select specific vertices with conditions.
	Expand from vertices: find neighborhood of the specified vertices.
	Find paths: find paths between the selected source vertex and target vertex.
	Find connections: find connecting paths between a set of vertices.
	Run queries: run installed GSQL queries.

The Parameter Panel

Set filters, conditions, and other parameters for the selected option from the Inner Navigation Bar.

The Graph Exploration Panel

The exploration result is displayed in this panel. Below is an example of an exploration result:



You can adjust the results display, take a snapshot of the display, and modify selected data objects in the result.

The menu buttons, from left to right, are the following:

- **Open exploration history**: Open a previously saved graph exploration result.
- **Save exploration**: Save the current visualization result.

> If the graph schema is modified after the exploration result is saved, the result cannot be opened anymore.

- **Save screenshot**: Save the current visualization result as a png file.
- **Locate vertices in result**: Search the exploration result by vertex id or attribute value.
- **Only show selections**: First select one or more objects. Clicking the button will hide all the objects which are not selected.
- **Hide**: First select one or more objects. Clicking the button will hide the selected vertices and edges (or all if none is selected).

- **Undo**: Undo the last change to the visualization result set (that is, changes to which objects are included in the result set).
- **Redo**: Redo the most recent undone change to the visualization result set (that is, changes to which objects are included in the result set).

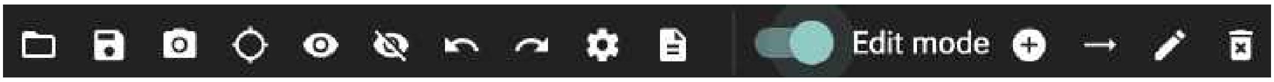
> Database changes (adding or deleting vertices/edges, editing attributes) cannot be undone with the Undo feature.
> Also, Undo/Redo do not include layout and display change (e.g., positioning of objects and display of attributes).

- **Add new vertex**: Add a new vertex into the visualization result **as well as to the graph database**.
- **Add new edge**: Add a new edge into the visualization result **as well as to the graph database**.
- **Edit attributes**: Change the attributes of the selected object in the visualization result **as well as the graph database**.
- **Delete selected elements**: Delete the selected elements from the visualization result **as well as the graph database**.
- **Change settings**: Select which attribute values to display with each vertex or edge type. Enable/disable popup display of all attributes when the cursor hovers over a vertex or edge.

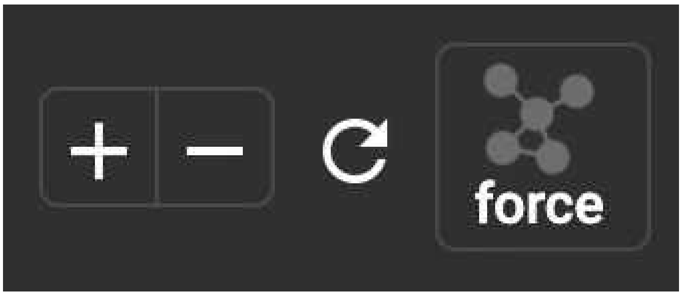
The Parameter Panel can be hidden by clicking its corresponding button in the Explore Graph Menu.

Graph Exploration Panel

The Graph Exploration panel displays the result of your Graph Exploration actions. After you have a subgraph displayed in the Graph Exploration Panel, you can use the buttons in the Explorer View Menu or the options in the bottom right corner to customize the display. Additionally, you can make modifications to the graph database itself in edit mode.



Explorer View Menu

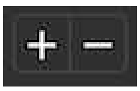


Options in the bottom right corner

Customize visualization

The Graph Exploration panel provides you with multiple ways of customizing the visualization of your graph exploration results.

Zoom in/out

Although you can use a mouse scroller to zoom in and zoom out the graph exploration result, you can also use the two buttons at the bottom-right corner to do that: 

Refresh result

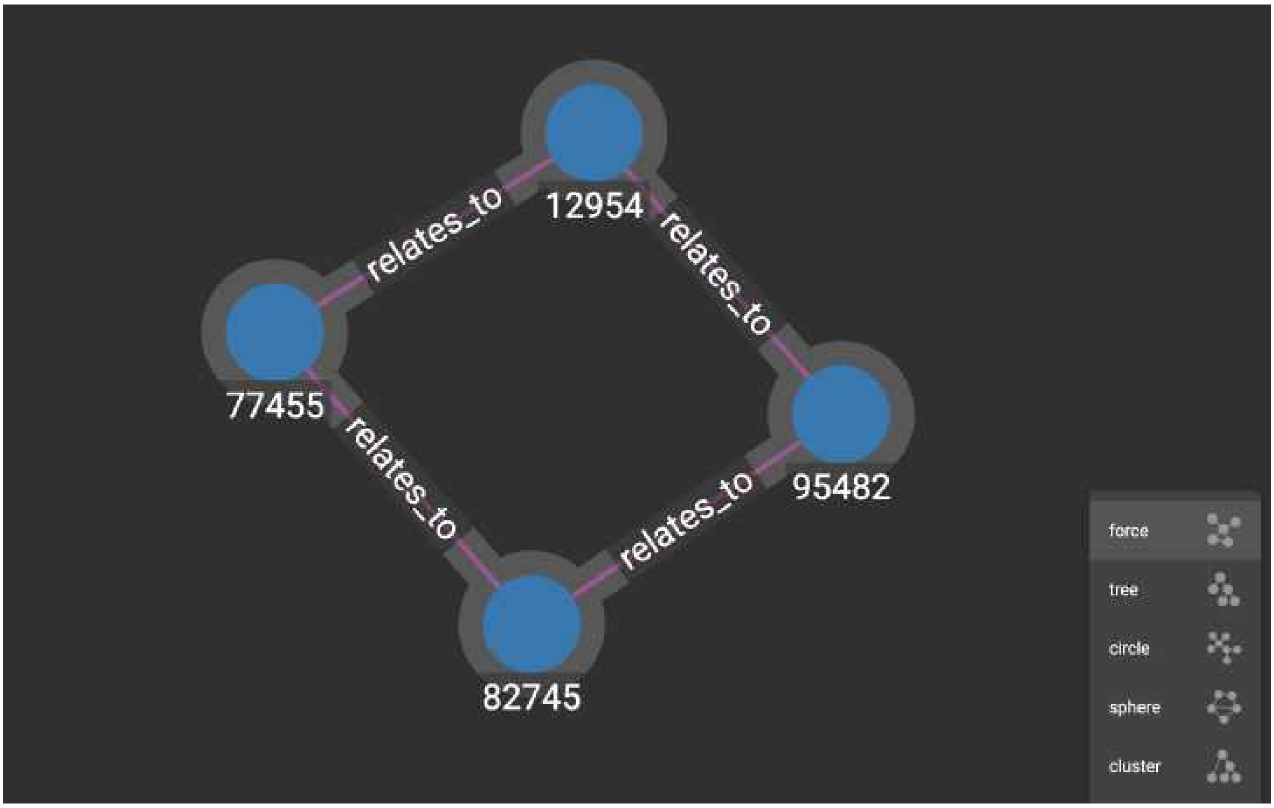
The graph exploration result only reflects a snapshot of the graph data. If the data is changed due to CRUD operations (maybe in another session), the snapshot is outdated.

You can click the refresh button  to get all vertices and edges synced with the graph database. All vertices and edges that already got removed will be removed from the exploration panel, and all the attribute values will also get updated.

Change layout

You can click the change layout button 

to change layout:



Locate vertex

The Locate Vertex In Result feature  searches for and then zooms in on vertices that match the given value for ID and/or attribute. For example, if you type "Mary" in the Locate Vertices in Result popup window, and have both of the checkboxes selected, then this feature will look for any vertices where "Mary" is an exact match for either the ID or any of the attribute values. Those vertices will be selected (and all other objects will be unselected). The display will zoom in to focus on the selected objects.

The vertices with the matching ID or attributes will be selected:

Show selected vertices and edges

Click the Show Selections button to hide all the vertices and edges which are not currently selected. However, if the two endpoints of an edge are selected, the edge will be selected as well. Also, if nothing is selected, nothing will be hidden.

Hide vertices and edges

Click the Hide button to hide the currently selected vertices and edges. If nothing is selected, all vertices and edges in the Graph Exploration Panel will be hidden.

Undo and redo

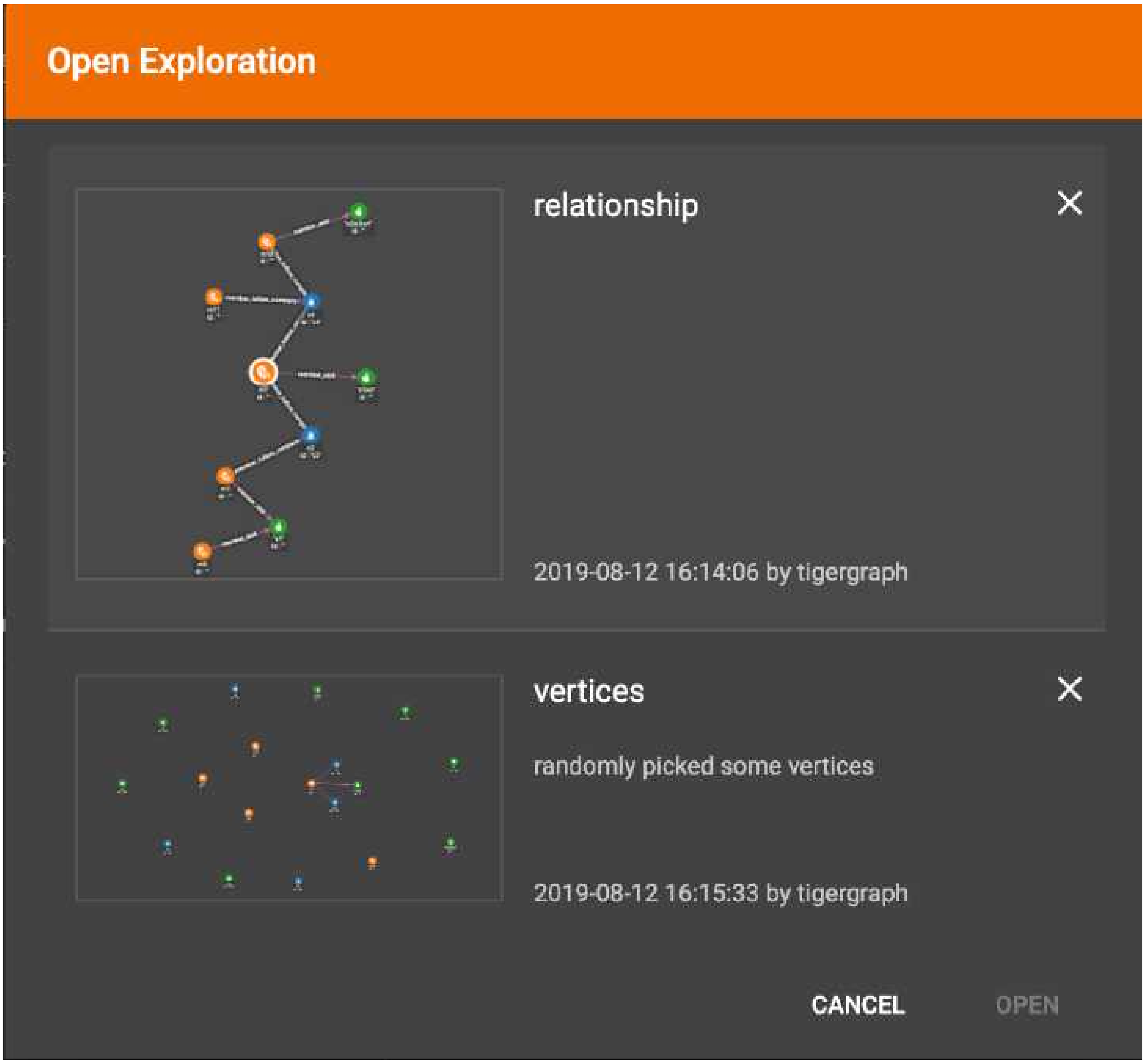
The Explore Graph page records the whole history of the current session's changes to the visualization result set. Click the Undo and the Redo buttons to go back or forward in the history.

> Database changes (adding or deleting vertices/edges, editing attributes) cannot be undone with the Undo feature.
> Also, Undo/Redo do not include layout and display changes (e.g., positioning of objects and display of attributes).

Save and open graph exploration result

When you find something interesting during exploration and want to save the result as a picture, you can click the Save Exploration button. In the popup window, you can give the result a file name and an optional description, then click Save:

In the future, you can open a previously saved exploration result by clicking the Open Exploration History button  and choose one result from the list:

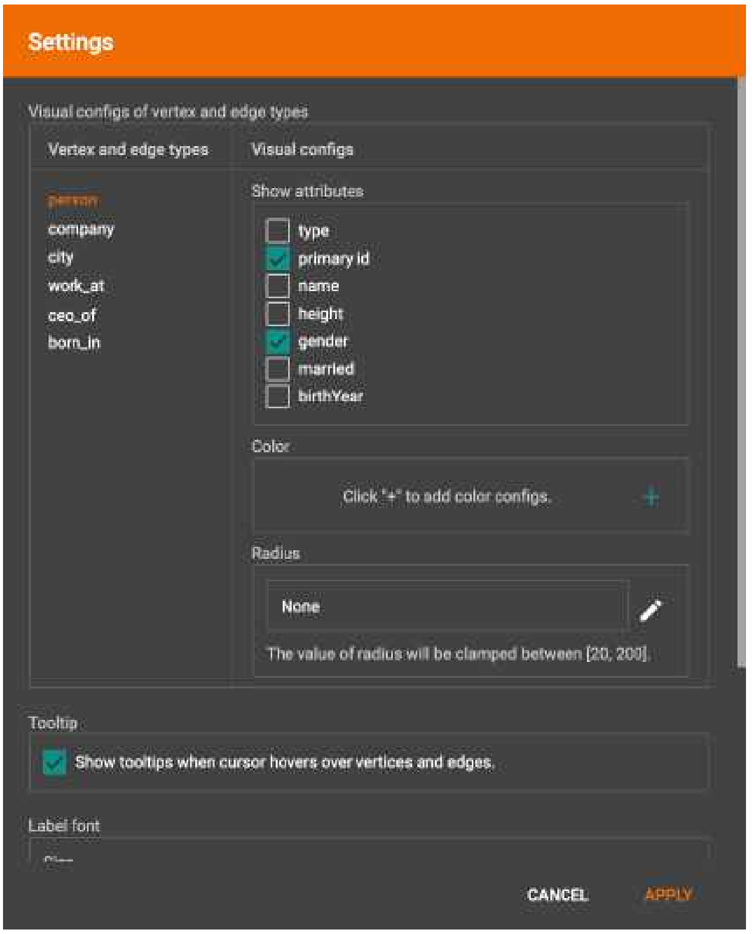


Save screenshot

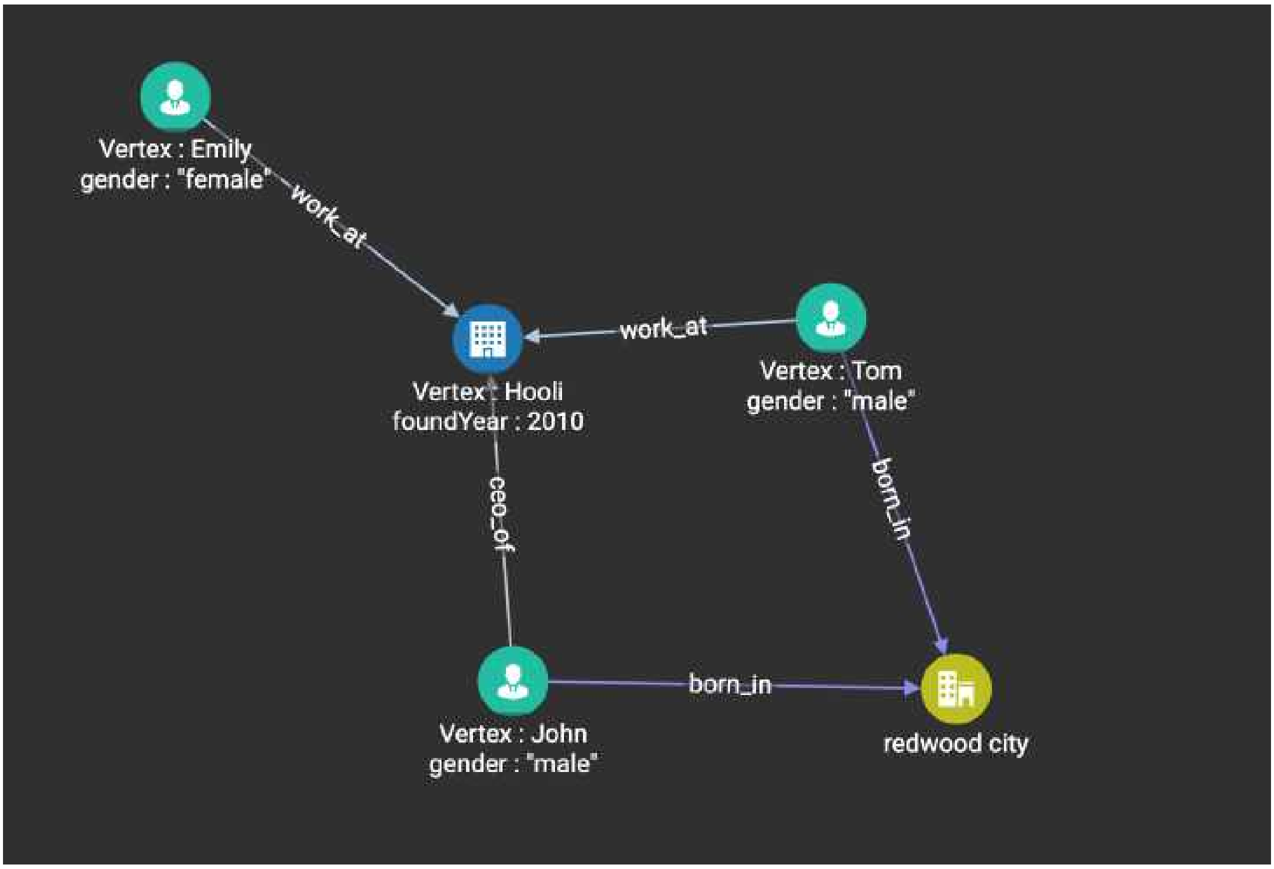
When you find something interesting during exploration and want to save the result as a picture, you can click the Save Screenshot button. The exploration result will be saved as a PNG picture to your local file system.

Change settings

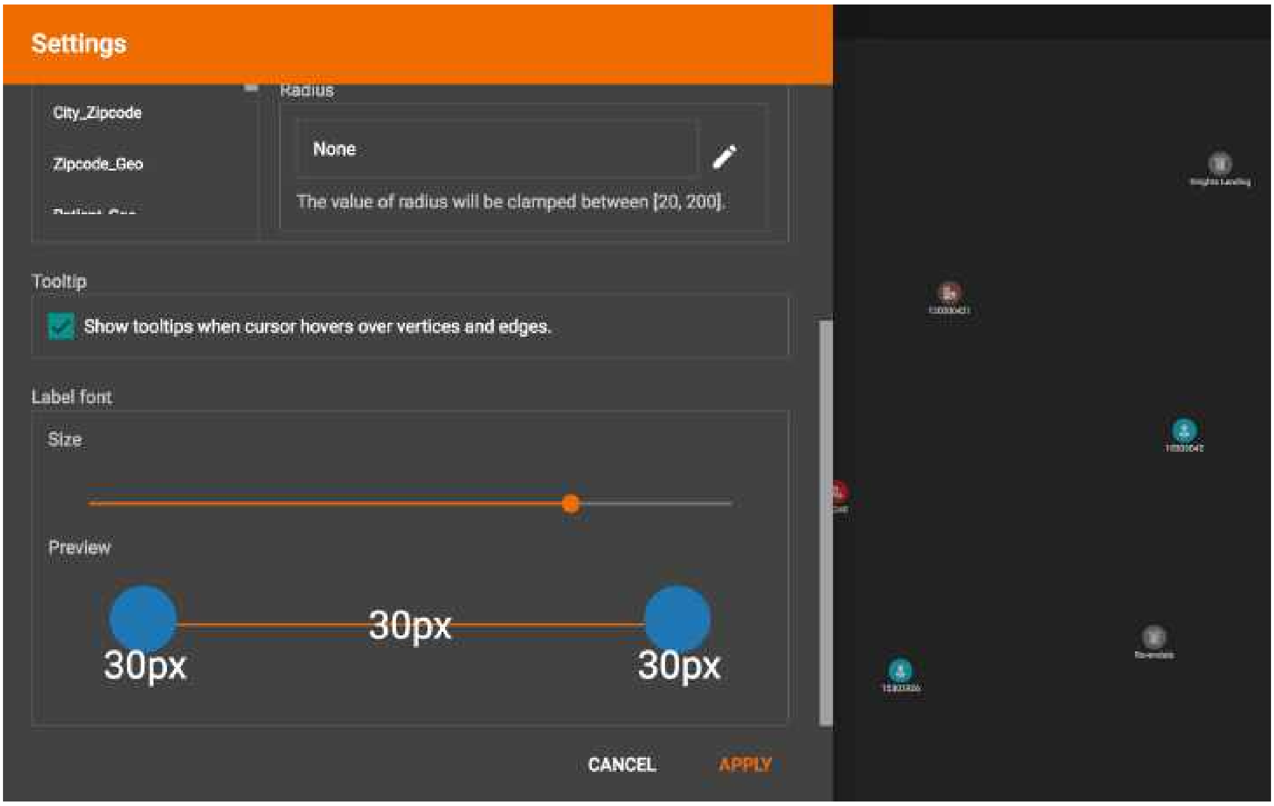
To change graph exploration settings by clicking Settings button. Currently you can select what attributes to show for each vertex type and edge type, and set whether to show an object's detailed information in a popup tooltip when the cursor hovers over it. Click Apply and the new settings will take effect.



In the example below, the ID and gender for person vertices are shown. The ID and the foundYear attributes for company vertices are shown.



You can also configure the text font size of vertex and edge labels and properties.

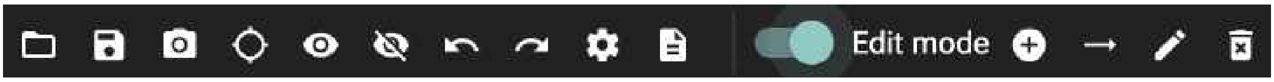


Other than the above, you can also configure vertex and edge size and color to augment the visualization in settings. See Augment Visualization Result.

Edit graph

Besides customizing the view of your graph exploration results, you can also enable the edit mode and and make edits to the graph itself by adding or removing vertices and edges, as well as changing their attributes.

In order to perform any of the actions in this section, you need to enable edit mode by clicking on the toggle next to Edit mode. Once you are in the edit mode, the four buttons that allow you to edit the graph will appear on the right.



Explorer View Menu

Add new vertex

In edit mode, click the Add New Vertex button  to add a new vertex to the graph database. The Add New Vertex window will pop up. Choose a vertex type and then fill in values for the ID and the attributes. Click ADD and the vertex will be inserted into the TigerGraph database. It will also be shown in the Graph Exploration Panel.

> If you provide a vertex ID that is already used, GraphStudio will ask you whether you want to overwrite the existing vertex. If you say no, then it will not add or update anything.

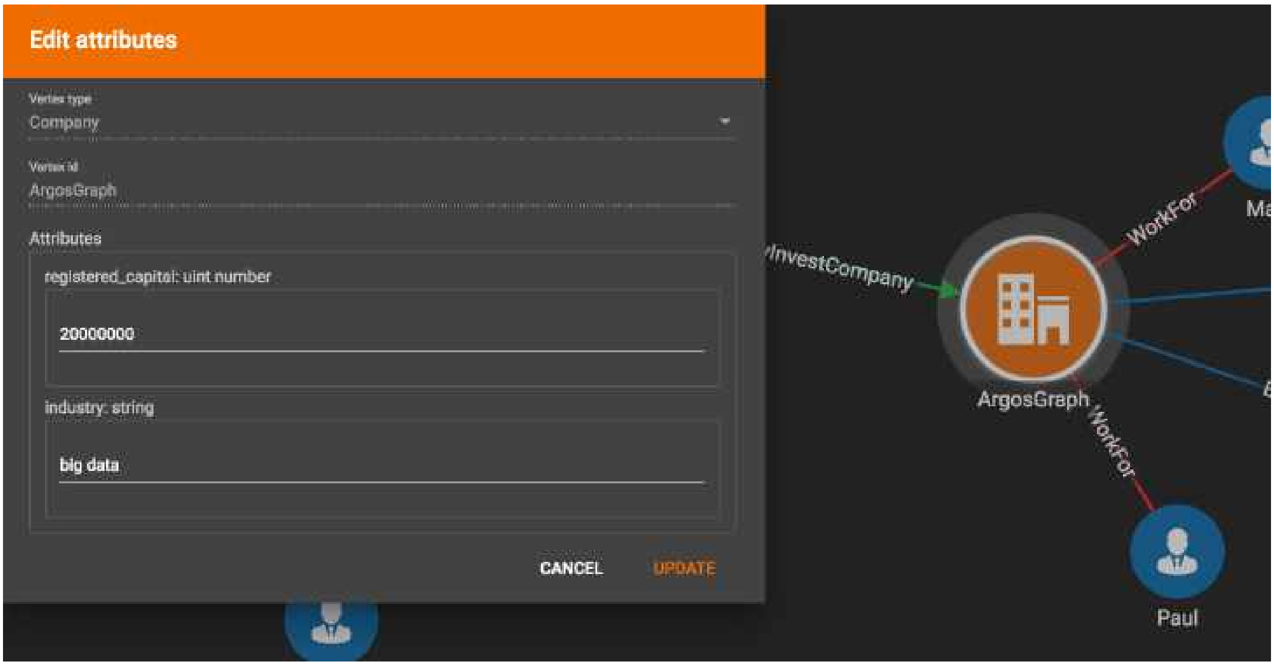
Add new edge

In edit mode, click the Add New Edge button to add a new edge to the graph database. Next, click the source vertex of the edge in the Graph Exploration Panel, and then click the target vertex of the edge. Then the Add New Edge panel will pop up. Choose the edge type from the dropdown menu. Only types that match the two vertices you selected are shown. (It is possible that there are no eligible edge types). Fill in values for attributes and click ADD. Your new edge will be inserted into the TigerGraph database. It will also be shown in the Graph Exploration Panel.

If you select an edge type that already exists between the two vertices, GraphStudio will ask if you want to overwrite the existing edge. If you say no, nothing will be added or updated. The current TigerGraph system does not support having multiple edges of the same type between two specific vertices.

Edit attributes

To edit the attributes of one vertex or edge, select one object and then click the Edit Attributes button in edit mode. The edit attributes panel will pop up.



When you finish editing, click the Update button to apply the change.

Delete vertices and edges

To delete vertices or edges, select the objects you want to delete, and click the Delete Selected Elements button  in edit mode.

"Delete" permanently removes data from the graph database. Deleted vertices and edges cannot be restored with Undo. To restore them, you must manually add them back.

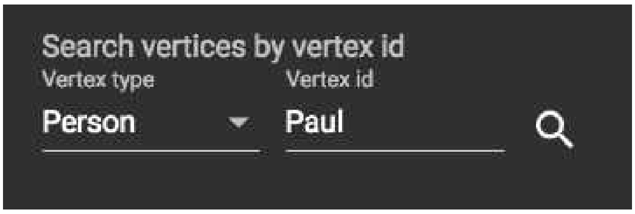
If you delete a vertex, all of its outgoing and incoming edges will also be deleted

Search for Vertices

The first button in the Explore Graph Menu is the "Search Vertices" option. This option lets you select an initial set of vertices for your exploration. It is also the default option when you first enter the Explore Graph page. Clicking the button again will hide the Parameter Panel to increase space for the Graph Exploration Panel.

Search for vertices by ID

Choose a vertex type from the Vertex type dropdown list, and enter the vertex id in the **Vertex id** input box, then click the Search button. If there is one vertex that matches the vertex type and ID, it will be shown in the Graph Exploration panel.

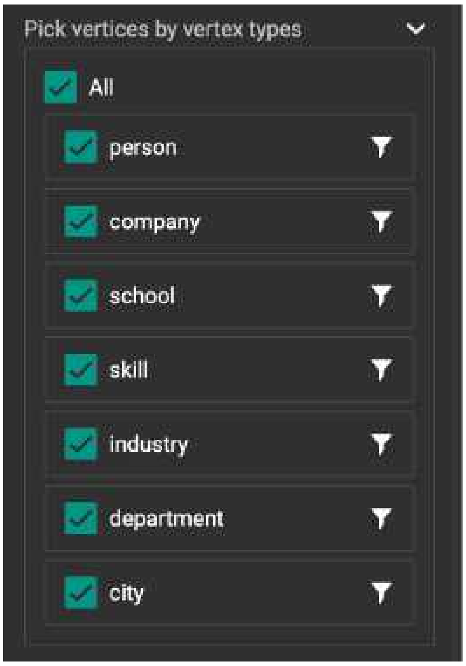


Let GraphStudio pick vertices

If you don't have a particular vertex ID in mind, you can have GraphStudio pick some vertices for you. In the Parameter Panel, enter a number of vertices to pick, and click on the Pick vertices button

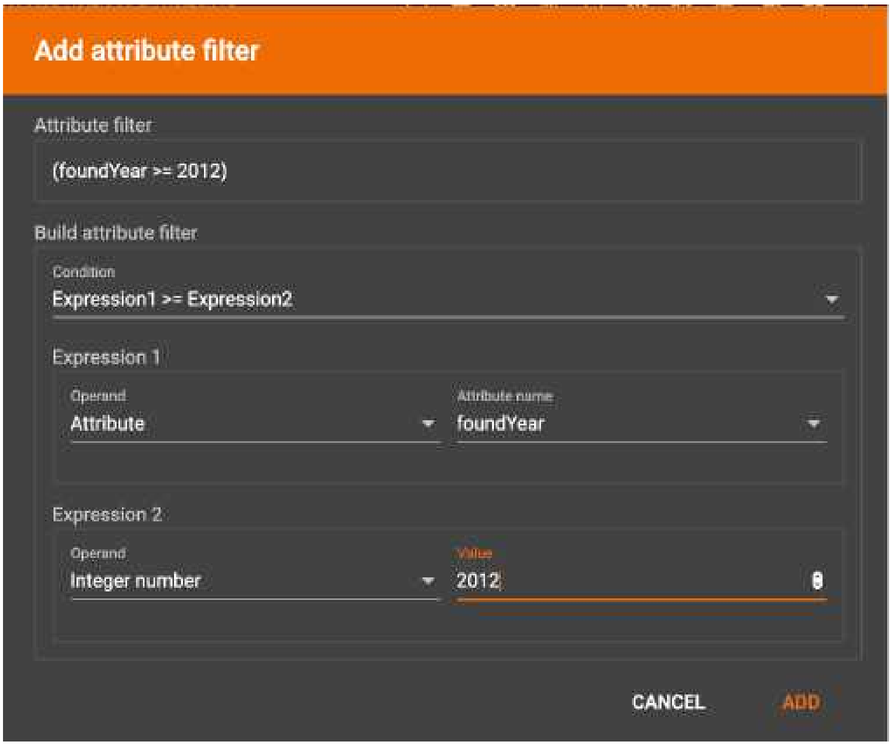
. The explorer will pick this number of vertices for each vertex type included in your search.

The Configuration section in the Parameter Panel specifies which types of vertices you want to include in your selection. By default, all vertex types are selected. Uncheck some boxes if you want to narrow your selection.

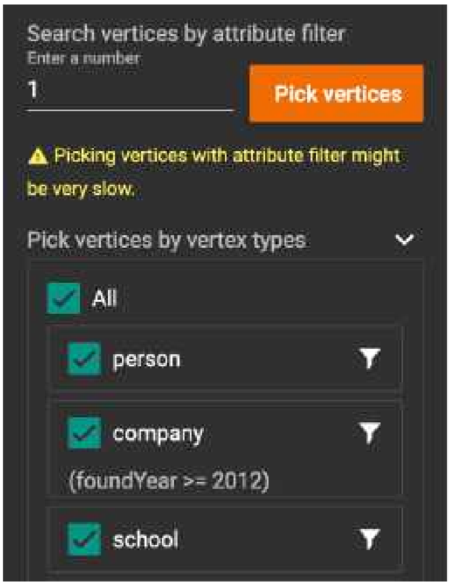


Search with attribute filters

You can control vertex search in finer granularity by creating attribute filters. Click the filter button to the right of any vertex type. In the pop-up window, you can create a condition involving attributes of the vertex type. The user experience is the same as creating data filters when you do data mapping. Here is an example attribute filter for searching company vertices with `foundYear >= 2012`:



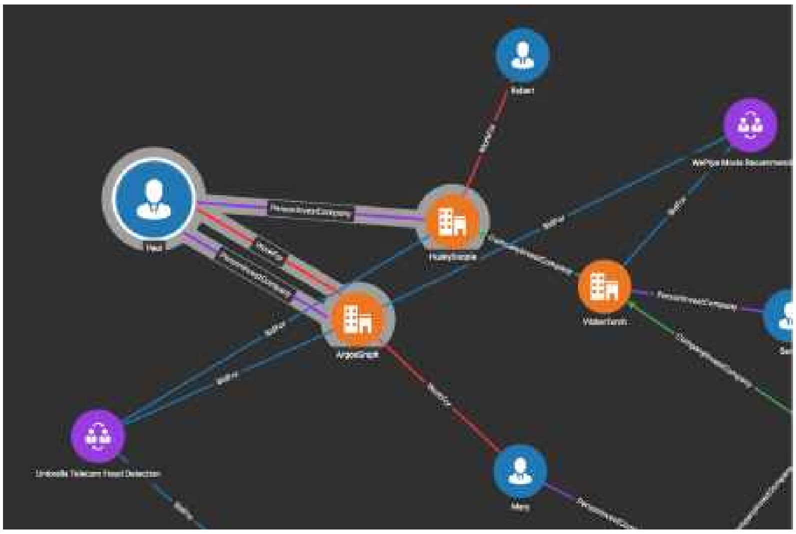
Click ADD, then the filter condition is shown below company vertex type:



Click the Pick vertices button Pick vertices again, TigerGraph will search for up to 1 company vertex with a `foundYear >= 2012`.

> If your graph contains a large number of vertices, searching vertices with attribute filters can be extremely slow.

If you keep exploring the graph on the Explore Graph page, the previous exploration result won't be automatically erased. Instead, your new exploration result will be merged together with the previous visualized graph. The objects from the most recent exploration will be selected (highlighted with a thick gray border) to distinguish them from the previous visualized graph.



Expand from Vertices

The second button in the Explore Graph Menu is the "Expand from vertices" option . "Expand" in this context means find 1-step or multi-step neighbors of the selected vertices. Clicking the button again will hide the Parameter Panel to increase space for the Graph Exploration Panel. To expand from vertices, you need to have at least one selected vertex in the Graph Exploration Panel. If no vertices are visible, please refer to the previous section "Search Vertices in Graph" to search for some vertices.

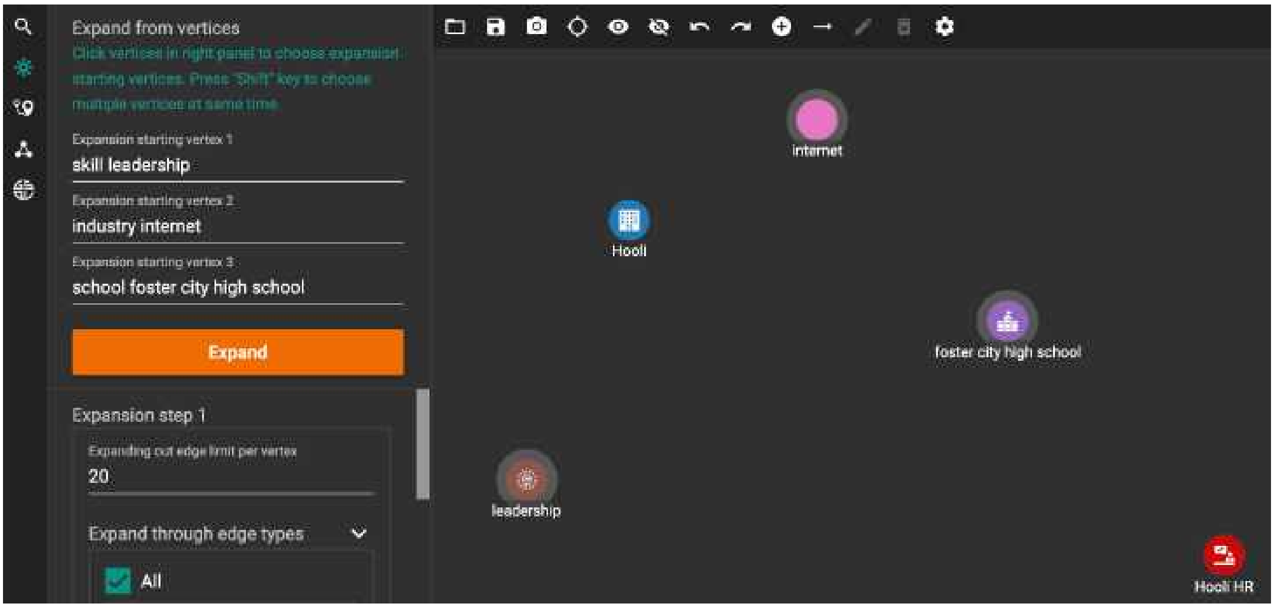
> Shortcut: double-clicking on a vertex will expand to up to 200 neighbors of that vertex.

1. Choose vertices To expand from

The standard click and shift-click behavior applies for selecting one or multiple objects:

- Click on a vertex to select it. Any previously selected objects are unselected.
- Shift-click on an unselected object to add it to the selection set.
- Shift-click on a selected object to remove it from the selection set.

A vertex that is selected has a thick gray border around it. To unselect all vertices, click on a blank area of the panel.



2. Set expansion conditions

GraphStudio lets you expand multiple steps from the target vertices, as long as the resulting number of vertices and edges does not exceed the limit for visualization (default limit is 5000 vertices and 10000 edges). The conditions for each expansion step are specified independently.

In the Parameter Panel, set the conditions for each expansion step:

- Maximum number of edges include for each vertex. The effect is that vertices with more neighbors than this limit will not have all their neighbors included in the expansion.
- Edge types and the attribute filter for each edge type to include.
- Target vertex types and the attribute filter for each vertex type to include.

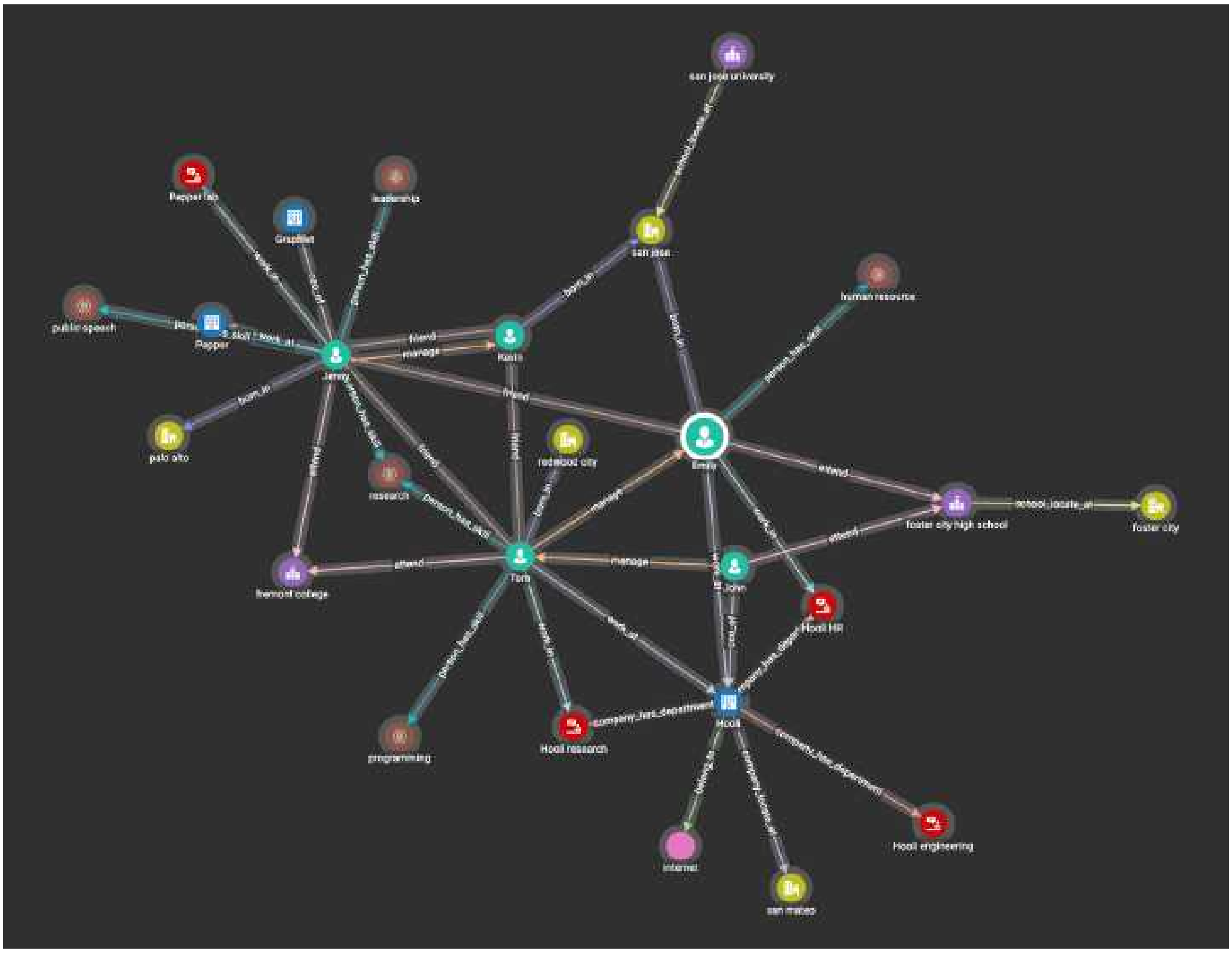
Initially, the expansion conditions panel for only one expansion step is shown. Click "Add Expansion Step" to add more expansion steps.

Similarly, you can remove expansion steps by clicking the "Remove Expansion Step" button.

3. Expand

After setting the conditions for each expansion step, click on the "Expand" button 

to perform the expansion. The Graph Exploration Panel will be updated to include the expansion result. The expansion starting vertices will be highlighted with a white border. Here is a sample two-step expansion starting from 1 vertex:



Find Paths between Vertices

You can use the Find Paths and the Find Connections option to find paths between two or more vertices:

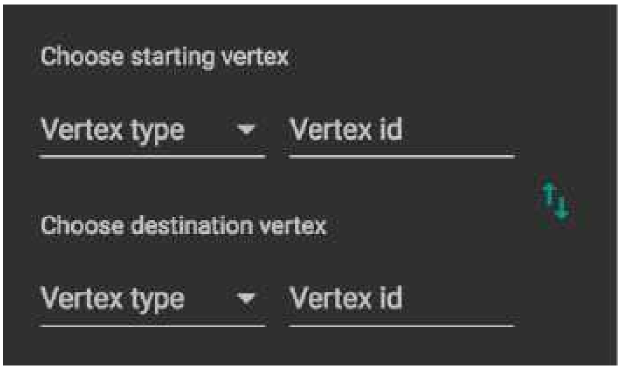
- Find paths: find path between two vertices
- Find connections: find paths connecting more than 2 vertices

Find paths between two vertices

The third button in the Explore Graph Menu is the "Find paths" option. This option finds paths between two vertices with your specified conditions. Clicking the button again will hide the Parameter Panel.

Choose Starting Vertex and Destination Vertex

The top section of the Parameter Panel asks for your desired starting vertex and destination vertex.

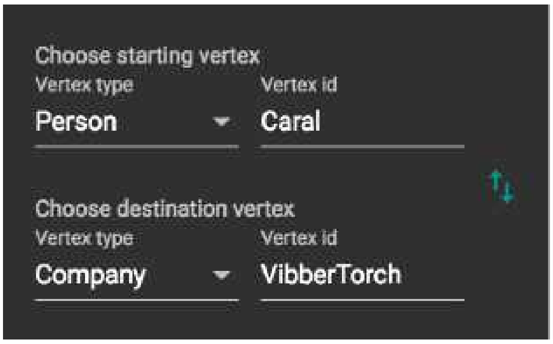


There are two ways to provide this information. Each of the two vertices can be selected by either method.

If you know the ID and vertex type for a vertex, you can choose vertex type from dropdown list and type vertex id in the input box. The vertex does not need to be currently displayed in the Graph Exploration Panel.

If the vertex you want is already displayed in the Graph Exploration Panel, a more convenient way is the following:

1. Click on the input box.
2. Click on the desired vertex in the Graph Exploration Panel. Then, GraphStudio will automatically fill in the values for you.



You can click the swap icon (two green arrows) on the right to switch the starting vertex and the destination vertex.

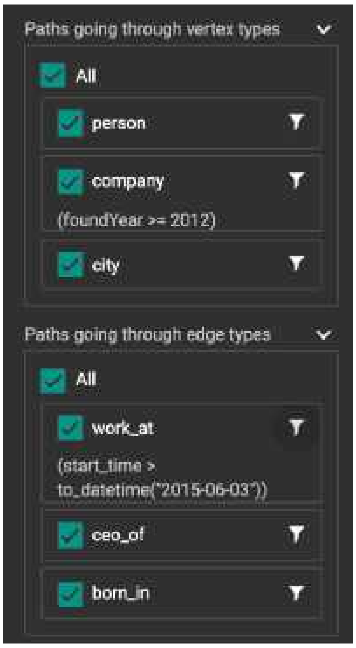
Set Conditions For Paths

GraphStudio provides three types of path searches:

1. One shortest path: search for and highlight the shortest path between the two vertices.
2. All shortest paths: search for and highlight all shortest paths between the two vertices.
3. All paths: search for and highlight all valid paths between the two vertices.

Since path-finding queries may have high computational cost if the graph is very large, a parameter is available to limit the path length.

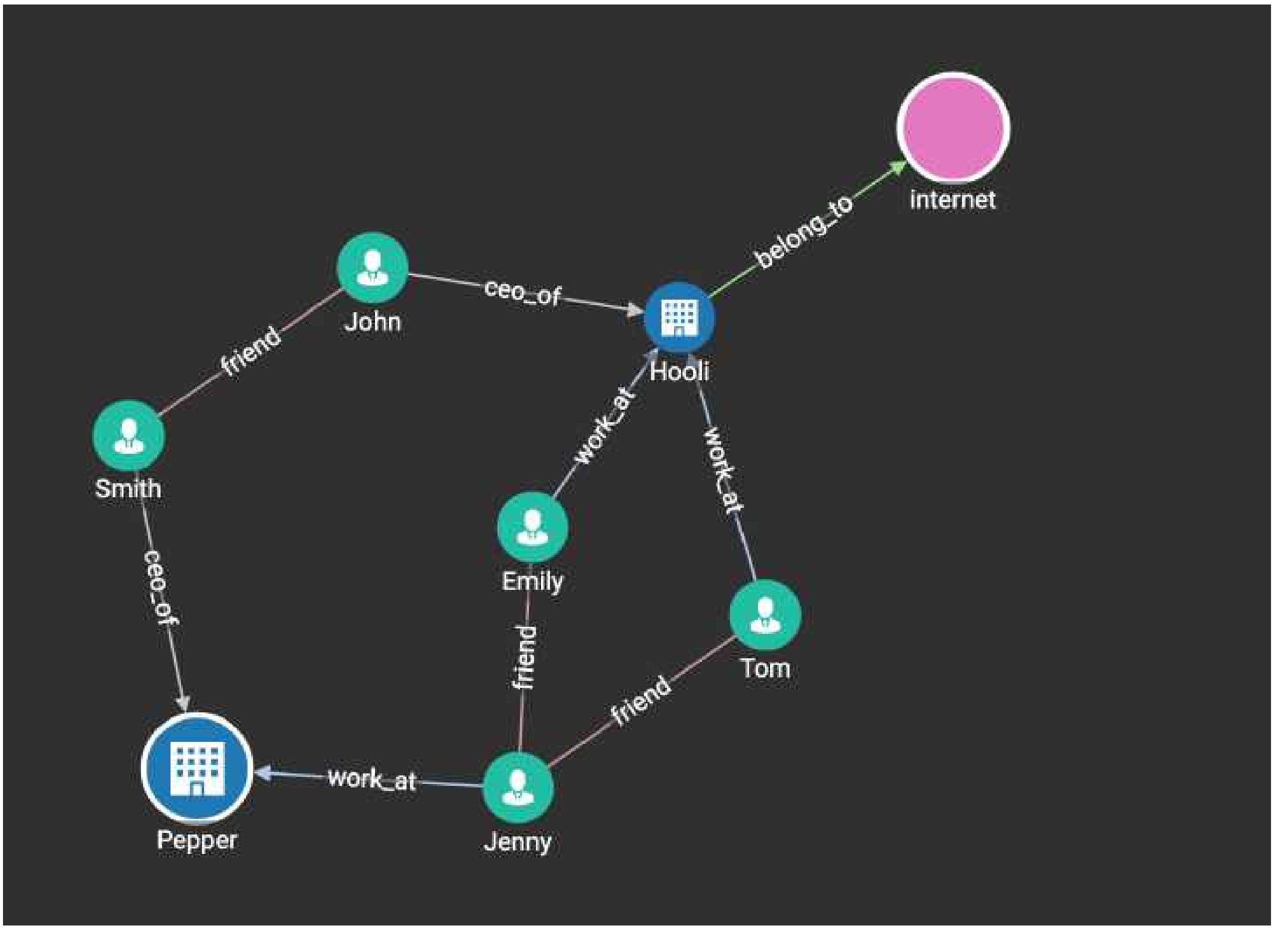
In addition to the search type and the maximal length, you can also specify the valid vertex types and edge types and their attribute conditions which may be included in the paths.



Find Paths

After selecting the endpoint vertices and setting the search conditions, click on the "Find Paths" button 

to start the search.



Find connections between multiple vertices

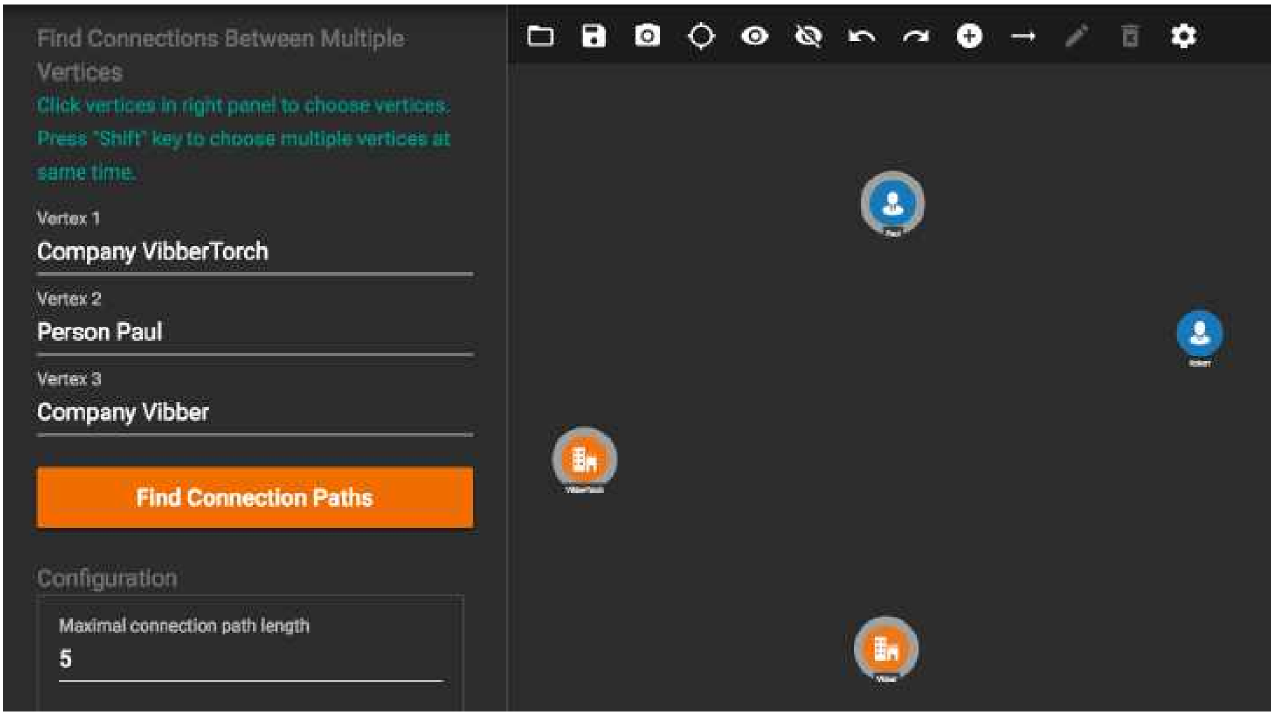
The fourth button in the Explore Graph Menu is the "Find connections" option . Given a set of starting vertices, this feature finds a "connection community" which is defined as follows:

1. For each pair of vertices in the vertex set, if there is a shortest path no longer than the maximum path length parameter, include that path in the result.
2. The final result is the union of all of these shortest paths (one path per vertex pair).

This feature is equivalent to running the "Show One Shortest Path" option for each pair of vertices in the selected set.

Choose Vertices for Finding Connections

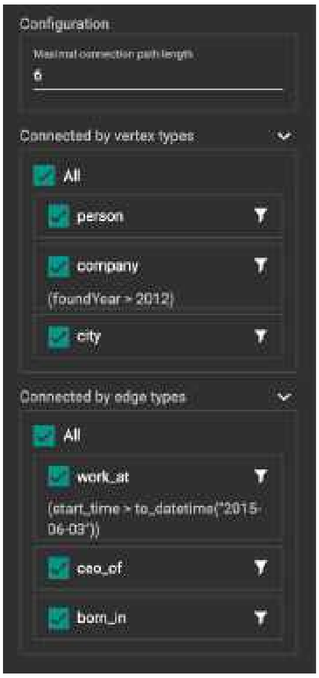
Click on a vertex to select it. Use shift-click to select more than one object. Each time you select another vertex, it will be added to the list in the Parameter Panel.



Set Conditions For Connection Finding

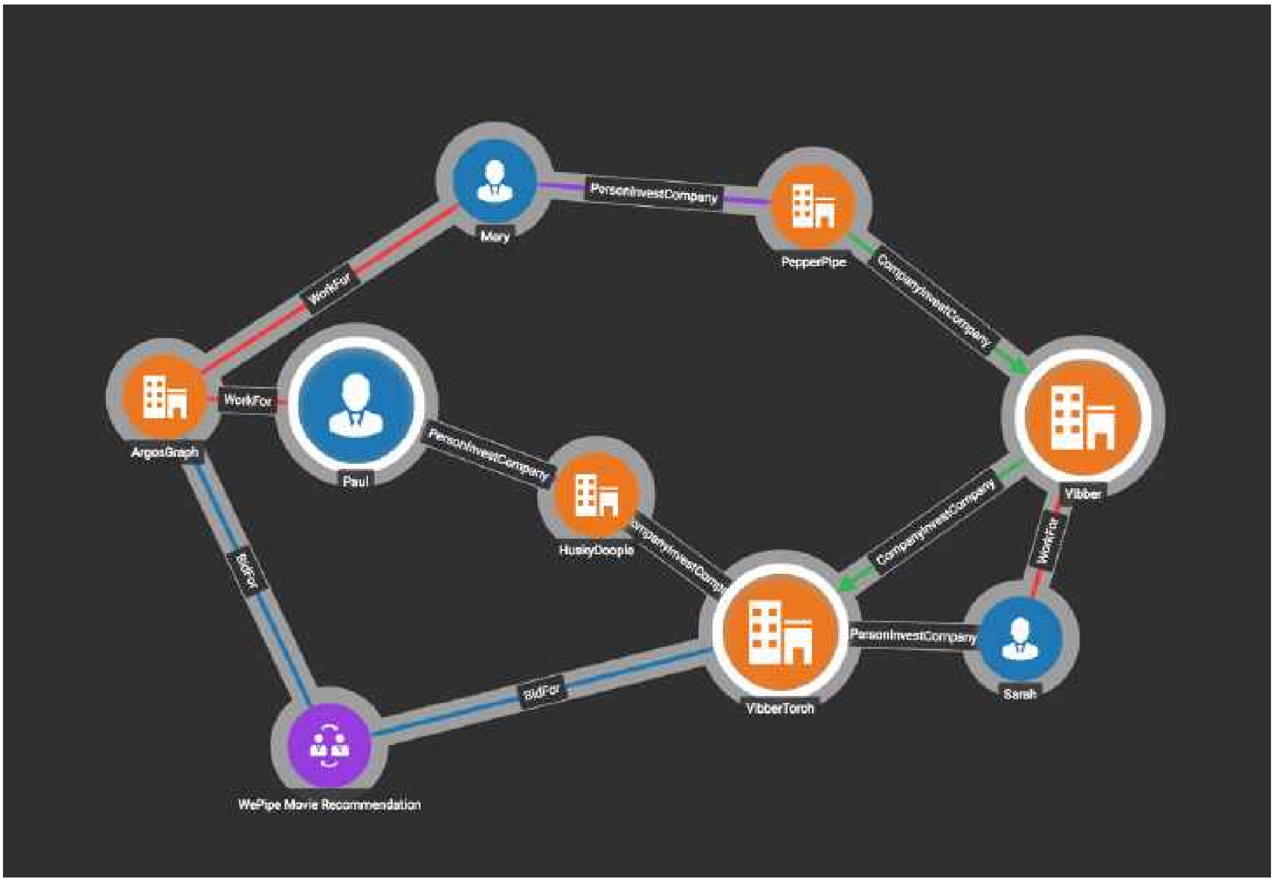
Since this query may have high computational cost if the graph is very large, a parameter is available to limit the path length.

You can also specify the valid vertex types and edge types which may be included in the connections.



Find Connections

After selecting the vertices and setting the search conditions, click on the "Find Connection Paths" button  to start the search.

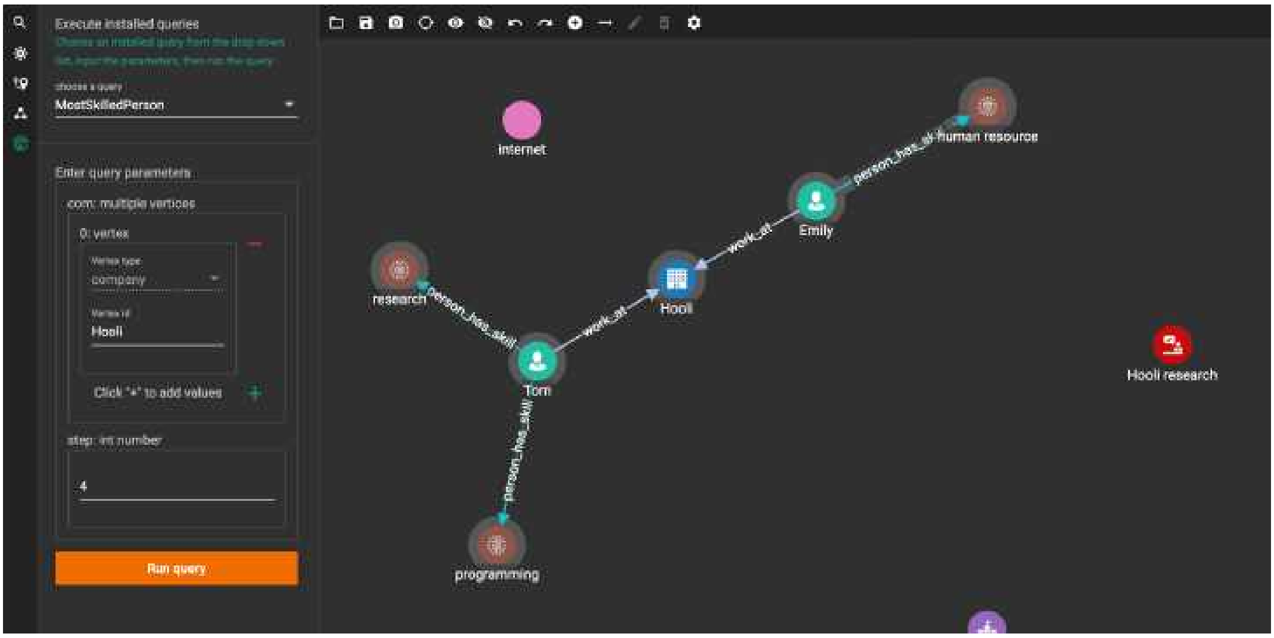


Run GSQL Queries

If you have written and installed some GSQL queries (see more at Write Queries), you can run the queries mixed with the graph exploration functionalities mentioned above.

Click the fifth button in the Explore Graph Menu, which is the "Run queries" option . In the dropdown list, choose the query you want to run. Input the parameters and click Run query button 

. The query execution result subgraph will be merged with previous graph exploration result and highlighted:



Allowing running GSQL queries mixed with other graph exploration functionalities enables better data analysis possibilities since you can refer to your previous exploration result, and keep gaining insights from your data.

Augment Visualization Result

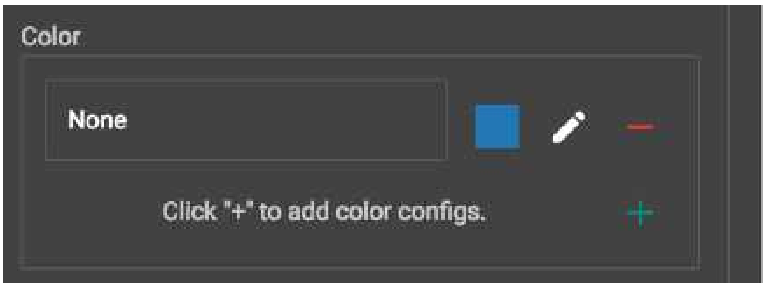
The Graph Exploration panel allows you to augment the visualization of your exploration results in the following ways:

- Set different colors according to attributes and accumulator values
- Set different vertex radius and edge thickness according to attributes and accumulator values

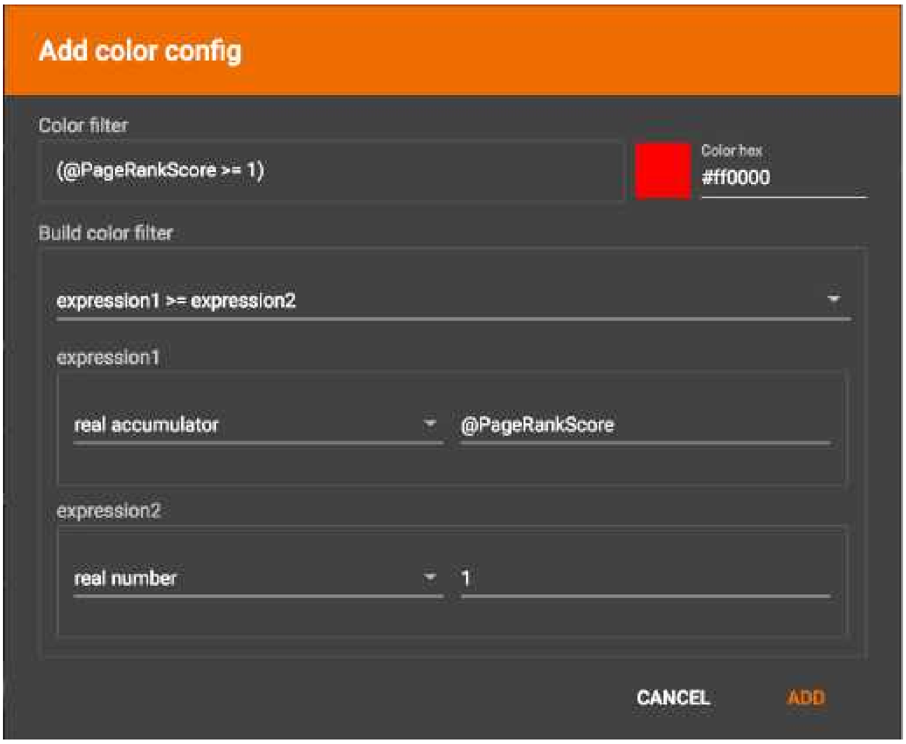
Set different colors according to attributes and accumulator values

By default, each vertex and edge is rendered as the color you selected in Schema Design page. However, if you want to emphasize some vertices and edges in your visualization result, you can set a different color for them by creating a set of conditions, and assign a different color for each condition. Then vertices and edges satisfying the conditions will be rendered as the newly assigned color.

In the Color section of Settings panel, first choose the vertex or edge type you want to set colors, then click the add button +. A new color configuration entry appears:



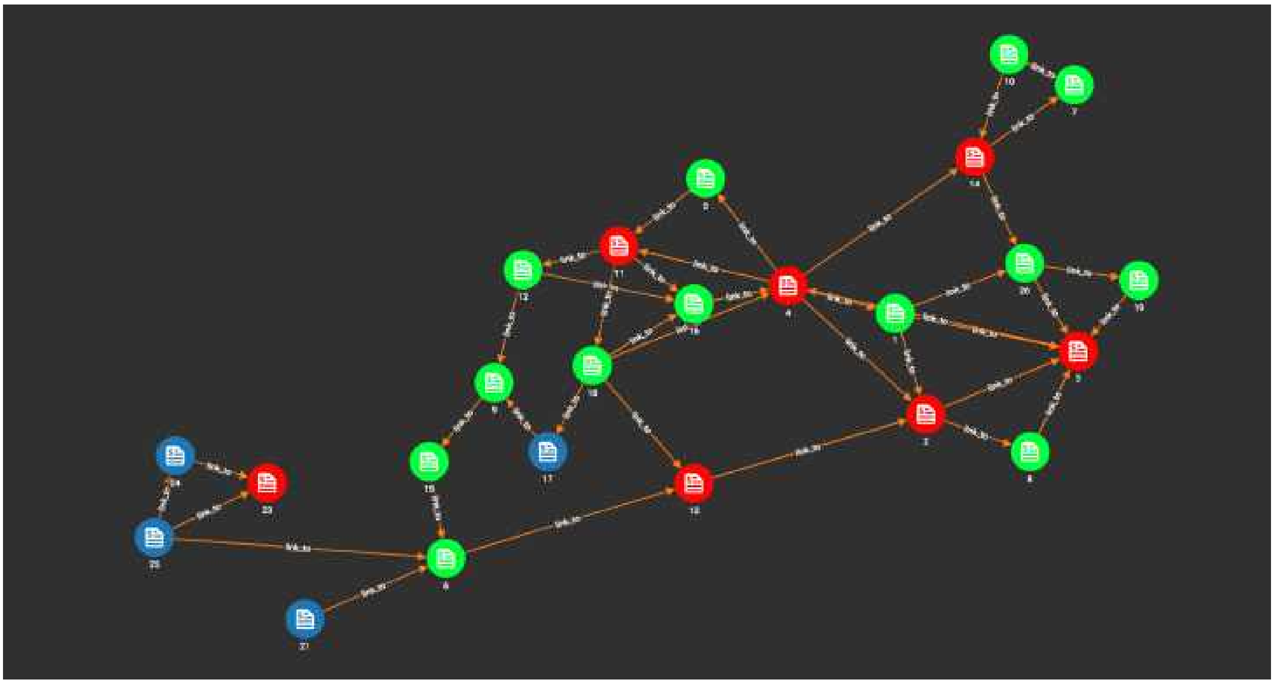
Click the Edit color config button, in the pop up window choose red color, and build a condition specifying @PageRankScore >= 1.0:



Click ADD, and the condition and updated color is shown in the Color settings section:

Similarly, you can add another color configuration that @PageRankScore between [0.5, 1) will be green. The final Color settings section will look like:

Click the APPLY button, then the different vertices will be rendered as different colors based on their page rank score ranges:



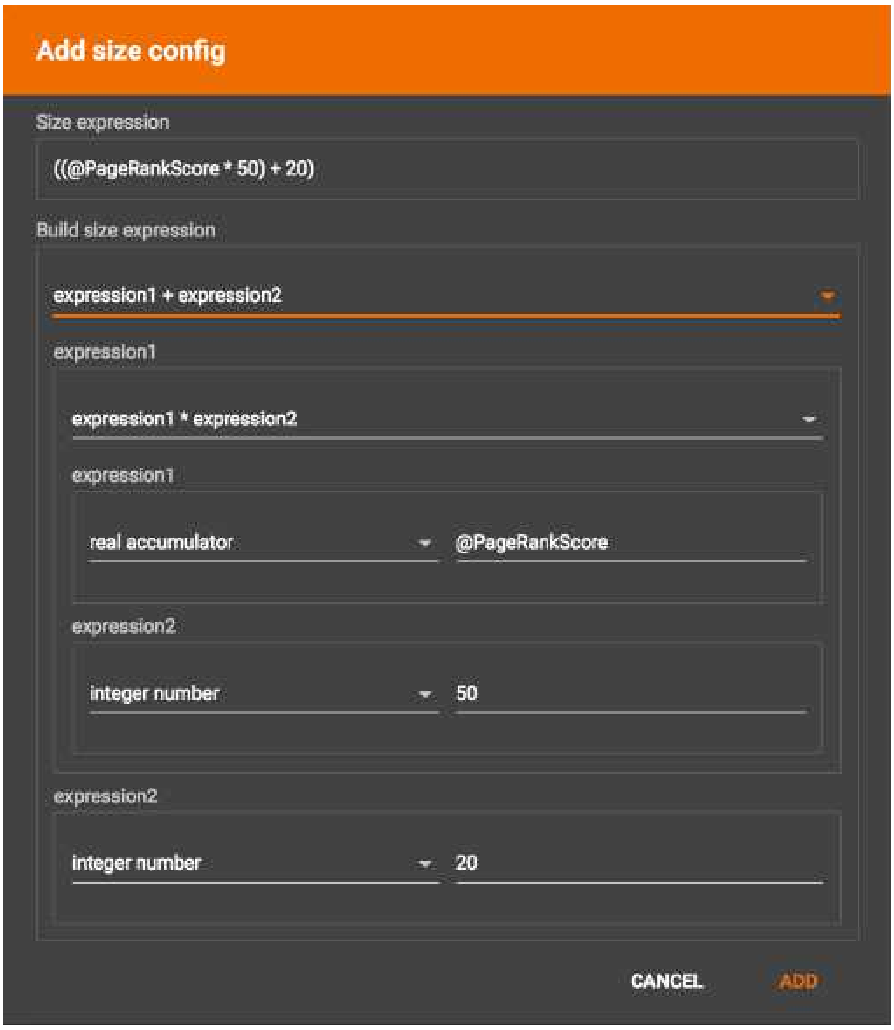
Similarly, you can change the color of edges.

If you want to cancel one color configuration, just click the remove button to the right side of that configuration.

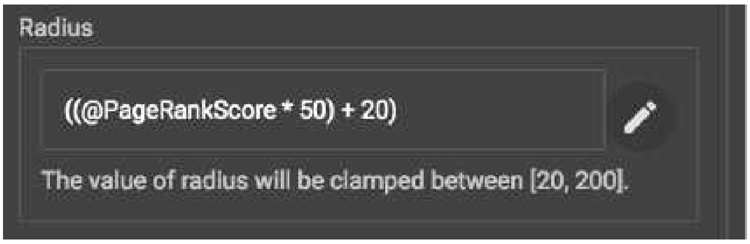
Set different vertex radius and edge thickness according to attributes and accumulator values

By default, all vertices are of radius 40, and all edges are of thickness 2. You can configure vertex radius and edge thickness according to their attributes or numeric accumulator values of GSQL query result. A classical example is page rank. You can set vertices radius proportional to their page rank values, then the importance of each vertex is visually apparent according to its size.

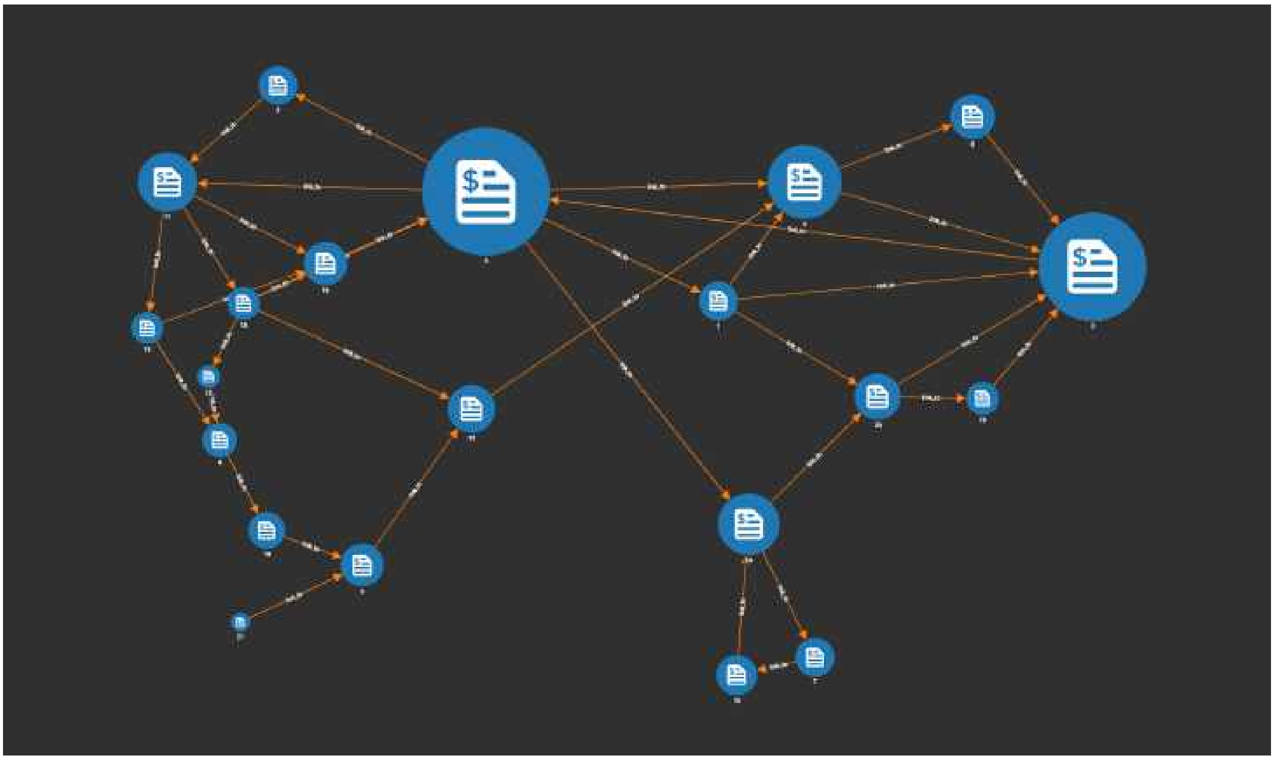
First choose the vertex type whose radius you want to configure, then click the Edit button in Radius section. In the popup window you can create the radius expression:



After click ADD button, the radius expression will be shown in Radius section:

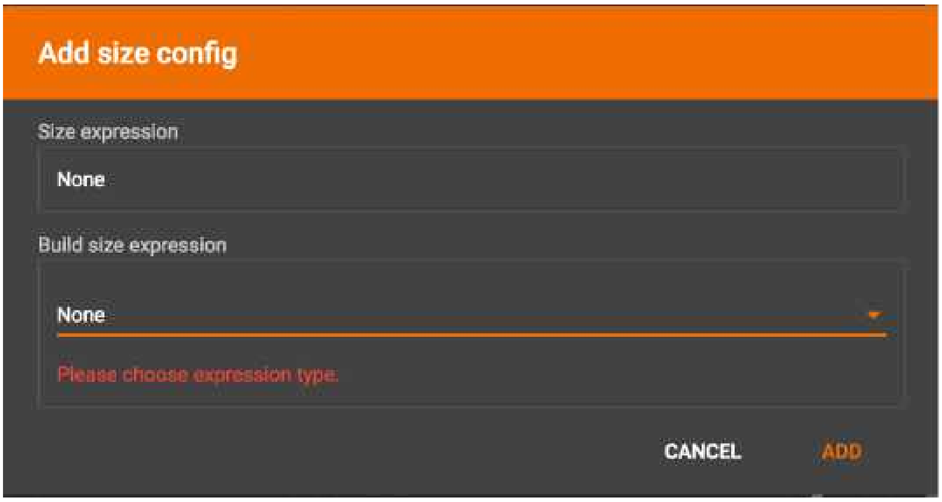


After click APPLY button, the vertices will be rendered in different size according to the expression value:



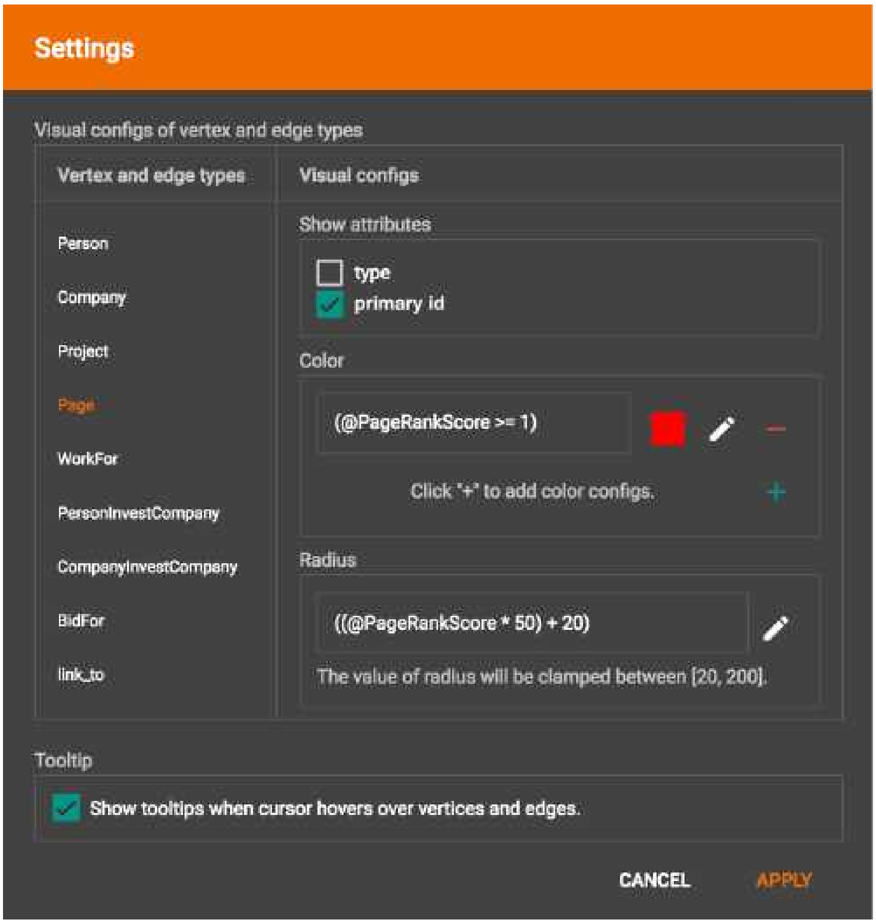
Similarly, you can set different thickness for the edges.

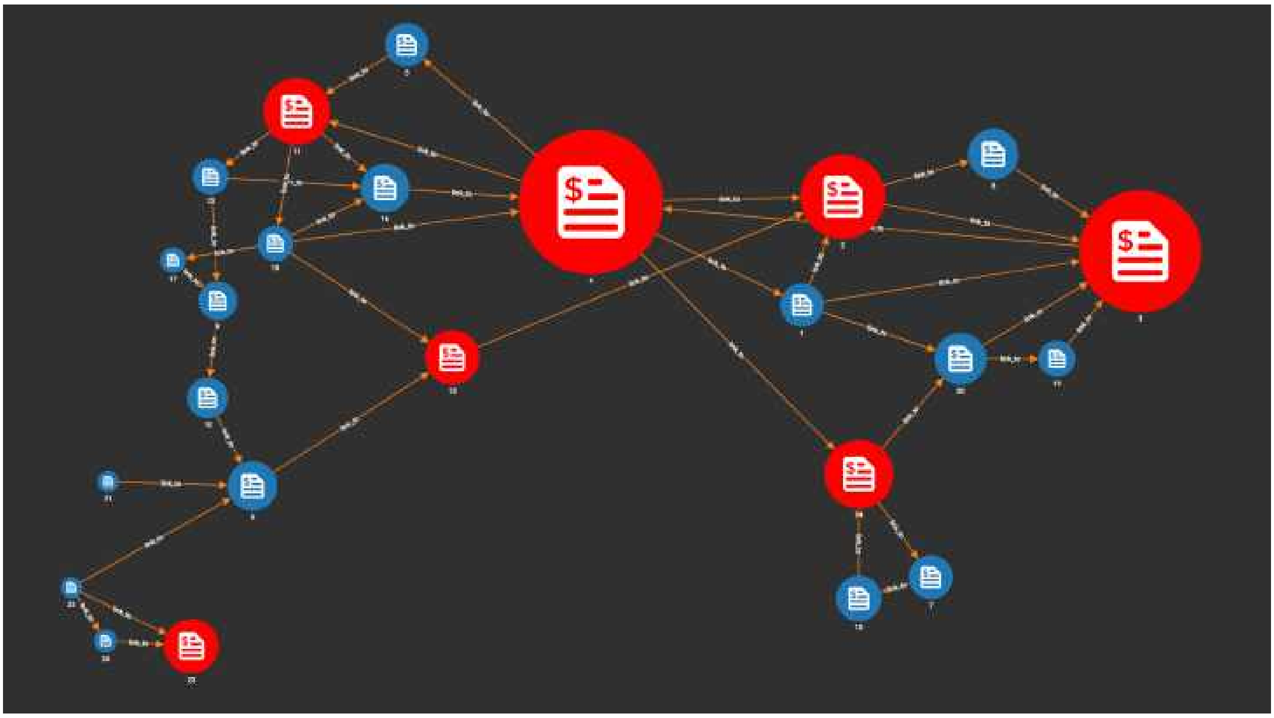
If you want to cancel the vertex radius or edge thickness configuration, click Edit button in Radius or Thickness section, in the pop-up window choose None in the top level expression dropdown list:



Click ADD, then click APPLY. The size will be changed back to uniform.

The size and color can be configured at same time. Here is the effect of setting both color and size for page rank vertices:


Settings
Visual configs of vertex and edge types
Vertex and edge types
Visual configs
Person
Company
Project
Page
WorkFor
PersonInvestCompany
CompanyInvestCompany
BidFor
link_to
Show attributes
type
primary id
Color
(@PageRankScore >= 1)
Click "+" to add color configs.
Radius
((@PageRankScore * 50) + 20)
The value of radius will be clamped between [20, 200].
Tooltip
Show tooltips when cursor hovers over vertices and edges.
CANCEL
APPLY



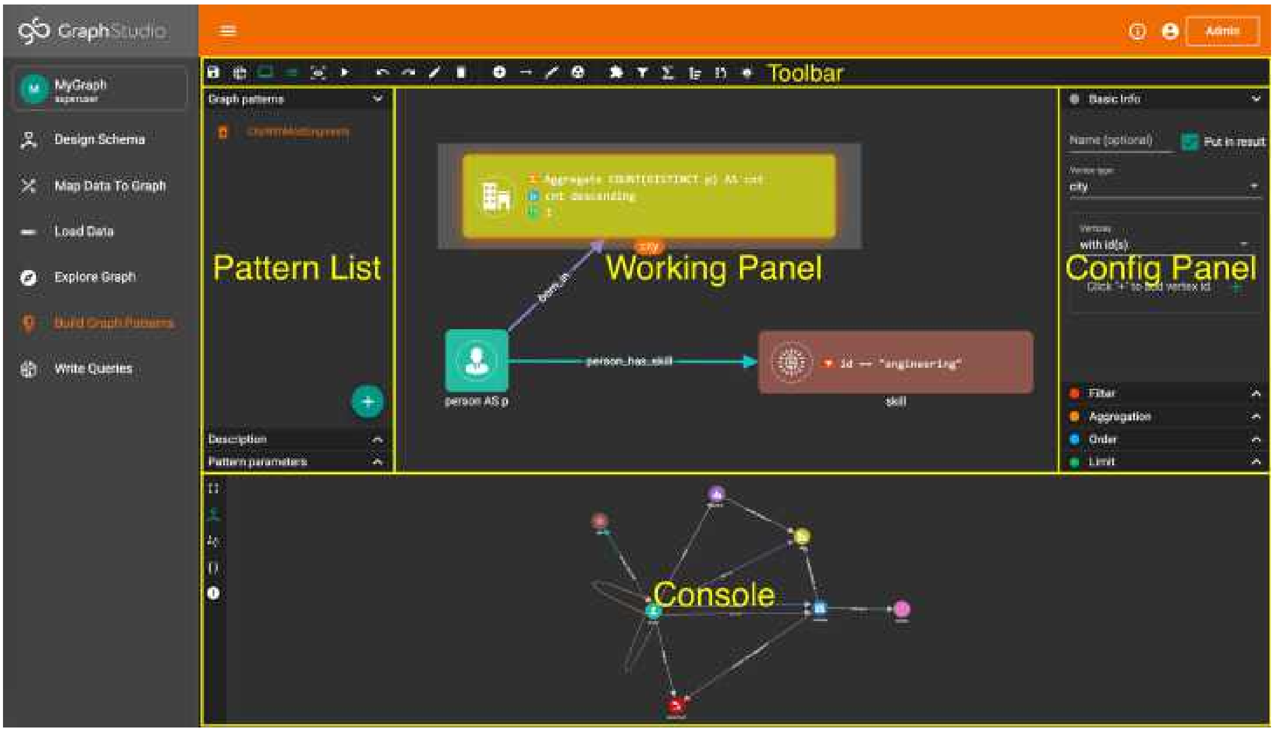
Build Graph Patterns

Visual Query Builder Overview

Visual Query Builder

In TigerGraph 3.0, we are proudly introducing *Visual Query Builder* -- a visual way of building your graph business logic. In the Build Graph Patterns page, you can create visual patterns in drag-and-drop fashion, which intuitively represent the questions you want to ask. For these cases, you don't need to write GSQL anymore!

By adding the Visual Query Builder component, GraphStudio becomes a complete visual SDK for users to *build graph applications from end to end without writing a single line of code*!



Currently, Visual Query Builder is still in beta phase. Please expect lots of improvements and changes coming in the future!

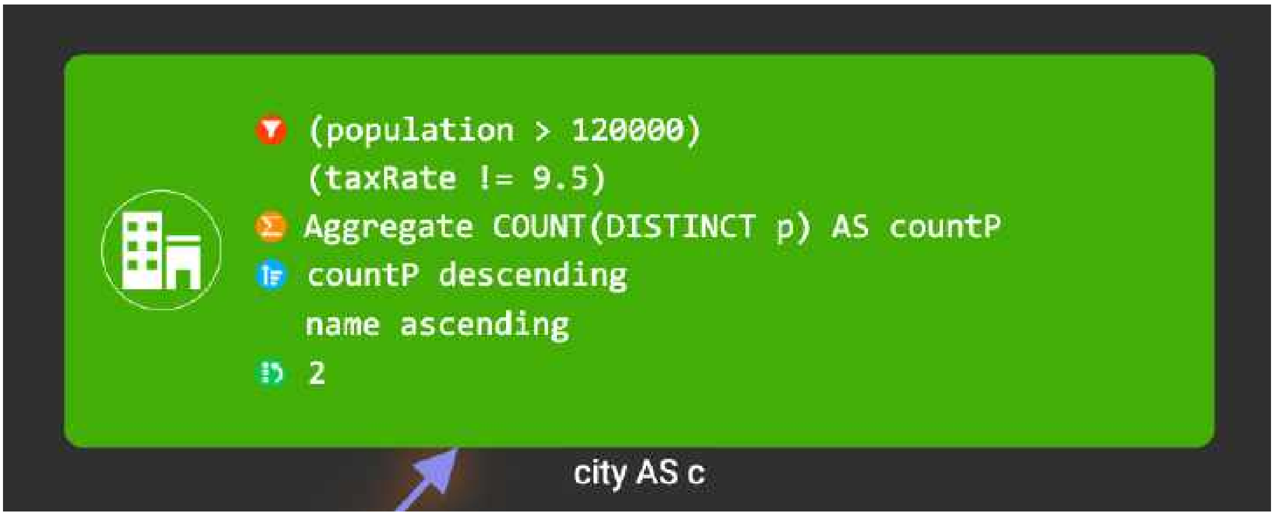
Basic Concepts

A visual pattern is a declarative way of describing a template subgraph structure, and all the subgraph structures that the visual pattern is **homomorphic** with are valid matching results.

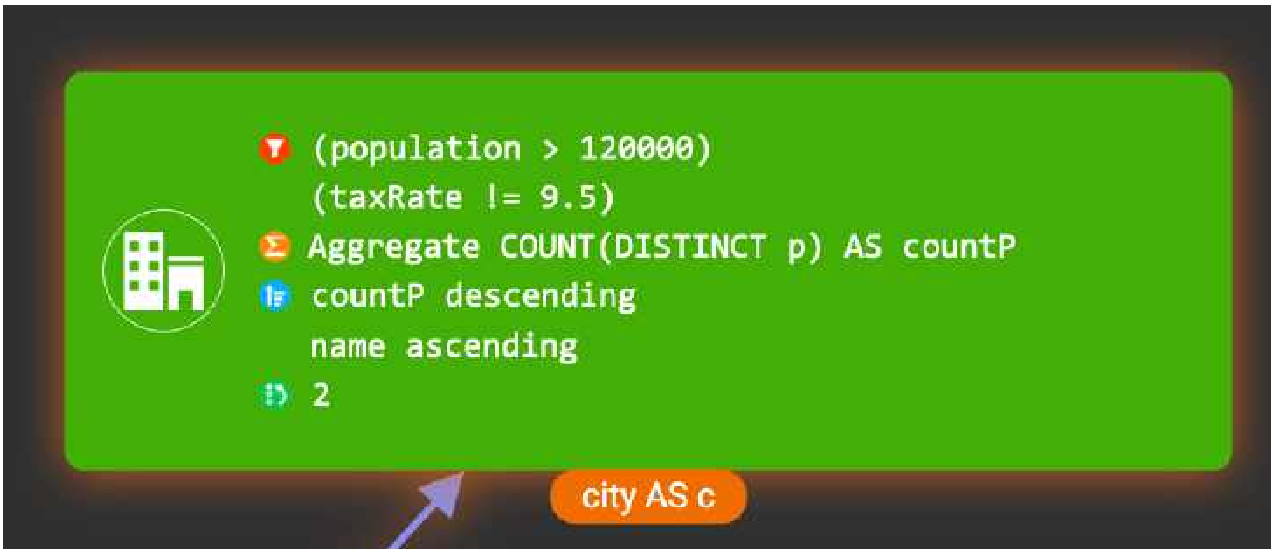
Visual patterns are constructed by vertex patterns, edge patterns and widgets.

This is a basic vertex pattern within a visual pattern:

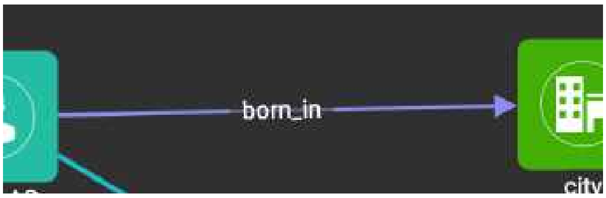
This is a vertex pattern with 2 filters, 1 aggregation, 2 ordering, and limit:



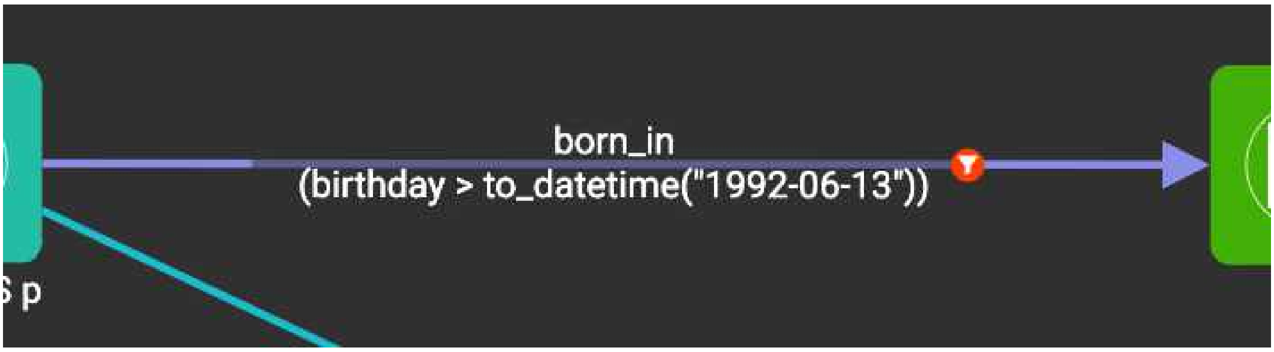
This is a vertex pattern that will be output in the pattern execution result. You can see the glow around the border and the highlight at the label:



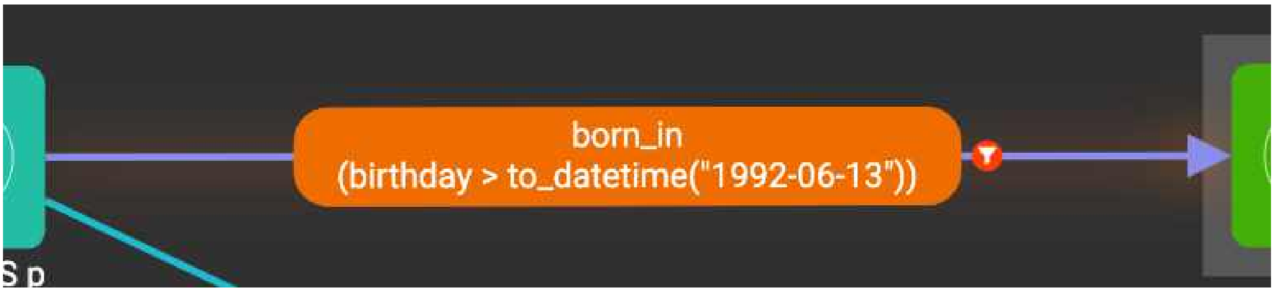
This is a basic edge pattern:



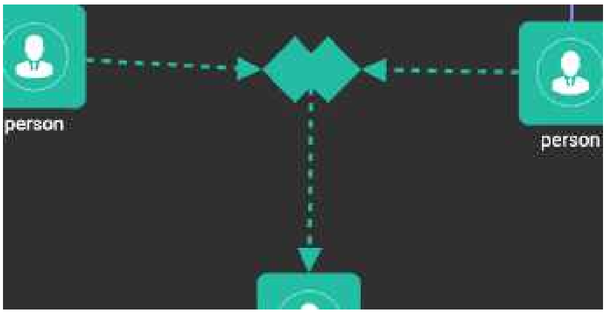
This is an edge pattern with 1 filter:



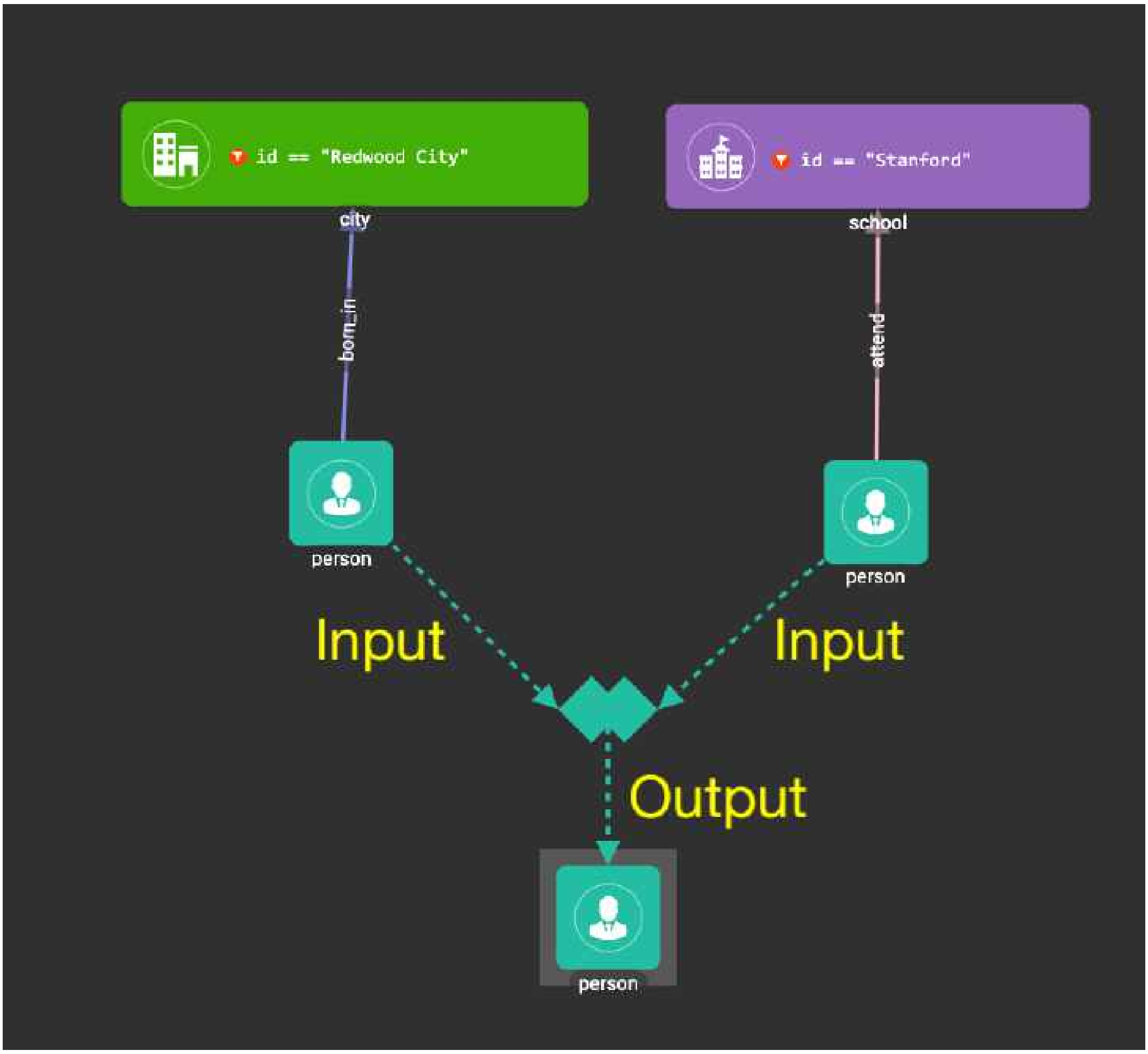
This is an edge pattern that will be output in the pattern execution result:



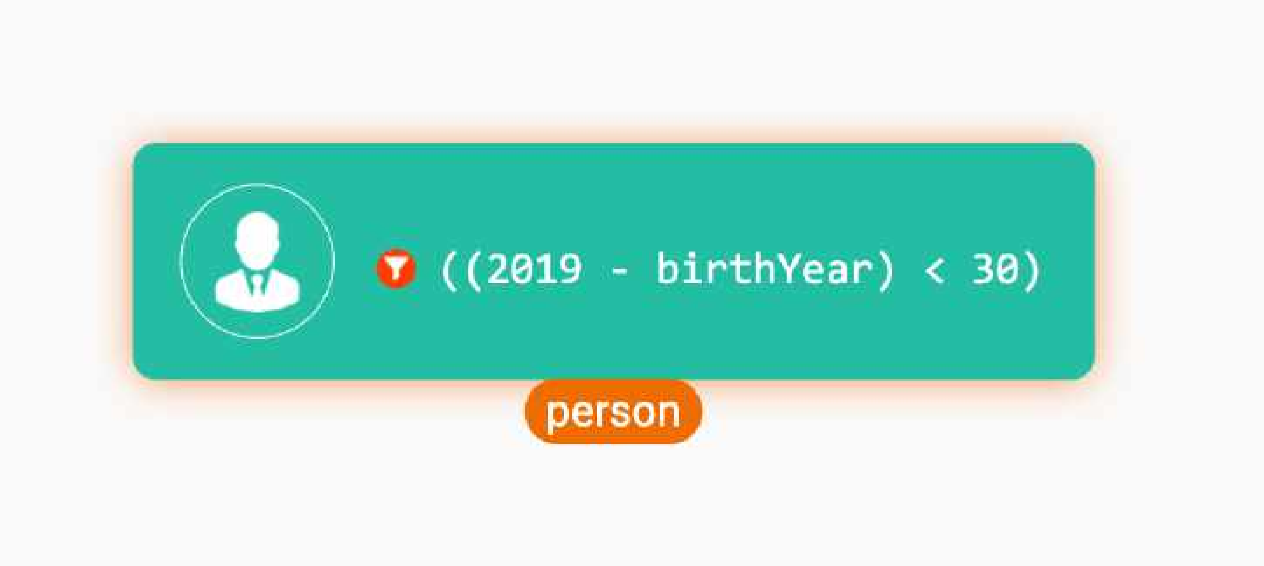
This is a union widget:



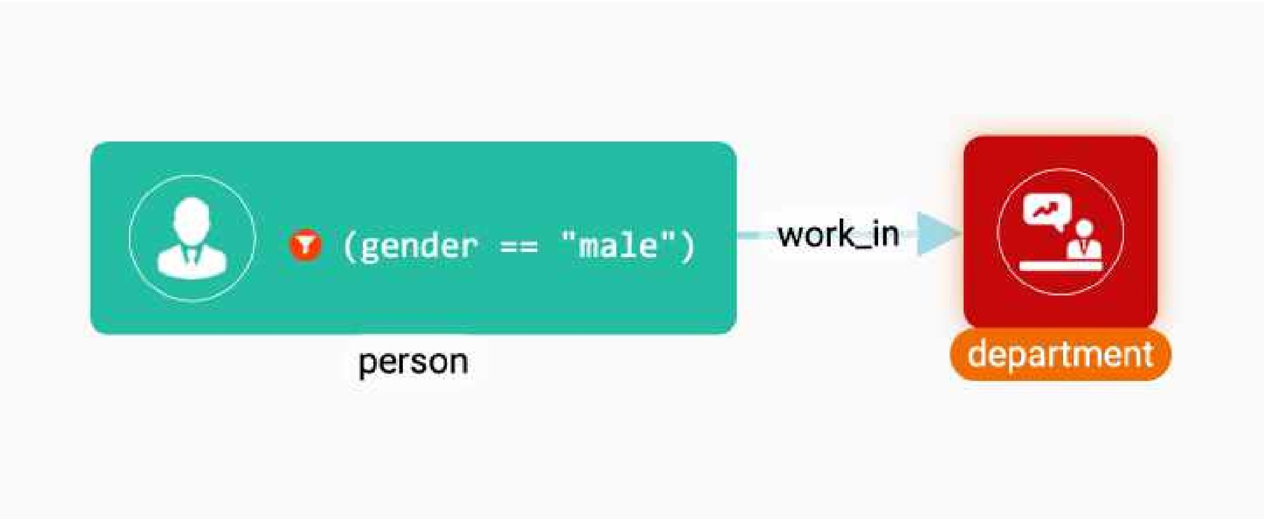
A widget has input vertex patterns and output vertex patterns:



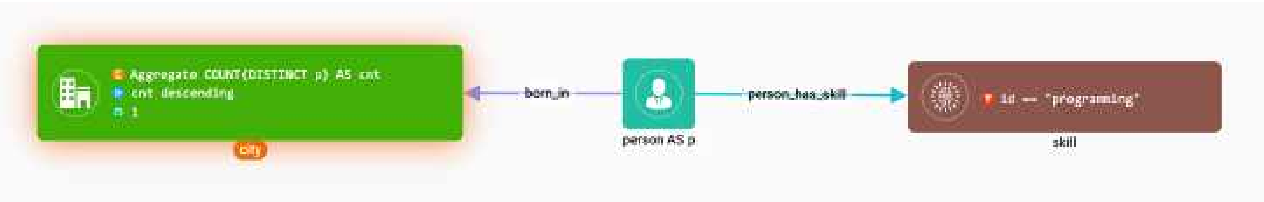
This is the visual pattern matching all the people who are younger than 30 years old in the year 2019. The filter **((2019 - birthYear) < 30)** on the person vertex pattern provides the matching condition:



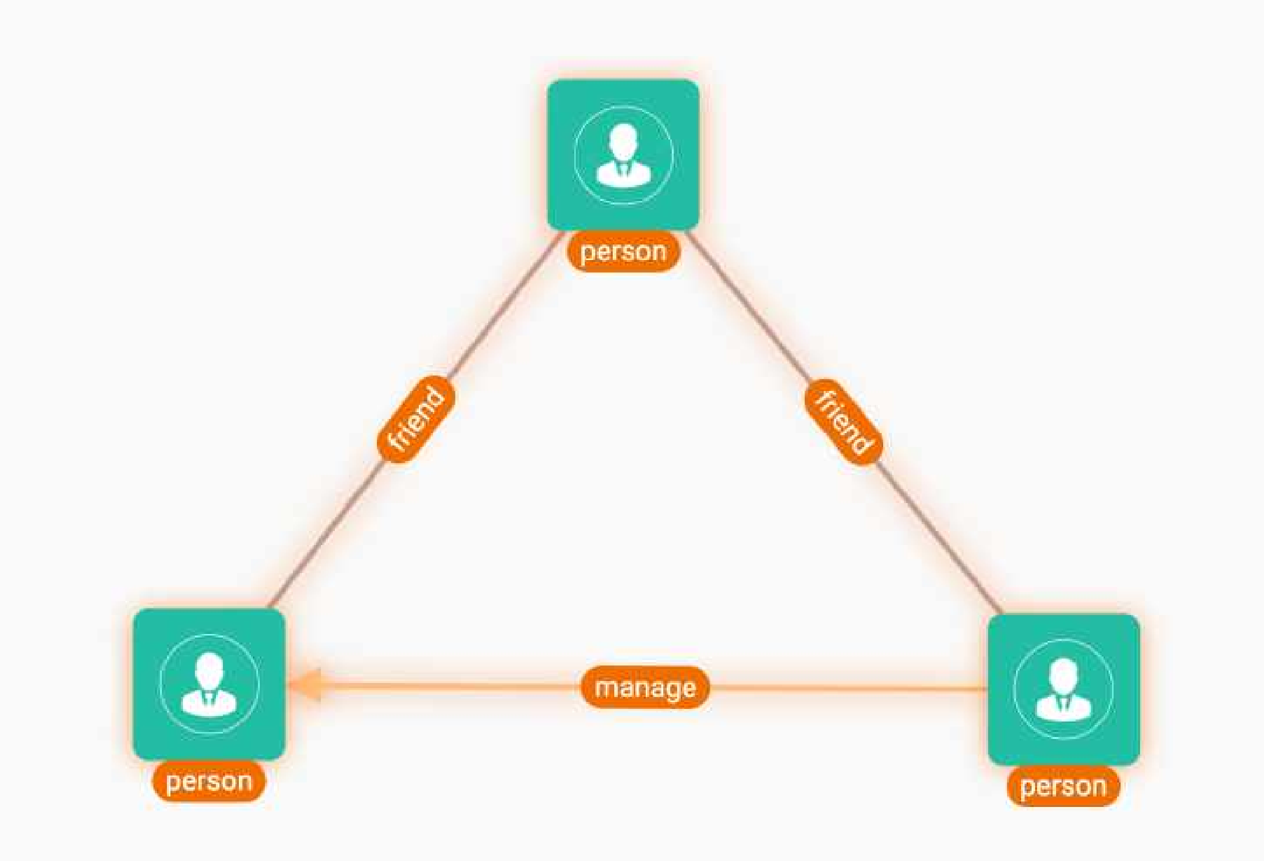
This is the visual pattern matching all the departments having at least one male employee:



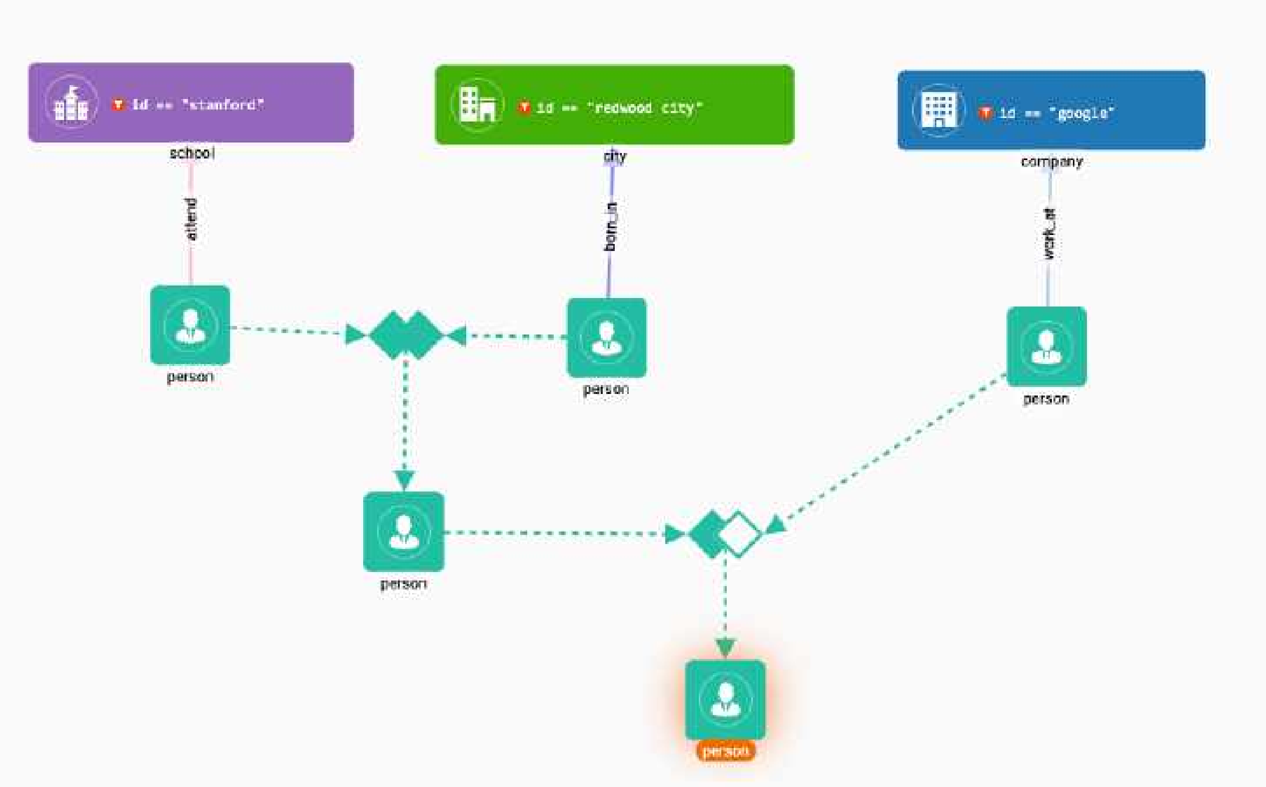
This is the visual pattern that matches the hometown of the most people having the programming skill. The aggregation **COUNT(DISTINCT p) as cnt** annotates the number of people born in each city that have the programming skill. Then we order the city vertices **by their descending cnt value**, and choose the top 1 city:



This is the visual pattern matching all such three persons A, B and C, where A is B's friend, and A is C's friend, and B is C's manager:



This visual pattern matches all the people that born in Redwood City and attended Stanford university, but didn't work for Google:

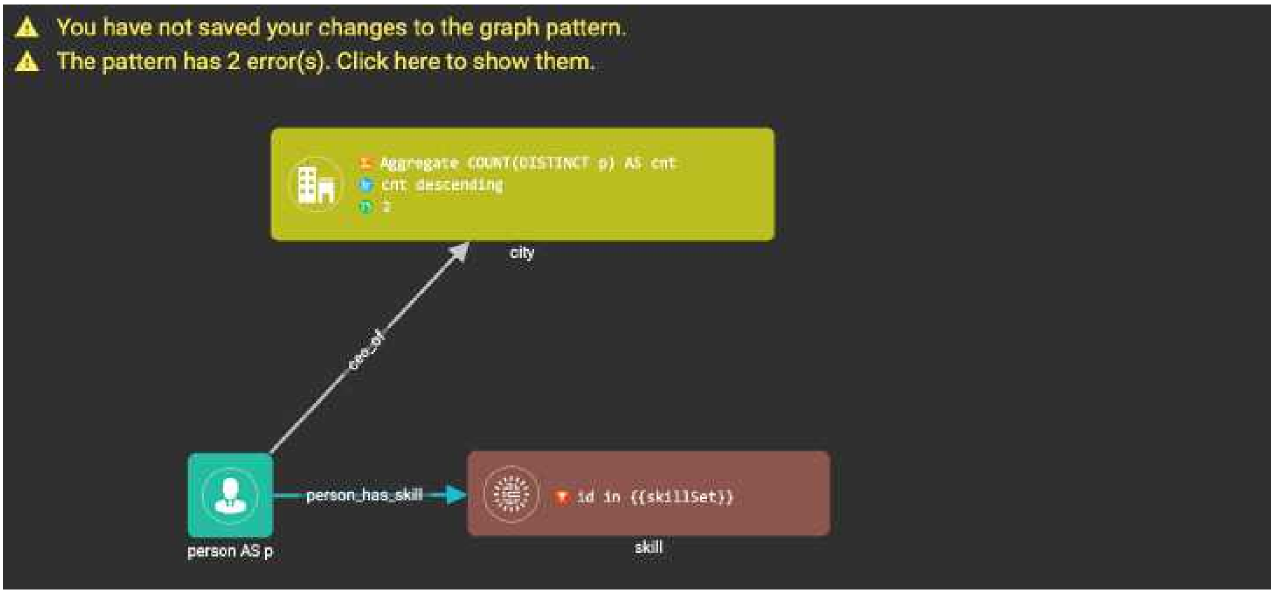


Now you have a first sight about how the visual patterns look like. Let's walk through all the concepts in depth.

Working Panel

Working panel is the central component (both visually and logically) of Visual Query Builder. Your visual patterns are rendered here.

Like other pages in GraphStudio, you can zoom-in, zoom-out and drag the visual pattern. If you have unsaved changes to the pattern, or containing errors in the pattern, some warning messages will show at the top left corner of working panel:



If you click the message indicating errors, the Console will switch to Problems tab:

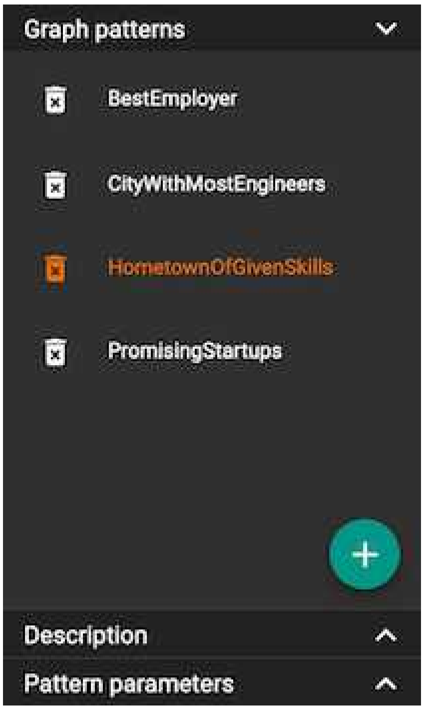
You can click the error messages. For some errors, the vertex patterns, edge patterns and widgets involved in the error will be marked red to facilitate your debugging:

Pattern List

The pattern list panel contains three sections:

Graph patterns

The graph patterns section lists all the visual patterns you have created for current graph. You can click  to create new patterns, or click  to delete existing ones.



Description

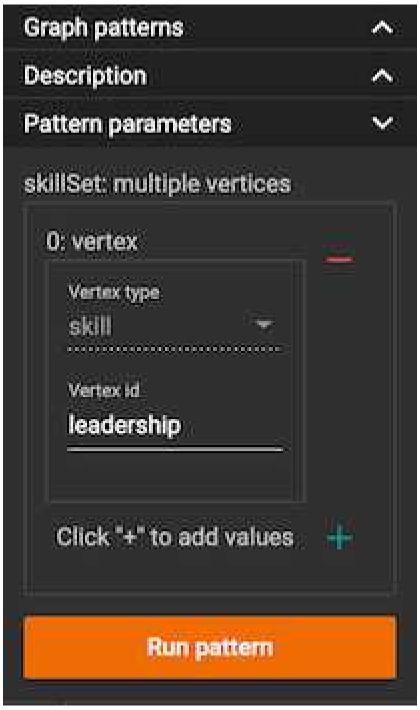
Other than the visual view of the graph pattern, you can optionally add a description for the pattern. This can help other users to understand what problem you are trying to solve.

After changing the description, don't forget to confirm the change:

Changing description is considered as a change to the visual pattern, don't forget to save the pattern in the end.

Pattern parameters

If you added some parameters in your visual pattern, you need to provide the values to the parameters when running the pattern.



Click 

to run the pattern.

Toolbar

The toolbar options, from left to right, are the following:

toolbar option	functionality
	Save the graph pattern.
	Save as GSQL query: show the GSQL query generated from the pattern and save. See more information here.
	Console: open/close the console panel.
	Configuration panel: open/close the config panel. By default the config panel is closed. You can either open the panel by clicking this button, or double click a vertex pattern or edge pattern in the visual pattern to open the config panel.
	Render pattern options: config how much detail is shown on the pattern. See more

	information here.
	Run: run the visual pattern. If the pattern doesn't have any parameters, it will run directly, otherwise the **Pattern parameters** section will expand for you to provide the parameter values.
	Undo and redo: undo and redo the changes on the visual pattern. The whole editing history of each visual pattern since entering Build Graph Patterns page is preserved.
	Edit: edit the selected vertex pattern or edge pattern. This is same as double-clicking one vertex or edge pattern.
	Delete: delete selected vertex patterns, edge patterns and widgets. You can hold the Shift key to select multiple elements to delete.
	Add a vertex pattern: add a new vertex pattern into the current visual pattern. See more information here.
	Add an edge pattern: add a new edge pattern into the current visual pattern. See more information here.
	Pick: a shortcut for adding vertex patterns and edge patterns into the visual pattern. See more information here.
	Merge: select multiple vertex patterns, and click this button to merge them together. This is a fast way to connect multiple shorter patterns into a longer one. See more information here.
	Widget: see more information here.
	Filter: click this button then click a vertex pattern or edge pattern, and the config panel will enter editing mode for the selected vertex or edge pattern, with the Filter section expanded.

	Aggregation: click this button then click a vertex pattern, and the config panel will enter editing mode for the selected vertex pattern, with the Aggregation section expanded.
	Order by: click this button then click a vertex pattern, and the config panel will enter editing mode for the selected vertex pattern, with the Order section expanded.
	Limit: click this button then click a vertex pattern, and the config panel will enter editing mode for the selected vertex pattern, with the Limit section expanded.
	Output: click this button, then click vertex patterns and edge patterns of the visual pattern to toggle whether to output them or not. You can see the output glow of the selected vertices or edges turning on and off

Configuration Panel

You can edit vertex patterns and edge patterns from config panel.

Basic Info for vertex pattern

If you enter editing mode of a vertex pattern and expand the Basic Info section, you can edit its name, decide whether or not to put it into result, change its vertex type, provide optional matching conditions by giving a list of ids, or provide a parameter name.

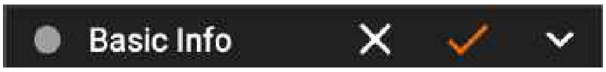
You can add/drop id in the list:

Or add an input parameter:

Basic Info for edge pattern

If you enter editing mode of an edge pattern and expand the Basic Info section, you can edit its name, decide whether or not to put it into result, or change its edge type. You can also specify this edge pattern as a regex match by providing Repeat as least (a non-negative integer) and Repeat at most (a positive integer). Due to current GSQL limitations, if you decide to change these numbers, you cannot give the edge pattern a name or put it into result.

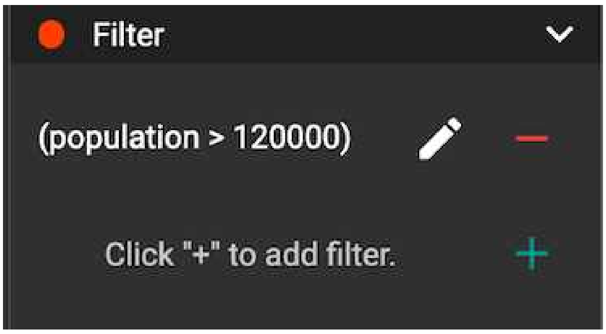
When finished editing, you need to confirm the change. You can also cancel the change if you made a mistake:



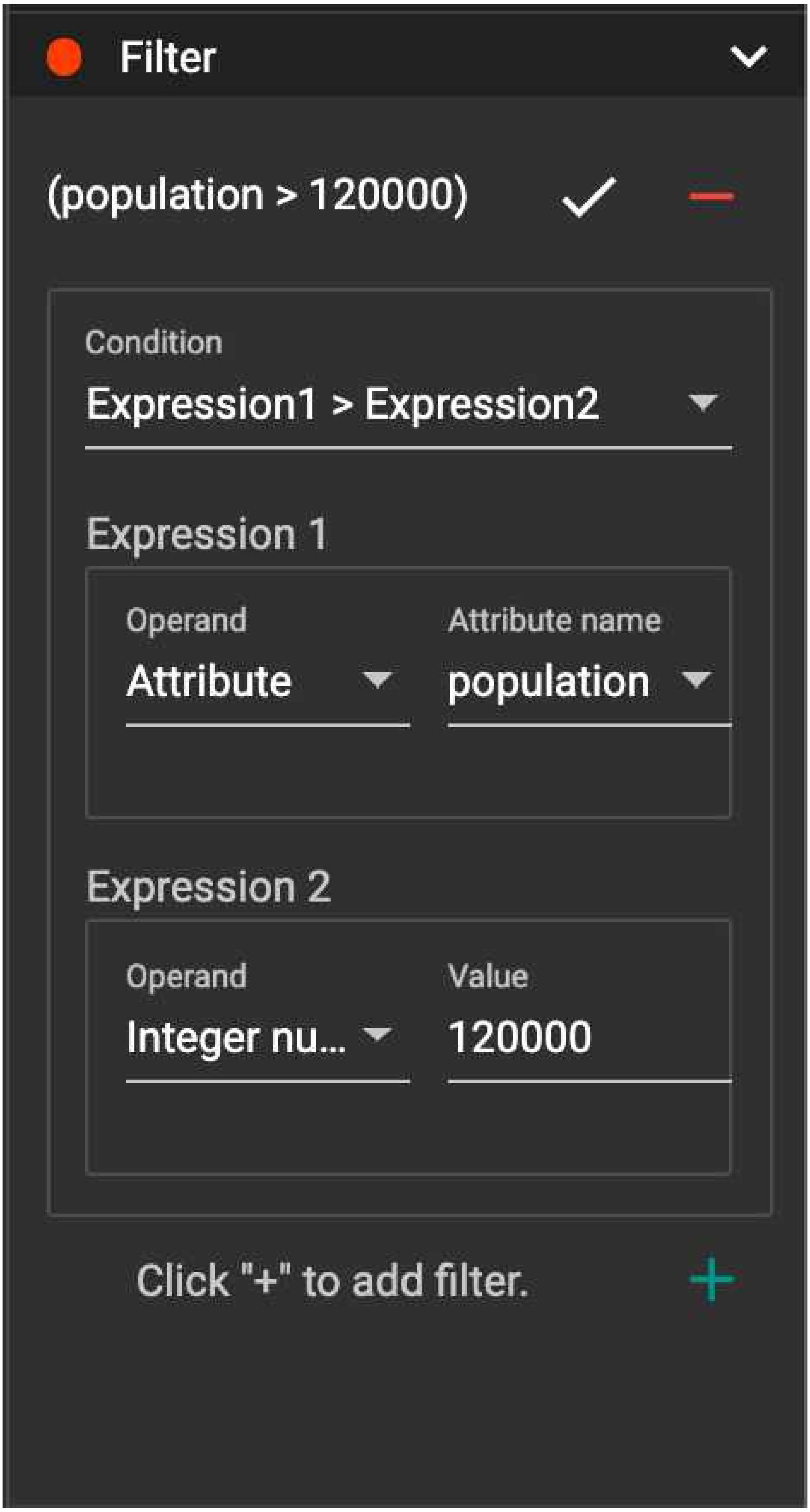
> If you want another vertex pattern or edge pattern to refer to current selected vertex or edge pattern's attributes, you need to give it a name.

Filter

If you expand the Filter section, you can add/edit/delete filters for the selected vertex or edge pattern:



Click  besides the filter expression and enter editing mode of the filter:

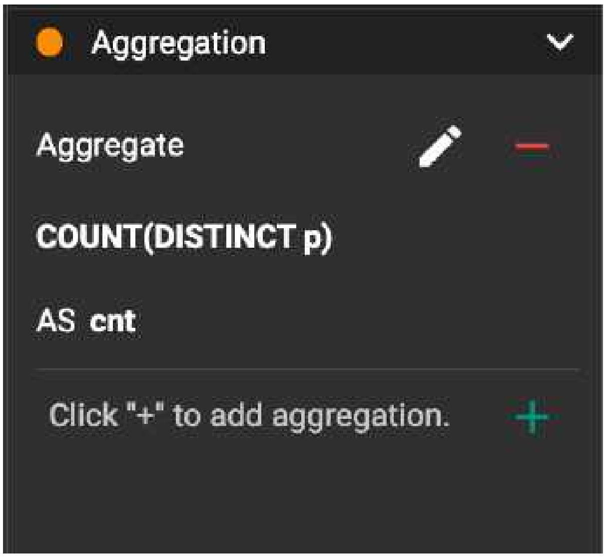


You can add multiple filters for each vertex and edge pattern, and they are AND relationship when executing the pattern. Building the filter is similar with building attribute filter in Explore Graph page. One thing special here is that you can refer to attributes on other vertex patterns and edge patterns. Choose **Attribute of vertex or edge** as expression type, then choose the name of the vertex or edge pattern whose attribute you want to refer to (see above why we need give a name to vertex or edge patterns), then choose the attribute you want to refer to.

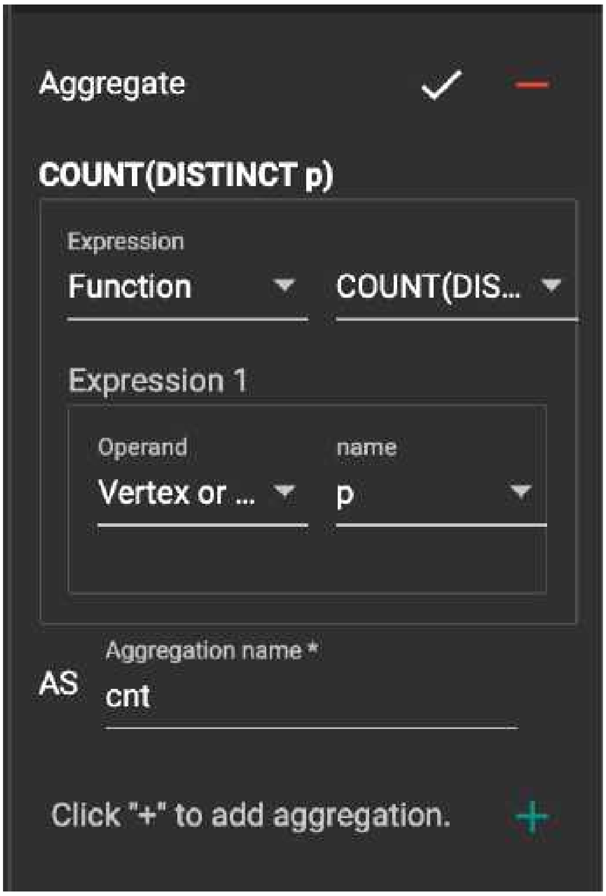
When finished editing, you need confirm or cancel the change:

Aggregation

If you expand the Aggregation section, you can add/edit/delete aggregations for the selected vertex pattern (aggregation on edge patterns is not supported):

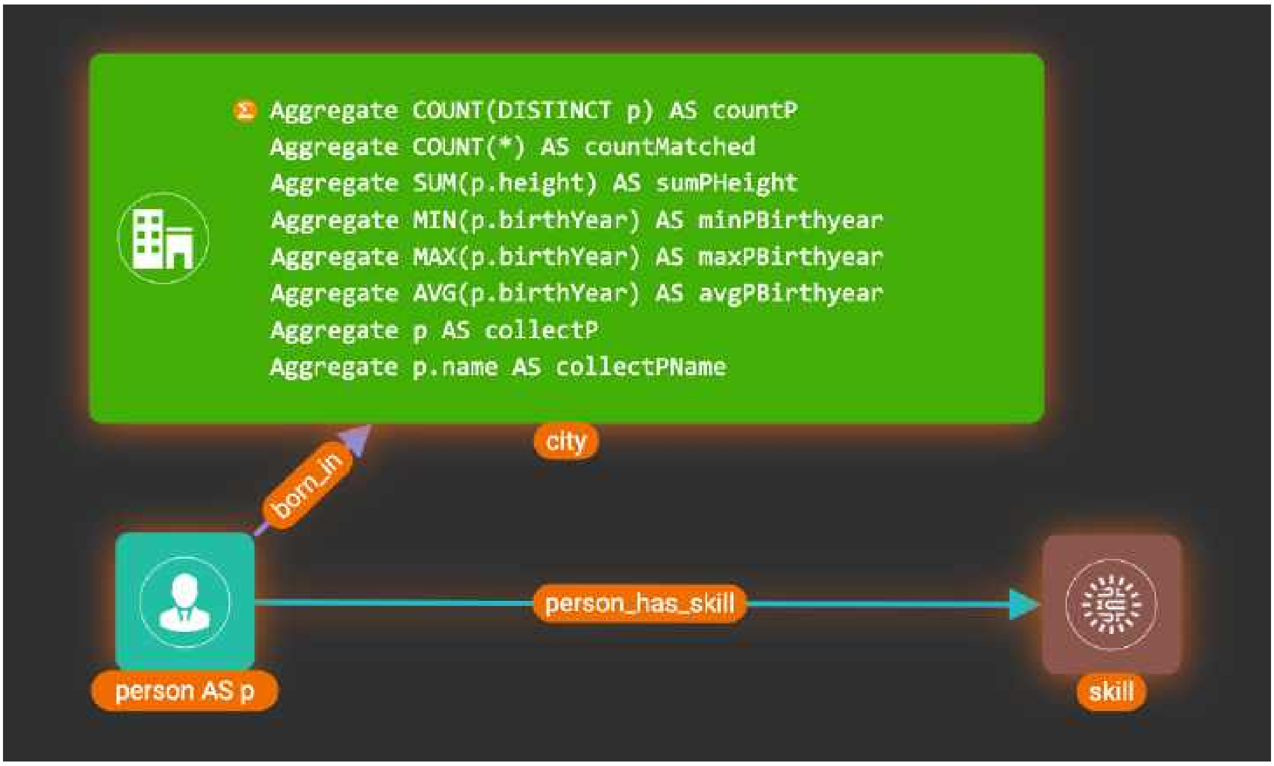


Click beside one aggregation and enter editing mode for the aggregation. You can edit both the aggregation expression and aggregation name:



When finished editing, you need to confirm or cancel the change:

When talking about aggregation, it is actually grouping all the matching results by the vertex entity which match the vertex pattern, and then aggregated based on the expression. Take this example:

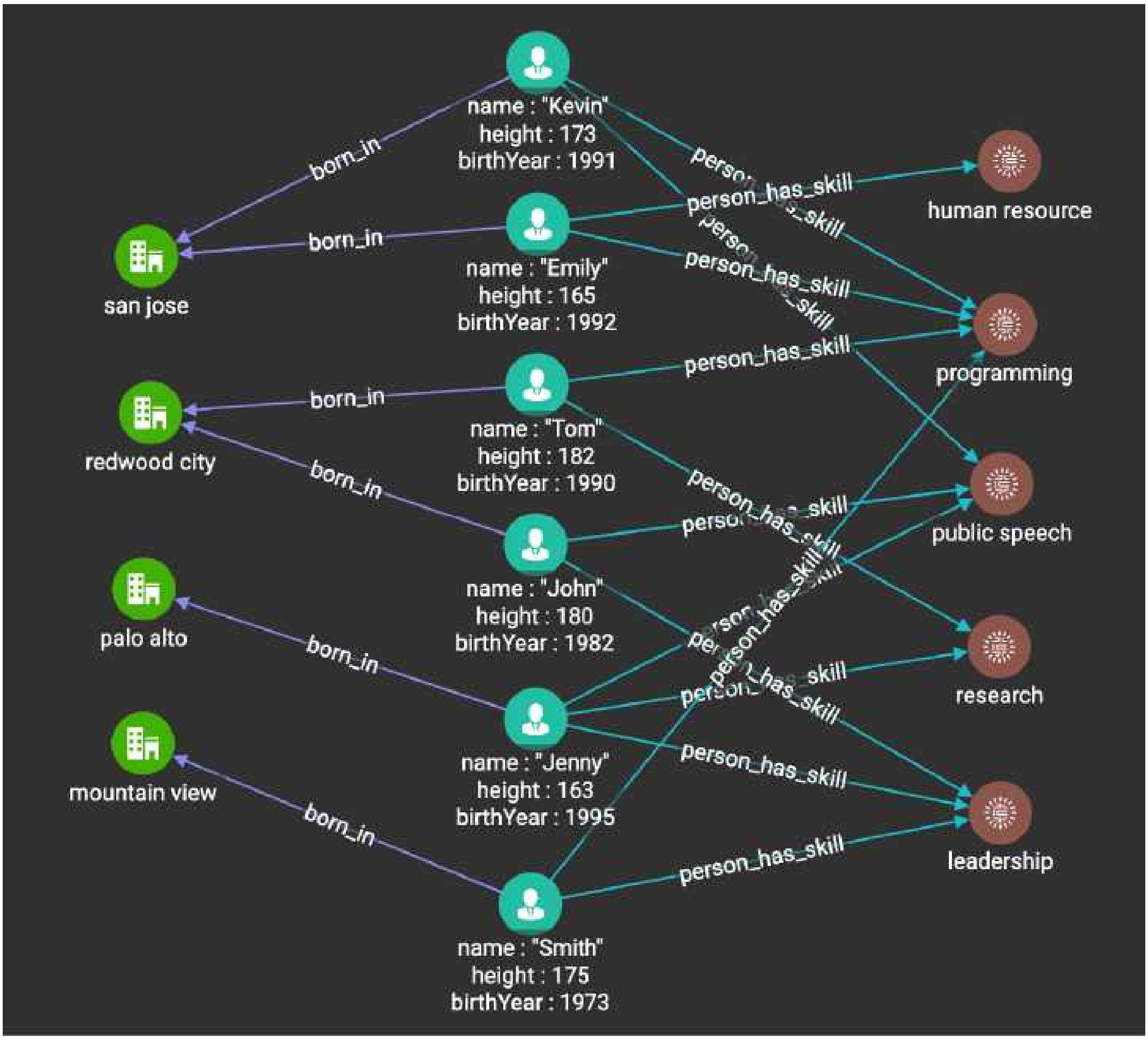


In the city vertex pattern, we have created 8 different aggregations:

name	aggregation expression	explanation
countP	COUNT(DISTINCT p)	Count number of distinct person vertices matched to each city vertex.
countMatched	COUNT(*)	Count number of matches each city vertex involves in.
sumPHeight	SUM(p.height)	Sum the height attribute of all person vertices matched to each city vertex.
minPBirthyear	MIN(p.birthYear)	Get the minimal height attribute of all person vertices matched to each city vertex.
maxPBirthyear	MAX(p.birthYear)	Get the maximal height attribute of all person vertices matched to each city vertex.
		Get the average value of height attribute of all person

avgPBirthyear	AVG(p.birthYear)	vertices matched to each city vertex.
collectP	p	Collect all person vertices matched to each city vertex.
collectPName	p.name	Collect all person vertices' name matched to each city vertex.

Consider we have the following matching result:

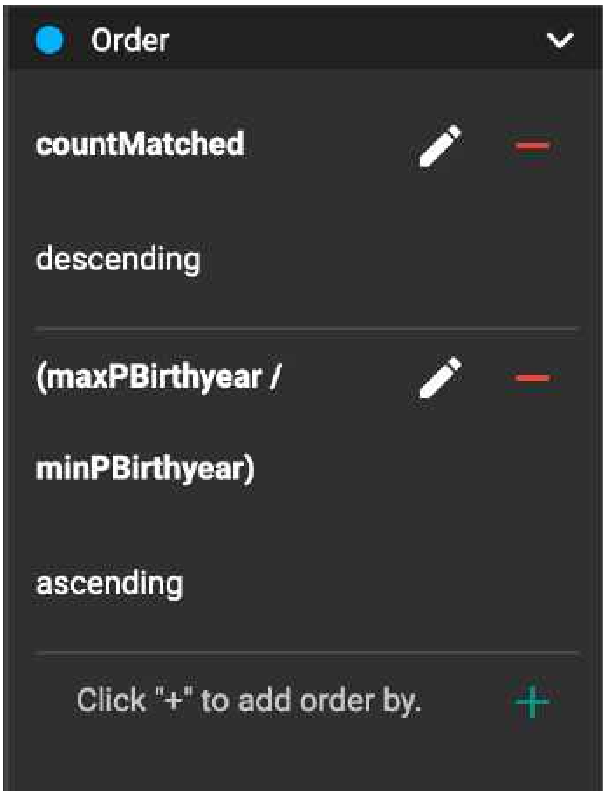


We have the following aggregation result table for city vertex *san jose*:

city vertex	aggregation result	explanation
san jose	countP = 2	Kevin and Emily
san jose	countMatched = 4	(san jose)←[born_in]-(Kevin)-[person_has_skill]→(programming) (san jose)←[born_in]-(Kevin)-[person_has_skill]→(public speech) (san jose)←[born_in]-(Emily)-[person_has_skill]→(human resource) (san jose)←[born_in]-(Emily)-[person_has_skill]→(programming)
san jose	sumPHeight = 676	Kevin.height + Kevin.height + Emily.height + Emily.height = 173 + 173 + 165 + 165
san jose	minPBirthyear = 1991	Min(Kevin.birthYear, Kevin.birthYear, Emily.birthYear, Emily.birthYear)
san jose	maxPBirthyear = 1992	Max(Kevin.birthYear, Kevin.birthYear, Emily.birthYear, Emily.birthYear)
san jose	avgPBirthyear = 1991.5	(Kevin.birthYear + Kevin.birthYear + Emily.birthYear + Emily.birthYear) / 4
san jose	collectP = [Kevin, Kevin, Emily, Emily]	
san jose	collectPName = ["Kevin", "Kevin", "Emily", "Emily"]	

Order

If you expand the Order section, you can add/edit/delete ordering for the selected vertex pattern (ordering on edge patterns is not supported):



Click  beside the ordering and enter editing mode for the ordering. You can edit both the ordering expression and whether results are in ascending or descending order:

When finished editing, you need to confirm the change:

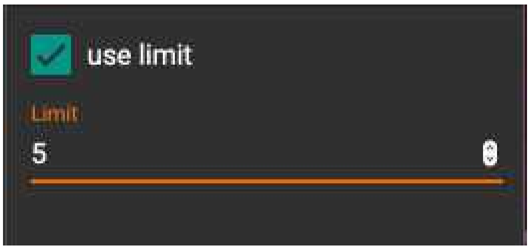
You can refer to aggregations in ordering expression. You can add multiple orderings, which follow the multi-key ordering rule (upper ordering dominates).

Limit

If you only want a subset of your matching result, you can use limit. Only the top limit results will be returned based on your ordering settings. If you don't have orderings, the result will be randomly picked from all matchings.

If you expand the Limit section, you can add/edit/delete limit for the selected vertex pattern (limit on edge patterns is not supported):

Toggle **use limit** checkbox to enable/disable limit. You can also edit the limit number:



When finished editing, you need confirm or cancel the change:

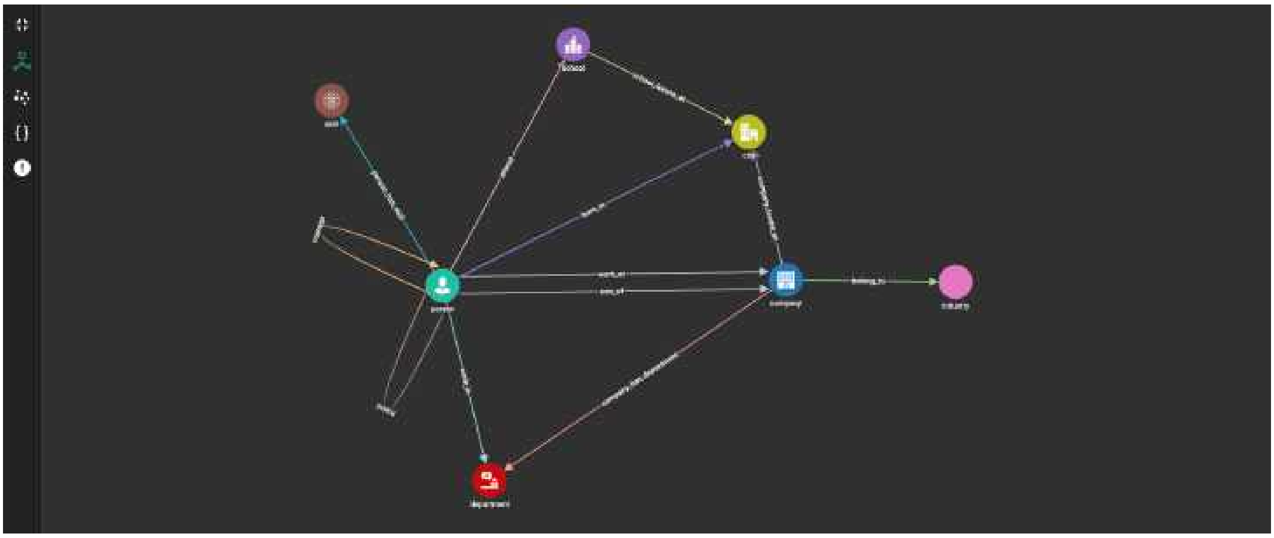
Console

The Console panel shows the graph schema, the result of the last pattern execution result, and errors the visual pattern has. Each execution of a pattern generates two types of results: a visualized graph and JSON text. On the left is a toolbar with buttons for switching between the tabs. The buttons, from top to bottom, are the following:

menu option	functionality
	Expand/Collapse: expand or collapse the Console panel.
	Graph schema: show the graph schema.
	Visualize graph result: show the visual result of the last run pattern.
	View JSON result: show the raw text result in JSON format of the last run pattern.
	Problems: show the errors in the visual pattern.

Graph schema

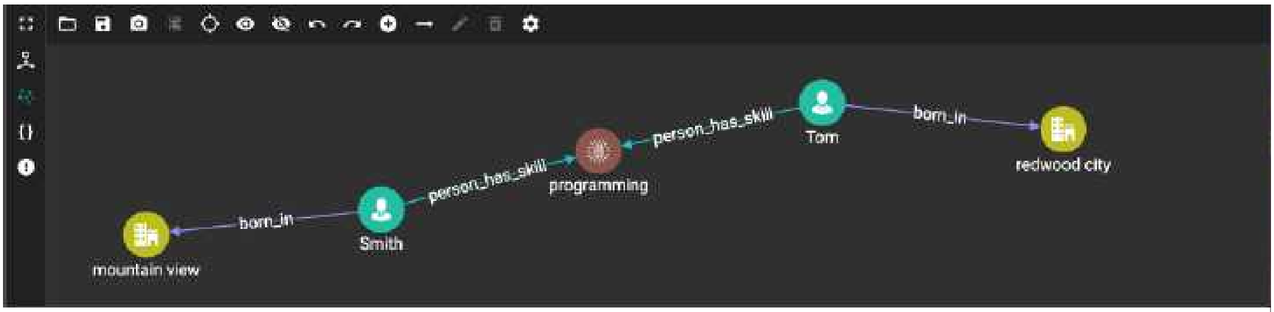
Viewing graph schema makes it more convenient for developers to refer to the schema topology logic and easier to construct the visual pattern.



Visualize graph result

If the pattern execution result contains a graph structure, the result will be visualized in this panel as a graph. The panel is the same as the [Explore Graph](#)

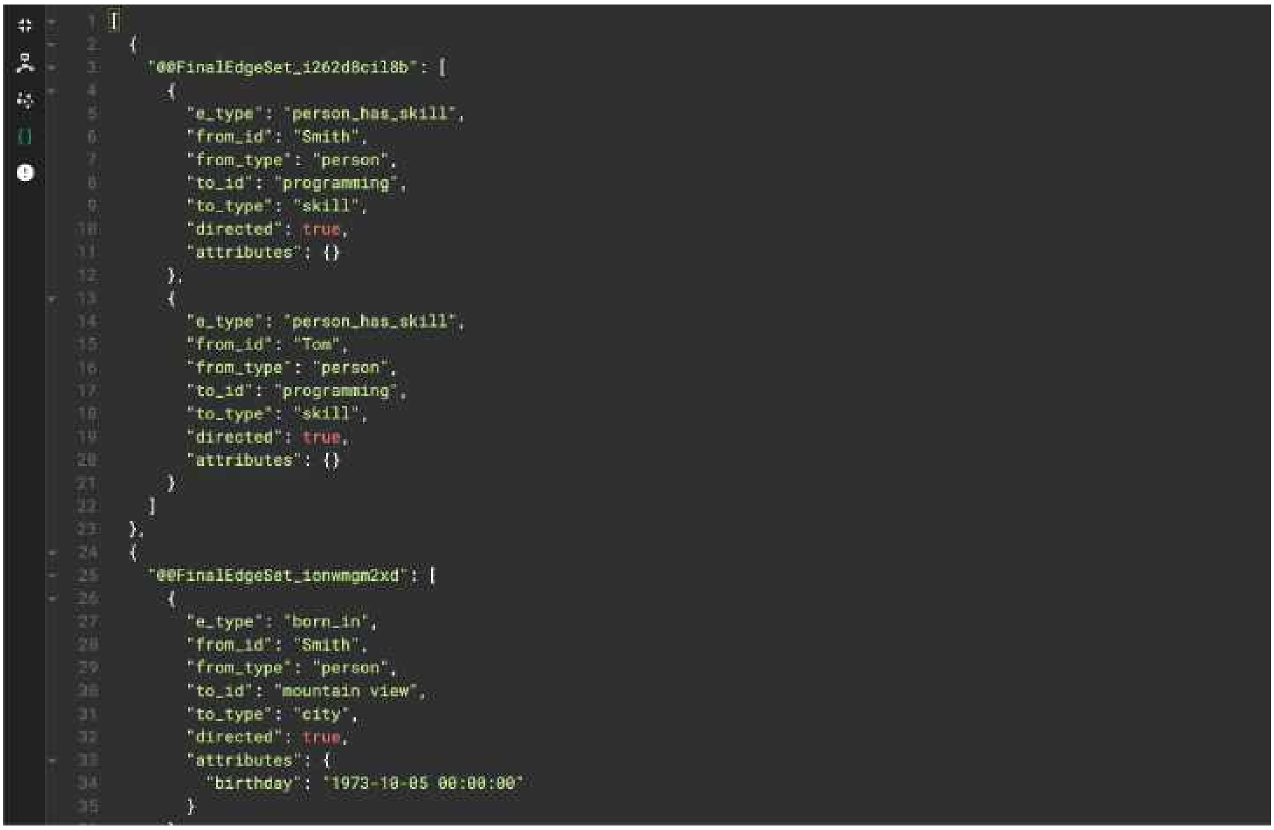
panel. The only difference is that each time you run a pattern, the previous result will be erased. In Explore Graph, the results are added incrementally.



You can switch to the JSON Result panel to see the result in JSON format.

View JSON result

You can see the JSON response of running the visual pattern from this tab:

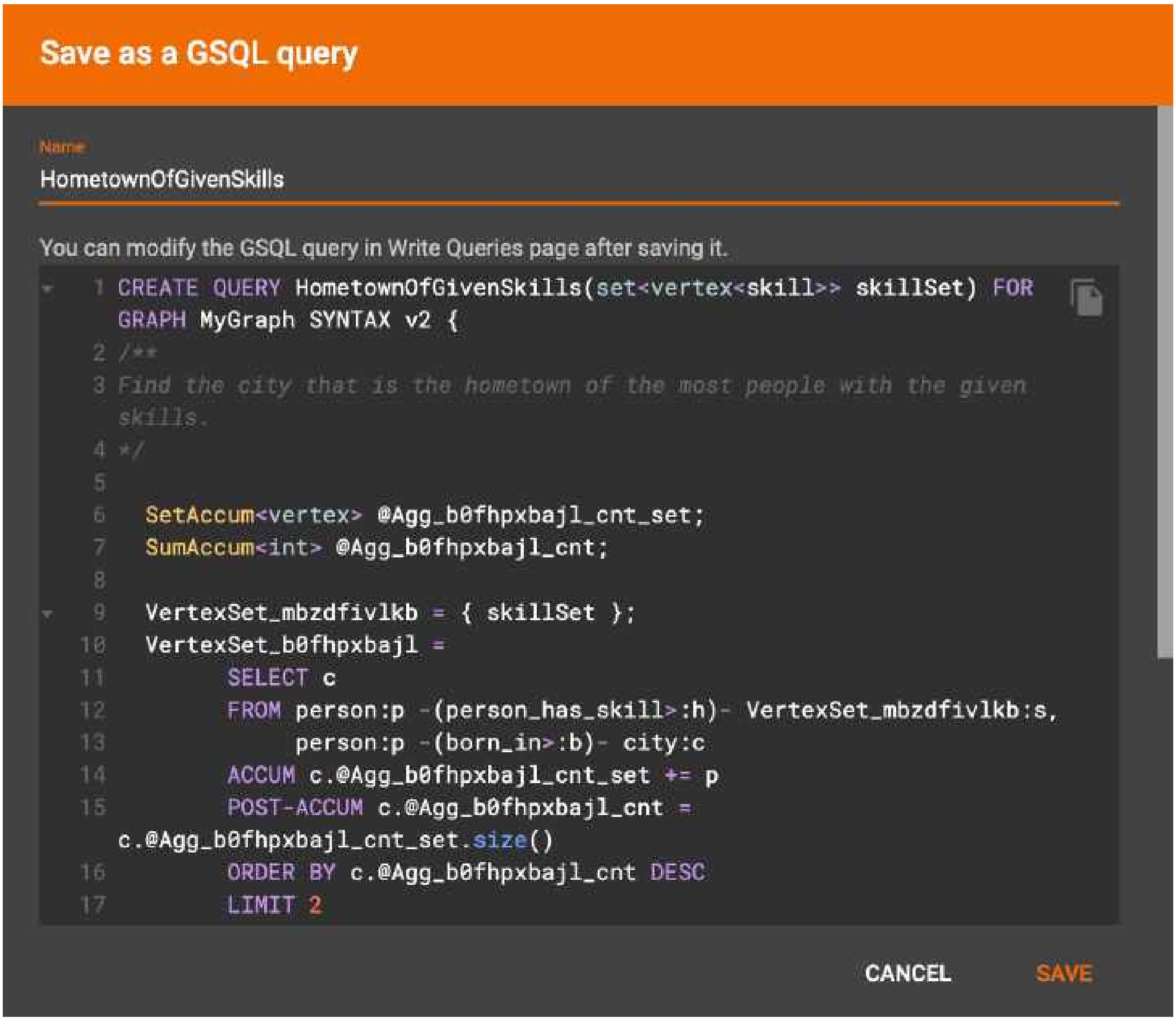


Problems

If the visual pattern contains errors, you can see them here and debug:

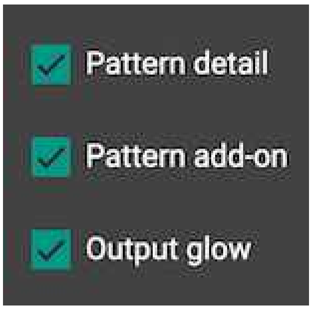
Save as GSQL query

You can view the GSQL query generated from your visual pattern and save it. Then you can access this query from Write Queries page, modify your query, interpret it, install it and run it.

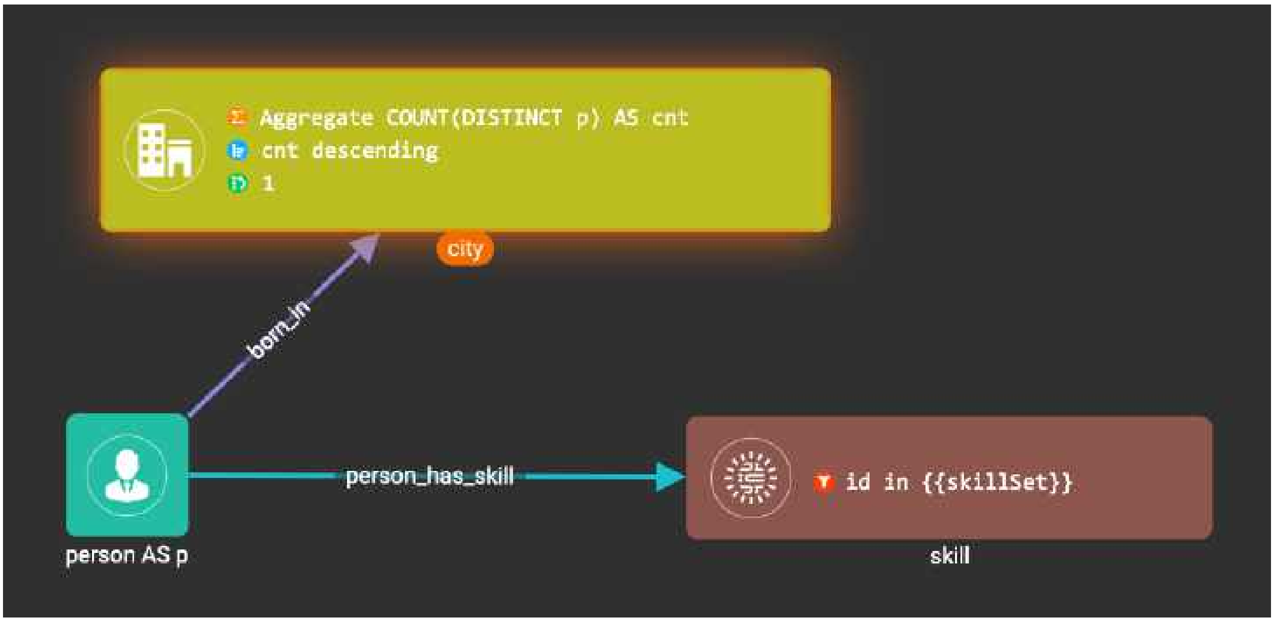


Render pattern options

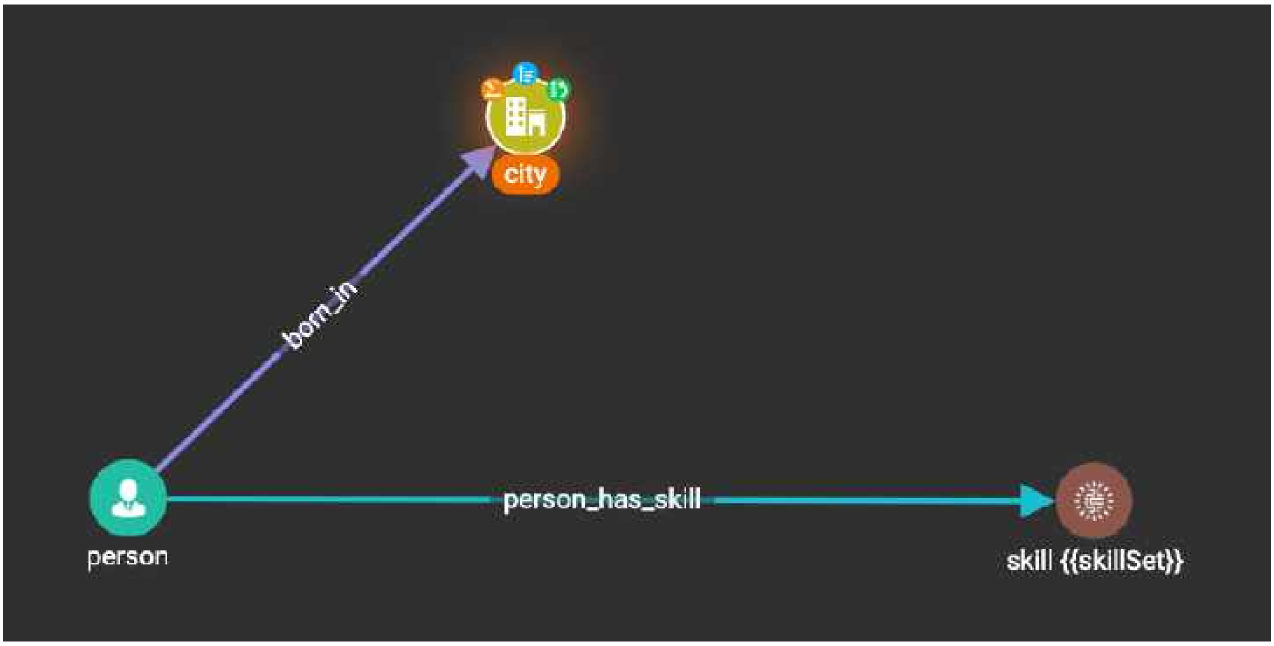
There are three different rendering options.



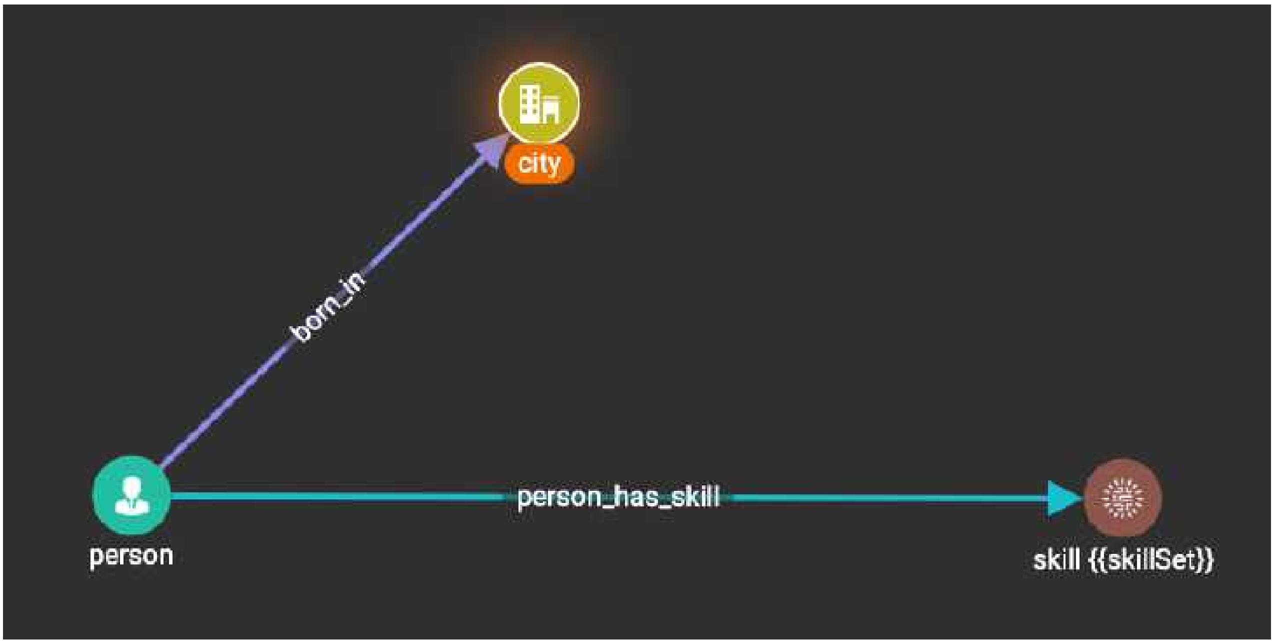
By default, Pattern detail and Output glow are checked. All the filters, aggregations, ordering conditions and limits are rendered, and the vertex and edge patterns that will be in result will be highlighted with glow:



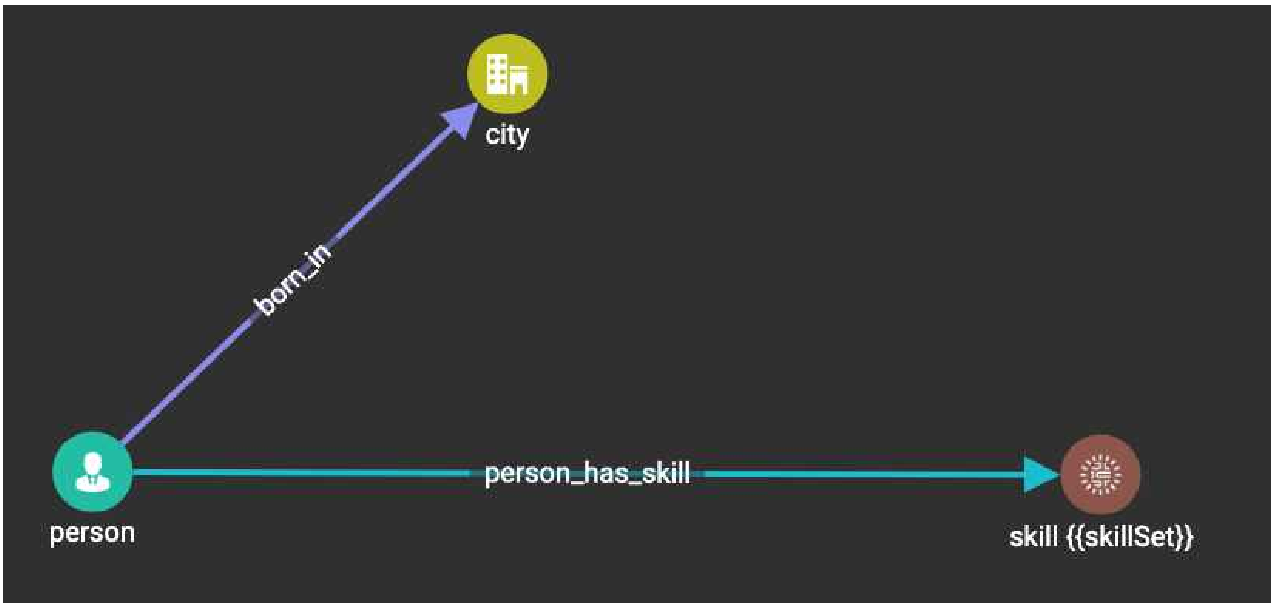
If Pattern detail is not checked, add-on marks will indicate that there are filters, aggregations, ordering conditions and limits on corresponding vertex patterns and edge patterns:



If Pattern add-on is not checked, the add-ons will be hidden:

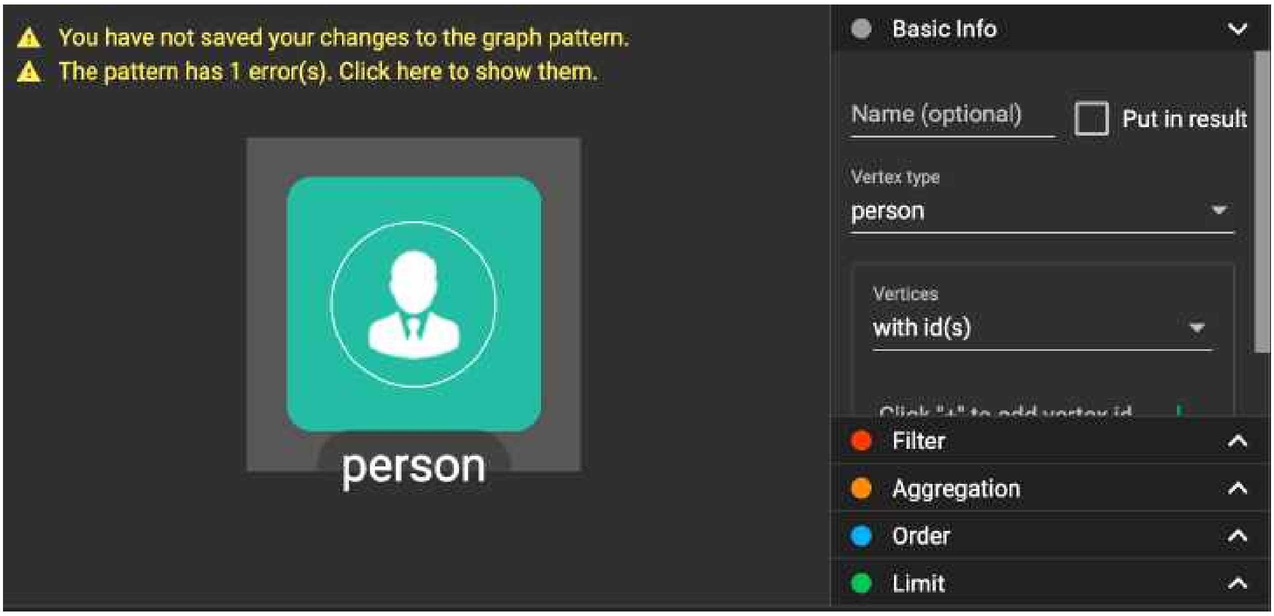


If Output glow is not checked, the output indicating glow is hidden:



Add a vertex pattern

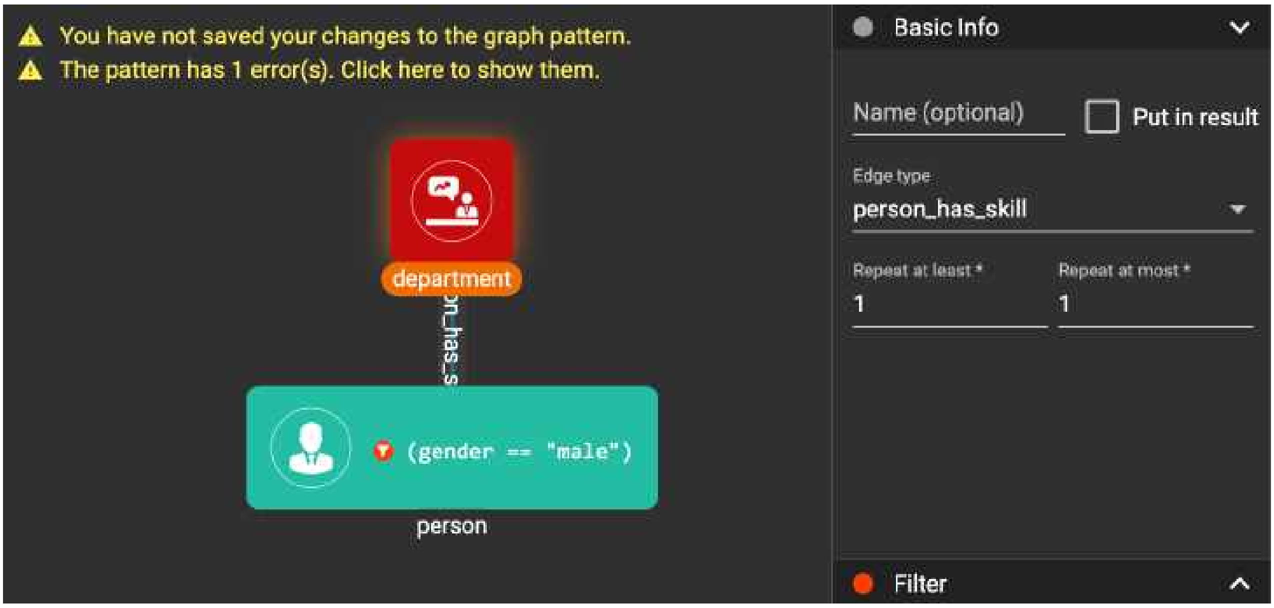
Click , and a new vertex pattern will be added to the visual pattern. You are in the editing mode of the newly added vertex pattern.



Add an edge pattern

Click , then click the source vertex pattern of the edge pattern:

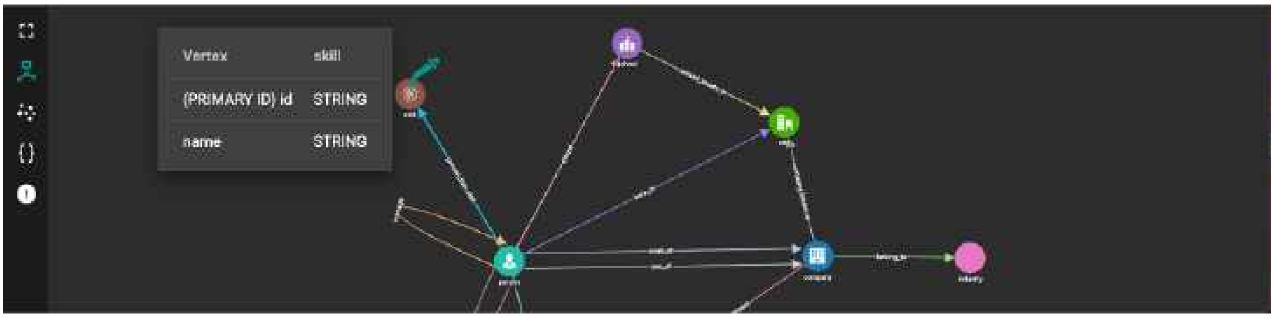
Then click the target vertex pattern of the edge pattern. A new edge pattern will be added to the visual pattern. You are in the editing mode of the newly added edge pattern:



Pick

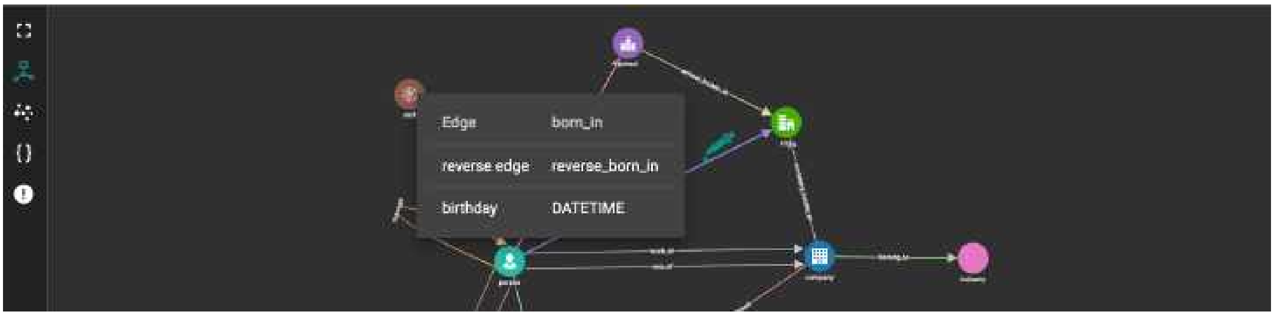
Pick is a fast way to build your visual pattern. You can pick from either graph schema or visual result.

Click , then click one vertex type in the graph schema tab:

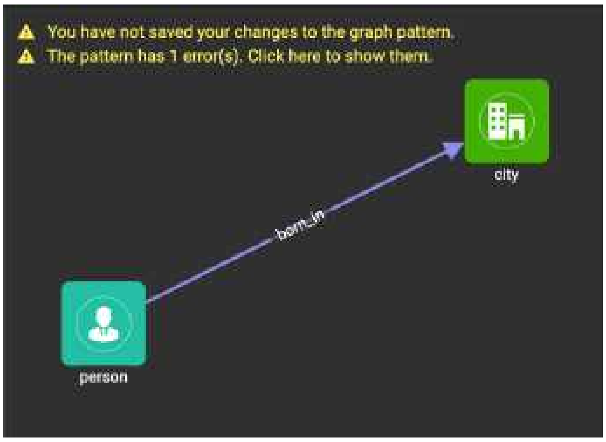


A vertex pattern will be added to the visual pattern:

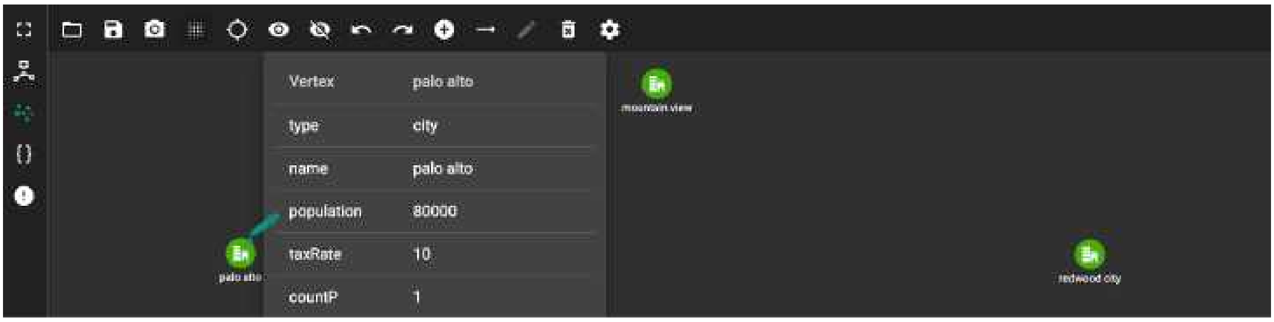
Click, then click one edge type in the graph schema tab:



An edge pattern together with two vertex patterns will be added to the visual pattern:



Click , then click one vertex in the visualize graph result tab:

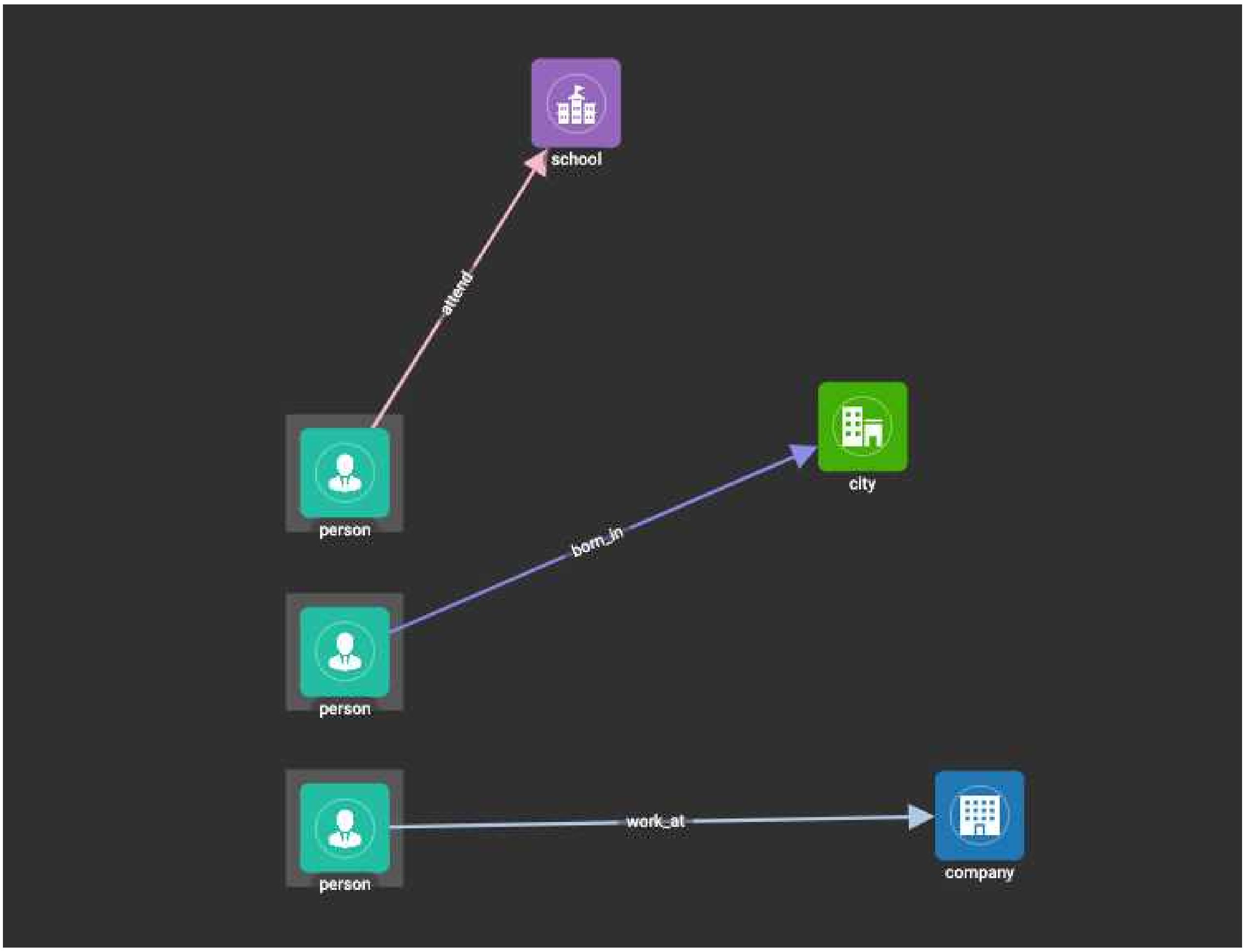


A vertex pattern will be added to the visual pattern. Note that the vertex pattern contains an id condition because it is picked from an actual vertex entity from the graph:

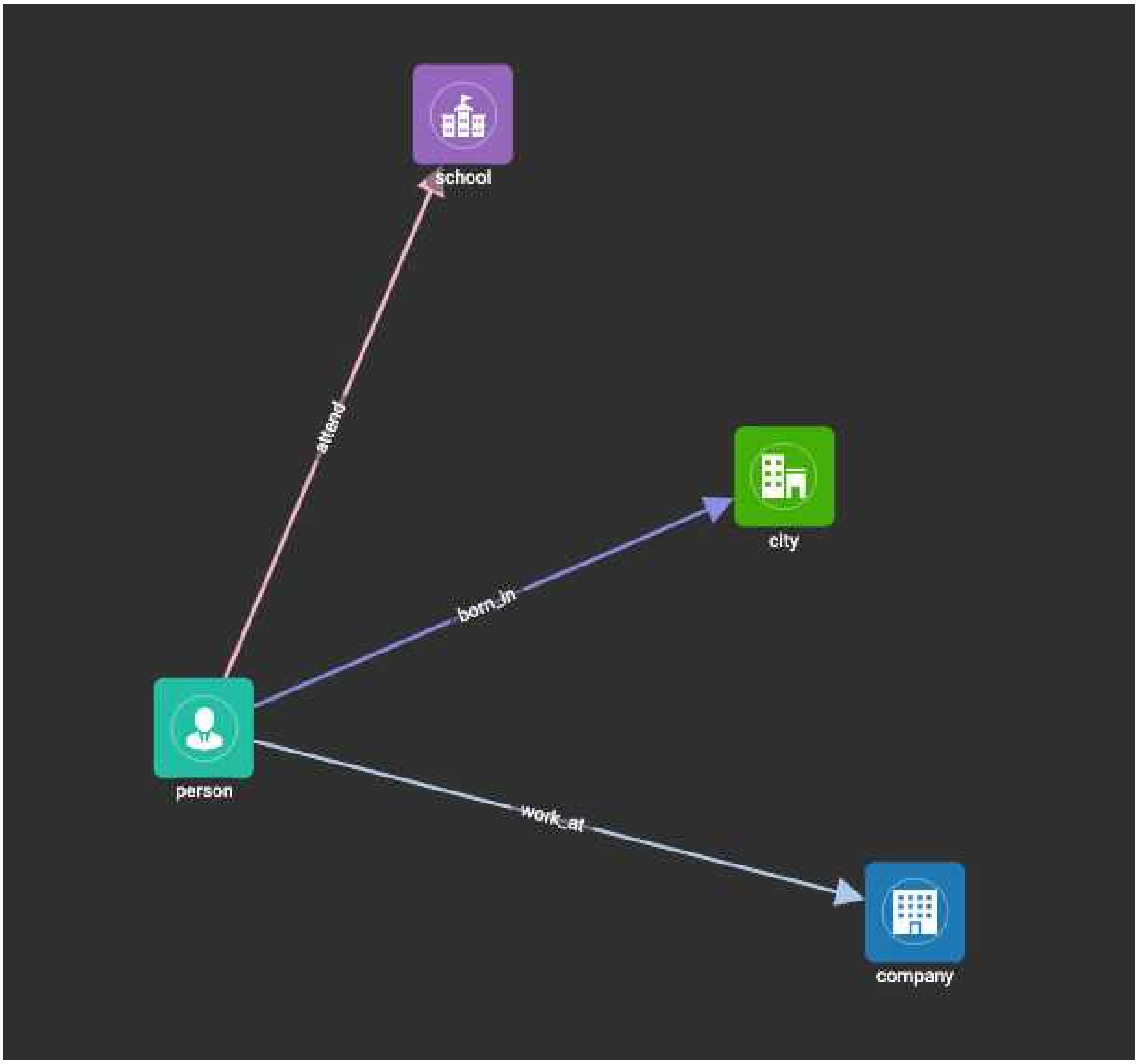
Merge

You can merge multiple vertex patterns of the same vertex type into one vertex pattern.

Hold Shift key to select multiple vertex patterns:



Then click , and you will get a larger visual pattern:



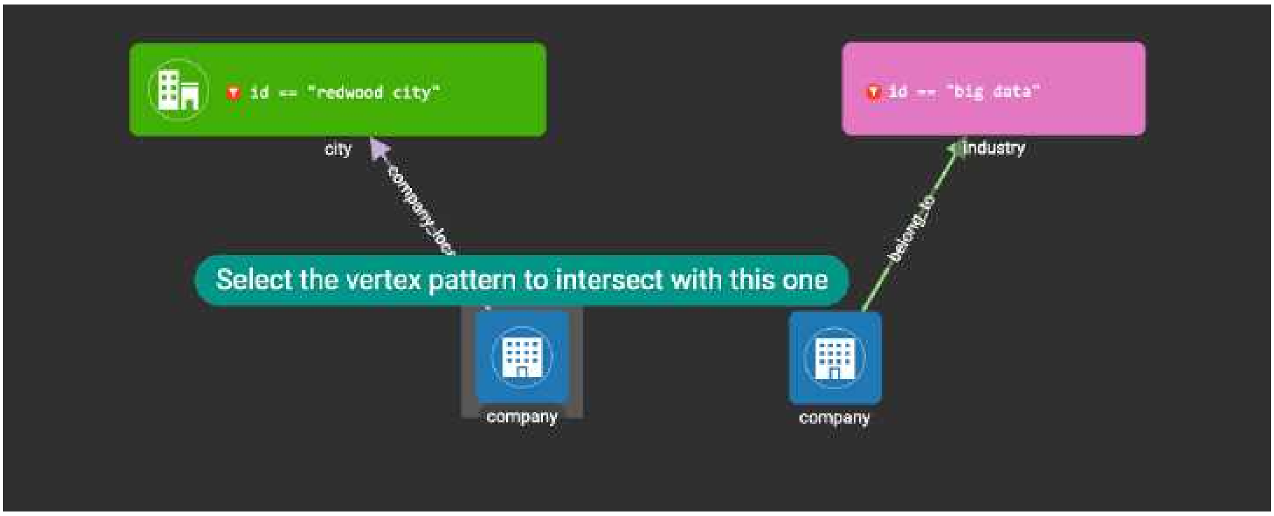
Use pick and merge together and you can create a complicated visual pattern quickly.

Widget

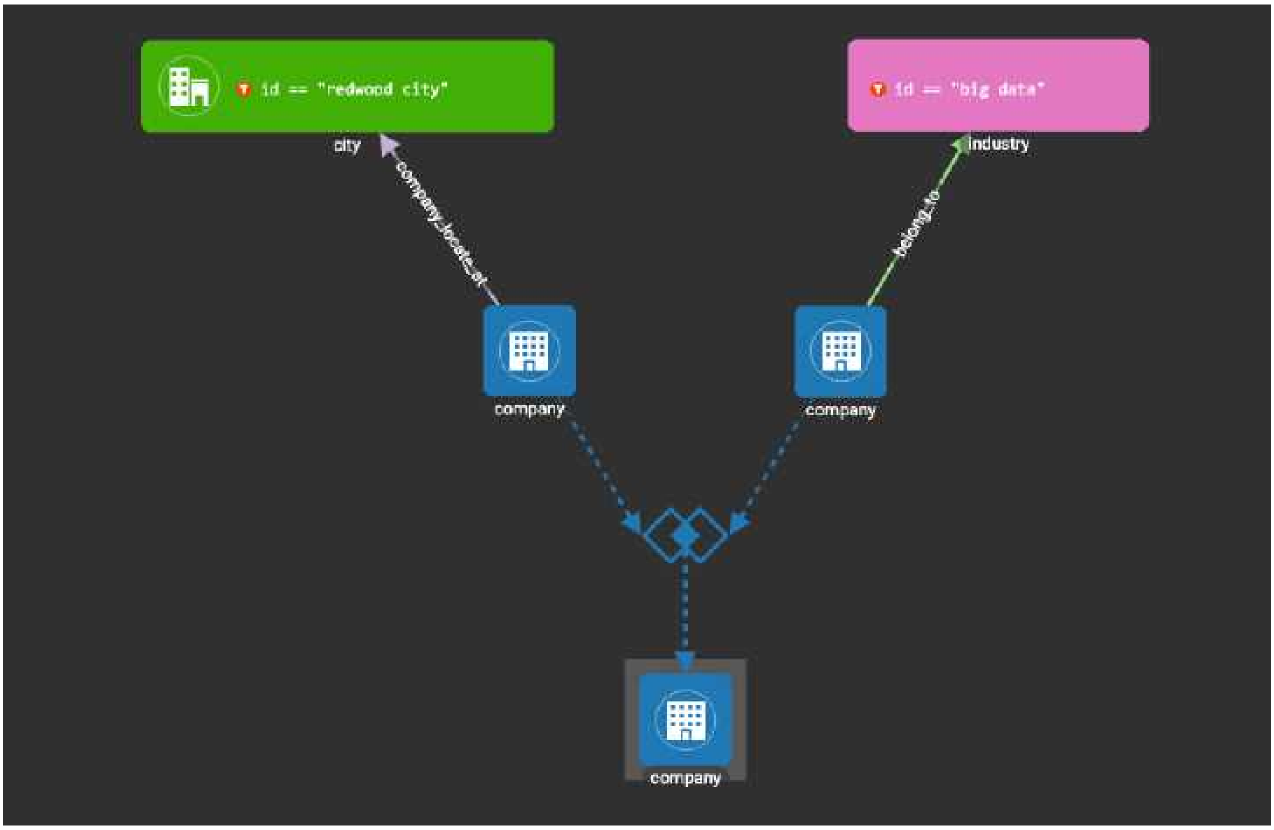
As described in the Basic Concepts, a visual pattern represents a graph pattern matching problem. In graph theory, graph pattern matching is declarative. However, graph pattern matching is not the full story. In a lot of cases you need to represent procedural computation flow. That's why we are introducing widgets.

Intersect

Click  , then click two vertex patterns of the same vertex type:



And an intersection widget is added to the visual pattern:

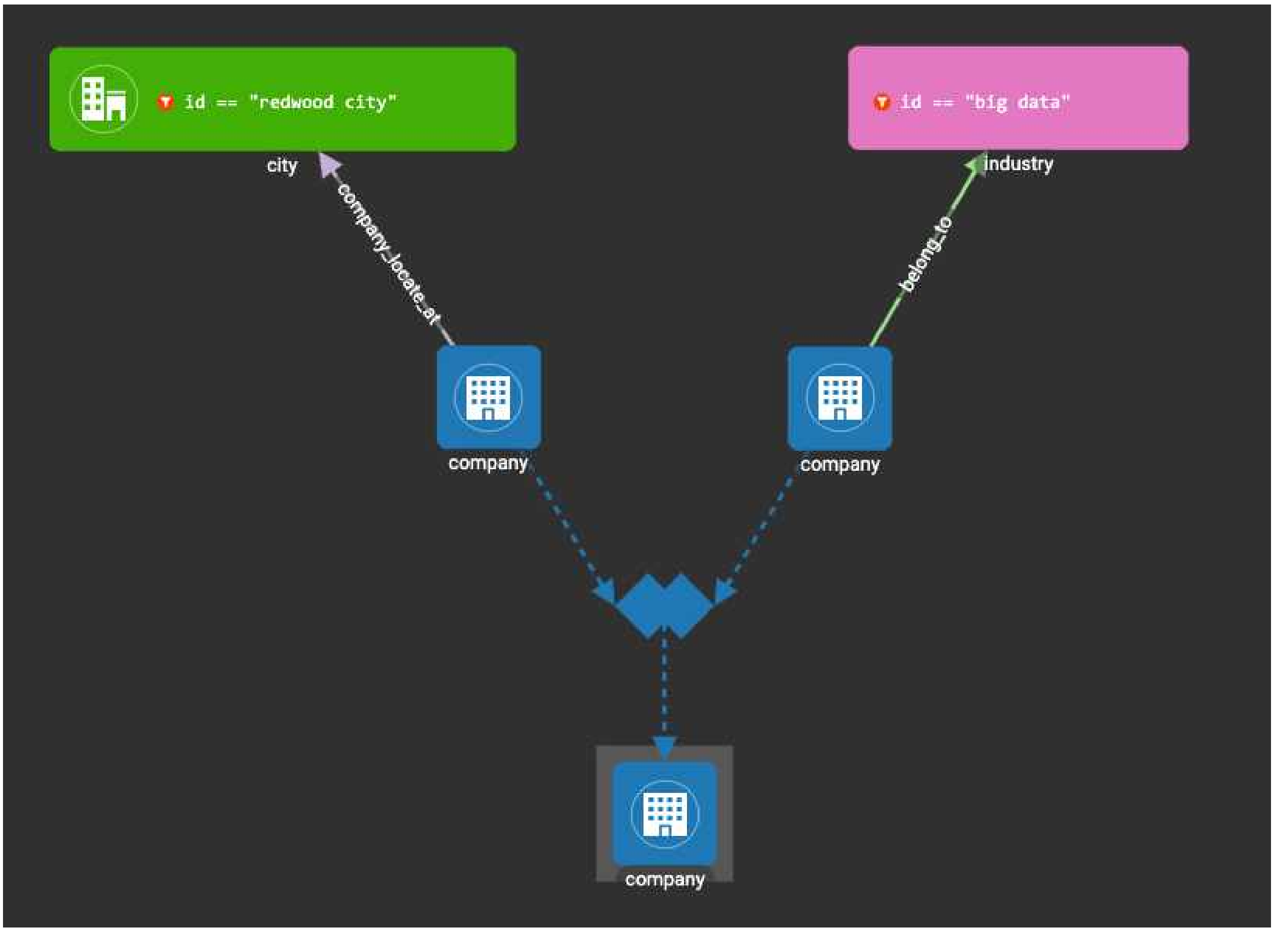


The output vertex pattern means matching all company vertices located at redwood city, **and** belongs to big data industry.

The output company vertex pattern can be part of another larger pattern. You can think of the input vertex patterns of the widget as constraints of the output vertex pattern.

Union

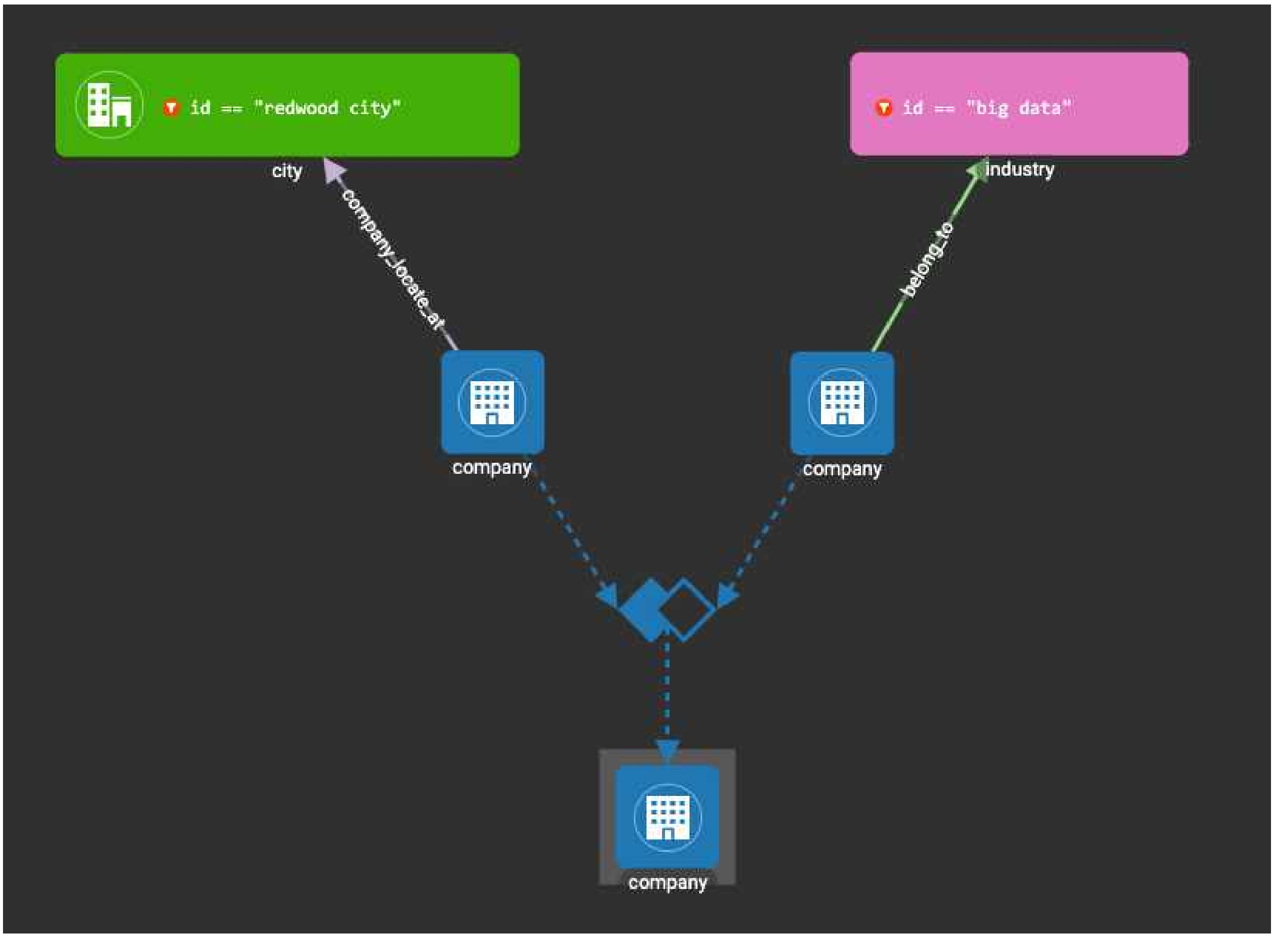
Click Union, then click two vertex patterns of the same vertex type. A union widget is added to the visual pattern:



The output vertex pattern means matching all company vertices located at redwood city, **or** belongs to big data industry.

Subtract

Click Subtract, then click two vertex patterns of the same vertex type. A subtract widget is added to the visual pattern:



The output vertex pattern means matching all company vertices located at redwood city, **but not** belong to big data industry.

Next Step

Now you have a basic idea about all different functionalities of Visual Query Builder. Let's go to the next page to walk through how to build some visual patterns to solve your business questions!

Visual Pattern Examples

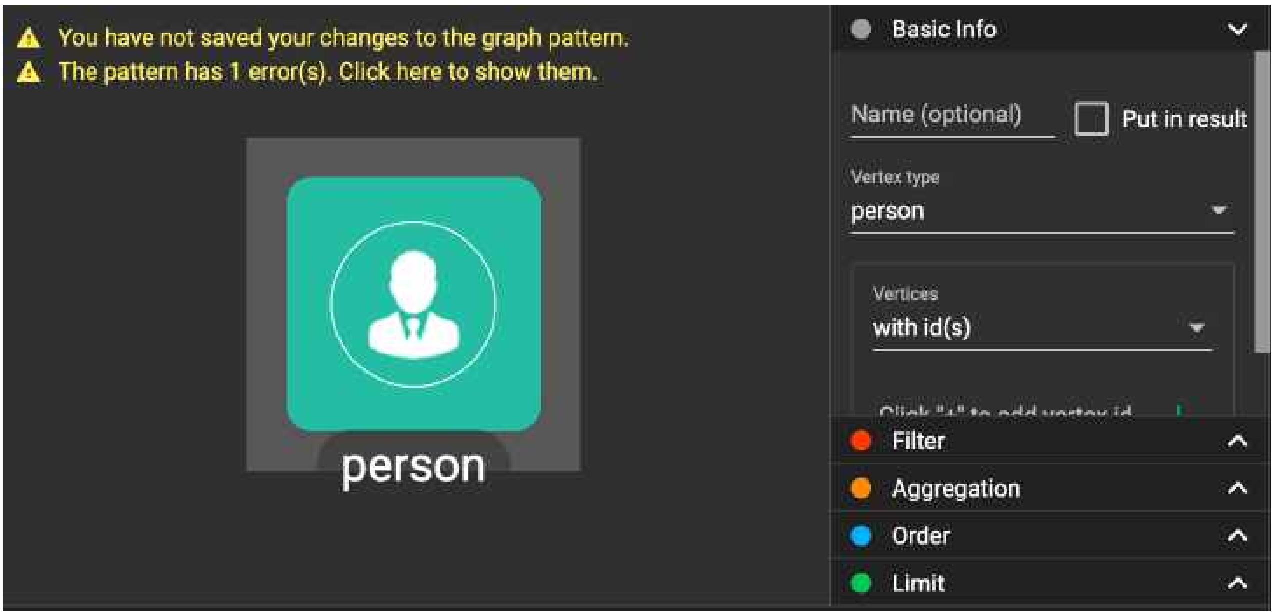
Visual Pattern Examples

You have learnt all the basic concepts of Visual Query Builder in the previous article. In this article we will walk you through building several visual patterns step by step, and then show you more graph analytics questions and how to solve them with visual patterns.

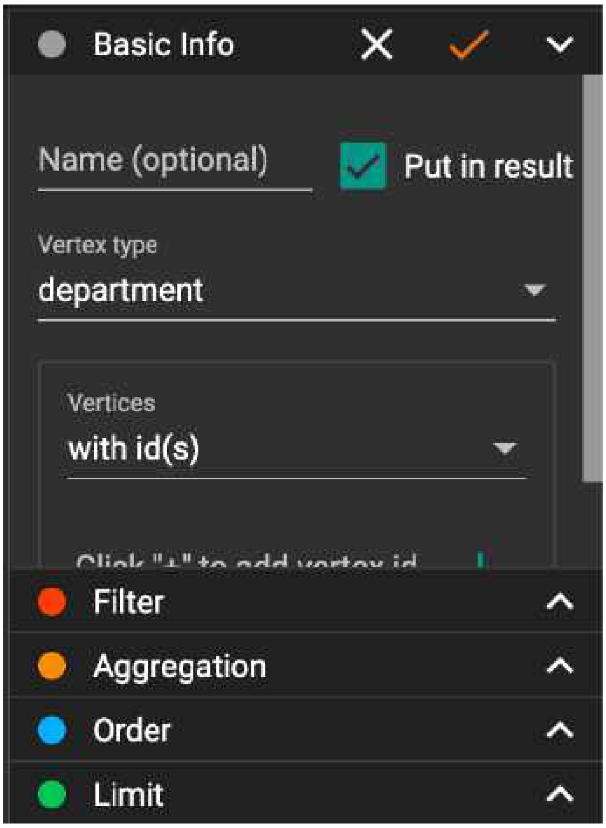
Example 1. Find all the departments with male employees.

Clickin the pattern list pattern, input the pattern name, and click:

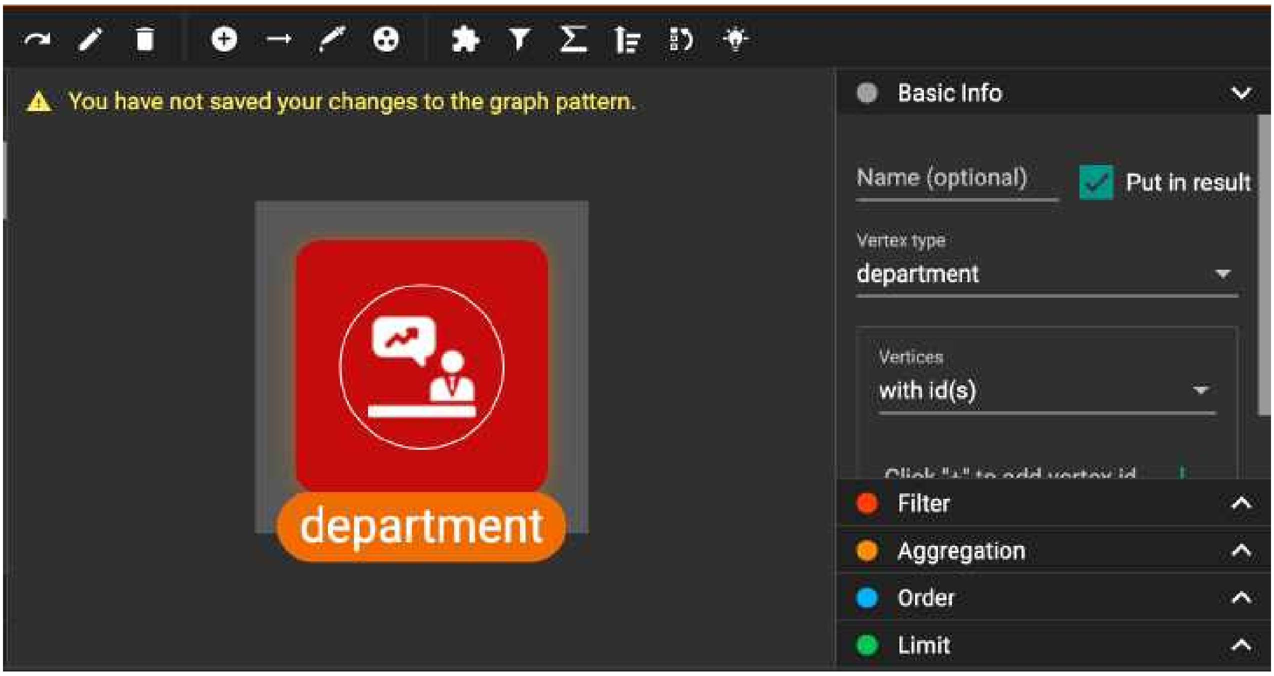
Click  in the toolbar to add a vertex pattern:



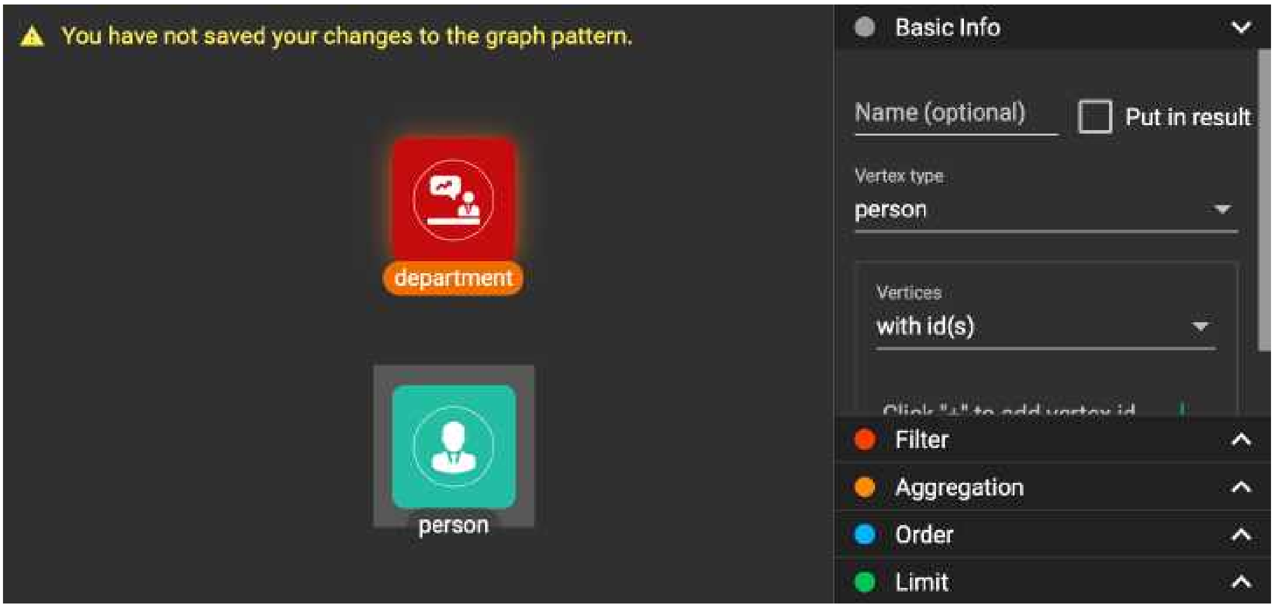
In the config panel, change the vertex type to be department, and put it in result:



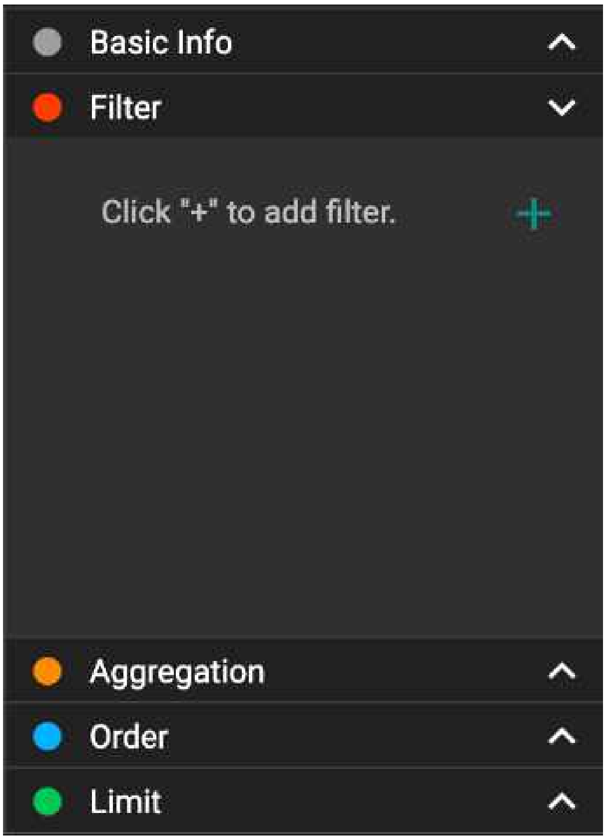
Confirm the change by click :



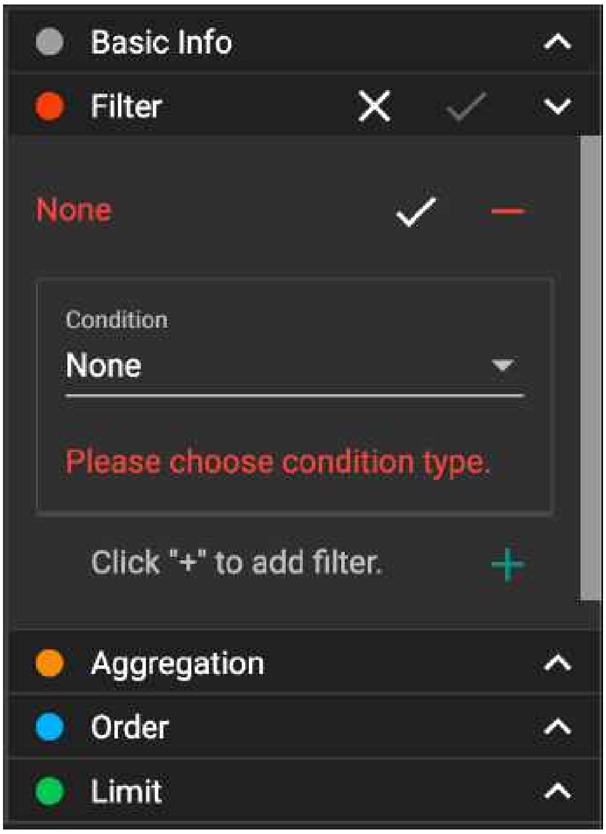
Click in the toolbar to add another vertex pattern:



Expand the Filter section in the config panel:

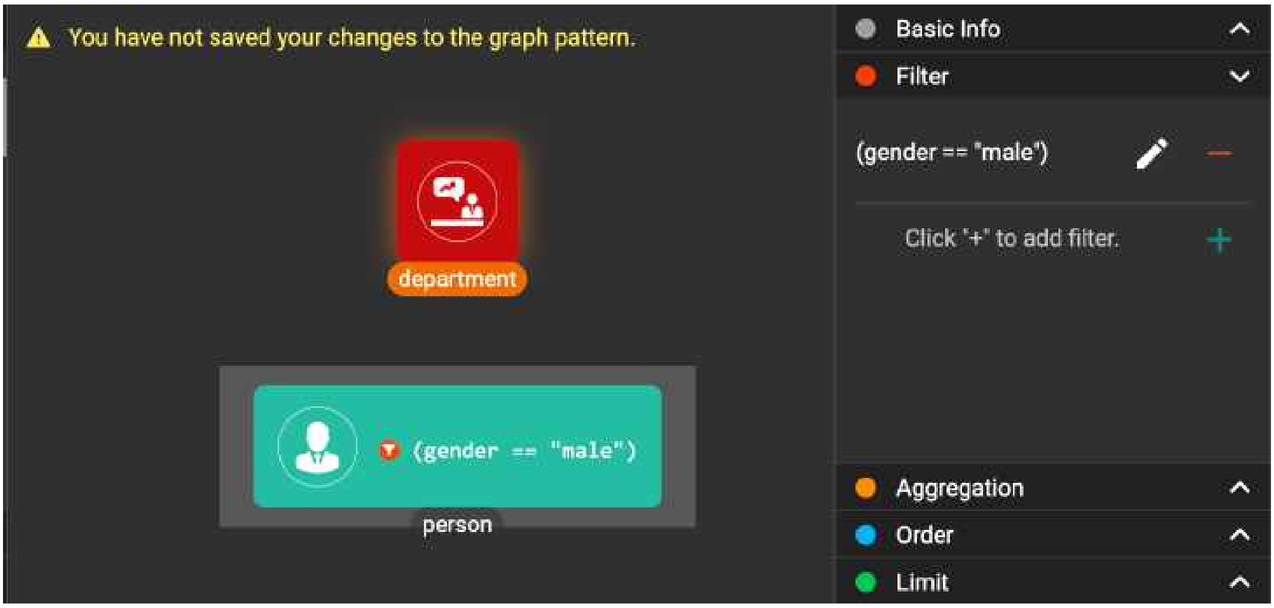


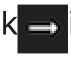
Click  to add a filter:

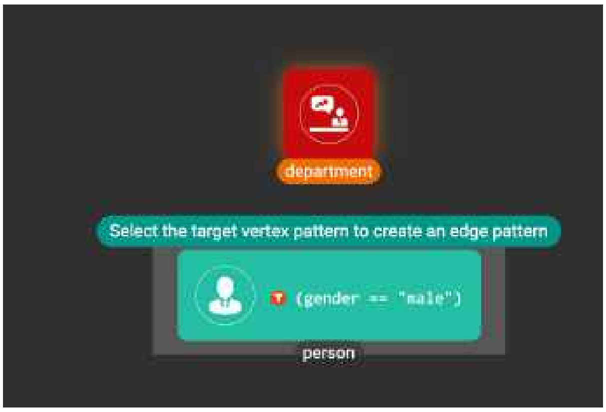


Edit the filter to be **(gender == "male")**:

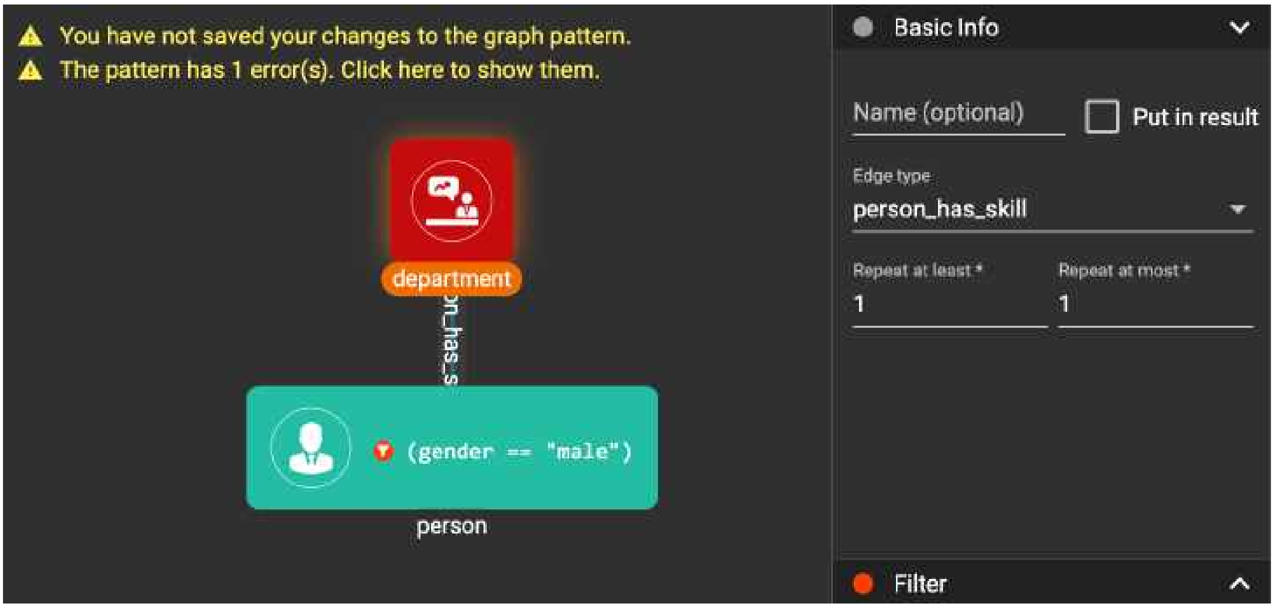
Confirm the change:



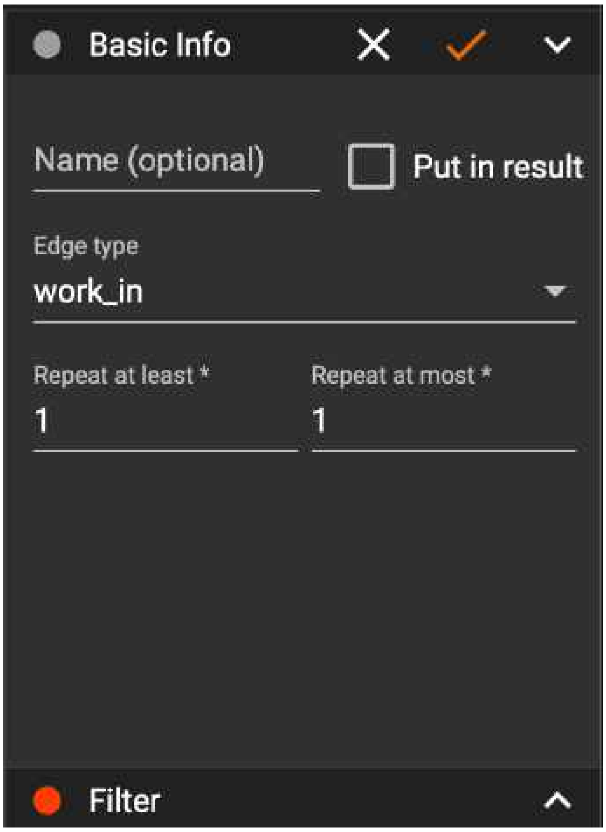
Click  in the toolbar, then click the person vertex pattern in the working panel:



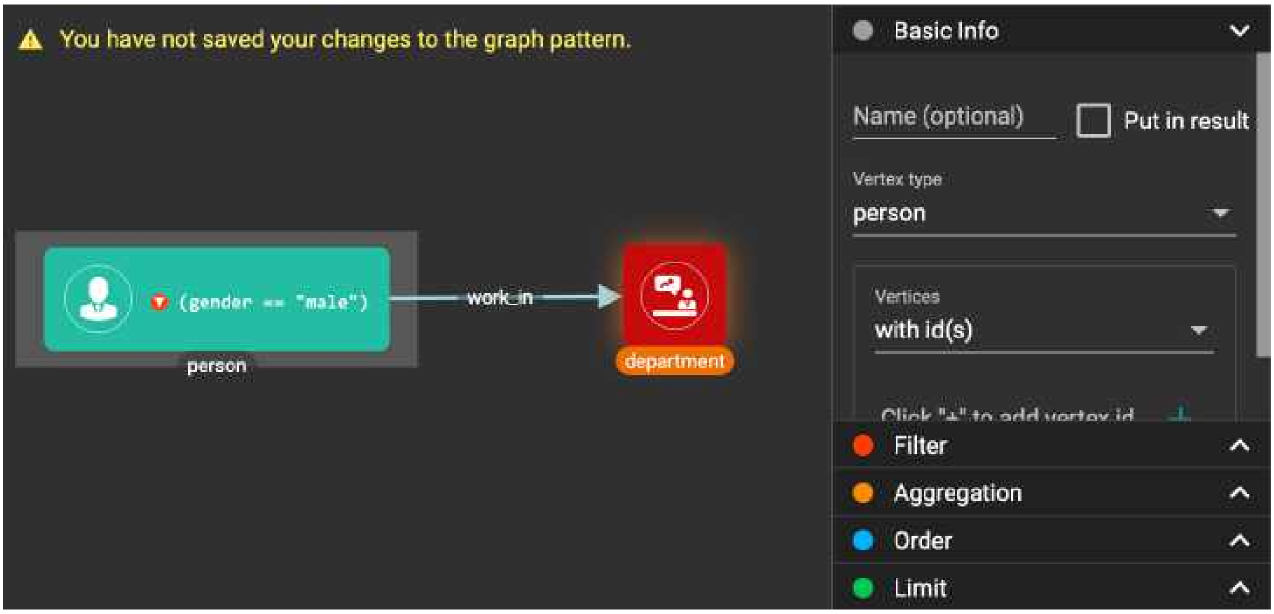
Then click department vertex pattern in the working panel to create an edge pattern between the two vertex patterns:



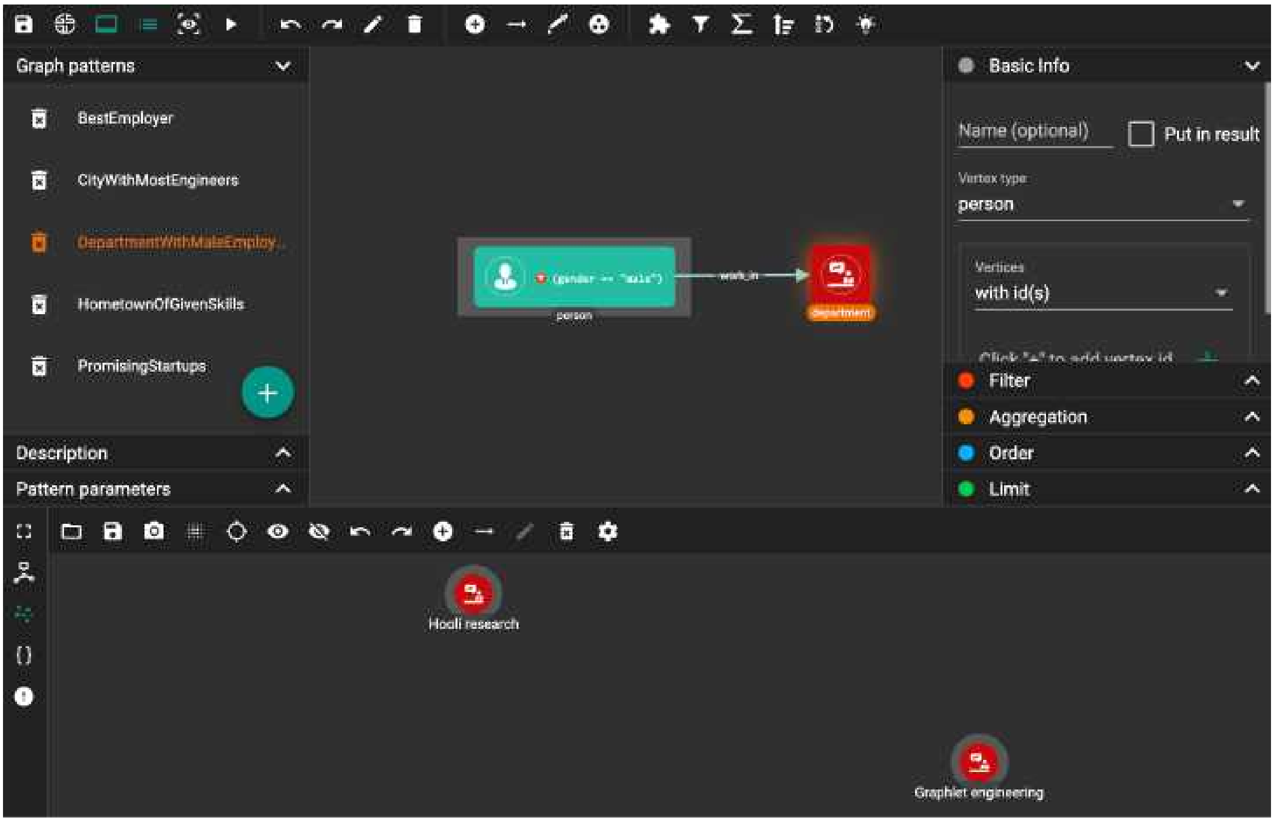
In the config panel, change the edge type to be work_in:



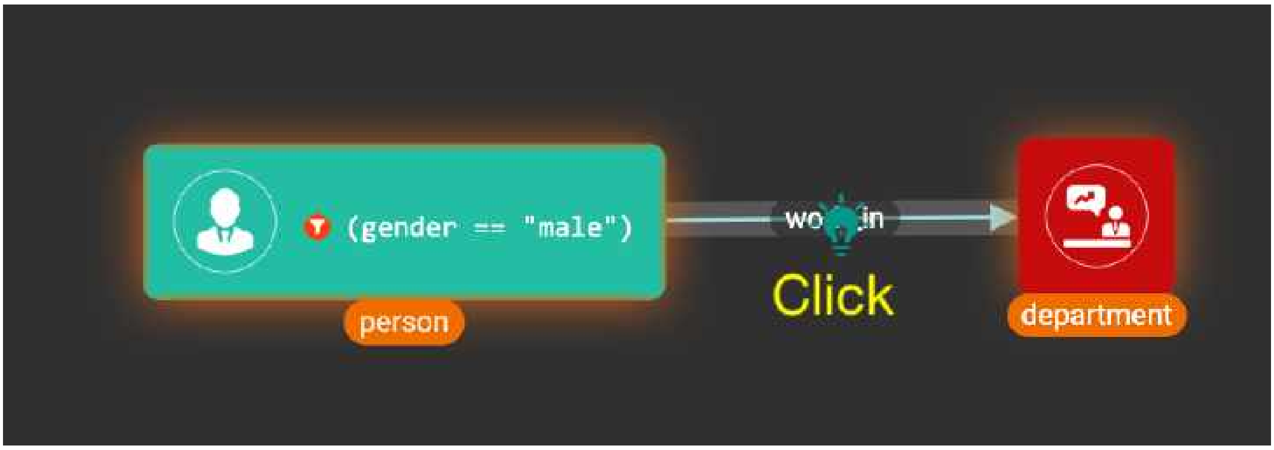
Apply the change, and drag the person vertex pattern to a better position:



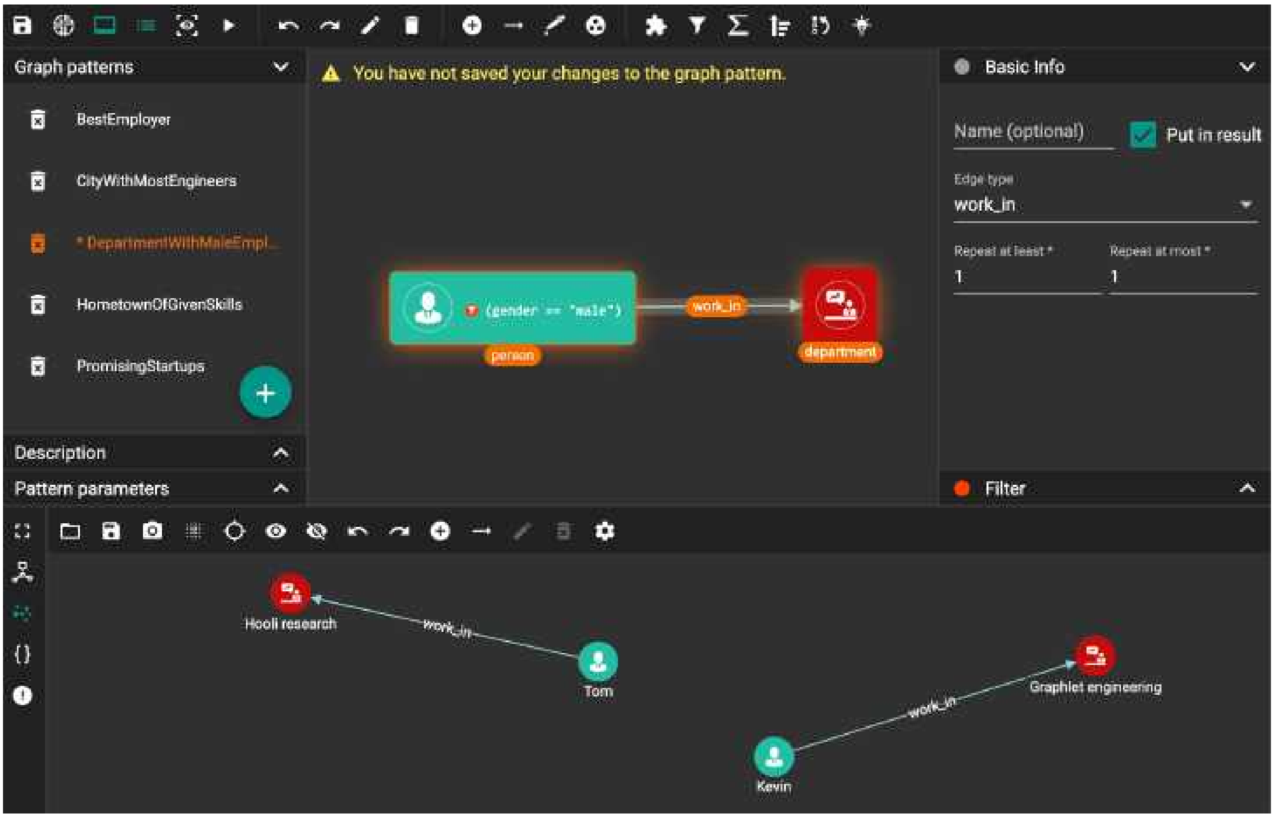
Click in the toolbar to save the pattern, and click in the toolbar to run the pattern:



You can also try to output the entire matched subgraphs. Click in the toolbar to enter output toggle mode, then click the person vertex pattern and the work_in edge pattern:



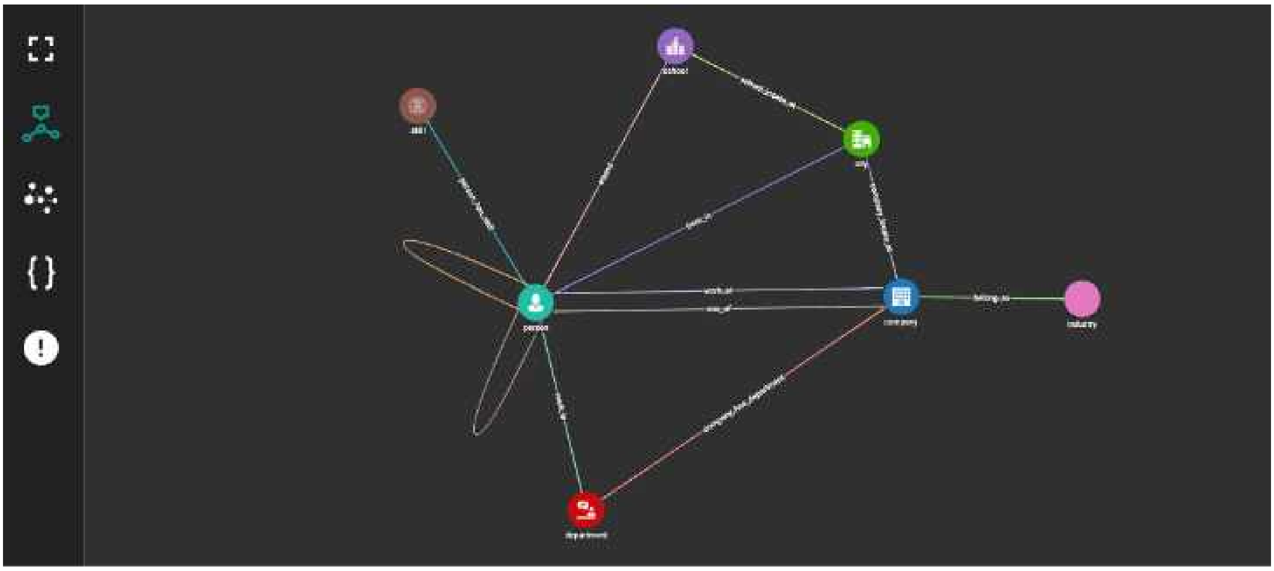
Click in the toolbar again to exit output toggle mode, then click to run the pattern:



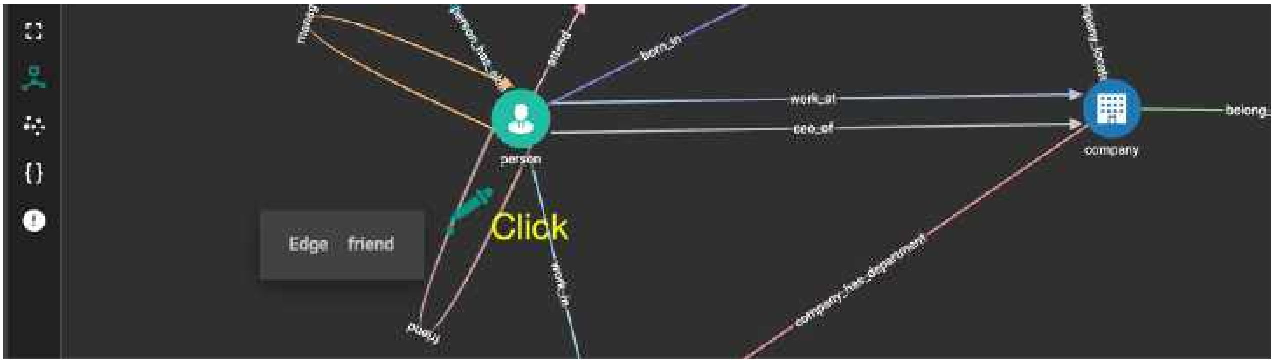
Example 2. Find all such people A, B, C: A and B are friends, A and C are friends, and B manages C.

Create a new visual pattern named FriendsManagerABC.

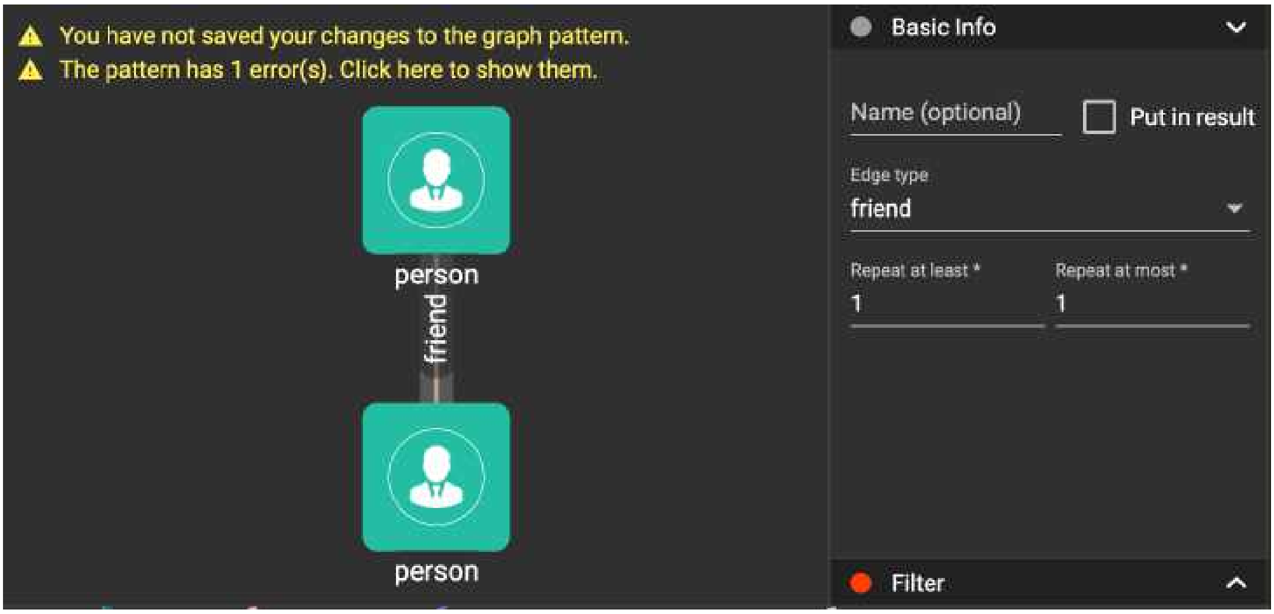
Click in the console panel to switch to graph schema:



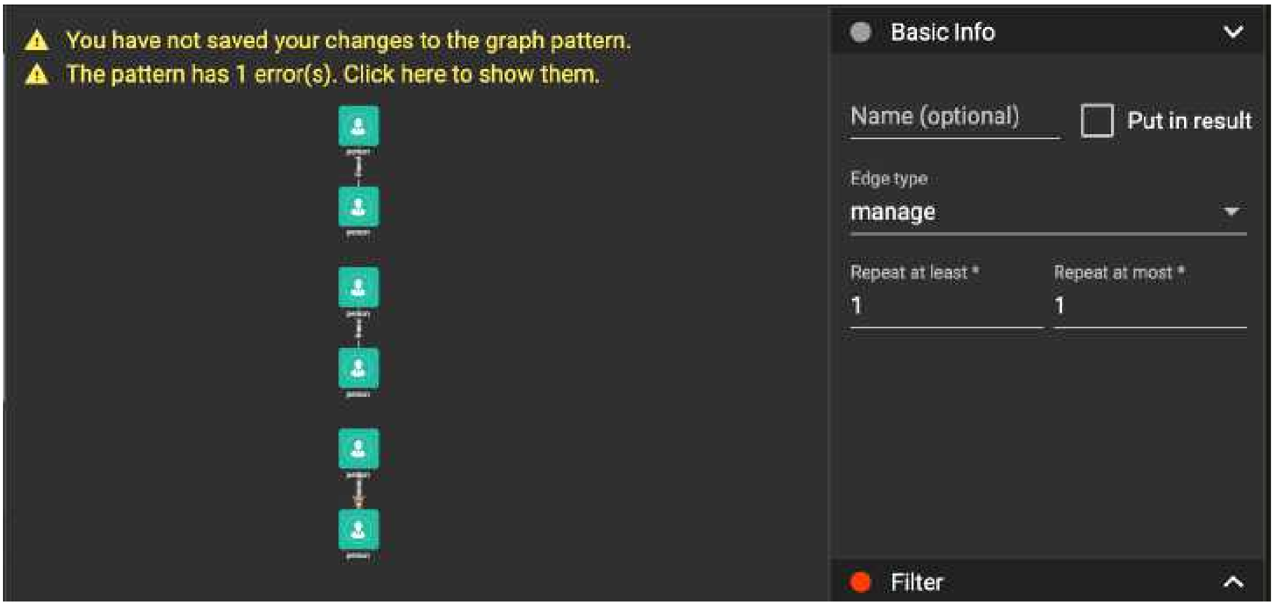
Click  in the toolbar, then click the friend edge type in the graph schema:



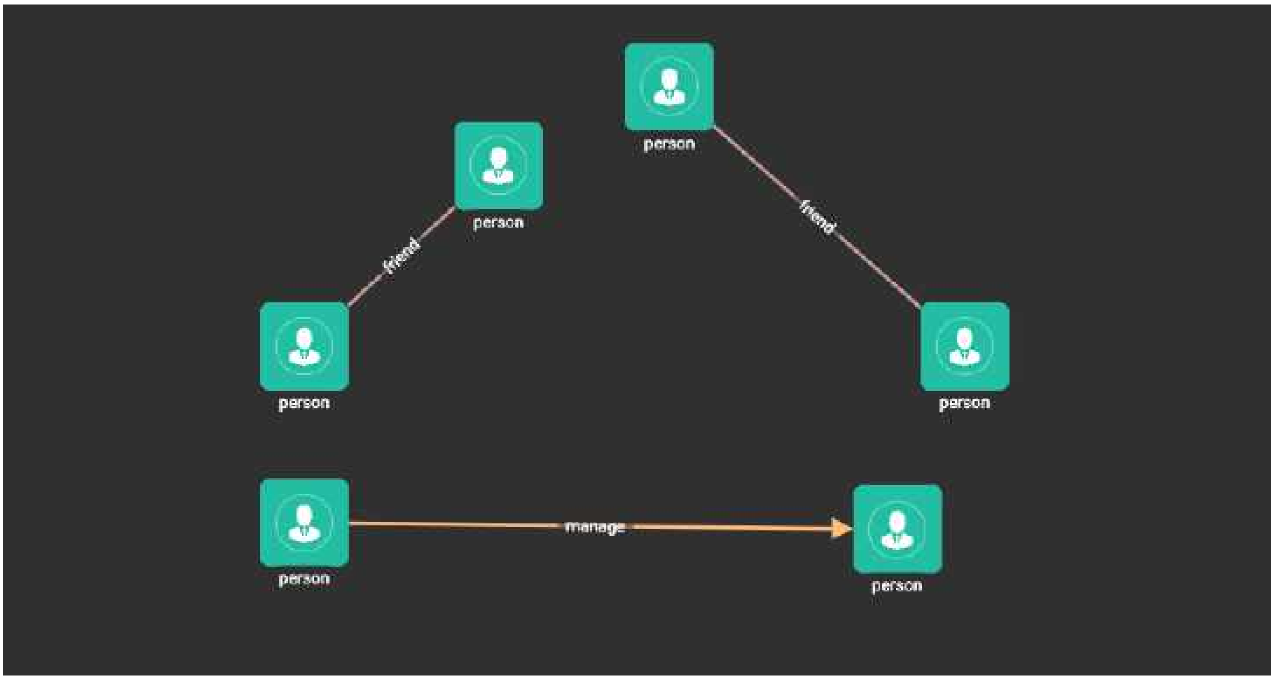
An edge pattern is added into the working panel:



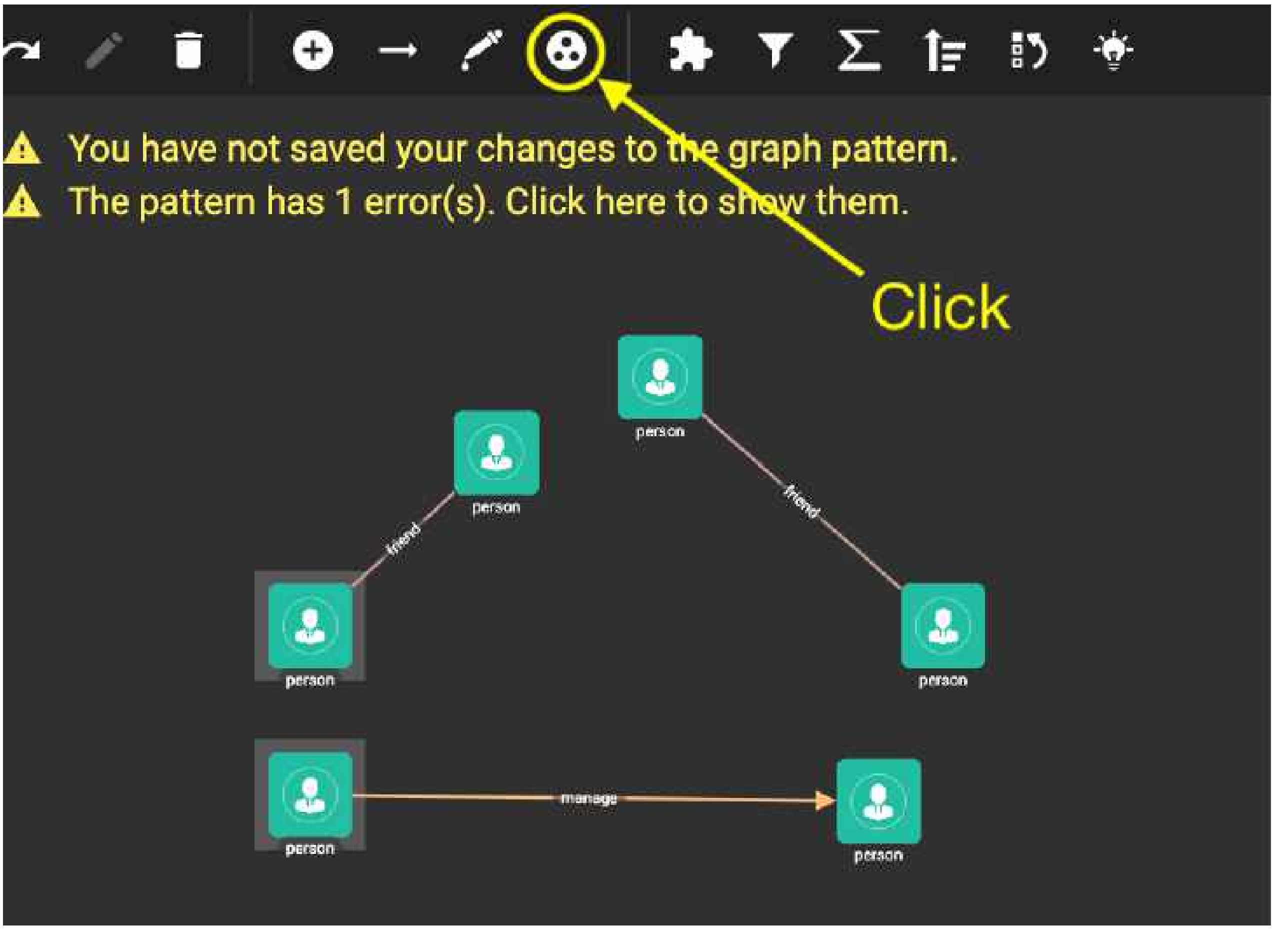
Similarly, use to add another friend edge pattern and a manage edge pattern into the working panel:



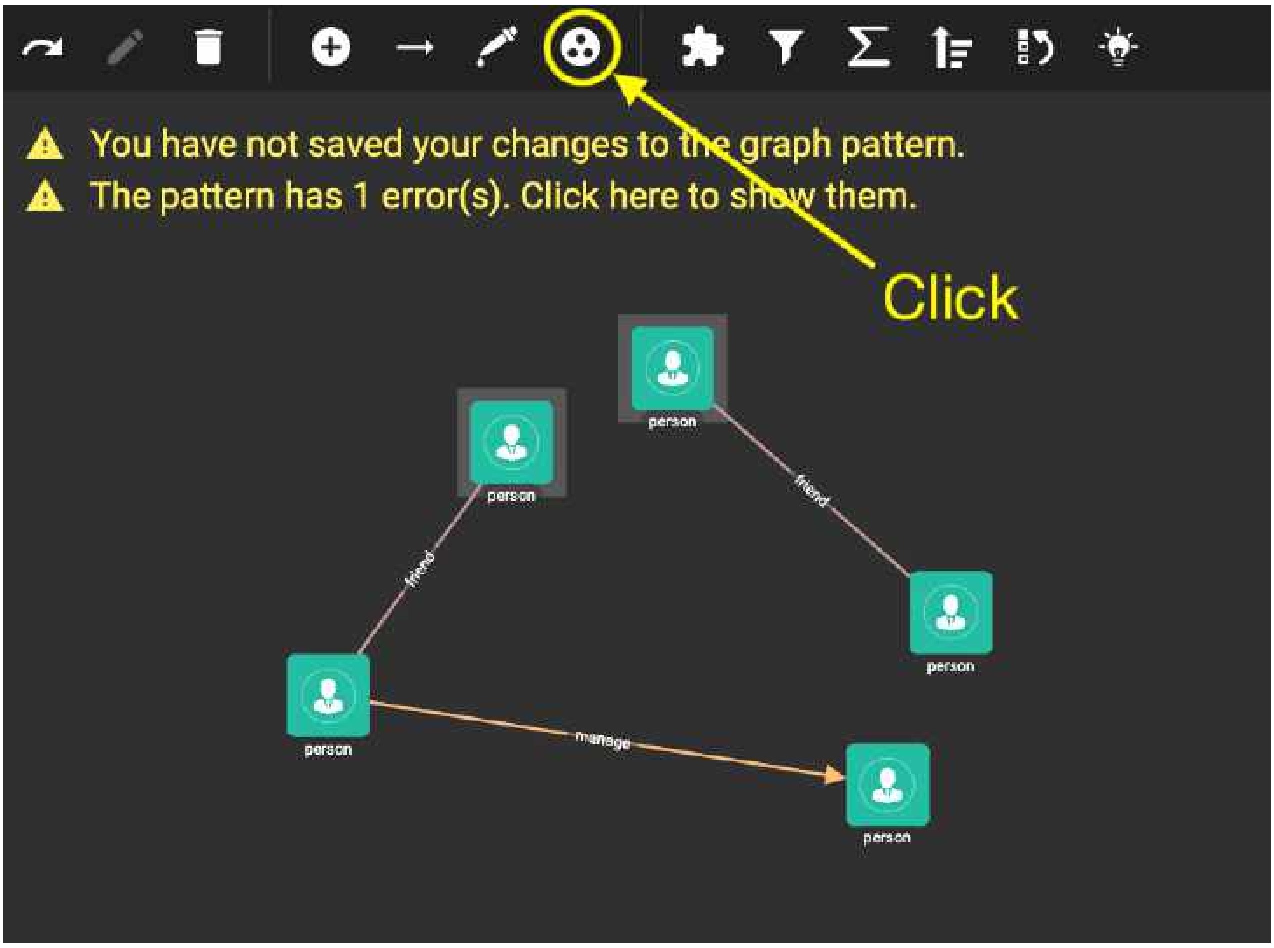
Adjust the vertex patterns' positions:



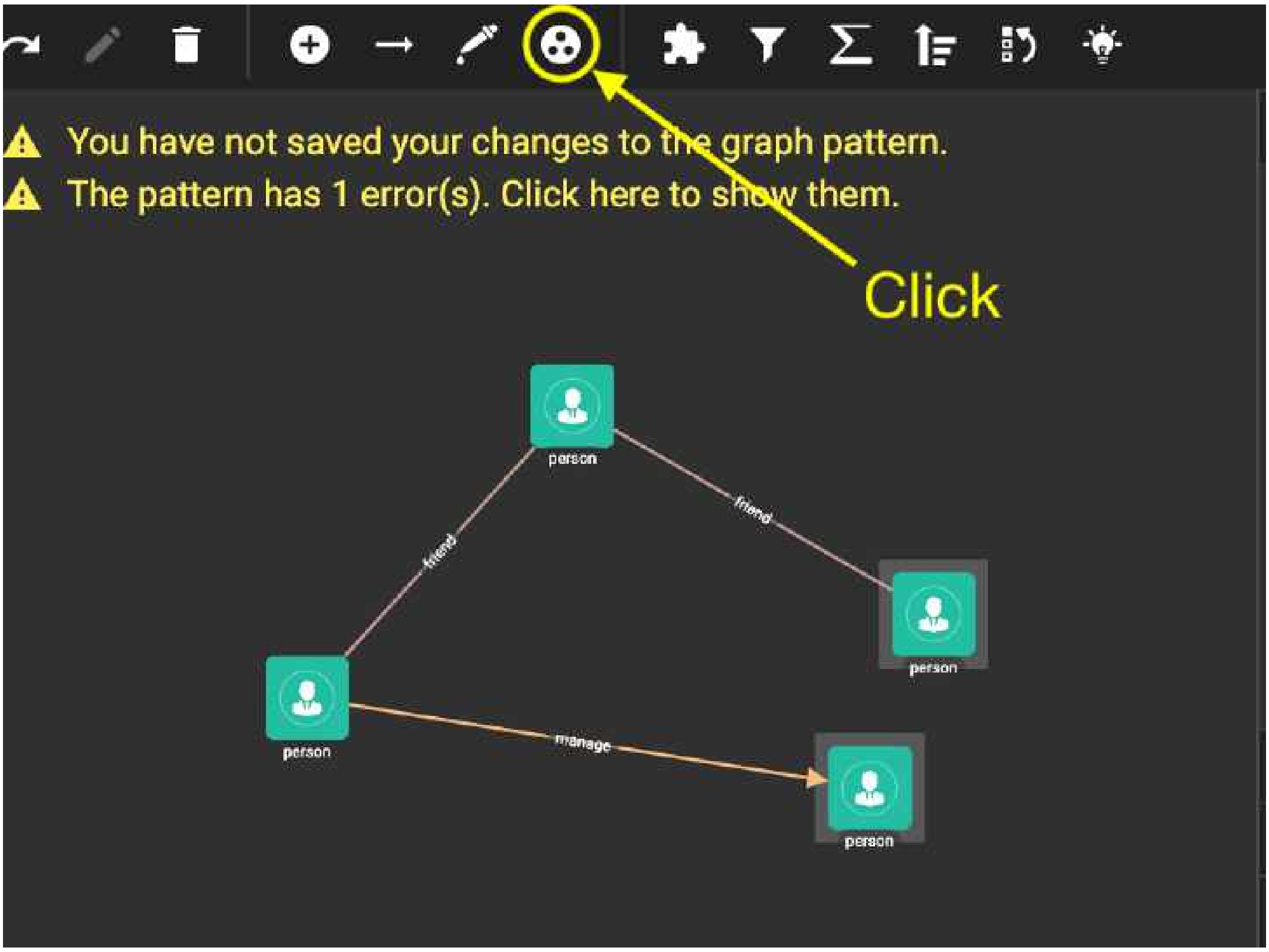
Hold Shift key and choose the two vertex patterns on the left, then click in the toolbar to merge them together:



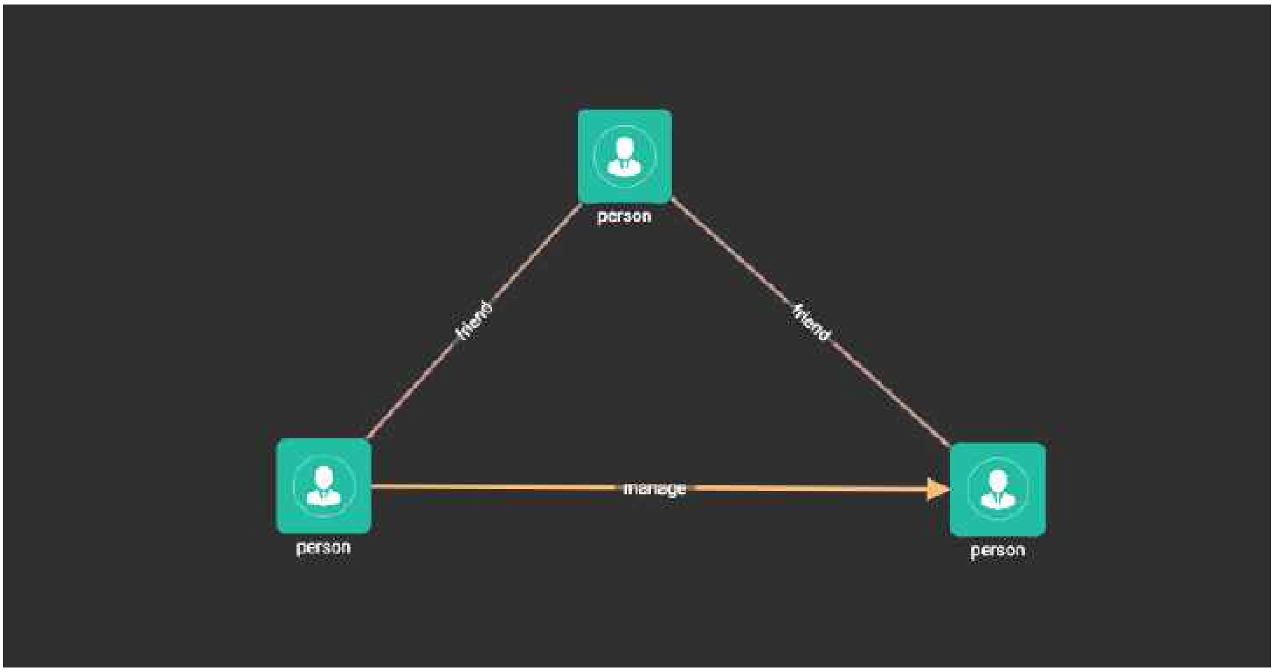
Then, merge the top two vertex patterns:



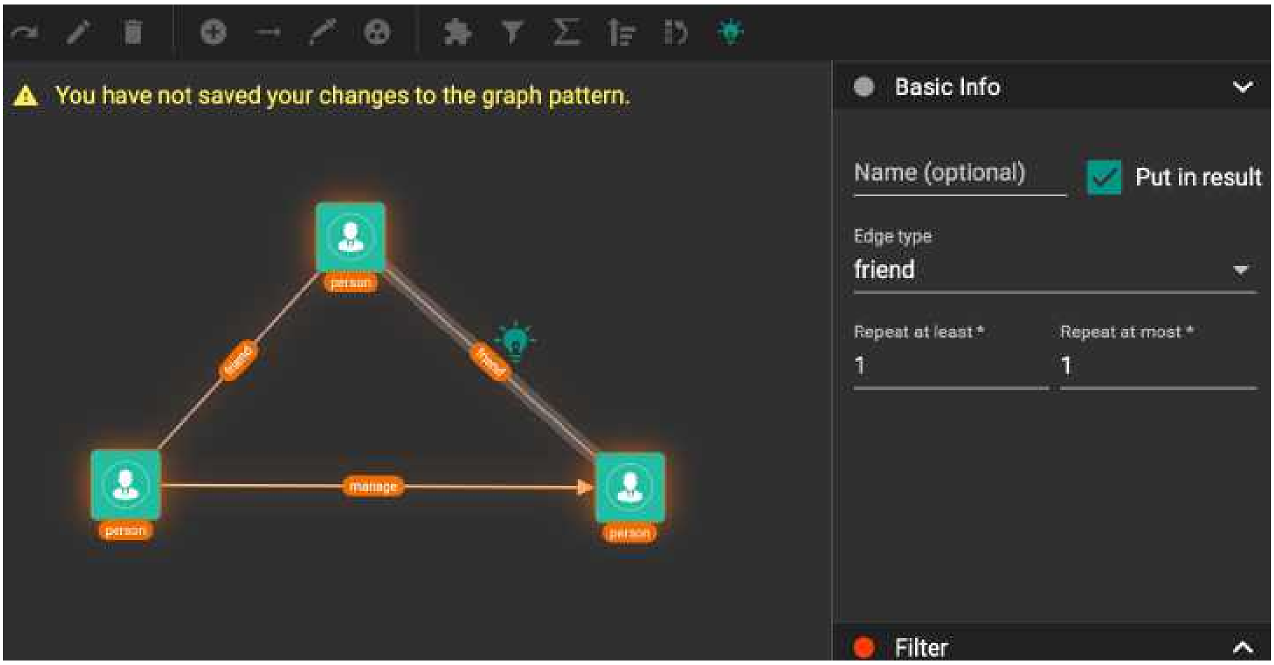
Then, merge the right two vertex patterns:



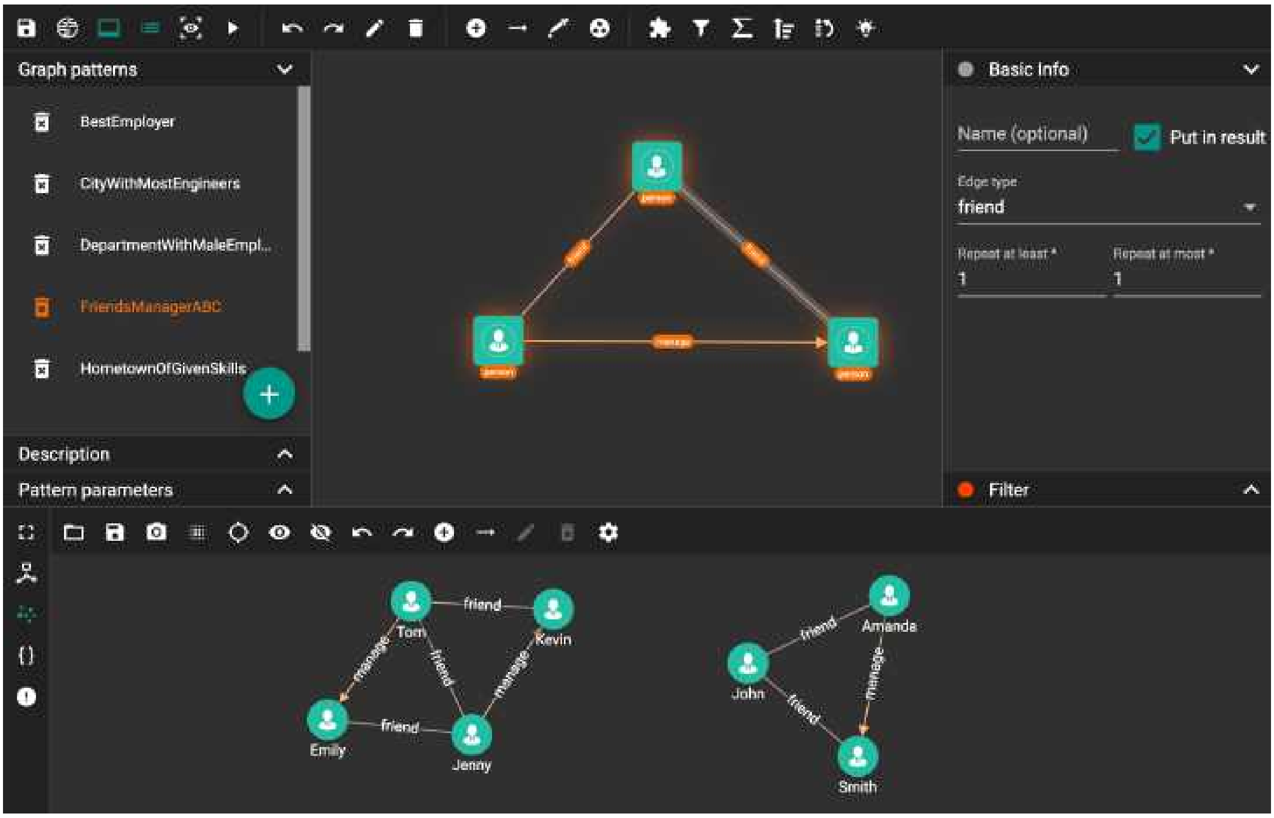
Now you have a triangle visual pattern representing the three people relationship!



Now, click in the toolbar and put all the vertex patterns and edge patterns into result:



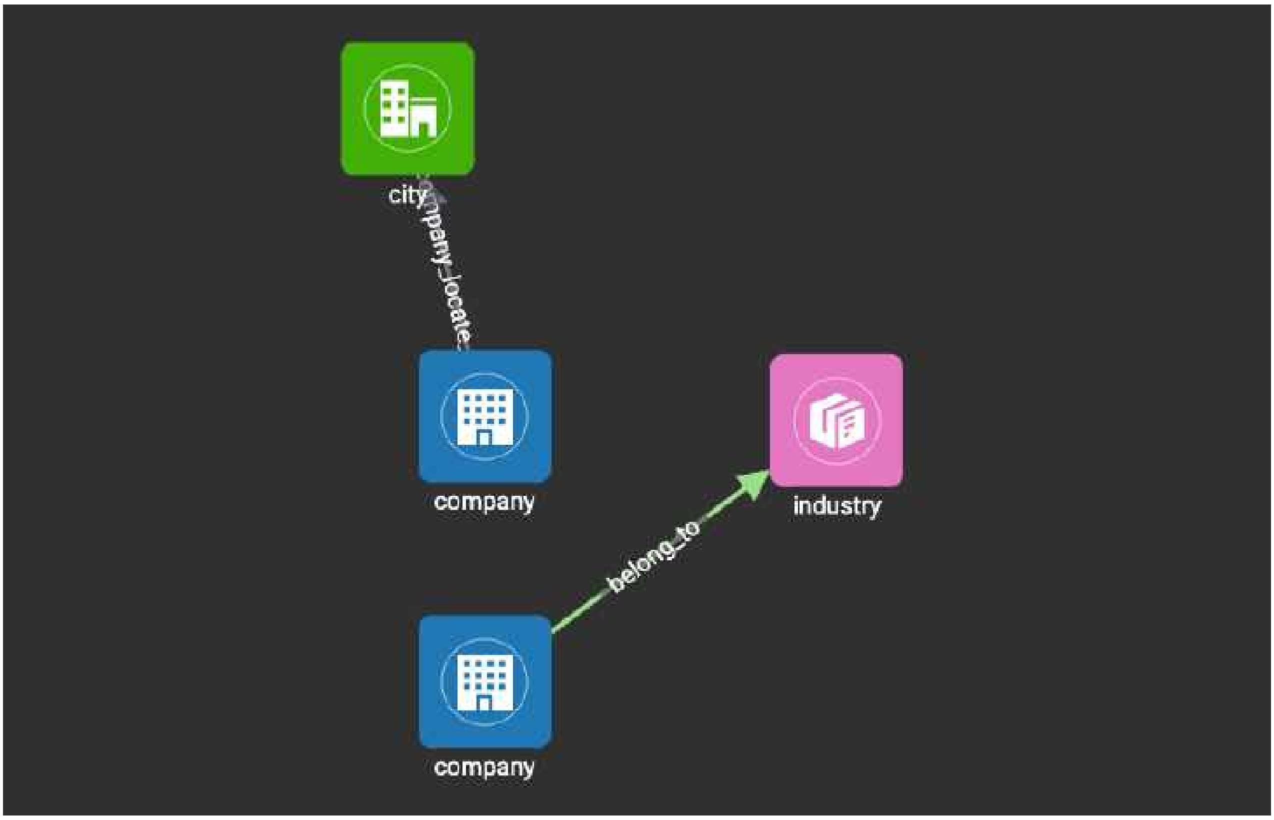
Click  again to exit output toggle mode, click  to save the pattern, and click  to run the pattern:



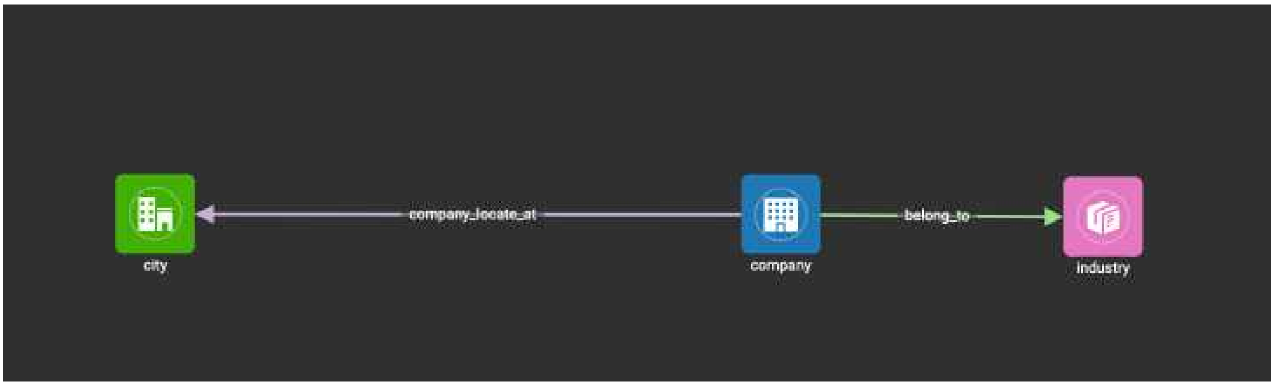
Example 3. Find the top 2 cities where the most number of companies of an industry are located in.

Create a new visual pattern named IndustryCenterCity. Click in the console panel to switch to graph schema.

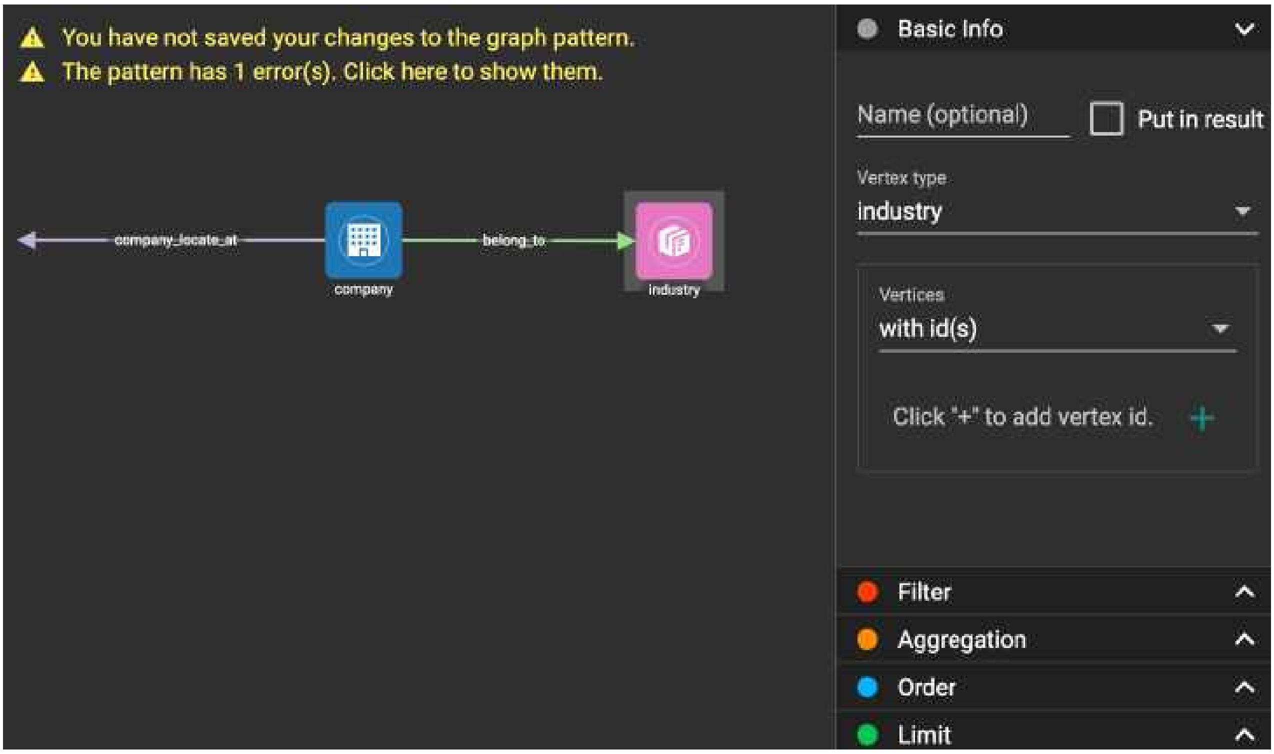
Use in the toolbar to pick the company_locate_at edge type and belong_to edge type in graph schema:



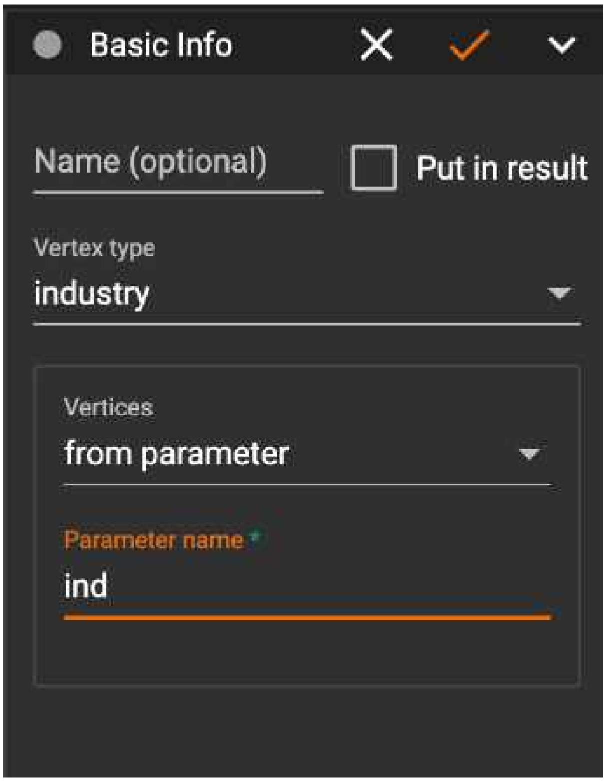
Use Shift key to select the two company vertex patterns, and click to merge them into one. Drag the vertex patterns to make their positions look good:



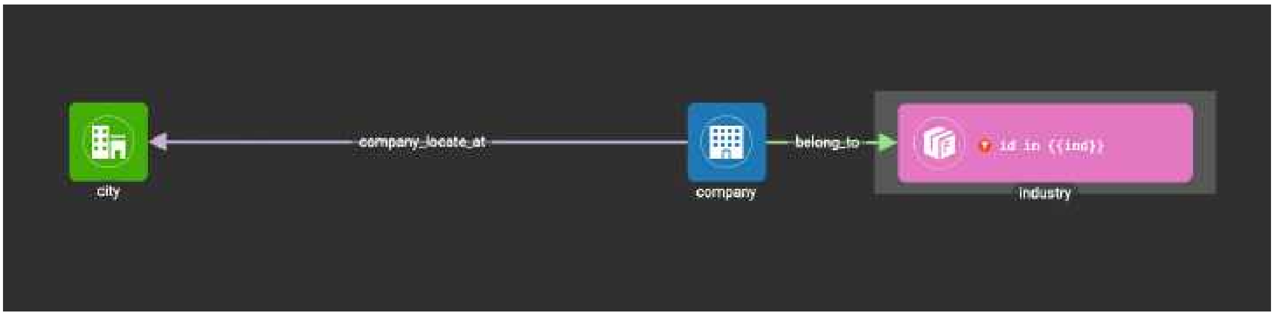
Click the industry vertex pattern, then click in the toolbar (or double click the industry vertex pattern), now you are in edit mode:



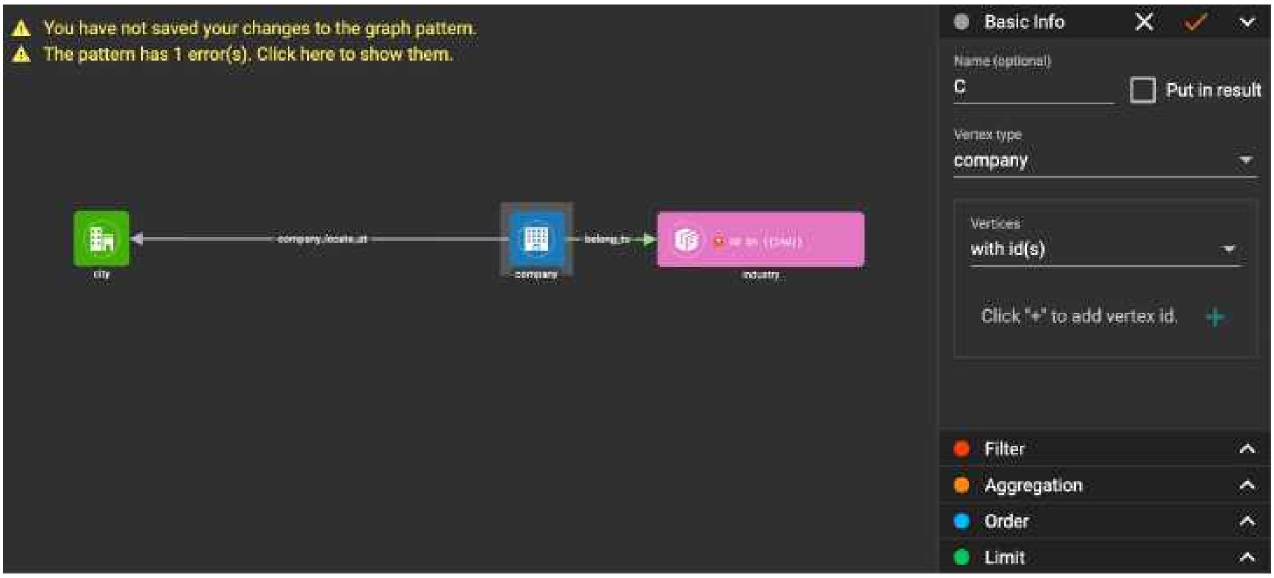
Change the vertices to be from parameter, and give a parameter name **ind**. This annotates the industry as an input vertex set when running the pattern:



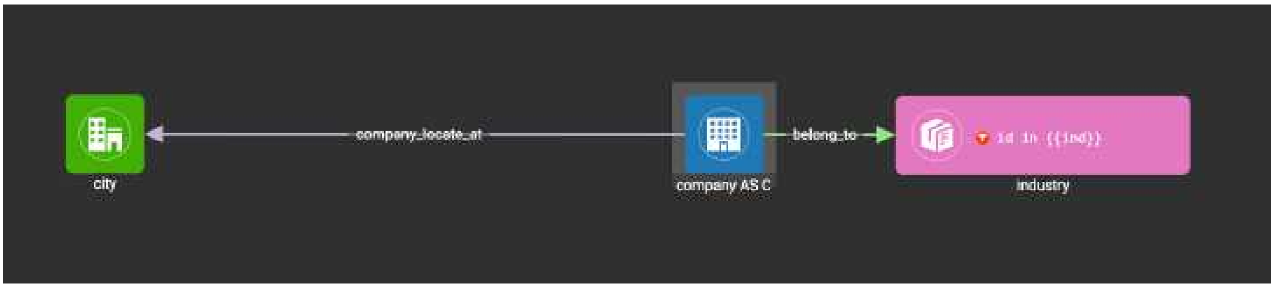
Confirm the change, and notice there is an id condition on the industry vertex pattern:



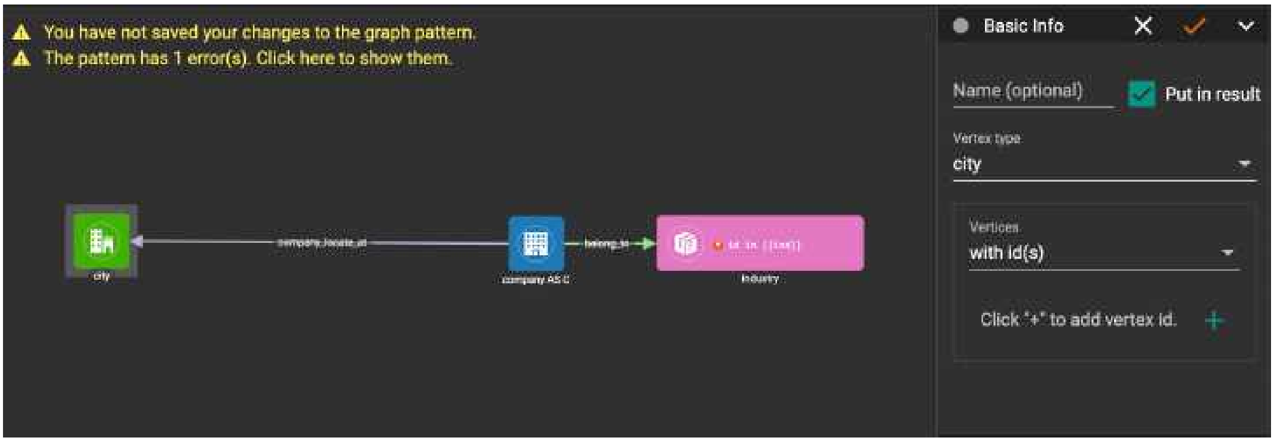
Click the company vertex pattern, then click in the toolbar, and change the Name to **C**:



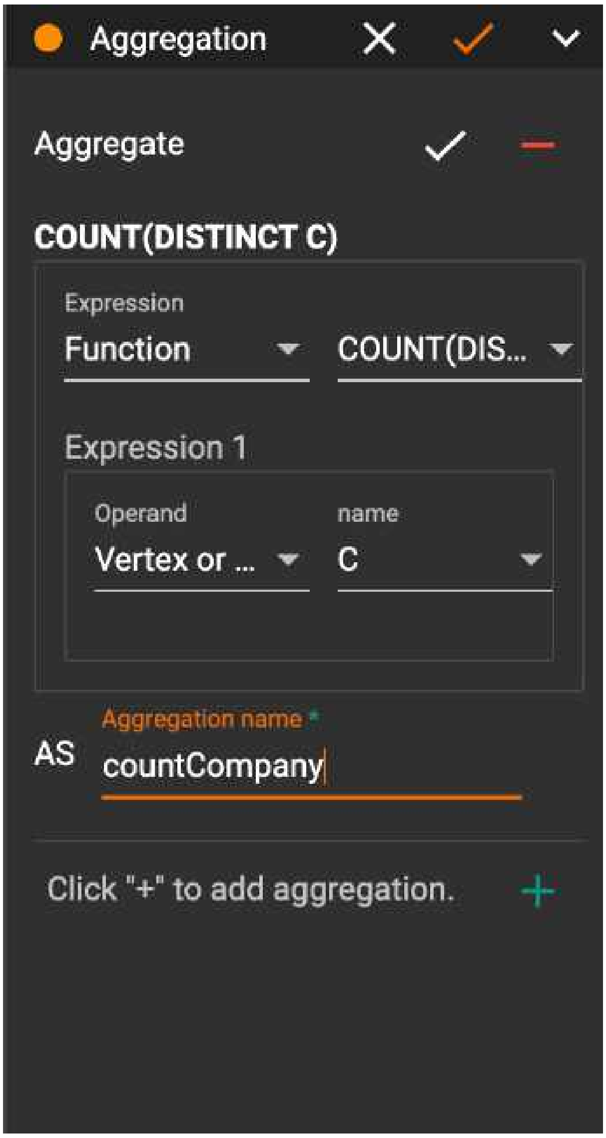
Confirm the change, and the label of company vertex pattern becomes **company as C**:



Click the city vertex pattern, then click in the toolbar:

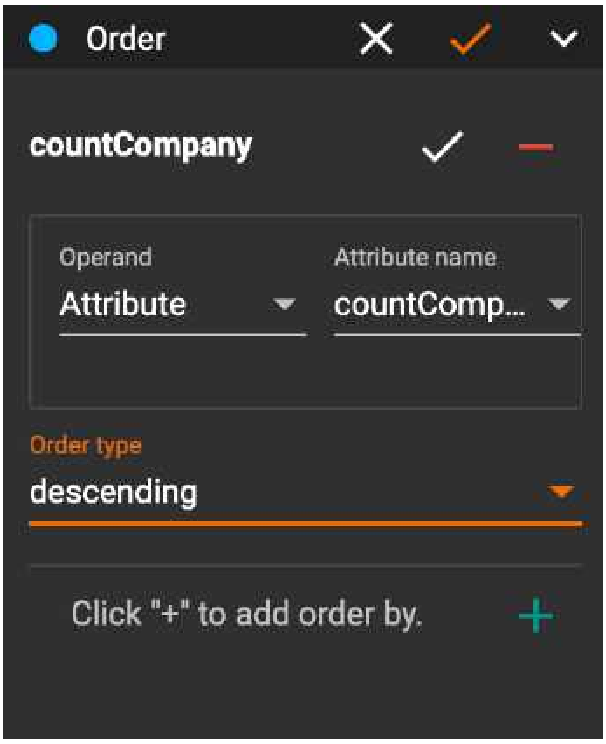


Expand the Aggregation section in the config panel, and add an aggregation:



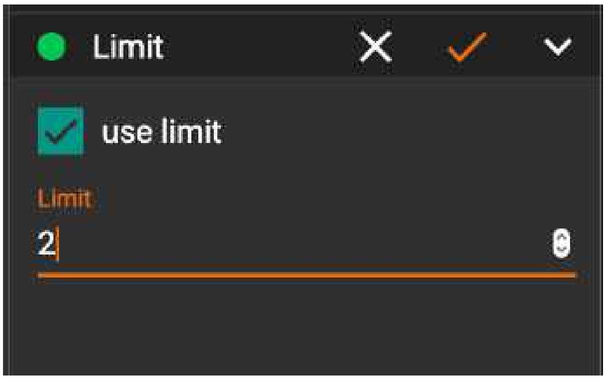
Confirm the change:

Expand the Order section in the config panel, and add a descending order of aggregation result **countCompany**:

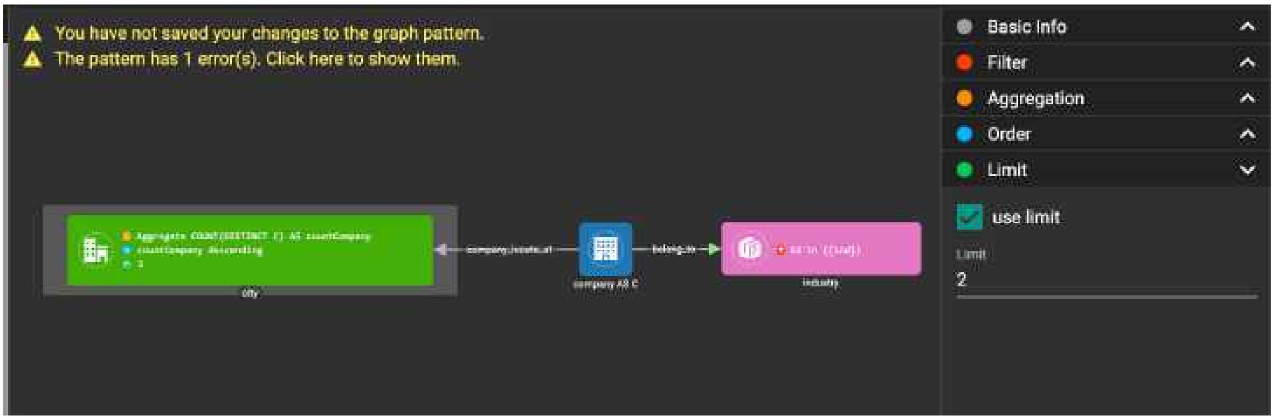


Confirm the change:

Expand the Limit section in the config panel, check the use limit checkbox, and change limit to 2:

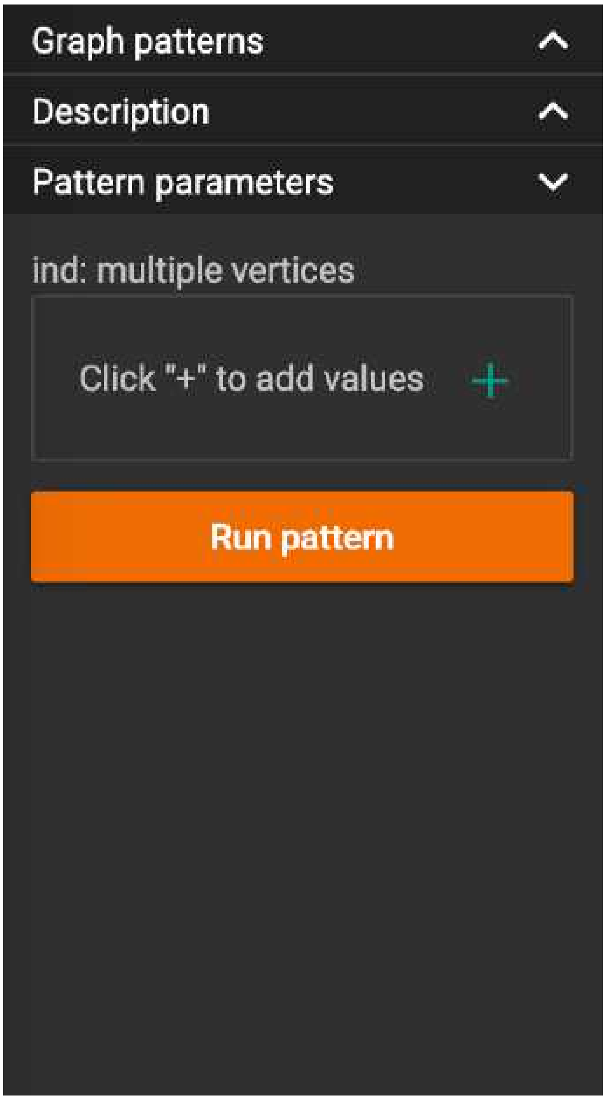


Confirm the change:



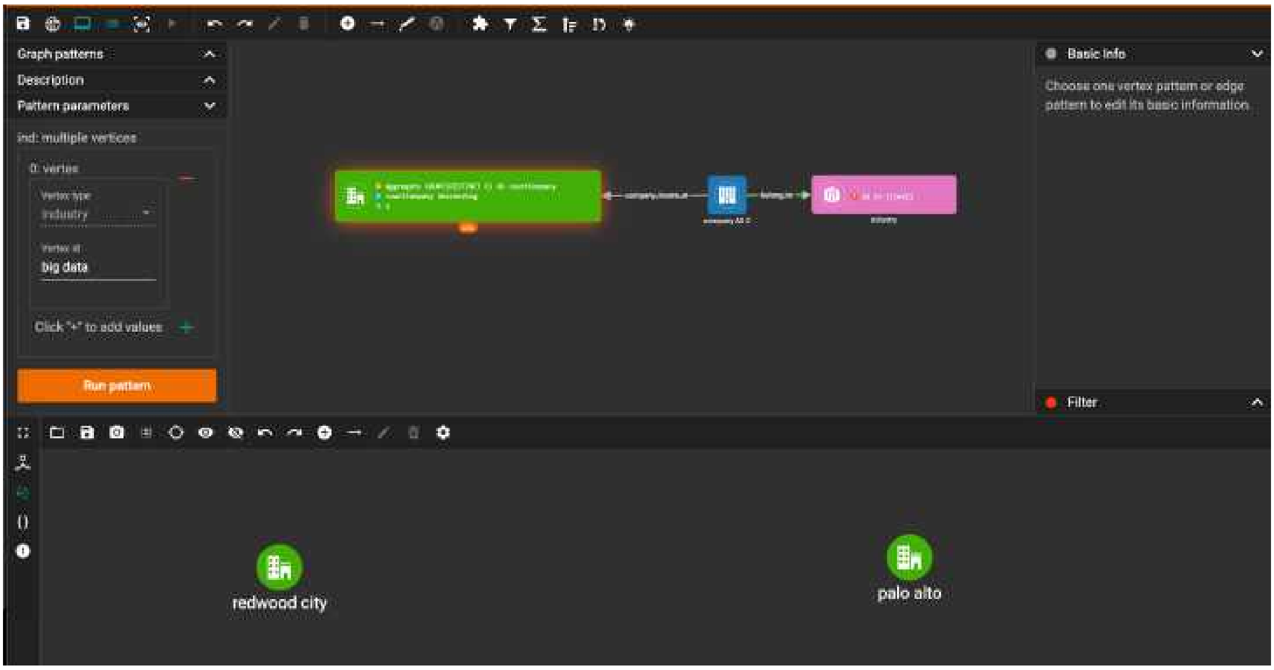
Click , then click the city vertex pattern, then click again to exit output toggle mode, click to save the pattern:

Click . Because the pattern has one input parameter **ind**, the Pattern parameters section in the pattern list panel expands:



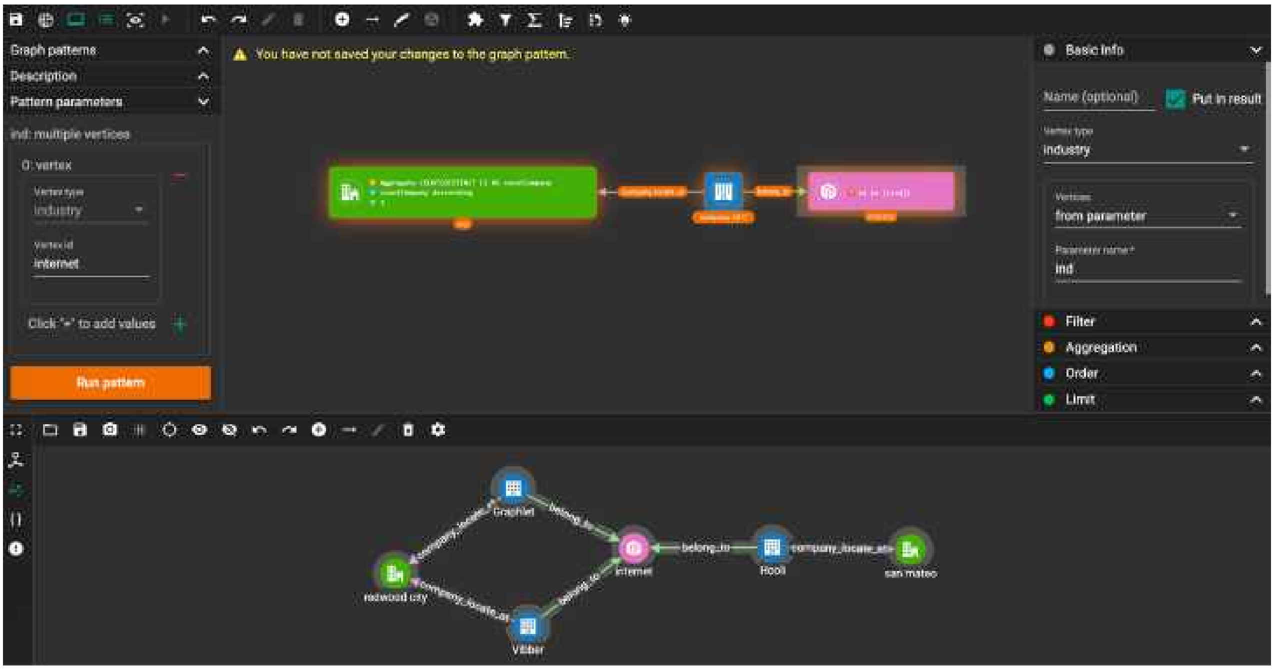
Click to add one industry vertex, and input "big data" for vertex id, then click 

:



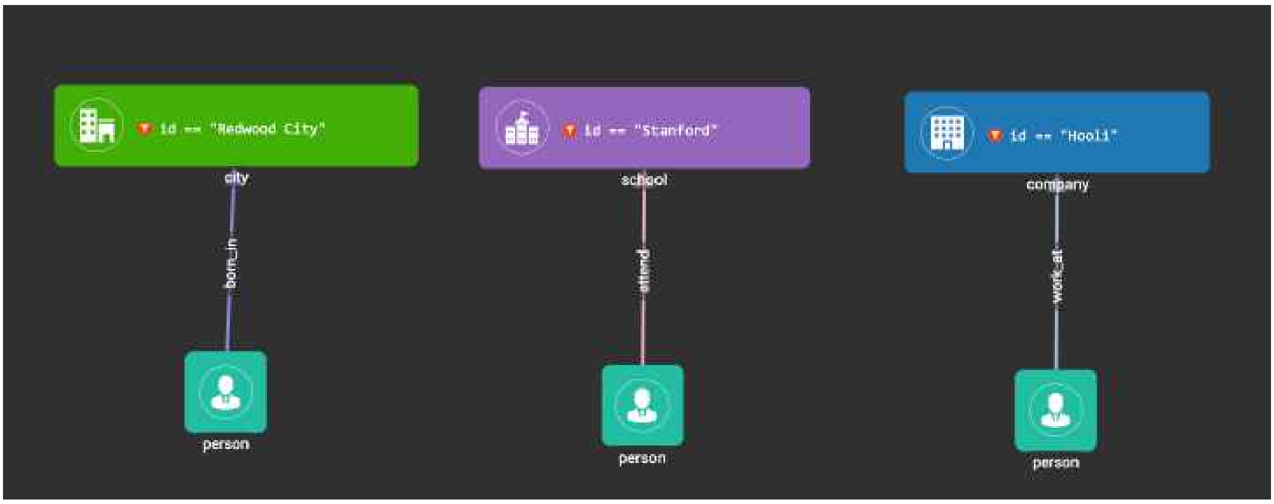
Change the industry vertex id to be "internet", and run the pattern again:

Again, you can choose to output everything in the matched subgraphs:



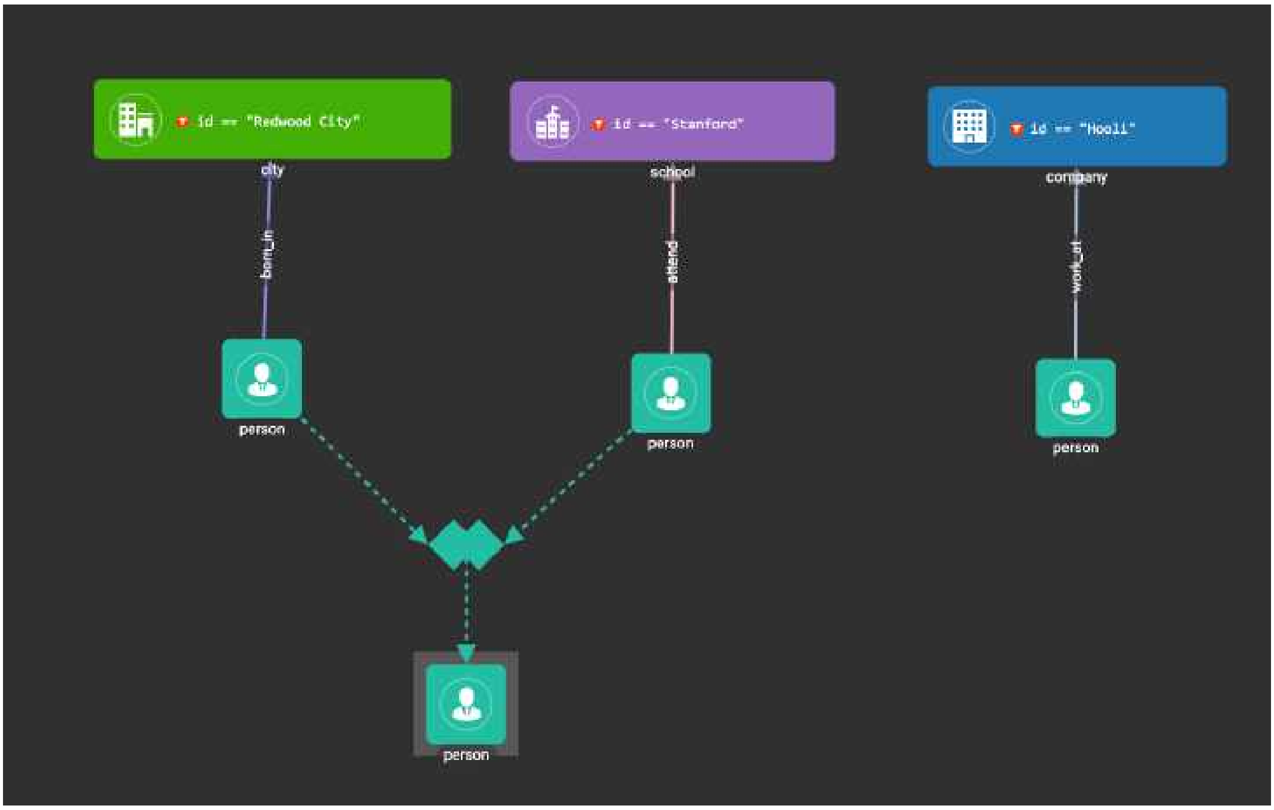
Example 4. Find the people who either were born in Redwood City or attended Stanford university, but didn't work for Hooli.

Expand Graph patterns section in the pattern list panel, then create a new visual pattern named SearchPeople. Create the following visual pattern following similar steps as above:



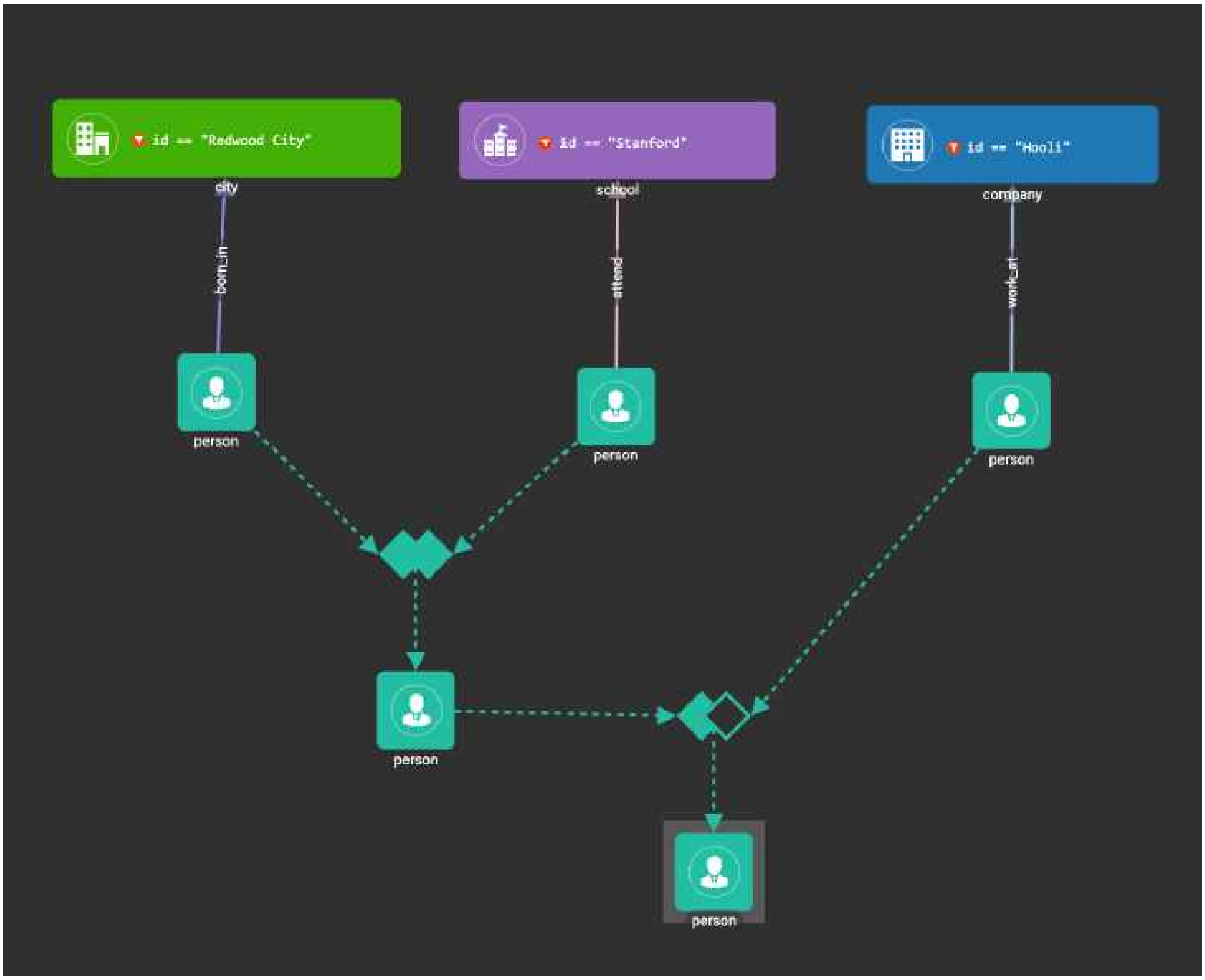
Click  in the toolbar, and choose **Union**:

Click the left two person vertex patterns respectively to add the union widget:

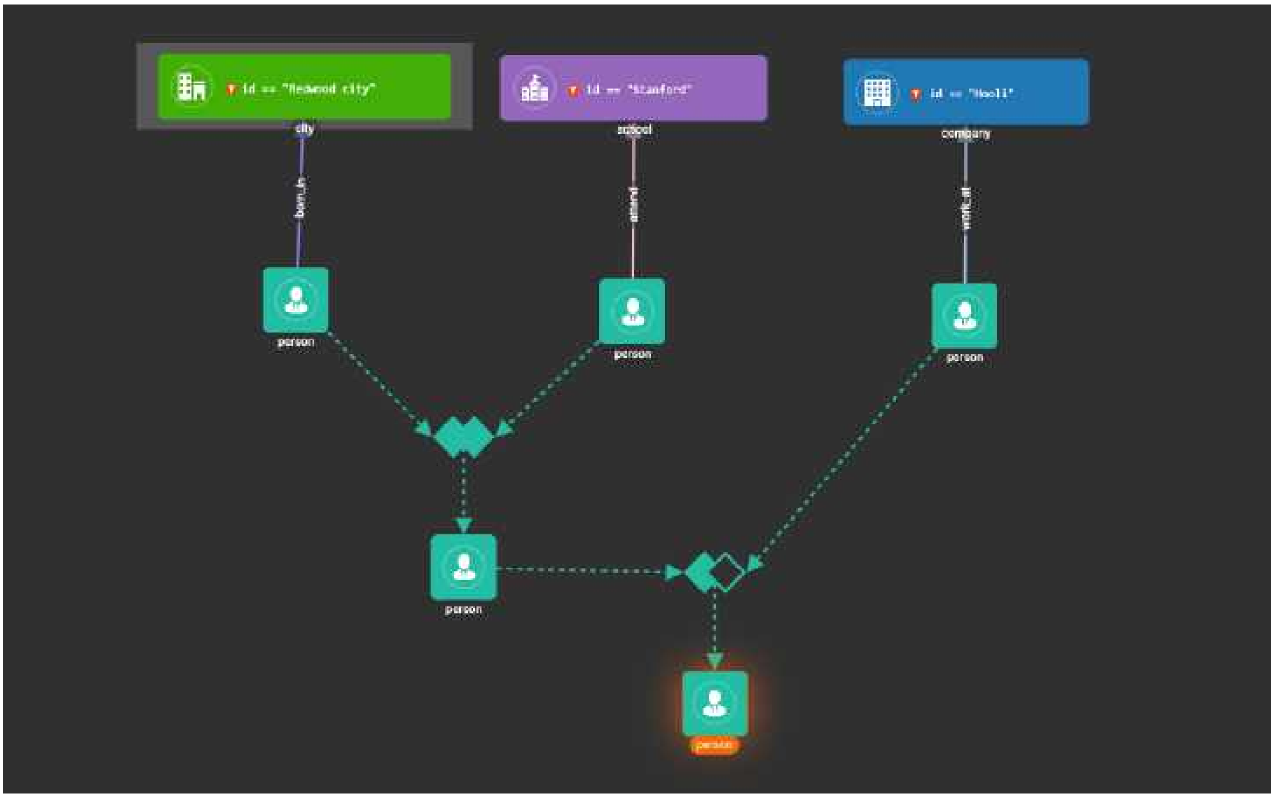


Clickin the toolbar, and choose **Subtract**:

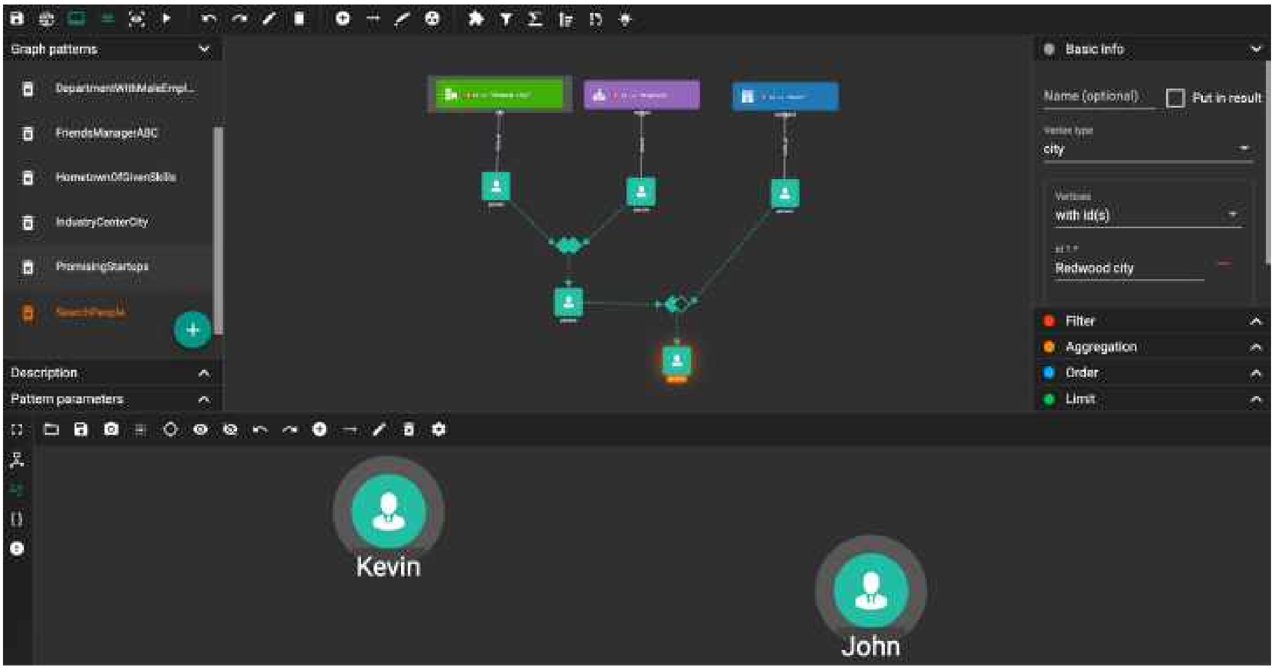
Click the output person vertex pattern of the union widget, then click the right person vertex pattern, a subtract widget is added into the visual pattern:



Put the output person vertex pattern of the subtract widget into result:

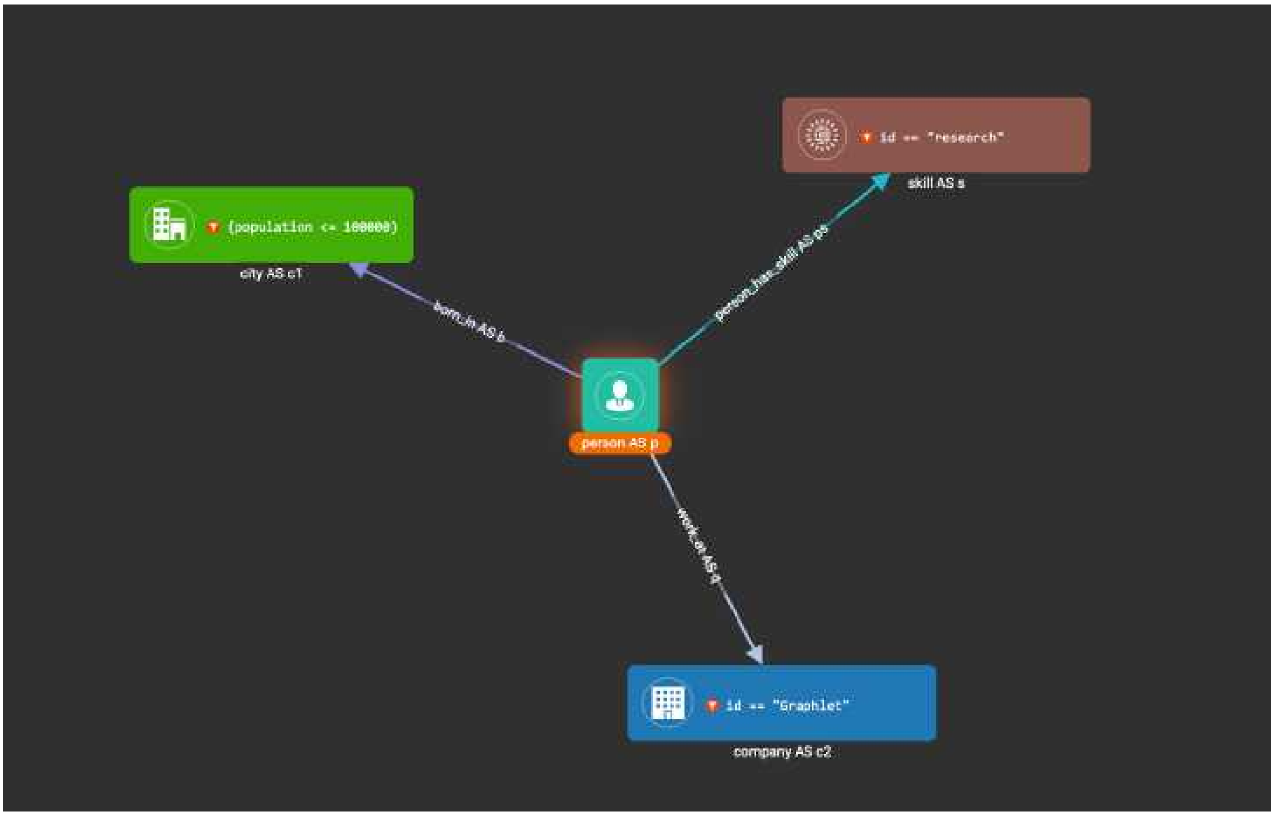


Save and run the pattern:



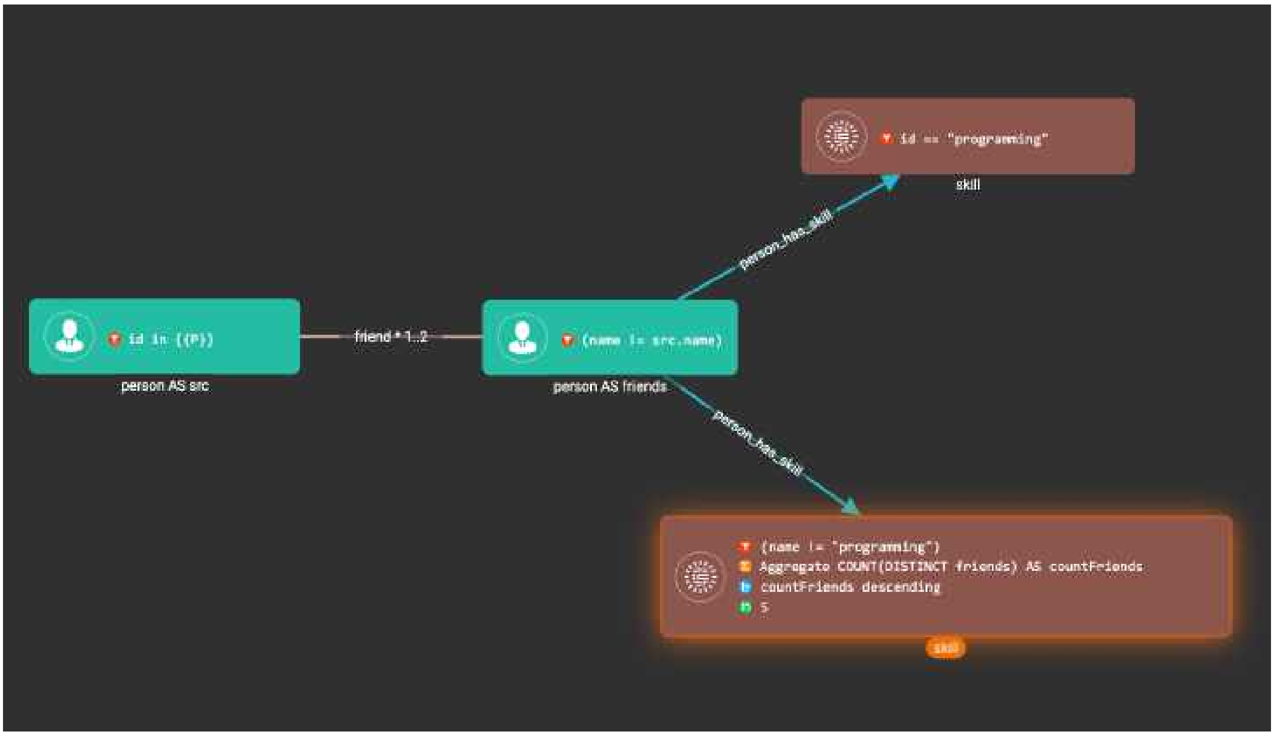
Example 5. Find all the people that were born in a city with population no more than 100,000, worked for Graphlet, and have research skill.

A snowflake-like visual pattern is commonly used in this type of searching problems:



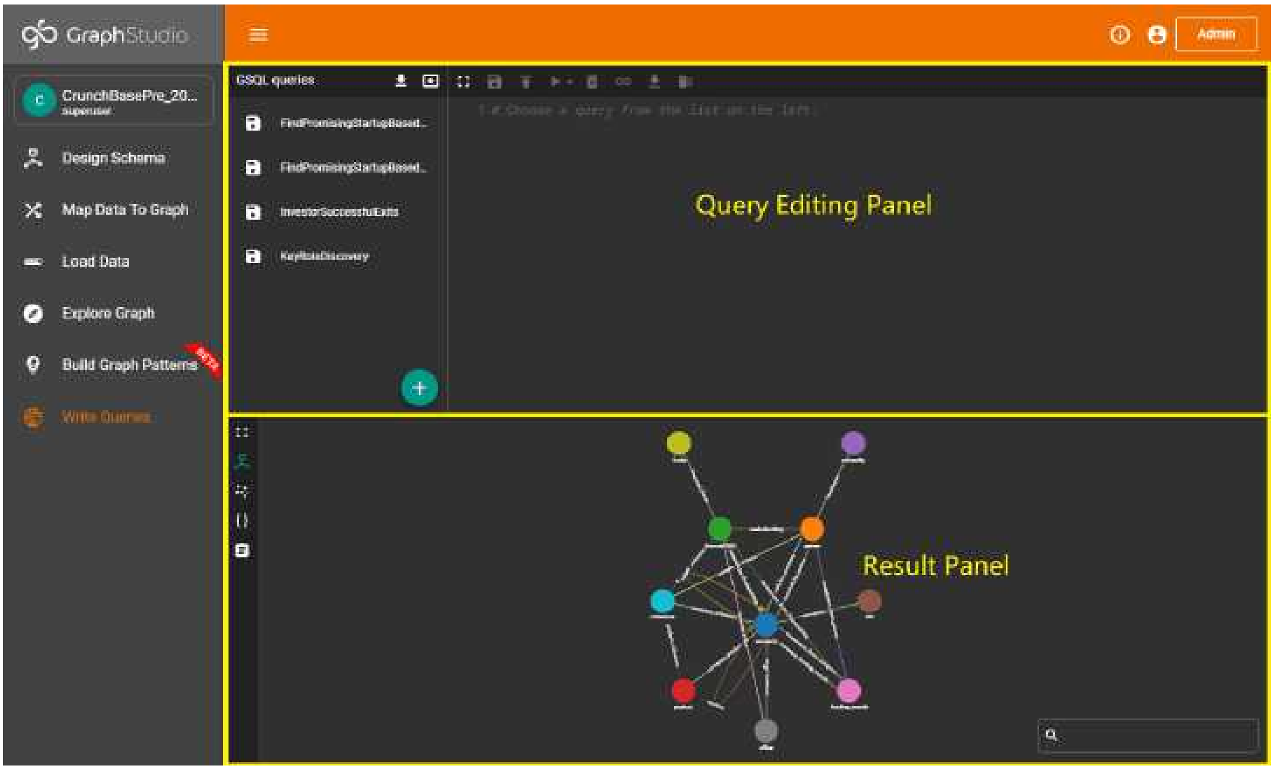
Example 6. Giving a person P, among all his direct friends and friends' friends who have programming skill, what are the top 5 other skills that most of these people have?

First, we get the friends and friends' friends of P. We use the programming skill to filter these friends, then we find the other skills that these people have. Then we count the number of people having these skills, and choose the top 5 skills as the result.


id == "programming"
skill
person_has_skill
id in {(P)}
friend * 1..2
(name != src.name)
person AS src
person AS friends
person_has_skill
(name != "programming")
Aggregate COUNT(DISTINCT friends) AS countFriends
countFriends descending
5
skill

Write Queries

On the Write Queries page, you can write and run custom queries with GSQL.



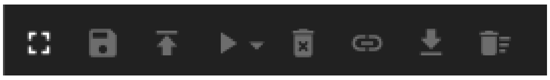
The Write Query page is horizontally divided into two parts:

- Query Editing Panel
- Result, Log and Visualization Panel

Query Editing Panel

The Query Editing panel is divided into two sub-panels: the left sub-panel is used to select a query to edit, and the right, larger sub-panel displays the selected query for editing. Here you can edit, save, delete query and/or its draft, install and run the query. The query editor features syntax highlighting customized for the GSQL language. Also, the query editor performs real-time semantic checking.

Above the query editing pane is a toolbar, with the following buttons, from left to right:

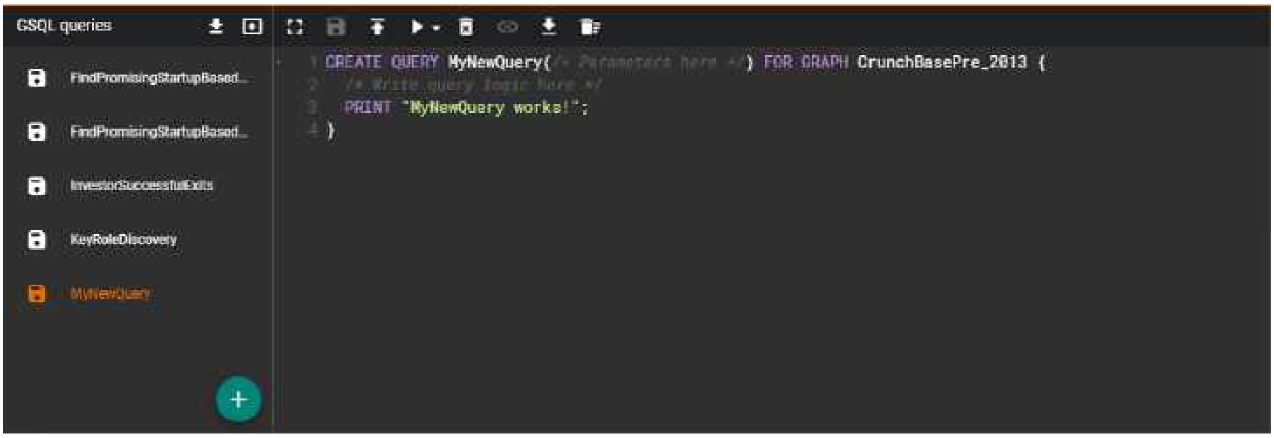


- **Expand/Collapse** : Expand or collapse the Query Editing panel to or from full page mode. The icon changes depending on whether the panel is currently expanded or collapsed.
- **Save** : Save the current query draft.
- **Install** : Install the query into the database.
- **Run** : Run the query. If the query is not installed, it will run the query in interpreted mode. Note there is performance penalty to run queries in interpreted mode.
    - **Run Configuration:** Set the query runtime configurations.
- **Delete** : Delete the selected query and its draft.
- **Show query endpoint:** Show the RESTFul endpoint to execute the query. Only installed queries can see their RESTFul endpoints.
- **Download:** Download the query as a gsql file.
- **Discard query draft :** Delete the selected query draft.

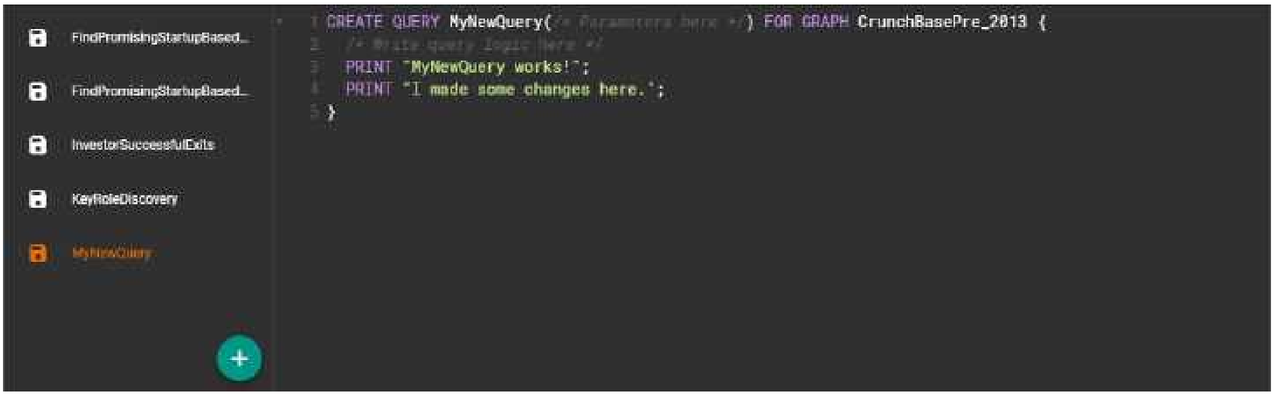
Add Or Edit Query

To create a new query, simply click on the "New GSQL Query" button  at the bottom-right corner of the left sub-panel, and type in the name of the new query in the popup window:

A query draft will be created with a template:



To edit an existing query, click on the query name in the list in the left sub panel:

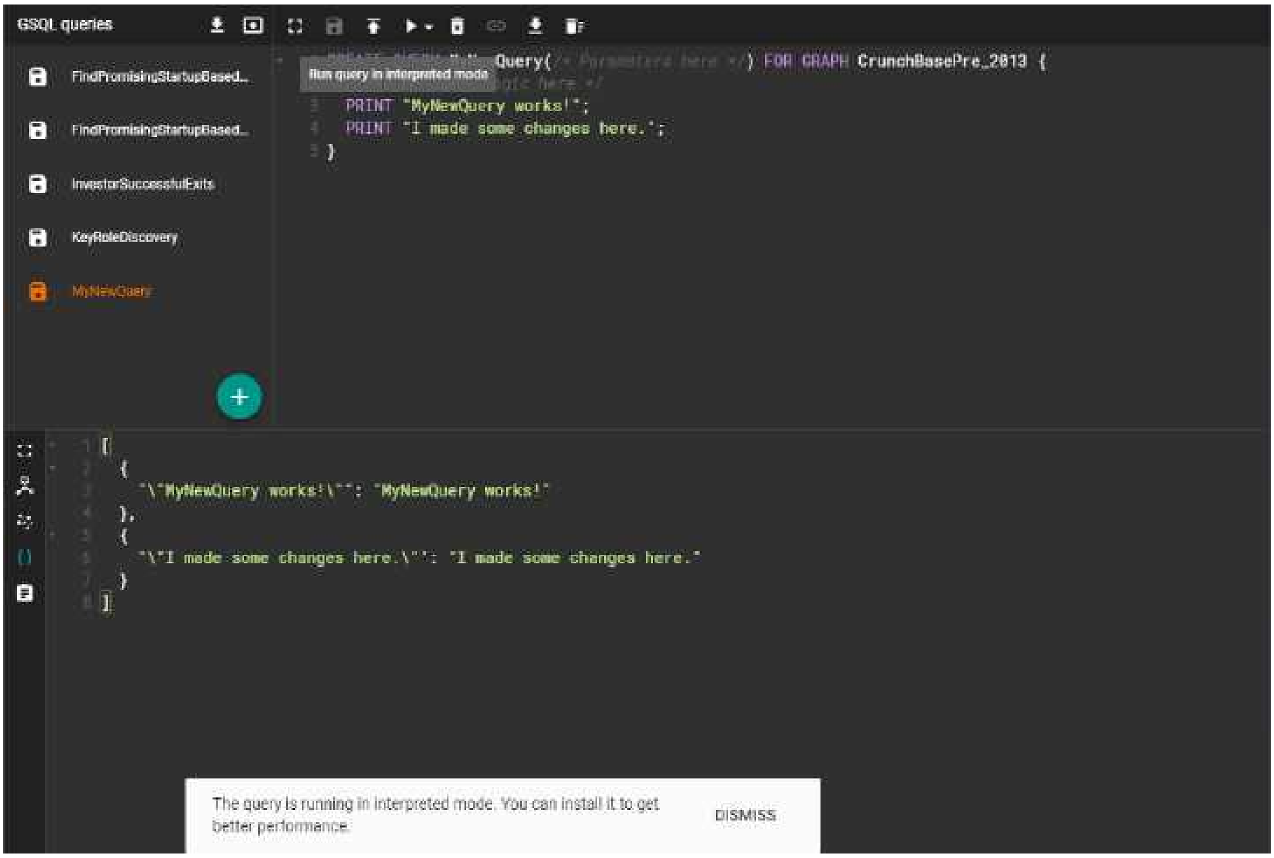


Save Query Draft

Once you made some changes to the query code and want to save it as a query draft, click on the "save" button in the toolbar.

Run Query in Interpreted Mode

If you have saved the query and there is no errors in the query, you can run the query in interpreted mode. Click the run query button . Together with the query execution result, a warning message will appear to notify user of the performance concerns running the query in interpreted mode. To speed up the query performance after you are satisfied with the GSQL query developed in interpreted mode, you can follow the instructions in the next section "Install Query".



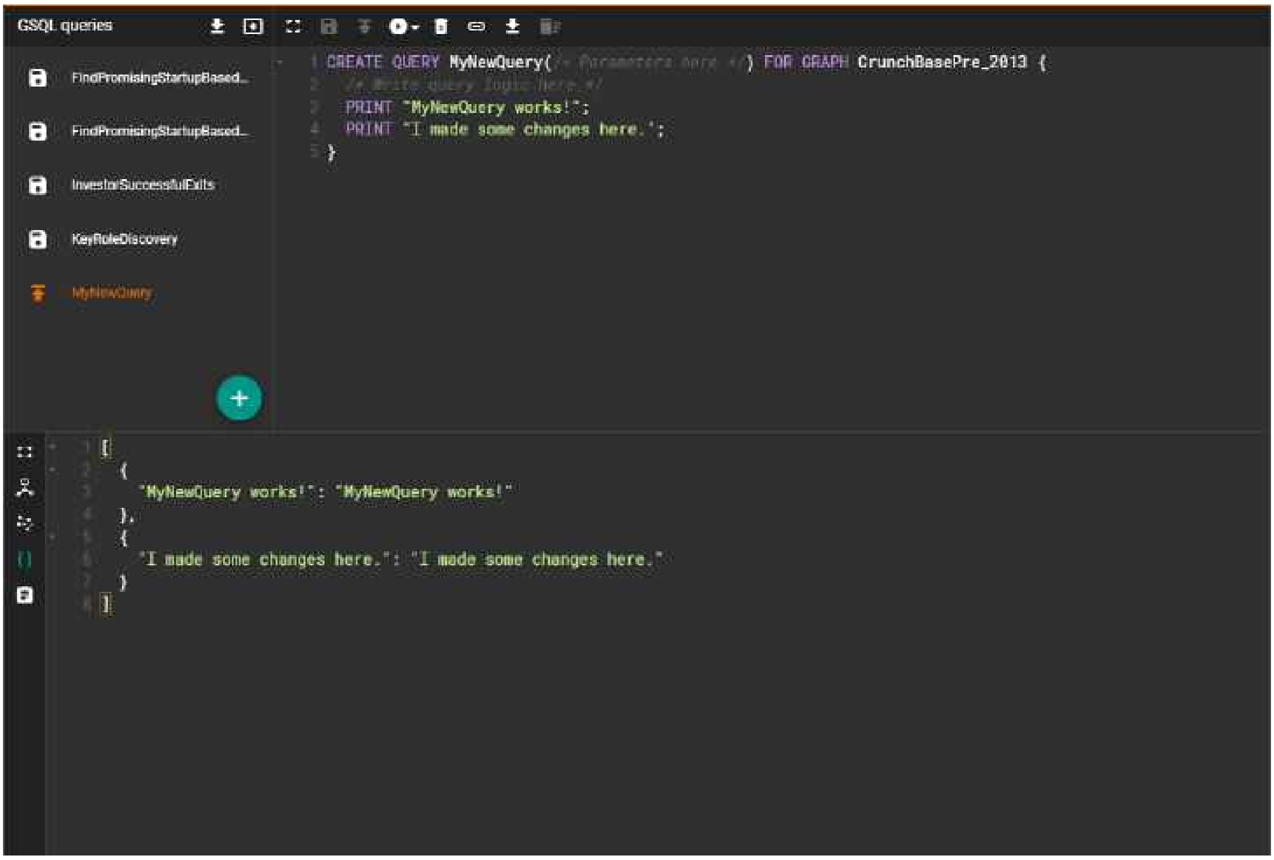
Install Query

If you saved a query, the "install query" button will be enabled. Click it to install the query. The installation process may take about 1 minute:

After the installation, the run query button changes from to .

Run Query

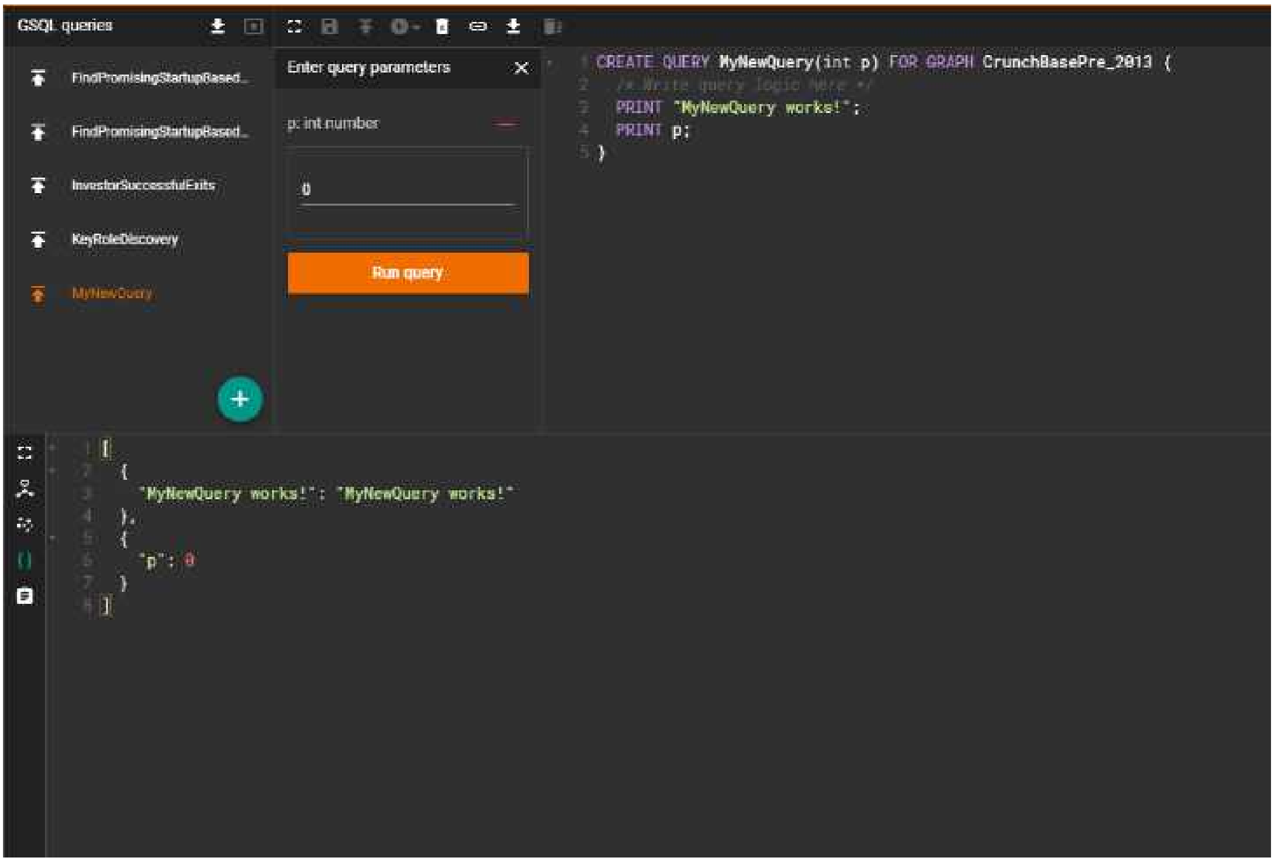
To run the query, click on the "run" button in the toolbar. If the query has no parameters, it will run directly and the result will be shown in the Result panel.



If the query requires parameters, the Enter Query Parameters panel will appear. Enter your parameter values and then click the "Run Query" button

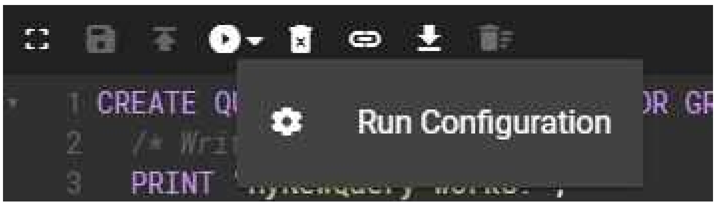
at the bottom of the panel. If there are several parameters, you might need to scroll the panel to the bottom to find the Run Query button.

The query will be executed, and the results will be shown in the Result Panel.

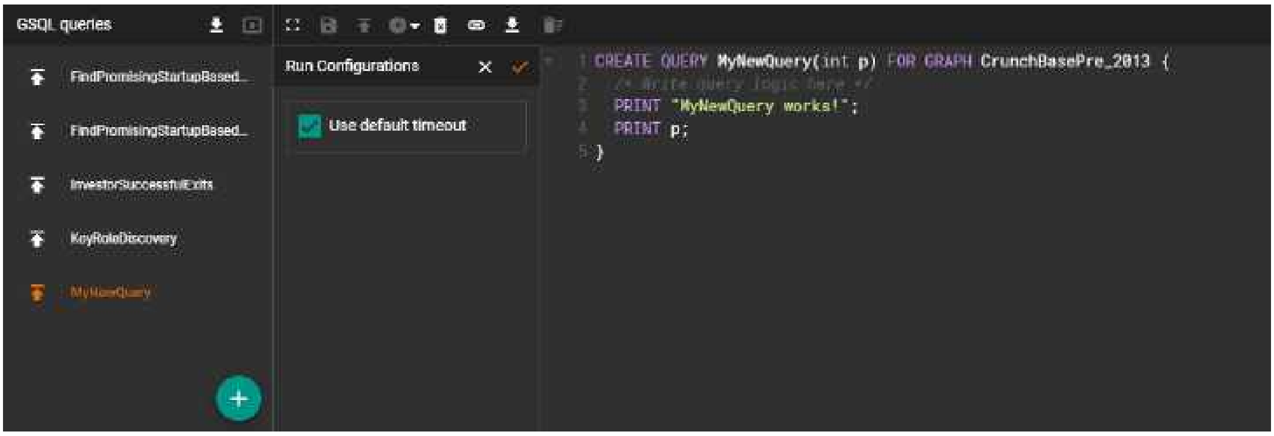


Run Configuration

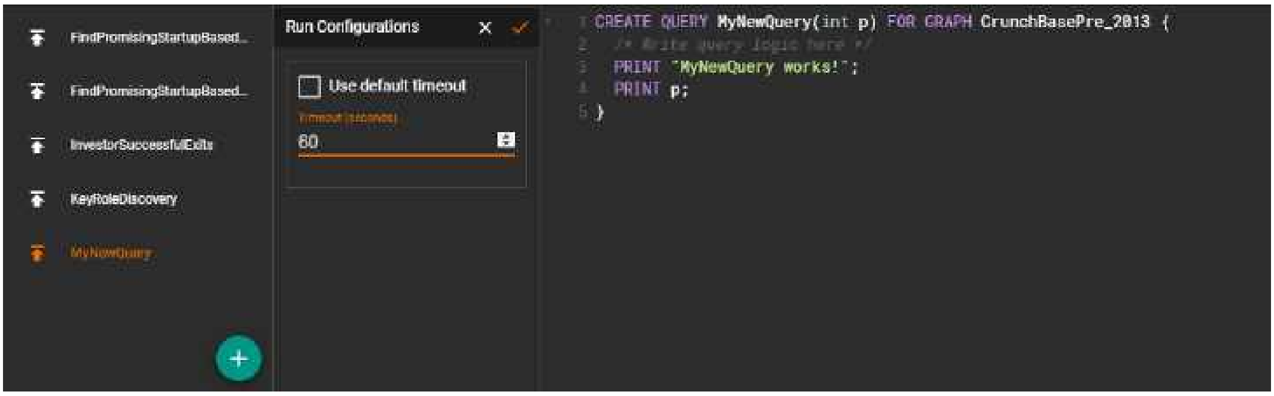
You can set runtime configurations of the GSQL query by clicking the small button to the right of run query button, then a drop-down menu will appear.



Click the "Run configuration" item, and the Run Configuration panel will be opened.



You can set GSQL query timeout here. By default it uses the timeout of TigerGraph configuration (specified by gadmin commands). You can change it by unchecking the "Use default timeout" label, and then set a new timeout:



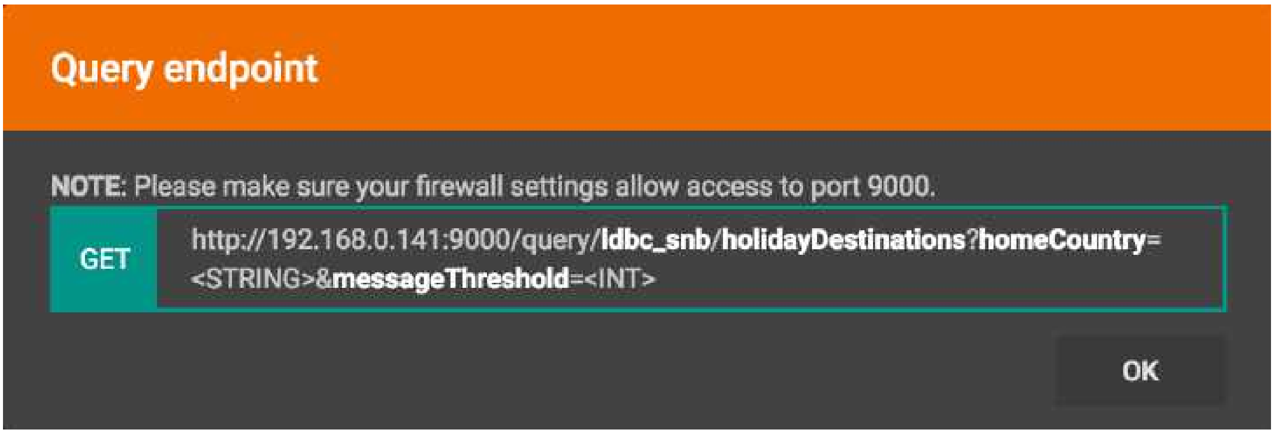
Click the submit button to apply your changes.

Delete Query

Choose the query you want to delete and click on the "delete" button. The query will be deleted permanently.

Show Query Endpoint

After finishing writing the GSQL queries and installing the queries, you can access the queries via REST endpoints. By clicking the "show query endpoint" button, you can see the format of the endpoint to access this query, so that you can integrate the query with your applications.



Download

You can download your query by click , or download all your queries as a tarball by click 

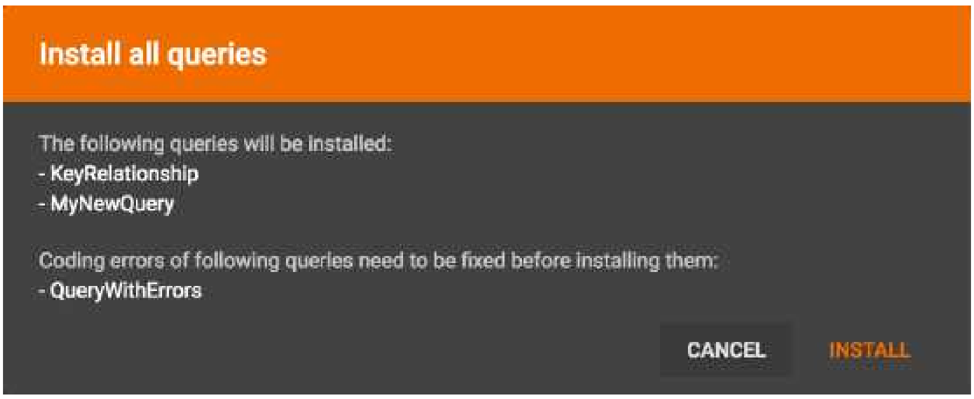
Delete query draft

You can delete your query draft by clicking 

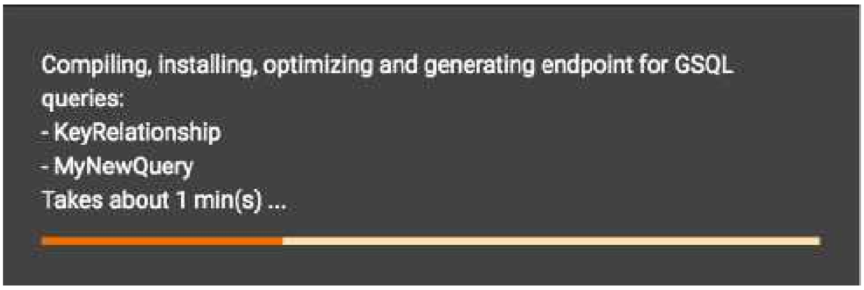
.

Install All Queries

If you want to install all queries that you haven't installed yet, you can click "Install all queries" button  in GSQL Queries list. After some verification time, a pop up window listing all queries to be installed will show:



Click INSTALL button, then the listed queries will be installed:



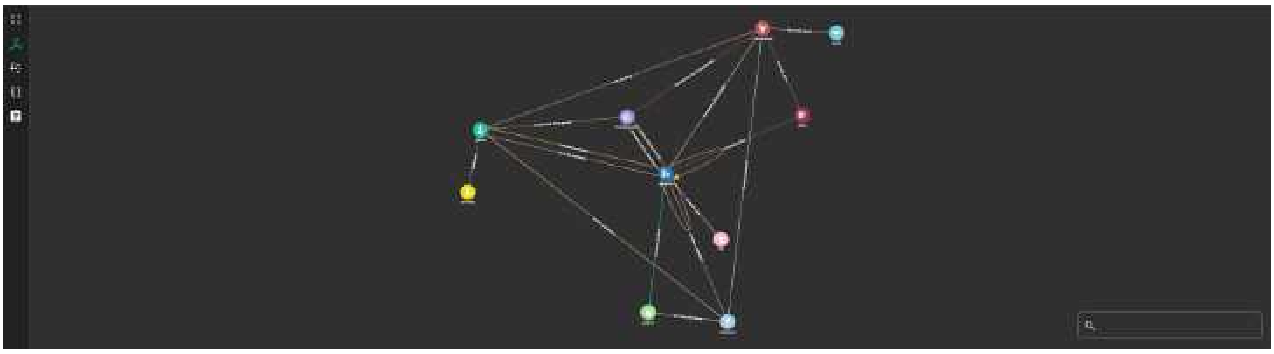
Result Panel

The Result panel shows the result of the last run query. Each query generates up to three types of result: visualized graph, JSON text, or log messages. On the left is a toolbar with buttons for changing the the panel size or for switching to a different type of result. The buttons, from top to bottom, are the following:

menu option	functionality
	Expand/Collapse: Expand or collapse the Result panel.
	View schema: Show the graph schema.
	Visualize graph result: Show the visual result of the last run query.
	View JSON result: Show the raw text result in JSON format of the last run query.
	View logs: Show the log for the last run query.

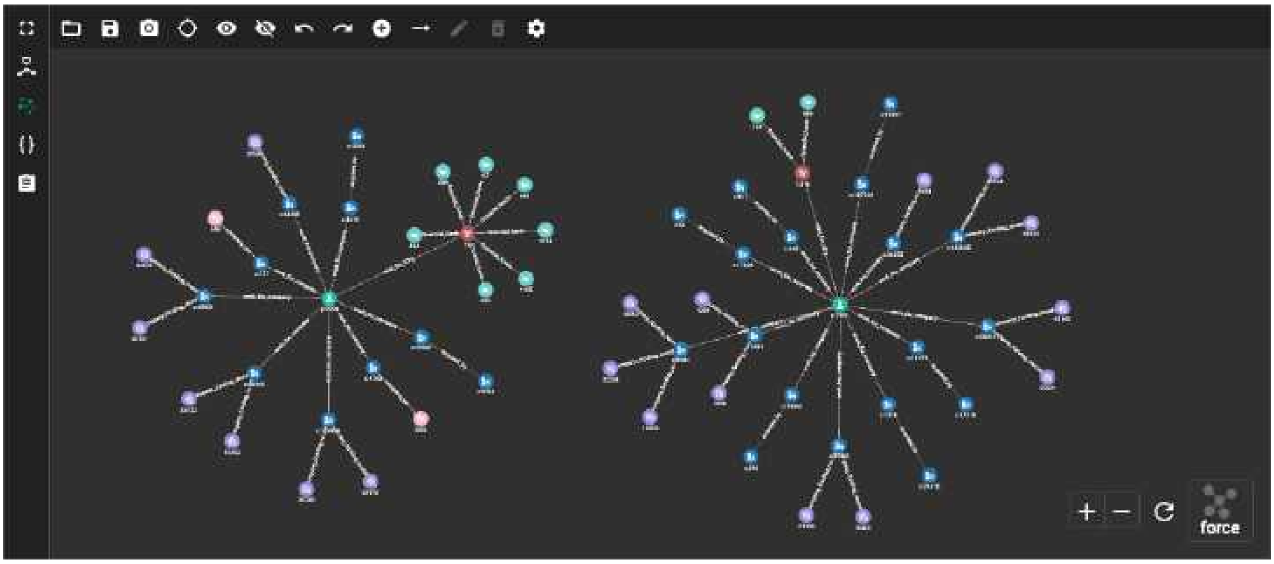
View schema:

Viewing graph schema makes it more convenient for developers to refer to the schema topology logic and easier to write correct GSQL queries.



Visualize graph result

If the query execution result contains a graph structure, the result will be visualized in this panel as a graph. The panel is the same as the Explore Graph panel. Please refer to the documentation for the Explore Graph panel. The only difference is that each time you run a query, the previous result will be erased. In Explore Graph the results are added incrementally.



You can switch to the JSON Result panel to see the result in JSON format.

View JSON result

If there is no graph structure in the result, the result will be displayed in this panel as a JSON object.

You can learn about the JSON format in the GSQL Language documentation , and integrate it with your applications. In this fashion, the TigerGraph system can serve as a backend or embedded graph data service.

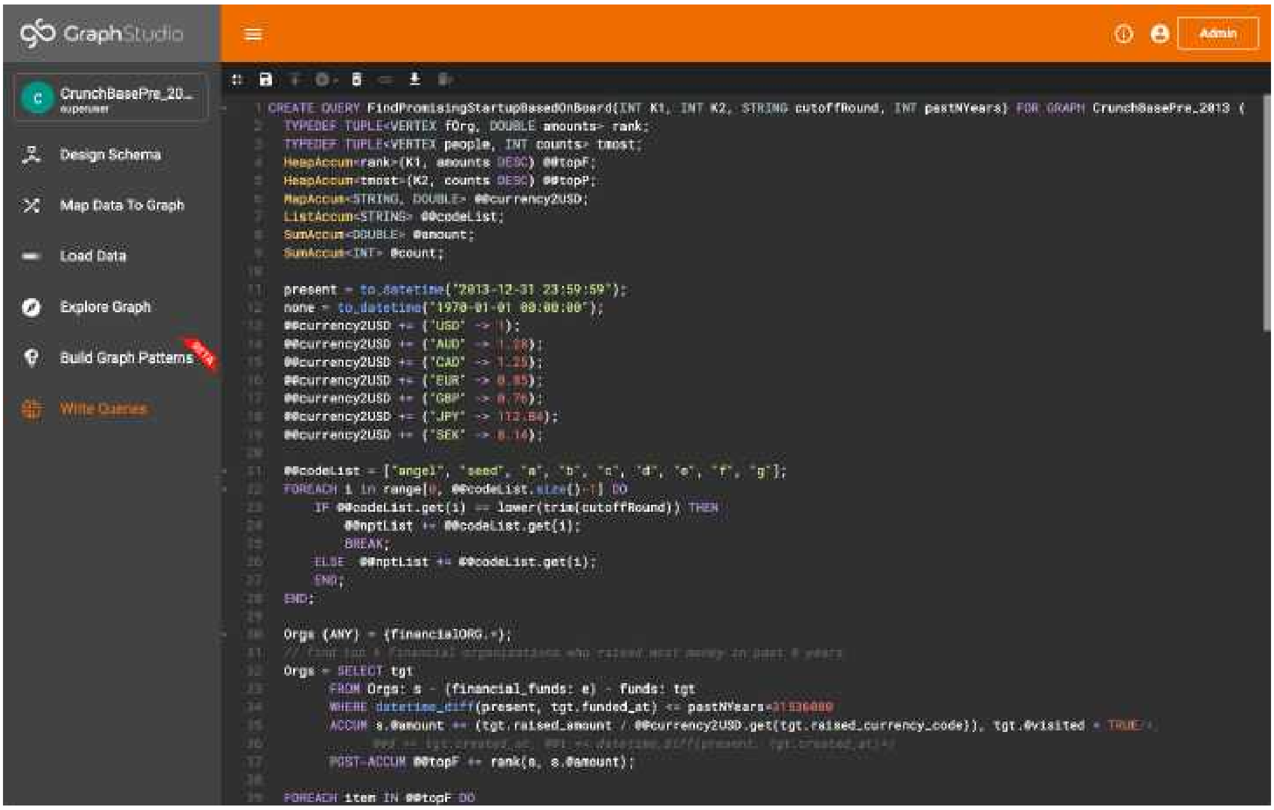
View logs

If a query ran successfully, the Query Log message will be "query ran successfully" or something similar. If there was anything wrong when executing your query, such as invalid parameters or runtime errors, an error message will be shown in the Query Log panel:

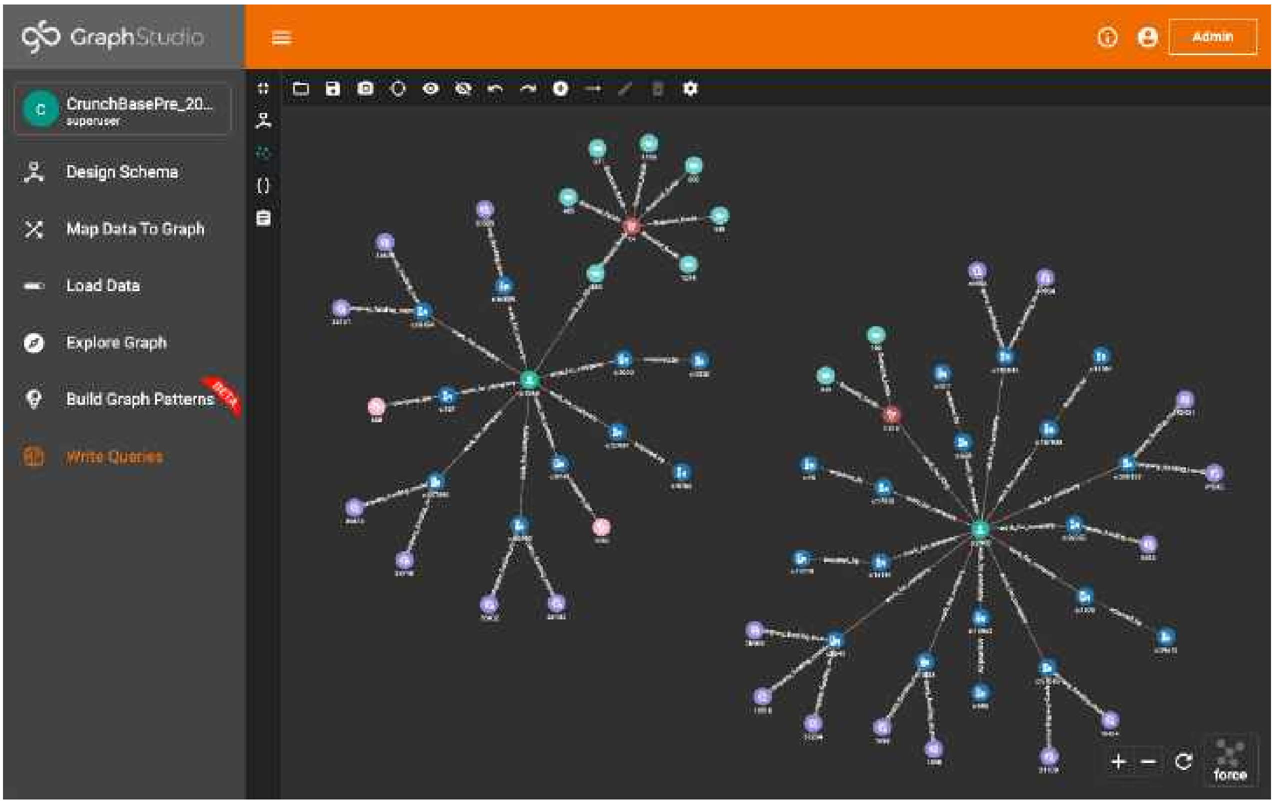
Expand Panels

If you just want to focus on developing your query, or want to have more space to view your result, click the Expand button in either the Query Editing panel or the Result panel.

If you expand the Query Editing panel, it looks like this:



If you expand the Result panel, it looks like this:



When the panel is expanded, the Expand button becomes the Collapse button . Clicking it will return the display to the split panel view.

Loading...

Loading...

Keyboard Shortcuts

The page lists the keyboard shortcuts for common actions in GraphStudio.

Windows

Application

Action	Shortcut
Switch focus	Tab

Graph Canvas

In order to use these shortcuts, make sure you first select the graph canvas with the tab key. When the graph canvas is selected, the edges of the canvas will be highlighted with a blue box:

View when the graph canvas is selected

Action	Shortcut
Pick a vertex/edge	Ctrl + Alt + P
Switch hover over vertex/edge (must have picked a vertex/edge first)	Ctrl + Alt + W/A/S/D
Edit/show a vertex	Ctrl + Alt + C

Mac

Application

Action	Shortcut
Switch focus	Tab

Graph Canvas

In order to use these shortcuts, make sure you first select the graph canvas with the tab key. When the graph canvas is selected, the edges of the canvas will be highlighted with a blue box:

View when the graph canvas is selected

Action	Shortcut
Pick a vertex/edge	Ctrl + Option + P
Switch hover over vertex/edge (must have picked a vertex/edge first)	Ctrl + Option + W/A/S/D
Edit/show a vertex	Ctrl + Option + C

Loading...

Loading...

Admin Portal UI Guide

Overview

The TigerGraph Admin Portal is a browser-based DevOps tool that provides users an overview of a running TigerGraph system, from an application and infrastructure point of view. It also allows the users to configure the TigerGraph system through a user-friendly interface. This guide serves as an introduction and quick-start manual for Admin Portal.

Admin Portal is certified on the following browsers:

Browser	Chrome	Safari	Firefox	Opera	Edge	Inte Exp
Supported version	54.0+	11.1+	59.0+	52.0+	80.0+	10+

Not all features are guaranteed to work on other browsers.

Please make sure to enable JavaScript and cookies in your browser settings.

Log On

The Admin Portal and GraphStudio share the same port (14240). If you are logged in to one of the servers for your TigerGraph system, you can use `localhost` for your server IP address. The Admin Portal is on the admin page:

```
http://<tigergraph_server_ip_address>:14240/admin/
```

If user authentication has been enabled, users need to log in to access the Admin Portal.

If you are already at GraphStudio, click the Admin button 

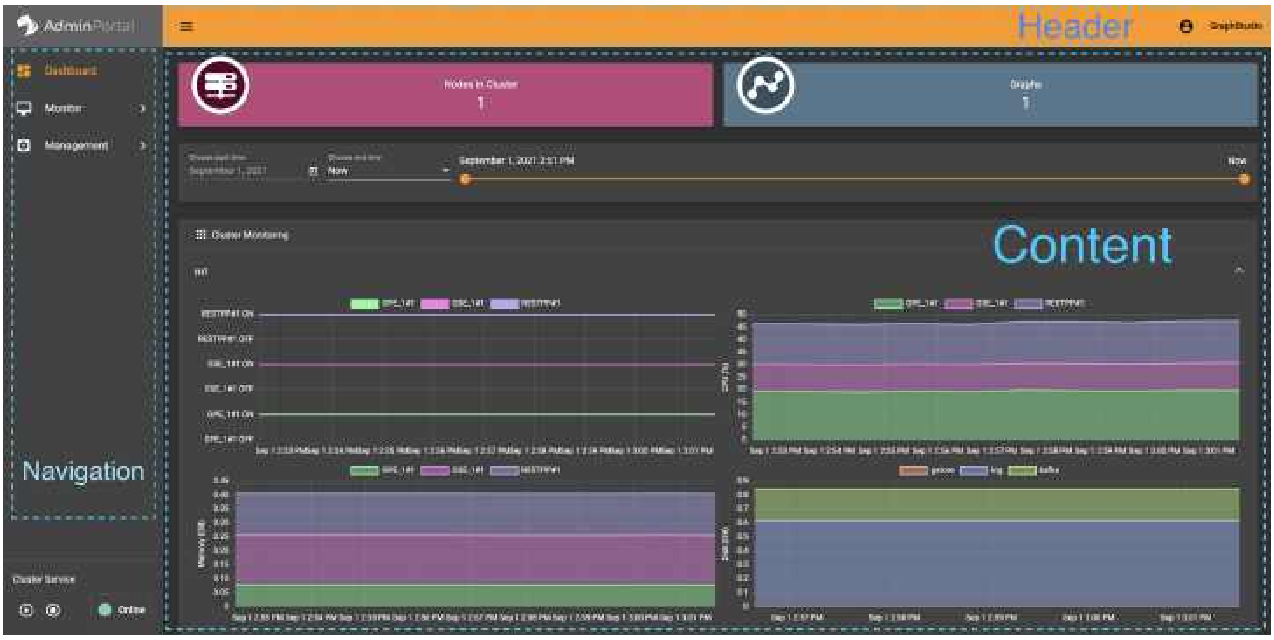
at the right end of the top menu bar to navigate to Admin Portal.

Page Layout

The Admin Portal v3.1 has three pages: Dashboard, Monitor, and Management. All the pages share the same Header, and Navigation Menu.

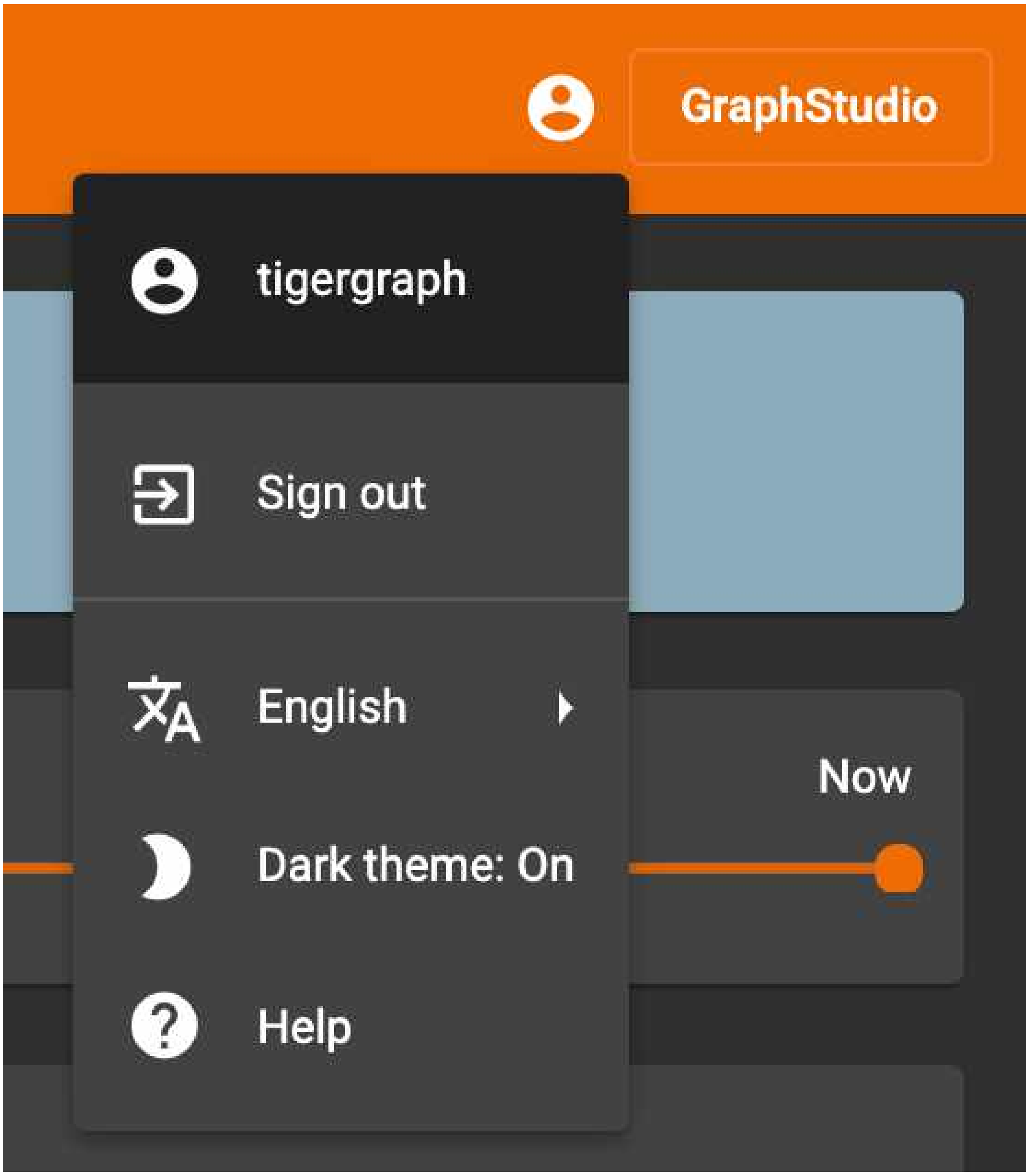
> The layout of the Admin Portal is responsive to screen size. The layout will automatically adjust for devices with small screens like phones and tablets.

The full-screen version of the Admin Portal is shown below, with the Dashboard page selected.

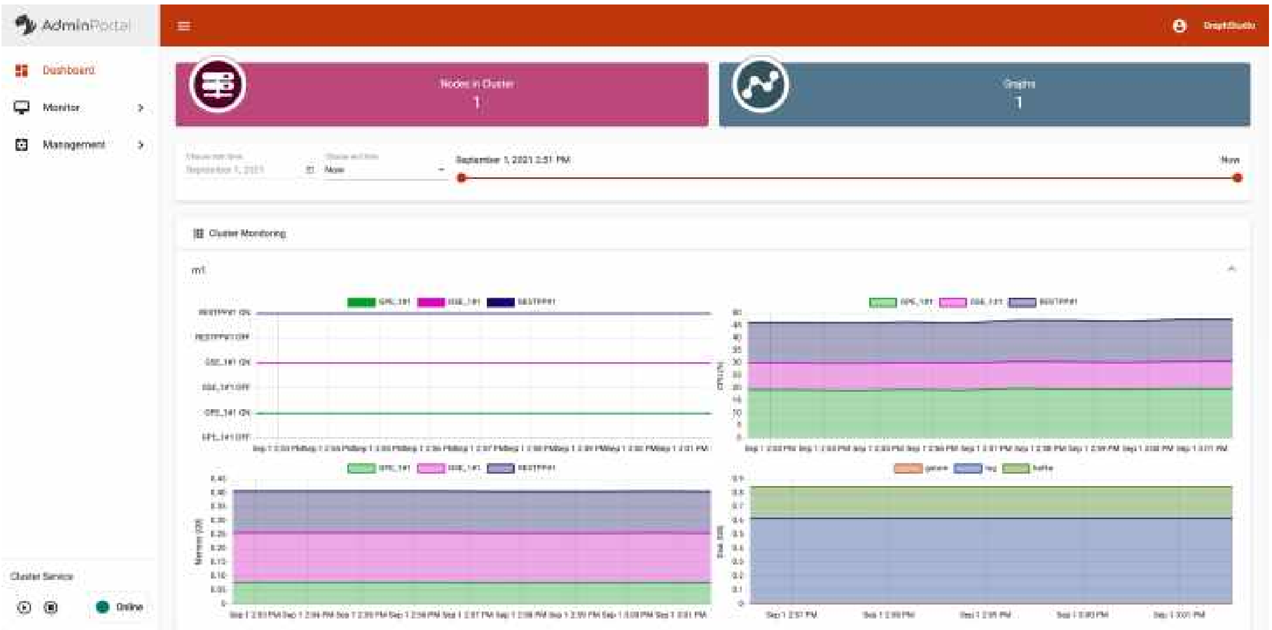


Page Header

The **Account** icon  will open the user menu:



You can switch between a dark theme and a light theme. The light theme is shown below:



To sign out of the Admin Portal, click on the **Sign out** button in the Account menu.

Clicking on the **Help** button will take you to the documentation page containing this guide.

You can navigate to GraphStudio by clicking on .

Cluster service status

The bottom of the navigation menu shows the cluster service status.

Session Timeout

Admin Portal has a default session timeout of 1 week. If, during this time, the user has no interaction with the page, the session will expire and the user will be logged out automatically. The timeout can be configured with:

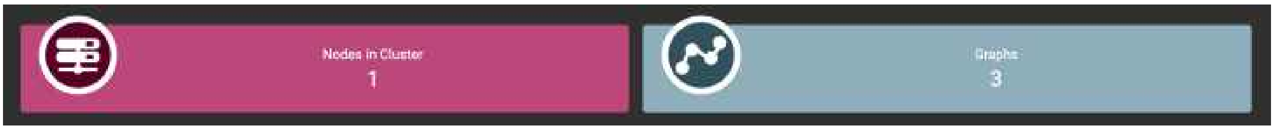
```
$ gadmin config set GUI.ClientIdleTimeSec[timeout]
```

Dashboard

Dashboard Page

The Dashboard page has three main parts: Overall statistics, the time range picker, and several charts.

Overall Cluster Statistics



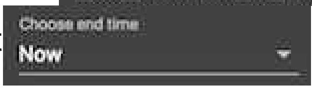
Just below the page header, there are four cards showing statistics of your system, including the number of nodes and the number of graphs. These statistics are refreshed live. (The default refresh interval is 1 minute).

Time Range Picker

The next card lets you set the time range to be used for the statistics in the charts below.

The leftmost input 

lets you select the start time of the range.

The next input 

lets you select the end time of the range. This input has two options:

1. "Now" means that the charts will be continually updated with the most recent data.
2. "Custom" lets you select a fixed date. The time range is historical, so the charts will be static.

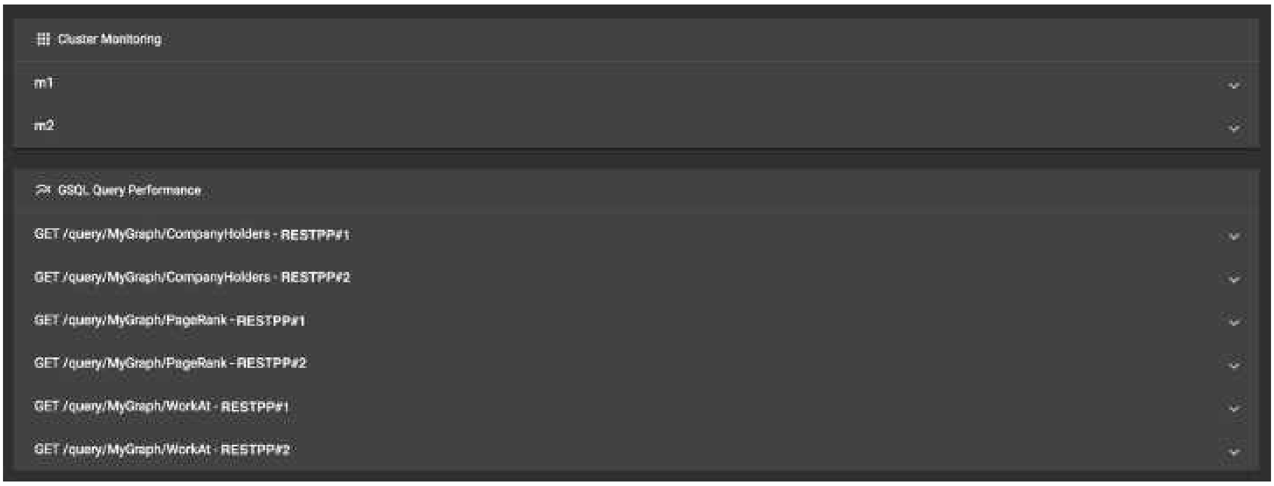
The sliding bar on the right lets you fine tune the range. Click and drag an endpoint to adjust the start or end time.

Changing any of these selections will trigger a request for statistics data and the chart will be re-rendered accordingly.

Charts

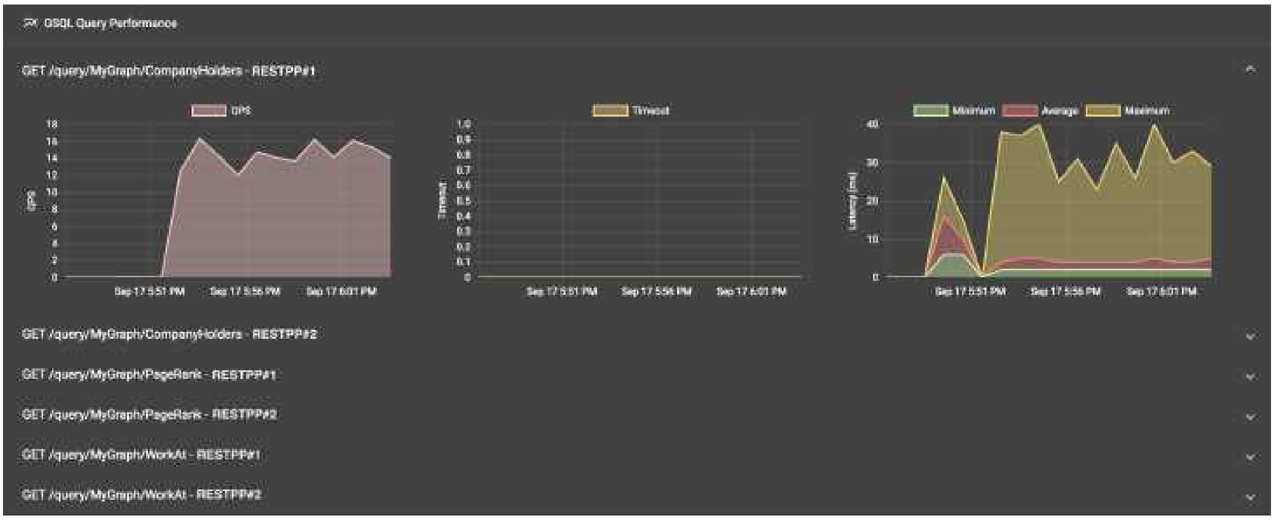
Each chart displays some statistic or states information on the vertical axis and time on the horizontal axis.

There are two chart sections. The first section is GSQL Query Performance. This lists all of the queries accessible to the current user. If you click on a query name, the display will expand to show detailed charts about that query. You can expand only one query panel at a time. The second section is Cluster Monitoring. This lists all of the machines within the TigerGraph cluster. Similar to the first section, you can only expand one panel at a time.



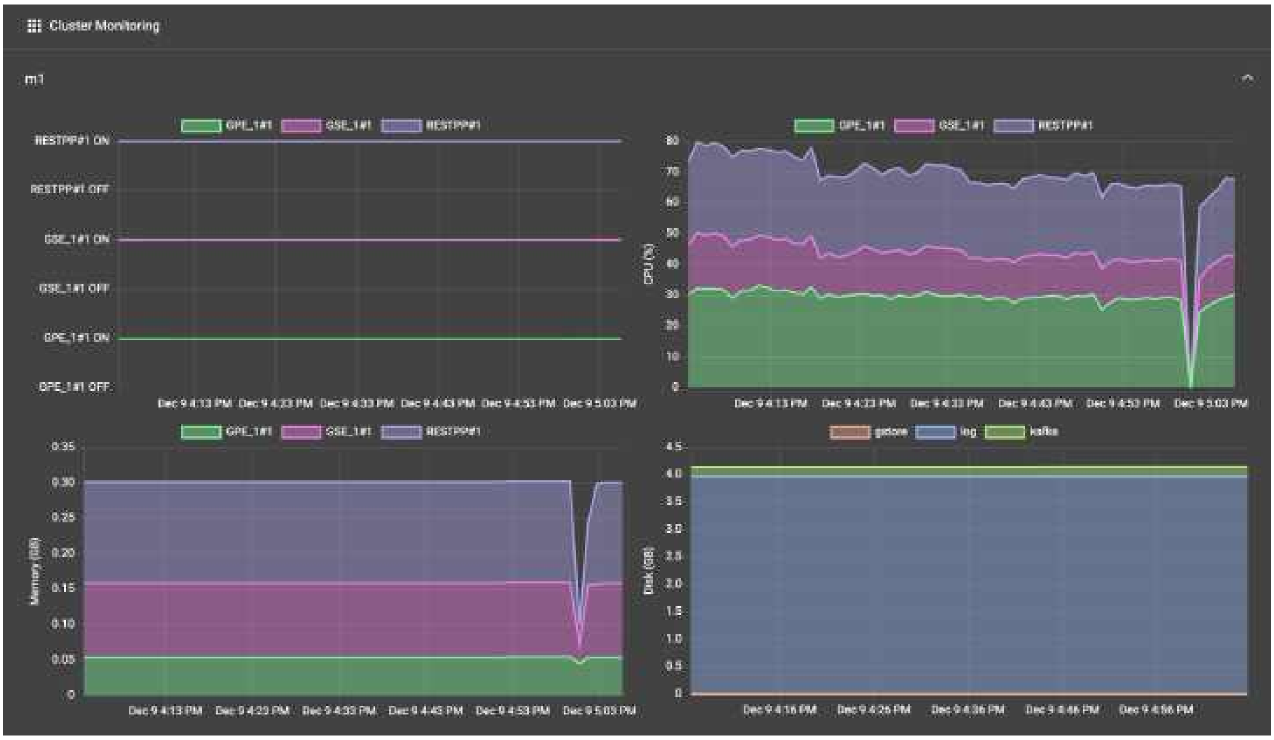
A Query Monitoring Panel includes three charts:

- QPS (number of queries completed per second)
- Timeout (fraction of the query calls which timed out and therefore did not finish)
- Latency (minimum, maximum, and average time to complete a query)



A Machine Monitoring Panel includes 4 charts. The first three charts break down the information among three processing-focused components (GPE, GSE, RESTPP). The last chart breaks down information among three components which may have large storage needs (GStore, Log files, and Apache Kafka).

- Service status: ON or OFF status for the given component
- CPU Usage: percentage of available CPU time used by the given component
- Memory Usage: GB used by the given component
- Disk Usage: GB used by the given component


Cluster Monitoring
m1
GPE_1#1
GSE_1#1
RESTPP#1
RESTPP#1 ON
RESTPP#1 OFF
GSE_1#1 ON
GSE_1#1 OFF
GPE_1#1 ON
GPE_1#1 OFF
CPU (%)
Memory (GB)
Disk (GB)
gstore
log
kafka
Dec 9 4:13 PM
Dec 9 4:23 PM
Dec 9 4:33 PM
Dec 9 4:43 PM
Dec 9 4:53 PM
Dec 9 5:03 PM
Dec 9 4:16 PM
Dec 9 4:26 PM
Dec 9 4:36 PM
Dec 9 4:46 PM
Dec 9 4:56 PM

Monitor

Monitor section in Admin Portal provides functionality for monitoring the TigerGraph cluster. The current functionality includes log management, where you can view, search and download the logs.

Loading...

Management

Loading...

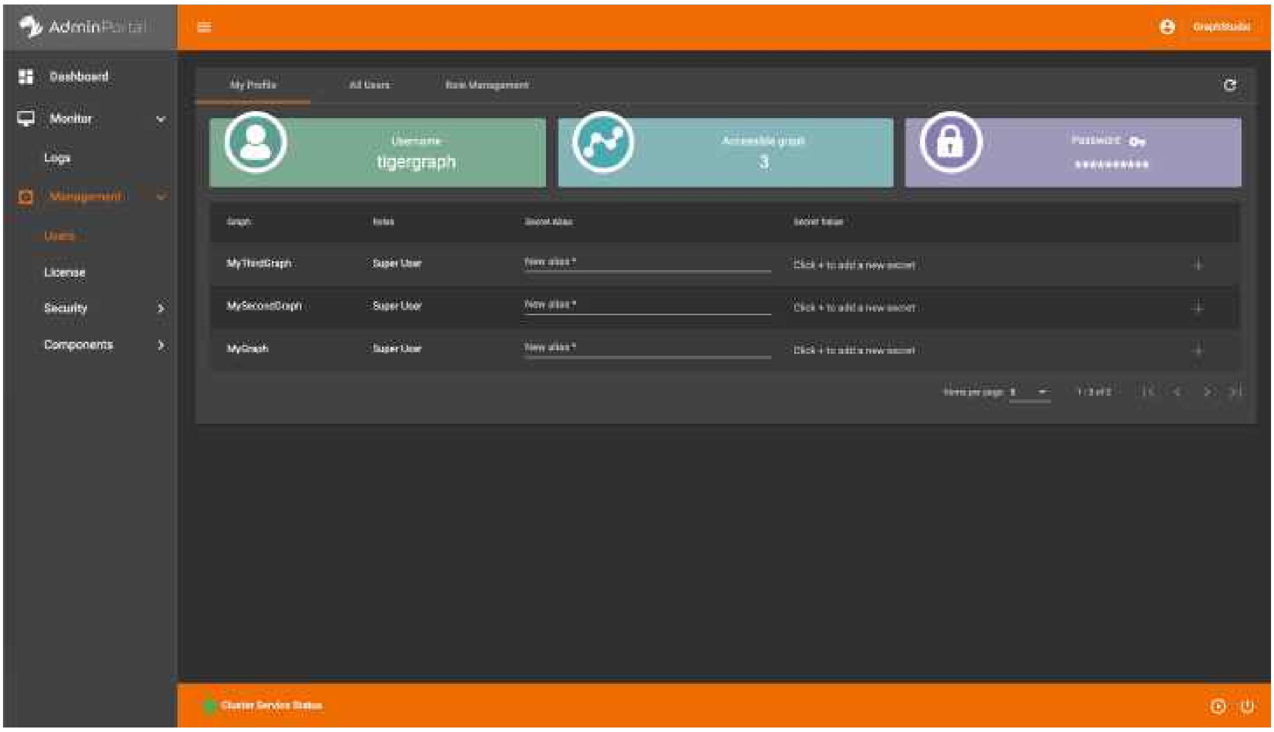
User Management

User Management Page

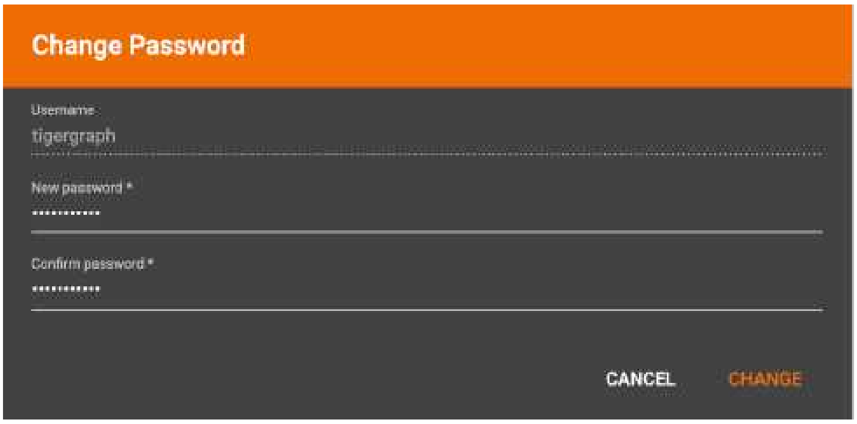
The User Management page has three tabs: My Profile, All Users and Role Management.

My Profile

All users can access the My Profile tab to manage themselves.



There are three information cards at the top: Username shows the user's username; Accessible graph shows how many graphs the user can access, and Password allows user to change password by clicking the key icon :



Below the cards are secret management section. You can create as many secrets for each graph as you want, but remember to copy the secret value out at the creation time. Once you refresh the page or go to another page and come back, the actual secret value will be masked for security. You need to provide an alias for the secret, and click the green plus sign to add the secret.

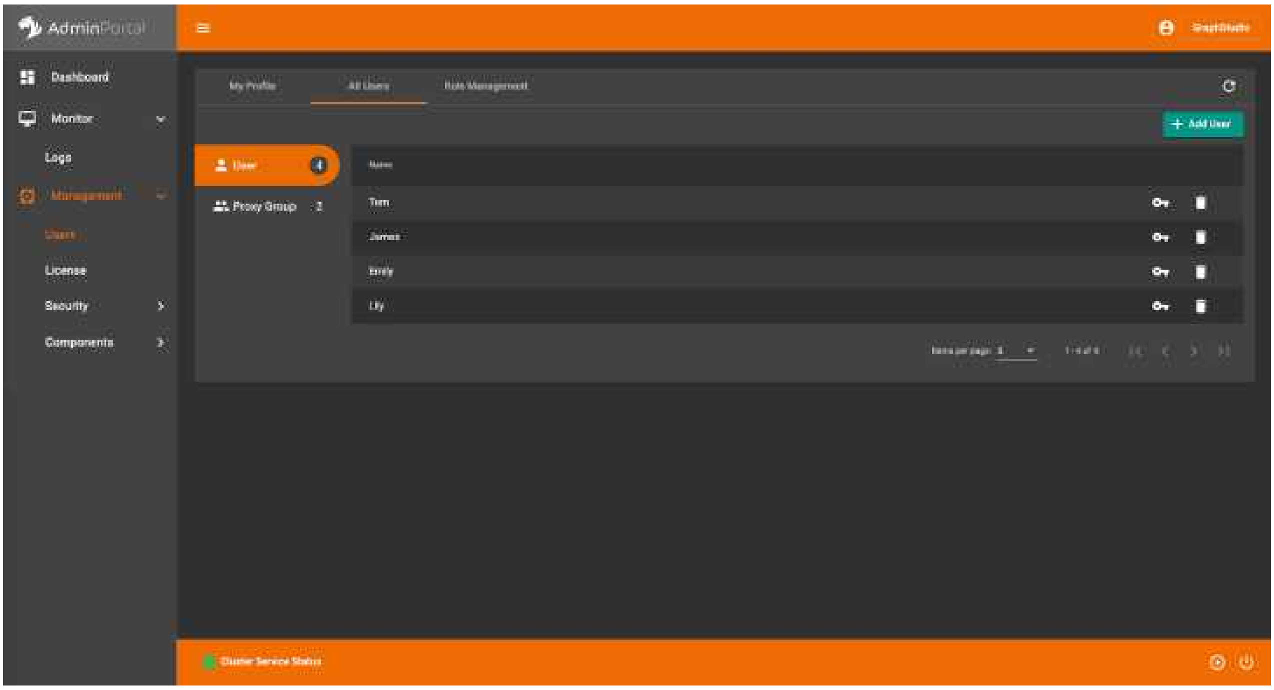
When token authentication is enabled, you need to use the secrets to request tokens and add the tokens in the HTTP request headers to call TigerGraph RESTful endpoints. Read more at: User Privileges and Authentication.

All Users

Due to the difference in users' information display compared to GSQL, Admin Portal doesn't allow users with admin role to a graph to create/edit/delete users or assign a user with a role to a graph. Superuser can manage users in this case. Users with Admin role to a local graph can revoke roles to other users for this local graph.

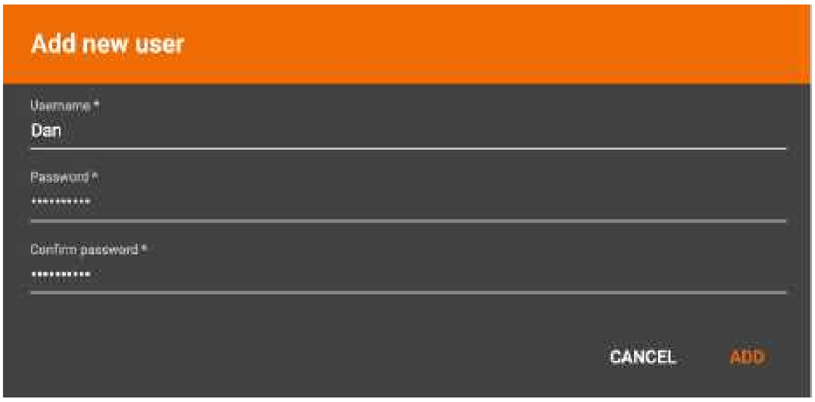
Both native users and proxy users are displayed in the User section. However, you can only change password for native users.

Only superuser can access this tab. Superuser can add new users, change other users' password, delete users, create proxy groups, change proxy rules of the groups, and delete proxy groups.

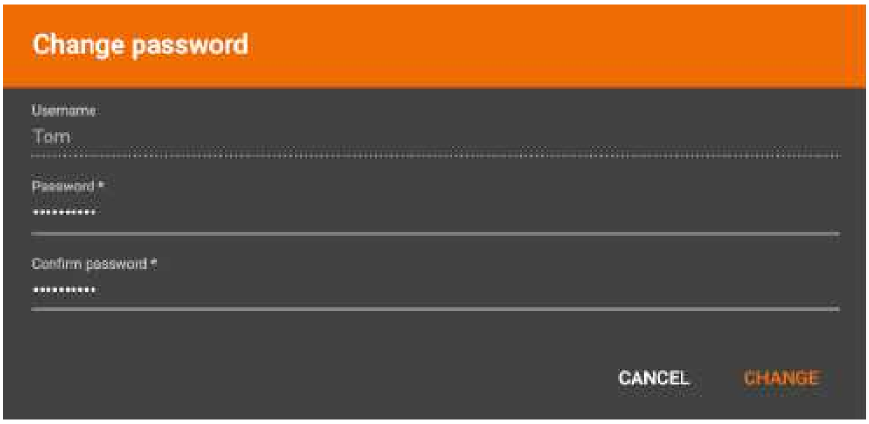


Click 

to add new users:



Click  to change other users' password:



Click  to delete a user.

Click 

to see all proxy groups:

Click 

to add new proxy groups:

Click  to edit the proxy rule of an existing proxy group:

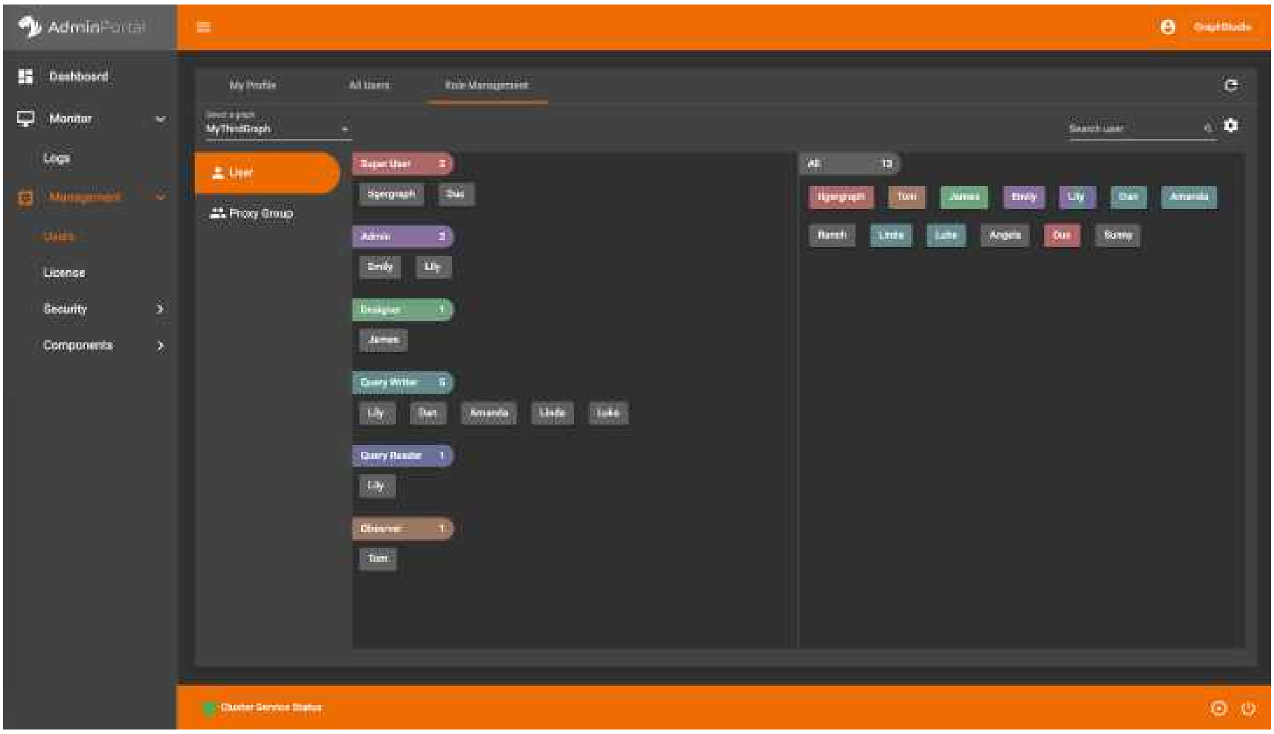
Click to delete a proxy group.

Read more about proxy group at User Privileges and Authentication.

Role Management

> Admin Portal does not indicate whether a user belongs a proxy group. You can find out more from GSQL client.
>
> Admin Portal shows a combination of roles for proxy users. Proxy user which belongs to a proxy group inherits all the roles from the proxy group plus all the roles the proxy user has. Therefore, if you want to revoke a role from a proxy user, you need to revoke the role from the proxy group first. Then you might need to revoke the same role from the proxy user if the proxy user also has that role.

Superusers and Admin users can access this tab to assign different roles to users on different graphs. Superusers can assign superuser role to other users, while admin users cannot.

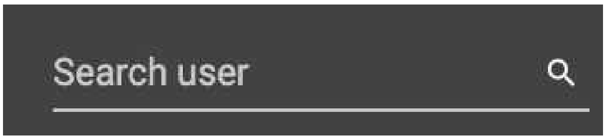


Select a graph

You can use the dropdown menu at the top left corner to select on which graph you want to assign different roles to different users:

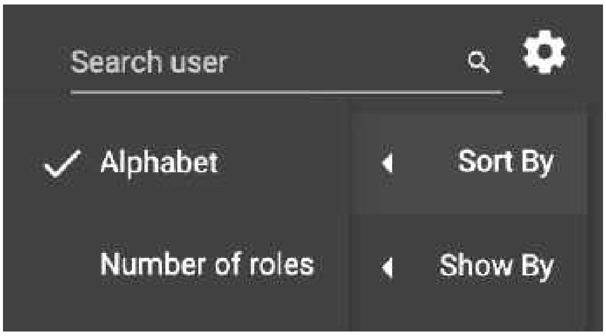
Search user

You can type in text to search for users:

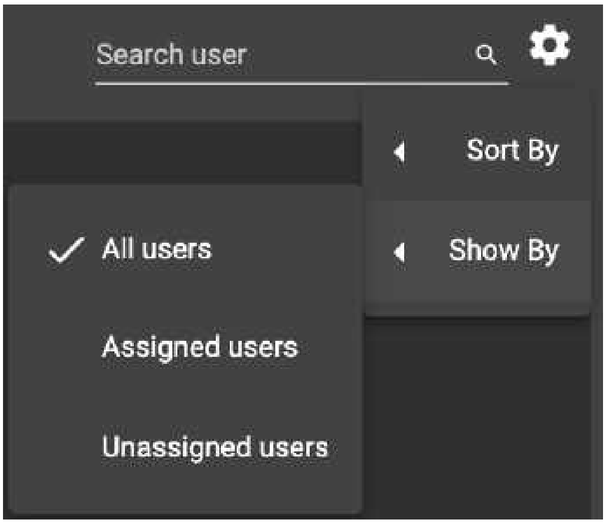


Settings for users

You can sort the users by alphabetic order or number of roles they have:



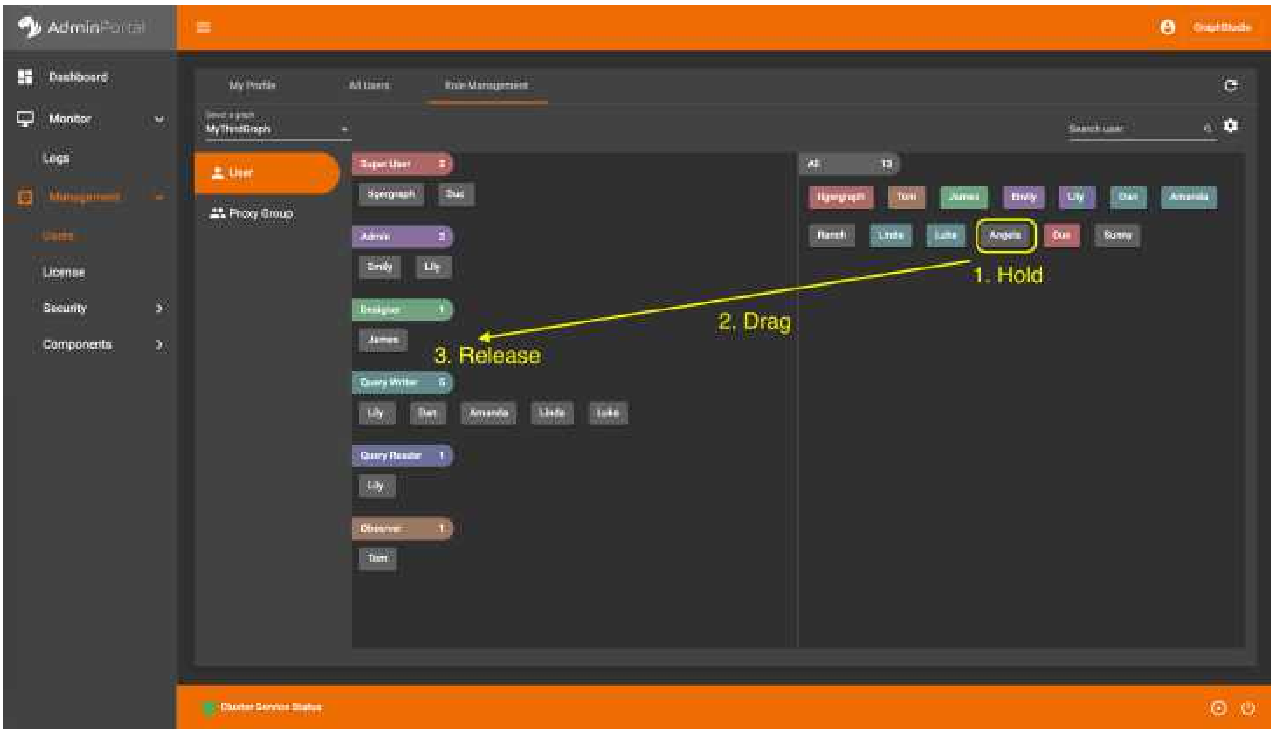
You can choose to show all users, show the users who have role(s), or show the users who don't have a role:



Grant a new role to a user

You have two ways to grant a new role to a user.

The first way is drag-and-drop. You can drag a user from the right section to under the role you want to grant.



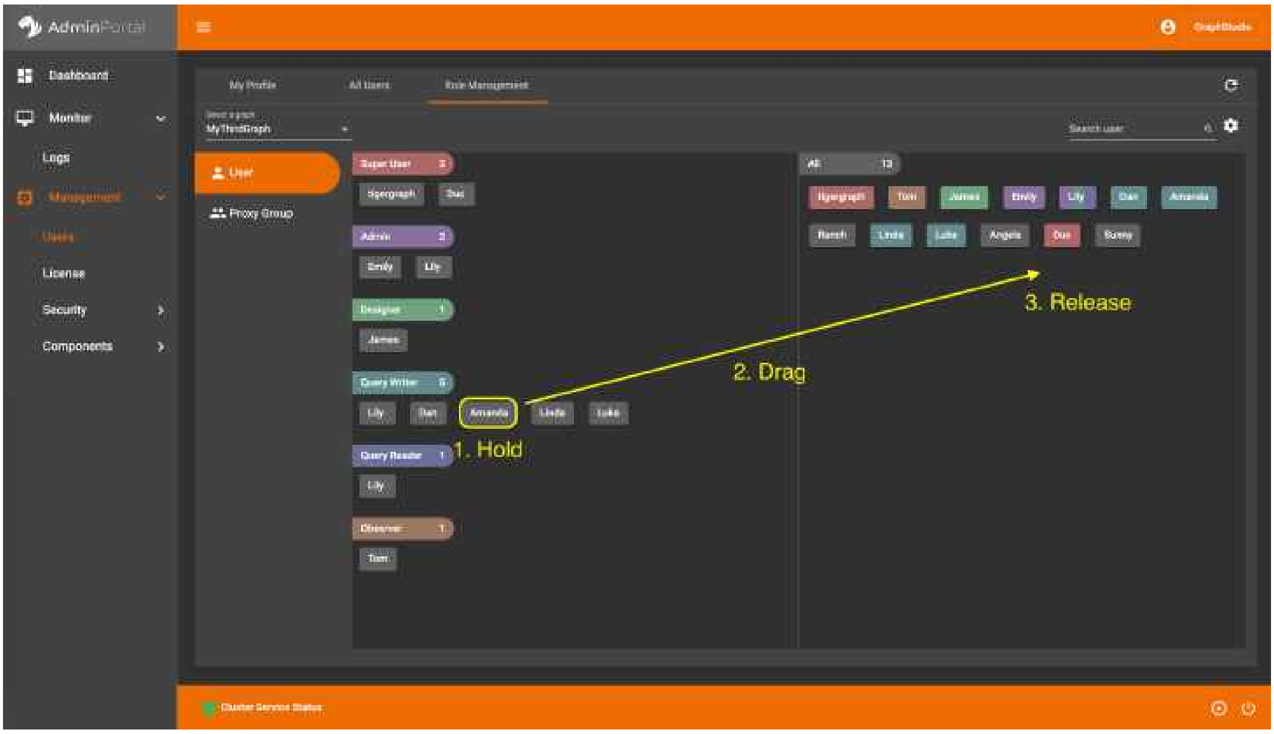
After the role is granted successfully, the user's name will appear under the corresponding role, and the role's color will appear on the user at the right section. You can grant multiple roles to the same user, and multiple colors will show on the user at the right section to indicate these roles.

The second way is right click a user, and choose the role you want to grant:

Revoke a role from a user

You have two ways to revoke a role from a user.

The first way is drag-and-drop. You can drag a user from the role you want to revoke to the right section:

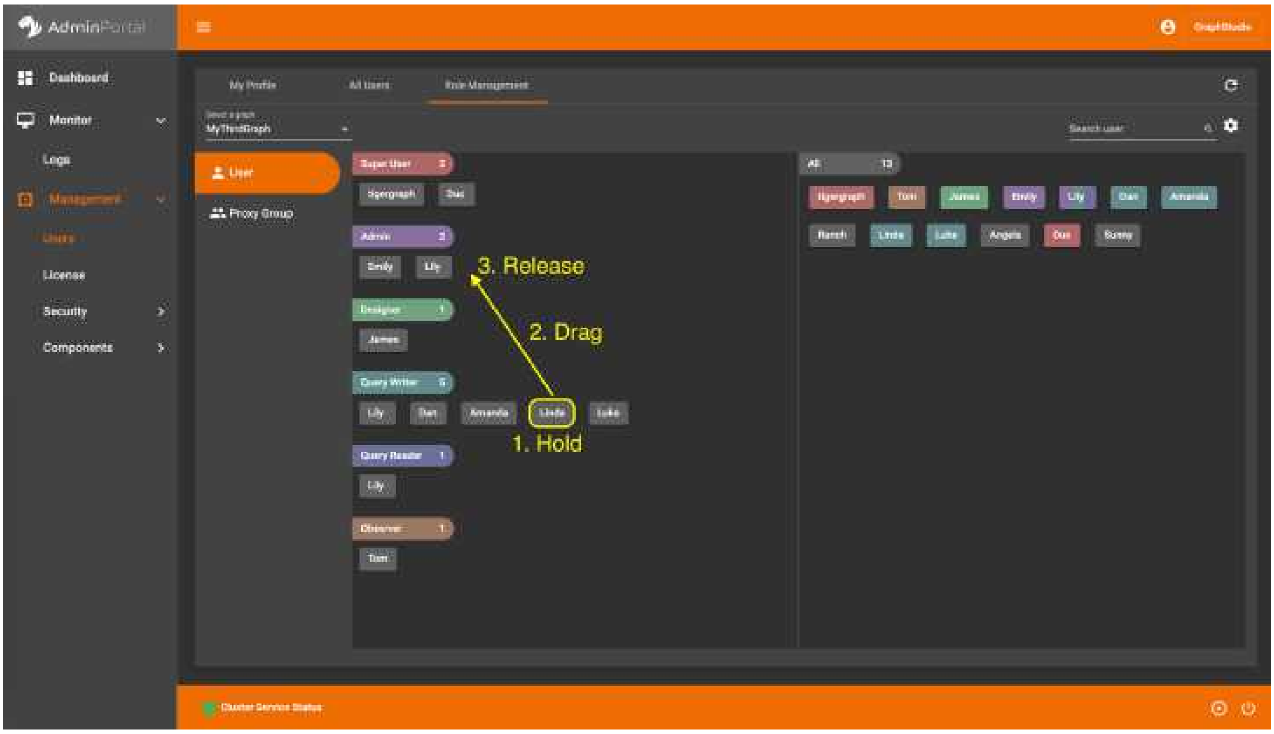


After the role is revoked successfully, the user's name will disappear under the corresponding role, and the role's color will disappear from the user at the right section.

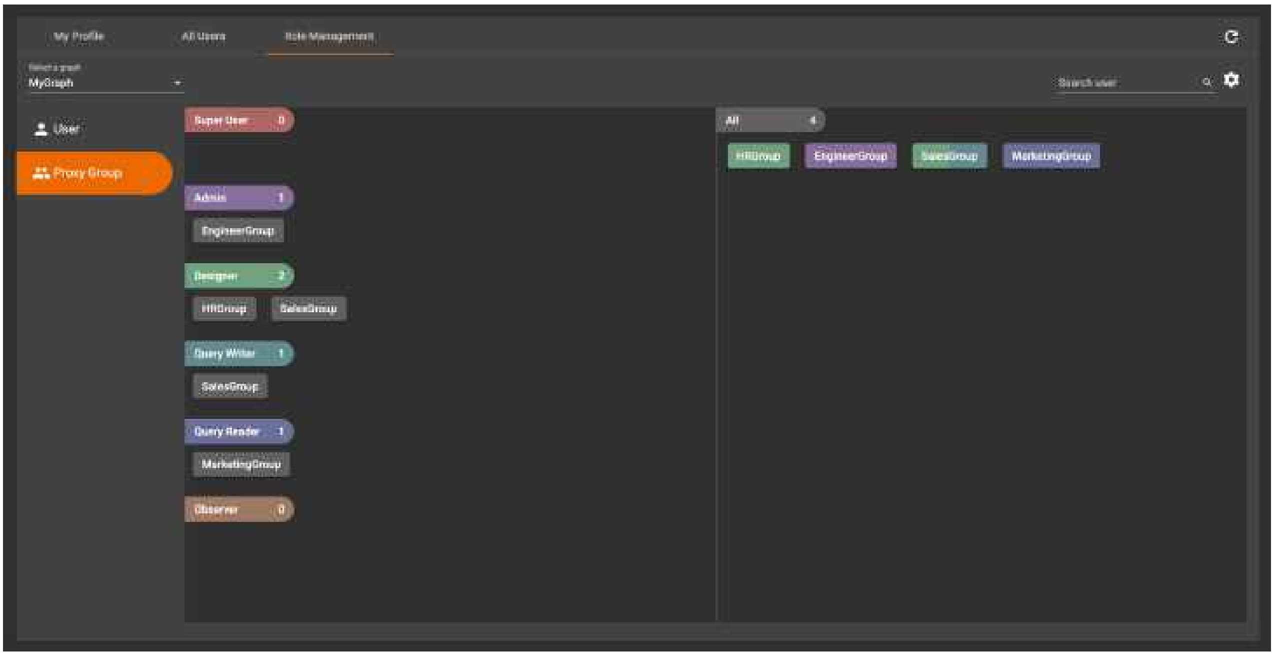
The second way is right click a user, and choose the role you want to revoke:

Change a user's role

You can drag a user from under one role to under another role at the left section. This is a shortcut to change the user's role:



Similarly, you can also click  to switch to proxy groups and manage their roles.



Tip: If you manage users through GSQL, please click on refresh button on the top right corner to reflect the changes in Admin Portal.

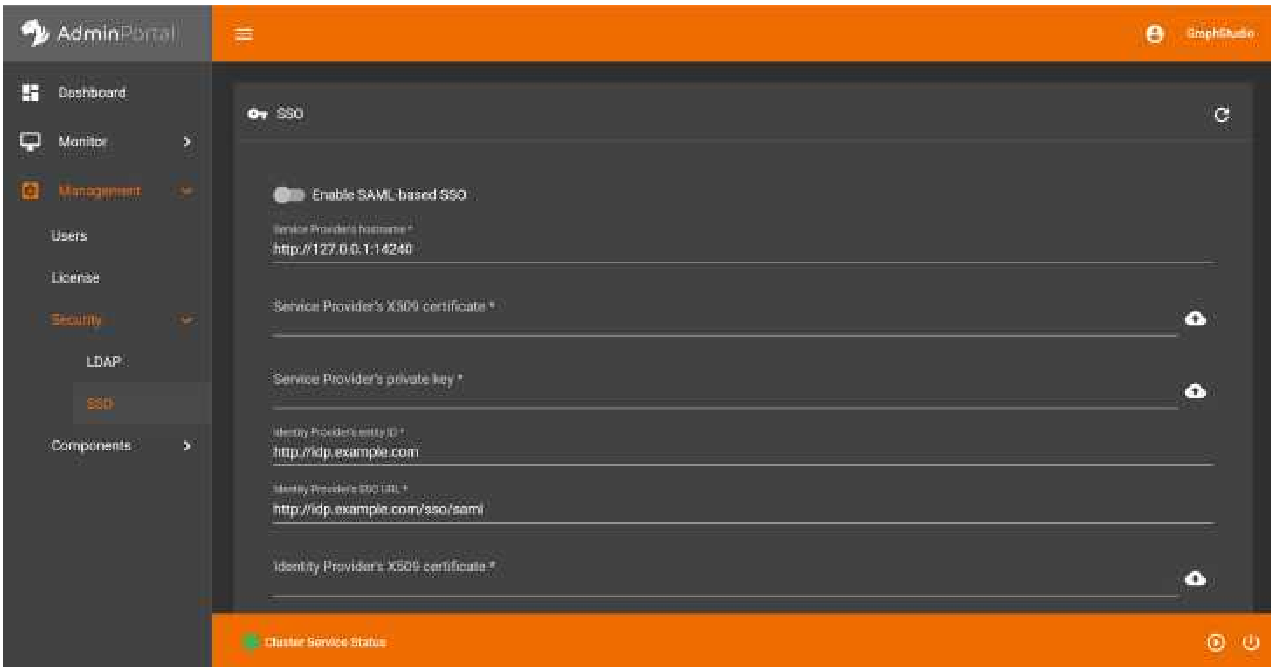
Security

Loading...

SSO

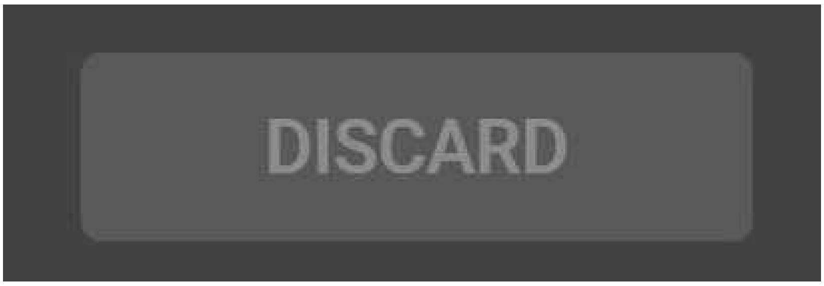
SSO Page

You can set up and configure SSO settings on this page.



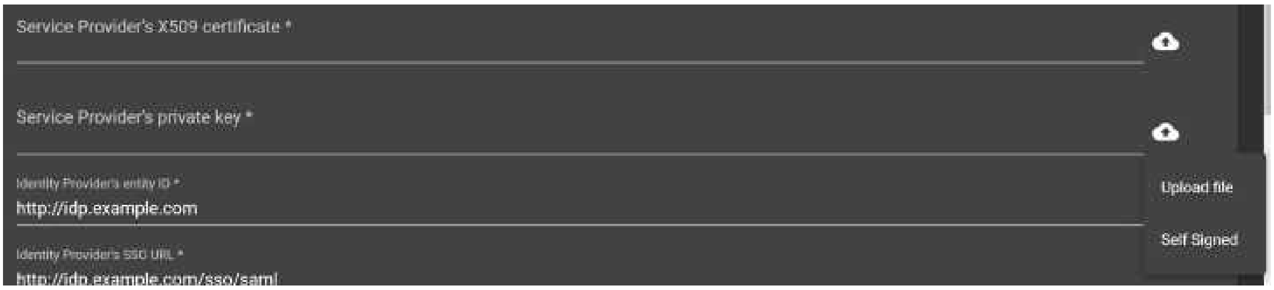
When you don’t have any changes, the DISCARD and APPLY buttons are disabled.

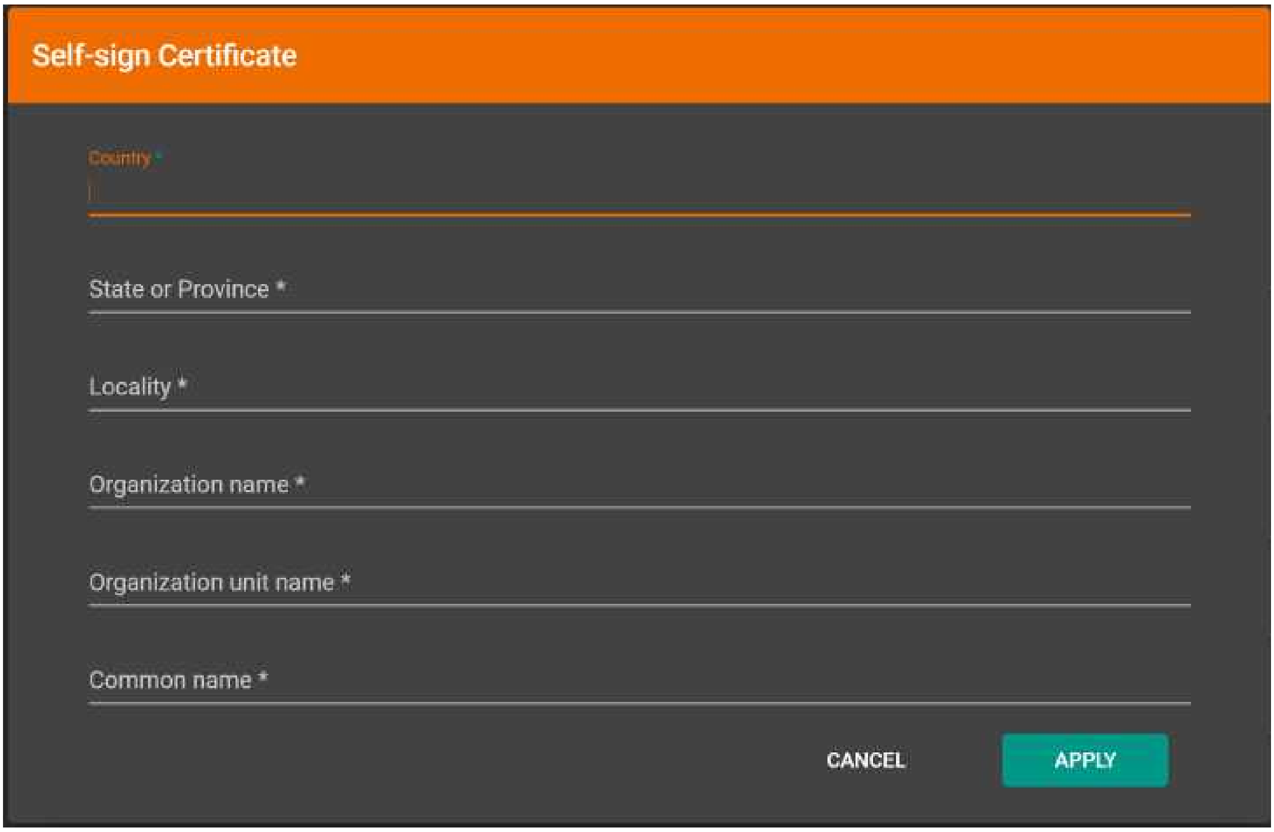
You can clear your changes by clicking the DISCARD button at the bottom right corner of the page.



`Service Provider's X509 certificate` and `Service Provider's private key`

1. Can replace input by uploading files.
    - Click the file upload button next to the input box, and a drop-down menu for uploading files will appear. Click the upload file option to upload the desired file.
2. Can be generated by manual input.
    - Click the file upload button next to the input box, and a drop-down for uploading files will appear. Click the self signed button. There will be a pop-up box to fill in the information. Items marked with * are required.

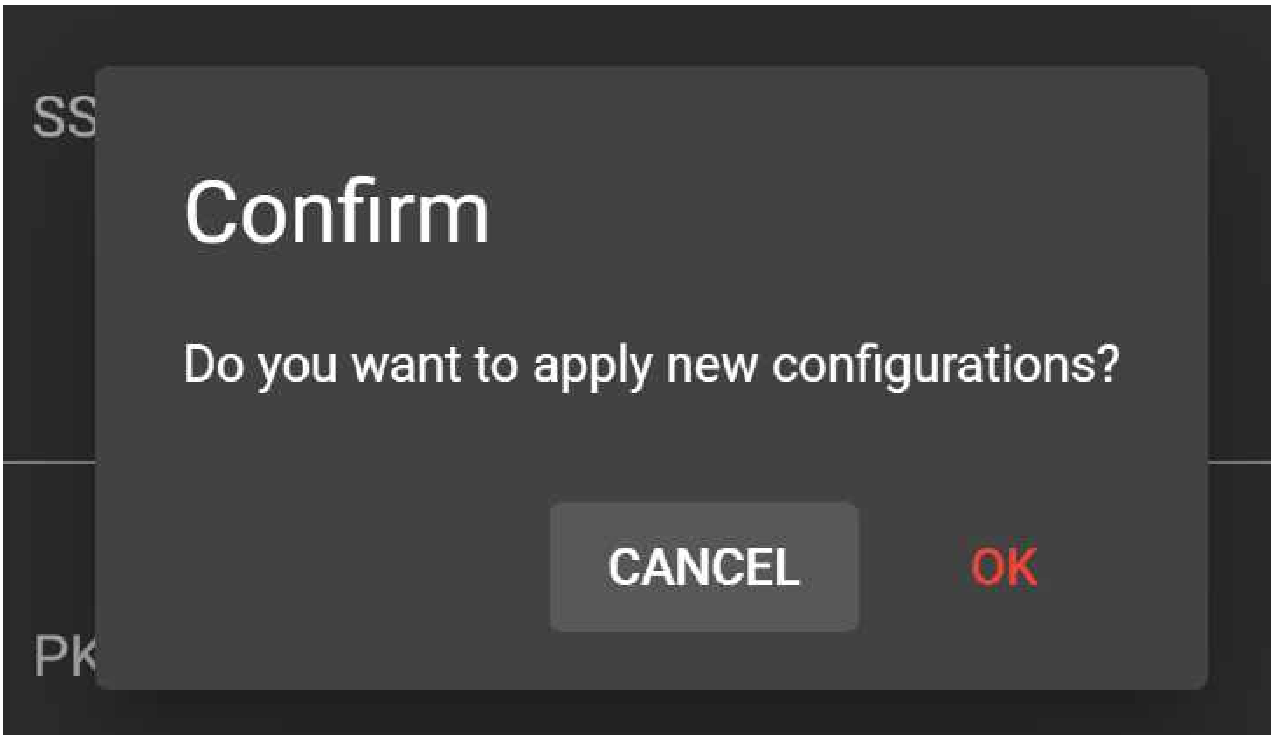




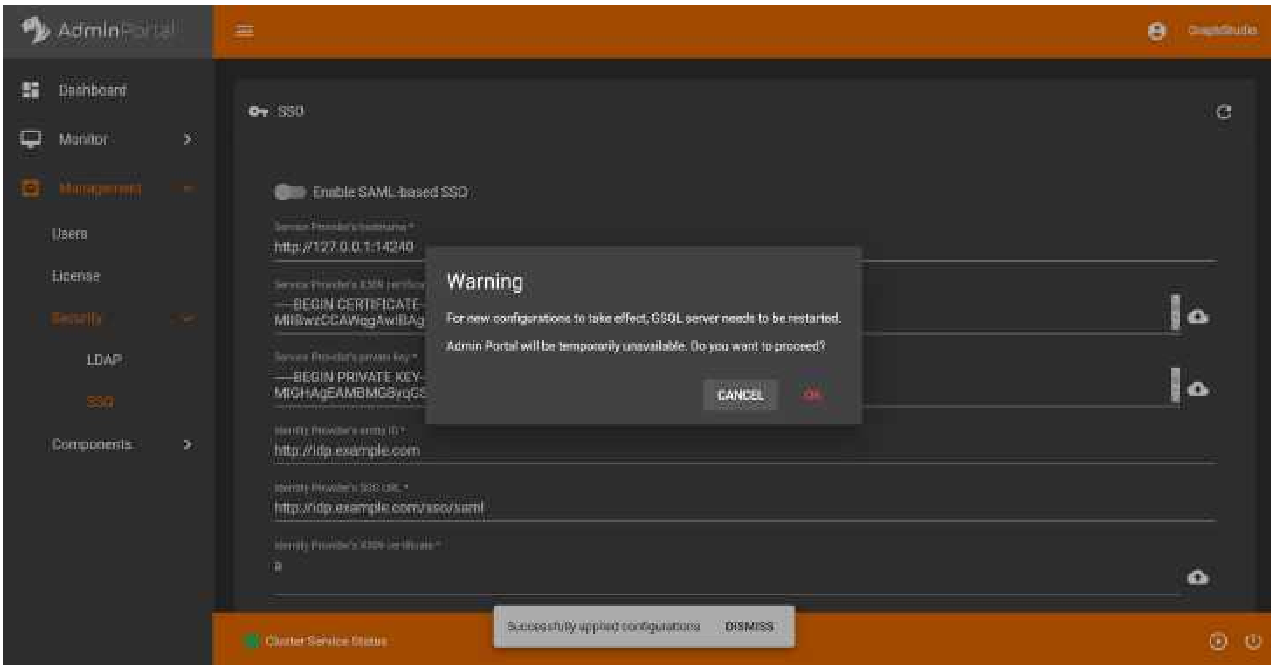
If you want to get the default data again, you can click the refresh button in the upper right corner of the page.

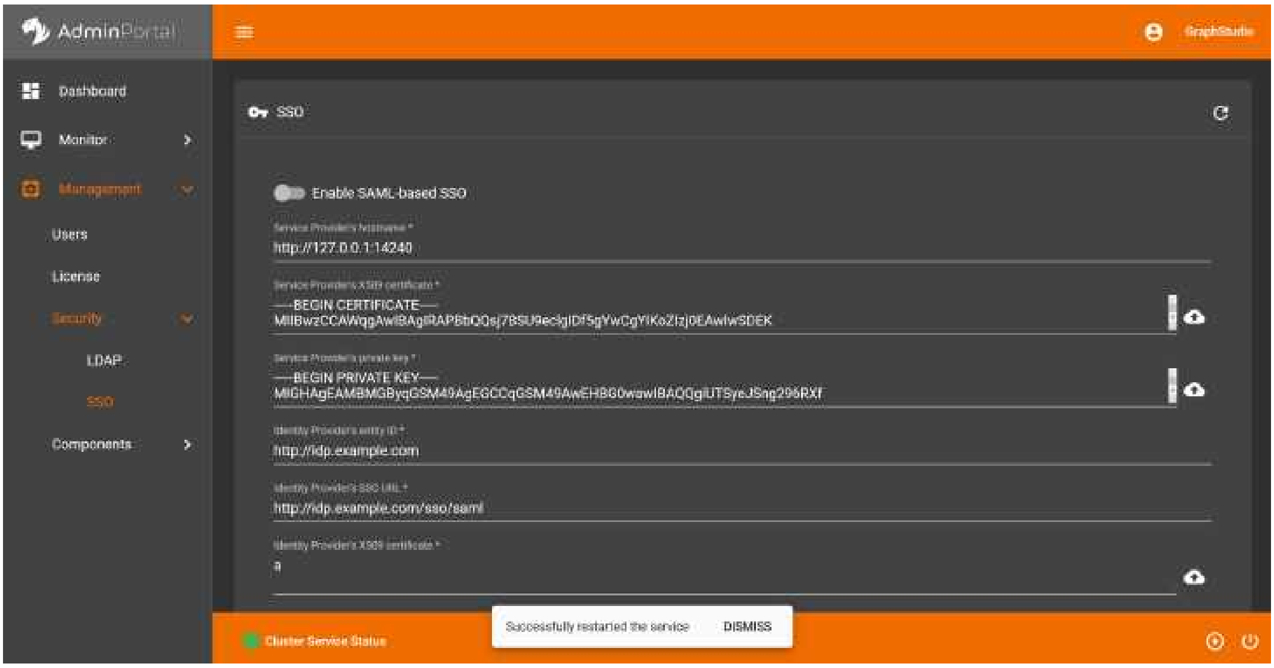
The fields with * in the label are required. When users do not fill the required fields and click the APPLY button, the following error will appear:

If everything is ready, click the APPLY button and a confirmation box will pop up.



Click the CANCEL button, the pop-up window will be closed. Then click the OK button, it will prompt the success message. Since any configuration modification requires restarting the server service, a pop-up box will appear to confirm whether to restart the service.After clicking the OK button, configuration changes will start. Users will be notified after server restarting when all the process is completed.


AdminPortal
GraphStudio
Dashboard
Monitor
Management
Users
License
Security
LDAP
SSO
Components
SSO
Enable SAML-based SSO
http://127.0.0.1:14240
-----BEGIN CERTIFICATE-
MIIBwzCCAWqgAwIBAg
-----BEGIN PRIVATE KEY-
MIGHAgEAMBMGByqGS
http://idp.example.com
http://idp.example.com/sso/saml
a
Warning
For new configurations to take effect, GSQL server needs to be restarted.
Admin Portal will be temporarily unavailable. Do you want to proceed?
CANCEL
OK
Cluster Service Status
Successfully applied configurations
DISMISS

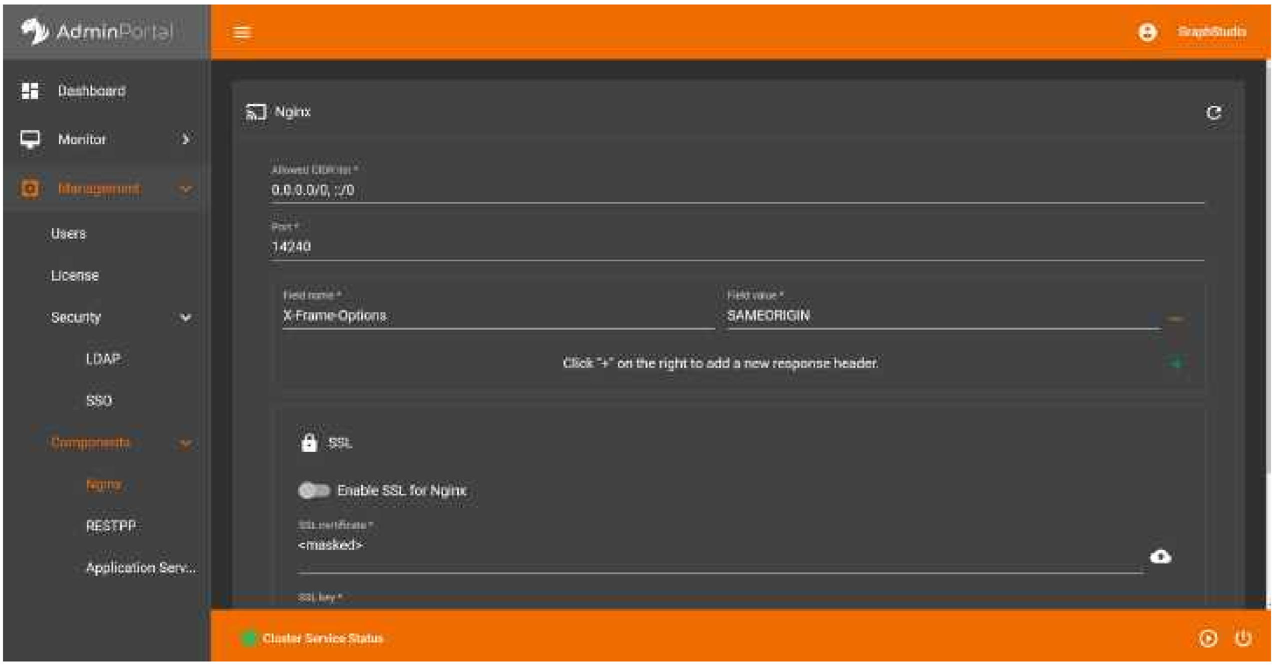

AdminPortal
GraphStudio
Dashboard
Monitor
Management
Users
License
Security
LDAP
SSO
Components
SSO
Enable SAML-based SSO
Service Provider's hostname *
http://127.0.0.1:14240
Service Provider's X509 certificate *
-----BEGIN CERTIFICATE-----
MIIBwzCCAWqgAwIBAgIRAPBbQQsj7BSU9eclgIDf5gYwCgYIKoZIzj0EAwIwSDEK
Service Provider's private key *
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgIUTSyeJSng296RXf
Identity Provider's entity ID *
http://idp.example.com
Identity Provider's SSO URL *
http://idp.example.com/sso/saml
Identity Provider's X509 certificate *
a
Cluster Service Status
Successfully restarted the service
DISMISS

Components

Nginx

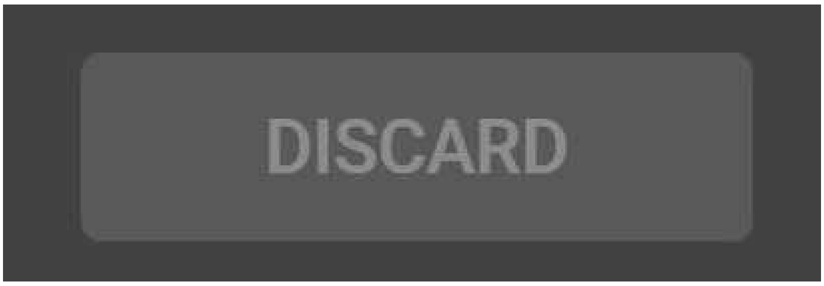
Nginx Page

You can configure Nginx settings on this page. The following settings can be configured: SSL configuration, allowed CIDR list (whitelist IP), and response header.



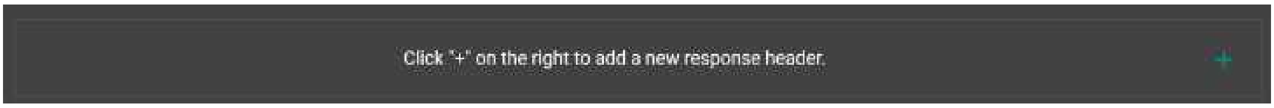
When you don’t have any changes, the DISCARD and APPLY buttons are disabled.

You can clear your changes by clicking the DISCARD button at the bottom right corner of the page.

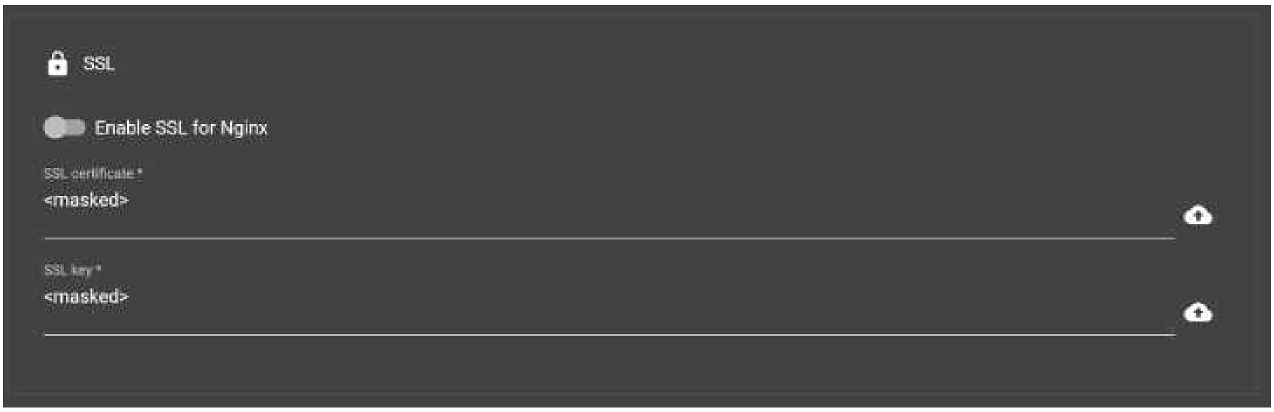


`Allowed CIDR list` are separated by `,`

The `response header` can be empty. In addition, you can also add one by clicking the + on the right side of the `response header` box, or you can delete the specified one by clicking-on the right side of a `response header`.



SSL for Nginx

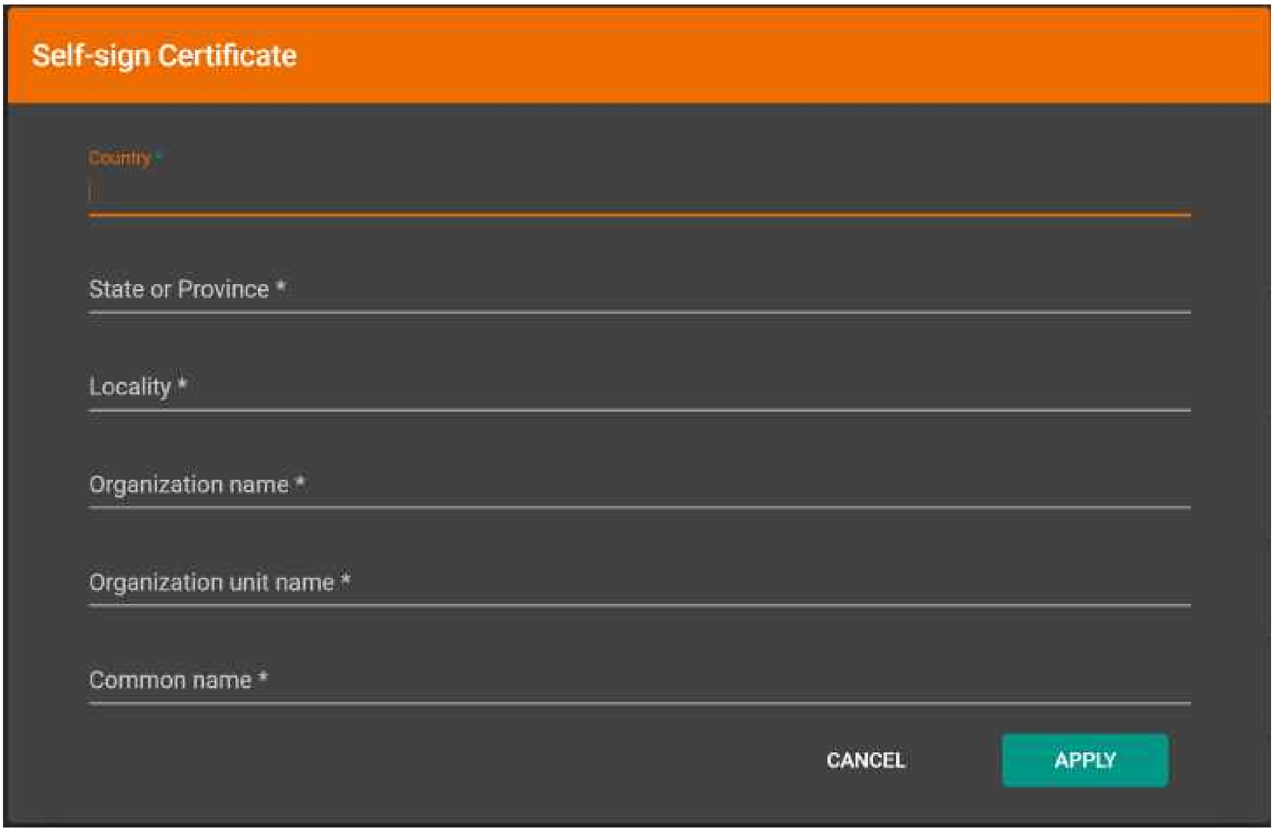


`SSL certificate` and `SSL key`

1. Can update input by uploading files.

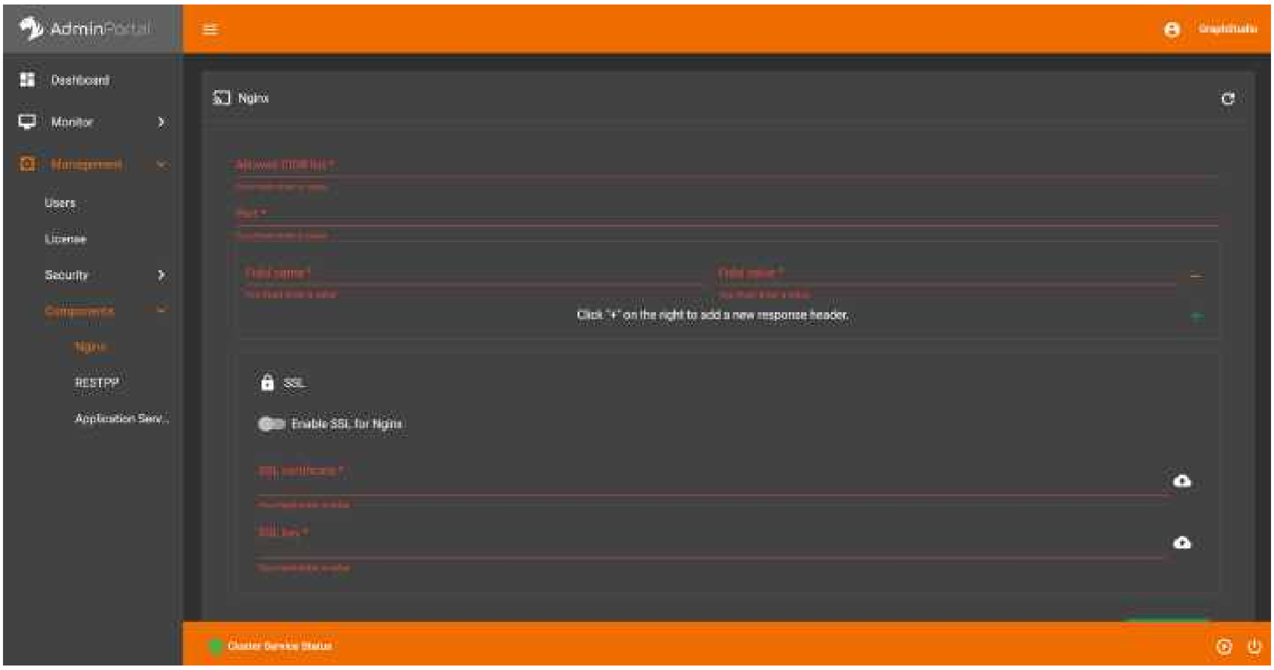
- Click the file upload button next to the input box, and a drop-down menu for uploading files will appear. Click the upload file option to upload the desired file.

2. Can be generated by manual input.
    - Click the file upload button next to the input box, and a drop-down menu for uploading files will appear. Click the self signed button. There will be a pop-up box to fill in the information. Items marked with * are required.

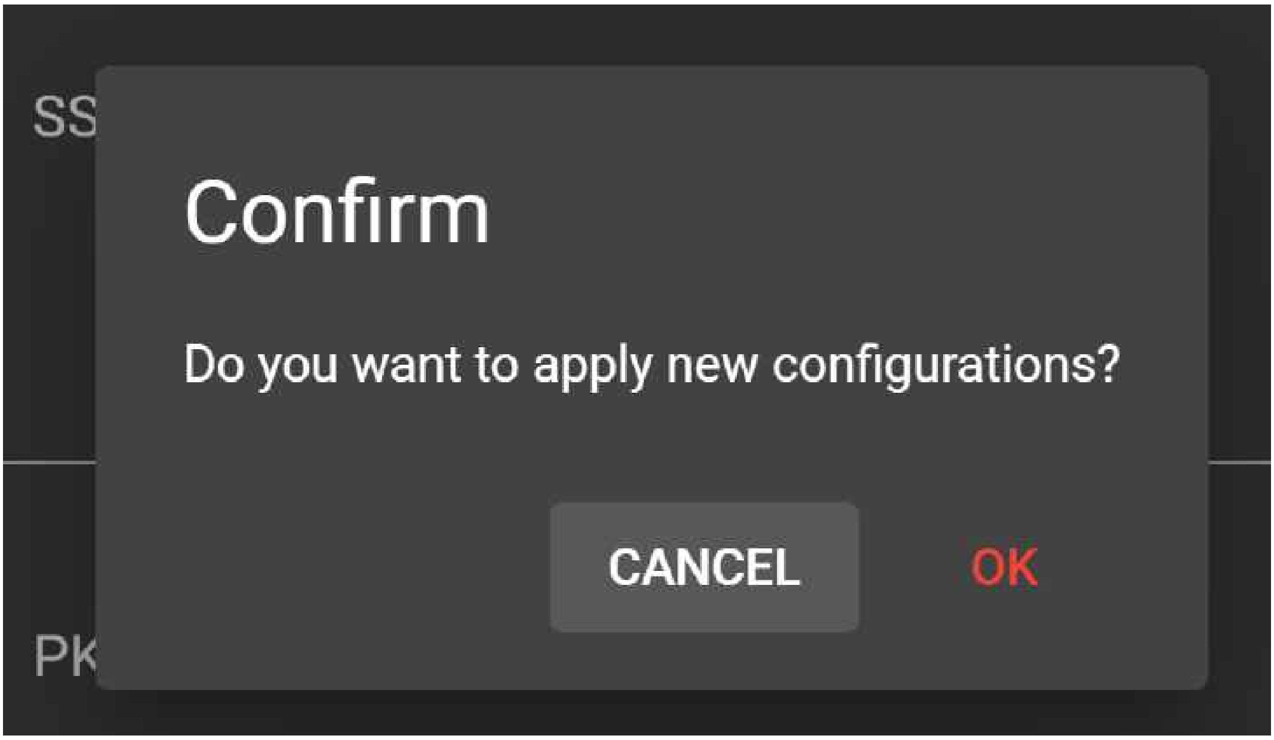


If you want to get the default data again, you can click the refresh button on the upper right corner of the page.

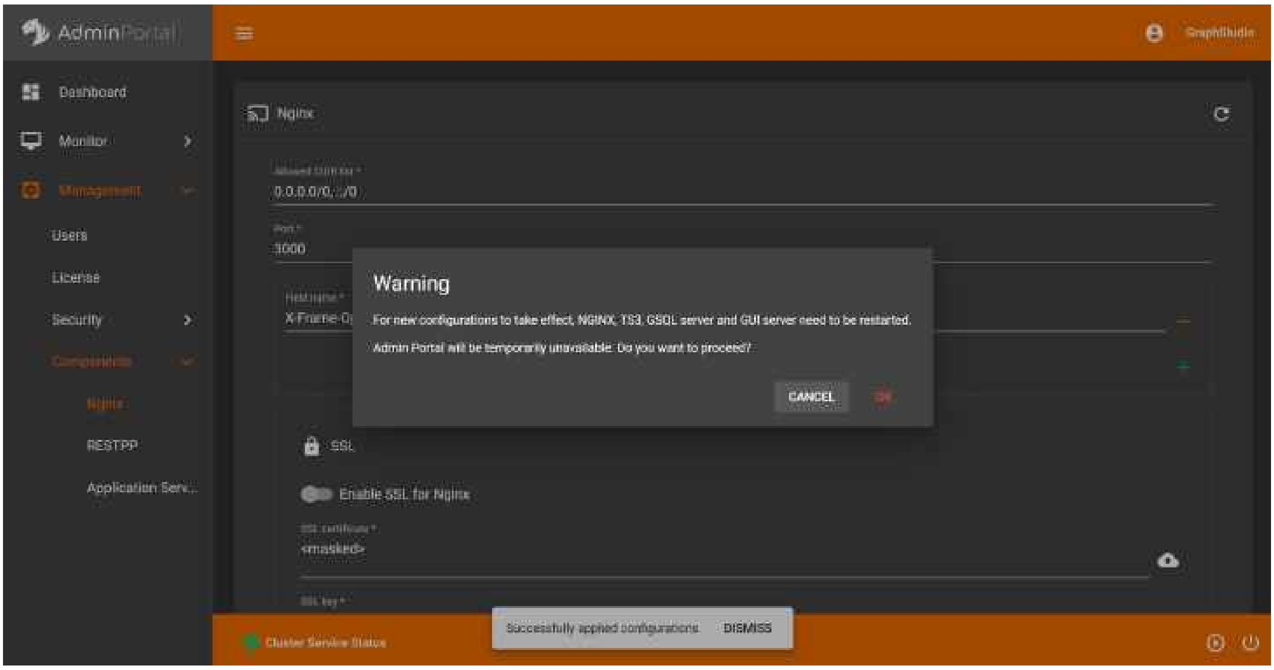
The fields with * in the label are required entries. When users do not fill in the required items and click the APPLY button, the following error will appear:

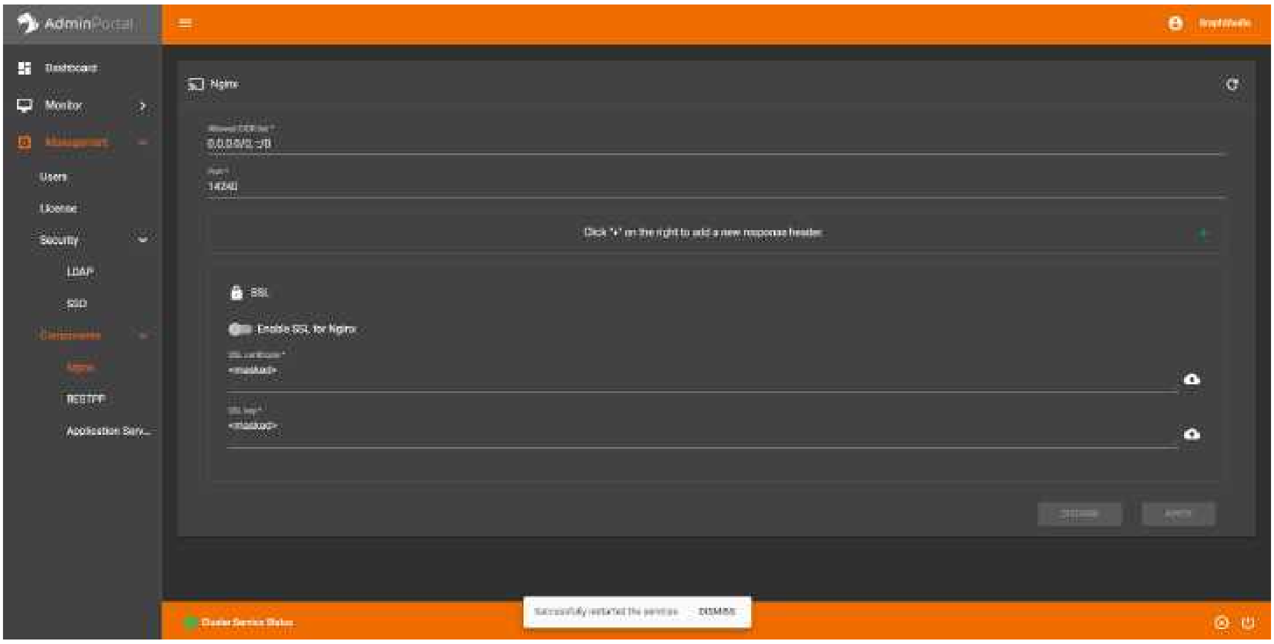


If everything is ready, click the APPLY button and a confirmation box will pop up



Click the CANCEL button, the pop-up window will be closed. Click the OK button, it will prompt the success message.Since configuration change requires restarting the server service, a pop-up box will appear to confirm whether to restart the service. After clicking the OK button, it will take a while for the configuration changes and server restart. You will be notified after all the process is completed.





After upgrading to v3.1, downgrading from HTTPS to HTTP (turn off SSL) will result in user authentication failure

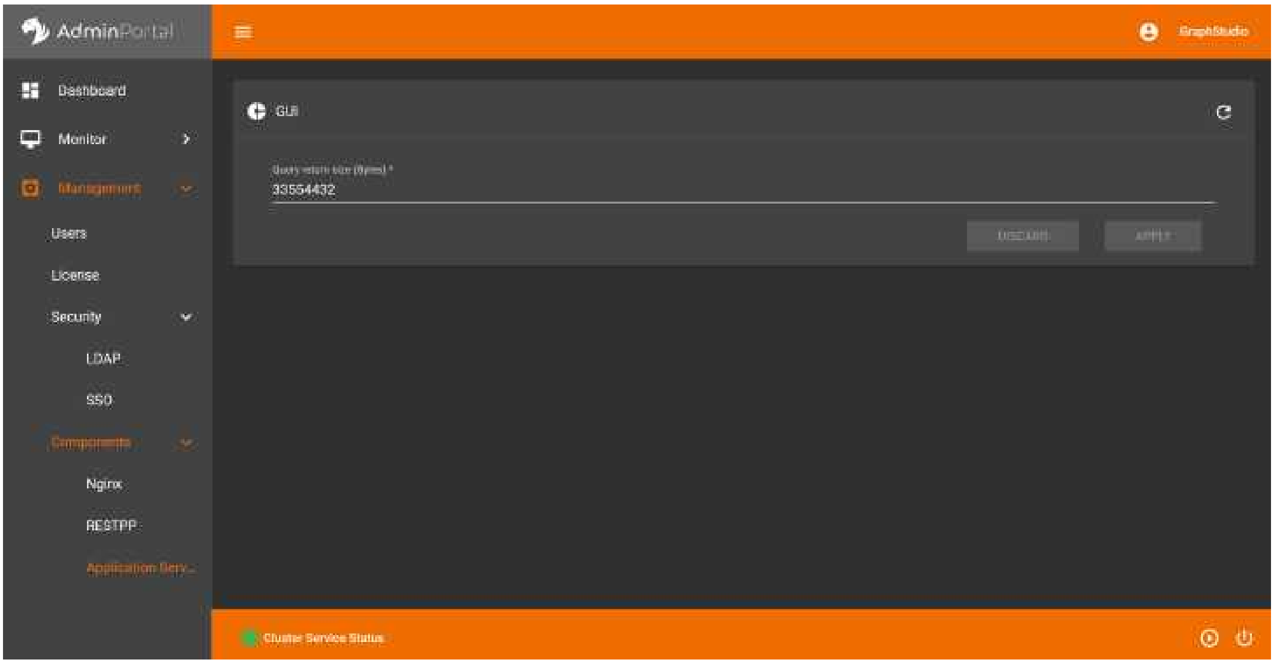
In v3.1, TigerGraph enhanced the security configuration for cookie by setting a secure flag when SSL is enabled, which may trigger an issue when using Chrome to turn off SSL. Due to this security feature, when downgrading from HTTPS to HTTP, Chrome will not allow setting the same cookie. This will result in user authentication failure because the cookie fails to be set. To solve this problem, users can clear the cache from Chrome and refresh the browser. Enabling from HTTP to HTTPS will work as expected.

Loading...

Application Server (GUI)

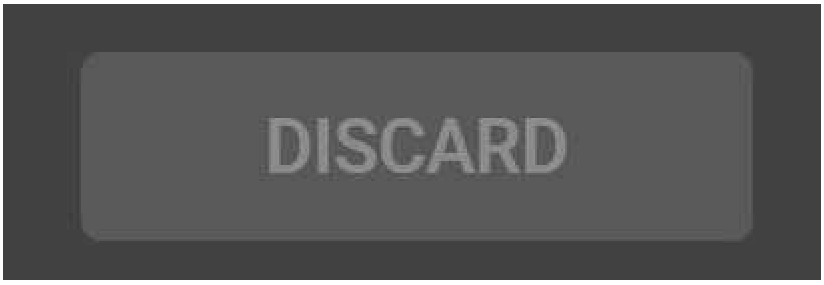
Application Server (GUI) Page

You can configure Application Server (GUI) settings through this page. Application Server is the backend component powering both GraphStudio and Admin Portal. Allowed setting includes: Query return size.



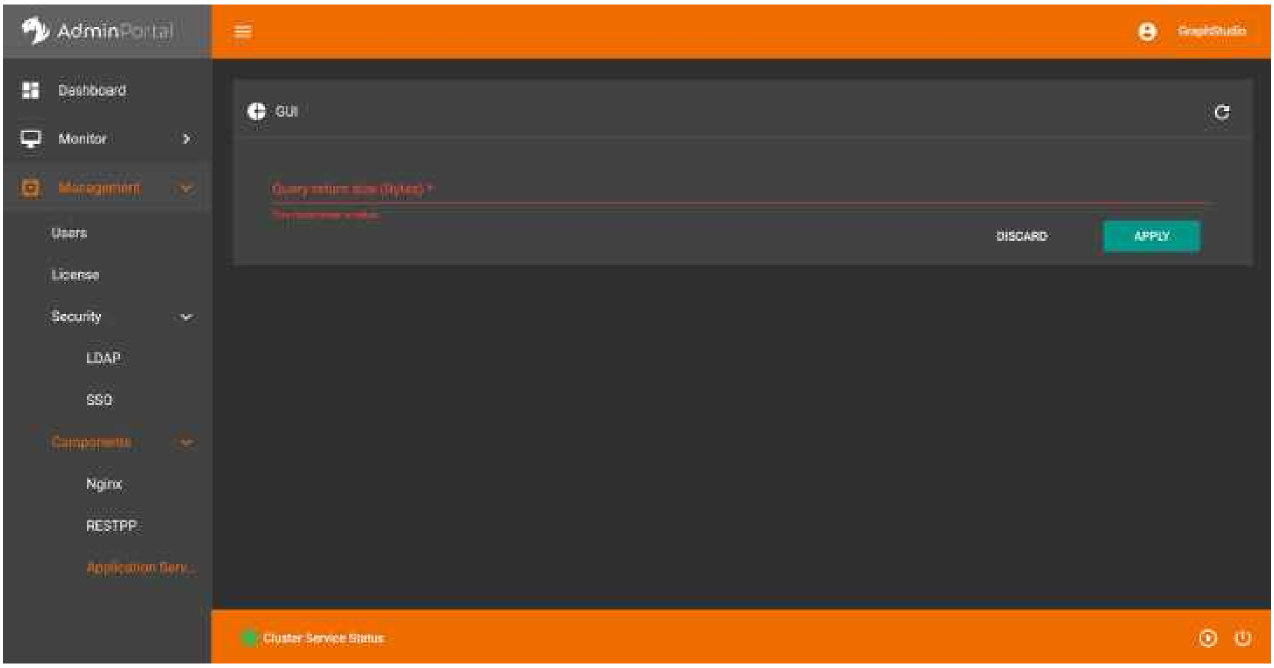
When you don’t have any changes, the DISCARD and APPLY buttons are disabled.

You can clear your changes by clicking the DISCARD button at the bottom right corner of the page.

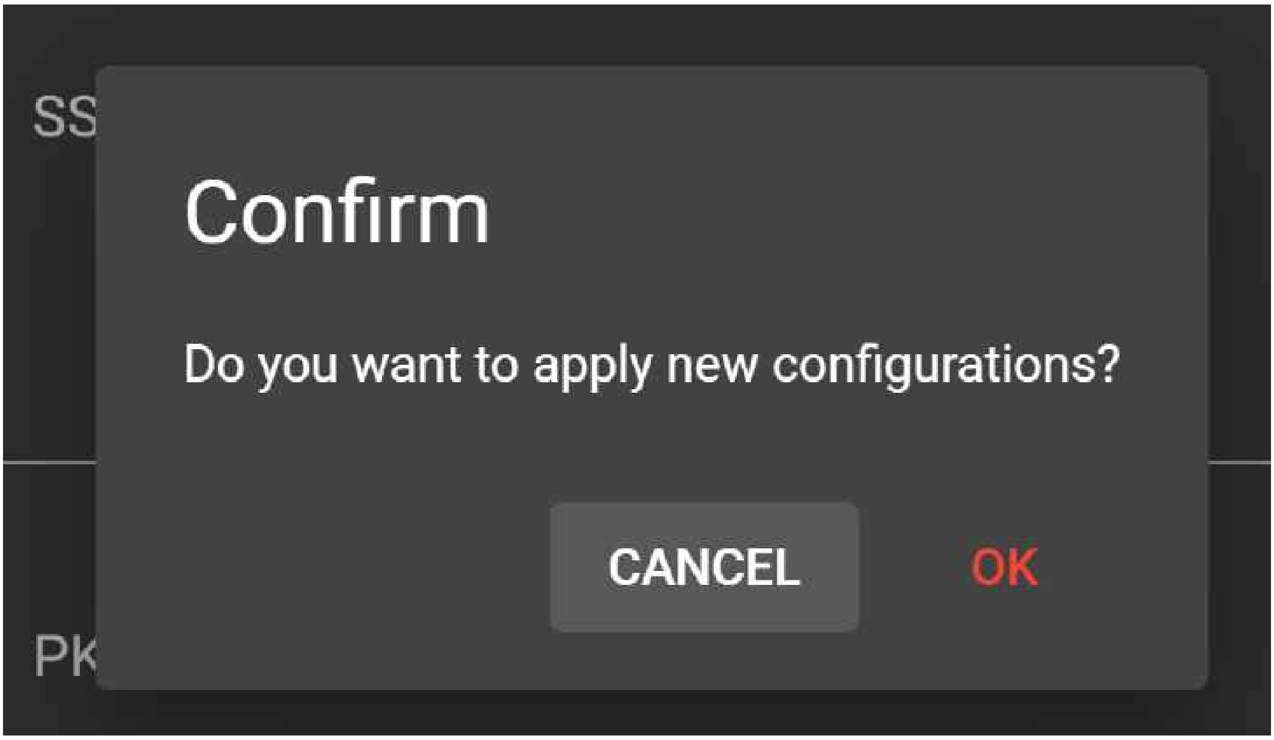


If you want to get the default data again, you can click the refresh button in the upper right corner of the page.

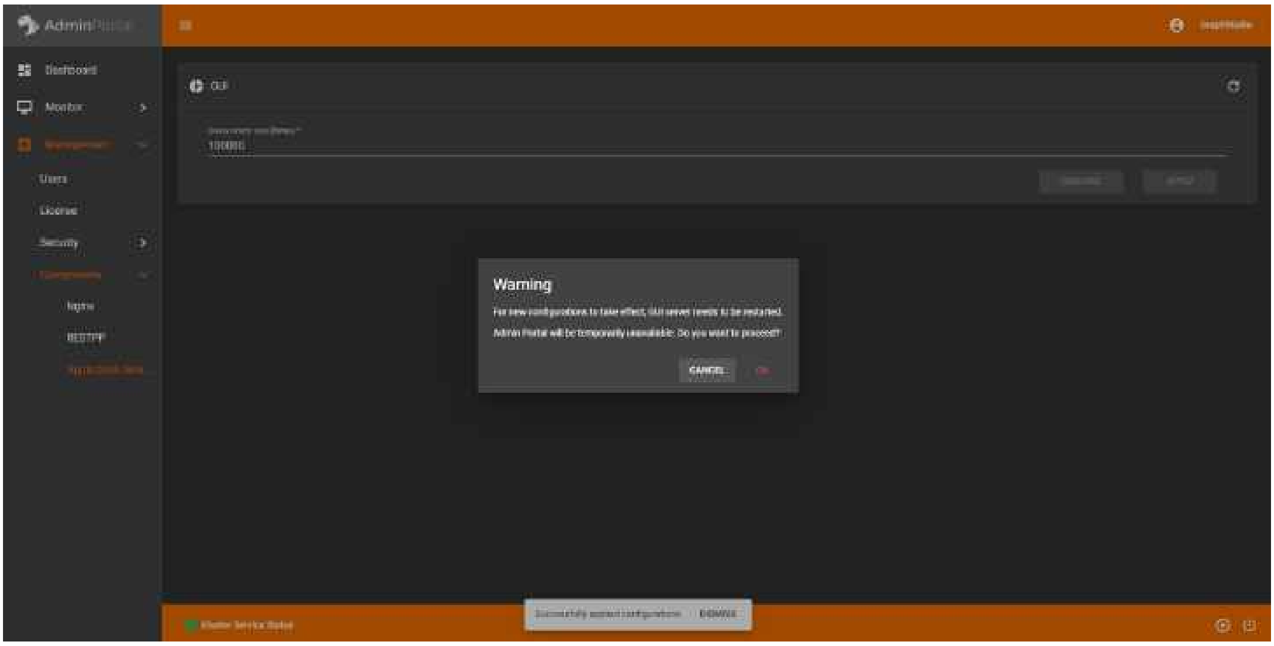
The fields with * in the label are required entries. When users do not fill in the required items and click the APPLY button, the following error will appear:

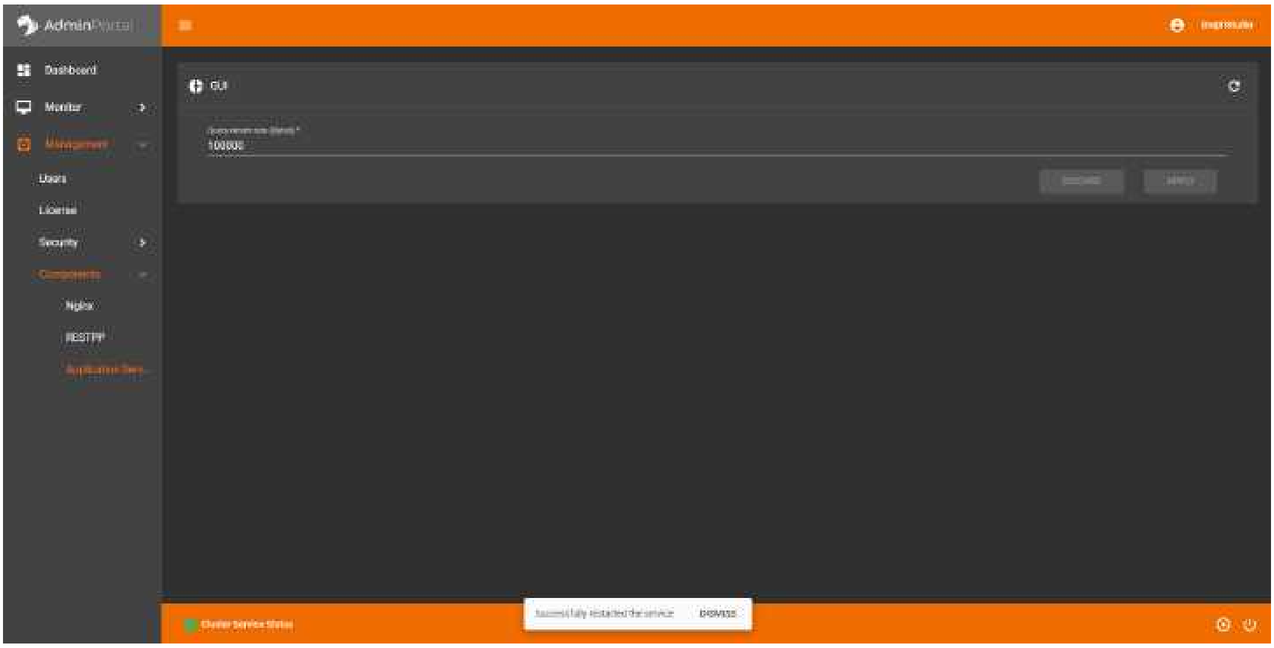


If everything is ready, click the APPLY button and a confirmation box will pop up.



Click the CANCEL button, the pop-up window will be closed. Click the OK button, it will prompt the success message.Since configuration change requires restarting the server service, a pop-up box will appear to confirm whether to restart the service. After clicking the OK button, it will take a while for the configuration changes and server restart. You will be notified after all the process is completed.




AdminPortal
Dashboard
Monitor
Management
Users
License
Security
Components
Nginx
RESTPP
GUI
100000
Cluster Service Status
DISMISS

Known Issues

Solution import from version lower than v3.1 is not compatible with v3.1

Since TigerGraph supports HA for the application server in v3.1, GraphStudio and Admin Portal data storage has to be updated. The HA enhancement caused a known compatibility issue during solution import process from GraphStudio: a solution that is created from "Export Current Solution" from a version released prior to v3.1 cannot be imported into v3.1. A solution exported from v3.1 will work with v3.1 and can be imported through "Import An Existing Solution". Contact support@tigergraph.com for a detailed workaround if you want to import a solution tarball prior to TG3.1.

Users with Admin role cannot Create/Edit/Delete users

Due to the difference in users' information display compared to GSQL, Admin Portal doesn't allow users with admin role to a graph to create/edit/delete users or assign a user with a role to a graph. Superuser can manage users in this case. Users with Admin role to a local graph can revoke roles to other users for this local graph.

Proxy user's roles might still exist after revoking

Admin Portal shows a combination of roles for proxy users. Proxy user which belongs to a proxy group inherits all the roles from the proxy group plus all the roles the proxy user has. Therefore, if you want to revoke a role from a proxy user, you need to revoke the role from the proxy group first. Then you might need to revoke the same role from the proxy user if the proxy user also has that role.

After upgrading to v3.1, downgrading from HTTPS to HTTP (turn on/off SSL) will result in user authentication failure

In v3.1, TigerGraph enhanced the security configuration for cookie by setting a secure flag when SSL is enabled, which may trigger an issue when using Chrome to turn off SSL. Due to this security feature, when downgrading from HTTPS to HTTP, Chrome will not allow setting the same cookie. This will result in user authentication failure because the cookie fails to be set. To solve this problem, users can clear the cache from Chrome and refresh the browser. Enabling from HTTP to HTTPS will work as expected.

Monitoring Dashboard in Admin Portal is not Continuous Available (Highly Available) in TG3.1

In v3.1, Cluster Monitoring Dashboard on Admin Portal with various metrics is not highly available, as an exception to the continuous availability feature for Admin Portal. This is because TS3 component which collects the metrics through TS3 API is not highly available.

Report bugs to us

If you find any bugs, please report them to support@tigergraph.com. ↗We really appreciate it!

Developer's Guides

Loading...

Part 1 - Data Definition & Loading

System & Language Basics

Language Basics

- **Identifiers**
  Identifiers are user-defined names. An identifier consists of letters, digits, and the underscore. Identifiers may not begin with a digit. Identifiers are case sensitive.
- **Keywords and Reserved Words**
  Keywords are words with a predefined semantic meaning in the language. Keywords are not case sensitive. Reserved words are set aside for use by the language, either now or in the future. Reserved words may not be reused as user-defined identifiers. In most cases, a keyword is also a reserved word. For example, VERTEX is a keyword. It is also a reserved word, so VERTEX may not be used as an identifier.
- **Statements**
  Each line corresponds to one statement (except in multi-line mode). Usually, there is no punctuation at the end of a top-level statement. Some statements, such as CREATE LOADING JOB, are block statements which enclose a set of statements within themselves. Some punctuation may be needed to separate the statements within a block.
- **Comments**
  Within a command file, comments are text that is ignored by the language interpreter.
  Single line comments begin with either # or //. A comment may be on the same line with interpreted code . Text to the left of the comment marker is interpreted, and text to the right of the marker is ignored.
  Multi-line comment blocks begin with /* and end with */

Running GSQL

To enter the GSQL shell and work in interactive mode, type `gsql` from an operating system shell prompt. A user name, password, and a graph name may also be provided on the command line.

GSQL command syntax for entering interactive mode

```
gsql [-u username] [-p password] [-g gname]
```

If a user name is provided but not a password, the GSQL system will then ask for the user's password:

Login example with user name

```
os$ gsql -u victor
Password for victor : ***
GSQL >
```

If a user name is not given, then GSQL will assume that you are attempting to log in as the default tigergraph user:

Login example without user name

```
os$ gsql
Password for tigergraph : *****
GSQL >
```

To exit the GSQL shell, type either `exit` or `quit` at the GSQL prompt:

`GSQL> EXIT` or `GSQL> QUIT`

GSQL Client Session Timeout

For added security, you can configure your GSQL client session to automatically timeout after a period of inactivity. Set the GSQL_CLIENT_IDLE_TIMEOUT_SEC bash environment variable=<num_sec>. Then every time you start a GSQL session, the idle timeout will be applied. To disable the timeout, omit <num,_sec>. The default setting is no timeout. Example, using the Linux export command to set the environment variable:

```
tigergraph@123:~$ export GSQL_CLIENT_IDLE_TIMEOUT_SEC=10
tigergraph@123:~$ gsql
Welcome to TigerGraph Developer Edition, free for non-production, researc
GSQL-Dev > Session timeout after 10 seconds idle.
tigergraph@123:~$ export GSQL_CLIENT_IDLE_TIMEOUT_SEC=
tigergraph@123:~$ gsql
Welcome to TigerGraph Developer Edition, free for non-production, researc
```

Multiple Shell Sessions

Multiple shell sessions of GSQL may be run at the same time. This feature can be used to have multiple clients (human or machine) using the system to perform concurrent operations. A basic locking scheme is used to maintain isolation and consistency.

Multi-line Mode - BEGIN, END, ABORT

In interactive mode, the default behavior is to treat each line as one statement; the GSQL interpreter will activate as soon as the End-Of-Line character is entered.

Multi-line mode allows the user to enter several lines of text without triggering immediate execution. This is useful when a statement is very long and the user would like to split it into multiple lines. It is also useful when defining a JOB, because jobs typically contain multiple statements.

To enter multi-line mode, use the command BEGIN. The end-of-line character is now disabled from triggering execution. The shell remains in multi-line mode until the command END is entered. The END command also triggers the execution of the multi-line block. In the example below, BEGIN and END are used to allow the SELECT statement to be split into several lines:

Example: BEGIN and END defining a multi-line block

```
BEGIN
SELECT member_id, last_name, first_name,  date_joined, status
    FROM Member
    WHERE age >= 21
    ORDER BY last_name, first_name
END
```

Alternately, the ABORT command exits multi-line mode and discards the multi-line block.

Command Files and Inline Commands

A command file is a text file containing a series of GSQL statements. Blank lines and comments are ignored. By convention, GSQL command files end with the suffix . `gsql` , but this is not a requirement. Command files are automatically treated as multi-line mode, so BEGIN and END statements are not needed. Command files may be run either from within the GSQL shell by prefixing the filename with an @ symbol:

GSQL> **@file.gsql**

or from the operating system (i.e., a Linux shell) by giving the filename as the argument after gsql:

os$ **gsql file.gsql**

Similarly, a single GSQL command can be run by enclosing the command string in quotation marks and placing it at the end of the GSQL statement. Either single or double quotation marks. It is recommended to use single quotation marks to enclose the entire command and double quotation marks to enclose any strings within the command.

Login example with inline command or command file

```
gsql [-u username] [-g graphname] ['command_string' | command_file]
```

In the example below, the file name_query.gsql contains the multi-line CREATE QUERY block to define the query namesSimilar.

Example using command files and inline commands

```
os$ gsql pagerank_query.gsql
os$ gsql 'INSTALL QUERY namesSimilar'
os$ gsql 'RUN QUERY namesSimilar (0,"michael","jackson",100)'
```

Help and Information

The `help` command displays a summary of the available GSQL commands:

GSQL> HELP [BASIC|QUERY]

Note that the HELP command has options for showing more details about certain categories of commands.

The `ls` command displays the *catalog* : all the vertex types, edge types, graphs, queries, jobs, and session parameters which have been defined by the user.

--reset option

The --reset option will clear the entire graph data store and erase all related definitions (graph schema, loading jobs, and queries) from the Dictionary. The data deletion cannot be undone; use with extreme caution. The REST++, GPE, and GSE modules will be turned off.

```
$ gsql --reset

Resetting the catalog.

Shutdown restpp gse gpe ...
Graph store /home/tigergraph/tigergraph/gstore/0/ has been cleared!
The catalog was reset and the graph store was cleared.
```

Summary

The table below summaries the basic system commands introduced so far.

Command	Description
HELP[BASIC|QUERY]	Display the help menu for all or a subset of the commands
LS	Display the catalog, which records all the vertex types, edge types, graphs, queries, jobs, and session parameters that have been defined for the current active graph. See notes below concerning graph- and role-dependent visibility of the catalog.
BEGIN	Enter multi-line edit mode (only for console mode within the shell)
END	Finish multi-line edit mode and execute the multi-line block.
ABORT	Abort multi-line edit mode and discard the multi-line block.
@file.gsql	Run the gsql statements in the command file `file.gsql` from within the GSQL shell.
`os$ gsql file.gsql`	Run the gsql statements in the command file `file.gsql` from an operating system shell.
`os$ gsql 'command_string'`	Run a single gsql statement from the operating system shell.
`os$ gsql --reset`	Clear the graph store and erase the dictionary.

Notes on the LS command

Starting with v1.2, the output of the LS command is sensitive to the user and the active graph:

1. If the user has not set an active graph or specified "USE GLOBAL":

   a. If the user is a superuser, then LS displays global vertices, global edges, and all graph schemas.
   b. If the user is not a superuser, then LS displays nothing (null).
2. If the user has set an active graph, then LS displays the schema, jobs, queries, and other definitions for that particular graph.

Session Parameters

Session parameters are built-in system variables whose values are valid during the current session; their values do not endure after the session ends. In interactive command mode, a session starts and ends when entering and exiting interactive mode, respectively. When running a command file, the session lasts during the execution of the command file.

Use the SET command to set the value of a session parameter:

```
SET session_parameter = value
```

Session Parameter	Meaning and Usage
sys.data_root	The value should be a string, representing the absolute or relative path to the folder where data files are stored. After the parameter has been set, a loading statement can reference this parameter with $sys.data_root.
gsql_src_dir	The value should be a string, representing the absolute or relative path to the root folder for the gsql system installation. After the parameter has been set, a loading statement can reference this parameter with $gsql_src_dir.
	When this parameter is true (default), if a semantic error occurs while running a GSQL command file, the GSQL shell will terminate. Accepted parameter values: true, false (case insensitive). If the parameter is set to

exit_on_error	false, then a command file which is syntactically correct will continue running, even if certain runtime errors in individual commands occur. Specifically, this affects these commands: • CREATE • INSTALL QUERY • RUN JOB Semantic errors include a reference to a nonexistent entity or an improper reuse of an entity. This session parameter does not affect GSQL interactive mode; GSQL interactive mode does not exit on any error. This session parameter does not affect syntactic errors: GSQL will always exit on a syntactic error.

Example of exit_on_error = FALSE

```
# exitOnError.gsql
SET exit_on_error = FALSE

CREATE VERTEX v(PRIMARY_ID id INT, name STRING)
CREATE VERTEX v(PRIMARY_ID id INT, weight FLOAT) #error 1: can't define V

CREATE UNDIRECTED EDGE e2 (FROM u, TO v) #error 2: vertex type u doesn't
CREATE UNDIRECTED EDGE e1 (FROM v, TO v)

CREATE GRAPH g(v) #error 3: no graph definition has no edge type
CREATE GRAPH g2(*)
```

Results

```
os$ gsql exitOnError.gsql

The vertex type v is created.
Semantic Check Fails: The vertex name v is used by another object! Please
failed to create the vertex type v
Semantic Check Fails: FROM or TO vertex type does not exist!
failed to create the edge type e2
The edge type e1 is created.
Semantic Check Fails: There is no edge type specified! Please specify at
The graph g could not be created!

Restarting gse gpe restpp ...

Finish restarting services in 11.955 seconds!
The graph g2 is created.
```

Attribute Data Types

Each attribute of a vertex or edge has an assigned data type. The following types are currently supported.

Primitive Types

Name	Default value	Valid input format (regex)	Range and precision	Description
`INT`	0	[-+]?[0-9]+	−2^63 to +2^63 - 1 (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)	8-byte signed integer
`UINT`	0	[0-9]+	0 to 2^64 - 1 (18,446,744,073,709,551,615)	8-byte unsigned integer
				4-byte single-precision floating point

`FLOAT`	0.0	[-+] ? [0 - 9] * \. ? [0 - 9] + ([eE] [-+] ? [0 - 9] +) ?	+/- 3.4 E +/-38, ~7 bits of precision	number Examples: 3.14159, .0065e14, 7E23 See note below.
`DOUBLE`	0.0	[-+] ? [0 - 9] * \. ? [0 - 9] + ([eE] [-+] ? [0 - 9] +) ?	+/- 1.7 E +/-308, ~15 bits of precision	8-byte double-precision floating point number. Has the same input and output format as FLOAT, but the range and precision are greater. See note below.
`BOOL`	false	true, false (case insensitive), 1, 0	true, false	boolean true and false, represented within GSQL as *true* and *false* , and represented in input and output as 1 and 0
`STRING`	Empty string	.*	UTF-8	character string. The string value can optionally be enclosed by single quote marks or double quote marks. Please see the QUOTE parameter in Section "Other Optional LOAD Clauses".

⚠ For FLOAT and DOUBLE values, the GSQL Loader supports exponential notation as shown (e.g., 1.25 E-7).

The GSQL Query Language currently only reads values without exponents. It may display output values with exponential notation, however.

⚠ Some numeric expressions may return a non-numeric string result, such as "inf" for Infinity or "NaN" for Not a Number.

Advanced Types

Name	Default value	Supported data format	Range and Precision	Description
`STRING COMPRESS` (⚠ **suitable only in limited circumstances**)	Empty string	.*	UTF-8	String with a finite set of categorical values. More compact storage of STRING, **if there is a limited number of different values and the value is rarely accessed. Otherwise, it may use more memory.**
`DATETIME`	UTC time 0	See Section " Loading DATETIME Attribute "	1582-10-15 00:00:00 to 9999-12-31 23:59:59	Date and time (UTC) as the number of seconds elapsed since the start of Jan 1, 1970. Time zones are not supported. Displayed in

				YYYY-MM-DD hh:mm:ss format.
`FIXED_BINARY` `(` *n* `)`	N/A		N/A	Stream of n binary-encoded bytes

Additionally, GSQL also supports the following complex data types:

Collection Types

- `LIST`: A list is an **ordered** collection of elements of the same type.
  - Default value: an empty list `[]`
  - Supported element types: `INT`, `UINT`, `DOUBLE`, `FLOAT`, `STRING`, `STRING COMPRESS`, `DATETIME`, and `UDT`
  - To declare a list type, use angle brackets `<>` to enclose the element type, e.g. `LIST<STRING>.`

> ⚠ Due to multithreaded GSQL loading, the initial order of elements loaded into a LIST might be different than the order in which they appeared in the input data.

- `SET`: A set is an **unordered** collection of **unique** elements of the same type.
  - Default value: an empty set `()`
  - Supported element types: `INT`, `UINT`, `DOUBLE`, `FLOAT`, `STRING`, `STRING COMPRESS`, `DATETIME`, and `UDT`.
  - To declare a set type, use angle brackets `<>` to enclose the element type, e.g. `SET<INT>`
- `MAP`: A map is a collection of key-value pairs. It cannot contain duplicate keys and each key maps to one value.
  - Default value: an empty map
  - Supported key types: `INT`, `STRING`, `STRING COMPRESS`, and `DATETIME`
  - Supported value types: `INT`, `DOUBLE`, `STRING`, `STRING COMPRESS`, `DATETIME`, and `UDT`.

- To declare a map type, use `<>` to enclose the types, with a comma to separate the key and value types, e.g., `MAP<INT, DOUBLE>`.

TYPEDEF TUPLE

A **User-Defined Tuple (UDT)** represents an ordered structure of several fields of the same or different types. The supported field types are listed below. Each field in a UDT has a fixed size. A `STRING` field must be given a size in characters, and the loader will only load the first given number of characters. An `INT` or `UINT` field can optionally be given a size in bytes.

TYPEDEF TUPLE syntax

```
TYPEDEF TUPLE "<" fieldName fieldType ["(" fieldSize ")"]
                ( "," fieldName fieldType ["(" fieldSize ")"] )* ">"
```

Field type	User-specified size	Size	Range (N is size)
`INT`	Optional	1, 2, 4 (default), or 8 bytes	0 to 2^(N*8) - 1
`UINT`	Optional	1, 2, 4 (default), or 8 bytes	-2^(N*8 - 1) to 2^(N*8 - 1) - 1
`FLOAT`	No	4 bytes	-3.4 E-38 to 3.4 E38
`DOUBLE`	No	8 bytes	-1.7 E-308 to 1.7 E308
`DATETIME`	No		1582-10-15 00:00:00 to 9999-12-31 23:59:59
`BOOL`	No		`true` or `false`
`STRING`	Required	Any number of characters	Any string under N characters

A UDT must be defined before being used as a field in a vertex type or edge type. To define a UDT, use the `TYPEDEF TUPLE` statement. Below is an example of a `TYPEDEF TUPLE` statement:

Example: UDT Definition

```
TYPEDEF TUPLE <field1 INT(1), field2 UINT, field3 STRING(10), field4
```

In the above example, `myTuple` is the name of the UDT. It contains four fields: an 1-byte `INT` field named `field1`, a 4-byte `UINT` field named `field2`, a 10-character `STRING` field named `field3`, and an (8-byte) `DOUBLE` field named `field4`.

Loading...

Modifying a Graph Schema

After a graph schema has been created , it can be modified. Data already stored in the graph and which is not logically part of the change will be retained. For example, if you had 100 Book vertices and then added an attribute to the Book schema, you would still have 100 Books, with default values for the new attribute. If you dropped a Book attribute, you still would have all your books, but one attribute would be gone.

To safely update the graph schema, the user should follow this procedure:

- Create a SCHEMA_CHANGE JOB, which defines a sequence of ADD, ALTER and/or DROP statements.
- Run the SCHEMA_CHANGE JOB (i.e. `RUN SCHEMA_CHANGE JOB job_name`), which will do the following:
  - Attempt the schema change.
  - If the change is successful, invalidate any loading job or query definitions which are incompatible with the new schema.
  - if the change is unsuccessful, report the failure and return to the state before the attempt.

A schema change will invalidate any loading jobs or query jobs which relate to an altered part of the schema. Specifically:

- A loading job becomes invalid if it refers to a vertex or and an edge which has been **dropped** (deleted) or **altered** .
- A query becomes invalid if it refers to a vertex, and edge, or an attribute which has been **dropped** .

Invalid loading jobs are dropped, and invalid queries are uninstalled. After the schema update, the user will need to create and install new load and query jobs based on the new schema.

Jobs and queries for unaltered parts of the schema will still be available and do not need to be reinstalled. However, even though these jobs are valid (e.g., they **can** be run), the user may wish to examine whether they still perform the preferred operations (e.g., do you **want** to run them?)

Load or query operations which begin before the schema change will be completed based on the pre-change schema. Load or query operations which begin after the schema change, and which have not been invalidated, will be completed based on the post-change schema.

Global vs. Local Schema Changes

Only a superuser or globaldesigner can add, alter, or drop global vertex types or global edge types, which are those that are created using CREATE VERTEX or CREATE ... EDGE. This rule applies even if the vertex or edge type is used in only one graph. To make these changes, the user uses a GLOBAL SCHEMA_CHANGE JOB.

An admin or designer user can add, alter, or drop local vertex types or local edge types which are created in the context of that graph. Local vertex and edge types are created using an ADD statement inside a SCHEMA_CHANGE JOB. To alter or drop any of these local types, the admin user uses a regular SCHEMA_CHANGE JOB.

Local graphs can define vertex and edge types independently of the vertex and edge types in other graph. That is, the same name can be used in different graphs for (different) vertex or edge types.

It is even permitted for a local graph and the global graph to use the same name for their own vertex or edge types, as long as the global vertex/edge type is not used within the local graph.

The two types of schema change jobs are described below.

`CREATE SCHEMA_CHANGE JOB` (local)

The CREATE SCHEMA_CHANGE JOB block defines a sequence of ADD, ALTER, and DROP statements for changing a particular graph. It does not perform the schema change.

CREATE SCHEMA_CHANGE JOB syntax

```
CREATE SCHEMA_CHANGE JOB job_name FOR GRAPH graph_name {
    [sequence of DROP, ALTER, and ADD statements, each line ending with a
}
```

One use of CREATE SCHEMA_CHANGE JOB is to define an additional vertex type and edge type to be the structure for a secondary index. For example, if you wanted to index the postalCode attribute of the User vertex, you could create a postalCode_idx (PRIMARY_ID id string, code string) vertex type and hasPostalCode (FROM User, TO postalCode_idx) edge type. Then create an index structure having one edge from each User to a postalCode_idx vertex.

By its nature, a SCHEMA_CHANGE JOB may contain multiple statements. If the job block is used in the interactive GSQL shell, then the BEGIN and END commands should be used to permit the SCHEMA_CHANGE JOB to be entered on several lines. if the job is stored in a command file to be read in batch mode, then BEGIN and END are not needed.

Remember to include a semicolon at the end of each DROP, ALTER, or ADD statement within the JOB block.

If a SCHEMA_CHANGE JOB defines a new edge type which connects to a new vertex type, the ADD VERTEX statement should precede the related ADD EDGE statement. However, the ADD EDGE and ADD VERTEX statements can be in the same SCHEMA_CHANGE JOB.

`ADD VERTEX | EDGE` (local)

The ADD statement defines a new type of vertex or edge and automatically adds it to a graph schema. The syntax for the ADD VERTEX | EDGE statement is analogous to that of the CREATE VERTEX | EDGE | GRAPH statements. It may only be used within a SCHEMA_CHANGE JOB.

ADD VERTEX / UNDIRECTED EDGE / DIRECTED EDGE

```
ADD VERTEX v_type_name (PRIMARY_ID id type [',' attribute_list]) [WITH ST
ADD UNDIRECTED EDGE e_type_name (FROM v_type_name',' TO v_type_name [','
ADD DIRECTED EDGE e_type_name (FROM v_type_name',' TO v_type_name [',' ed
    [WITH REVERSE_EDGE '=' "rev_name"];
```

`ALTER VERTEX | EDGE`

The `ALTER` statement adds attributes to or removes attributes from an existing vertex type or edge type. It may only be used within a `SCHEMA_CHANGE JOB`. The basic format is as follows:

ALTER VERTEX / EDGE

```
ALTER VERTEX|EDGE object_type_name ADD|DROP (attribute_list);
```

`ALTER ... ADD`

`ALTER ... ADD` can add attributes to vertex or edge types.

Added attributes are appended to the end of the schema. The new attributes may include DEFAULT fields. To add attributes to a vertex type, the syntax is as follows:

ALTER VERTEX ... ADD

```
ALTER VERTEX vertex_type_name ADD
    ATTRIBUTE (attribute_name type [DEFAULT default_value]
    [',' attribute_name type [DEFAULT default_value]]* );
```

For example:

```
ALTER VERTEX Company ADD ATTRIBUTE (industry
STRING, marketcap DOUBLE)
```

To add to an edge's endpoint vertex types or attributes, the syntax is as follows:

ALTER EDGE... ADD

```
ALTER EDGE edge_type_name ADD
    [ATTRIBUTE (attribute_name type [DEFAULT default_value]
    [',' attribute_name type [DEFAULT default_value]]* )];
```

`ALTER EDGE .. ADD PAIR`

`ALTER EDGE ... ADD PAIR` adds one or more edge pairs, which refer to the `FROM` and `TO` vertex types of an edge type. To add an edge pair, put the vertex type names in parentheses after keywords `FROM` and `TO`.

Syntax

```
ALTER EDGE edgeType ADD PAIR
"(" FROM vertexType, TO vertexType (| FROM vertexType, TO vertexType)* ")
```

Example

In the example below, the first statement in the schema change job will add an edge pair (`FROM person, TO company`) to the edge type `visit`. The second example adds two edge pairs to the edge type `has_pet`; the edge type can now connect both `person` and `dog` vertices, as well as `person` and `bird` vertices.

```
CREATE SCHEMA_CHANGE JOB job2 FOR GRAPH example_graph {
  ALTER EDGE visit ADD PAIR (FROM person, TO company);
  ALTER EDGE has_pet ADD PAIR (FROM person, TO dog | FROM person, TO bird
}
```

`ALTER ... DROP`

The syntax for ALTER ... DROP is analogous to that of ALTER ... ADD.

ALTER ... DROP

```
ALTER VERTEX|EDGE object_type_name DROP ATTRIBUTE (
      attribute_name [',' attribute_name]* );
```

`ALTER VERTEX ... WITH` (Beta)

ALTER VERTEX WITH TAGGABLE

```
ALTER VERTEX v_type_name WITH TAGGABLE = ("true" | "false")
```

`DROP VERTEX | EDGE` (local)

The DROP statement removes the specified vertex type or edge type from the database dictionary. The DROP statement should only be used when graph operations are not in progress.

drop vertex / edge

```
DROP VERTEX v_type_name [',' v_type_name]*
DROP EDGE e_type_name [',' e_type_name]*
```

`DROP TUPLE`

For tuples that are defined within a graph schema, you can drop them by using the following command.

drop tuple

```
DROP TUPLE tuple_name [',' tuple_name]*
```

`ADD TAG`

`ADD TAG` defines a tag for the graph. Tags can be used to create tag-based graphs, allowing for finer grain access control.

Syntax for ADD TAG

```
ADD TAG <tag_name> [DESCRIPTION <tag_description>]
```

`DROP TAG`

`DROP TAG` drops a tag or multiple tags from the schema, and deletes the tag from each vertex to which it is attached. `DROP TAG` cannot be run if the tag to be dropped is used in the definition of a tag-based graph; the graph must be dropped first.

Syntax for DROP TAG

```
DROP TAG <tag_name> ["," <tag_name>]*
```

`RUN SCHEMA_CHANGE JOB`

RUN SCHEMA_CHANGE JOB job_name performs the schema change job. After the schema has been changed, the GSQL system checks all existing GSQL queries. If an existing GSQL query uses a dropped vertex, edge, or attribute, the query becomes invalid, and GSQL will show the message "Query *query_name* becomes invalid after schema update, please update it.".

Below is an example. The schema change job add_reviews adds a Review vertex type and two edge types to connect reviews to users and books, respectively.

SCHEMA_CHANGE JOB example

```
USE GRAPH Book_rating
CREATE SCHEMA_CHANGE JOB add_reviews FOR GRAPH Book_rating {
    ADD VERTEX Review (PRIMARY_ID id UINT, review_date DATETIME, url STRI
    ADD UNDIRECTED EDGE wrote_review (FROM User, TO Review);
    ADD UNDIRECTED EDGE review_of_book (FROM Review, TO Book);
}
RUN SCHEMA_CHANGE JOB add_reviews
```

`DROP SCHEMA_CHANGE JOB`

To drop (remove) a schema change job, run `DROP JOB schema_change_job` name from the GSQL shell. The specific schema change job will be removed from GSQL. Refer to the Creating a Loading Job page for more information about dropping jobs.

```
GSQL > USE GRAPH Book_rating
GSQL > DROP JOB local_schema_change123
The job local_schema_change_change123 is dropped!
```

USE GLOBAL

The USE GLOBAL command changes a superuser's mode to Global mode. In global mode, a superuser can define or modify global vertex and edge types, as well as specifying which graphs use those global types. For example, the user should run USE GLOBAL before creating or running a GLOBAL SCHEMA_CHANGE JOB.

CREATE GLOBAL SCHEMA_CHANGE JOB

The `CREATE GLOBAL SCHEMA_CHANGE JOB` block defines a sequence of `ADD`, `ALTER`, and `DROP` statements that modify either the attributes or the graph membership of global vertex or edge types. Unlike the non-global schema change job, the header does not include a graph name. However, the `ADD`/`ALTER`/`DROP` statements in the body do mention graphs.

CREATE GLOBAL SCHEMA_CHANGE JOB syntax

```
CREATE GLOBAL SCHEMA_CHANGE JOB job_name {
    [sequence of global DROP, ALTER, and ADD statements, each line ending
}
```

Although both global and local schema change jobs have `ADD` and `DROP` statements, they have different meanings. The table below outlines the differences.

	Local	Global
`ADD`	Defines a new local vertex/edge type; adds it to the graph's domain	Adds one or more existing global vertex/edge types to a graph's domain.

`DROP`	Deletes a local vertex/edge type and its vertex/edge instances	Removes one or more existing global vertex/edge types from a graph's domain.
`ALTER`	Adds or drops attributes from a local	Adds or drops attributes from a global vertex/edge

Remember to include a semicolon at the end of each `DROP`, `ALTER`, or `ADD` statement within the JOB block.

`ADD VERTEX | EDGE` (global)

The ADD statement adds existing global vertex or edge types to one of the graphs.

ADD VERTEX / UNDIRECTED EDGE / DIRECTED EDGE (Global)

```
ADD VERTEX v_type_name [',' v_type_name...] TO GRAPH gname;
ADD EDGE e_type_name [',' e_type_name...] TO GRAPH gname;
```

`ALTER VERTEX | EDGE`

The `ALTER` statement is used to add attributes to or remove attributes from an existing global vertex type or edge type. The `ALTER VERTEX / EDGE` syntax for global schema changes is the same as that for local schema change jobs.

The ALTER statement is used to add attributes to or remove attributes from an existing vertex type or edge type. It can also be used to add or remove source (FROM) vertex types or destination (TO) vertex types of an edge type. It may only be used within a SCHEMA_CHANGE JOB. The basic format is as follows:

ALTER VERTEX / EDGE

```
ALTER VERTEX|EDGE object_type_name ADD|DROP (attribute_list);
```

`ALTER ... ADD`

Added attributes are appended to the end of the schema. The new attributes may include DEFAULT fields. To add attributes to a vertex type, the syntax is as follows:

ALTER VERTEX ... ADD

```
ALTER VERTEX vertex_type_name ADD
    ATTRIBUTE (attribute_name type [DEFAULT default_value]
    [',' attribute_name type [DEFAULT default_value]]* );
```

For example:

```
ALTER VERTEX Company ADD ATTRIBUTE (industry
STRING, marketcap DOUBLE)
```

To add to an edge's endpoint vertex types or attributes, the syntax is as follows:

ALTER EDGE... ADD

```
ALTER EDGE edge_type_name ADD
    [FROM (vertex_type_name [','vertex_type_name])]
    [TO (vertex_type_name [','vertex_type_name])]
    [ATTRIBUTE (attribute_name type [DEFAULT default_value]
    [',' attribute_name type [DEFAULT default_value]]* )];
```

For example:

```
ALTER EDGE Like ADD TO (Animal) ATTRIBUTE (suggested_by STRING)
```

`ALTER EDGE .. ADD PAIR`

`ALTER EDGE ... ADD PAIR` adds one or more edge pairs, which refer to the `FROM` and `TO` vertex types of an edge type. To add an edge pair, put the vertex type names in parentheses after keywords `FROM` and `TO`.

Syntax

```
ALTER EDGE edgeType ADD PAIR
"(" FROM vertexType, TO vertexType (| FROM vertexType, TO vertexType)* ")
```

Example

In the example below, the first statement in the schema change job will add an edge pair (`FROM person, TO company`) to the edge type `visit`. The second example adds two edge pairs to the edge type `has_pet`; the edge type can now connect both `person` and `dog` vertices, as well as `person` and `bird` vertices.

```
CREATE GLOBAL SCHEMA_CHANGE JOB job2 FOR GRAPH example_graph {
  ALTER EDGE visit ADD PAIR (FROM person, TO company);
  ALTER EDGE has_pet ADD PAIR (FROM person, TO dog | FROM person, TO bird
}
```

`ALTER ... DROP`

The syntax for `ALTER ... DROP` is analogous to that of `ALTER ... ADD`.

ALTER ... DROP

```
ALTER VERTEX|EDGE object_type_name DROP ATTRIBUTE (
      attribute_name [',' attribute_name]* );
```

`ALTER VERTEX ... WITH` (Beta)

ALTER VERTEX WITH TAGGABLE

```
ALTER VERTEX v_type_name WITH TAGGABLE = ("true" | "false")
```

`DROP VERTEX | EDGE` (global)

The DROP statement removes specified global vertex or edge types from one of the graphs. The command does not delete any data.

drop vertex / edge

```
DROP VERTEX v_type_name [',' v_type_name...] FROM GRAPH gname;
DROP EDGE e_type_name   [',' e_type_name...] FROM GRAPH gname;
```

RUN GLOBAL SCHEMA_CHANGE JOB

RUN GLOBAL SCHEMA_CHANGE JOB job_name performs the global schema change job. After the schema has been changed, the GSQL system checks all existing GSQL queries. If an existing GSQL query uses a dropped vertex, edge, or attribute, the query becomes invalid, and GSQL will show the message "Query *query_name* becomes invalid after schema update, please update it.".

Below is an example. The schema change alter_friendship_make_library drops the on_date attribute from the friend_of edge and adds Book type to the library graph.

GLOBAL SCHEMA_CHANGE JOB example

```
USE GLOBAL
CREATE GRAPH library()
CREATE GLOBAL SCHEMA_CHANGE JOB alter_friendship_make_library {
    ALTER EDGE friend_of DROP ATTRIBUTE (on_date);
    ADD VERTEX Book TO GRAPH library;
}
RUN GLOBAL SCHEMA_CHANGE JOB alter_friendship_make_library
```

DROP GLOBAL SCHEMA_CHANGE JOB

Global schema change jobs can be dropped by using the DROP JOB command. Refer to the Creating a Loading Job page for more information about dropping jobs.

DROP GLOBAL SCHEMA_CHANGE JOB example

```
USE GLOBAL
DROP JOB alter_friendship_make_library
```

DROP ALL

The DROP ALL command clears all graph data, all graph schemas, all loading jobs, and all queries. It should only be used when the intent is to erase an entire database design and to start over.

This command is only available to superusers and only when they are in global mode.

Creating a Loading Job

After a graph schema has been created, the system is ready to load data into the graph store. The GSQL language offers easy-to-understand and easy-to-use commands for data loading which perform many of the same data conversion, mapping, filtering, and merging operations that are found in enterprise ETL (Extract, Transform, and Load) systems.

The GSQL system can read structured or semistructured data from text files. The loading language syntax is geared towards tabular or JSON data, but conditional clauses and data manipulation functions allow for reading data that is structured in a more complex or irregular way. For tabular data, each line in the data file contains a series of data values, separated by commas, tabs, spaces, or any other designated ASCII characters (only single character separators are supported). A line should contain only data values and separators, without extra whitespace. From a tabular view, each line of data is a row, and each row consists of a series of column values.

Loading data is a two-step process. First, a loading job is defined. Next, the job is executed with a `RUN LOADING JOB` statement. These two statements, and the components of the loading job, are detailed below.

The structure of a loading job will be presented hierarchically, top-down:

CREATE LOADING JOB, which may contain a set of DEFINE and LOAD statements

- DEFINE statements
- LOAD statements, which can have several clauses

> **All blank spaces are meaningful in string fields in CSV and JSON**. Either pre-process your data files to remove extra spaces, or use GSQL's token processing functions `gsql_trim`, `gsql_ltrim`, and `gsql_rtrim` (Built-in Loader Token Functions).

Loading job capabilities

TigerGraph's syntax for defining and running loading jobs offers several advantages:

- The TigerGraph platform can handle concurrent loading jobs, which can greatly increase throughput.
- The data file locations can be specified at compile time or at run time. Run-time settings override compile-time settings.
- A loading job definition can include several input files. When running the job, the user can choose to run only part of the job by specifying only some of the input files.
- Loading jobs can be monitored, aborted, and restarted.

Concurrent Loading

Among its several duties, the RESTPP component manages loading jobs. There can be multiple RESTPP-LOADER subcomponents, each of which can handle a loading job independently. The maximum number of concurrent loading jobs is set by the configuration parameter `FileLoader.ReplicaNumber`.

Furthermore, if the TigerGraph graph is distributed (partitioned) across multiple machine nodes, each machine's RESTPP-LOADER(s) can be put into action. Each RESTPP-LOADER only reads local input data files, but the resulting graph data can be stored on any machine in the cluster.

> To maximize loading performance in a cluster, use at least two loaders per machine, and assign each loader approximately the same amount of data.

A concurrent-capable loading job can logically be separated into parts according to each file variable. When a concurrent-capable loading job is compiled, a [RESTPP endpoint] is generated for the loading job, which you can call to load data into your graph as an alternative to `RUN LOADING JOB`.

> Example loading jobs and data files for the `book_rating` schema defined earlier in the document are available in the `$(gadmin config get System.AppRoot)/document/examples` folder in your TigerGraph platform installation.

CREATE LOADING JOB

The CREATE LOADING JOB statement is used to define a block of DEFINE, LOAD, and DELETE statements for loading data to or removing data from a particular graph. The sequence of statements is enclosed in curly braces. Each statement in the block, including the last one, should end with a semicolon.

CREATE LOADING JOB syntax

```
CREATE LOADING JOB job_name FOR GRAPH graph_name {
   [zero or more DEFINE statements;]
   [zero or more LOAD statements;] | [zero or more DELETE statements;
}
```

LOAD or DELETE Statements
****A loading job may contain either `LOAD` or `DELETE` statements but not both. A loading job that includes both will be rejected when the `CREATE` statement is executed.

Example:

Loading Job Example

```
CREATE LOADING JOB job1 FOR GRAPH graph1 {
   # File path specified at complile time
   DEFINE FILENAME file1 = "/data/v1.csv";
   DEFINE FILENAME file2;

   LOAD file1 TO VERTEX v1 VALUES ($0, $1, $2);
   LOAD file2 TO EDGE e2 VALUES ($0, $1);
}

# Run-time specification will override path specified at compile time
RUN LOADING JOB job1 USING file1="m1:/data/v1_1.csv", file2="m2:/data
```

DROP JOB statement

To drop (remove) a job, run `DROP JOB job_name`. The job will be removed from GSQL. To drop all jobs, run either of the following commands:

```
DROP JOB ALL
DROP JOB *
```

The scope of `ALL` depends on the user's current scope. If the user has set a working graph, then `DROP ALL` removes all the jobs for that graph. If a superuser has set their scope to be global, then `DROP ALL` removes all jobs across all graph spaces.

`DEFINE` statements

A `DEFINE` statement is used to define a local variable or expression to be used by the subsequent `LOAD` statements in the loading job.

`DEFINE FILENAME`

The `DEFINE FILENAME` statement defines a filename variable. The variable can then be used later in the `JOB` block by a `LOAD` statement to identify its data source. Every concurrent loading job must have at least one `DEFINE FILENAME` statement.

```
DEFINE FILENAME filevar ["=" filepath_string ];
filepath_string = ( path | " all :" path | " any :" path | mach_aliases "
mach_aliases = name["|"name]*
```

The `filevar`_ _is optionally followed by a `filepath_string`, which tells the job where to find input data. As the name suggests, _filepath_string _is a string value. Therefore, it should start and end with double quotes.

filepath_string

There are four options for _filepath_string _:

- `path` _: _either an absolute path or relative path for either a file or a folder on the machine where the job is run. If it is a folder, then the loader will attempt to load each non-hidden file in the folder.

path examples

```
"/data/graph.csv"
```

> ⚠ If this path is not valid when CREATE LOADING JOB is executed, GSQL will report an error.

An absolute path may begin with the session variable $sys.data_root.

Example: using sys.data_root in a loading job

```
CREATE LOADING JOB filePathEx FOR GRAPH gsql_demo {
  LOAD "$sys.data_root/persons.csv" TO ...
}
```

Then, when running this loading job, first set a value for the parameter, and then run the job:

Example: Setting sys.data_root session parameter

```
SET sys.data_root="/data/mydata"
RUN JOB filePathEx
```

As the name implies, session parameters only retain their value for the duration of the current GSQL session. If the user exits GSQL, the settings are lost.

- `"all:" path`: If the path is prefixed with `all:`, then the loading job will attempt to run on every machine in the cluster which has a RESTPP component, and each machine will look locally for data at *path*. **If the path is invalid on any of the machines, the job will be aborted**. Also, the session parameter $sys.data_root may not be used.

ALL:path examples

```
"ALL:/data/graph.csv"
```

- `"any:" path`: If the path is prefixed with `any:`, then the loading job will attempt to run on every machine in the cluster which has a RESTPP component, and each machine will look locally for data at the specified path. **If the path is invalid on any of the machines, those machines where the path is not valid are skipped. **Also, the session parameter $sys.data_root may not be used.

ANY:path examples

```
"ANY:/data/graph.csv"
```

- **A list of machine-specific paths**: A machine_alias is a name such as m1, m2, etc. which is defined when the cluster configuration is set. For this option, the _filepath_string _may include a list of paths, separated by commas. If several machines have the same path, the paths can be grouped together by using a list of machine aliases, with the vertical bar "|" as a separator. The loading job will run on whichever machines are named; each RESTPP-LOADER will work on its local files.

machine-specific path example

```
"m1:/data1.csv, m2|m3|m5:/data/data2.csv"
```

DEFINE HEADER

The DEFINE HEADER statement defines a sequence of column names for an input data file. The first column name maps to the first column, the second column name maps to the second column, etc.

```
DEFINE HEADER header_name = " column_name "[," column_name "]*;
```

DEFINE INPUT_LINE_FILTER

The DEFINE INPUT_LINE_FILTER statement defines a named Boolean expression whose value depends on column attributes from a row of input data. When combined with a USING reject_line_rule clause in a LOAD statement, the filter determines whether an input line is ignored or not.

```
DEFINE INPUT_LINE_FILTER filter_name = boolean_expression_using_column_va
```

`LOAD` statement

A `LOAD` statement describes how to parse a data line into column values (tokens), and then describes how the values should be used to create a new vertex or edge instance. One `LOAD` statement can be used to generate multiple vertices or edges, each vertex or edge having its own *destination clause*, as shown below. Additionally, two or more LOAD statements may refer to the same input data file. In this case, the GSQL loader will merge their operations so that both of their operations are executed in a single pass through the data file.

The `LOAD` statement has many options. This reference guide provides examples of key features and options. The Knowledge Base and FAQs_** **_and the tutorials, such as GSQL 101, provide additional solution- and application-oriented examples.

Different `LOAD` statement types have different rules for the `USING` clause; see the `USING` clause section below for specifics.

Syntax

```
LOAD [filepath_string|filevar|TEMP_TABLE table_name ]
     Destination_Clause [, Destination_Clause ]*
     [TAGS clause ]
     [USING clause ];
```

`filevar` _ _must have been previously defined in a DEFINE FILENAME statement.

`filepath_string` _ _must satisfy the same rules given above in the DEFINE FILENAME section.

Destination clause

A *Destination clause*** **describes how the tokens from a data source should be used to construct one of three types of **data objects**: a vertex, an edge, or a row in a temporary table (`TEMP_TABLE`). The destination clause formats for the three types are very similar, but we show them separately for clarity:

Vertex Destination Clause

```
TO VERTEX vertex_type_name VALUES (id_expr [, attr_expr]*)
    [WHERE conditions] [OPTION (options)]
```

Edge Destination Clause

```
TO EDGE edge_type_name VALUES (source_id_expr [source_type_expr],
                               target_id_expr [target_type_expr]
                               [, attr_expr]*)
    [WHERE conditions] [OPTION (options)]
```

For the `TO VERTEX` and `TO EDGE` destination clauses, the following rules for its parameters apply:

- The _ `vertex_type_name` _or _ `edge_type_name` _must match the name of a vertex or edge type previously defined in a `CREATE VERTEX` or `CREATE UNDIRECTED|DIRECTED EDGE` statement.

- The values in the value list_(`id_expr`, `attr_expr1`, `attr_expr2`,...) _are assigned to the id(s) and attributes of a new vertex or edge instance, in the same order in which they are listed in the `CREATE` statement.
- _ `id_expr` _obeys the same attribute rules as *`attr_expr`*, except that _only `attr_expr` _can use the reducer function, which is introduced later.
- For edge clauses, the *`source_id_expr`* and *`target_id_expr`* can each optionally be followed by a *`source_type_expr`* and *`target_type_expr`*, respectively. The *`source_type_expr`* and *`target_type_expr`* must evaluate to one of the allowed endpoint vertex types for the given edge type. By specifying the vertex type, this tells the loader what id types to expect. This may be important when the edge type is defined to accept more than one type of source/target vertex.

TEMP_TABLE Destination Clause

```
TO TEMP_TABLE table_name (id_name [, attr_name]*)
    VALUES (id_expr [, attr_expr]*)
    [WHERE conditions] [OPTION (options)]
```

The `TO TEMP_TABLE` clause defines a new, temporary data structure. Its unique characteristics will be described in a separate subsection. For now, we focus on `TO VERTEX` and `TO EDGE`.

For fast loading of edge data, referential integrity checking is disabled by default. For an edge to be valid, it must refer to endpoint vertices that exist. To support fast, out-of-order loading,** if one or both of the endpoint vertices do not yet exist, the loader will create vertices with the necessary IDs and default attribute values.** Due to the loader's UPSERT semantics, if the vertex data is loaded later, it will be automatically merged with the dummy vertices. The user can disable this feature and perform regular referential integrity checking by setting the `VERTEX_MUST_EXIST=true` option.

Examples

Suppose we have the following vertex and edge types:

```
CREATE VERTEX Person (pid STRING PRIMARY KEY, birthdate DATETIME)
CREATE VERTEX Company (cid INT PRIMARY KEY, industry STRING)
CREATE DIRECTED EDGE Visit (FROM Person, TO Person
                            | FROM Person, TO Company, year INT)
```

A `Visit` edge can connect two `Person` vertices or a `Person` to a `Company`. A `Person` has a string ID, while a Company has an `INT` ID. Then suppose the `Visit` edge source data comes from a single CSV file, containing both variants of edges. Note that the 2nd column ($1) contains either `Person` or `Company`, and that the 3rd column ($2) contains either a string or an integer.

```
Sam,Person,Joe,2012
Sam,Company,4057,2017
Chris,Company,9401,2016
Pat,Person,Taylor,2020
```

Using the optional `target_type_expr` field, we can load both variants of the `Visit` edge with a single clause.

```
LOAD file1 TO EDGE Visit VALUES ($0, $2 $1, $3) USING separator=",";
```

Known issue: you must include a `USING clause` when loading data into edge types with different `FROM-TO` vertex pairs, even if all options are default.

Attributes and Attribute Expressions

A `LOAD` statement processes each line of an input file, splitting each line (according to the `SEPARATOR` character, see Section "Other Optional `LOAD` Clauses" for more details) into a sequence of tokens. Each destination clause provides a token-to-attribute mapping which defines how to construct a new vertex, an edge, or a temp table row instance (e.g., one data object). The tokens can also be thought of as the column values in a table. There are two ways to refer to a column, by position or by name. Assuming a column has a name, either method may be used, and both methods may be used within one expression.

By Position: The columns (tokens) are numbered from left to right, starting with $0. The next column is $1, and so on.

By Name: Columns can be named, either through a header line in the input file, or through a `DEFINE HEADER` statement. If a header line is used, then the first line of the input file should be structured like a data line, using the same separator characters, except that each column contains a column name string instead of a data value. Names are enclosed in double quotes, e.g. `$"age"`.

**Data file name: **$sys.file_name refers to the current input data file.

In a simple case, a token value is copied directly to an attribute. For example, in the following `LOAD` statement,

Example: using $sys.file_name in an attribute expression

```
LOAD "xx/yy/a.csv" TO VERTEX person VALUES ($0, $1, $sys.file_name)
```

- The `PRIMARY_ID` of a person vertex comes from column `$0` of the file "xx/yy/a.csv".
- The next attribute of a person vertex comes from column `$1`.
- The next attribute of a person vertex is given the value "xx/y/a.csv" (the filename itself).

ⓘ Users do not need to explicitly define a primary ID. Given the attributes, one will be selected as the primary key.

Cumulative Loading

A basic principle in the GSQL Loader is cumulative loading. Cumulative loading means that a particular data object might be written to (i.e., loaded) multiple times, and the result of the multiple loads may depend on the full sequence of writes. This usually means that If a data line provides a valid data object, and the `WHERE` clause and `OPTION` clause are satisfied, then the data object is loaded.

- **Valid input**: For each input data line, each destination clause constructs one or more new data objects. To be a **valid data object, **it must have an ID value of the correct type, have correctly typed attribute values, and satisfy the optional `WHERE` clause. If the data object is not valid, the object is rejected (skipped) and counted as an error in the log file. The rules for invalid attributes values are summarized below:

1. `UINT`: Any non-digit character. (Out-of-range values cause overflow instead of rejection)
2. `INT`: Any non-digit or non-sign character. (Out-of-range values cause overflow instead of rejection)
3. `FLOAT` and `DOUBLE`: Any wrong format
4. `STRING`, `STRING COMPRESS`, `FIXED_BINARY`: N/A
5. `DATETIME`: Wrong format, invalid date time, or out of range.
6. `BOOL`: Any value not listed later.
7. Complex type: Depends on the field type or element type. Any invalid field (in `UDT`), element (in `LIST` or `SET`), key or value (in `MAP`) causes rejection.

- **New data objects: **If a valid data object has a new ID value, then the data object is added to the graph store. Any attributes which are missing are assigned the default value for that data type or for that attribute.
- **Overwriting existing data objects**: If a valid data object has a ID value for an existing object, then the new object overwrites the existing data object, with the following clarifications and exceptions:

1. The attribute values of the new object overwrite the attribute values of the existing data object.
2. **Missing tokens**: If a token is missing from the input line so that the generated attribute is missing, then that attribute retains its previous value.

> ⚠ A STRING token is never considered missing; if there are no characters, then the string is the empty string

- **Skipping an attribute**: A `LOAD` statement can specify that a particular attribute should **not** be loaded by using the special character _ (underscore) as its attribute expression (attr_expr). For example,

```
LOAD TO VERTEX person VALUES ($0, $1, _, $2)
```

means to skip the next-to-last attribute. This technique is used when it is known that the input data file does not contain data for every attribute.

1. If the load operation is creating a new vertex or edge, then the skipped attribute will be assigned the default value.
2. If the load operation is overwriting an existing vertex or edge, then the skipped attribute will retain its existing value.

More Complex Attribute Expressions

An attribute expression may use column tokens (e.g., `$0`), literals (constant numeric or string values), any of the built-in loader token functions, or a user-defined token function. Attribute expressions may **not **contain mathematical or boolean operators (such as `+`, `*`, `AND`). The rules for attribute expressions are the same as those for id expressions, but an attribute expression can additionally use a reducer function:

- _id_expr _:= $column_number | $"column_name" | constant | $sys.file_name | token_function_name(_id_expr _[, _id_expr _]*)
- _attr_expr _:= _id_expr _| REDUCE(reducer_function_name(id __expr _))

Note that token functions can be nested, that is, a token function can be used as an input parameter for another token function. The built-in loader token/reducer functions and user-defined token functions are described in the section "Built-In Loader Token Functions".

The subsections below describe details about loading particular data types.

Loading a DOUBLE or FLOAT Attribute

A floating point value has the basic format

```
[sign][digits].[digits](e|E)[sign][digits]
```

In the first case, the decimal point and following digits are required. In the second case, some digits are required (looking like an integer), and the following decimal point and digits are optional.

In both cases, the leading sign ("+" or "-") is optional. The exponent, using "e" or "E", is optional. Commas and extra spaces are not allowed.

Examples of valid and invalid floating point values

```
# Valid floating point values
-198256.03
+16.
-.00036
7.14285e15
9.99E-22


# Invalid floating point values
-198,256.03
9.99 E-22
```

Loading a `DATETIME` Attribute

When loading data into a `DATETIME` attribute, the GSQL loader will automatically read a string representation of DateTime information and convert it to internal DateTime representation. The loader accepts any of the following string formats:

- `%Y-%m-%d %H:%M:%S` (e.g., 2011-02-03 01:02:03)
- `%Y/%m/%d %H:%M:%S` (e.g., 2011/02/03 01:02:03)
- `%Y-%m-%dT%H:%M:%S.000z` (e.g., 2011-02-03T01:02:03.123z, 123 will be ignored)
- `%Y-%m-%d` (only date, no time, e.g., 2011-02-03)
- `%Y/%m/%d` (only date, no time, e.g., 2011/02/03)
- Any integer value (Unix Epoch time, where Jan 1, 1970 at 00:00:00 is integer 0)

Format notation:

- `%Y` is a 4-digit year. A 2-digit year is not a valid value.
- `%m` and `%d` represent month (1 to 12) and day (1 to 31), respectively. Leading zeroes are optional.
- `%H`, `%M`, `%S` are hours (0 to 23), minutes (0 to 59) and seconds (0 to 59), respectively. Leading zeroes are optional.

When loading data, the loader checks whether the values of year, month, day, hour, minute, second are out of the valid range. If any invalid value is present, e.g. '2010-13-05' or '2004-04-31 00:00:00', the attribute is invalid and the object (vertex or edge) is not created.

Loading a BOOL Attribute

When loading data from CSV files the following values are accepted for BOOL attributes :

- True: `TRUE`, `True`, `true`, `1`
- False: `FALSE`, `False`, `false`, `0`

When loading data from JSON documents, the valid BOOL values are `true` and `false`.

Loading a User-Defined Type (UDT) Attribute

To load a UDT attribute, state the name of the UDT type, followed by the list of attribute expressions for the UDT's fields, in parentheses. See the example below.

Load UDT example

```
TYPEDEF TUPLE <f1 INT (1), f2 UINT, f3 STRING (10), f4 DOUBLE > myTup
CREATE VERTEX v_udt  (id STRING PRIMARY KEY, att_udt myTuple)
CREATE GRAPH test_graph (v_udt)
CREATE LOADING JOB load_udt FOR GRAPH test_graph {
    DEFINE FILENAME f;
    LOAD f TO VERTEX v_udt VALUES ($0, myTuple($1, $2, $3, $4) );
    # $1 is loaded as f1, $2 is loaded as f2, and so on
}
RUN LOADING JOB load_udt USING f="./udt.csv"
```

Loading a LIST or SET Attribute

There are three methods to load a `LIST` or a `SET`.

The first method is to load multiple rows of data that share the same ID values and append the individual attribute values to form a collection of values. The collections are formed incrementally by reading one value from each eligible data line and appending the new value into the collection. When the loading job processes a line, it checks to see whether a vertex or edge with that id value(s) already exists or not. If the id value(s) is new, then a new vertex or edge is created with a new list/set containing the single value. If the id(s) has been used before, then the value from the new line is appended to the existing list/set. Below shows an example:

Example: Cumulative loading of multiple rows to a SET/LIST

```
CREATE VERTEX test_vertex (PRIMARY_ID id STRING, iset SET<INT>, ilist
CREATE UNDIRECTED EDGE test_edge(FROM test_vertex, TO test_vertex)
CREATE GRAPH test_set_list (*)

CREATE LOADING JOB load_set_list FOR GRAPH test_set_list {
  DEFINE FILENAME f;
  LOAD f TO VERTEX test_vertex VALUES ($0, $1, $1);
}
RUN LOADING JOB load_set_list USING f="./list_set_vertex.csv"
```

29B list_set_vertex.csv

list_set_vertex.csv

list_set_vertex.csv

```
1,10
3,30
1,20
3,30
3,40
1,20
```

The job `load_set_list` will load two `test_vertex` vertices because there are two unique id values in the data file. Vertex 1 has attribute values with `iset = [10,20]` and `ilist = [10,20,20]`. Vertex 3 has values `iset = [30,40]` and `ilist = [30, 30, 40]`. Note that a set doesn't contain duplicate values, while a list can contain duplicate values.

Because GSQL loading is multi-threaded, the order of values loaded into a LIST might not match the input order.

If the input file contains multiple columns which should be all added to the LIST or SET, then a second method is available. Use the LIST() or SET() function as in the example below:

Example: loading multiple columns to a SET/LIST

```
CREATE VERTEX v_set  (PRIMARY_ID id STRING, nick_names SET<STRING>)
CREATE VERTEX v_list (PRIMARY_ID id STRING, lucky_nums LIST<INT>)
CREATE GRAPH test_graph (*)
CREATE LOADING JOB load_set_list FOR GRAPH test_graph {
    DEFINE FILENAME f;
    LOAD f TO VERTEX v_set  VALUES ($0, SET($1,$2,$3) );
    LOAD f TO VERTEX v_list VALUES ($0, LIST($2,$4) );
}
```

The third method is to use the **`SPLIT()`** function to read a compound token and split it into a collection of elements, to form a `LIST` or `SET` collection. The

`SPLIT()` function takes two arguments: the column index and the element separator. The element separator should be distinct from the separator throughout the whole file. Below shows an example:

Example: SET/LIST loading by SPLIT() example

```
CREATE VERTEX test_vertex (PRIMARY_ID id STRING, ustrset SET<STRING>,
CREATE UNDIRECTED EDGE test_edge(FROM test_vertex, TO test_vertex)
CREATE GRAPH test_split (*)

CREATE LOADING JOB set_list_job FOR GRAPH test_split {
  DEFINE FILENAME f;
  LOAD f TO VERTEX test_vertex VALUES ($0, SPLIT($1,"|") , SPLIT($2,"
}
RUN LOADING JOB set_list_job USING f="./split_list_set.csv"
```

54B split_list_set.csv

split_list_set.csv

split_list_set.csv

```
vid,names,numbers
v1,mike|tom|jack, 1 # 2 # 3
v2,john, 5 # 4 # 8
```

⚠ The `SPLIT()` function cannot be used for UDT type elements.

Loading a MAP Attribute

There are three methods to load a `MAP`.

The first method is to load multiple rows of data that share the same id values. The maps are formed incrementally by reading one key-value pair from each eligible data line. When the loading job processes a line, it checks to see whether a vertex or edge with that id value(s) already exists or not. If the id value(s) is new, then a new vertex or edge is created with a new map containing the single key-value pair. If the id(s) has been used before, then the loading job checks whether the key exists in the map or not. If the key doesn't exist in the map, the new key-value pair is inserted. Otherwise, the value will be replaced by the new value.

⚠ The loading order might not be the same as the order in the raw data. If a data file contains multiple lines with the same id and same key but different values, loading them together results in a nondeterministic final value for that key.

Method 1: Below is the syntax to load a `MAP` by the first method: Use an arrow (→) to separate the map's key and value.

Loading a MAP by method 1: → separator

```
CREATE VERTEX v_map  (PRIMARY_ID id STRING, att_map MAP<INT, STRING>)
CREATE GRAPH test_graph (*)
CREATE LOADING JOB load_map FOR GRAPH test_graph {
    DEFINE FILENAME f;
    LOAD f TO VERTEX v_map  VALUES ($0, ($1 -> $2) );
}
```

****

Method 2: The second method is to use the `MAP()` function. If there are multiple key-value pairs among multiple columns, `MAP()` can load them together. Below is an example:

Loading a MAP by method 2: MAP() function

```
CREATE VERTEX v_map  (PRIMARY_ID id STRING, att_map MAP<INT, STRING>)
CREATE GRAPH test_graph (*)
CREATE LOADING JOB load_map FOR GRAPH test_graph {
    DEFINE FILENAME f;
    LOAD f TO VERTEX v_map  VALUES ($0, MAP( ($1 -> $2), ($3 -> $4) )
}
```

Method 3: The third method is to use the `SPLIT()` function. Similar to the `SPLIT()` in loading `LIST` or `SET`, the `SPLIT()` function can be used when the key-value pair is in one column and separated by a key-value separator, or multiple key-value pairs are in one column and separated by element separators and key-value separators. `SPLIT()` here has three parameters: The first is the column index, the second is the key-value separator, and the third is the element separator. The third parameter is optional. If one row of raw data only has one key-value pair, the third parameter can be skipped. Below are the examples without and with the given element separator.

42B one_key_value.csv

one_key_value.csv

example data with one key-value pair per line

```
vid,key_value
v1,1:mike
v2,2:tom
v1,3:lucy
```

67B multi_key_value.csv

multi_key_value.csv

example data with multiple key-value pairs per line

```
vid,key_value_list
v1,1:mike#4:lin
v2,2:tom
v1,3:lucy#1:john#6:jack
```

Loading a MAP by method 3: SPLIT() function

```
CREATE VERTEX v_map  (PRIMARY_ID id STRING, att_map MAP<INT, STRING>)
CREATE GRAPH test_graph (*)
CREATE LOADING JOB load_map FOR GRAPH test_graph {
    DEFINE FILENAME f;
    LOAD f TO VERTEX v_map  VALUES ($0, SPLIT($1, ":", "#") );
}
```

The SPLIT() function cannot be used for UDT type elements.

Loading Composite Key Attributes

Loading a Composite Key for a vertex works no differently than normal loading. Simply load all the attributes as you would for a vertex with a single-attribute primary key. The primary key will automatically be constructed from the appropriate attributes.

When loading to an edge where either `TO_VERTEX` or `FROM_VERTEX` contains a composite key, the composite set of attributes must be enclosed in parentheses. See the example below.

Example: loading composite key to vertex and edge

```
#schema setup
CREATE VERTEX compositePerson (id uint, name string, PRIMARY KEY (nam
CREATE VERTEX compositeMovie (id uint, title string, country string,
CREATE DIRECTED EDGE compositeRoles (from compositePerson,to composit
CREATE GRAPH MyGraph(*)

#loading job
CREATE LOADING JOB composite_load FOR GRAPH MyGraph {
  LOAD "$sys.data_root/movies.csv" TO VERTEX compositeMovie VALUES
       ($"id", $"title", $"country" ,$"year") USING header ="true", s

  LOAD "$sys.data_root/persons.csv" TO VERTEX compositePerson VALUES
       ($"id",$"name") USING header = "true", separator =",";

  LOAD "$sys.data_root/compositeroles.csv" TO EDGE compositeRoles VAL
       (($"personName", $"personId"),($"movieTitle",$"movieYear",$"mo
       USING header="true", separator = ",";
}
```

Loading Wildcard Type Edges

If an edge has been defined using a wildcard vertex type, a vertex type name must be specified, following the vertex id, in a `LOAD` statement for the edge. An example is shown below:

Example: explicit vertex typing for an untyped edge

```
#schema setup
CREATE VERTEX user(PRIMARY_ID id UINT)
CREATE VERTEX product(PRIMARY_ID id UINT)
CREATE VERTEX picture(PRIMARY_ID id UINT)
CREATE UNDIRECTED EDGE purchase (FROM *, TO *)
CREATE GRAPH test_graph(*)

#loading job
CREATE LOADING JOB test2 FOR GRAPH test_graph {
  DEFINE FILENAME f;
  LOAD f
     TO EDGE purchase VALUES ($0 user, $1 product),
     TO EDGE purchase VALUES ($0 user, $2 picture);
  }
```

Token Functions

Token functions are functions which operate on tokens. Some may be used to construct attribute expressions and some may be used for conditional expressions in the `WHERE` clause.

To use a token function, replace the attribute in the destination clause of the `LOAD` statement with the function call. The arguments of the function can be a column from the `FILE` object.

Example

```
CREATE LOADING JOB loadOrders {
    DEFINE FILENAME f;
    LOAD f
        TO VERTEX order VALUES(gsql_trim($"id"), $"date");
}
```

Built-in token functions for attribute expressions

The following token functions can be used in an id or attribute expression

Function	Output type	Description
gsql_reverse(_in_string _)	string	Returns a string with the characters in the reverse order of the input string *in_string*.
gsql_concat(_string1, string2,...,stringN _)	string	Returns a string which is the concatenation of all the input strings.
gsql_uuid_v4()	string	Returns a version-4 UUID.
gsql_split_by_space(_in_string _)	string	Returns a modified version of *in_string*, in which each space character is replaced with ASCII 30 (decimal).
gsql_substring(*str*, *beginIndex* [, *length*])	string	Returns the substring beginning at *beginIndex*, having the given *length*.
gsql_find(*str*, _substr _)	int	Returns the start index of the substring within the string. If it is not found, then return -1.
gsql_length(_str _)	int	Returns the length of the string.
gsql_replace(*str*, *oldToken*, *newToken* [, *max*])	string	Returns the string resulting from replacing all matchings of *oldToken* with *newToken* in the original string. If a _max _count is provided, there can only be up to that many replacements.
gsql_regex_replace(*str*, *regex*, *replaceSubstr*)	string	Returns the string resulting from replacing all substrings in the input string that match the given *regex* token with the substitute string.
		Returns true if the given string token matches the

gsql_regex_match(*str*, *regex*)	bool	given regex token and false otherwise.
gsql_to_bool(_in_string _)	bool	Returns true if the _in_string _is either "t" or "true", with case insensitive checking. Returns false otherwise.
gsql_to_uint(_in_string _)	uint	If *in_string* is the string representation of an unsigned int, the function returns that integer. If *in_string* is the string representation of a nonnegative float, the function returns that number cast as an int.
gsql_to_int(_in_string _)	int	If *in_string* is the string representation of an int, the function returns that integer. If *in_string* is the string representation of a float, the function returns that number cast as an int.
gsql_ts_to_epoch_seconds(_timestamp _)	uint	Converts a timestamp in canonical string format to Unix epoch time, which is the int number of seconds since Jan. 1, 1970. Refer to the timestamp input format note below.
gsql_current_time_epoch(0)	uint	Returns the current time in Unix epoch seconds. *By convention, the input parameter should be 0, but it is ignored.
flatten(*column_to_be_split, group_separator, 1*) flatten(*column_to_be_split, group_separator, sub_field_separator,*		See the section "TEMP_TABLE and Flatten Functions" below.

number_of_sub_fields_in_one_group)		
flatten_json_array (*$"array_name"*) flatten_json_array (*$"array_name", $"sub_obj_1", $"sub_obj_2", ..., $"sub_obj_n"*)		See the section "TEMP_TABLE and Flatten Functions" below.
split(*column_to_be_split, element_separator*) split(*column_to_be_split, key_value_separator, element _separator*)		See the section "Loading a LIST or SET Attribute" above. See the section "Loading a MAP Attribute" above.
gsql_upper(_in_string _)	string	Returns the input string in upper-case.
gsql_lower(_in_string _)	string	Returns the input string in lower-case.
gsql_trim(_in_string _)	string	Trims whitespace from the beginning and end of the input string.
gsql_ltrim(*in_string*) gsql_rtrim(*in_string*)	string	Trims white space from either the beginning or the end of the input string (Left or right).
gsql_year(*timestamp*)	int	Returns 4-digit year from timestamp. Refer to timestamp input format note below.
gsql_month(timestamp)	int	Returns month (1-12) from timestamp. Refer to timestamp input format note below.
gsql_day(*timestamp*)	int	Returns day (1-31) from timestamp. Refer to timestamp input format note below.

gsql_year_epoch(*epoch*)	int	Returns 4-digit year from Unix epoch time, which is the int number of seconds since Jan. 1, 1970.
gsql_month_epoch(*epoch*)	int	Returns month (1-12) from Unix epoch time, which is the int number of seconds since Jan. 1, 1970.
gsql_day_epoch(*epoch*)	int	Returns day (1-31) from Unix epoch time, which is

Timestamp Input Format

The timestamp parameter should be in one of the following formats:
`"%Y-%m-%d %H:%M:%S"`
`"%Y/%m/%d %H:%M:%S"`
`"%Y-%m-%dT%H:%M:%S.000z" // text after the dot . is ignored`

User-defined token functions

Users can write their own token functions in C++ and install them in the GSQL system. To learn how to add a user-defined token function, see Add a User-Defined Token Function.

Reducer Functions

A reducer function aggregates multiple values of a non-id attribute into one attribute value of a single vertex or edge. Reducer functions are computed incrementally; that is, each time a new input token is applied, a new resulting value is computed.

To reduce and load aggregate data to an attribute, the attribute expression has the form

```
REDUCE( reducer_function ( input_expr ) )
```

where _reducer_function _is one of the functions in the table below. _input_expr _can include non-reducer functions, but reducer functions cannot be nested.

Each reducer function is overloaded so that one function can be used for several different data types. For primitive data types, the output type is the same as the _input_expr _type. For LIST, SET, and MAP containers, the _input_expr _type is one of the allowed element types for these containers (see "Complex Types" in the Attribute Data Types section). The output is the entire container.

Function name	Data type of *arg*: Description of function's return value
max(_arg _)	INT, UINT, FLOAT, DOUBLE: maximum of all _arg _values cumulatively received
min(_arg _)	INT, UINT, FLOAT, DOUBLE: minimum of all _arg _values cumulatively received
add(_arg _)	INT, UINT, FLOAT, DOUBLE: sum of all *arg* values cumulatively received STRING: concatenation of all arg values cumulatively received LIST, SET element: list/set of all *arg* values cumulatively received MAP (key → value) pair: key-value dictionary of all key-value pair *arg* values cumulatively received
and(_arg _)	BOOL: AND of all *arg* values cumulatively received INT, UINT: bitwise AND of all *arg* values cumulatively received
or(_arg _)	BOOL: OR of all *arg* values cumulatively received INT, UINT: bitwise OR of all *arg* values cumulatively received
overwrite(_arg _)	non-container: *arg* LIST, SET: new list/set containing only *arg*

ignore_if_exists(_arg _)	Any: If an attribute value already exists, return(retain) the existing value. Otherwise,

> ⚠ Each function supports a certain set of attribute types. Calling a reducer function with an incompatible type crashes the service. In order to prevent that, use the WHERE clause (introduced below) together with IS NUMERIC or other operators, functions, predicates for type checking if necessary.

WHERE Clause

The `WHERE` clause is an optional clause. The `WHERE` clause's condition is a boolean expression. The expression may use column token variables, token functions, and operators which are described below. The expression is evaluated for each input data line. If the condition is true, then the vertex or edge instance is loaded into the graph store. If the condition is false, then this instance is skipped. Note that all attribute values are treated as string values in the expression, so the type conversion functions `to_int()` and `to_float()`, which are described below, are provided to enable numerical conditions.

Operators in the WHERE Clause

The GSQL Loader language supports most of the standard arithmetic, relational, and boolean operators found in C++. Standard operator precedence applies, and parentheses provide the usual override of precedence.

- **Arithmetic Operators: +, -, *, /, ^ **
  ****Numeric operators can be used to express complex operations between numeric types. Just as in ordinary mathematical expressions, parentheses can be used to define a group and to modify the order of precedence.

> ⚠ Because computers necessarily can only store approximations for most `DOUBLE` and `FLOAT` type values, it is not recommended to test these data types for exact equality or inequality. Instead, one should allow for an acceptable amount of error. The following example checks if `$0 = 5`, with an error of 0.00001 permitted:

```
WHERE to_float($0) BETWEEN 5-0.00001 AND 5+0.00001
```

- **Relational Operators: <, >, ==, !=, <=, >= **
****Comparisons can be performed between two numeric values or between two string values. \
- **Predicate Operators:**
  - **AND, OR, NOT **operators are the same as in SQL. They can be used to combine multiple conditions together.
E.g., _$0 < "abc" AND $1 > "abc" _selects the rows with the first token less than "abc" and the second token greater than "abc".
E.g., _NOT $1 < "abc" _selects the rows with the second token greater than or equal to "abc". \
  - **IS NUMERIC
_token _IS NUMERIC **returns true if **token **is in numeric format. Numeric format include integers, decimal notation, and exponential notation. Specifically, IS NUMERIC is true if token matches the following regular expression: (+/-) ? [0-9] + (.[0-9]) ? [0-9] * ((e/E)(+/-) ? [0-9] +) ? . Any leading space and trailing space is skipped, but no other spaces are allowed.
E.g., _$0 IS NUMERIC _checks whether the first token is in numeric format. \
  - **IS EMPTY
_token _IS EMPTY **returns true if **token **is an empty string.
E.g., _$1 IS EMPTY _checks whether the second token is empty. \
  - **IN
****_token _IN (_set_of_values _) **returns true if **token **is equal to one member of a set of specified values. The values may be string or numeric types.
E.g., _$2 IN ("abc", "def", "lhm") _tests whether the third token equals one of the three strings in the given set.
E.g., _to_int($3) IN (10, 1, 12, 13, 19) _tests whether the fourth token equals one of the specified five numbers. \
  - **BETWEEN ... AND
_token _BETWEEN _lowerVal _AND _upperVal **_returns true if **token **is within the specified range, inclusive of the endpoints. The values may be string or numeric types.
E.g., _$4 BETWEEN "abc" AND "def" _checks whether the fifth token is

greater than or equal to "abc" and also less than or equal to "def"
E.g., _to_float($5) BETWEEN 1 AND 100.5 _checks whether the sixth token is greater than or equal to 1.0 and less than or equal to 100.5.

Token functions in the WHERE clause

The GSQL loading language provides several built-in functions for the WHERE clause.

Function name	Output type	Description of function
to_int(_main_string _)	int	Converts _main_string _to an integer value.
to_float(_main_string _)	float	Converts _main_string _to a float value.
concat(_string1, string2 _)	string	Returns a string which is the concatenation of _string1 _and _string2 _.
token_len(_main_string _)	int	Returns the length of *main_string.*
gsql_is_not_empty_string(_main_string _)	bool	Returns true if _main_string _is empty after removing white space. Returns false otherwise.
gsql_token_equal(_string1, string2 _)	bool	Returns true if _string1 _is exactly the same (case sensitive) as _string2 _. Returns false otherwise.
gsql_token_ignore_case_equal(_string1, string2 _)	bool	Returns true if _string1 _is exactly the same (case insensitive) as _string2 _. Returns false otherwise.
gsql_is_true(_main_string _)	bool	Returns true if _main_string _is either "t" or "true" (case insensitive). Returns false otherwise.

gsql_is_false(_main_string _)	bool	Returns true if _main_string _is either "f" or "false" (case insensitive). Returns false otherwise.

The token functions in the WHERE clause and those token functions used for attribute expression are different. They cannot be used exchangeably.

Other Optional LOAD Clauses

`OPTION` clause

There are no supported options for the OPTION clause at this time.

`TAGS` clause (Beta)

The `TAGS` clause specifies the tags to be applied to the vertices loaded by the `LOAD` statement.

```
TAGS "(" tag_name (, tag_name)* ")" BY [ OR | OVERWRITE ]
```

If a `LOAD` statement has a `TAGS` clause, it will tag the vertices with the tags specified in the `TAGS` clause. Before vertices can be loaded and tagged with a `LOAD` statement, the vertex type must first be marked as taggable, and the tags must be defined.

Users have two options when it comes to how to merge tags if the target vertices exist in the graph:

- `BY OR` : Add the new tags to the existing set of tags.
- `BY OVERWRITE`: Overwrite existing tags with the new tags.

`USING` clause

A `USING` clause contains one or more optional parameter value pairs:

```
USING parameter=value [,parameter=value]*
```

> If multiple LOAD statements use the same source (the same file path, the same TEMP_TABLE, or the same file variable), the USING clauses in these LOAD statements must be the same. Therefore, we recommend that if multiple destination clauses share the same source, put all of these destination clauses into the same LOAD statement.

Parameter	Meaning of Value	Allowed Values
SEPARATOR	specifies the special character that separates tokens (columns) in the data file	any single ASCII character. Default is comma "," `"\t"` for tab `"\xy"` for ASCII decimal code xy
EOL	the end-of-line character	any ASCII sequence Default = **`"\n"`** (system-defined newline character or character sequence)
QUOTE (See note below)	specifies explicit boundary markers for string tokens, either single or double quotation marks. See more details below.	"single" for ' "double" for "
HEADER	whether the data file's first line is a header line. The header assigns names to the columns. The LOAD statement must refer to an actual file with a valid header.	"true", "false" Default is "false"
USER_DEFINED_HEADER	specifies the name of the header variable, when a header has been defined in	the variable name in the preceding DEFINE HEADER

	the loading job, rather than in the data file	statement
REJECT_LINE_RULE	if the filter expression evaluates to true, then do not use this input data line.	name of filter from a preceding DEFINE INPUT_LINE_FILTER statement
JSON_FILE (See Loading JSON Data section below)	whether each line is a json object (see Section "JSON Loader" below for more details)	"true", "false" Default is "false"
NEW_VERTEX_ONLY	If true, treat vertices as insert-only. If the input data refers to a vertex which already exists, do not update it. If false, upsert vertices.	"true", "false" Default is "false"
VERTEX_MUST_EXIST (See VERTEX_MUST_EXIST section below)	If true, only insert or update an edge If both endpoint vertices already exist. If false, always insert new edges, creating endpoint vertices as needed, using given id and default values for other parameters.	"true", "false" Default is "false"

QUOTE parameter

The parser will not treat separator characters found within a pair of quotation marks as a separator. For example, if the parsing conditions are `QUOTE="double", SEPARATOR=","`, the comma in `"Leonard,Euler"` will not separate Leonard and Euler into separate tokens. \

- If QUOTE is not declared, quotation marks are treated as ordinary characters.
- If QUOTE is declared, but a string does not contain a matching pair of quotation marks, then the string is treated as if QUOTE is not declared.

- Only the string inside the first pair of quote (from left to right) marks are loaded. For example QUOTE="double", the string a"b"c"d"e will be loaded as b.
- There is no escape character in the loader, so the only way to include quotation marks within a string is for the string body to use one type of quote (single or double) and to declare the other type as the string boundary marker.

> Previously, ill-formatted strings such as **a"a,b"ac,d **would be parsed as a,b,d ignoring a,a,c. The expected input string should be **a,"a,b",ac,d**. In v2.4, incorrectly formatted strings such as this example will be parsed normally, giving you this result: **a"a,b"ac** and **d.**

Loading JSON Data

When the USING option `JSON_FILE="true"` is used, the loader loads JSON objects instead of tabular data. A JSON object is an unordered set of key/value pairs, where each value may itself be an array or object, leading to nested structures. A colon separates each key from its value, and a comma separates items in a collection. A more complete description of JSON format is available at www.json.org. The JSON loader requires that each input line has exactly one JSON object. Instead of using column values as tokens, the JSON loader uses JSON values as tokens, that is, the second part of each JSON key/value pair. In a GSQL loading job, a JSON field is identified by a dollar sign $ followed by the colon-separated sequence of nested key names to reach the value from the top level. For example, given the JSON object `{"abc":{"def": "this_value"}}`, the identifier `$"abc":"def"` is used to access `"this_value"`. The double quotes are mandatory.

An example is shown below:

USING JSON_FILE test schema and loading job

```
CREATE VERTEX encoding (PRIMARY_ID id STRING, length FLOAT default 10
CREATE UNDIRECTED EDGE encoding_edge (FROM encoding, TO encoding)
CREATE GRAPH encoding_graph (*)

CREATE LOADING JOB json_load FOR GRAPH encoding_graph {
  LOAD "encoding.json" TO VERTEX encoding
    VALUES ($"encoding", $"indent":"length") USING JSON_FILE="true";
}
RUN JOB json_load
```

To specify an end-of-line character other than the standard one, use the EOL option, as shown below.

JSON loading using EOL

```
CREATE LOADING JOB json_load2 FOR GRAPH companyGraph {

  LOAD "/tmp/data.json"
    TO VERTEX company VALUES($"company":"name":"value",$"company":"na
    TO VERTEX members VALUES($"members",$"members") USING JSON_FILE="
}
```

267B

encoding.json

encoding.json - Download

encoding.json

```
{"encoding": "UTF-7","plug-ins":["c"],"indent" : { "length" : 30, "us
{"encoding":"UTF-1","indent":{"use_space": "dontloadme"}, "plug-ins"
{"plug-ins":["C","c++"],"indent":{"length" : 3, "use_space": false},"
```

In the above data encoding.json, the order of fields are not fixed and some fields are missing. The JSON loader ignores the order and accesses the fields by the nested key names. The missing fields are loaded with default values. The result vertices are:

id	attr1
"UTF-7"	30
"UTF-1"	0
"UTF-6"	3

Loading Parquet Data

TigerGraph can load data from Parquet files if they are stored in AWS S3 buckets. For more details on how to set up S3 data sources and loading jobs, read the AWS S3 Loader User Guide. In the background TigerGraph uses the JSON loading functionality to read data from Parquet files, so the JSON specific information in the previous section applies.

In order to load Parquet data, you need to:

1. Specify `"file.reader.type": "parquet"` in the S3 file configuration file or argument
2. Specify `JSON_FILE="true"` in the USING clause of the LOAD statements
3. Refer to JSON keys (≈ Parquet "column names") instead of column numbers

You will probably want to add `USING EOF="true"` to your RUN LOADING JOB statement to explicitly indicate to the loading job to stop after consuming all data from the Parquet source, not to expect further entries.

An example of a Parquet loading setup is shown below:

```
CREATE DATA_SOURCE S3 s3ds = "{\"file.reader.settings.fs.s3a.access.key\"

CREATE LOADING JOB parquet_load FOR GRAPH companyGraph {

    DEFINE FILENAME f = "$s3ds:{\"file.uris\": \"s3://mybucket/mydata.par

    LOAD f
      TO VERTEX members VALUES($"members", $"members") USING JSON_FILE="t
}

RUN LOADING JOB parquet_load USING EOF="true"
```

`VERTEX_MUST_EXIST` Parameter

Normally, if vertices do not exist when loading data to edges, a vertex will be created for the connecting edge, using default values for all attributes. Using the VERTEX_MUST_EXIST="true" option will load data only if the vertices on both sides of an edge already exist, therefore no longer creating extra vertices.

```
CREATE LOADING JOB load_edge FOR GRAPH MyGraph {
    DEFINE FILENAME f;
    LOAD f
        TO EDGE MyEdge VALUES ($1, $2, $3,) USING VERTEX_MUST_EXIST="true
}
```

`TEMP_TABLE` and Flatten Functions

The keyword `TEMP_TABLE` triggers the use of a temporary data table which is used to store data generated by one `LOAD` statement, for use by a later `LOAD` statement. Earlier we introduced the syntax for loading data to a `TEMP_TABLE`:

TEMP_TABLE Destination Clause

```
TO TEMP_TABLE table_name (id_name [, attr_name]*) VALUES (id_expr [,
    [WHERE conditions] [OPTION (options)]
```

This clause is designed to be used in conjunction with the `flatten` or `flatten_json_array` function in one of the `attr_expr` expressions. The flatten function splits a multi-value field into a set of records. Those records can first be stored in a temporary table, and then the temporary table can be loaded into vertices and/or edges. Only one flatten function is allowed in one temp table destination clause.

There are two versions of the flatten function: One parses single-level groups and the other parses two-level groups. There are also two versions of the flatten_json_array function: One splits an array of primitive values, and the other splits an array of JSON objects.

One-Level Flatten Function

`flatten( column_to_be_split, separator, 1 )` is used to parse a one-level group into individual elements. An example is shown below:

140B book1.dat

book1.dat

book1.dat

```
101|"Harry Potter and the Philosopher's Stone"|"fiction,fantasy,young
102|"The Three-Body Problem"|"fiction,science fiction,Chinese"
```

The following loading job contains two `LOAD` statements. The first one loads input data to `Book` vertices and to a `TEMP_TABLE`. The second one loads the `TEMP_TABLE` data to `Genre` vertices and `book_genre` edges.

One-level Flatten Function loading (load_book_flatten1.gsql)

```
CREATE LOADING JOB load_books_flatten1 FOR GRAPH Book_rating {
  DEFINE FILENAME f;
  LOAD f
      TO VERTEX Book VALUES ($0, $1, _),
      TO TEMP_TABLE t1(bookcode,genre) VALUES ($0, flatten($2,",",1))
      USING QUOTE="double", SEPARATOR="|";

  LOAD TEMP_TABLE t1
      TO VERTEX Genre VALUES($"genre", $"genre"),
      TO EDGE book_genre VALUES($"bookcode", $"genre");
}
RUN LOADING JOB load_books_flatten1 USING f="../data/book1.dat"
```

Line 5 says that the third column (`$2`) of each input line should be split into separate tokens, with comma (`,`) as the separator. Each token will have its own row in table `t1`. The first column is labeled `bookcode` with value `$0` and the second column is `genre` with one of the `$2` tokens. The contents of `TEMP_TABLE t1` are shown below:

bookcode	genre
101	fiction
101	fantasy
101	young_adult
102	fiction
102	science_fiction
102	Chinese

Then, lines 8 to 10 say to read `TEMP_TABLE t1` and to do the following for each row:

- Create a `Genre` vertex for each new value of `genre`.
- Create a `book_genre` edge from `bookcode` to `genre`. In this case, each row of `TEMP_TABLE t1` generates one `book_genre` edge.

The final graph will contain two `Book` vertices (101 and 102), five Genre vertices, and six `book_genre` edges.

List of all book_genre edges after loading

```
{
  "results": [{"@@edgeSet": [
    {
      "from_type": "Book",
      "to_type": "Genre",
      "directed": false,
      "from_id": "101",
      "to_id": "fiction",
      "attributes": {},
      "e_type": "book_genre"
    },
    {
      "from_type": "Book",
      "to_type": "Genre",
      "directed": false,
      "from_id": "101",
      "to_id": "fantasy",
      "attributes": {},
      "e_type": "book_genre"
    },
    {
      "from_type": "Book",
      "to_type": "Genre",
      "directed": false,
      "from_id": "102",
      "to_id": "sciencevfiction",
      "attributes": {},
      "e_type": "book_genre"
    },
    {
      "from_type": "Book",
      "to_type": "Genre",
      "directed": false,
      "from_id": "101",
      "to_id": "young adult",
      "attributes": {},
      "e_type": "book_genre"
    },
    {
      "from_type": "Book",
      "to_type": "Genre",
      "directed": false,
      "from_id": "102",
      "to_id": "fiction",
      "attributes": {},
      "e_type": "book_genre"
    },
    {
```

```
        "from_type": "Book",
        "to_type": "Genre",
        "directed": false,
        "from_id": "102",
        "to_id": "Chinese",
        "attributes": {},
        "e_type": "book_genre"
      }
    ]}]
}
```

Two-Level Flatten Function

flatten(_column_to_be_split, group_separator, sub_field_separator, number_of_sub_fields_in_one_group _) is used for parse a two-level group into individual elements. Each token in the main group may itself be a group, so there are two separators: one for the top level and one for the second level. An example is shown below.

book2.dat

```
101|"Harry Potter and the Philosopher's Stone"|"FIC:fiction,FTS:fanta
102|"The Three-Body Problem"|"FIC:fiction,SF:science fiction,CHN:Chin
```

The flatten function now has four parameters instead of three. The additional parameter is used to record the genre_name in the Genre vertices.

Two-level Flatten Function loading (book_flatten2_load.gsql)

```
CREATE LOADING JOB load_books_flatten2 FOR GRAPH Book_rating {
  DEFINE FILENAME f;
  LOAD f
      TO VERTEX Book VALUES ($0, $1, _),
      TO TEMP_TABLE t2(bookcode,genre_id,genre_name) VALUES ($0, flat
      USING QUOTE="double", SEPARATOR="|";

  LOAD TEMP_TABLE t2
      TO VERTEX Genre VALUES($"genre_id", $"genre_name"),
      TO EDGE book_genre VALUES($"bookcode", $"genre_id");
}
RUN LOADING JOB load_books_flatten2 USING f="book2.dat"
```

In this example, in the genres column ($2), there are multiple groups, and each group has two sub-fields, genre_id and genre_name. After running the loading job, the file book2.dat will be loaded into the TEMP_TABLE t2 as shown below.

bookcode	genre_id	
101	FIC	fiction
101	FTS	fantasy
101	YA	young adult
102	FIC	fiction
102	SF	science fiction
102	CHN	Chinese

Flatten a JSON Array of Primitive Values

flatten_json_array($" _array_name _") parses a JSON array of primitive (string, numberic, or bool) values, where "array_name" is the name of the array. Each value in the array creates a record. Below is an example:

flatten_json_array_values loading

```
CREATE VERTEX encoding (PRIMARY_ID id STRING, length FLOAT default 10
CREATE UNDIRECTED EDGE encoding_edge (FROM encoding, TO encoding)
CREATE GRAPH encoding_graph (*)

CREATE LOADING JOB json_flatten FOR GRAPH encoding_graph {
  LOAD "encoding2.json" TO TEMP_TABLE t2 (name, length)
    VALUES (flatten_json_array($"plug-ins"), $"indent":"length") USIN
  LOAD TEMP_TABLE t2
    TO VERTEX encoding VALUES ($"name", $"length");
}
RUN LOADING JOB json_flatten
```

95B

encoding2.json

encoding2.json - Download

encoding2.json

```
{"plug-ins" : ["C", "c++"],"encoding" : "UTF-6","indent" : { "length"
```

The above data and loading job creates the following temporary table:

id	length
C	3
c++	3

Flatten a JSON Array of JSON Objects

flatten_json_array (_$"array_name", $"sub_obj_1", $"sub_obj_2", ..., $"sub_obj_n" _) parses a JSON array of JSON objects. "array_name" is the name of the array, and the following parameters _$"sub_obj_1", $"sub_obj_2", ..., $"sub_obj_n" _are the field key names in each object in the array. See complete example below:

encoding3.json

594B

encoding3.json - Download

encoding3.json

```
{"encoding":"UTF-1","indent":{"use_space": "dontloadme"}, "plug-ins"
{"encoding": "UTF-8", "plug-ins" : [{"lang": "pascal", "score":"1.0",
{"encoding": "UTF-7",  "plug-ins" : [{"lang":"java", "score":2.22}, {
{"plug-ins" : ["C", "c++"],"encoding" : "UTF-6","indent" : { "length"
```

json_flatten_array_test.gsql

```
CREATE VERTEX encoding3 (PRIMARY_ID id STRING, score FLOAT default -1
CREATE UNDIRECTED EDGE encoding3_edge (FROM encoding3, TO encoding3)
CREATE GRAPH encoding_graph (*)

CREATE LOADING JOB json_flatten_array FOR GRAPH encoding_graph {
  LOAD "encoding3.json" TO TEMP_TABLE t3 (name, score, prop_age, inde
    VALUES (flatten_json_array($"plug-ins", $"lang", $"score", $"prop
    USING JSON_FILE="true";
  LOAD TEMP_TABLE t3
    TO VERTEX encoding3 VALUES ($"name", $"score", $"prop_age", $"ind
}
RUN LOADING JOB json_flatten_array
```

When splitting a JSON array of JSON objects, the primitive values are skipped and only JSON objects are processed. As in the example above, the 4th line's "plug-ins" field will not generate any record because its "plug-ins" array doesn't contain any JSON object. Any field which does not exist in the object will be loaded with default value. The above example generates the temporary table shown below:

id	score	age	length
"golang"	*default*	"noidea"	*default*
"pascal"	1.0	"old"	12
"c++"	2.0	*default*	12

"java"	2.22	*default*	30
"python"	3.0	*default*	30

Flatten a JSON Array in a CSV file

**flatten_json_array() **can also be used to split a column of a tabular file, where the column contains JSON arrays. An example is given below:

336B encoding.csv

encoding.csv

encoding.csv

```
golang|{"prop":{"age":"noidea"}}
pascal|{"score":"1.0", "prop":{"age":"old"}}
c++|{"score":2.0, "indent":{"length":12, "use_space": true}}
java|{"score":2.22, "prop":{"age":"new"}, "indent":{"use_space":"true
python|{ "prop":{"compiled":"false"}, "indent":{"length":4}, "score":
go|{"score":4.0, "prop":{"age":"new"}}
```

The second column in the csv file is a JSON array which we want to split. flatten_json_array() can be used in this case without the USING JSON_FILE="true" clause:

json_flatten_cvs.gsql

```
CREATE VERTEX encoding3 (PRIMARY_ID id STRING, score FLOAT default -1
CREATE UNDIRECTED EDGE encoding3_edge (FROM encoding3, TO encoding3)
CREATE GRAPH encoding_graph (*)

CREATE LOADING JOB json_flatten_cvs FOR GRAPH encoding_graph {
  LOAD "encoding.csv" TO TEMP_TABLE t4 (name, score, prop_age, indent
    VALUES ($0,flatten_json_array($1, $"score", $"prop":"age", $"inde
    USING SEPARATOR="|";
  LOAD TEMP_TABLE t4
    TO VERTEX encoding3 VALUES ($"name", $"score", $"prop_age", $"ind
}
RUN LOADING JOB json_flatten_cvs
```

The above example generates the temporary table shown below:

id	score	age	length
golang	-1 (default)	noidea	-1 (default)
pascal	1	old	-1 (default)
c++	2	unknown (default)	12
java	2.22	new	2
python	3	unknown (default)	4
go	4	new	-1 (default)

flatten_json_array in csv

flatten_json_array() does not work if the separator appears also within the json array column. For example, if the separator is comma, the csv loader will erroneously divide the json array into multiple columns. Therefore, it is recommended that the csv file use a special column separator, such as "|" in the above example .

DELETE statement

In addition to loading data, a LOADING JOB can be used to perform the opposite operation: deleting vertices and edges, using the DELETE statement. DELETE cannot be used in offline loading. Just as a LOAD statement uses the tokens from each input line to set the id and attribute values of a vertex or edge to be created, a DELETE statement uses the tokens from each input line to specify the id value of the item(s) to be deleted.

> In the v2.0 syntax, there is now a " `FROM (filepath_string | filevar)` " clause just before the WHERE clause.

There are four variations of the DELETE statement. The syntax of the four cases is shown below.

DELETE VERTEX | EDGE Syntax

```
CREATE LOADING JOB abc FOR GRAPH graph_name {
  DEFINE FILENAME f;
  # 1. Delete each vertex which has the given vertex type and primary
  DELETE VERTEX vertex_type_name (PRIMARY_ID id_expr) FROM f
    [WHERE condition] [USING options];

  # 2. Delete each edge which has the given edge type, source vertex
  DELETE EDGE edge_type_name (FROM id_expr, TO id_expr) FROM f
    [WHERE condition] [USING options];

  # 3. Delete all edges which have the given edge type and source ver
  DELETE EDGE edge_type_name (FROM id_expr) FROM f
    [WHERE condition] [USING options];

  # 4. Delete all edges which have the given source vertex id. (Edge
  DELETE EDGE * (FROM id_expr vertex_type_name) FROM f
    [WHERE condition] [USING options];
}
```

An example using book_rating data is shown below:

DELETE example

```
# Delete all user occupation edges if the user is in the new files, t
CREATE LOADING JOB clean_user_occupation FOR GRAPH Book_rating {
  DEFINE FILENAME f;
  DELETE EDGE user_occupation (FROM $0) FROM f;
}
CREATE LOADING JOB load_user_occupation FOR GRAPH Book_rating {
  DEFINE FILENAME f;
  LOAD f TO EDGE user_occupation VALUES ($0,$1);
}
RUN LOADING JOB clean_user_occupation USING f="./data/user_occupation
RUN LOADING JOB load_user_occupation USING f="./data/user_occupation_
```

There is a separate DELETE statement in the GSQL Query Language. The query delete statement can leverage the query language's ability to explore the graph and to use complex conditions to determine which items to delete. In contrast, the loading job delete statement requires that the id values of the items to be deleted must be specified in advance in an input file.

Loading...

Loading...

Appendix

DDL Keywords & Reserved Words

GSQL Reserved Words

The following words are reserved for use by the Data Definition Language. That is, a graph schema or loading job may not use any of these words for the **name of a vertex type, edge type, graph name, tag, or attribute**.
There is a separate list for Reserved Words in the **Query Language**.
The compiler will reject the use of a reserved word as well as any word beginning with a reserved prefix as a user-defined identifier.

History

- v3.1: Added reserved prefix `gsql_sys_`

Reserved prefixes:

- `gsql_sys_`

Reserved word list:

```
ACCUM              ADD                ADMIN              ALL
ALLOCATE           ALTER              AND                ANY
AS                 ASC                AVG                BAG
BATCH              BETWEEN            BIGINT             BLOB
BOOL               BOOLEAN            BOTH               BREAK
BY                 CALL               CASCADE            CASE
CATCH              CHAR               CHARACTER          CHECK
CLOB               COALESCE           COMPRESS           CONST
CONSTRAINT         CONTINUE           COUNT              CREATE
CURRENT_DATE       CURRENT_TIME       CURRENT_TIMESTAMP  CURSOR
DATA_SOURCE        DATETIME           DATETIME_ADD       DATETIME_SUB
DECIMAL            DECLARE            DEFAULT            DELETE
DESC               DISTRIBUTED        DO                 DOUBLE
DROP               EDGE               ELSE               ELSEIF
END                ESCAPE             EXCEPTION          EXISTS
EXPRFUNCTIONS      EXPRUTIL           FALSE              FILE
FILENAME           FILTER             FIXED_BINARY       FLATTEN_JSON_
FLOAT              FOR                FOREACH            FROM
GLOBAL             GRAPH              GROUP              GROUPBYACCUM
HAVING             HEADER             HEAPACCUM          IF
IGNORE             IN                 INDEX              INPUT_LINE_FI
INSERT             INT                INT16              INT32
INT32_T            INT64_T            INT8               INTEGER
INTERPRET          INTERSECT          INTO               IS
ISEMPTY            JOB                JOIN               JSONARRAY
JSONOBJECT         KAFKA              KEY                LEADING
LIKE               LIMIT              LIST               LOAD
LOADACCUM          LOG                LONG               MAP
NOT                NOW                NULL               OFFSET
ON                 OR                 ORDER              PINNED
POST_ACCUM         PRIMARY            PRIMARY_ID         PRINT
PROXY              QUERY              QUIT               RAISE
RANGE              REDUCE             REPLACE            RETURN
RETURNS            S3                 SAMPLE             SELECT
SELECTVERTEX       SET                STATIC             STRING
SUM                TARGET             TEMP_TABLE         THEN
TO                 TOKEN              TOKENBANK          TOKEN_LEN
TO_CSV             TO_DATETIME        TO_FLOAT           TO_INT
TRAILING           TRIM               TRUE               TRY
TUPLE              TYPEDEF            UINT               UINT16
UINT32             UINT32_T           UINT64_T           UINT8
UINT8_T            UNION              UPDATE             UPSERT
USING              VALUES             VERTEX             WHEN
WHERE              WHILE
```

GSQL Non-Reserved Keywords

These keywords in the language are non-reserved, so users may use them for user-defined identifiers.

History

- v3.1: Added `DESCRIPTION,` `TAG`, `TAGS`

Non-reserved keyword list:

```
ABORT            API              APPROX_COUNT     ATTRIBUTE
BEGIN            CHANGE           CLEAR            CONCAT
DATA             DATASRC          DECRYPT          DEFAULT
DEFINE           DESCRIPTION      DIRECTED         EMPTY
EXIT             EXPORT           EXPR_FUNC        EXPR_UTIL
EXTERN           FILENAMEVAR      FLATTEN          FLATTENJSON
GENERATEDATA     GET              GRANT            HELP
ICON             IMPORT           INSTALL          JSON
LOADING          LOCAL            LS               MAX
MIN              MINUS            NUMERIC          OPTION
OVERWRITE        PAIR             PASSWORD         PUT
RECOMPILE        REJECT_LINE_RULE RESUME           REVOKE
ROLE             RUN              SCHEMA_CHANGE    SCHEMA
SECONDARY_ID     SECRET           SEPARATOR        SHOW
SPLIT            STATS            STATUS           STORE
SUBSTR           SYNTAX           TAG              TAGS
TEMPLATE         TK               TOFLOAT          TOINT
TOKEN_BANK       TOKENLEN         UNDIRECTED       USE
USER             USERS            VAL              VECTOR
VERSION          VOID             WITH
```

GSQL Start-to-End Process and Data Flow

The figures below illustrates the sequence of steps and the dependencies to progress from no graph to a loaded graph and a query result, for TigerGraph platform version 0.8 and higher. Note that online and offline follow the same flow.

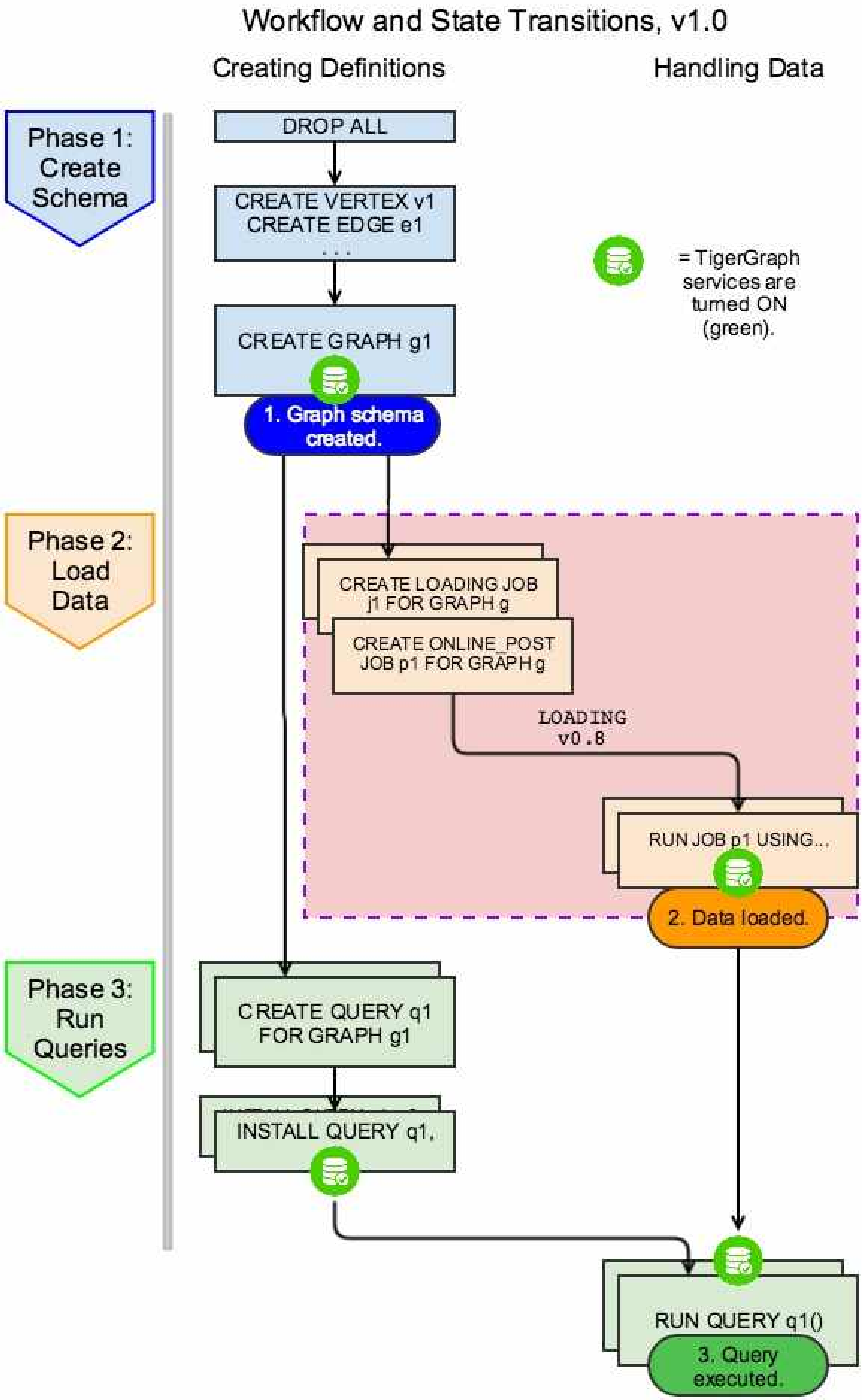

Workflow and State Transitions, v1.0
Creating Definitions
Handling Data
Phase 1: Create Schema
DROP ALL
CREATE VERTEX v1
CREATE EDGE e1
. . .
= TigerGraph services are turned ON (green).
CREATE GRAPH g1
1. Graph schema created.
Phase 2: Load Data
CREATE LOADING JOB j1 FOR GRAPH g
CREATE ONLINE_POST JOB p1 FOR GRAPH g
LOADING v0.8
RUN JOB p1 USING...
2. Data loaded.
Phase 3: Run Queries
CREATE QUERY q1 FOR GRAPH g1
INSTALL QUERY q1,
RUN QUERY q1()
3. Query executed.

Figure B1: Complete GSQL Workflow

Part 2 - Querying

Table of Contents

Exception Statements

Loading...

CREATE/INTERPRET/INSTALL/RUN QUERY

Query Privileges

Users with the `querywriter` role or greater (`designer`, `admin`, `globaldesigner`, and `superuser`) can create, install, and drop queries.

Any user with the `queryreader` role or greater for a given graph can run the queries for that graph.

To implement fine-grained control over which queries can be executed by which sets of users:

1. Group your queries into your desired privilege groups.
2. Define a graph for each privilege group. These graphs can all have the same domain if you wish.
3. Create your queries, assigning each to its appropriate privilege group.

EBNF for CREATE QUERY

```
createQuery := CREATE [OR REPLACE] [DISTRIBUTED] QUERY queryName
               "(" [parameterList] ")"
               [FOR GRAPH graphName]
               [RETURNS "("  baseType | accumType ")"]
               [API "(" stringLiteral ")"]
               [SYNTAX syntaxName]
               "{" queryBody "}"

interpretQuery := INTERPRET QUERY "(" ")"
               [FOR GRAPH graphName]
               [SYNTAX syntaxName]
               "{" queryBody "}"

parameterValueList := parameterValue ["," parameterValue]*
parameterValue := parameterConstant
                | "[" parameterValue ["," parameterValue]* "]"  // BAG or
                | "(" stringLiteral "," stringLiteral ")"        // gener
parameterConstant := numeric | stringLiteral | TRUE | FALSE
parameterList := parameterType paramName ["=" constant]
                 ["," parameterType paramName ["=" constant]]*

syntaxName := name

queryBody := [typedefs] [declStmts] [declExceptStmts] queryBodyStmts
typedefs := (typedef ";")+
declStmts := (declStmt ";")+
declStmt := baseDeclStmt | accumDeclStmt | fileDeclStmt
declExceptStmts := (declExceptStmt ";")+
queryBodyStmts := (queryBodyStmt ";")+

installQuery := INSTALL QUERY [installOptions] ( "*" | ALL |queryName [",
runQuery := RUN QUERY [runOptions] queryName "(" parameterValueList ")"

showQuery := SHOW QUERY queryName
dropQuery := DROP QUERY ( "*" | ALL | queryName ["," queryName]* )
```

A GSQL query is a sequence of data retrieval-and-computation statements executed as a single operation. Users can write queries to explore a data graph however they like, to read and make computations on the graph data along the way, to update the graph, and to deliver resulting data. A query is analogous to a user-defined procedure or function: it can have one or more input parameters, and it can produce output in two ways: by returning a value or by printing. A query can be run in one of three ways:

1. Define and run an unnamed query immediately:
   a. `INTERPRET QUERY`: execute the query's statements

   Alternately, there is also a built-in REST++ endpoint to interpret a query string:
   `POST /gsqlserver/interpreted_query`
   See the RESTPP API User Guide for details.
2. Define a named query and then run it.
   a. `CREATE QUERY`: define the functionality of the query
   b. `INTERPRET QUERY`: execute the query with input values
3. Define a named query, compile it to optimize performance, and then run it.
   a. `CREATE QUERY`: define the functionality of the query
   b. `INSTALL QUERY`: compile the query
   c. `RUN QUERY`: execute the query with input values

There are some limitations to Interpreted Mode. See the section on `INTERPRET QUERY` and the appendix section Interpreted GSQL Limitations.

`CREATE QUERY`

```
createQuery := CREATE [OR REPLACE] [DISTRIBUTED] QUERY queryName
               "(" [parameterList] ")"
               [FOR GRAPH graphName]
               [RETURNS "("  baseType | accumType ")"]
               [API "(" stringLiteral ")"]
               [SYNTAX syntaxName]
               "{" queryBody "}"

queryBody := [typedefs] [declStmts] [declExceptStmts] queryBodyStmts
```

`CREATE QUERY` defines the functionality of a query on a given graph schema.

A query has a name, a parameter list, the name of the graph being queried, an optional `RETURNS` type (see Section "`RETURN` Statement" for more details), optional specifiers for the output API and the language syntax version, and a body. The body consists of an optional sequence of `typedefs`, followed by an optional sequence of declarations, then followed by one or more statements. The body defines the behavior of the query.

> **Dynamic Query Support**
>
> As of TigerGraph 3.0+, **`FOR GRAPH graphName`** is optional, as long as the graph has been specified already, either when entering gsql:
> **`GSQL -g graphName [<gsql_command>]`**
> or once inside the GSQL shell, by using the **`USE GRAPH graphName`** command.
> This is one aspect of Dynamic Querying.

If the optional keywords `OR REPLACE` are included, then this query definition, if error-free, will replace a previous definition with the same query name. The new query will not be installed. That is,
`CREATE OR REPLACE QUERY name`
acts like
`DROP QUERY name`
`CREATE QUERY name`
However, if there are any errors in this query definition, then the previous query definition will be maintained. If the `OR REPLACE` option is not used, then GSQL will reject a `CREATE QUERY` command that uses an existing name.

The `DISTRIBUTED` option applies only to installations where the graph has been distributed across a cluster. If specified, the query will run with a different execution model which may give better performance for queries that traverse a large portion of the cluster. For details, see Distributed Query Mode.

Typedefs allow the definition of custom types for use within the body. The declarations support the definition of *accumulators* (see Chapter "Accumulators" for more details) and global/local variables. All accumulators and global variables must be declared before any statements. There are various types of statements that can be used within the body. Typically, the core statement(s) in the body of a query is one or more `SELECT`, `UPDATE`, `INSERT`, `DELETE` statements. The language supports conditional statements such as an `IF` statement as well as looping constructs such as `WHILE` and `FOREACH`. It also supports calling functions, assigning variables, printing, and modifying the graph data.

The query body may include calls to other queries. That is, the other queries are treated as subquery functions. See the subsection on "Queries as Functions".

Example of a CREATE QUERY statement

```
CREATE QUERY createQueryEx (STRING uid) FOR GRAPH socialNet RETURNS (int)
  SYNTAX v2 {
  # declaration statements
  users = {person.*};
  # body statements
  posts = SELECT p
    FROM users:u-(posted)->:p
    WHERE u.id == uid;
  PRINT posts;
  RETURN posts.size();
}
```

Query parameter and return types

This table lists the supported data types for input parameters and return values.

Type	Supported Data Types
Parameter Types	• Any base type (except EDGE or JSONOBJECT): INT, UINT, FLOAT, DOUBLE, STRING, BOOL, STRING, DATETIME, VERTEX, JSONARRAY • SET<baseType>, BAG<baseType> ◦ Edge and JSONOBJECT
Return Types	• any baseType (including EDGE): INT, UINT, FLOAT, DOUBLE, STRING, BOOL, STRING, VERTEX, EDGE, JSONOBJECT, JSONARRAY • any accumulator type, except GroupByAccum
API (JSON output format)	Currently, the only option is "v2" (default)
SYNTAX	v1 (default) or v2 (pattern matching). See the SELECT Statement section for an outline

> of the differences. See Pattern Matching

Default parameter values

You can specify default values for parameters of primitive types when creating a query. Primitive types include:

- `INT`
- `UINT`
- `FLOAT`
- `DOUBLE`
- `STRING`
- `BOOL`
- `STRING`
- `DATETIME`

To specify the default value for a parameter, use the assignment operator (`=`) after the parameter name and specify the default value:

Example of a CREATE QUERY command with a default parameter value

```
CREATE QUERY createQueryEx (STRING uid = "Tom") FOR GRAPH socialNet RETUR
  SYNTAX v2 {
  # declaration statements
  users = {person.*};
  # body statements
  posts = SELECT p
    FROM users:u-(posted)->:p
    WHERE u.id == uid;
  PRINT posts;
  RETURN posts.size();
}
```

Dynamic querying

TigerGraph 3.0+ supports Dynamic Querying. This means the query can be written and installed as a saved procedure without referencing a particular graph. Schema details -- the name of the graph, vertex types, edge types, and attributes -- can all be parameterized and only need to be specified at run time.

Here are the ingredients for a dynamic query:

- **Graph name:** When creating a query, `FOR GRAPH graphName` is optional, as long as the graph has been specified already, either when entering gsql:
  `GSQL -g graphName [<gsql_command>]`
  or once inside the GSQL shell, by using the `USE GRAPH graphName` command.
- **Vertex type and edge type in `SELECT` statements**. Typically, the `FROM` clause mentions the name of specific vertex types and edge types. String or string set parameters can be used for edge and target types instead.
- **Attribute names**. The `getAttr` and setAttr functions, which take attribute name and data type as string parameters, can be used to parameterize attribute access.
- `INSERT` **statements**: If you are using `INSERT` to add data to your graph, you need to specify what type of vertex or edge you want to add. This can also be parameterized.

Here is a simple example to demonstrate how to apply Dynamic GSQL Query techniques. Here is the PageRank algorithm from our GSQL Graph Algorithm library. Here is it written with schema information embedded statically in the query:

- graph name = social
- vertex type = Page
- edge type = Link
- vertex attribute = Score

```
CREATE QUERY pageRank (FLOAT maxChange=0.00, INT maxIter=25,
  FLOAT damping=0.85)  //parameters
  FOR GRAPH gsql_demo
{
  MaxAccum<float> @@maxDiff = 9999;
  SumAccum<float> @rcvd_score = 0;
  SumAccum<float> @score = 1;

  Start = {Page.*};
  WHILE @@maxDiff > maxChange LIMIT maxIter DO
      @@maxDiff = 0;
      V = SELECT s
          FROM Start:s -(Linkto:e)- Page:t       //hardcoded types
          ACCUM t.@rcvd_score += s.@score/(s.outdegree("Linkto")) //Param
          POST-ACCUM s.@score = (1.0-damping) + damping * s.@rcvd_score,
                      s.@rcvd_score = 0,
                      @@maxDiff += abs(s.@score - s.@score');
  END;

  V = SELECT s FROM Start:s
      POST-ACCUM s.Score = s.@score;    //hardcoded attribute
}
RUN QUERY pageRank(_,_,_)
```

Here is the same algorithm written in Dynamic Querying style:

```
CREATE QUERY pageRankDyn (FLOAT maxChange=0.00, INT maxIter=25,
  FLOAT damping=0.85,
  STRING vType, STRING eType, STRING attr)  //parameterized schema
//no graph name
{
  MaxAccum<float> @@maxDiff = 9999;
  SumAccum<float> @rcvd_score = 0;
  SumAccum<float> @score = 1;

  Start = {vType};
  WHILE @@maxDiff > maxChange LIMIT maxIter DO
      @@maxDiff = 0;
      V = SELECT s
          FROM Start:s -(eType:e)- vType:t //parameterized
          ACCUM t.@rcvd_score += s.@score/(s.outdegree(eType)) //param
          POST-ACCUM s.@score = (1.0-damping) + damping * s.@rcvd_score,
                       s.@rcvd_score = 0,
                       @@maxDiff += abs(s.@score - s.@score');
  END;

  V = SELECT s FROM Start:s
      POST-ACCUM s.setAttr(attr, s.@score); //parameterized
}
RUN QUERY pageRankDyn(_,_,_,"Page", "Link", "Score")
```

Statement types

A *statement* is a standalone instruction that expresses an action to be carried out. The most common statements are ***data manipulation language (DML) statements***. DML statements include the `SELECT`, `UPDATE`, `INSERT INTO`, `DELETE FROM`, and `DELETE` statements.

A GSQL query has two levels of statements. The upper-level statement type is called *query-body-level statement*, or ***query-body statement*** for short. This statement type is part of either the top-level block or a query-body control flow block. For example, each of the statements at the top level directly under `CREATE QUERY` is a query-body statement. If one of the statements is a `CASE` statement with several `THEN` blocks, each of the statements in the `THEN` blocks is also a query-body statement. Each query-body statement ends with a semicolon.

The lower-level statement type is called *DML-sub-level statement* or ***DML-sub statement*** for short. This statement type is used inside certain query-body DML statements, to define particular data manipulation actions. DML-sub-statements are comma-separated. There is no comma or semicolon after the last DML-sub-statement in a block. For example, one of the top-level statements is a `SELECT` statement, each of the statements in its `ACCUM` clause is a DML-sub-statement. If one of those DML-sub-statements is a `CASE` statement, each of the statement in the `THEN` blocks is a DML-sub-statement.

There is some overlap in the types. For example, an assignment statement can be used either at the query-body level or the DML-sub-level.

```
queryBodyStmts := (queryBodyStmt ";")+

queryBodyStmt := assignStmt           // Assignment
               | vSetVarDeclStmt      // Declaration
               | gAccumAssignStmt     // Assignment
               | gAccumAccumStmt      // Assignment
               | lAccumAccumStmt      // Assignment
               | funcCallStmt         // Function Call
               | selectStmt           // Select
               | queryBodyCaseStmt    // Control Flow
               | queryBodyIfStmt      // Control Flow
               | queryBodyWhileStmt   // Control Flow
               | queryBodyForEachStmt // Control Flow
               | BREAK                // Control Flow
               | CONTINUE             // Control Flow
               | updateStmt           // Data Modification
               | insertStmt           // Data Modification
               | queryBodyDeleteStmt  // Data Modification
               | printStmt            // Output
               | printlnStmt          // Output
               | logStmt              // Output
               | returnStmt           // Output
               | raiseStmt            // Exception
               | tryStmt              // Exception

DMLSubStmtList := DMLSubStmt ["," DMLSubStmt]*

DMLSubStmt := assignStmt           // Assignment
            | funcCallStmt         // Function Call
            | gAccumAccumStmt      // Assignment
            | lAccumAccumStmt      // Assignment
            | attrAccumStmt        // Assignment
            | vAccumFuncCall       // Function Call
            | localVarDeclStmt     // Declaration
            | DMLSubCaseStmt       // Control Flow
            | DMLSubIfStmt         // Control Flow
            | DMLSubWhileStmt      // Control Flow
            | DMLSubForEachStmt    // Control Flow
            | BREAK                // Control Flow
            | CONTINUE             // Control Flow
            | insertStmt           // Data Modification
            | DMLSubDeleteStmt     // Data Modification
            | printlnStmt          // Output
            | logStmt              // Output
```

Guidelines for understanding statement type hierarchy:

- Top-level statements are Query-Body type (each statement ending with a semicolon).
- The statements within a DML statement are DML-sub statements (comma-separated list).
- The blocks within a Control Flow statement have the same type as the entire Control Flow statement itself.

Schematic illustration of relationship between queryBodyStmt and DMLSubStmt

```
# Each statement's operation type is either ControlFlow, DML, or other.
# Each statement's syntax type is either queryBodyStmt or DMLSubStmt.

CREATE QUERY stmtTypes (parameterList) FOR GRAPH g [
    other queryBodyStmt1;
    ControlFlow queryBodyStmt2   # ControlFlow inside top level.
        other queryBodyStmt2.1;      # subStmts in ControlFlow are queryB
        ControlFlow queryBodyStmt2.2 # ControlFlow inside ControlFlow ins
            other queryBodyStmt2.2.1;
            other queryBodyStmt2.2.2;
        END;
        DML queryBodyStmt2.3     # DML inside ControlFlow inside top-leve
            other DMLSubStmt2.3.1,   # switch to DMLSubStmt
            other DMLSubStmt2.3.2
        ;
    END;
    DML queryBodyStmt3           # DML inside top level.
        other DMLSubStmt3.1,      # All subStmts in DML must be DMLSubStm
        ControlFlow DMLSubStmt3.2 # ControlFlow inside DML inside top lev
            other DMLSubStmt3.2.1,
            other DMLSubStmt3.2.2
        ,
        DML DMLsubStmt3.3
            other DMLSubStmt3.3.1,
            other DMLSubStmt3.3.2
    ;
    other queryBodyStmt4;
```

Here is a descriptive list of query-body statements:

EBNF term	Common Name	Description
assignStmt	Assignment Statement	See "Declaration and Assignment Statements"

vSetVarDeclStmt	Vertex Set Variable Declaration Statement	See "Declaration and Assignment Statements"
gAccumAssignStmt	Global Accumulator Assignment Statement	See "Declaration and Assignment Statements"
gAccumAccumStmt	Global Accumulator Accumulation Statement	See "Declaration and Assignment Statements"
lAccumAccumStmt	Local Accumulator Accumulation Statement	See "Declaration and Assignment Statements"
funcCallStmt	Functional Call or Query Call Statement	See "Declaration and Assignment Statements"
selectStmt	SELECT Statement	See "SELECT Statement"
queryBodyCaseStmt	query-body CASE statement	See "Control Flow Statements"
queryBodyIfStmt	query-body IF statement	See "Control Flow Statements"
queryBodyWhileStmt	query-body WHILE statement	See "Control Flow Statements"
queryBodyForEachStmt	query-body FOREACH statement	See "Control Flow Statements"
updateStmt	UPDATE Statement	See "Data Modification Statements"
insertStmt	INSERT INTO statement	See "Data Modification Statements"
queryBodyDeleteStmt	Query-body DELETE Statement	See "Data Modification Statements"
printStmt	PRINT Statement	See "Output Statements"
logStmt	LOG Statement	See Output Statements"
returnStmt	RETURN Statement	See "Output Statements"
raiseStmt	PRINT Statement	See "Exception Statements"
tryStmt	TRY Statement	See "Exception Statements"

Here is a descriptive list of DML-sub-statements:

EBNF term	Common Name	Description
assignStmt	Assignment Statement	See "Declaration and Assignment Statements"
funcCallStmt	Functional Call Statement	See "Declaration and Assignment Statements"
gAccumAccumStmt	Global Accumulator Accumulation Statement	See "Declaration and Assignment Statements"
lAccumAccumStmt	Local Accumulator Accumulation Statement	See "Declaration and Assignment Statements"
attrAccumStmt	Attribute Accumulation Statement	See "Declaration and Assignment Statements"
vAccumFuncCall	Vertex-attached Accumulator Function Call Statement	See "Declaration and Assignment Statements"
localVarDeclStmt	Local Variable Declaration Statement	See "SELECT Statement"
insertStmt	INSERT INTO Statement	See "Control Flow Statements"
DMLSubDeleteStmt	DML-sub DELETE Statement	See "Data Modification Statements"
DMLSubcaseStmt	DML-sub CASE statement	See "Data Modification Statements"
DMLSubIfStmt	DML-sub IF statement	See "Data Modification Statements"
DMLSubForEachStmt	DML-sub FOREACH statement	See "Data Modification Statements"
DMLSubWhileStmt	DML-sub WHILE statement	See "Data Modification Statements"
logStmt	LOG Statement	See "Output Statements"

`INTERPRET QUERY`

`INTERPRET QUERY` runs a query by translating it line-by-line. This is in contrast to the 2-step flow: (1) `INSTALL` to pre-translate and optimize a query, then (2) `RUN` to execute the installed query. The basic trade-off between `INTERPRET QUERY` and `INSTALL/RUN QUERY` is as follows:

- `INTERPRET`:
  - Starts running immediately but may take longer to finish than running an installed query.
  - Suitable for ad hoc exploration of a graph or when developing and debugging an application, and rapid experimentation is desired.
  - Supports most but not all of the features of the full GSQL query language. See the Appendix section Interpreted GSQL Limitations.
- `INSTALL` / `RUN`:
  - Takes up to a minute to `INSTALL`.
  - Runs faster than `INTERPRET`, from only a few percent faster to twice as fast.
  - Should always be used for production environments with fixed queries.

There are two GSQL syntax options for Interpreted GSQL: Immediate mode and Saved-query mode. In addition there is also a predefined RESTful endpoint for running interpreted GSQL: `POST /gsqlserver/interpreted_query`. The query body is sent as the payload of the request. The syntax is like the Immediate query option, except that it is possible to provide parameters, using the query string of the endpoint's request URL. The example below shows a parameterized query using the `POST /gsqlserver/interpreted_query` endpoint. For more details, see the RESTPP API User Guide.

Interpreted GSQL REST Endpoint with Immediate Query

```
curl --user tigergraph:tigergraph -X POST 'localhost:14240/gsqlserver/int
  INTERPRET QUERY (int a) FOR GRAPH gsql_demo {
    PRINT a;
  }
'
```

Immediate mode: define and interpret

interpret-anonymous-query syntax

```
interpretQuery := INTERPRET QUERY "(" ")"
                  [FOR GRAPH graphName]
                  [SYNTAX syntaxName]
                  "{" queryBody "}"
```

This syntax is similar in concept to SQL queries. Queries are not named, do not accept parameters, and are not saved after being run. Syntax differences from compiled GSQL:

- The keyword `CREATE` is replaced with `INTERPRET`.
- The query is executed immediately by the `INTERPRET` statement. The `INSTALL` and `RUN` statements are not used.
- Parameters are not accepted.

Compare the example below to the example in the `Create Query` section:

- No query name, no parameters, no `RETURN` statement.
- Because no parameter is allowed, the parameter `uid` is set within the query.

Example of Immediate Mode for INTERPRET QUERY

```
INTERPRET QUERY () FOR GRAPH socialNet {
  # declaration statements
  STRING uid = "Jane.Doe";
  users = {person.*};
  # body statements
  posts = SELECT p
    FROM users:u-(posted)->:p
    WHERE u.id == uid;
  PRINT posts, posts.size();
}
```

Interpret a created query

interpret-saved-query syntax

```
runQuery := (RUN | INTERPRET) QUERY [runOptions] queryName "(" parameterV
```

This syntax is like `RUN` query, except

1. The keyword `RUN` is replaced with `INTERPRET`.
2. Some options may not be supported.

Example of Interpret-Only Mode for INTERPRET QUERY

```
INTERPRET QUERY createQueryEx ("Jane.Doe")
```

`INSTALL QUERY`

```
installQuery := INSTALL QUERY [installOptions] ( "*" | ALL | queryName ["
```

`INSTALL QUERY` installs a query or multiple queries on a graph. Installing a query compiles the procedures described by the query as well as generates a REST endpoint for running the query.

Installing a query allows the query to be run through the `RUN QUERY` command as well as through its REST endpoint, both offering stronger performance as compared to running the query through the `INTERPRET QUERY` command. The `INSTALL QUERY` command will install the queries specified, with query names separated by a comma.

If a query calls a subquery, the query can only be installed after one of the following conditions is met:

- The subquery has already been installed
- The subquery is being installed in the same `INSTALL QUERY` command as the query itself

If a subquery that was previously installed is dropped from the graph, all installed queries that call the subquery will be disabled. To re-enable a disabled query, all its subqueries need to be installed with the same parameters and return type.

When a single `INSTALL QUERY` command installs multiple queries, each query is installed independently. If one query fails to be installed, it will not affect the installation of other queries.

To install a query, the user needs to have the `WRITE_QUERY` privilege on the graph where the query is to be installed or on the global scope.

Users can also install all uninstalled queries on a graph with `INSTALL QUERY`, using either of the following commands:

- `INSTALL QUERY *`
- `INSTALL QUERY ALL`

> Installing takes several seconds for each query. The current version does not support concurrent installation and running of queries. Other concurrent graph operations will be delayed until the installation finishes.
>
> Concurrent `INSTALL QUERY` commands are allowed as long as only one `INSTALL QUERY` command is running on a single graph. Concurrent `INSTALL QUERY` commands are *not* allowed on a single graph.

Options for `INSTALL QUERY`

The following options are available:

Option	Effect
`-FORCE`	Reinstall the query even if the system indicates the query is already installed. This is useful for overwriting an installation that is corrupted or otherwise outdated, without having to drop and then recreate the query. If this option is not used, the GSQL shell will refuse to re-install a query that is already installed.

`-DISTRIBUTED`	If you have a distributed database deployment, installing the query in Distributed Query Mode can increase performance for single queries - using a single worker from each available machine to yield results. Certain cases may benefit more from this option than others -- more detailed information is available on the next

Optimize installed queries

`INSTALL QUERY -OPTIMIZE`

Users can run `INSTALL QUERY -OPTIMIZE` to optimize all installed queries. The names of the individual queries are not needed. This operation optimizes all previously installed queries, reducing their run times by about 20%. Optimize a query if query run time is more important to you than query installation time.

RUN QUERY

The `RUN QUERY` command runs an installed query. To run a query with the `RUN QUERY` command, specify the query name, followed by the query parameters enclosed in parentheses. Running a query executes all statements in the query body and produces output as specified by the output statements in the query.

You can also run an installed query through REST requests - see Run an installed query.

Syntax

RUN QUERY syntax

```
runQuery := (RUN | INTERPRET) QUERY [runOptions] queryName
    "(" parameterValueList | parameterValueJSON ")"

runOptions := ( "-av" | "-d" )*
parameterValueList := parameterValue ["," parameterValue]*
parmeterValueJSON ::= '{"'parameterName'":' parameterValue(', "'parameter
```

Query parameters

There are two ways of passing parameters to a query in a `RUN QUERY` command:

- Pass parameters as an ordered list separated by commas
- Pass parameters by name in JSON

Parameter list

To pass parameters to a query with a list, the parameters must be put in the same order as they were in the query definition. Each value passed in will correspond to the parameter at the same index when the query was created.

To use the default value for a parameter, use the `_` character for the value of the parameter. You can also omit parameters to use their default value. However, if you omit one parameter, you also have to omit all parameters that come after that parameter.

For example, if we have the following query definition:

```
CREATE QUERY greetPerson(INT age = 3, STRING name = "John",
  DATETIME birthday = to_datetime("2019-02-19 19:19:19"))
{
  PRINT age, name, birthday;
}
```

To run the query with default values for the parameter `name,` use `_` in the place of the second parameter value:

```
GSQL > RUN QUERY greetPerson (21, _, "2020-02-02 20:02:20")
# Below is the query output
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{
    "birthday": "2020-02-02 20:02:20",
    "name": "John",
    "age": 21
  }]
}
```

To use the default values for both the second and the third parameters, you can omit both parameters and only provide a value for the first parameter.

```
GSQL > RUN QUERY greetPerson(21)
# Below is the query output
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{
    "birthday": "2019-02-19 19:19:19",
    "name": "John",
    "age": 21
  }]
}
```

Parameter JSON object

To pass query parameters by name with a JSON object, map the parameter names to their values in a JSON object enclosed in parentheses. Parameters that are not named in the JSON object will keep their default values for the execution of the query.

For example, if we have the following query:

```
CREATE QUERY greetPerson(INT age = 3, STRING name = "John",
  DATETIME birthday = to_datetime("2019-02-19 19:19:19"))
{
  PRINT age, name, birthday;
}
```

Supplying the parameters with a JSON object will look like the following. The parameter `birthday` is not named in the parameter JSON object and therefore takes the default value:

```
RUN QUERY greetPerson( {"name": "Emma", "age": 21} )
```

Complex type parameter passing

This subsection describes how to format the complex type parameter values when executing a query by `RUN QUERY`. More details about all parameter types are described in Section "Query Parameter Types".

Parameter list

Parameter type	Syntax	Example
`DATETIME`	Use a string formatted as `"YYYY-MM-DD HH-MM-SS"`	`"2019-02-19 19:19:19"`
Set or bag of primitives	Use square brackets to enclose the collection of values.	A set of integers: `[1,5,10]`
`VERTEX<type>`	If the vertex type is specified in the query definition, then the vertex argument is **`vertex_id`**	The vertex type is `person` and the desired ID is `person2`. `"person2"`

`VERTEX` (type not pre-specified)	If the type is not defined in the query definition, then the argument must provide both the id and type in parentheses:**(vertex_id, vertex_type)**	A vertex with ID `"person1"` and `type="person`: `("person1","person")`
Set or bag of `VERTEX<type>`	Same as a SET or BAG of primitives, where the primitive type is vertex_id.	`[ "person3", "person4" ]`
Set or bag of `VERTEX` (type not pre-specified)	Same as a SET or BAG of vertices, with vertex type not pre-specified. Square brackets enclose a comma-separated list of vertex (id, type) pairs. Mixed types are permitted	`[ ("person1","person"), ("11","post") ]`

Parameter JSON object

Parameter type	Syntax	Example
`DATETIME`	Use a string formatted as `"YYYY-MM-DD HH-MM-SS"`	`"2019-02-19 19:19:19"`
Set or bag of primitives	Use a JSON array containing the primitive values	`["a", "list", "of", "args"]`
`VERTEX<type>`	Use a JSON object containing a field `"id"` for the vertex ID and a field `"type"` for the type of the vertex	`{"id": "person1",` `"type": "person"}`
`VERTEX` (type not specified)	Use a JSON object containing a field `"id"` for the vertex ID	`{"id": "person1"}`
Set or bag of `VERTEX<type>`	Use a JSON array containing a list of JSON `VERTEX<type>` object	`[{"id": "person1"}, {"id": "person2"}]`
		`[{"id": "person1",`

Set or bag of vertices of unspecified types	Use a JSON array containing a list of JSON `VERTEX`	`"type": "person"},{"id": "person2",` `"type": "person"}]`

Options

All-vertex mode -av option

Some queries run with all or almost all vertices in a SELECT statement s, e.g. PageRank algorithm. In this case, the graph processing engine can run much more efficiently in all-vertex mode. In the all-vertex mode, all vertices are always selected, and the following actions become ineffective:

- Filtering with selected vertices or vertex types. The source vertex set must be all vertices.
- Filtering with the WHERE clause.
- Filtering with the HAVING clause.
- Assigning designated vertex or designated type of vertexes. E.g. X = { *vertex_type* .*}

To run the query in all-vertex mode, use the -av option in shell mode or include `__GQUERY__USING_ALL_ACTIVE_MODE=true` in the query string of an HTTP request.

```
GSQL > RUN QUERY -av test()

## In a curl URL call.  Note the use of both single and double underscore
curl -X GET 'http://localhost:9000/query/graphname/queryname?__GQUERY__US
```

Detached mode -async option

Typically, the GSQL `RUN QUERY` command runs in the foreground and does not produce output until the query completes, which is inconvenient in the case of long-running queries. Starting with TigerGraph 3.1, you can run queries in Detached Mode to enable background execution of long-running queries.

Queries executed in Detached mode are still subject to the system timeout limit. The default timeout limit is 16 seconds and can be set using the `GSQL-TIMEOUT` header.

To run a query in Detached Mode from the command line, use the `-async` option for the `RUN QUERY` command:

```
GSQL > RUN QUERY -async <queryName>
```

You will receive a JSON response immediately containing a query ID (`request_id`):

```
{
  "error": false,
  "message": "The query is successfully submitted. Please check query sta
  "request_id": "<RequestID>"
}
```

To run queries in Detached Mode via RESTPP endpoint call, use the `GSQL-ASYNC` header and set its value to true. If the query takes parameters, put them in the query string:

```
$ curl -s -H "GSQL-ASYNC:true" GET "http://localhost:9000/query/<graphNam
```

To check the status and results of the queries executed in Detached Mode, use the `/query_status` and the `/query_result` RESTPP endpoints.

GSQL Query output format

The standard output of GSQL queries is in industry-standard JSON format. A JSON **object** is an unordered set of **key-value pairs**, enclosed in curly braces. Among the acceptable data types for a JSON **value** are **array** and **object**. A JSON **array** is an ordered list of **values**, enclosed in square brackets. Since values can be objects or arrays, JSON supports hierarchical, nested structures. Strings are enclosed in double quotation marks. We also use the term **field** to refer to a key (or a key-value pair) of a given object.

At the top level of the JSON structure are four required fields ("version", "error", "message", and "results") and one dependent field ("code"). If a query is successful, the value of "error" will be "false", the "message" value will be empty, and the "results" value will be the intended output of the query. If an error or exception occurred during query execution, the "error" value will be "true", the "message" value will be a string message describing the error condition, and the "results" field will be empty. Also, the "code" field will contain an error code.

Beginning with version 2 (v2) of the output specification, an additional top-level field is required: `"version"`. The `"version"` value is an object with the following fields:

Field	Description
`api`	String specifying the output API version. Values are specified as follows: • "v1": Output API used in TigerGraph platform v0.8 through v1.0. **NOTE: "v1" support is no longer available as of TigerGraph v3.0.** • "v2" (default): Output API introduced in TigerGraph platform v1.1 This is the latest API.
`edition`	String indicating the edition of the product.
`schema`	Integer representing which version of the user's graph schema is currently in use. When a `CREATE GRAPH` statement is executed, the version is initialized to 0. Each time a `SCHEMA_CHANGE JOB` is run, the schema value is incremented by 1 (e.g., 1, 2, etc.).

Other top-level objects, such as "code" may appear in certain circumstances. Note that the top-level objects are enclosed in curly braces, meaning that they form an unordered set. They may appear in any order.

Below is an example of the output of a successful query:

Top Level JSON of a Valid Query - Example

```
{
  "version": {"edition": "developer","api": "v2","schema": "1"},
  "error": false,
  "message": "",
  "results": [
    {results_of_PRINT_statement_1},
    ...,
    {results_of_PRINT_statement_N}
  ]
}
```

The value of the "results" key-value pair is a sequential list of the data objects specified by the PRINT statements of the query. The list order follows the order of PRINT execution. The detailed format of the PRINT statement results is described in Output Statements and FILE Objects.

The following REST response misspells the name of the endpoint

GET echo/ Request and Response

```
curl -X GET "http://localhost:9000/eco"
```

and generates the following output:

```
{
  "version": {"edition":"developer","api":"v2","schema":0},
  "error": true,
  "message": "Endpoint is not found from url = /eco, please use GET /endp
  "code": "REST-1000"
}
```

Changing the default output API

The following GSQL statement can be used to set the JSON output API configuration.

```
SET json_api = <version_string>
```

This statement sets a persistent system parameter. Each version of the TigerGraph platform is pre-configured to what was the latest output API that at the time of release. For example, platform version 1.1 is configured so that each query will produce v2 output by default.

As of TigerGraph v3.0, the only supported JSON API is "v2".

SHOW QUERY

```
showQuery := SHOW QUERY queryName
```

Additionally, the `ls` GSQL command lists all created queries and identifies which queries have been installed.

DROP QUERY

```
dropQuery := DROP QUERY ( "*" | ALL | queryName ["," queryName]* )
```

To drop a query, run `DROP QUERY query_name` . The query will be uninstalled (if it has been installed) and removed from the dictionary. The GSQL language will refuse to drop an installed query if another query is installed which calls query Q. That is, all calling queries must be dropped before or at the same time that their called subqueries are dropped.

To drop all queries, either of the following commands can be used:
`DROP QUERY ALL`
`DROP QUERY *`

> The scope of ALL depends on the user's current scope. If the user has set a working graph, then DROP ALL removes all the jobs for that graph. If a superuser has set their

scope to be global, then DROP ALL removes all jobs across all graph spaces.

Loading...

Data Types

This section describes the data types that are native to and are supported by the GSQL Query Language. Most of the data objects used in queries come from one of three sources:

- The query's input parameters
- The vertices, edges, and their attributes which are encountered when traversing the graph
- The variables defined within the query to assist in the computational work of the query

This section covers the following subset of the EBNF language definitions:

EBNF for Data Types

```
lowercase          := [a-z]
uppercase          := [A-Z]
letter             := lowercase | uppercase
digit              := [0-9]
integer            := ["-"]digit+
real               := ["-"]("."digit+) | ["-"](digit+"."digit*)
numeric            := integer | real
stringLiteral      := '"' [~["] | '\\' ('"' | '\\')]* '"'

name := (letter | "_") [letter | digit | "_"]*   // Can be a single "_" o
graphName := name
queryName := name
paramName := name
vertexType := name
edgeType := name
accumName := name
vertexSetName := name
attrName := name
varName := name
tupleType := name
fieldName :=name
funcName := name

type := baseType | tupleType | accumType | STRING COMPRESS

baseType := INT
          | UINT
          | FLOAT
          | DOUBLE
          | STRING
          | BOOL
          | VERTEX ["<" vertexType ">"]
          | EDGE
          | JSONOBJECT
          | JSONARRAY
          | DATETIME

filePath := paramName | stringLiteral

typedef := TYPEDEF TUPLE "<" tupleFields ">" tupleType

tupleFields := (baseType fieldName) | (fieldName baseType)
           ["," (baseType fieldName) | (fieldName baseType)]*

parameterType := baseType
               | [ SET | BAG ] "<" baseType ">"
               | FILE
```

Identifiers

An identifier is the name for an instance of a language element. In the GSQL query language, identifiers are used to name elements such as a query, a variable, or a user-defined function. In the EBNF syntax, an identifier is referred as `name`. It can be a sequence of letters, digits, or underscores (`"_"`). Other punctuation characters are not supported. The initial character can only be a letter or an underscore.

name (identifier)

```
name := (letter | "_") [letter | digit | "_"]*
```

Overview of Types

Different types of data can be used in different contexts. The EBNF syntax defines several classes of data types. The most basic is called base type (`baseType`). The other independent types are `FILE` and `STRING COMPRESS`. The remaining types are either compound data types built from the independent data types, or supersets of other types. The table below gives an overview of their definitions and their uses.

EBNF term	Description	Use Case
`baseType`	`INT`, `UINT`, `FLOAT`, `DOUBLE`, `STRING`, `BOOL`, `DATETIME,` `VERTEX`, `EDGE`, `JSONOBJECT`, or `JSONARRAY`	• Global variable • Query return value
`tupleType`	Sequence of base types	• User-defined tuple
`accumType`	Family of specialized data objects which support accumulation operations	• Accumulate and aggregate data, when traversing a set of vertices or edges (Details are in the

		Query Lang Spec - Accumulators chapter.)
`FILE`	`FILE` object	• Global sequential data object, linked to a text file
`parameterType`	`baseType` (except `EDGE` or `JSONOBJECT`), a set or bag of `baseType`, or `FILE` object	• Query parameter
`STRING COMPRESS` (⚠**suitable only in limited circumstances**)	STRING COMPRESS	• more compact storage of STRING, **if there is a limited number of different values and the value is rarely accessed. Otherwise, it may use more memory.**
`elementType`	`baseType`, `STRING COMPRESS`, or identifier	• Element for most types of container accumulators: `SetAccum`, `BagAccum`, `GroupByAccum`, key of a `MapAccum` element
`type`	`baseType`, `STRING COMPRESS`, identifier, or `accumType`	• Element of a `ListAccum`, value of a `MapAccum` element • Local variable

Base Types

The query language supports the following *base types*, which can be declared and assigned anywhere within their scope. Any of these base types may be used when defining a global variable, a local variable, a query return value, a parameter, part of a tuple, or an element of a container accumulator. Accumulators are described in detail in a later section.

EBNF

```
baseType := INT
          | UINT
          | FLOAT
          | DOUBLE
          | STRING
          | BOOL
          | VERTEX ["<" vertexType ">"]
          | EDGE
          | JSONOBJECT
          | JSONARRAY
          | DATETIME
```

The default value of each base type is shown in the table below. The default value is the initial value of a base type variable (see Section "Variable Types" for more details), or the default return value for some functions (see Section "Operators, Functions, and Expressions" for more details).

The first seven types (`INT`, `UINT`, `FLOAT`, `DOUBLE`, `BOOL`, `STRING`, and `DATETIME`) are the same ones mentioned in the "Attribute Data Types" section of ***GSQL Language Reference, Part 1***.

Type	Default value	Literal
`INT`	`0`	A signed integer: `-3`
`UINT`	`0`	An unsigned integer: `5`
`FLOAT`	`0`	A decimal: `3.14159`
`DOUBLE`	`0`	A decimal with greater precision than `FLOAT`
`BOOL`	`false`	`TRUE` or `FALSE`
`STRING`	`""`	Characters enclosed by double quotes: `"Hello"`
`DATETIME`	`1970-01-01 00:00:00`	No literal. Can be converted from a correctly formatted string with `to_datetime()`.
`VERTEX`	`"Unknown"`	No literal.
`EDGE`	No edge: `{}`	No literal.

`JSONOBJECT`	An empty object: `{}`	No literal. Can be converted from a correctly formatted string with `parse_json_object()`.
`JSONARRAY`	An empty array: `[]`	No literal. Can be converted from a correctly formatted string with `parse_json_array()`.

`FLOAT` and `DOUBLE` input values must be in fixed point `d.dddd` ****format, where `d` is a digit. Output values will be printed in either fixed point for exponential notation, whichever is more compact.

The GSQL Loader can read FLOAT and DOUBLE values with exponential notation (e.g., 1.25 E-7).

Vertex

`VERTEX` is considered a base type in the GSQL query language. Both query parameters and variables in a query body can be of type `VERTEX`.

Vertex types

A graph schema defines specific vertex types. Each vertex type has its own set of attributes. The parameter or variable type can be restricted by giving the vertex type in angle brackets `<>` after the keyword `VERTEX`. A vertex variable declared without a specifier is called a *generic* vertex variable.

```
VERTEX anyVertex;
VERTEX<person> owner;
```

All vertices have a built-in attribute `type`. The built-in attribute is of type string. You can access it with the dot (`.`) operator.

For example, if you declare a vertex variable `VERTEX<person> personVertex`, then `personVertex.type` returns `"person"`.

Edge

`EDGE` is considered a base type in the GSQL query language. Both query parameters and variables in a query body can be of type `EDGE`.

Edge types

A graph schema defines specific edge types. Each edge type has its own set of attributes. The parameter or variable type can be restricted by giving the edge type in angle brackets `<>` after the keyword `EDGE`. An edge variable declared without a specifier is called a *generic* edge variable.

```
EDGE anyEdge;
EDGE<friendship> friendEdge;
```

All edges have a built-in attribute `type`. The built-in attribute is of type string. You can access it with the dot (`.`) operator.

For example, if you define an edge variable `EDGE<friendship> friendEdge`, then `friendEdge.type` returns `"Friendship"`.

Vertex and Edge Attribute Types

The following table maps vertex or edge attribute types in the Data Definition Language (DDL) to GSQL query language types. If an attribute of a vertex or edge is referenced in a GSQL query, they will be automatically converted to their corresponding data type in the GSQL query language.

DDL	GSQL Query
`INT`	`INT`
`UINT`	`UINT`
`FLOAT`	`FLOAT`

`DOUBLE`	`DOUBLE`
`BOOL`	`BOOL`
`STRING`	`STRING`
`STRING COMPRESS`	`STRING`
`SET< type >`	`SetAccum< type >`
`LIST< type >`	`ListAccum< type >`
`MAP <key_type, value_type>`	`MapAccum <key_type, value_type>`

`SET` and `LIST` literals

In the GSQL query language, one cannot declare a variable of `SET` (vertex sets are an exception), `LIST` , or `MAP` types. However, one can still use `SET` and `LIST` literals to update the value of a vertex attribute of type `SET` or `LIST`, insert a vertex or edge with attributes of type `SET` or `LIST` , and initialize an accumulator.

```
// Elements within a set or a list need to be of the same type
set_literal := "(" expr ("," expr)* ")"
list_literal := "[" expr ("," expr)* "]"
expr := INT | UINT | FLOAT | DOUBLE | BOOL | STRING | UDT | DATETIME
```

Currently, GSQL query language syntax does not support `MAP` literals.

`JSONOBJECT` and `JSONARRAY`

These two base types allow users to pass a complex data object or to write output in a customized format. These types follow the industry-standard definition of JSON ↗. A `JSONOBJECT` instance's external representation (as input and output) is a string, starting and ending with curly braces (`{}`) which enclose an unordered list of key-value _****_pairs. A `JSONARRAY` is represented as a string, starting and ending with square brackets (`[]`)which enclose an ordered list of *values*. Since a _value ****_can be an object or an array, JSON supports hierarchical, nested data structures.

More details are introduced in the Section [JSONOBJECT and JSONARRAY Functions](#).

A `JSONOBJECT` or `JSONARRAY` value is immutable. No operator is allowed to modify its value.

Tuple

A tuple is a user-defined data structure consisting of a fixed sequence of base type variables. Tuple types can be created and named using a `TYPEDEF` statement. Tuples must be defined first, before any other statements in a query.

ENBF for tuples

```
typedef := TYPEDEF TUPLE "<" tupleFields ">" tupleType

tupleFields := (baseType fieldName) | (fieldName baseType)
             ["," (baseType fieldName) | (fieldName baseType)]*
```

A tuple can also be defined in a graph schema and then can be used as a vertex or edge attribute type. A tuple type that has been defined in the graph schema does not need to be re-defined in a query.

The vertex type `person` contains two complex attributes:

- `secretInfo` of type `SECRET_INFO`, which a user-defined tuple
- `portfolio` of type `MAP<STRING, DOUBLE>`

investmentNet schema

```
TYPEDEF TUPLE <age UINT (4), mothersName STRING(20) > SECRET_INFO
CREATE VERTEX person(PRIMARY_ID personId STRING, portfolio MAP<STRING, DO
CREATE VERTEX stockOrder(PRIMARY_ID orderId STRING, ticker STRING, orderS
CREATE UNDIRECTED EDGE makeOrder(FROM person, TO stockOrder, orderTime DA
CREATE GRAPH investmentNet (*)
```

The query below reads both the `SECRET_INFO` tuple and the portfolio MAP. The tuple type does not need to redefine `SECRET_INFO`. To read and save the map, we define a `MapAccum` with the same types for key and value as the `portfolio` attribute. In addition, the query creates a new tuple type, `ORDER_RECORD`.

tupleEx query

```
CREATE QUERY tupleEx(VERTEX<person> p) FOR GRAPH investmentNet{
  #TYPEDEF TUPLE <UINT age, STRING mothersName> SECRET_INFO;       # alre
  TYPEDEF TUPLE <STRING ticker, FLOAT price, DATETIME orderTime> ORDER_RE

  SetAccum<SECRET_INFO> @@info;
  ListAccum<ORDER_RECORD> @@orderRecords;
  MapAccum<STRING, DOUBLE> @@portf;       # corresponds to MAP<STRING, DO

  INIT = {p};

  # Get person p's secret_info and portfolio
  X = SELECT v FROM INIT:v
      ACCUM @@portf += v.portfolio, @@info += v.secretInfo;

  # Search person p's orders to record ticker, price, and order time.
  # Note that the tuple gathers info from both edges and vertices.
  orders = SELECT t
      FROM INIT:s -(makeOrder:e)->stockOrder:t
      ACCUM @@orderRecords += ORDER_RECORD(t.ticker, t.price, e.orderTime

  PRINT @@portf, @@info;
  PRINT @@orderRecords;
}
```

tupleEx.json

```
GSQL > RUN QUERY tupleEx("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@info": [{
        "mothersName": "JAMES",
        "age": 25
      }],
      "@@portf": {
        "AAPL": 3142.24,
        "MS": 5000,
        "G": 6112.23
      }
    },
    {"@@orderRecords": [
      {
        "ticker": "AAPL",
        "orderTime": "2017-03-03 18:42:28",
        "price": 34.42
      },
      {
        "ticker": "B",
        "orderTime": "2017-03-03 18:42:30",
        "price": 202.32001
      },
      {
        "ticker": "A",
        "orderTime": "2017-03-03 18:42:29",
        "price": 50.55
      }
    ]}
  ]
}
```

STRING COMPRESS

`STRING COMPRESS` is an integer type encoded by the system to represent string values. `STRING COMPRESS` uses less memory than `STRING`. The `STRING COMPRESS` type is designed to act like `STRING`: data are loaded and printed just as string data, and most functions and operators which take `STRING` input can also take `STRING COMPRESS` input. The difference is in how the data are stored internally. A `STRING COMPRESS` value can be obtained from a `STRING_SET COMPRESS` or `STRING_LIST COMPRESS` attribute or from converting a `STRING` value.

> ⚠ Using `STRING COMPRESS` instead of `STRING` is a trade-off: smaller storage vs. slower access times. The storage space will only be smaller if (1) the original strings are long, and (2) there are only a small number of different strings. Performance will always be slower; the slowdown is greater if the `STRING COMPRESS` attributes are accessed more often.
>
> We recommend performing comparison tests for both performance and memory usage before settling on `STRING COMPRESS`.

`STRING COMPRESS` type is beneficial for sets of string values when the same values are used multiple times. In practice, `STRING COMPRESS` are most useful for container accumulators like `ListAccum<STRING COMPRESS>` or `SetAccum<STRING COMPRESS>`.

An accumulator containing `STRING COMPRESS` stores the dictionary when it is assigned an attribute value or from another accumulator containing `STRING COMPRESS`. An accumulator containing `STRING COMPRESS` can store multiple dictionaries. A `STRING` value can be converted to a `STRING COMPRESS` value only if the value is in the dictionaries. If the `STRING` value is not in the dictionaries, the original string value is saved. A `STRING COMPRESS` value can be automatically converted to a `STRING` value.

When a `STRING COMPRESS` value is output (e.g. by a [`PRINT` statement]), it is shown as a `STRING`.

> ⚠ `STRING COMPRESS` is not a base type.

STRING COMPRESS example

```
CREATE QUERY stringCompressEx(VERTEX<person> m1) FOR GRAPH workNet {
  ListAccum<STRING COMPRESS> @@strCompressList, @@strCompressList2;
  SetAccum<STRING COMPRESS> @@strCompressSet, @@strCompressSet2;
  ListAccum<STRING> @@strList, @@strList2;
  SetAccum<STRING> @@strSet, @@strSet2;

  S = {m1};

  S = SELECT s
      FROM S:s
      ACCUM @@strSet += s.interestSet,
            @@strList += s.interestList,
            @@strCompressSet += s.interestSet,   # use the dictionary fro
            @@strCompressList += s.interestList; # use the dictionary fro

  @@strCompressList2 += @@strCompressList;  # @@strCompressList2 gets the
  @@strCompressList2 += "xyz";   # "xyz" is not in the dictionary, so sto

  @@strCompressSet2 += @@strCompressSet;
  @@strCompressSet2 += @@strSet;

  @@strList2 += @@strCompressList;  # string compress integer values are
  @@strSet2 += @@strCompressSet;

  PRINT @@strSet, @@strList, @@strCompressSet, @@strCompressList;
  PRINT @@strSet2, @@strList2, @@strCompressSet2, @@strCompressList2;
}
```

stringCompressEx.json Results

```
GSQL > RUN QUERY stringCompressEx("person12")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@strCompressList": [
        "music",
        "engineering",
        "teaching",
        "teaching",
        "teaching"
      ],
      "@@strSet": [ "teaching", "engineering", "music" ],
      "@@strCompressSet": [ "music", "engineering", "teaching" ],
      "@@strList": [
        "music",
        "engineering",
        "teaching",
        "teaching",
        "teaching"
      ]
    },
    {
      "@@strSet2": [ "music", "engineering", "teaching" ],
      "@@strCompressList2": [
        "music",
        "engineering",
        "teaching",
        "teaching",
        "teaching",
        "xyz"
      ],
      "@@strList2": [
        "music",
        "engineering",
        "teaching",
        "teaching",
        "teaching"
      ],
      "@@strCompressSet2": [ "teaching", "engineering", "music" ]
    }
  ]
```

```
}
```

FILE Object

A `FILE` object is a sequential data storage object, associated with a text file on the local machine.

> When referring to a `FILE` object, we always capitalize the word `FILE` to distinguish it from ordinary files.

When a `FILE` object is declared, associated with a particular text file, any existing content in the text file will be erased. During the execution of the query, content written to the `FILE` will be appended to the `FILE`. When the query where the `FILE` was declared finishes running, the `FILE` contents are saved to the text file.

A `FILE` object can be passed as a parameter to another query. When a query receives a `FILE` object as a parameter, it can append data to that `FILE`, as can every other query which receives this `FILE` object as a parameter.

Query Parameter Types

Input parameters to a query can be base type (except `EDGE` , `JSONARRAY`, or `JSONOBJECT`). A parameter can also be a `SET` or `BAG` which uses base type (except `EDGE` , `JSONARRAY`, or `JSONOBJECT`) as the element type. A `FILE` object can also be a parameter. Within the query, `SET` and `BAG` are converted to `SetAccum` and `BagAccum`, respectively.

> A query parameter is immutable. It cannot be assigned a new value within the query.
>
> The `FILE` object is a special case. It is passed by reference, meaning that the receiving query gets a link to the original `FILE` object. The receiving query can write to the `FILE` object.

EBNF

```
parameterType := INT
               | UINT
               | FLOAT
               | DOUBLE
               | STRING
               | BOOL
               | VERTEX ["<" vertexType ">"]
               | DATETIME
               | [ SET | BAG ] "<" baseType ">"
               | FILE
```

Examples of collection type parameters

```
(SET<VERTEX<person> p1, BAG<INT> ids, FILE f1)
```

Accumulators

Accumulators are special types of variables that accumulate information about the graph during its traversal and exploration. Because they are a unique and important feature of the GSQL query language, we devote a separate section for their introduction, but additional detail on their usage will be covered in other sections, the "SELECT Statement" section in particular. This section covers the following subset of the EBNF language definitions:

EBNF

```
accumDeclStmt :=
          accumType localAccumName ["=" constant]
                    ["," localAccumName ["=" constant]]*
        | [STATIC] accumType globalAccumName ["=" constant]
                    ["," globalAccumName ["=" constant]]*
localAccumName := "@"accumName;
globalAccumName := "@@"accumName;


accumType := "SumAccum" "<" ( INT | FLOAT | DOUBLE | STRING | STRING COMP
       | "MaxAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
       | "MinAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
       | "AvgAccum"
       | "OrAccum"
       | "AndAccum"
       | "BitwiseOrAccum"
       | "BitwiseAndAccum"
       | "ListAccum" "<" type ">"
       | "SetAccum"  "<" elementType ">"
       | "BagAccum"  "<" elementType ">"
       | "MapAccum"  "<" elementType "," (baseType | accumType | tupleTyp
       | "HeapAccum" "<" tupleType ">" "(" simpleSize "," fieldName [ASC
                                         ["," fieldName [ASC | DESC]]* "
       | "GroupByAccum" "<" elementType fieldName ["," elementType fieldN
                            accumType fieldName ["," accumType fieldName
       | "ArrayAccum" "<" accumName ">"

elementType := baseType | tupleType | STRING COMPRESS

gAccumAccumStmt := globalAccumName "+=" expr

accumClause := ACCUM DMLSubStmtList

postAccumClause := "POST-ACCUM" DMLSubStmtList
```

There are a number of different types of accumulators, each providing specific accumulation functions. Accumulators are declared to have one of two types of association: *global* or *vertex-attached*.

More technically, accumulators are mutable mutex variables shared among all the graph computation threads exploring the graph within a given query. To improve performance, the graph processing engine employs multithreaded processing. Modification of accumulators is coordinated at run-time so the accumulation operator works correctly (i.e., mutually exclusively) across all threads. This is particularly relevant in the ACCUM clause. During traversal of the graph, the selected set of edges or vertices is partitioned among a group of threads. These threads have shared mutually exclusive access to the accumulators.

Declaration of Accumulators

Global accumulators can be declared anywhere in the query. Vertex-attached accumulators can be declared anywhere in the query except for in a `FOREACH` loop or `WHILE` loop. Accumulators are block-scoped and can only be accessed in the block where they are declared.

The name of a vertex-attached accumulator begins with a single `@`. The name of a global accumulator begins with `@@`. Additionally, a global accumulator may be declared to be static.

EBNF for Accumulator Declaration

```
accumDeclStmt :=
          accumType localAccumName ["=" expr]
                    ["," localAccumName ["=" expr]]*
        | [STATIC] accumType globalAccumName ["=" expr]
                    ["," globalAccumName ["=" expr]]*
localAccumName := "@"accumName;
globalAccumName := "@@"accumName;
```

Vertex-attached Accumulators

Vertex-attached accumulators are mutable state variables that are attached to each vertex in the graph for the duration of the query's lifetime. They act as run-time attributes of a vertex. They are shared, mutual exclusively, among all of the query's processes.

Vertex-attached accumulators can be set to a value with the = operator. Additionally, an accumulate operator += can be used to update the state of the accumulator; the function of += depends on the accumulator type.

In the example below, there are two accumulators attached to each vertex. The initial value of an accumulator of a given type is predefined, however it can be changed at declaration as in the accumulator @weight below. All vertex-attached accumulator names have a single leading at-sign "@".

Vertex-Attached Accumulators

```
SumAccum<int>   @neighbors;
MaxAccum<float> @weight = 2.8;
```

If there is a graph with 10 vertices, then there is an instance of `@neighbors` and `@weight` for each vertex (hence 10 of each, and 20 total accumulator instances). These are accessed via the dot operator on a vertex variable or a vertex alias (e.g., `v.@neighbor`). The accumulator operator += only impacts the accumulator for the specific vertex being referenced. A statement such as `v1.@neighbors += 1` will only impact **v1** 's `@neighbors` and not the `@neighbors` for other vertices.

Vertex-attached accumulators can only be accessed or updated (via = or +=) in an `ACCUM` or `POST-ACCUM` clause within a `SELECT` block. The only exception to this rule is that vertex-attached accumulators can be referenced in a PRINT statement, as the PRINT has access to all information attached to a vertex set.

⚠ Edge-attached accumulators are not supported.

Global Accumulators

A global accumulator is a single mutable accumulator that can be accessed or updated within a query. The names of global accumulators start with a double at-sign "@@".

Global Accumulators

```
SumAccum<int>   @@totalNeighbors;
MaxAccum<float> @@entropy = 1.0;
```

Global accumulators can only be assigned (using the = operator) outside a SELECT block (i.e., not within an ACCUM or POST-ACCUM clause). Global accumulators can be accessed or updated via the accumulate operator += anywhere within a query, including inside a SELECT block.

It is important to note that the accumulation operation for global accumulators in an ACCUM clause executes once for each process. That is, if the FROM clause uses an edge-induced selection (introduced in Section "SELECT Statement"), the ACCUM clause executes one process for each edge in the selected edge set. If the FROM clause uses a vertex-induced selection (introduced in Section "SELECT Statement"), the ACCUM clause executes one process for each vertex in the selected vertex set. Since global accumulators are shared in a mutually exclusive manner among processes, they behave very differently than a non-accumulator variable (see Section "Variable Types" for more details) in an ACCUM clause. Take the following code example. The global accumulator**@@globalRelationshipCount** is accumulated for every **worksFor** edge traversed since it is shared among processes. Conversely, **relationshipCount** appears to have only been incremented once. This is because a non-accumulator variable is not shared among processes. Each process has its own separate unshared copy of **relationshipCount** and increments ***the original value*** by one. (E.g., each process increments **relationshipCount** from 0 to 1.) There is no accumulation and the final value is one.

Global Variable vs Global Accumulator

```
#Count the total number of employment relationships for all companies
CREATE QUERY countEmploymentRelationships() FOR GRAPH workNet {

  INT localRelationshipCount;
  SumAccum<INT> @@globalRelationshipCount;

  start = {company.*};

  companies = SELECT s FROM start:s -(worksFor)-> :t
          ACCUM @@globalRelationshipCount += 1,
                localRelationshipCount = localRelationshipCount + 1;

  PRINT localRelationshipCount;
  PRINT @@globalRelationshipCount;
}
```

countEmploymentRelationship.json Results

```
GSQL > RUN QUERY countEmploymentRelationships()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"localRelationshipCount": 1},
    {"@@globalRelationshipCount": 17}
  ]
}
```

Static Global Accumulators

A static global accumulator retains its value after the execution of a query. To declare a static global accumulator, include the STATIC keyword at the beginning of the declaration statement. For example, if a static global accumulator is incremented by 1 each time a query is executed, then its value is equal to the number of times the query has been run, since the query was installed. Each static global accumulator belongs to the particular query in which it is declared; it cannot be shared among different queries. The value only persists in the context of running the same query multiple times. The value will reset to the default value when the GPE is restarted.

Static Global Accumulators example

```
CREATE QUERY staticAccumEx(INT x) FOR GRAPH minimalNet {
  STATIC ListAccum<INT> @@testList;
  @@testList += x;
  PRINT @@testList;
}
```

staticAccumEx.json Result

```
GSQL > RUN QUERY staticAccumEx(3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@testList": [
    3,
    -5,
    3
  ]}]
}
GSQL > RUN QUERY staticAccumEx(-5)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@testList": [
    3,
    -5,
    3,
    -5
  ]}]
}
```

There is no command to deallocate a static global accumulator. If a static global accumulator is a collection accumulator and is no longer needed, it should be cleared to minimize memory usage.

Accumulator Types

The following are the accumulator types we currently support. Each type of accumulator supports one or more data types.

EBNF for Accumulator Types

```
accumType := "SumAccum" "<" ( INT | FLOAT | DOUBLE | STRING | STRING COMP
           | "MaxAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
         | "MinAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
         | "AvgAccum"
           | "OrAccum"
           | "AndAccum"
       | "BitwiseOrAccum"
       | "BitwiseAndAccum"
           | "ListAccum" "<" type ">"
           | "SetAccum"  "<" elementType ">"
           | "BagAccum"  "<" elementType ">"
       | "MapAccum"  "<" elementType "," (baseType | accumType | tupleTyp
       | "HeapAccum" "<" tupleType ">" "(" simpleSize "," fieleName [ASC
                               ["," fieldName [ASC | DESC]]* ")"
           | "GroupByAccum" "<" elementType fieldName ["," elementType fi
                                accumType fieldName ["," accumType fieldN
       | "ArrayAccum" "<" accumName ">"

elementType := baseType | tupleType | STRING COMPRESS

gAccumAccumStmt := globaAccumName "+=" expr
```

The accumulators fall into two major groups :

- **Scalar Accumulators** store a single value:
  - SumAccum
  - MinAccum, MaxAccum
  - AvgAccum
  - AndAccum, OrAccum
  - BitwiseAndAccum, BitwiseOrAccum
- **Collection Accumulators** store a set of values:
  - ListAccum
  - SetAccum
  - BagAccum
  - MapAccum
  - ArrayAccum
  - HeapAccum
  - GroupByAccum

The details of each accumulator type are summarized in the table below. The Accumulation Operation column explains how the accumulator **accumName** is updated when the statement **accumName += newVal** is executed. Following the table are example queries for each accumulator type.

Table Ac1: Accumulator Types and Their Accumulation Behavior

Accumulator Type (Case Sensitive)	Default Initial Value	Accumulation operation (result of `accumName` `+=` `newVal`)
SumAccum<INT>	0	*accumName* plus *newVal*
SumAccum<FLOAT or DOUBLE>	0.0	*accumName* plus *newVal*
SumAccum<STRING>	empty string	String concatenation of *accumName* and *newVal*
MaxAccum<INT>	INT_MIN	The greater of *newVal* and *accumName*
MaxAccum<FLOAT or DOUBLE>	FLOAT_MIN or DOUBLE_MIN	The greater of *newVal* and *accumName*
MaxAccum<STRING>	empty string	The greater of *newVal* and *accumName* , according to UTF-8 lexicographical ordering
MaxAccum<VERTEX>	the vertex with internal id 0	The vertex with the greater internal id , either *newVal* or *accumName*
MaxAccum<tupleTyple>	default for each field of the tuple	The greater of *newVal* and *accumName.* tupleType is a user-defined sequence of baseTypes. Ordering is hierarchical, using the leftmost field of the tuple first, then the next field, and so on.
MinAccum<INT>	INT_MAX	The lesser of *newVal* and *accumName*

MinAccum<FLOAT or DOUBLE>	FLOAT_MAX or DOUBLE_MAX	The lesser of *newVal* and *accumName*
MinAccum<STRING>	empty string	The lesser of *newVal* and *accumName* , according to UTF-8 lexicographical ordering
MinAccum<VERTEX>	unknown	The vertex with the lesser internal id, either *newVal* or *accumName*
MinAccum<tupleType>	default for each field of the tuple	The lesser of *newVal* and *accumName.* tupleType is a user-defined sequence of baseTypes. Ordering is hierarchical, using the leftmost field of the tuple first, then the next field, and so on.
AvgAccum	0.0 (double precision)	Double precision average of *newVal* and all previous values accumulated to*accumName*
AndAccum	True	Boolean AND of *newVal* and *accumName*
OrAccum	False	Boolean OR of *newVal* and *accumName*
BitwiseAndAccum	-1 (INT) = 64-bit sequence of 1s	Bitwise AND of *newVal* and *accumName*
BitwiseOrAccum	0 (INT) = 64-bit sequence of 0s	Bitwise OR of *newVal* and *accumName*
ListAccum< *type* > (ordered collection of elements)	empty list	List with *newVal* appended to end of *accumName.* *newVal* can be a single value or a list. If *accumName* is [2, 4, 6], then *accumName* += 4 produces *accumName* equal to [2, 4, 6, 4]

SetAccum< t*ype* > (unordered collection of elements, duplicate items not allowed)	empty set	Set union of *newVal* and *accumName* . *newVal* can be a single value or a set/bag.If *accumName* is (2, 4, 6), then *accumName* += 4 produces *accumName* equal to (2, 4, 6)
BagAccum<t*ype* > (unordered collection of elements, duplicate items allowed)	empty bag	Bag union of *newVal* and *accumName* . *newVal* can be a single value or a set/bag.If *accumName* is (2, 4, 6), then *accumName* += 4 would result in *accumName* equal to (2, 4, 4, 6)
MapAccum< *type, type* > (unordered collection of (key,value) pairs)	empty map	Add or update a key:value pair to the *accumName* map. If *accumName* is [("red",3), ("green",4), ("blue",2)], then *accumName* += ("black"→ 5) produces *accumName* equal to [("red",3), ("green",4),("blue",2), ("black",5)]
ArrayAccum< *accumType* >	empty list	See the ArrayAccum section below for details.
HeapAccum< *tuple* >(heapSize, sortKey [, sortKey_i]*) (sorted collection of tuples)	empty heap	Insert *newVal* into the *accumName* heap, maintaining the heap in sorted order, according to the sortKey(s) and size limit declared for this HeapAccum
GroupByAccum< *type [, type]** , *accumType [, accumType]** >	empty group by map	Add or update a key:value pair in *accumName* . See Section "GroupByAccum"

SumAccum

The SumAccum type computes and stores the cumulative sum of numeric values or the cumulative concatenation of text values. The output of a SumAccum is a single numeric or string value. SumAccum variables operate on values of type INT, UINT, FLOAT, DOUBLE, or STRING only.

The **+=** operator updates the accumulator's state. For INT, FLOAT, and DOUBLE types, **+=** `arg` performs a numeric addition, while for the STRING value type **+=** `arg` concatenates **arg** to the current value of the SumAccum.

SumAccum Example

```
# SumAccum Example
CREATE QUERY sumAccumEx() FOR GRAPH minimalNet {

  SumAccum<INT>    @@intAccum;
  SumAccum<FLOAT>  @@floatAccum;
  SumAccum<DOUBLE> @@doubleAccum;
  SumAccum<STRING> @@stringAccum;

  @@intAccum  = 1;
  @@intAccum += 1;

  @@floatAccum = @@intAccum;
  @@floatAccum = @@floatAccum / 3;

  @@doubleAccum  = @@floatAccum * 8;
  @@doubleAccum += -1;

  @@stringAccum  = "Hello ";
  @@stringAccum += "World";

  PRINT @@intAccum;
  PRINT @@floatAccum;
  PRINT @@doubleAccum;
  PRINT @@stringAccum;
}
```

sumAccumEx.json Result

```
GSQL > RUN QUERY sumAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@intAccum": 2},
    {"@@floatAccum": 0.66667},
    {"@@doubleAccum": 4.33333},
    {"@@stringAccum": "Hello World"}
  ]
}
```

MinAccum / MaxAccum

The `MinAccum` and `MaxAccum` types calculate and store the cumulative minimum or the cumulative maximum of a series of values. The output of a `MinAccum` or a `MaxAccum` is a single value of the type that was passed in. `MinAccum` and `MaxAccum` variables operate on values of type `INT, UINT, FLOAT, DOUBLE, STRING, TUPLE`, and `VERTEX` (with optional specific vertex type) only.

For `MinAccum`, **+=** **`arg`** checks if the current value held is less than **arg** and stores the smaller of the two. `MaxAccum` behaves the same, with the exception that it checks for and stores the greater instead of the lesser of the two.

MinAccum and MaxAccum Example

```
# MinAccum and MaxAccum Example
CREATE QUERY minMaxAccumEx() FOR GRAPH minimalNet {

  MinAccum<INT> @@minAccum;
  MaxAccum<FLOAT> @@maxAccum;

  @@minAccum += 40;
  @@minAccum += 20;
  @@minAccum += -10;

  @@maxAccum += -1.1;
  @@maxAccum += 2.5;
  @@maxAccum += 2.8;

  PRINT @@minAccum;
  PRINT @@maxAccum;
}
```

minMaxAccumEx.json Result

```
GSQL > RUN QUERY minMaxAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@minAccum": -10},
    {"@@maxAccum": 2.8}
  ]
}
```

String minimum and maximum values are based on their UTF-8 codes, which is a multilingual superset of the ASCII codes. Within ASCII, a < z, uppercase is less than lowercase, and digits are less than alphabetic characters.

MinAccum and MaxAccum operating on VERTEX type have a special comparison. They do not compare vertex ids, but TigerGraph internal ids, which might not be in the same order as the external ids. Comparing internal ids is much faster, so MinAccum/MaxAccum<VERTEX> provides an efficient way to compare and select vertices. This is helpful for some graph algorithms that require the vertices to be numbered and sortable. For example, the following query returns one post from each person. The returned vertex is not necessarily the vertex with the alphabetically largest id.

MaxAccum<VERTEX> example

```
# Output one random post vertex from each person
CREATE QUERY minMaxAccumVertex() FOR GRAPH socialNet api("v2") {

  MaxAccum<VERTEX> @maxVertex;
  allUser = {person.*};
  allUser = SELECT src
            FROM allUser:src -(posted)-> post:tgt
            ACCUM src.@maxVertex += tgt
            ORDER BY src.id;
  PRINT allUser[allUser.@maxVertex]; // api v2
}
```

minMaxAccuxVertex.json Result

```
GSQL > RUN QUERY minMaxAccumVertex()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"allUser": [
    {
      "v_id": "person1",
      "attributes": {"allUser.@maxVertex": "0"},
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {"allUser.@maxVertex": "1"},
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {"allUser.@maxVertex": "2"},
      "v_type": "person"
    },
    {
      "v_id": "person4",
      "attributes": {"allUser.@maxVertex": "3"},
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {"allUser.@maxVertex": "11"},
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {"allUser.@maxVertex": "10"},
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {"allUser.@maxVertex": "9"},
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {"allUser.@maxVertex": "7"},
```

```
        "v_type": "person"
      }
    ]}]
  }
```

Tuple data types are treated as hierarchical structures, where the first field used for ordering is the leftmost one. When a tuple is used as an element of a `MinAccum` or `MaxAccum`, tuple fields can be directly accessed from the accumulator. For example, if we have the following tuple type and `MaxAccum` :

```
TYPEDEF TUPLE <FLOAT weight> EDGE_WEIGHT
MinAccum<EDGE_WEIGHT> @@AccTest;
```

Then the `weight` field of the tuple can be accessed directly from the `MacAccum` through the doc operator(`.`):

```
@@AccTest.weight // Will return the weight field value for the EDGE_WEIGH
                 // type tuple stored in the MaxAccum
```

AvgAccum

The AvgAccum type calculates and stores the cumulative mean of a series of numeric values. Internally, its state information includes the sum value of all inputs and a count of how many input values it has accumulated. The output is the mean value; the sum and the count values are not accessible to the user. The data type of an AvgAccum variable is not declared; all AvgAccum accumulators accept inputs of type INT, UINT, FLOAT, and DOUBLE. The output is always DOUBLE type.

The **+= arg** operation updates the AvgAccum variable's state to be the mean of all the previous arguments along with the current argument; The **`= arg`** operation clears all the previously accumulated state and sets the new state to be **arg** with a count of one.

AvgAccum Example

```
# AvgAccum Example
CREATE QUERY avgAccumEx() FOR GRAPH minimalNet {

  AvgAccum @@averageAccum;

  @@averageAccum += 10;
  @@averageAccum += 5.5; # avg = (10+5.5) / 2.0
  @@averageAccum += -1;  # avg = (10+5.5-1) / 3.0

  PRINT @@averageAccum;  # 4.8333...

  @@averageAccum = 99;   # reset
  @@averageAccum += 101; # avg = (99 + 101) / 2

  PRINT @@averageAccum;  # 100
}
```

avgAccumEx.json Result

```
GSQL > RUN QUERY avgAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@averageAccum": 4.83333},
    {"@@averageAccum": 100}
  ]
}
```

AndAccum / OrAccum

The AndAccum and OrAccum types calculate and store the cumulative result of a series of boolean operations. The output of an AndAccum or an OrAccum is a single boolean value (True or False). AndAccum and OrAccum variables operate on boolean values only. The data type does not need to be declared.

For AndAccum, **+= arg** updates the state to be the logical AND between the current boolean state and **arg**. OrAccum behaves the same, with the exception that it stores the result of a logical OR operation.

AndAccum and OrAccum Example

```
# AndAccum and OrAccum Example
CREATE QUERY andOrAccumEx() FOR GRAPH minimalNet {
  # T = True
  # F = False

  AndAccum @@andAccumVar; # (default value = T)
  OrAccum  @@orAccumVar;  # (default value = F)

  @@andAccumVar += True;  # T and T = T
  @@andAccumVar += False; # T and F = F
  @@andAccumVar += True;  # F and T = F

  PRINT @@andAccumVar;

  @@orAccumVar += False;  # F or F == F
  @@orAccumVar += True;   # F or T == T
  @@orAccumVar += False;  # T or F == T

  PRINT @@orAccumVar;
}
```

andOrAccumEx.json Result

```
GSQL > RUN QUERY andOrAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@andAccumVar": false},
    {"@@orAccumVar": true}
  ]
}
```

BitwiseAndAccum / BitwiseOrAccum

The BitwiseAndAccum and BitwiseOrAccum types calculate and store the cumulative result of a series of bitwise boolean operations and store the resulting bit sequences. BitwiseAndAccum and BitwiseOrAccum operator on INT only. The data type does not need to be declared.

Fundamental for understanding and using bitwise operations is the knowledge that integers are stored in base-2 representation as a 64-bit sequence of 1s and 0s. "Bitwise" means that each bit is treated as a separate boolean value, with 1 representing true and 0 representing false. Hence, an integer is equivalent to a sequence of boolean values. Computing the Bitwise AND of two numbers A and B means to compute the bit sequence C where the j th bit of C, denoted C j , is equal to (A j AND B j).

For BitwiseAndAccum, **+=** `arg` updates the accumulator's state to be the Bitwise AND of the current state and **arg** `.` BitwiseOrAccum behaves the same, with the exception that it computes a Bitwise OR.

> Bitwise Operations and Negative Integers
>
> Most computer systems represent negative integers using "2's complement" format, where the uppermost bit has special significance. Operations that affect the uppermost bit are crossing the boundary between positive and negative numbers, and vice versa.

BitwiseAndAccum and BitwiseOrAccum Example

```
# BitwiseAndAccum and BitwiseOrAccum Example
CREATE QUERY bitwiseAccumEx() FOR GRAPH minimalNet {

  BitwiseAndAccum @@bwAndAccumVar; # default value = 64-bits of 1 = -1 (I
  BitwiseOrAccum  @@bwOrAccumVar;  # default value = 64-bits of 0 = 0 (IN

  # 11110000 = 240
  # 00001111 =  15
  # 10101010 = 170
  # 01010101 =  85

  # BitwiseAndAccum
  @@bwAndAccumVar += 170; # 11111111 & 10101010 -> 10101010
  @@bwAndAccumVar +=  85; # 10101010 & 01010101 -> 00000000
  PRINT @@bwAndAccumVar;  # 0

  @@bwAndAccumVar = 15;   # reset to 00001111
  @@bwAndAccumVar += 85;  # 00001111 & 01010101 -> 00000101
  PRINT @@bwAndAccumVar;  # 5

  # BitwiseOrAccum
  @@bwOrAccumVar += 170; # 00000000 | 10101010 -> 10101010
  @@bwOrAccumVar +=  85; # 10101010 | 01010101 -> 11111111 = 255
  PRINT @@bwOrAccumVar;  # 255

  @@bwOrAccumVar = 15;   # reset to 00001111
  @@bwOrAccumVar += 85;  # 00001111 | 01010101 -> 01011111 = 95
  PRINT @@bwOrAccumVar;  # 95
}
```

bitwiseAccumEx.json Result

```
GSQL > RUN QUERY bitwiseAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@bwAndAccumVar": 0},
    {"@@bwAndAccumVar": 5},
    {"@@bwOrAccumVar": 255},
    {"@@bwOrAccumVar": 95}
  ]
}
```

ListAccum

The ListAccum type maintains a sequential collection of elements. The output of a ListAccum is a list of values in the order the elements were added. The element type can be any base type, tuple, or STRING COMPRESS. Additionally, a ListAccum can contain a nested collection of type ListAccum. Nesting of ListAccums is limited to a depth of three.

The **+= arg** operation appends `arg` to the end of the list. In this case, `arg` may be either a single element or another ListAccum.

ListAccum supports two additional operations:

- **@list1 + @list2** creates a new ListAccum, which contains the elements of @list1 followed by the elements of @list2. The two ListAccums must have identical data types.

> Change in "+" definition
>
> The pre-v2.0 definition of the ListAccum "+" operator (**@list + arg**: Add arg to each member of @list) is no longer supported.

- **@list1 * @list2** (STRING data only) generates a new list of strings consisting of all permutations of an element of the first list followed by an element of the second list.

ListAccum also supports the following class functions.

> Functions that modify the ListAccum (mutator functions) can be used only under the following conditions:
>
> - Mutator functions of global accumulators may only be used at the query-body level.
> - Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function (T is the element type)	Return type	Accessor / Mutator	Description
.size()	INT	Accessor	Returns the number of elements in the list.
.contains(T *val* **)**	BOOL	Accessor	Returns true/false if the list does/doesn't contain the *value* .
.get(INT *idx* **)**	T	Accessor	Returns the value at the given *index* position in the list. The index begins at 0. If the index is out of bound (including any negative value), the default value of the element type is returned.
.clear()	VOID	Mutator	Clears the list so it becomes empty with size 0.
.update (INT *index,* T *value*)	VOID	Mutator	Assigns *value* to the list element at

position *index*

ListAccum Example

```
# ListAccum Example
CREATE QUERY listAccumEx() FOR GRAPH minimalNet {

  ListAccum<INT> @@intListAccum;
  ListAccum<STRING> @@stringListAccum;
  ListAccum<STRING> @@stringMultiplyListAccum;
  ListAccum<STRING> @@stringAdditionAccum;
  ListAccum<STRING> @@letterListAccum;
  ListAccum<ListAccum<STRING>> @@nestedListAccum;

  @@intListAccum = [1,3,5];
  @@intListAccum += [7,9];
  @@intListAccum += 11;
  @@intListAccum += 13;
  @@intListAccum += 15;

  PRINT @@intListAccum;
  PRINT @@intListAccum.get(0), @@intListAccum.get(1);
  PRINT @@intListAccum.get(8); # Out of bound: default value of int: 0

  #Other built-in functions
  PRINT @@intListAccum.size();
  PRINT @@intListAccum.contains(2);
  PRINT @@intListAccum.contains(3);

  @@stringListAccum += "Hello";
  @@stringListAccum += "World";

  PRINT @@stringListAccum; // ["Hello","World"]

  @@letterListAccum += "a";
  @@letterListAccum += "b";

  # ListA + ListB produces a new list equivalent to ListB appended to Lis
  # Ex: [a,b,c] + [d,e,f] => [a,b,c,d,e,f]
  @@stringAdditionAccum = @@stringListAccum + @@letterListAccum;

  PRINT @@stringAdditionAccum;

  #Multiplication produces a list of all list-to-list element combination
  # Ex: [a,b] * [c,d] = [ac, ad, bc, bd]
  @@stringMultiplyListAccum = @@stringListAccum * @@letterListAccum;

  PRINT @@stringMultiplyListAccum;

  #Two dimensional list (3 dimensions is possible as well)
  @@nestedListAccum += [["foo", "bar"], ["Big", "Bang", "Theory"], ["Stri
```

```
  PRINT @@nestedListAccum;
  PRINT @@nestedListAccum.get(0);
  PRINT @@nestedListAccum.get(0).get(1);
}
```

listAccumEx.json Result

```
GSQL > RUN QUERY listAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [ {"@@intListAccum": [ 1, 3, 5, 7, 9, 11, 13, 15 ]},
    {
      "@@intListAccum.get(0)": 1,
      "@@intListAccum.get(1)": 3
    },
    {"@@intListAccum.get(8)": 0},
    {"@@intListAccum.size()": 8},
    {"@@intListAccum.contains(2)": false},
    {"@@intListAccum.contains(3)": true},
    {"@@stringListAccum": [ "Hello", "World" ]},
    {"@@stringAdditionAccum": [ "Hello", "World", "a", "b"]},
    {"@@stringMultiplyListAccum": [ "Helloa", "Worlda", "Hellob", "Worldb
    {"@@nestedListAccum": [
      [ "foo", "bar" ],
      [ "Big", "Bang", "Theory" ],
      [ "String", "Theory" ]
    ]},
    {"@@nestedListAccum.get(0)": [ "foo", "bar" ]},
    {"@@nestedListAccum.get(0).get(1)": "bar"}
  ]
}
```

Example for update function on a global ListAccum

```
CREATE QUERY listAccumUpdateEx() FOR GRAPH workNet {

  # Global ListAccum
  ListAccum<INT> @@intListAccum;
  ListAccum<STRING> @@stringListAccum;
  ListAccum<BOOL> @@passFail;

  @@intListAccum += [0,2,4,6,8];
  @@stringListAccum += ["apple","banana","carrot","daikon"];

  # Global update at Query-Body Level
  @@passFail += @@intListAccum.update(1,-99);
  @@passFail += @@intListAccum.update(@@intListAccum.size()-1,40);  // la
  @@passFail += @@stringListAccum.update(0,"zero"); // first element
  @@passFail += @@stringListAccum.update(4,"four"); // FAIL: out-of-range

  PRINT @@intListAccum, @@stringListAccum, @@passFail;
}
```

Results in listAcccumUpdateEx.json

```
GSQL > RUN QUERY listAccumUpdateEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "@@passFail": [ true, true, true, false ],
    "@@intListAccum": [ 0, -99, 4, 6, 40 ],
    "@@stringListAccum": [ "zero", "banana", "carrot", "daikon" ]
  }]
}
```

Example for update function on a vertex-attached ListAccum

```
CREATE QUERY listAccumUpdateEx2(SET<VERTEX<person>> seed) FOR GRAPH workN

  # Each person has an LIST<INT> of skills and a LIST<STRING COMPRESS> of
  # This function copies their lists into ListAccums, and then udpates th
  # int with -99 and updates the last string with "fizz".
  ListAccum<INT> @intList;
  ListAccum<STRING COMPRESS> @stringList;
  ListAccum<STRING> @@intFails, @@strFails;

  S0 (person) = seed;
  S1 = SELECT s
    FROM S0:s
    ACCUM
      s.@intList = s.skillList,
      s.@stringList = s.interestList
    POST-ACCUM
      INT len = s.@intList.size(),
      IF NOT s.@intList.update(len-1,-99) THEN
        @@intFails += s.id END,
      INT len2 = s.@stringList.size(),
      IF NOT s.@stringList.update(len2-1,"fizz") THEN
        @@strFails += s.id END
  ;
  PRINT S1[S1.skillList, S1.interestList, S1.@intList, S1.@stringList]; /
  PRINT @@intFails, @@strFails;
}
```

Results for listAccumUpdateEx2

```
GSQL > RUN QUERY listAccumUpdateEx2(["person1","person5"])
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"S1": [
      {
        "v_id": "person1",
        "attributes": {
          "S1.@stringList": [ "management","fizz" ],
          "S1.interestList": [ "management", "financial"],
          "S1.skillList": [  1, 2, 3 ],
          "S1.@intList": [ 1, 2, -99 ]
        },
        "v_type": "person"
      },
      {
        "v_id": "person5",
        "attributes": {
          "S1.@stringList": [ "sport", "financial", "fizz" ],
          "S1.interestList": [ "sport", "financial", "engineering" ],
          "S1.skillList": [ 8, 2, 5 ],
          "S1.@intList": [ 8, 2, -99 ]
        },
        "v_type": "person"
      }
    ]},
    {
      "@@strFails": [],
      "@@intFails": []
    }
  ]
}
```

SetAccum

The SetAccum type maintains a collection of unique elements. The output of a SetAccum is a list of elements in arbitrary order. A SetAccum instance can contain values of one type. The element type can be any base type, tuple, or STRING COMPRESS.

For SetAccum, the **+= arg** operation adds a non-duplicate element or set of elements to the set. If an element is already represented in the set, then the SetAccum state does not change.

SetAccum also can be used with the three canonical set operators: UNION, INTERSECT, and MINUS (see Section "Set/Bag Expression and Operators" for more details).

SetAccum also supports the following class functions.

Functions that modify the SetAccum (mutator functions) can be used only under the following conditions:

- Mutator functions of global accumulators may only be used at the query-body level.
- Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function (T is the element type)	Return type	Accessor / Mutator	Description
size()	INT	Accessor	Returns the number of elements in the set.
contains(T *value* **)**	BOOL	Accessor	Returns true/false if the set does/doesn't contain the *value*.
remove(T *value* **)**	VOID	Mutator	Removes *value* from the set.
clear()	VOID	Mutator	Clears the set so it becomes empty with size 0.

SetAccum Example

```
# SetAccum Example
CREATE QUERY setAccumEx() FOR GRAPH minimalNet {

  SetAccum<INT> @@intSetAccum;
  SetAccum<STRING> @@stringSetAccum;

  @@intSetAccum += 5;
  @@intSetAccum.clear();

  @@intSetAccum += 4;
  @@intSetAccum += 11;
  @@intSetAccum += 1;
  @@intSetAccum += 11; # Sets do not store duplicates

  @@intSetAccum += (1,2,3,4); # Can create simple sets this way
  PRINT @@intSetAccum;
  @@intSetAccum.remove(2);
  PRINT @@intSetAccum AS RemovedVal2; # Demostrate remove.

  PRINT @@intSetAccum.contains(3);

  @@stringSetAccum += "Hello";
  @@stringSetAccum += "Hello";
  @@stringSetAccum += "There";
  @@stringSetAccum += "World";
  PRINT @@stringSetAccum;

  PRINT @@stringSetAccum.contains("Hello");
  PRINT @@stringSetAccum.size();
}
```

setAccumEx.json Result

```
GSQL > RUN QUERY setAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [ {"@@intSetAccum": [ 3, 2, 1, 11, 4 ]},
    {"@@intSetAccum.contains(3)": true},
    {"@@stringSetAccum": [ "World", "There", "Hello" ]},
    {"@@stringSetAccum.contains(Hello)": true},
    {"@@stringSetAccum.size()": 3}
  ]
}
```

BagAccum

The BagAccum type maintains a collection of elements with duplicated elements allowed. The output of a BagAccum is a list of elements in arbitrary order. A BagAccum instance can contain values of one type. The element type can be any base type, tuple, or STRING COMPRESS.

For BagAccum, the **+= arg** operation adds an element or bag of elements to the bag.

BagAccum also supports the **+** operator:

- **@bag1 + @bag2** creates a new BagAccum, which contains the elements of @bag1 and the elements of @bag2. The two BagAccums must have identical data types.

BagAccum also supports the following class functions.

⚠ Functions which modify the BagAccum (mutator functions) can be used only under the following conditions:

- Mutator functions of global accumulators may only be used at the query-body level.

- Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function (T is the element type)	Return type	Accessor / Mutator	Description
size()	INT	Accessor	Returns the number of elements in the bag.
contains(T *value* **)**	BOOL	Accessor	Returns true/false if the bag does/doesn't contain the *value* .
clear()	VOID	Mutator	Clears the bag so it becomes empty with size 0.
remove(T *value* ***)***	VOID	Mutator	Removes one instance of *value* from the bag.
removeAll(T *value* **)**	VOID	Mutator	Removes all instances of the given value from the bag.

BagAccum Example

```
# BagAccum Example
CREATE QUERY bagAccumEx() FOR GRAPH minimalNet {

  #Unordered collection
  BagAccum<INT>    @@intBagAccum;
  BagAccum<STRING> @@stringBagAccum;

  @@intBagAccum += 5;
  @@intBagAccum.clear();

  @@intBagAccum += 4;
  @@intBagAccum += 11;
  @@intBagAccum += 1;
  @@intBagAccum += 11;        #Bag accums can store duplicates
  @@intBagAccum += (1,2,3,4);
  PRINT @@intBagAccum;

  PRINT @@intBagAccum.size();
  PRINT @@intBagAccum.contains(4);

  @@stringBagAccum += "Hello";
  @@stringBagAccum += "Hello";
  @@stringBagAccum += "There";
  @@stringBagAccum += "World";
  PRINT @@stringBagAccum.contains("Hello");
  @@stringBagAccum.remove("Hello");    #Remove one matching element
  @@stringBagAccum.removeAll("There"); #Remove all matching elements
  PRINT @@stringBagAccum;
}
```

bagAccumEx.json Result

```
GSQL > RUN QUERY bagAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [ {"@@intBagAccum": [ 2, 3, 1, 1, 11, 11, 4, 4 ]},
    {"@@intBagAccum.size()": 8},
    {"@@intBagAccum.contains(4)": true},
    {"@@stringBagAccum.contains(Hello)": true},
    {"@@stringBagAccum": [ "World", "Hello" ]}
  ]
}
```

MapAccum

The MapAccum type maintains a collection of (key → value) pairs. The output of a MapAccum is a set of key and value pairs in which the keys are unique.

The key type of a MapAccum can be all base types or tuples. If the key type is VERTEX, then only the vertex's id is stored and displayed.

The value type of a MapAccum can be all base types, tuples, or any type of accumulator, except for HeapAccum.

For MapAccum, the **+=** `(key->val)` operation adds a key-value element to the collection if **key** is not yet used in the MapAccum. If the MapAccum already contains **key** , then **val** is *accumulated* to the current value, where the accumulation operation depends on the data type of **val** . (Strings would get concatenated, lists would be appended, numerical values would be added, etc.)

MapAccum also supports the **+** operator:

- **@map1 + @map2** creates a new MapAccum, which contains the (key,value) pairs of @map2 added to the (key,value) pairs of @map1. The two MapAccums must have identical data types.

MapAccum also supports the following class functions.

Functions that modify the MapAccum (mutator functions) can be used only under the following conditions:

- Mutator functions of global accumulators may only be used at the query-body level.
- Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function (KEY is the key type)	Return type	Accessor / Mutator	Description
size()	INT	Accessor	Returns the number of elements in the map.
containsKey(KEY *key* **)**	BOOL	Accessor	Returns true/false if the map does/doesn't contain *key* .
get(KEY *key* **)**	*value* type	Accessor	Returns the value which the map associates with *key* . If the map doesn't contain*key* , then the return value is undefined.
clear()	VOID	Mutator	Clears the map so it becomes empty with size 0.

MapAccum Example

```
#MapAccum Example
CREATE QUERY mapAccumEx() FOR GRAPH minimalNet {

  #Map(Key, Value)
  # Keys can be INT or STRING only
  MapAccum<STRING, INT> @@intMapAccum;
  MapAccum<INT, STRING> @@stringMapAccum;
  MapAccum<INT, MapAccum<STRING, STRING>> @@nestedMapAccum;

  @@intMapAccum += ("foo" -> 1);
  @@intMapAccum.clear();

  @@intMapAccum += ("foo" -> 3);
  @@intMapAccum += ("bar" -> 2);
  @@intMapAccum += ("baz" -> 2);
  @@intMapAccum += ("baz" -> 1); #add 1 to existing value

  PRINT @@intMapAccum.containsKey("baz");
  PRINT @@intMapAccum.get("bar");
  PRINT @@intMapAccum.get("root");

  @@stringMapAccum += (1 -> "apple");
  @@stringMapAccum += (2 -> "pear");
  @@stringMapAccum += (3 -> "banana");
  @@stringMapAccum += (4 -> "a");
  @@stringMapAccum += (4 -> "b"); #append "b" to existing value
  @@stringMapAccum += (4 -> "c"); #append "c" to existing value

  PRINT @@intMapAccum;
  PRINT @@stringMapAccum;

  #Checking and getting keys
  if @@stringMapAccum.containsKey(1) THEN
    PRINT @@stringMapAccum.get(1);
  END;

  #Map nesting
  @@nestedMapAccum += ( 1 -> ("foo"  -> "bar") );
  @@nestedMapAccum += ( 1 -> ("flip" -> "top") );
  @@nestedMapAccum += ( 2 -> ("fizz" -> "pop") );
  @@nestedMapAccum += ( 1 -> ("foo"  -> "s") );

  PRINT @@nestedMapAccum;

  if @@nestedMapAccum.containsKey(1) THEN
    if @@nestedMapAccum.get(1).containsKey("foo") THEN
       PRINT @@nestedMapAccum.get(1).get("foo");
    END;
```

```
  END;
}
```

mapAccumEx.json Result

```
GSQL > RUN QUERY mapAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@intMapAccum.containsKey(baz)": true},
    {"@@intMapAccum.get(bar)": 2},
    {"@@intMapAccum.get(root)": 0},
    {"@@intMapAccum": {
      "bar": 2,
      "foo": 3,
      "baz": 3
    }},
    {"@@stringMapAccum": {
      "1": "apple",
      "2": "pear",
      "3": "banana",
      "4": "abc"
    }},
    {"@@stringMapAccum.get(1)": "apple"},
    {"@@nestedMapAccum": {
      "1": {
        "foo": "bars",
        "flip": "top"
      },
      "2": {"fizz": "pop"}
    }},
    {"@@nestedMapAccum.get(1).get(foo)": "bars"}
  ]
}
```

ArrayAccum

The ArrayAccum type maintains an array of accumulators. An array is a fixed-length sequence of elements, with direct access to elements by position. The ArrayAccum has these particular characteristics:

- The elements are accumulators, not primitive or base data types. All accumulators, except HeapAccum, MapAccum, and GroupByAccum, can be used.
- An ArrayAccum instance can be multidimensional. There is no limit to the number of dimensions.
- The size can be set at run-time (dynamically).
- There are operators which update the entire array efficiently.

When an ArrayAccum is declared, the instance name should be followed by a pair of brackets for each dimension. The brackets may either contain an integer constant to set the size of the array, or they may be empty. In that case, the size must be set with the reallocate function before the ArrayAccum can be used.

ArrayAccum declaration example

```
ArrayAccum<SetAccum<STRING>> @@names[10];
ArrayAccum<SetAccum<INT>> @@ids[][];  // 2-dimensional, size to be determ
```

Because each element of an ArrayAccum itself is an accumulator, the operators =, +=, and + can be used in two contexts: accumulator-level and element-level.

Element-level operations

If @A is an ArrayAccum of length 6, then @A[0] and @A[5] refer to its first and last elements, respectively. Referring to an ArrayAccum element is like referring to an accumulator of that type. For example, given the following definitions:

```
ArrayAccum<SumAccum<INT>> @@Sums[3];
ArrayAccum<ListAccum<STRING>> @@Lists[2];
```

then @@Sums[0], @@Sums[1], and @@Sums[2] each refer to an individual SumAccum<INT>, and @@Lists[0] and @@Lists[1] each refer to a

ListAccum<STRING>, supporting all the operations for those accumulator and data types.

```
@@Sums[1] = 1;
@@Sums[1] += 2;  // value is now 3
@@Lists[0] = "cat";
@@Lists[0] += "egory";  // value is now "category"
```

Accumulator-level operations

The operators =, +=, and + have special meanings when applied to an ArrayAccum as a whole. There operations efficiently update an entire ArrayAccum. All of the ArrayAccums must have the same element type.

Operator	Description	Example
=	Sets the ArrayAccum on the left equal to the ArrayAccum on the right. The two ArrayAccums must have the same element type, but the left-side ArrayAccum will change its size and dimensions to match the one on the right side.	@A = @B;
+	Performs element-by-element addition of two ArrayAccums of the same type and size. The result is a new ArrayAccum of the same size.	@C = @A + @B; // @A and @B must be the same size
+=	Performs element-by-element accumulation (+=) from the right-side ArrayAccum to the left-side ArrayAccum. They must be the same type and size.	@A += @B; // @A and @B must be the same size

ArrayAccum also supports the following class functions.

⚠ Functions that modify the ArrayAccum (mutator functions) can be used only under the following conditions:

- Mutator functions of global accumulators may only be used at the query-body level.
- Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function	Return type	Accessor / Mutator	Description
size()	INT	Accessor	Returns the total number of elements in the (multi-dimensional) array. For example, the size of an ArrayAccum declared as @A[3][4] is 12.
reallocate(INT, ... **)**	VOID	Mutator	Discards the previous ArrayAccum instance and creates a new ArrayAccum, with the size(s) given. An N-dimensional ArrayAccum requires N integer parameters. The reallocate function cannot be used to change the number of dimensions.

Example of ArrayAccum Element-level Operations

```
CREATE QUERY ArrayAccumElem() FOR GRAPH minimalNet {

    ArrayAccum<SumAccum<DOUBLE>> @@aaSumD[2][2];  # 2D Sum Double
    ArrayAccum<SumAccum<STRING>> @@aaSumS[2][2];  # 2D Sum String
    ArrayAccum<MaxAccum<INT>> @@aaMax[2];
    ArrayAccum<MinAccum<UINT>> @@aaMin[2];
    ArrayAccum<AvgAccum> @@aaAvg[2];
    ArrayAccum<AndAccum<BOOL>> @@aaAnd[2];
    ArrayAccum<OrAccum<BOOL>> @@aaOr[2];
    ArrayAccum<BitwiseAndAccum> @@aaBitAnd[2];
    ArrayAccum<BitwiseOrAccum> @@aaBitOr[2];
    ArrayAccum<ListAccum<INT>> @@aaList[2][2];    # 2D List
    ArrayAccum<SetAccum<FLOAT>> @@aaSetF[2];
    ArrayAccum<BagAccum<DATETIME>> @@aaBagT[2];

    ## for test data
    ListAccum<STRING> @@words;
    BOOL toggle = false;
    @@words += "1st"; @@words += "2nd"; @@words += "3rd"; @@words += "4th

    # Int:  a[0] += 1, 2;   a[1] += 3, 4
    # Bool: alternate true/false
    # Float: a[0] += 1.111, 2.222;  a[1] += 3.333, 4.444
    # 2D Doub: a[0][0] += 1.111, 2.222;   a[0][1] += 5.555, 6.666;
    #          a[1][0] += 3.333, 4.444;   a[0][1] += 7.777, 8.888;

    FOREACH i IN RANGE [0,1] DO
        FOREACH n IN RANGE [1, 2] DO
            toggle = NOT toggle;
            @@aaMax[i] += i*2 + n;
            @@aaMin[i] += i*2 + n;
            @@aaAvg[i] += i*2 + n;
            @@aaAnd[i] += toggle;
            @@aaOr[i] += toggle;
            @@aaBitAnd[i] += i*2 + n;
            @@aaBitOr[i] += i*2 + n;
            @@aaSetF[i] += (i*2 + n)/0.9;
            @@aaBagT[i] += epoch_to_datetime(i*2 + n);

            FOREACH j IN RANGE [0,1] DO
                @@aaSumD[i][j] += (j*4 + i*2 + n)/0.9;
                @@aaSumS[i][j] += @@words.get((j*2 + i + n)%4);
                @@aaList[i][j] += j*4 +i*2 + n ;
            END;
        END;
    END;

    PRINT @@aaSumD;     PRINT @@aaSumS;
```

```
    PRINT @@aaMax;      PRINT @@aaMin;      PRINT @@aaAvg;
    PRINT @@aaAnd;      PRINT @@aaOr;
    PRINT @@aaBitAnd;   PRINT @@aaBitOr;
    PRINT @@aaList;     PRINT @@aaSetF;     PRINT @@aaBagT;
}
```

ArrayAccumElem.json Results

```
GSQL > RUN QUERY ArrayAccumElem()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@aaSumD": [
      [ 3.33333, 12.22222 ],
      [ 7.77778, 16.66667 ]
    ]},
    {"@@aaSumS": [
      [ "2nd3rd", "4th1st" ],
      [ "3rd4th", "1st2nd" ]
    ]},
    {"@@aaMax": [ 2, 4 ]},
    {"@@aaMin": [ 1, 3 ]},
    {"@@aaAvg": [ 1.5, 3.5 ]},
    {"@@aaAnd": [ false, false ]},
    {"@@aaOr": [ true, true ]},
    {"@@aaBitAnd": [ 0, 0 ]},
    {"@@aaBitOr": [ 3, 7]},
    {"@@aaList": [
      [
        [ 1, 2 ],
        [ 5, 6]
      ],
      [
        [ 3, 4 ],
        [ 7, 8 ]
      ]
    ]},
    {"@@aaSetF": [
      [ 2.22222, 1.11111],
      [ 4.44444, 3.33333 ]
    ]},
    {"@@aaBagT": [
      [ 2, 1 ],
      [ 4, 3 ]
    ]}
  ]
}
```

Example of Operations between Whole ArrayAccums

```
CREATE QUERY ArrayAccumOp3(INT lenA) FOR GRAPH minimalNet {

    ArrayAccum<SumAccum<INT>> @@arrayA[5]; // Original size
    ArrayAccum<SumAccum<INT>> @@arrayB[2];
    ArrayAccum<SumAccum<INT>> @@arrayC[][]; // No size
    STRING msg;
    @@arrayA.reallocate(lenA);  # Set/Change size dynamically
    @@arrayB.reallocate(lenA+1);
    @@arrayC.reallocate(lenA, lenA+1);

    // Initialize arrays
    FOREACH i IN RANGE[0,lenA-1] DO
        @@arrayA[i] += i*i;
        FOREACH j IN RANGE[0,lenA] DO
            @@arrayC[i][j] += j*10 + i;
        END;
    END;
    FOREACH i IN RANGE[0,lenA] DO
        @@arrayB[i] += 100-i;
    END;
    msg = "Initial Values";
    PRINT msg, @@arrayA, @@arrayB, @@arrayC;

    msg = "Test 1: A = C, C = B";   // = operator
    @@arrayA = @@arrayC;        // change dimensions: 1D <- 2D
    @@arrayC = @@arrayB;        // change dimensions: 2D <- 1D
    PRINT msg, @@arrayA, @@arrayC;

    msg = "Test 2: B += C";         // += operator
    @@arrayB += @@arrayC;       // B and C must have same size & dim
    PRINT msg, @@arrayB, @@arrayC;

    msg = "Test 3: A = B + C";      // + operator
    @@arrayA = @@arrayB + @@arrayC; // B & C must have same size & dim
    PRINT msg, @@arrayA;            // A changes size & dim
}
```

ArrayAccumOp3.json Results

```
GSQL > RUN QUERY ArrayAccumOp3(3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "msg": "Initial Values",
      "@@arrayC": [
        [ 0, 10, 20, 30 ],
        [ 1, 11, 21, 31 ],
        [ 2, 12, 22, 32 ]
      ],
      "@@arrayB": [ 100, 99, 98, 97 ],
      "@@arrayA": [ 0, 1, 4 ]
    },
    {
      "msg": "Test 1: A = C, C = B",
      "@@arrayC": [ 100, 99, 98, 97 ],
      "@@arrayA": [
        [ 0, 10, 20, 30 ],
        [ 1, 11, 21, 31 ],
        [ 2, 12, 22, 32 ]
      ]
    },
    {
      "msg": "Test 2: B += C",
      "@@arrayC": [ 100, 99, 98, 97 ],
      "@@arrayB": [ 200, 198,196, 194 ]
    },
    {
      "msg": "Test 3: A = B + C",
      "@@arrayA": [ 300, 297, 294, 291 ]
    }
  ]
}
```

Example for Vertex-Attached ArrayAccum

```
CREATE QUERY arrayAccumLocal() FOR GRAPH socialNet api("v2") {
    # Count each person's edges by type
    # friend/liked/posted edges are type 0/1/2, respectively
    ArrayAccum<SumAccum<INT>> @edgesByType[3];
    Persons = {person.*};

    Persons = SELECT s
        FROM Persons:s -(:e)-> :t
        ACCUM CASE e.type
            WHEN "friend" THEN s.@edgesByType[0] += 1
            WHEN "liked"  THEN s.@edgesByType[1] += 1
            WHEN "posted" THEN s.@edgesByType[2] += 1
            END
        ORDER BY s.id;

    #PRINT Persons.@edgesByType; // api v1
    PRINT Persons[Persons.@edgesByType]; // api v2
}
```

Results for Query ArrayAccumLocal

```
GSQL > RUN QUERY arrayAccumLocal()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Persons": [
    {
      "v_id": "person1",
      "attributes": {"Persons.@edgesByType": [ 2, 1, 1 ]},
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {"Persons.@edgesByType": [ 2, 2, 1 ]},
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {"Persons.@edgesByType": [ 2, 1, 1 ]},
      "v_type": "person"
    },
    {
      "v_id": "person4",
      "attributes": {"Persons.@edgesByType": [ 3, 1, 1 ]},
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {"Persons.@edgesByType": [ 2, 1, 2 ]},
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {"Persons.@edgesByType": [ 2, 1, 2 ]},
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {"Persons.@edgesByType": [ 2, 1, 2 ]},
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {"Persons.@edgesByType": [ 3, 1, 2 ]},
```

```
      "v_type": "person"
    }
  ]}]
}
```

HeapAccum

The HeapAccum type maintains a sorted collection of tuples and enforces a maximum number of tuples in the collection. The output of a HeapAccum is a sorted collection of tuple elements. The **+= arg** operation adds a tuple to the collection in sorted order. If the HeapAccum is already at maximum capacity when the **+=** operator is applied, then the tuple which is last in the sorted order is dropped from the HeapAccum. Sorting of tuples is performed on one or more defined tuple fields ordered either ascending or descending. Sorting precedence is performed based on defined tuple fields from left to right.

The declaration of a HeapAccum is more complex than for most other accumulators, because the user must define a custom tuple type, set the maximum capacity of the HeapAccum, and specify how the HeapAccum should be sorted. The declaration syntax is outlined in the figure below:

HeapAccum declaration syntax

```
TYPEDEF TUPLE<type field_1,.., type field_n> tupleType;
...
HeapAccum<tupleType>(capacity, field_a [ASC|DESC],... , field_z [ASC|DESC
```

First, the HeapAccum declaration must be preceded by a TYPEDEF statement which defines the tuple type. At least one of the fields (field_1, ..., field_n) must be of a data type that can be sorted.

In the declaration of the HeapAccum itself, the keyword "HeapAccum" is followed by the tuple type in angle brackets < >. This is followed by a parenthesized list of two or more parameters. The first parameter is the maximum number of tuples that the HeapAccum may store. This parameter must be a positive integer. The subsequent parameters are a subset of the tuple's field, which are used as sort keys. The sort key hierarchy is from left to right, with the leftmost key being the primary sort key. The keywords ASC and DESC indicate Ascending (lowest value first) or Descending (highest value first) sort order. Ascending order is the default.

HeapAccum also supports the following class functions.

Functions that modify the HeapAccum (mutator functions) can be used only under the following conditions:

- Mutator functions of global accumulators may only be used at the query-body level.
- Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function	Return type	Accessor / Mutator	Description
size()	INT	Accessor	Returns the number of elements in the heap.
top()	tupleType	Accessor	Returns the top tuple. If this heap is empty, returns a tuple with each element equal to the default value.
pop()	tupleType	Mutator	Returns the top tuple and removes it from the heap. If this heap is empty, returns a tuple with each element equal to the default value.
resize(INT **)**	VOID	Mutator	Changes the maximum capacity of the heap.
clear()	VOID	Mutator	Clears the heap so it becomes empty with size 0.

HeapAccum Example

```
#HeapAccum Example
CREATE QUERY heapAccumEx() FOR GRAPH minimalNet {
  TYPEDEF tuple<STRING firstName, STRING lastName, INT score> testResults

  #Heap with max size of 4 sorted decending by score then ascending last
  HeapAccum<testResults>(4, score DESC, lastName ASC) @@topTestResults;

  PRINT @@topTestResults.top();

  @@topTestResults += testResults("Bruce", "Wayne", 80);
  @@topTestResults += testResults("Peter", "Parker", 80);
  @@topTestResults += testResults("Tony", "Stark", 100);
  @@topTestResults += testResults("Bruce", "Banner", 95);
  @@topTestResults += testResults("Jean", "Summers", 95);
  @@topTestResults += testResults("Clark", "Kent", 80);

  #Show element with the highest sorted position
  PRINT @@topTestResults.top();
  PRINT @@topTestResults.top().firstName, @@topTestResults.top().lastName

  PRINT @@topTestResults;

  #Increase the size of the heap to add more elements
  @@topTestResults.resize(5);

  #Find the size of the current heap
  PRINT @@topTestResults.size();

  @@topTestResults += testResults("Bruce", "Wayne", 80);
  @@topTestResults += testResults("Peter", "Parker", 80);

  PRINT @@topTestResults;

  #Resizing smaller WILL REMOVE excess elements from the HeapAccum
  @@topTestResults.resize(3);
  PRINT @@topTestResults;

  #Increasing capacity will not restore dropped elements
  @@topTestResults.resize(5);
  PRINT @@topTestResults;

  #Removes all elements from the HeapAccum
  @@topTestResults.clear();
  PRINT @@topTestResults.size();
}
```

heapAccumEx.json Results

```
GSQL > RUN QUERY heapAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@topTestResults.top()": {
      "firstName": "",
      "lastName": "",
      "score": 0
    }},
    {"@@topTestResults.top()": {
      "firstName": "Tony",
      "lastName": "Stark",
      "score": 100
    }},
    {
      "@@topTestResults.top().firstName": "Tony",
      "@@topTestResults.top().lastName": "Stark",
      "@@topTestResults.top().score": 100
    },
    {"@@topTestResults": [
      {
        "firstName": "Tony",
        "lastName": "Stark",
        "score": 100
      },
      {
        "firstName": "Bruce",
        "lastName": "Banner",
        "score": 95
      },
      {
        "firstName": "Jean",
        "lastName": "Summers",
        "score": 95
      },
      {
        "firstName": "Clark",
        "lastName": "Kent",
        "score": 80
      }
    ]},
    {"@@topTestResults.size()": 4},
```

```
{"@@topTestResults": [
  {
    "firstName": "Tony",
    "lastName": "Stark",
    "score": 100
  },
  {
    "firstName": "Bruce",
    "lastName": "Banner",
    "score": 95
  },
  {
    "firstName": "Jean",
    "lastName": "Summers",
    "score": 95
  },
  {
    "firstName": "Clark",
    "lastName": "Kent",
    "score": 80
  },
  {
    "firstName": "Peter",
    "lastName": "Parker",
    "score": 80
```

GroupByAccum

GroupByAccum syntax

```
GroupByAccum<type [, type]* , accumType [, accumType]* >
```

```
    "firstName": "Tony",
    "lastName": "Stark",
    "score": 100
  },
  {
    "firstName": "Bruce",
    "lastName": "Banner",
    "score": 95
  },
  {
    "firstName": "Jean",
    "lastName": "Summers",
    "score": 95
  }
]},
{"@@topTestResults": [
  {
    "firstName": "Tony",
    "lastName": "Stark",
```

```
Typedef tuple <id int, name string, age int> myTuple;
Typedef HeapAccum <myTuple> (2, name desc, age desc, id asc) myHeap;
GroupByAccum<INT a, STRING b,
             MaxAccum<INT> maxa,
             ListAccum<ListAccum<INT>> lists,
             myHeap h> @@group;
```

```
        "firstName": "Jean",
        "lastName": "Summers",
        "score": 95
      }
    ]},
    {"@@topTestResults.size()": 0}
  ]
}
```

Functions that modify the GroupByAccum (mutator functions) can be used only under the following conditions:

- Mutator functions of global accumulators may only be used at the query-body level.
- Mutator functions of vertex-attached accumulators may only be used in a POST-ACCUM clause.

Function (KEY1..KEYn are the key types)	Return type	Accessor / Mutator	Description
size()	INT	Accessor	Returns the number of elements in the heap.
get(KEY1 *key_value1* , KEY2 *key_value2* ...)	element type(s) of the accumulator(s)	Accessor	Returns the values from each accumulator in the group associating with the given key(s). If the key(s) doesn't exist, return the default value(s) of the accumulator type(s).
containsKey(KEY1 *key_value1* , KEY2	BOOL	Accessor	Returns true/false if the accumulator

key_value2... **)**			contains the key(s)
clear()	VOID	Mutator	Clears the heap so it becomes empty with size 0.
remove (KEY1 *key_value1* , KEY2 *key_value2* ... **)**	VOID	Mutator	Removes the group associating with the key(s)

GroupByAccum Example

```
#GroupByAccum Example
CREATE QUERY groupByAccumEx () FOR GRAPH socialNet {
  ##declare HeapAccum type and tuple used in the HeapAccum
  Typedef tuple <id int, name string, age int> myTuple;
  Typedef HeapAccum <myTuple> (2, name desc, age desc, id asc) myHeap;
  ## declaration, first two primitive type are group by keys; the rest ac
  GroupByAccum<INT a, STRING b, MaxAccum<INT> maxa, ListAccum<ListAccum<I
  GroupByAccum<STRING gender, MapAccum<VERTEX<person>, DATETIME> m> @@gro
  GroupByAccum<INT age, myHeap h> @@group4;
  # nested GroupByAccum
  GroupByAccum<INT a, MaxAccum<INT> maxa, GroupByAccum<INT a, MaxAccum<IN
  Start = { person.* };

  ## usage of global GroupByAccum
  @@group += (1, "a" -> 1, [1]);
  @@group += (1, "a" -> 2, [2]);
  @@group += (2, "b" -> 1, [4]);

  @@group3 += (2 -> 1, (2 -> 0) );
  @@group3 += (2 -> 1, (2 -> 5) );
  @@group3 += (2 -> 5, (3 -> 3) );
  PRINT @@group, @@group.get(1, "a"), @@group.get(1, "a").lists,  @@group

  ## HeapAccum inside GroupByAccum
  @@group4 += (29->myTuple(1,"aaa", 18));
  @@group4 += (29->myTuple(2,"bbb", 19));
  @@group4 += (29->myTuple(3,"ccc", 20));
  PRINT @@group4;

  ## two kinds of foreach
  FOREACH g IN @@group DO
    PRINT g.a, g.b, g.maxa, g.lists;
  END;
  FOREACH (g1,g2,g3,g4) IN @@group DO
    PRINT g1,g2,g3,g4;
  END;

  S = SELECT v
      FROM Start:v - (liked:e) - post:t
      ACCUM @@group2 += (v.gender -> (v -> e.actionTime));

  PRINT @@group2, @@group2.get("Male").m, @@group2.get("Female").m;
}
```

Result for Query groupByAccum

```
GSQL > RUN QUERY groupByAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@group.get(1,a).lists": [
        [1],
        [2]
      ],
      "@@group3": [{
        "a": 2,
        "heap": [
          {
            "a": 3,
            "maxa": 3
          },
          {
            "a": 2,
            "maxa": 5
          }
        ],
        "maxa": 5
      }],
      "@@group.containsKey(1,c)": false,
      "@@group.get(1,a)": {
        "lists": [
          [1],
          [2]
        ],
        "maxa": 2
      },
      "@@group": [
        {
          "a": 2,
          "b": "b",
          "lists": [[4]],
          "maxa": 1
        },
        {
          "a": 1,
          "b": "a",
          "lists": [
```

```
          [1],
          [2]
        ],
        "maxa": 2
      }
    ]
  },
  {
    "g.b": "b",
    "g.maxa": 1,
    "g.lists": [[4]],
    "g.a": 2
  },
  {
    "g.b": "a",
    "g.maxa": 2,
    "g.lists": [
      [1],
      [2]
    ],
    "g.a": 1
  },
  {
    "g1": 2,
    "g2": "b",
    "g3": 1,
    "g4": [[4]]
  },
  {
    "g1": 1,
    "g2": "a",
    "g3": 2,
    "g4": [
      [1],
      [2]
```


Nested Accumulators

```
ListAccum<ListAccum<INT>> @@matrix; # a 2-dimensional jagged array of int
```


```
  {
    "@@group2.get(Male).m": {
      "person3": 1263618953,
      "person1": 1263209520,
      "person8": 1263180365,
      "person7": 1263295325,
      "person6": 1263468185
    },
    "@@group2": [
      {
        "gender": "Male",
```

```
ListAccum<ListAccum<INT>>
ListAccum<ListAccum<ListAccum<INT>>>
ListAccum<SetAccum<INT>> # illegal
```

```
            "person7": 1263295325,
            "person6": 1263468185
          }
        },
        {
```

```
MapAccum<STRING, ListAccum<INT>>
MapAccum<INT, MapAccum<INT, STRING>>
MapAccum<VERTEX, SumAccum<INT>>
MapAccum<STRING, SetAccum<VERTEX>>
MapAccum<STRING, GroupByAccum<VERTEX a, MaxAccum<INT> maxs>>
MapAccum<SetAccum<INT>, INT> # illegal
```

```
      ],
      "@@group2.get(Female).m": {
        "person4": 1263352565,
        "person2": 2526519281,
        "person5": 1263330725
```

```
GroupByAccum<INT a, STRING b, MaxAccum<INT> maxs, ListAccum<ListAccum<INT
```

```
  ]
}
```

nesting is mandatory for ArrayAccum. See the ArrayAccum section above.

It is legal to define nested ListAccums to form a multi-dimensional array. Note the declaration statements and the nested [bracket] notation in the example below:

```
CREATE QUERY nestedAccumEx() FOR GRAPH minimalNet {
  ListAccum<ListAccum<INT>> @@_2d_list;
  ListAccum<ListAccum<ListAccum<INT>>> @@_3d_list;
  ListAccum<INT> @@_1d_list;
  SumAccum <INT> @@sum = 4;

  @@_1d_list += 1;
  @@_1d_list += 2;
  // add 1D-list to 2D-list as element
  @@_2d_list += @@_1d_list;

  // add 1D-enum-list to 2D-list as element
  @@_2d_list += [@@sum, 5, 6];
  // combine 2D-enum-list and 2d-list
  @@_2d_list += [[7, 8, 9], [10, 11], [12]];

  // add an empty 1D-list
  @@_1d_list.clear();
  @@_2d_list += @@_1d_list;

  // combine two 2D-list
  @@_2d_list += @@_2d_list;

  PRINT @@_2d_list;

  // test 3D-list
  @@_3d_list += @@_2d_list;
  @@_3d_list += [[7, 8, 9], [10, 11], [12]];
  PRINT @@_3d_list;
}
```

nestedAccumEx.json Results

```
GSQL > RUN QUERY nestedAccumEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@_2d_list": [
      [1,2],
      [4,5,6],
      [7,8,9],
      [10,11],
      [12],
      [],
      [1,2],
      [4,5,6],
      [7,8,9],
      [10,11],
      [12],
      []
    ]},
    {"@@_3d_list": [
      [
        [1,2],
        [4,5,6],
        [7,8,9],
        [10,11],
        [12],
        [],
        [1,2],
        [4,5,6],
        [7,8,9],
        [10,11],
        [12],
        []
      ],
      [
        [7,8,9],
        [10,11],
        [12]
      ]
    ]}
  ]
}
```

Operators and Expressions

An *expression* is a combination of fixed values, variables, operators, function calls, and groupings that specify a computation, resulting in a data value. This section of the specification describes the literals (fixed values), operators, and functions available in the GSQL query language. It covers the subset of the EBNF definitions shown below. However, more so than in other sections of the specification, syntax alone is not an adequate description. The semantics (functionality) of the particular operators and functions are an essential complement to the syntax.

EBNF for Operations, Functions, and Expressions

```
constant := numeric | stringLiteral | TRUE | FALSE | GSQL_UINT_MAX
          | GSQL_INT_MAX | GSQL_INT_MIN | TO_DATETIME "(" stringLiteral "

mathOperator := "*" | "/" | "%" | "+" | "-" | "<<" | ">>" | "&" | "|"

condition := expr
           | expr comparisonOperator expr
           | expr [ NOT ] IN setBagExpr
           | expr IS [ NOT ] NULL
           | expr BETWEEN expr AND expr
           | "(" condition ")"
           | NOT condition
           | condition (AND | OR) condition
           | (TRUE | FALSE)
           | expr [NOT] LIKE expr [ESCAPE escape_char]

comparisonOperator := "<" | "<=" | ">" | ">=" | "==" | "!="

aggregator := COUNT | MAX | MIN | AVG | SUM

expr := name
    | globalAccumName
        | name ".type"
        | name "." name
        | name "." localAccumName ["\'"]
        | name "." name "." name "(" [argList] ")"
        | name "." name "(" [argList] ")" [ "." FILTER "(" condition ")"
        | name ["<" type ["," type]* ">"] "(" argList] ")"
        | name "." localAccumName ("." name "(" [argList] ")")+ ["." name
        | globalAccumName ("."name "(" [argList] ")")+ ["."name]
        | COALESCE "(" [argList] ")"
        | aggregator "(" [DISTINCT] setBagExpr ")"
        | ISEMPTY "(" setBagExpr ")"
        | expr mathOperator expr
        | "-" expr
        | "(" expr ")"
        | "(" argList "->" argList ")"  // key value pair for MapAccum
        | "[" argList "]"               // a list
        | constant
        | setBagExpr
        | name "(" argList ")"          // function call or a tuple objec

setBagExpr := name
        | globalAccumName
          | name "." name
            | name "." localAccumName
            | name "." localAccumName ("." name "(" [argList] ")")+
            | name "." name "(" [argList] ")" [ "." FILTER "(" condition
```

```
              | globalAccumName ("." name "(" [argList] ")")+
              | setBagExpr (UNION | INTERSECT | MINUS) setBagExpr
              | "(" argList ")"
              | "(" setBagExpr ")"


argList := expr ["," expr]*
```

Constants

```
constant := numeric | stringLiteral | TRUE | FALSE | GSQL_UINT_MAX
          | GSQL_INT_MAX | GSQL_INT_MIN | TO_DATETIME "(" stringLiteral "
```

Each primitive data type supports constant values:

Data Type	Constant	Examples
Numeric types (`INT`, `UINT`, `FLOAT`, `DOUBLE`)	`numeric`	123 -5 45.67 2.0e-0.5
`UINT`	`GSQL_UINT_MAX`	
`INT`	`GSQL_INT_MAX` `GSQL_INT_MIN`	
`boolean`	`TRUE` `FALSE`	
`string`	`stringLiteral`	`"atoz@com"` `"0.25"`

`GSL_UINT_MAX` = 2 ^ 64 - 1 = 18446744073709551615

`GSQL_INT_MAX` = 2 ^ 63 - 1 = 9223372036854775807

`GSQL_INT_MIN` = -2 ^ 63   = -9223372036854775808

Operators

An operator is a keyword token that performs a specific computational function to return a resulting value, using the adjacent expressions (its operands) as input values. An operator is similar to a function in that both compute a result from inputs, but syntactically they are different. The most familiar operators are the mathematical operators for addition `+` and subtraction `-`.

Tip: The operators listed in this section are designed to behave like the operators in MySQL.

Mathematical Operators and Expressions

We support the following standard mathematical operators and meanings. The latter four ("<<" | ">>" | "&" | "|") are for bitwise operations. See the section below: "Bit Operators".

```
mathOperator := "*" | "/" | "%" | "+" | "-" | "<<" | ">>" | "&" | "|"
```

Operator precedences are shown in the following list, from the highest precedence to the lowest. Operators that are shown together on a line have the same precedence:

Operator Precedence, highest to lowest

```
*, /, %
-, +
<<, >>
&
|
==, >=, >, <=, <, !=
```

Example 1. Math Operators + - * /

```
CREATE QUERY mathOperators() FOR GRAPH minimalNet api("v2")
{
    int x,y;
    int z1,z2,z3,z4,z5;
    float f1,f2,f3,f4;

    x = 7;
    y = 3;

    z1 = x * y;    # z = 21
    z2 = x - y;    # z = 4
    z3 = x + y;    # z = 10
    z4 = x / y;    # z = 2
    z5 = x / 4.0;  # z = 1
    f1 = x / y;    # v = 2
    f2 = x / 4.0;  # v = 1.75
    f3 = x % 3;    # v = 1
    f4 = x % y;    # z = 1

    PRINT x,y;
    PRINT z1 AS xTIMESy, z2 AS xMINUSy, z3 AS xPLUSy, z4 AS xDIVy, z5 AS
    PRINT f1 AS xDIVy,   f2 AS xDIV4f,  f3 AS xMOD3,  f4 AS xMODy;
}
```

mathOperators.json Results

```
GSQL > RUN QUERY mathOperators()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "x": 7,
      "y": 3
    },
    {
      "xTIMESy": 21,
      "xPLUSy": 10,
      "xMINUSy": 4,
      "xDIVy": 2,
      "xDIV4f": 1
    },
    {
      "xMODy": 1,
      "xMOD3": 1,
      "xDIVy": 2,
      "xDIV4f": 1.75
    }
  ]
}
```

Boolean Operators

We support the standard Boolean operators and standard order of precedence: AND, OR, NOT

Bit Operators

Bit operators (<<, >>, &, and |) operate on integers and return an integer.

Bit Operators

```
CREATE QUERY bitOperationTest() FOR GRAPH minimalNet{
  PRINT 80 >> 2;      # 20
  PRINT 80 << 2;      # 320
  PRINT 2 + 80 >> 4; # 5
  PRINT 2 | 3 ;       # 3
  PRINT 2 & 3 ;       # 2
  PRINT 2 | 3 + 2;    # 7
  PRINT 2 & 3 - 2;    # 0
}
```

String Operators

Operator + can be used for concatenating strings.

Tuple Fields

The fields of the tuple can be accessed using the dot operator.

Comparison Operators and Conditions

A condition is an expression that evaluates to a boolean value of either true or false. One type of condition uses the familiar comparison operators. A comparison operator compares two numeric or string values.

```
comparisonOperator := "<" | "<=" | ">" | ">=" | "==" | "!="

condition := expr
           | expr comparisonOperator expr
           | expr [ NOT ] IN setBagExpr
           | expr IS [ NOT ] NULL
           | expr BETWEEN expr AND expr
           | "(" condition ")"
           | NOT condition
           | condition (AND | OR) condition
           | (TRUE | FALSE)
           | expr [NOT] LIKE expr [ESCAPE escape_char]
```

Strings are compared based on standard lexicographical ordering:
(space) < (digit) < (uppercase_letter) < (lowercase_letter).

> The comparison operators treat the STRING COMPRESS type as though it is STRING type.

BETWEEN expr AND expr

The expression expr1 BETWEEN expr2 AND expr3 is true if the value expr1 is in the range from expr2 to expr3, including the endpoint values. Each expression must be numeric.

" expr1 BETWEEN expr2 AND expr3 " is equivalent to " expr1 <= expr3 AND expr1 >= expr2".

BETWEEN AND example

```
CREATE QUERY mathOperatorBetween() FOR GRAPH minimalNet
{
    int x;
    bool b;
    x = 1;
    b = (x BETWEEN 0 AND 100); PRINT b;  # True
    b = (x BETWEEN 1 AND 2); PRINT b;    # True
    b = (x BETWEEN 0 AND 1); PRINT b;    # True
}
```

IS NULL, IS NOT NULL

IS NULL and IS NOT NULL can be used for checking whether an optional parameter is given any value.

IS NULL example

```
CREATE QUERY parameterIsNULL (INT p) FOR GRAPH minimalNet {
  IF p IS NULL THEN
    PRINT "p is null";
  ELSE
    PRINT "p is not null";
  END;
}
```

parameterIsNULL.json Results

```
GSQL > RUN QUERY parameterIsNULL(_)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"p is null": "p is null"}]
}
GSQL > RUN QUERY parameterIsNULL(3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"p is not null": "p is not null"}]
}
```

Every attribute value stored in GSQL is a valid value, so IS NULL and IS NOT NULL is only effective for query parameters.

LIKE

```
expr [NOT] LIKE expr [ESCAPE escape_char]
```

The `LIKE` operator is used for string pattern matching and can only be used in `WHERE` clauses. The expression `string1 LIKE string_pattern` evaluates to boolean true if `string1` matches the pattern in `string_pattern`; otherwise, it is false.

Both operands must be strings. Additionally, while `string1` can be a function call (e.g. `lower(string_variable)`, `string_pattern` must be a string literal or a parameter. A `string_pattern` can contain characters as well as the following wildcard and other special symbols, in order to express a pattern (`<char_list>` indicates a placeholder):

Character or syntax	Description	Example
`%`	Matches zero or more characters.	`%abc%` matches any string which contains the sequence `"abc"`.
`_` (underscore)	Matches any single character.	`_abc_e` matches any 6-character string where the 2nd to 4th characters are `"abc"` and the last character is `"e"`.
`[<char_list>]`	Matches any character in a char list. A char list is a concatenated character set, with no separators.	`[Tiger]` matches either `T`, `i`, `g`, `e`, or `r`.
`[^<char_list>]`	Matches any character NOT in a char list.	`[^qxz]` matches any character other than `q`, `x`, or `z`.
`[!<char_list>]`	Matches any character NOT in a char list.	
`α-β`	(Special syntax within a char list) matches a character in the range from α to β. A char list can have multiple ranges.	`[a-mA-M0-3]` matches a letter from a to m, upper or lower case, or a digit from 0 to 3.
`\\`	(Special syntax within a char list) matches the character `\`	

`\\]`	(Special syntax within a char list) matches the character `]` No special treatment is needed for [inside a char list.	`%[\\]!]` matches any string which ends with either `]` or `!`

`ESCAPE escape_char`

The optional `ESCAPE escape_char` clause is used to define an escape character. When `escape_char` occurs in `string_pattern`, then the next character is interpreted literally, instead of as a pattern matching operator. For example, if we want to specify the pattern "any string ending with the `'%'` character", we could use
`"%\%" ESCAPE "\"`

The first `"%"` has its usual pattern-matching meaning "zero or more characters". `"\%"` means a literal percentage character, because it starts with the escape character `"\"`.

Example

Example query using LIKE operator

```
CREATE QUERY printAPosts(/* Parameters here */) FOR GRAPH socialNet {
  /* Write query logic here */
  posts = {post.*};
    aPosts = SELECT v FROM posts:v
                WHERE v.subject LIKE "%a%";  // Return all posts that has
                                             // character "a" in its subj
    PRINT aPosts;
}
```

```
Results
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"aPosts": [
    {
      "v_id": "2",
      "attributes": {
        "postTime": "2011-02-03 01:02:42",
        "subject": "query languages"
      },
      "v_type": "post"
    },
    {
      "v_id": "8",
      "attributes": {
        "postTime": "2011-02-03 17:05:52",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "0",
      "attributes": {
        "postTime": "2010-01-12 11:22:05",
        "subject": "Graphs"
      },
      "v_type": "post"
    },
    {
      "v_id": "1",
      "attributes": {
        "postTime": "2011-03-03 23:02:00",
        "subject": "tigergraph"
      },
      "v_type": "post"
    }
  ]}]
}
```

Vertex, Edge, and Accumulator Attributes

Accessing attributes

Attributes on vertices or edges are defined in the graph schema. Additionally, each vertex and edge has a built-in STRING attribute called **type** which represents the user-defined type of that edge or vertex. These attributes, including **type,** can be accessed for a particular edge or vertex with the dot operator:

Accessing attributes with a known name.

```
name ".type"   // read only. Returns vertexType or edgeType of name
name "." attrName // read/write. Accesses attribute called attrName
```

DYNAMIC Query Support

The name of the attribute can be parameterized using the **getAttr** and **setAttr** vertex functions, described later in this section. This allows you to write dynamic query procedures where the attribute names are specified when you run the query.

For example, the following code snippet shows two different SELECT statements which produce equivalent results. The first uses the dot operator on the vertex variable **v** to access the "subject" attribute, which is defined in the graph schema. The FROM clause in the first SELECT statement necessitates that any target vertices will be of type "post" (also defined in the graph schema). The second SELECT schema checks that the vertex variable v's type is a "post" vertex by using the dot operator to access the built-in **type** attribute.

Accessing vertex variable attributes

```
CREATE QUERY coffeeRelatedPosts() FOR GRAPH socialNet
{
    allVertices = {ANY};
    results = SELECT v FROM allVertices:s -(:e)-> post:v WHERE v.subject
    PRINT results;
    results = SELECT v FROM allVertices:s -(:e)-> :v WHERE v.type == "pos
    PRINT results;
}
```

Results for Query coffeeRelatedPosts

```
GSQL > RUN QUERY coffeeRelatedPosts()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"results": [{
      "v_id": "4",
      "attributes": {
        "postTime": "2011-02-07 05:02:51",
        "subject": "coffee"
      },
      "v_type": "post"
    }]},
    {"results": [{
      "v_id": "4",
      "attributes": {
        "postTime": "2011-02-07 05:02:51",
        "subject": "coffee"
      },
      "v_type": "post"
    }]}
  ]
}
```

Accumulator Functions

This section describes functions that apply to all or most accumulators. Other accumulator functions for each accumulator type are illustrated in the "Accumulator Type" section.

Previous value of accumulator

The tick operator (') can be used to read the value of an accumulator as it was at the start an ACCUM clause, before any changes that took place within the ACCUM clause. It can only be used in the POST-ACCUM clause. A typical use is to compare the value of the accumulator before and after the ACCUM clause. The PageRank algorithm provides a good example:

```
v = SELECT s
    FROM start:s - (e_type:e) -> :t
    ACCUM t.@received_score += s.@score/(s.outdegree(e_type))
    POST-ACCUM s.@score = (1.0 - damping) + damping * s.@received_score,
               s.@received_score = 0,
               @@max_diff += abs(s.@score - s.@score');
```

In the last line, we compute `@@max_diff` as the absolute value of the difference between the post-ACCUM score (`s.@score`) and the pre-ACCUM score (`s.@score'`).

Set/Bag Expression and Operators

SELECT blocks take an input vertex set and perform various selection and filtering operations to produce an output set. Therefore, *set/bag expressions* and their operators are a useful and powerful part of the GSQL query language. A set/bag expression can use either SetAccum or BagAccum.

EBNF

```
setBagExpr := name
        | globalAccumName
          | name "." name
            | name "." localAccumName
            | name "." localAccumName ("." name "(" [argList] ")")+
            | name "." name "(" [argList] ")" [ "." FILTER "(" condition
            | globalAccumName ("." name "(" [argList] ")")+
            | setBagExpr (UNION | INTERSECT | MINUS) setBagExpr
            | "(" argList ")"
            | "(" setBagExpr ")"
```

Set/Bag Expression Operators - UNION, INTERSECT, MINUS

The operators are straightforward, when two operands are both sets, the result expression is a set. When at least one operand is a bag, the result expression is a bag. If one operand is a bag and the other is a set, the operator treats the set operant as a bag containing one of each value.

Set/Bag Operator Examples

```
# Demonstrate Set & Bag operators
CREATE QUERY setOperatorsEx() FOR GRAPH minimalNet   {
  SetAccum<INT> @@setA, @@setB, @@AunionB, @@AintsctB, @@AminusB;
  BagAccum<INT> @@bagD, @@bagE, @@DunionE, @@DintsctE, @@DminusE;
  BagAccum<INT> @@DminusA, @@DunionA, @@AunionBbag;

  BOOL x;

  @@setA = (1,2,3,4);      PRINT @@setA;
  @@setB = (2,4,6,8);      PRINT @@setB;

  @@AunionB = @@setA UNION @@setB ;      PRINT @@AunionB;   // (1, 2, 3,
  @@AintsctB = @@setA INTERSECT @@setB;  PRINT @@AintsctB;   // (2, 4)
  @@AminusB = @@setA MINUS @@setB ;      PRINT @@AminusB;   // C = (1, 3)

  @@bagD = (1,2,2,3);      PRINT @@bagD;
  @@bagE = (2,3,5,7);      PRINT @@bagE;

  @@DunionE = @@bagD UNION @@bagE;     PRINT @@DunionE;   // (1, 2, 2, 2,
  @@DintsctE = @@bagD INTERSECT @@bagE; PRINT @@DintsctE; // (2, 3)
  @@DminusE = @@bagD MINUS @@bagE;     PRINT @@DminusE;   // (1, 2)
  @@DminusA = @@bagD MINUS @@setA;     PRINT @@DminusA;   // (2)
  @@DunionA = @@bagD UNION @@setA;     PRINT @@DunionA;   // (1, 1, 2, 2,
                                                          // because bag
  @@AunionBbag = @@setA UNION @@setB;  PRINT @@AunionBbag;  // (1, 2, 3,
                                                          // because set
}
```

setOperatorsEx Query Results

```
GSQL > RUN QUERY setOperatorsEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [ {"@@setA": [ 4, 3, 2, 1 ]},
    {"@@setB": [ 8, 6, 4, 2 ]},
    {"@@AunionB": [ 4, 3, 2, 1, 8, 6 ]},
    {"@@AintsctB": [ 4, 2 ]},
    {"@@AminusB": [ 3, 1 ]},
    {"@@bagD": [ 1, 2, 2, 3 ]},
    {"@@bagE": [ 2, 7, 3, 5 ]},
    {"@@DunionE": [ 1, 2, 2, 2, 3, 3, 7, 5 ]},
    {"@@DintsctE": [ 2, 3 ]},
    {"@@DminusE": [ 1, 2 ]},
    {"@@DminusA": [2]},
    {"@@DunionA": [ 1, 1, 2, 2, 2, 3, 3, 4 ]},
    {"@@AunionBbag": [ 6, 8, 1, 2, 3, 4 ]}
  ]
}
```

The result of these operators is another set or bag, so these operations can be nested and chained to form more complex expressions, such as

```
(setBagExpr_A INTERSECT (setBagExpr_B UNION setBagExpr_C) ) MINUS setBagE
```

Set/Bag Expression Membership Operators

For example , suppose setBagExpr_A is ("a", "b", "c")

```
"a" IN setBagExpr_A            => true
"d" IN setBagExpr_A            => false
"a" NOT IN setBagExpr_A        => false
"d" NOT IN setBagExpr_A        => true
```

The IN and NOT IN operators support all base types on the left-hand side, and any set/bag expression on the right-hand side. The base type must be the same as the accumulator's element type. IN and NOT IN return a BOOL value.

The following example uses NOT IN to exclude neighbors that are on a blocked list.

Set Membership example

```
CREATE QUERY friendsNotInblockedlist (VERTEX<person> seed, SET<VERTEX<per
  Start = {seed};
  Result = SELECT v
      FROM Start:s-(friend:e)-person:v
      WHERE v NOT IN blockedList;
  PRINT Result;
}
```

Results for Query friendsNotInblockedlist

```
GSQL > RUN QUERY friendsNotInblockedlist("person1", ["person2"])
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "person8",
    "attributes": {
      "gender": "Male",
      "id": "person8"
    },
    "v_type": "person"
  }]}]
}
```

Subqueries

A query defined with a `RETURNS` header following its `CREATE` statement is called a subquery. A subquery acts as a callable function in GSQL. They take parameters, perform a set of actions, and return a value at the end. A subquery must end with a return statement to pass its output value to a query. Exactly one type is allowed in the `RETURNS` header, and thus the `RETURN` statement can only return one expression.

A subquery must be created before the query that calls the subquery. A subquery must be installed either before or in the same `INSTALL QUERY` command with the query that calls the subquery.

Main Components of a Subquery

```
CREATE QUERY <query_name>() FOR GRAPH <graph_name> // Parameters are opti
RETURNS (INT) /* A subquery has a RETURNS header specifying
                              its return type */
{
    // ...
    // Query body goes here
    // ...
    RETURN <return_value> /* The return statement of a subquery. Return
                            value must be the same type as specified in
                            the RETURNS header */
}
```

Parameter types

A subquery parameter can only be one of the following types:

- Primitives: `INT`, `UINT`, `FLOAT`, `DOUBLE`, `STRING`, `BOOL`
- `VERTEX`
- A set or bag of primitive or `VERTEX` elements

Return types

A subquery's return value can be any base type or accumulator type with the following exceptions.

- If the return type is a `BagAccum`. `SetAccum`, or `ListAccum` with a tuple as its element, the tuple does not need to be defined at the catalog level and can be anonymous.

Recursive subqueries

Recursion is supported for subqueries and a subquery can call itself. Here is an example of a recursive subquery: The following subquery takes a set of persons as starting points, and returns all the friends within a given distance.

While recursive subqueries may look simpler in writing, they are usually not as efficient as iterative subqueries in GSQL.

Recursive Query Example

```
CREATE QUERY subFindFriendsInDistance(SET<VERTEX> seeds, INT distance
FOR GRAPH friendNet RETURNS (SET<VERTEX>)
{
    IF distance <= 0 THEN   // Base case
    // When distance is 0, return the seed vertices themselves
      RETURN seeds;
    ELSE
    seed_vs = seeds; // Initialize starting vertices
    // Select 1-hop neighbors from the starting points
      next_vs = SELECT v FROM seed_vs -(friendship:e)- :v;
        // Find the (distance-1)-hop neighbors of the 1-hop neighbors
        // and return the union of the starting vertices and neighbor
      RETURN seeds UNION subFindFriendsInDistance(next_vs, distance -
  END;
}

CREATE QUERY findFriendsInDistance(Vertex<person> p, INT distance) FO

    seed = {p};
  //PRINT All Persons;
    PRINT subFindFriendsInDistance(seed, distance) AS friends;
}
```

Test cases: Starting from `person1`, search to a distance of 1 and a distance of 2.

```
GSQL> RUN QUERY findFriendsInDistance("person1", 1)
[
  {
    "friends": [
      "person4",
      "person3",
      "person2",
      "person1"
    ]
  }
]
GSQL> RUN QUERY findFriendsInDistance("person1", 2)
[
  {
    "friends": [
      "person4",
      "person9",
      "person3",
      "person2",
      "person6",
      "person8",
      "person1"
    ]
  }
]
```

Examples of Expressions

Below is a list of examples of expressions. Note that (argList) is a set/bag expression, while [argList] is a list expression.

Expression Examples

```
#Show various types of expressions
CREATE QUERY expressionEx() FOR GRAPH workNet {
  TYPEDEF tuple<STRING countryName, STRING companyName> companyInfo;

  ListAccum<STRING> @companyNames;
  SumAccum<INT> @companyCount;
  SumAccum<INT> @numberOfRelationships;
  ListAccum<companyInfo> @info;
  MapAccum< STRING,ListAccum<STRING> > @@companyEmployeeRelationships;
  SumAccum<INT> @@totalRelationshipCount;

  ListAccum<INT> @@valueList;
  SetAccum<INT> @@valueSet;

  SumAccum<INT> @@a;
  SumAccum<INT> @@b;

  #expr := constant
  @@a = 10;

  #expr := ["@@"] name
  @@b = @@a;

  #expr := expr mathOperator expr
  @@b = @@a + 5;

  #expr := "(" expr ")"
  @@b = (@@a + 5);

  #expr := "-" expr
  @@b = -(@@a + 5);

  PRINT @@a, @@b;

  #expr := "[" argList "]"   // a list
  @@valueList = [1,2,3,4,5];
  @@valueList += [24,80];

  #expr := "(" argList ")"  // setBagExpr
  @@valueSet += (1,2,3,4,5);

  #expr := ( COUNT | ISEMPTY | MAX | MIN | AVG | SUM ) "(" setBagExpr ")"
  PRINT MAX(@@valueList);
  PRINT AVG(@@valueList);

  seed = {ANY};

  company1 = SELECT t FROM seed:s -(worksFor)-> :t WHERE (s.id == "compan
```

```
  company2 = SELECT t FROM seed:s -(worksFor)-> :t WHERE (s.id == "compan

  #expr := setBagExpr
  worksForBoth = company1 INTERSECT company2;
  PRINT worksForBoth;

  #expr := name "." "type"
  employees = SELECT s FROM seed:s WHERE (s.type == "person");

  employees = SELECT s FROM employees:s -(worksFor)-> :t

    ACCUM
      #expr := name "." ["@"] name
      s.@companyNames += t.id,

      #expr := name "."name "(" [argList] ")" [ "."FILTER "(" condition "
      s.@numberOfRelationships += s.outdegree(),

      #expr := name ["<" type ["," type"]* ">"] "(" [argList] ")"
      s.@info += companyInfo(t.country, t.id)

   POST-ACCUM
     #expr := name "."localAccumName ("."name "(" [argList] ")")+ ["."nam
     s.@companyCount += s.@companyNames.size(),

    #expr := name "."localAccumName ["\'"]
    @@totalRelationshipCount += s.@companyCount,

  FOREACH comp IN s.@companyNames DO
     #expr := "(" argList "->" argList ")"
     @@companyEmployeeRelationships += (s.id -> comp)
  END;

  PRINT employees;
  PRINT @@totalRelationshipCount;
  PRINT @@companyEmployeeRelationships;

  #expr := globalAccumName ("."name "(" [argList] ")")+ ["."name]
  PRINT @@companyEmployeeRelationships.size();
}
```

expressionEx.json Results

```
GSQL > RUN QUERY expressionEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@a": 10,
      "@@b": -15
    },
    {"max(@@valueList)": 80},
    {"avg(@@valueList)": 17},
    {"worksForBoth": [
      {
        "v_id": "person2",
        "attributes": {
          "interestList": ["engineering"],
          "@companyCount": 0,
          "@numberOfRelationships": 0,
          "skillSet": [ 6, 5, 3, 2 ],
          "skillList": [ 2, 3, 5, 6 ],
          "locationId": "chn",
          "interestSet": ["engineering"],
          "@info": [],
          "id": "person2",
          "@companyNames": []
        },
        "v_type": "person"
      },
      {
        "v_id": "person1",
        "attributes": {
          "interestList": [ "management", "financial" ],
          "@companyCount": 0,
          "@numberOfRelationships": 0,
          "skillSet": [ 3, 2, 1 ],
          "skillList": [ 1, 2, 3 ],
          "locationId": "us",
          "interestSet": [ "financial", "management" ],
          "@info": [],
          "id": "person1",
          "@companyNames": []
        },
        "v_type": "person"
```

```
        "v_type": "person"
      }
    ]},
    {"employees": [
      {
        "v_id": "person4",
        "attributes": {
          "interestList": ["football"],
          "@companyCount": 1,
          "@numberOfRelationships": 1,
          "skillSet": [ 10, 1, 4 ],
          "skillList": [ 4, 1, 10 ],
          "locationId": "us",
          "interestSet": ["football"],
          "@info": [{ "companyName": "company2", "countryName": "chn" }],
          "id": "person4",
          "@companyNames": ["company2"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person12",
        "attributes": {
          "interestList": [
            "music",
            "engineering",
            "teaching",
            "teaching",
            "teaching"
          ],
          "@companyCount": 1,
          "@numberOfRelationships": 1,
          "skillSet": [ 2, 5, 1 ],
          "skillList": [ 1, 5, 2, 2, 2 ],
          "locationId": "jp",
          "interestSet": [ "teaching", "engineering", "music" ],
          "@info": [{ "companyName": "company4", "countryName": "us" }],
          "id": "person12",
          "@companyNames": ["company4"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person3",
        "attributes": {
          "interestList": ["teaching"],
          "@companyCount": 1,
          "@numberOfRelationships": 1,
          "skillSet": [ 6, 1, 4 ],
```

```
        "skillList": [ 4, 1, 6 ],
        "locationId": "jp",
        "interestSet": ["teaching"],
        "@info": [{ "companyName": "company1", "countryName": "us" }],
        "id": "person3",
        "@companyNames": ["company1"]
      },
      "v_type": "person"
    },
    {
      "v_id": "person9",
      "attributes": {
        "interestList": [ "financial", "teaching" ],
        "@companyCount": 2,
        "@numberOfRelationships": 4,
        "skillSet": [ 2, 7, 4 ],
        "skillList": [ 4, 7, 2 ],
        "locationId": "us",
        "interestSet": [ "teaching", "financial" ],
        "@info": [
          {
            "companyName": "company3",
            "countryName": "jp"
          },
          {
            "companyName": "company2",
            "countryName": "chn"
          }
        ],
        "id": "person9",
        "@companyNames": [ "company3", "company2" ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person11",
      "attributes": {
        "interestList": [ "sport", "football" ],
        "@companyCount": 1,
        "@numberOfRelationships": 1,
        "skillSet": [10],
        "skillList": [10],
        "locationId": "can",
        "interestSet": [ "football", "sport" ],
        "@info": [{ "companyName": "company5", "countryName": "can" }],
        "id": "person11",
        "@companyNames": ["company5"]
      },
```

```
      "v_type": "person"
    },
    {
      "v_id": "person10",
      "attributes": {
        "interestList": [ "football", "sport" ],
        "@companyCount": 2,
        "@numberOfRelationships": 4,
        "skillSet": [3],
        "skillList": [3],
        "locationId": "us",
        "interestSet": [ "sport", "football" ],
        "@info": [
          {
            "companyName": "company3",
            "countryName": "jp"
          },
          {
            "companyName": "company1",
            "countryName": "us"
          }
        ],
        "id": "person10",
        "@companyNames": [ "company3", "company1" ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "interestList": [ "art", "sport" ],
        "@companyCount": 2,
        "@numberOfRelationships": 4,
        "skillSet": [ 6, 8 ],
        "skillList": [ 8, 6 ],
        "locationId": "us",
        "interestSet": [ "sport", "art" ],
        "@info": [
          {
            "companyName": "company3",
            "countryName": "jp"
          },
          {
            "companyName": "company2",
            "countryName": "chn"
          }
        ],
        "id": "person7",
```

```
          "@companyNames": [ "company3", "company2" ]
        },
        "v_type": "person"
      },
      {
        "v_id": "person1",
        "attributes": {
          "interestList": [ "management", "financial" ],
          "@companyCount": 2,
          "@numberOfRelationships": 4,
          "skillSet": [ 3, 2, 1 ],
          "skillList": [ 1, 2, 3 ],
          "locationId": "us",
          "interestSet": [ "financial", "management" ],
          "@info": [
            {
              "companyName": "company2",
              "countryName": "chn"
            },
            {
              "companyName": "company1",
              "countryName": "us"
            }
          ],
          "id": "person1",
          "@companyNames": [ "company2", "company1" ]
        },
        "v_type": "person"
      },
      {
        "v_id": "person5",
        "attributes": {
          "interestList": [ "sport", "financial", "engineering" ],
          "@companyCount": 1,
          "@numberOfRelationships": 1,
          "skillSet": [ 5, 2, 8 ],
          "skillList": [ 8, 2, 5 ],
          "locationId": "can",
          "interestSet": [ "engineering", "financial", "sport" ],
          "@info": [{ "companyName": "company2", "countryName": "chn" }],
          "id": "person5",
          "@companyNames": ["company2"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person6",
        "attributes": {
```

```
          "interestList": [ "music", "art" ],
          "@companyCount": 1,
          "@numberOfRelationships": 1,
          "skillSet": [ 10, 7 ],
          "skillList": [ 7, 10 ],
          "locationId": "jp",
          "interestSet": [ "art", "music" ],
          "@info": [{ "companyName": "company1", "countryName": "us" }],
          "id": "person6",
          "@companyNames": ["company1"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person2",
        "attributes": {
          "interestList": ["engineering"],
          "@companyCount": 2,
          "@numberOfRelationships": 4,
          "skillSet": [ 6, 5, 3, 2 ],
          "skillList": [ 2, 3, 5, 6 ],
          "locationId": "chn",
          "interestSet": ["engineering"],
          "@info": [
            {
              "companyName": "company2",
              "countryName": "chn"
            },
            {
              "companyName": "company1",
              "countryName": "us"
            }
          ],
          "id": "person2",
          "@companyNames": [ "company2", "company1" ]
        },
        "v_type": "person"
      },
      {
        "v_id": "person8",
        "attributes": {
          "interestList": ["management"],
          "@companyCount": 1,
          "@numberOfRelationships": 1,
          "skillSet": [ 2, 5, 1 ],
          "skillList": [ 1, 5, 2 ],
          "locationId": "chn",
          "interestSet": ["management"],
          "@info": [{ "companyName": "company1", "countryName": "us" }]
```

Examples of Expression Statements

Expression Statement Examples

```
#Show various types of expression statements
CREATE QUERY expressionStmntEx() FOR GRAPH workNet {
  TYPEDEF tuple<STRING countryName, STRING companyName> companyInfo;

  ListAccum<companyInfo> @employerInfo;
  SumAccum<INT> @@a;
  ListAccum<STRING> @employers;
  SumAccum<INT> @employerCount;
  SetAccum<STRING> @@countrySet;

  int x;

  #exprStmnt := name "=" expr
  x = 10;

  #gAccumAssignStmt := globalAccumName ("+=" | "=") expr
  @@a = 10;

  PRINT x, @@a;

  start = {person.*};

  employees = SELECT s FROM start:s -(worksFor)-> :t
              ACCUM #exprStmnt := name "."localAccumName ("+="| "=") expr
                    s.@employers += t.id,
                    #exprStmnt := name ["<" type ["," type"]* ">"] "(" [a
                    s.@employerInfo += companyInfo(t.country, t.id),
                    #gAccumAccumStmt := globalAccumName "+=" expr
                    @@countrySet += t.country
                    #exprStmnt := name "."localAccumName ["."name "(" [ar
              POST-ACCUM s.@employerCount += s.@employers.size();

  #exprStmnt := globalAccumName ["."name "(" [argList] ")"]+
  PRINT @@countrySet.size();
  PRINT employees;
}
```

```
GSQL > RUN QUERY expressionStmntEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@a": 10,
      "x": 10
    },
    {"@@countrySet.size()": 4},
    {"employees": [
      {
        "v_id": "person4",
        "attributes": {
          "interestList": ["football"],
          "skillSet": [ 10, 1, 4 ],
          "skillList": [ 4, 1, 10 ],
          "locationId": "us",
          "@employerInfo": [{
            "companyName": "company2",
            "countryName": "chn"
          }],
          "interestSet": ["football"],
          "@employerCount": 1,
          "id": "person4",
          "@employers": ["company2"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person11",
        "attributes": {
          "interestList": [ "sport", "football" ],
          "skillSet": [10],
          "skillList": [10],
          "locationId": "can",
          "@employerInfo": [{
            "companyName": "company5",
            "countryName": "can"
          }],
          "interestSet": [ "football", "sport" ],
          "@employerCount": 1,
          "id": "person11",
```

```
        "@employers": ["company5"]
      },
      "v_type": "person"
    },
    {
      "v_id": "person10",
      "attributes": {
        "interestList": [ "football", "sport" ],
        "skillSet": [3],
        "skillList": [3],
        "locationId": "us",
        "@employerInfo": [
          {
            "companyName": "company3",
            "countryName": "jp"
          },
          {
            "companyName": "company1",
            "countryName": "us"
          }
        ],
        "interestSet": [ "sport", "football" ],
        "@employerCount": 2,
        "id": "person10",
        "@employers": [ "company3", "company1" ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "interestList": [ "art", "sport" ],
        "skillSet": [ 6, 8 ],
        "skillList": [ 8, 6 ],
        "locationId": "us",
        "@employerInfo": [
          {
            "companyName": "company3",
            "countryName": "jp"
          },
          {
            "companyName": "company2",
            "countryName": "chn"
          }
        ],
        "interestSet": [ "sport", "art" ],
        "@employerCount": 2,
        "id": "person7",
```

```
      "@employers": [ "company3", "company2" ]
    },
    "v_type": "person"
  },
  {
    "v_id": "person1",
    "attributes": {
      "interestList": [ "management", "financial" ],
      "skillSet": [ 3, 2, 1 ],
      "skillList": [ 1, 2, 3 ],
      "locationId": "us",
      "@employerInfo": [
        {
          "companyName": "company2",
          "countryName": "chn"
        },
        {
          "companyName": "company1",
          "countryName": "us"
        }
      ],
      "interestSet": [ "financial", "management" ],
      "@employerCount": 2,
      "id": "person1",
      "@employers": [ "company2", "company1" ]
    },
    "v_type": "person"
  },
  {
    "v_id": "person6",
    "attributes": {
      "interestList": [ "music", "art" ],
      "skillSet": [ 10, 7 ],
      "skillList": [ 7, 10 ],
      "locationId": "jp",
      "@employerInfo": [{ "companyName": "company1", "countryName": "
      "interestSet": [ "art", "music" ],
      "@employerCount": 1,
      "id": "person6",
      "@employers": ["company1"]
    },
    "v_type": "person"
  },
  {
    "v_id": "person2",
    "attributes": {
      "interestList": ["engineering"],
      "skillSet": [ 6, 5, 3, 2 ],
```

```
        "skillList": [ 2, 3, 5, 6 ],
        "locationId": "chn",
        "@employerInfo": [
          {
            "companyName": "company2",
            "countryName": "chn"
          },
          {
            "companyName": "company1",
            "countryName": "us"
          }
        ],
        "interestSet": ["engineering"],
        "@employerCount": 2,
        "id": "person2",
        "@employers": [ "company2", "company1" ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {
        "interestList": [ "sport", "financial", "engineering" ],
        "skillSet": [ 5, 2, 8 ],
        "skillList": [ 8, 2, 5 ],
        "locationId": "can",
        "@employerInfo": [{
          "companyName": "company2",
          "countryName": "chn"
        }],
        "interestSet": [ "engineering", "financial", "sport" ],
        "@employerCount": 1,
        "id": "person5",
        "@employers": ["company2"]
      },
      "v_type": "person"
    },
    {
      "v_id": "person12",
      "attributes": {
        "interestList": [
          "music",
          "engineering",
          "teaching",
          "teaching",
          "teaching"
        ],
        "skillSet": [ 2, 5, 1 ],
```

```
          "skillList": [ 1, 5, 2, 2, 2 ],
          "locationId": "jp",
          "@employerInfo": [{ "companyName": "company4", "countryName": "
          "interestSet": [ "teaching", "engineering", "music" ],
          "@employerCount": 1,
          "id": "person12",
          "@employers": ["company4"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person3",
        "attributes": {
          "interestList": ["teaching"],
          "skillSet": [ 6, 1, 4 ],
          "skillList": [ 4, 1, 6 ],
          "locationId": "jp",
          "@employerInfo": [{ "companyName": "company1", "countryName": "
          "interestSet": ["teaching"],
          "@employerCount": 1,
          "id": "person3",
          "@employers": ["company1"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person9",
        "attributes": {
          "interestList": [ "financial", "teaching" ],
          "skillSet": [ 2, 7, 4 ],
          "skillList": [ 4, 7, 2 ],
          "locationId": "us",
          "@employerInfo": [
            {
              "companyName": "company3",
              "countryName": "jp"
            },
            {
              "companyName": "company2",
              "countryName": "chn"
            }
          ],
          "interestSet": [ "teaching", "financial" ],
          "@employerCount": 2,
          "id": "person9",
          "@employers": [ "company3", "company2" ]
        },
        "v_type": "person"
```

```
        },
        {
          "v_id": "person8",
          "attributes": {
            "interestList": ["management"],
            "skillSet": [ 2, 5, 1 ],
            "skillList": [ 1, 5, 2 ],
            "locationId": "chn",
            "@employerInfo": [{ "companyName": "company1", "countryName": "
            "interestSet": ["management"],
            "@employerCount": 1,
            "id": "person8",
            "@employers": ["company1"]
          },
          "v_type": "person"
        }
      ]}
    ]
}
```

avg()

Syntax

`avg( [DISTINCT] setExp )`

Description

Returns the average of all elements in a set or bag. The function can only take set/bag expressions whose members are numeric types.

Return type

A numeric type. If all members of the set/bag expression are integers, the return value will also be rounded down to be an integer.

Parameters

Parameter	Description	Data type
`setExp`	An expression that evaluates to a `SET`, `BAG`, `SetAccum`, `BagAccum`, `ListAccum`	`SET`, `BAG`, `SetAccum`, `BagAccum, ListAccum`

Example

```
avg([5, 4, 1, 0, 0, 0]) -> 1
avg([3, 2, 1]) -> 2
```

count()

Syntax

`count( [DISTINCT] setExp )`

Description

Returns the size of the set or bag.

Return type

`INT`

Parameters

Parameter	Description	Data type
`setExp`	An expression that evaluates to a `SET`, `BAG`, `SetAccum`, `BagAccum`, `ListAccum`	`SET`, `BAG`, `SetAccum`, `BagAccum, ListAccum`

max()

Syntax

`max( [DISTINCT] setExp )`

Description

Returns the member with the maximum value in a set or bag. The function can only take set/bag expressions whose members are numeric types.

Return type

A numeric type.

Parameters

Parameter	Description	Data type
`setExp`	An expression that evaluates to a `SET`, `BAG`, `SetAccum`, `BagAccum`, `ListAccum`	`SET`, `BAG`, `SetAccum`, `BagAccum, ListAccum`

min()

Syntax

`min( [DISTINCT] setExp )`

Description

Returns the member with the minimum value in a set or bag. The function can only take set/bag expressions whose members are numeric types.

Return type

A numberic type.

Parameters

Parameter	Description	Data type
`setExp`	An expression that evaluates to a `SET`, `BAG`, `SetAccum`, `BagAccum`, `ListAccum`	`SET`, `BAG`, `SetAccum`, `BagAccum, ListAccum`

sum()

Syntax

```
sum( [DISTINCT] setExp )
```

Description

Returns the sum of all members in a set or bag. The function can only take set/bag expressions whose members are numeric types.

Return type

A numeric type.

Parameters

Parameter	Description	Data type
`setExp`	An expression that evaluates to a `SET`, `BAG`, `SetAccum`, `BagAccum`, `ListAccum`	`SET`, `BAG`, `SetAccum`, `BagAccum, ListAccum`

Loading...

Loading...

Loading...

Loading...

Mathematical Functions

This page lists the mathematical functions that are available in the GSQL query language. They are divided into three categories:

- General
- Logarithmic
- Trigonometric

General

abs()

Syntax

`abs( num )`

Description

Returns the absolute value of a number.

Return type

Number

Parameters

Parameter	Description	Data type
`num`	The number to return the absolute value for	Number

ceil()

Syntax

`ceil(num)`

Description

Rounds a number up to the smallest integer that's greater than or equal to the number.

Return type

`INT`

Parameters

Parameter	Description	Data type
`num`	The number to round up from	num

exp()

Syntax

`exp(num)`

Description

Returns the base-e exponential of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The exponent	Number

float_to_int()

Syntax

`float_to_int (num)`

Description

Converts a floating-point number to an integer by truncating the floating part.

Return type

`INT`

Parameters

Parameter	Description	Data type
`num`	The floating-point number to convert to integer	`FLOAT`

floor()

Syntax

`floor(num)`

Description

Rounds a number down to the biggest integer that is smaller than or equal to the number.

Return type

`INT`

Parameter

Parameter	Description	Data type
`num`	The number to round down from	Number

fmod()

Syntax

`fmod(numer, denom)`

Description

Returns the floating-point remainder of `numer` divided by `denom`

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`numer`	The dividend	Number
`denom`	The divisor	Number

ldexp()

Syntax

`ldexp(x, exp)`

Description

Returns `x` multiplied by 2 raised to the power of `exp`

$$x * 2^{exp}$$

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`x`	The base	Number
`exp`	The exponent of 2	Number

PI()

Syntax

`PI()`

Description

Returns the value of π.

Return type

`DOUBLE`

Parameters

None.

Example

```
PI() * 1000000000 -> 3.141592653589793E9
```

pow()

Syntax

`pow(base, exp)`

Description

Returns the power of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`base`	The base	Number
`exp`	The exponent	Number

rand()

Syntax

`rand( [seed] )`

Description

Returns a completely random number >= 0 and <1. If `seed` is specified, it returns a repeatable sequence of random numbers. If no seed is specified, it returns a completely random number.

Return type

`DOUBLE`

Parameters

Parameter	Description	Data type
`seed`	Optional. If `seed` is specified, it returns a repeatable sequence of random numbers. If no seed is specified, it returns a completely random number	`UINT`

Example

```
rand(5) -> 0.05518
rand(5) -> 0.83133
rand(5) -> 0.36374
```

round()

Syntax

`round ( num[, integer] )`

Description

Rounds a number to a specified place relative to the decimal point and returns the result.

Return type

A numeric type.

Parameters

Parameter	Description	Data type
`num`	The number to be rounded	`NUM`
`integer`	Optional. An integer value indicating the place to round the first argument to. If `integer` is positive, the function returns `num` rounded to `integer` places to the right of the decimal point. If you omit `integer`, then `num` is rounded to zero places. If `integer` is negative, then	`INT`

is rounded off to the

Examples

```
round(15.213) => 15
round(15.213, -1) => 20
round(2.15, 1) => 2.2
round(2.25, 1) => 2.3
```

sign()

Syntax

`sign( num )`

Description

Returns the sign of a number. If the number is positive, return `1`; if the number is negative, return `-1`; if the number is `0`, return `0`

Return type

`INT`

Parameters

Parameter	Description	Data type
`num`	A numeric value	`INT`, `DOUBLE`

Examples

```
sign(100) => 1
sign(0) => 0
sign(-1.23) => -1
```

square()

Syntax

`square( num )`

Description

Returns the square of a number.

Return type

A numeric type.

Parameters

Parameter	Description	Data type
`num`	A numeric value.	`INT`, `FLOAT`, or `DOUBLE`

Examples

```
square(0) => 0
square(50) => 2500
square(-50) => 2500
```

sqrt()

Syntax

`sqrt(num)`

Description

Returns the square root of a number

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to get square root for.	Number

trunc()

Syntax

`trunc( num, [decimal_place] )`

Description

Returns a number truncated to a specified decimal place.

Return type

A numeric type.

Parameters

Parameter	Description	Data type
`num`	The number to be truncated	`INT`, `FLOAT` or `DOUBLE`
	Optional. The integer indicating the decimal place to truncate the number to.	

`decimal_place`	If `decimal_plac` is positive, the function returns the number truncated to `decimal_place` decimal places. If `decimal_place` is omitted, then the number is truncated to 0 places. `decimal_place` can be negative to truncate (make zero) `decimal_place` digits left of the decimal point.	`INT`

Examples

```
trunc(9.99) => 9
trunc(-9.99) => 9
trunc(99.999. -1) => 90
trunc(9.99, 1) => 9.9
```

Logarithmic

log()

Syntax

`log(num)`

Description

Returns the natural logarithm of a number (base e).

Return type

`DOUBLE`

Parameters

Parameter	Description	Data type
`num`	The number to compute natural logarithm for	Number

log2()

Syntax

`log2( num )`

Description

Returns the base-2 logarithm of a number.

Return type

`DOUBLE`

Parameters

Parameter	Description	Data type
`num`	A numeric value	`INT`, `FLOAT`, `DOUBLE`

Examples

```
log2(0.5) => -1
log2(1) => 0
log2(3) => 1.58
```

log10()

Syntax

`log10(num)`

Description

Return the common logarithm of a number (base 10).

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute common logarithm for	Number

Trigonometric

acos()

Syntax

`acos(num)`

Description

Returns the arc cosine of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute arccosine for	Number

asin()

Syntax

`asin(num)`

Description

Returns the arc sine of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute arcsine for	Number

atan()

Syntax

`atan(num)`

Description

Returns the arctangent of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute arctangent for	Number

atan2()

Syntax

`atan2(y, x)`

Description

Returns the arctangent of a fraction.

$$atan(\frac{y}{x})$$

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`y`	The dividend of the fraction to compute arctangent for	Number
`x`	The divisor of the fraction to compute arctangent for	Number

cos()

Syntax

`cos(num)`

Description

Returns the cosine of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to return cosine for	Number

cosh()

Syntax

`cosh(num)`

Description

Returns the hyperbolic cosine of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute hyperbolic cosine for	Number

cot()

Syntax

`cot( num )`

Description

Returns the cotangent of a number.

Return type

`DOUBLE`

Parameters

Parameter	Description	Data type
`num`	A numeric value	`INT`, `FLOAT`, or `DOUBLE`

Examples

```
cot(6) => -3.4363530041801278
cot(-1) => -0.64209261593433065
```

degrees()

Syntax

`degrees( num )`

Description

Converts a value in radians to degrees.

Return type

`DOUBLE`

Parameters

Parameter	Description	Data type
`num`	A numeric value	`INT`, `FLOAT`, or `DOUBLE`

Examples

```
degrees(2) => 114.59155902616465
degrees(1) => -57.29577951308232
```

radians()

Syntax

`radians( num )`

Description

Converts a value in degrees to radians.

Return type

`DOUBLE`

Parameters

Parameter	Description	Data type
`num`	A numeric value	`INT`, `FLOAT`, or `DOUBLE`

Examples

```
radians( 45 ) => -0.7853981633974483
radians( 30 ) => 0.5235987755982988
radians( 50 ) => 0.8726646259971648
```

sin()

Syntax

`sin(num)`

Description

Returns the sine of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute sine for	Number

sinh()

Syntax

`sinh(num)`

Description

Returns the hyperbolic sine of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute hyperbolic sine for	Number

tan()

Syntax

`tan(num)`

Description

Returns the tangent of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute tangent for	Number

tanh()

Syntax

`tanh(num)`

Description

Returns the hyperbolic tangent of a number.

Return type

`FLOAT`

Parameters

Parameter	Description	Data type
`num`	The number to compute hyperbolic tangent for	Number

String Functions

This page lists the string functions available in the GSQL query language

ascii()

Syntax

`ascii( str )`

Description

Returns the ASCII (numeric) position of the first character in a string. If the argument is an empty string, returns 0.

Return type

`INT`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`

Example

```
ascii("") => 0
ascii("A") => 65
```

chr()

Syntax

`chr( n )`

Description

Converts an integer to a character according to its ASCII position. If the input value `n` is greater than 255, returns the character at the position of the modulo of `n / 256`.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`n`	An integer value	`INT`

Examples

```
chr(65) => 'A'
chr(322) => 'B'
```

difference()

Syntax

`difference(str1, str2)`

Description

Compares the Soundex codes of two strings and returns an integer. The return value ranges from 0 to 4, and it indicates the similarity between the input strings' phonetic representation values.

0 indicates weak similarity, and 4 indicates strong similarity with identical phonetic representation values.

Return type

`INT`

Parameters

Parameter	Description	Data type
`str1`	A string value	`STRING`
`str2`	A string value	`STRING`

Examples

```
difference("Johnson", "Jonson") => 4
difference("Adams", "Addamms") => 4
difference("Mary", "Bob") => 2
```

find_in_set()

Syntax

`find_in_set(str, str_list)`

Description

Returns the position of a string within a list of strings separated by commas.

If `string` is not found in `string_list`, this function returns -1.

> **NOTE**: `find_in_set` does *not* ignore whitespace after the comma. For example, `find_in_set("a", "b a, a") = 0`, indicating that “a” is not an element in the list. This happens because the second and third elements of the list are space+a, rather than a.

Return type

`INT`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`
`str_list`	A string representation of a list of strings.	`STRING`

Example

```
find_in_set("a","") => 0
find_in_set("a","b,c,d") => 0
find_in_set("a","b,a,d") => 2
```

gsql_uuid_v4()

Syntax

`gsql_uuid_v4()`

Description

Generates and returns a version-4 universally unique identifier(UUID) ↗.

Return type

`STRING`

Parameters

None.

insert()

Syntax

`insert(str1, position[, number], str2)`

Description

Inserts a string within a string at the specified position and for a certain number of characters, and replaces a specified number of characters starting from the insertion position. The starting index is 0.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str1`	The string to insert another string into	`STRING`
`position`	The index of the starting position to insert the string	`INT`

`number`	Optional. The number of characters from the original string that will be replaced. If the argument is left off, it defaults to 0.	`STRING`
`str2`	The string to be inserted	`STRING`

Examples

```
insert("tigergraph.com", 0, 10, "Example") => "Example.com”
insert("tigergraph.com", 0, 2, "Example") => "Examplegergraph.com”
insert("tigergraph.com", 2, 20, "Example") => ”tiExample”
insert("Complete blank.", 9, "every ") => "Complete every blank."
```

instr()

Syntax

`instr (str, substr [, position, occurence])`

Description

Searches a string (`str`) for a substring (`substr`) and returns the location of the substring in the string. If a substring that is equal to `substr` is found, then the function returns an integer indicating the position of the first character of this substring. If no such substring is found, then the function returns -1.

Return type

`INT`

Parameters

Parameter	Description	Data type

`str`	The string to search	`STRING`
`substr`	The string to search for in `str`	`STRING`
`position`	Optional. The position is a nonzero integer indicating the character of `str` from where the search begins. If omitted, it defaults to 0. The first position in the string is 0. If `position` is negative, then the function counts backward from the end of `str` and then searches backward from the resulting position.	`STRING`
`occurrence`	Optional, The occurrence is an integer indicating which occurrence of `substr` in `str` the function should	`STRING`

Example

```
instr("This is the thing", "Th") -> 0;
instr("This is the thing", "is", 3) -> 5;
```

left()

Syntax

`left(str, number_of_chars)`

Description

Extracts a number of characters from a string starting from left.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`
`number_of_chars`	The number of characters to extract	`INT`

length()

Syntax

`length(str)`

Description

Returns the length of the input string.

Return type

`INT`

Parameters

Parameter	Description	Data type
`str`	The string whose length to evaluate	`STRING`

Example

```
length("hello world") -> 11
length("") -> 0
```

ltrim()

Syntax

`ltrim( str[, set] )`

Description

Removes all occurrences of the characters contained in a set from a string from the left side.

The function begins scanning the string from its first character and removes all characters that appear in `set` *until reaching a character not in* `set` *and then returns the result.*

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`
`set`	Optional. A string of characters. The distinct characters from the string form the set. If not specified, it defaults to a single space.	`STRING`

lower()

Syntax

`lower(str)`

Description

Returns the input string with all letters in lowercase.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	The string to convert to lowercase	`STRING`

Example

```
lower("GSQL") -> "gsql"
```

lpad()

Syntax

`lpad(str, padded_length [, pad_str] )`

Description

Pads the left side of a string with another pad string. If the pad string (`pad_str`) is omitted, it will pad with white space. If the parameter length is smaller than the original string, it will truncate the string from the right side.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	The string to pad characters to	`STRING`
`padded_length`	The number of characters to return. If the `padded_length` is smaller than the original string, the `lpad` function will truncate the string to the size of `padded_length`.	`INT`
`pad_str`	Optional. This is the string that will be padded to the left-hand side of `str`. If this parameter is omitted, the `lpad` function will pad spaces to the left-side of `str`.	`STRING`

Example

```
lpad("PQR", 5) -> "  PQR"
lpad("PQR", 2) -> "PQ"
lpad("PQR", 10, "ABC") -> "ABCABCAPQR"
```

replace()

Syntax

`replace(str, str_to_replace [, replacement_str])`

Description

Replaces a sequence of characters in a string with another set of characters.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	The original string whose substrings are to be replaced	`STRING`
`str_to_replace`	The string that will be searched for and replaced in `str`	`STRING`
`replacement_str`	Optional. The string that will replace `str_to_replace`. If omitted, `replace()` removes all occurrences of *string_to_replace*, and returns the resulting string.	`STRING`

Examples

```
replace("SSQLL", "S", "G") -> "GGQLL"
replace("SSQLL", "SQL", "Q") -> "SQL"
replace("SSQLL", "L") -> "SSQ"
```

right()

Syntax

`right(str, number_of_chars)`

Description

Extracts a number of characters from a string starting from the right.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`
`number_of_chars`	The number of characters to extract	`INT`

rpad()

Syntax

`rpad(str, padded_length [, pad_str] )`

Description

Pads the right side of a string (`str`) with another pad string. If the pad string (`pad_str`) is omitted, it will pad with white space. If the parameter length is smaller than the original string, it will truncate the string from the right side.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	The string to pad characters to	`STRING`
`padded_length`	The number of characters to return. If the `padded_length` is smaller than the original string, the `lpad` function will truncate the string to the size of `padded_length`.	`INT`
`pad_str`	Optional. This is the string that will be padded to the right-hand side of `str`. If this parameter is omitted, the `lpad` function will pad spaces to the right-side of `str`.	`STRING`

Example

```
rpad("PQR", 5) -> "PQF  "
lpad("PQR", 2) -> "PQ"
lpad("PQR", 10, "ABC") -> "ABCABCAPQR"
```

rtrim()

Syntax

`rtrim( str [,set] )`

Description

Removes all occurrences of the characters contained in a set from a string from the right side.

The function begins scanning the string from its last character and removes all characters that appear in `set` *until reaching a character not in* `set` *and then returns the result.*

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`
`set`	Optional. A string of characters. The distinct characters from the string form the set. If not specified, it defaults to a single space.	`STRING`

soundex()

Syntax

`soundex( str )`

Description

Returns a character string containing the Soundex ↗ code of `str`. This function lets you compare words that are spelled differently, but sound alike in English.

Soundex is a phonetic algorithm defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

1. Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
2. Assign numbers to the remaining letters (after the first) as follows:
   ```
   b, f, p, v = 1
   c, g, j, k, q, s, x, z = 2
   d, t = 3
   l = 4
   m, n = 5
   r = 6
   ```
3. If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, then retain the first letter and omit the rest of all the adjacent letters with the same number.
4. Return the first four bytes padded with 0.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`

Examples

```
soundex("Ashcraft") => "A261"
soundex("Burroughs") => "B620"
soundex("Burrows") => "B620"
```

space()

Syntax

`space( n )`

Description

Returns a string that contains the specified number of space characters

Return type

`STRING`

Parameters

Parameter	Description	Data type
`n`	An integer value	`INT`

Examples

```
space(0) = ""
space(1) = " "
space(5) = "     "
```

substr()

Syntax

`substr(str, start [, length])`

Description

Returns the substring indicated by the start point and length. If the parameter length is omitted, then it will extend to the end.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	The string to extract substring from	`STRING`
`start`	The position that indicates the start of the substring	`INT`
`length`	Optional. The length of the substring. If omitted, the substring will be	`INT`

Example

```
substr("ABCDE", 2) -> "CDE"
substr("ABCDE", 2, 2) -> "CD"
substr("ABCDE", -2, 1) -> "D"
```

translate()

Syntax

`translate( str_origin, characters, translations )`

Description

Returns the string from the first argument after the characters specified in the second argument are translated into the characters specified at the same index in the third argument.

> **NOTE**: The function will return an error if `characters` and `translations` have different lengths.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str_origin`	A string value	`STRING`
`characters`	A string of characters	`STRING`
`translations`	A string of characters	`STRING`

Examples

```
translate(”Hello world”, "", "") = ”Hello World”
translate(”Hello world”, "o", "U") = ”HellU WUrld”
translate(”Hello world”, "lo", "aU") = ”HeaaU WUrad”
translate(””, "lo", "aU") = ””
```

trim()

Syntax

`trim( [ [ LEADING | TRAILING | BOTH ] [removal_char FROM] ] str )`

Description

Trims characters from the leading and/or trailing ends of a string.

By using one of the keywords `LEADING`, `TRAILING`, or `BOTH`, the user can specify that characters are to be removed from the left end, right end, or both ends of the string, respectively. If none of these keywords is used, the function will remove from both ends.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`removal_char`	Optional. The character to remove. If `removal_char` is not specified, the function will remove whitespaces, including spaces, tabs, and newlines. If `removal_char` is specified, the user must also write the keyword `FROM` between `removal_char` and `str`.	`STRING`
`str`	A string value.	`STRING`

Example

```
trim("  Abc   ") => "Abc"
trim( LEADING " a A   ") => "a A   "
trim( TRAILING "a" FROM "aa ABC aaa") => "aa ABC "
```

upper()

Syntax

`upper(str)`

Description

Returns the input string with all letters in uppercase.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`str`	The string to convert to uppercase	`STRING`

Example

```
upper("gsql") -> "GSQL"
```

Type Conversion Functions

The functions on this page are used to convert data from one type to another in the GSQL query language.

`parse_json_array()`

Syntax

`parse_json_array( str )`

Description

Converts a string into a JSON array. The string must be properly formatted, or the function will generate a run-time error. To be properly formatted, besides having the proper nesting and matching of curly braces `{ }` and brackets `[ ]`, each value field must be one of the following:

- a string
- a number
- a boolean
- a JSONOBJECT - Each key of a key-value pair must be a string in double quotes.
- a JSON array

Return type

`JSONARRAY`

Parameters

Parameter	Description	Data type

	The string to be converted	

Example

```
parse_json_array("[123]") -> [123]
```

`parse_json_object()`

Syntax

`parse_json_object( str )`

Description

Converts a string into a JSON object. The string must be properly formatted, or the function will generate a run-time error. To be properly formatted, besides having the proper nesting and matching of curly braces `{ }` and brackets `[ ]`, each value field must be one of the following:

- a string
- a number
- a boolean
- a JSONOBJECT - Each key of a key-value pair must be a string in double quotes, and the quotes need to be escaped with a backlash-escape `\`. However, if you are supplying the string in GraphStudio as a parameter, you do not need the backlash-escape since string values are not enclosed in double quotes.
- a JSON array

Return type

`JSONOBJECT`

Parameters

Parameter	Description	Data type
`str`	The string to be converted into a JSON object.	`STRING`

Example

```
parse_json_object("{\"abc\":123}")  -> {"abc": 123}
```

`str_to_int ()`

Syntax

`str_to_int (str)`

Description

Converts a string to an integer.

Return type

`INT`

Parameters

Parameter	Description	Data type
`str`	The string that contains an integer to convert to an integer	`STRING`

`to_datetime ()`

Syntax

`to_datetime ( str )`

Description

Converts a string value into a `DATETIME` value.

Return type

`DATETIME`

Parameters

Parameter	Description	Data type
`str`	A string value	`STRING`

Example

```
to_datetime("2020-01-02 01:02:03") -> 2020-01-02 01:02:03
```

`to_string()`

Syntax

`to_string(num)`

Description

Converts a number to a string.

Return type

`STRING`

Parameters

Parameter	Description	Data type
`num`	The number to turn into a string	Number

Vertex Functions

This page lists the vertex functions that are available in the GSQL query language. The functions are divided into three categories.

- Vertex alias methods
  - Methods available to vertex aliases
- Vertex-level access control(VLAC) vertex alias methods
  - Methods available to vertex aliases that are related to vertex tags.
- Vertex functions
  - Functions that return a vertex or vertex set, or functions that are closely related to certain attributes of `VERTEX` type variables.

Vertex alias methods

This section lists the built-in methods of vertex aliases. Methods can be accessed by the dot (`.`) operator.

`edgeAttribute()`

Syntax

`v.edgeAttribute( edgeType, attrName )`

Description

From a vertex, traverse edges of a specified type and return the bag of values for a specified edge attribute.

Return type

`BagAccum<attrType>`

Parameters

Parameter	Description	Data type
`edgeType`	The edge type to traverse	`STRING`
`attrName`	The attribute whose value to retrieve	`STRING`

`filter()`

Syntax

```
v.neighbors().filter( condition )
v.neighborAttribute().filter( condition )
v.edgeAttribute().filter( condition )
```

Description

This function is appended to `neighbors()`, `neighborAttribute()`, or `edgeAttribute()` to filter the output set according to a filter condition. Only elements that satisfy the condition will be returned.

Return type

`BagAccum`

Parameters

Parameter	Description	Data type
`condition`	An expression that evaluates to a boolean value	`BOOL`

Example

Example query

```
CREATE QUERY filterEx (SET<STRING> pIds, INT yr) FOR GRAPH workNet api("v

  SetAccum<vertex<company>> @recentEmplr, @allEmplr;
  BagAccum<string> @diffCountry, @allCountry;

  Start = {person.*};

  L0 = SELECT v
       FROM  Start:v
       WHERE v.id IN pIds
       ACCUM
         # filter using edge attribute
         v.@recentEmplr += v.neighbors("worksFor").filter(worksFor.startY
         v.@allEmplr += v.neighbors("worksFor").filter(true),

        # vertex alias attribute and neighbor type attribute
        v.@diffCountry += v.neighborAttribute("worksFor", "company", "id"
                          .filter(v.locationId != company.country),
        v.@allCountry += v.neighborAttribute("worksFor", "company", "id")
       ;

  PRINT yr, L0[L0.@recentEmplr, L0.@allEmplr, L0.@diffCountry, L0.@allCou
}
```

Results

```
GSQL > RUN QUERY filterEx(["person1","person2"],2016)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "L0": [
      {
        "v_id": "person1",
        "attributes": {
          "L0.@diffCountry": ["company2"],
          "L0.@recentEmplr": ["company1"],
          "L0.@allCountry": [ "company1", "company2" ],
          "L0.@allEmplr": [ "company2", "company1" ]
        },
        "v_type": "person"
      },
      {
        "v_id": "person2",
        "attributes": {
          "L0.@diffCountry": ["company1"],
          "L0.@recentEmplr": [],
          "L0.@allCountry": [ "company1", "company2" ],
          "L0.@allEmplr": [ "company2", "company1" ]
        },
        "v_type": "person"
      }
    ],
    "yr": 2016
  }]
}
```

`getAttr()`

Syntax

`v.getAttr(attrName, attrType)`

Description

Returns the value of a vertex attribute on the vertex.

Return type

`attrType`

Parameters

Parameter	Description	Data type
`attrName`	A vertex attribute	`STRING`
`attrType`	The type of the attribute value	`STRING`

`neighborAttribute()`

Syntax

`v.neighborAttribute( edgeType, targetVertexType, attrName )`

Description

From a vertex, traverses edges of a specified type to its neighbors of a specified type, and returns the set of values for a specified attribute.

Return type

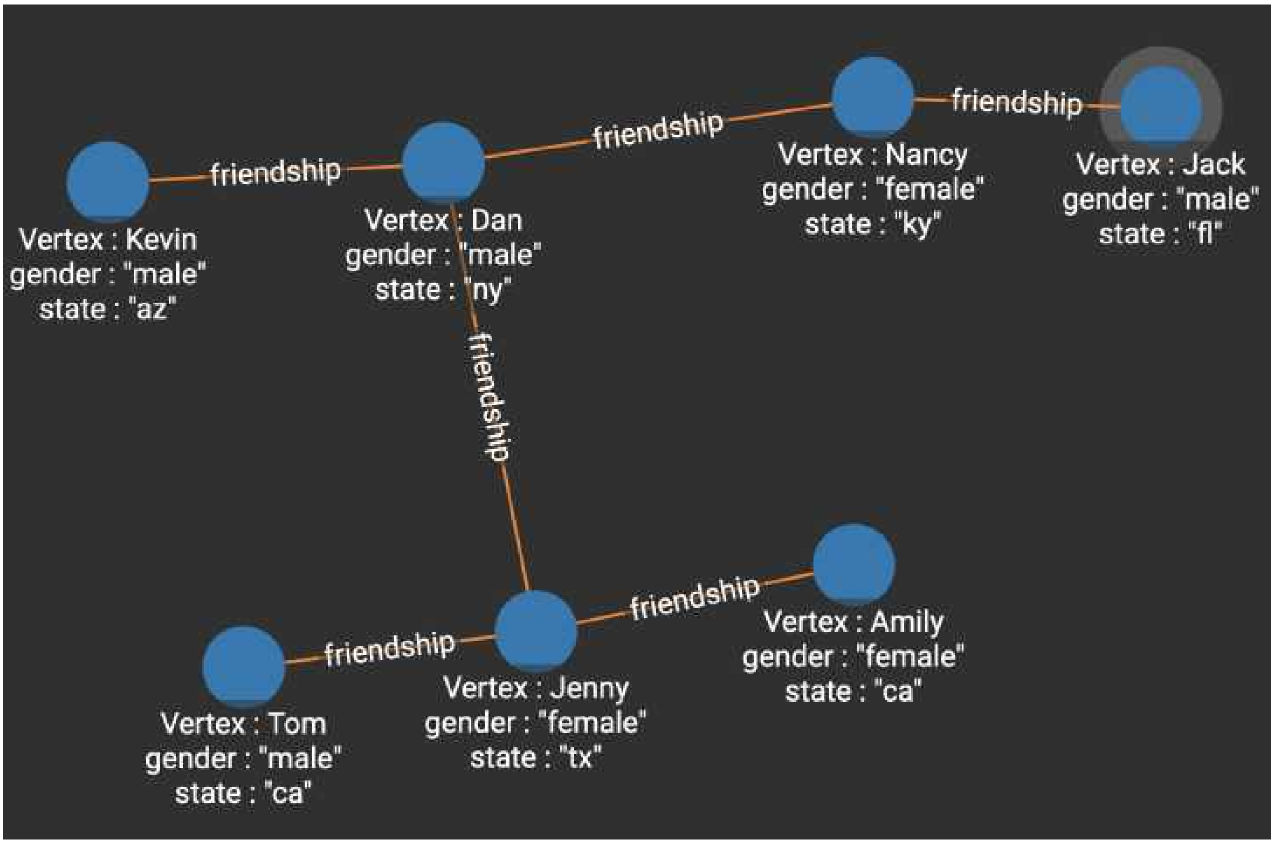
`BagAccum<attrType>`

Parameters

Parameter	Description	Data type
`edgeType`	The edge type to traverse	`STRING`
`targetVertexType`	The target vertex type to visit	`STRING`
`attrName`	An attribute of the target vertex type	`STRING`

Example

For the following graph:




```
# If v is Jenny
v.neighborAttribute("friendship", "person", "state") -> ["ca", "ny", "ca"
```

`neighbors()`

Syntax

`v.neighbors([edgeType])`

Description

Returns the out-neighbors or undirected neighbors of the vertex. If edge types are provided, it will only return the neighbors connected by the specified edge types.

Return type

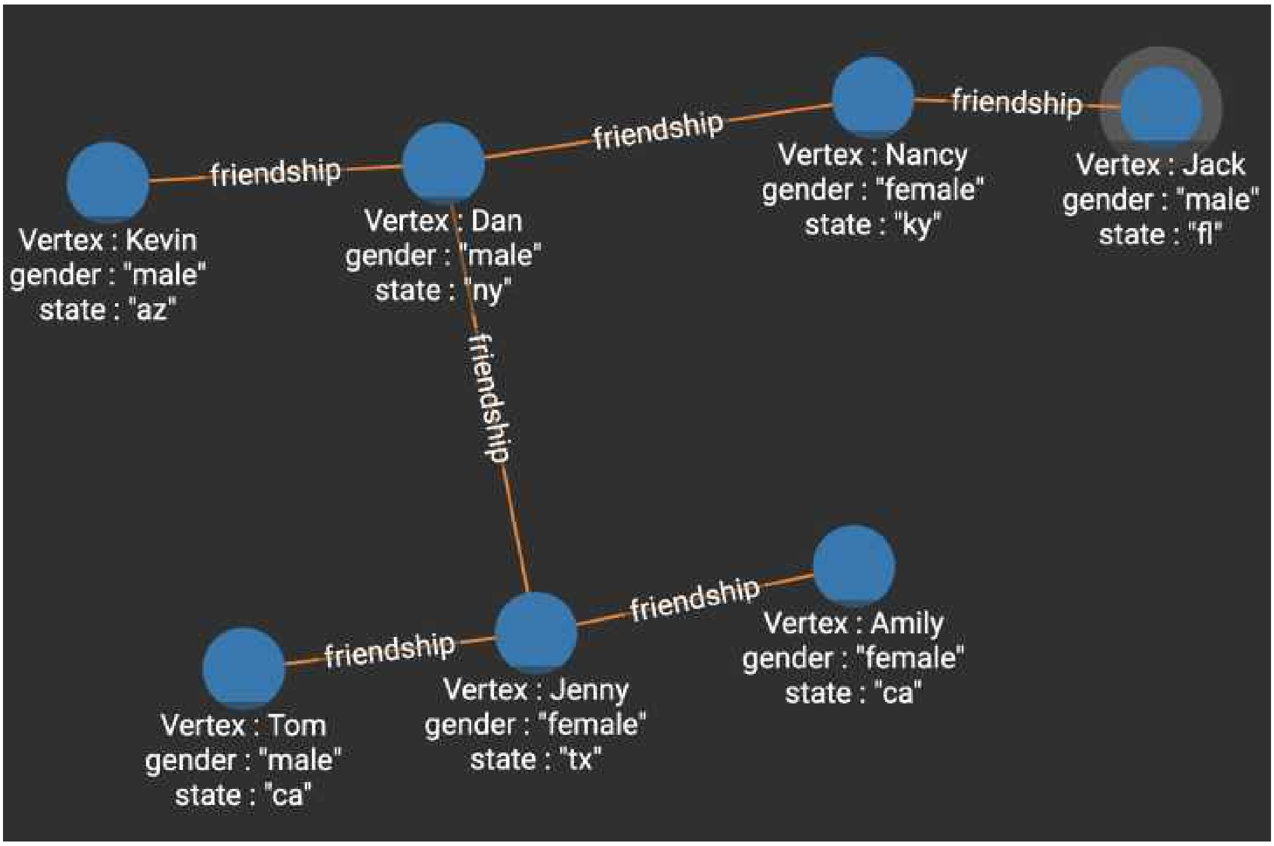
`BagAccum<VERTEX>`

Parameters

Parameter	Description	Data type
`edgeType`	Optional. An edge type or a collections of edge types.	`STRING`, `SET<STRING>`, `SetAccum<STRING>`, `BagAccum<STRING>`, `ListAccum<STRING>`

Example

For the following graph:




```
# If v is Jenny
v.neighbors() -> ["Dan", "Amily", "Tom"]
```

`outdegree()`

Syntax

`v.outdegree([edgeType])`

Description

Returns the number of outgoing or undirected edges connected to the vertex. If edge types are provided, it will only return the number of edges of the specified types.

Return type

`INT`

Parameters

Parameter	Description	Data type
`edgeType`	Optional. An edge type or a collection of edge types.	`STRING`, `SET<STRING>`, `SetAccum<STRING>`, `BagAccum<STRING>`, `ListAccum<STRING>`

Note on outdegree(): This function reads a metadata value stored with each vertex, to avoid traversing the graph and thus have a fast response. The snapshot transaction semantics means that outdegree() may sometimes read an old value if there are concurrent write transactions. To guarantee an accurate count, traverse the neighboring edges and count them with a SumAccum, or use a function like neighbors() and then use size() on the set.

`setAttr()`

Syntax

`v.setAttr( attrName, newValue )`

Description

Sets the specified attribute of a vertex to a new value.

Return type

No return value.

Parameters

Parameter	Description	Data type
`attrName`	The name of an attribute	`STRING`

`newValue`	The new value for the attribute	The type of the attribute

VLAC vertex alias methods

This section covers the vertex alias methods used to access and modify tags on vertices,

`addTags()`

Syntax

`v.addTags(STRING tag1,... STRING tagN)`

Description

Adds the tags provided in the argument list to the vertex.

Return type

No return value.

Parameters

Parameter	Description	Data type
`tagN`	A tag to add to the vertex	`STRING`

Example:

```
CREATE QUERY addTagsToPerson() {
  Seed = { any };
  # person1 ~ person5 will be tagged as public.
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id IN ("person1","person2","person3","person4","person5"
         ACCUM s.addTags("public");

  # person6 and person7 will be tagged as public and vip.
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id IN ("person6","person7")
         ACCUM s.addTags("vip", "public");

  # person8 will be tagged as vip
  vSet = SELECT s
         FROM Seed:s
         WHERE s.id == "person8"
         ACCUM s.addTags("vip");
}
```

`differenceTags()`

Syntax

`v.differenceTags( v2 )`

Description

Returns the difference in tags between the vertex and another vertex as a set.

Return type

`SET<STRING>`

Paramters

Parameter	Description	Data type

`v2`	A vertex	`VERTEX`

Example:

```
// return the difference set of tags between two vertices
CREATE QUERY exampleDifferencetags() {
  SetAccum<string> @vAcc;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s -(_)-> :t
         WHERE t.type == "person"
         ACCUM s.@vAcc += s.differenceTags(t);
  PRINT vSet[vSet.@vAcc];
}
```

`getTags()`

Syntax

`v.getTags()`

Descriptions

Returns the vertex's tags as a set. If the vertex has no tags or is untaggable, it returns an empty set.

Return type

`SET<STRING>`

Parameters

None.

Example:

```
//print the tags of each vertices, in 2 different ways.
CREATE QUERY exampleGettags() {
  SetAccum<string> @vAcc;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.@vAcc += s.getTags();
  PRINT vSet[vSet.@vAcc];
  PRINT vSet[vSet.gettags()];
}
```

Tip: `getTags()` can be used within a `PRINT` statement:

- `PRINT R [R.getTags()];`
- or `PRINT R WITH TAGS` which is syntax sugar, except that it won't print `"R.getTags()": []` for non-taggable vertices.

`hasTags()`

Syntax

`hasTags( tag1, tag2, ..., tagN )`

Description

Returns `true` if the vertex has every tag provided in the argument list and returns `false` if it does not.

Return type

`BOOL`

Parameters

Parameter	Description	Data type

Example:

```
USE GRAPH socialNet

CREATE QUERY findVertexWithTag(STRING tag) {
  seed = { ANY };
  res =
    SELECT v
    FROM seed:v
    WHERE v.hasTags(tag)
    ORDER BY v.id;
  PRINT res WITH TAGS;
}

INSTALL QUERY findVertexWithTag

RUN QUERY findVertexWithTag("vip")
```

The output of the query would be:

```
{
  "error": false,
  "message": "",
  "version": {
    "schema": 2,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"res": [
    {
      "v_id": "person6",
      "attributes": {
        "gender": "Male",
        "id": "person6",
        "res.gettags()": [
          "vip",
          "public"
        ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "id": "person7",
        "res.gettags()": [
          "vip",
          "public"
        ]
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {
        "gender": "Male",
        "id": "person8",
        "res.gettags()": ["vip"]
      },
      "v_type": "person"
    }
  ]}]
}
```

`isTaggable()`

Syntax

`v.isTaggable()`

Description

Returns true if the vertex is taggable.

Return type

`BOOL`

Parameters

None

Example:

```
//count the number of taggable vertices in the graph.
CREATE QUERY countIstaggable() for graph poc_graph_tag {
  SumAccum<int> @@count;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s
         WHERE s.isTaggable()
         ACCUM @@count += 1;
  PRINT @@count;
}
```

`intersectTags()`

Syntax

`v.intersectTags( v2 )`

Description

Returns the common tags between the vertex and another vertex as a set.

Return type

`SET<STRING>`

Example:

```
//return the intersect set of tags between two vertices.
CREATE QUERY exampleIntersecttags() {
  SetAccum<string> @vAcc;
  vSet = { any };
  vSet = SELECT s
         FROM vSet:s -(_)-> :t
         WHERE t.type == "person"
         ACCUM s.@vAcc += s.intersectTags(t);
  PRINT vSet[vSet.@vAcc];
}
```

`removeAllTags()`

Syntax

`v.removeAllTags()`

Description

Removes all tags from the vertex.

Return type

No return value.

Parameters

None

Example:

```
//remove all tags from all person vertices.
CREATE QUERY removealltagsFromPerson() {
  vSet = { person.* };
  # remove all tags from all person vertices
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.removeAllTags();
}
```

`removeTags()`

Syntax

`removeTags( tag1, tag2, ..., tagN )`

Description

Removes the tags provided in the argument list from the vertex.

Return type

No return value.

Parameters

Parameter	Description	Data type
`tagN`	A string value	`STRING`

Example

```
//remove tag “vip” and “public” from all person vertices.
CREATE QUERY removetagsFromPerson() {
  vSet = { person.* };
  # remove tag vip and public from all person vertices
  vSet = SELECT s
         FROM vSet:s
         ACCUM s.removeTags("vip", "public");
}
```

Vertex functions

The functions in this section either have return values of vertex or vertex set type or are closely related to vertex attributes.

`getvid()`

Syntax

`getvid( v )`

Description

Returns the *internal* ID number of a vertex.

The internal ID is not the primary ID which the user assigned when creating the vertex. However, there is a 1-to-1 mapping between the external ID (`primary_id`) and internal ID.

The engine can access the internal ID faster than accessing the external ID, so if a query needs unique values for a large number of vertices, but doesn't care about particular values, `getvid()` can be a useful option. For example, in many community detection algorithms, we start by assigning every vertex a unique community ID. Then, as the algorithm progresses, some vertices will join the community of one of their neighbors, giving up their current community ID and copying the IDs of their neighbors.

Return type

`INT`

Parameters

Parameter	Description	Data type
`v`	A vertex alias.	Vertex alias

Example

Query

```
CREATE QUERY getvid_ex () FOR GRAPH socialNet {
  MinAccum<int> @cc_id = 0;       //each vertex's tentative component

  Start = {person.*};
  # Initialize: Label each vertex with its own internal ID
  S = SELECT x FROM Start:x
      POST-ACCUM
         x.@cc_id = getvid(x);

  # Community detection steps omitted
  PRINT Start.@cc_id;
}
```

Result

```
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Start": [
    {
      "v_id": "person7",
      "attributes": {"Start.@cc_id": 0},
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {"Start.@cc_id": 4194304},
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {"Start.@cc_id": 4194305},
      "v_type": "person"
    },
    {
      "v_id": "person4",
      "attributes": {"Start.@cc_id": 11534336},
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {"Start.@cc_id": 13631488},
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {"Start.@cc_id": 20971520},
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {"Start.@cc_id": 22020096},
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {"Start.@cc_id": 24117248},
      "v_type": "person"
```

```
      }
    ]}]
  }
```

`selectVertex()`

Syntax

`selectVertex( filepath, vertexIdColumn, vertexTypeColumn, seperator, header)`

Description

Reads a data file that lists particular vertices of the graph and returns the corresponding vertex set. This function can only be used in a vertex set variable declaration statement as a seed set and the vertices in the data file **must already be in the graph**. The data file must be organized as a table with one or more columns. One column must be for vertex ID. Optionally, another column is for vertex type.

Return type

`SET<VERTEX>`

Parameters

Parameter	Description	Data type
`filePath`	The absolute file path of the input file to be read. A relative path is not supported.	`STRING`
	The vertex ID column position.	`$ num`.

`vertexIdColumn`	The index for column positions starts at 0. Therefore, to designate the first column as the ID column, set this parameter to `$0`.	If `header` is set to true, `$ "column_name"` is also acceptable.
		`$ num`.
`vertexTypeColumn`	The vertex type column position or a specific vertex type.	If `header` is set to true, `$ "column_name"` is also acceptable. Alternatively, a vertex type without double quotes.
`separator`	The column separator character.	`STRING`
`header`	Whether this file has a header.	`BOO

Example

selectVertex.csv

```
ID,type
Dan,person
Jenny,person
Amily,person
```

Example query

```
CREATE QUERY selectVertexEx(STRING filename) FOR GRAPH socialNet {
  S1 = {SelectVertex(filename, $"c1", $1, ",", true)};
  S2 = {SelectVertex(filename, $0, person, ",", true)};
  PRINT S1, S2; # Both sets of inputs product the same result
}
```

Results

```
RUN QUERY selectVertex("/home/tigergraph/mydata/selectVertex.csv")
{
    "S1": [
      {
        "attributes": {
          "age": 0,
          "gender": "female",
          "name": "Amily",
          "state": "ca"
        },
        "v_id": "Amily",
        "v_type": "person"
      },
      {
        "attributes": {
          "age": 1,
          "gender": "male",
          "name": "Dan",
          "state": "ny"
        },
        "v_id": "Dan",
        "v_type": "person"
      },
      {
        "attributes": {
          "age": 1,
          "gender": "female",
          "name": "Jenny",
          "state": "tx"
        },
        "v_id": "Jenny",
        "v_type": "person"
      }
    ],
    "S2": [
      {
        "attributes": {
          "age": 0,
          "gender": "female",
          "name": "Amily",
          "state": "ca"
        },
        "v_id": "Amily",
        "v_type": "person"
      },
      {
        "attributes": {
          "age": 1,
```

```
          "gender": "male",
          "name": "Dan",
          "state": "ny"
        },
        "v_id": "Dan",
        "v_type": "person"
      },
      {
        "attributes": {
          "age": 1,
          "gender": "female",
          "name": "Jenny",
          "state": "tx"
        },
        "v_id": "Jenny",
        "v_type": "person"
      }
    ]
  }
]
```

`to_vertex()`

Running `to_vertex() and to_vertex_set()` requires real-time conversion of an external ID to a GSQL internal ID, which is a relatively slow process. Therefore,

- If the user can always know the id before running the query, define the query with `VERTEX` or `SET<VERTEX>` parameters instead of `STRING` or `SET<STRING>` parameters, and avoid calling `to_vertex()` or `to_vertex_set()`.
- Calling `to_vertex_set()` one time is much faster than calling `to_vertex()` multiple times. Use `to_vertex_set()` instead of `to_vertex()` as much as possible.

Syntax

`to_vertex( id, vertex_type )`

Description

Returns a vertex from a string ID and vertex type. If a vertex with the provided ID and type does not exist, the function will throw a run-time error.

Return type

`VERTEX`

Parameters

Parameter	Description	Data type
`id`	The ID of a vertex	`STRING`
`vertex_type`	The type of the vertex	`STRING`

Example

Example query using to_vertex()

```
CREATE QUERY to_vertex_test (STRING uid, STRING vtype) FOR GRAPH social {
  VERTEX v;

  v = to_vertex (uid, vtype);           # to_vertex assigned to a vertex
  PRINT v;                              # vertex variable -> only vertex
}
```

Query result

```
GSQL > RUN QUERY to_vertex_test("Dan", "person")
{
  "error": false,
  "message": "",
  "version": {
    "schema": 1,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"v": "Dan"}]
}
```

`to_vertex_set()`

Syntax

`to_vertex_set( id_set, vertex_type)`

Description

Returns a vertex set from a set or bag of string IDs and a vertex type. If there are invalid IDs in the set, those IDs will be skipped and the response will contain a warning message. If the vertex type does not exist, the function will throw a run-time error.

Return type

`SET<VERTEX>`

Parameters

Parameter	Description	Data type
`id_set`	A set of vertex IDs	`SET<STRING>`, `BAG<STRING>`
`vertex_type`	The type of the vertices	`STRING`

Example

```
CREATE QUERY to_vertex_set_test (SET<STRING> uids, STRING vtype) FOR GRAP

  S2 = to_vertex_set (uids, vtype); # to_vertex_set assigned to a vertex
  PRINT S2;                          # vertex set variable-> full deta
}
```

```
GSQL > run query to_vertex_set_test(["Dan", "Amily", "Jeff"], "person")
{
  "error": false,
  "message": "Runtime Warning: 1 ids are invalid person vertex ids.",
  "version": {
    "schema": 1,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"S2": [
    {
      "v_id": "Amily",
      "attributes": {
        "gender": "female",
        "name": "Amily",
        "state": "ca",
        "age": 0
      },
      "v_type": "person"
    },
    {
      "v_id": "Dan",
      "attributes": {
        "gender": "male",
        "name": "Dan",
        "state": "ny",
        "age": 1
      },
      "v_type": "person"
    }
  ]}]
}
```

Loading...

Query User-Defined Functions

In GSQL, users can supplement the language by defining their own query user-defined functions (query UDF). Query UDFs can be called in queries and subqueries to perform a set of defined actions and return a value like the built-in functions.

This page introduces the process to define a query UDF. Once defined, the new functions will be added into GSQL automatically next time GSQL is executed.

Define a query UDF

Below are the steps to add a Query UDF to GSQL:

Step 1: Download current query UDF file

Use the `GET ExprFunctions` command in GSQL to download the current UDF file to any location on your machine. The file and the directores will be created if they do not exist, and the file must end with the file extention `.hpp`:

```
GSQL > GET ExprFunctions TO "/example/path/to/ExprFunctions.hpp"
```

If your query UDF requires a user-defined struct or helper function, also use the `GET ExprUtil` command to download the current `ExprUtil` file:

```
GSQL > GET ExprUtil TO "/example/path/ExprUtil.hpp"
```

Step 2: Define C++ function

Define the C++ function inside the `UDIMPL` namespace inside of the UDF file you just downloaded in Step 1. The definition of the function should include the keyword `inline`. Only `bool`, `int`, `float`, `double`, and `string` (NOT `std::string`) are allowed as the return value type and the function argument type. However, any C++ type is allowed inside a function body.

If the function requires a user-defined struct or helper function, define it in the `ExprUtil` file you downloaded in Step 1.

Below is an example of a query UDF definition:

New code in ExprFunctions.hpp

```
#include <algorithm>  // for std::reverse
inline bool greater_than_three (double x) {
  return x > 3;
}
inline string reverse(string str){
  std::reverse(str.begin(), str.end());
  return str;
}
```

If any code in `ExprFunctions.hpp` or `ExprUtil.hpp` causes a compilation error, GSQL cannot install any GSQL query, even if the GSQL query doesn't call any query UDF. Therefore, please test each new query UDF after adding it. One way of testing a function is to create a new file `test.cpp` and compile it:
`> g++ test.cpp`
`> ./a.out`
You might need to remove the include header `#include <gle/engine/cpplib/headers.hpp>` in `ExprFunctions.hpp` and `ExprUtil.hpp` in order to compile.

test.cpp

```
#include "ExprFunctions.hpp"
#include <iostream>
int main () {
  std::cout << to_string (123) << std::endl;    // to_string and s
  std::cout << str_to_int ("123") << std::endl;
  return 0;
}
```

Step 3: Upload files

After you have defined the function, use the `PUT` command to upload the files you modified.

```
GSQL > PUT ExprFunctions FROM "/path/to/udf_file.hpp"
PUT ExprFunctions successfully.
GSQL > PUT ExprUtil FROM "/path/to/utils_file.hpp"
PUT ExprUtil successfully.
```

The `PUT` command will automatically upload the files to all nodes in a cluster. Once the files are uploaded, you will be able to call the query UDF the next time GSQL is executed. This includes the next time you start the GSQL shell or executing GSQL scripts from a bash shell.

Example of a GSQL query that uses the UDF

```
CREATE QUERY udfExample() FOR GRAPH minimalNet {
  DOUBLE x;
  BOOL y;

  x = 3.5;
  PRINT greater_than_three(x);
  y = greater_than_three(2.5);
  PRINT y;

  PRINT reverse("abc");
}
```

Example

Suppose you are working in a distributed environment and want to add a function that that returns a random double between 0 and 1.

Start by downloading the current UDF file with the `GET` command:

```
GSQL > GET ExprFunctions TO "/home/tigergraph/documents/udf.hpp"
```

In the downloaded file, add the function definition for function `rng` and add the necessary include directives at the top:

udf.hpp

```
// Some code is left out for simplicity

// Copyright (c) 2015-2021, TigerGraph Inc.

#ifndef EXPRFUNCTIONS_HPP_
#define EXPRFUNCTIONS_HPP_

#include <stdlib.h>
#include <stdio.h>
#include <string>
#include <gle/engine/cpplib/headers.hpp>
#include <iostream>
#include <fstream>
#include <sstream>
#include <random> // include statement for rng()
#include <vector>
#include <map>

#include "ExprUtil.hpp"

namespace UDIMPL {
  typedef std::string string;

  // other UDFs ...

  inline double rng() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution < double > distribution(0.0, 1.0);

    return distribution(gen);
  }

  // other UDFs ...

}


#endif /* EXPRFUNCTIONS_HPP_ */
```

Lastly, use the `PUT` command to upload the file. This will uploaded the file to all nodes in a cluster:

```
GSQL > PUT ExprFunctions FROM "/home/tigergraph/documents/udf.hpp"
PUT ExprFunctions successfully.
```

The UDF has now been added to GSQL and you can start using the function in GSQL queries.

Declaration and Assignment Statements

In GSQL, different types of variables and objects follow different rules when it comes to variable declaration and assignment. This section discusses the different types of declaration and assignment statements and covers the following subset of the EBNF syntax:

EBNF

```
## Declarations ##
accumDeclStmt :=
          accumType localAccumName ["=" constant]
                    ["," localASccumName ["=" constant]]*
        | [STATIC] accumType globaAccumName ["=" constant]
                    ["," GlobalAccumName ["=" constant]]*
localAccumName := "@"accumName;
globalAccumName := "@@"accumName;

baseDeclStmt    := baseType name ["=" constant] ["," name ["=" constant]]
fileDeclStmt := FILE fileVar "(" filePath ")"
fileVar := name

localVarDeclStmt := baseType varName "=" expr

vSetVarDeclStmt := vertexSetName ["(" vertexType ")"] "=" (seedSet | simp

simpleSet := vertexSetName
           | "(" simpleSet ")"
           | simpleSet (UNION | INTERSECT | MINUS) simpleSet

seedSet := "{" [seed ["," seed ]*] "}"
seed := '_'
      | ANY
      | vertexSetName
      | globalAccumName
      | vertexType ".*"
      | paramName
      | "SelectVertex" selectVertParams

selectVertParams := "(" filePath "," columnId "," (columnId | name) ","
          stringLiteral "," (TRUE | FALSE) ")" ["." FILTER "(" condition

columnId := "$" (integer | stringLiteral)

## Assignment Statements ##
assignStmt := name "=" expr
            | name "." attrName "=" expr

attrAccumStmt := name "." attrName "+=" expr

lAccumAssignStmt := vertexAlias "." localAccumName ("+="| "=") expr

gAccumAssignStmt :=  globalAccumName ("+=" | "=") expr

loadAccumStmt := globalAccumName "=" "{" LOADACCUM loadAccumParams
                                  ["," LOADACCUM loadAccumParams]* "}"
```

```
loadAccumParams := "(" filePath "," columnId ["," columnId]* ","
           stringLiteral "," (TRUE | FALSE) ")" ["." FILTER "(" condition

## Function Call Statement ##
funcCallStmt := name ["<" type ["," type]* ">"] "(" [argList] ")"
              | globaAccumName ("." funcName "(" [argList] ")")+
              | "reset_collection_accum" "(" accumName ")"

argList := expr ["," expr]*
```

Variable scopes

Different types of variable declarations use different scoping rules. There are two types of scoping rules in a GSQL query:

- Block scoping
- Global scoping

Block scoping

In GSQL, curly brackets, as well as `IF .. THEN`, `ELSE`, `WHILE ... DO`, `FOREACH ... DO` statements create a block. A `SELECT` statement also creates a block. A block-scoped variable declared inside a block scope is only accessible inside that scope.

Additionally, variables declared in a lower scope can use the same name as a variable already declared in a higher scope. The lower-scope declaration will take precedence over the higher-scope declaration until the end of the lower scope.

The following types of variables use block scoping:

- Accumulators
- Base type variables
- Local base type variables
- File objects

- Vertex or edge aliases

Global scoping

A global-scoped variable is always accessible anywhere in the query once it has been declared regardless of where it is declared. One also cannot declare another variable with the same name as a global-scoped variable that has already been declared.

The following types of variables use global scoping:

- Vertex set variables

Declaration Statements

There are six types of variable declarations in a GSQL query:

- Accumulator
- Base type variable
- Local base type variable
- Vertex set
- File object
- Vertex or edge aliases

The first five types each have their own declaration statement syntax and are covered in this section. Aliases are declared implicitly in a `SELECT` statement.

Accumulators

Accumulator declaration is discussed in Accumulators.

Base type variables

In a GSQL query body, variables holding values of types `INT`, `UINT`, `FLOAT`, `DOUBLE`, `BOOL`, `STRING`, `DATETIME`, `VERTEX`, `EDGE`, `JSONOBJECT` and `JSONARRAY` are called base type variables. The scope of a base type variable is from the point of declaration until the end of the block where its declaration took place.

EBNF for base type variable declaration

```
baseVarDeclStmt    := baseType name ["=" expr]["," name ["=" expr]]*
```

A base type variable can be declared and accessed anywhere in the query. To declare a base type variable, specify the data type and the variable name. Optionally, you can initialize the variable by assigning it a value with the assignment operator (`=`) and the desired value on the right side. You can declare multiple variables of the same type in a single declaration statement.

```
CREATE QUERY baseTypeVariable() {
    STRING a;
    DOUBLE num1, num2 = 3.2;
    INT year = 2020, month = 12, day = 115;
    INT b = rand(5);
    PRINT a, b, num;
}
```

When a base type variable is assigned a new value in an `ACCUM` or `POST-ACCUM` clause, the change will not take place until exitng the clause. Therefore, if there are multiple assignment statements for the same base type variable in an `ACCUM` or `POST-ACCUM` clause, only the last one will take effect.

For example, in the following query, a base type variable is assigned a new value in the `ACCUM` clause, but the change will not take place until the clause ends. Therefore, the accumulator will not receive the value and will hold a value of 0 at the end of the query.

```
CREATE QUERY baseTypeVariable(vertex<person> m1) FOR GRAPH socialNet {
  MaxAccum<INT> @@maxDateGlob;
  DATETIME dt;

  allUser = {person.*};
  allUser = SELECT src
            FROM allUser:src - (liked:e) -> post
            ACCUM
            dt = e.actionTime,           # dt isn't updated yet
            @@maxDateGlob += datetime_to_epoch(dt);
  PRINT @@maxDateGlob, dt;  # @@maxDateGlob will be 0
}
```

Local base type variables

Base type variables declared in a DML-sub statement, such as in a statement inside a `ACCUM`, `POST-ACCUM`, or `UPDATE SET` clause, are called *local base type variables*.

Local base type variables are block-scoped and are accessible in the block where they are declared only. Within a local base type variable's scope, another local base type variable with the same name cannot be declared at the same level. However, a new local base type variable with the same name can be declared at a lower level (i.e., within a nested `SELECT` or `UPDATE` statement). The lower declaration takes precedence at the lower level.

In a `POST-ACCUM` clause, each local base type variable may only be used in source vertex statements or only in target vertex statements, not both.

EBNF for local base type variable declaration and initialization

```
localVarDeclStmt := baseType varName "=" expr
```

Local base type variables are not subject to the assignment restrictions of regular base type variables. Their values can be updated inside an `ACCUM` or `POST-ACCUM` clause and the change will take place immediately.

Example:

Base type variable declaration in DML statements

```
# An example showing a local base type variable succeeds
# where a base type variable fails
CREATE QUERY localVariable(vertex<person> m1) FOR GRAPH socialNet {
  MaxAccum<INT> @@maxDate, @@maxDateGlob;
  DATETIME dtGlob;

  allUser = {person.*};
  allUser = SELECT src
            FROM allUser:src - (liked:e) -> post
            ACCUM
            DATETIME dt = e.actionTime,      # Declare and assign local d
            dtGlob = e.actionTime,           # dtGlob isn't updated yet
            @@maxDate     += datetime_to_epoch(dt),
            @@maxDateGlob += datetime_to_epoch(dtGlob);
  PRINT @@maxDate, @@maxDateGlob, dtGlob;  # @@maxDateGlob will be 0
}
```

localVariable Query Results

```
GSQL > RUN QUERY localVariable("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "dtGlob": "2010-01-11 03:26:05",
    "@@maxDateGlob": 0,
    "@@maxDate": 1263618953
  }]
}
```

Vertex Set Variable Declaration and Assignment

Variables that contain a set of one or more vertices are called vertex set variables. Vertex set variables play a special role within GSQL queries. They are used for both the input and output of `SELECT` statements. Therefore, before the first `SELECT` statement in a query, a vertex set variable must be declared and initialized. This initial vertex set is called the *seed set*.

Vertex set variables are global-scoped. They are also the only type of variable that isn't explicitly typed during declaration. To declare a vertex set variable, assign an initial set of vertices to the variable name.

EBNF for Vertex Set Variable Declaration

```
vSetVarDeclStmt := vertexSetName ["(" vertexType ")"] "=" (seedSet | simp

simpleSet := vertexSetName
      | "(" simpleSet ")"
      | simpleSet (UNION | INTERSECT | MINUS) simpleSet

seedSet := "{" [seed ["," seed ]*] "}"
seed := '_'
      | ANY
      | vertexSetName
      | globalAccumName
      | vertexType ".*"
      | paramName
      | "SelectVertex" selectVertParams

selectVertParams := "(" filePath "," columnId "," (columnId | name) ","
     stringLiteral "," (TRUE | FALSE) ")" ["." FILTER "(" condition ")"]

columnId := "$" (integer | stringLiteral)
```

The query below lists all ways of assigning a vertex set variable an initial set of vertices (that is, forming a seed set).

- A vertex parameter, untyped or typed, enclosed in curly brackets
- A vertex set parameter, untyped or typed
- A global `SetAccum<VERTEX>` accumulator, untyped or typed
- All vertices of any type or of one type
- A list of vertex IDs in an external file
- Copy of another vertex set
- A combination of individual vertices, vertex set parameters, or base type variables, enclosed in curly brackets

- Union of vertex set variables

Seed Set Example

```
CREATE QUERY seedSetExample(VERTEX v1, VERTEX<person> v2, SET<VERTEX> v3,
  SetAccum<VERTEX> @@testSet;
  SetAccum<VERTEX<person>> @@testSet2;
  S1 = { v1 };    # untyped vertex parameter enclosed in curly brackets
  S2 = { v2 };    # typed vertex parameter enclosed in curly brackets
  S3 = v3;                      # untyped vertex set parameter
  S4 = v4;                      # typed vertex set parameter
  S5 = @@testSet;               # untyped global set accumulator
  S6 = @@testSet2;              # typed global set accumulator
  S7 = ANY;                     # All vertices
  S8 = person.*;                # All person vertices
  S9 = _;                       # Equivalent to ANY
  S10 = SelectVertex("absolute_path_to_input_file", $0, post, ",", false)
  S11 = S1;                     # copy of another vertex set
  S12 = {@@testSet, v2, v3};    # Individual vertex: v2
                                # Vertex set parameter: v3
                                # global accumulator: @@testSet
                                # Inside curly brackets cannot be put an
                                # seedset, e.g., S1
  S13 = S11 UNION S12;          # but we can use UNION to combine S1
}
```

When declaring a vertex set variable, a set of vertex types can be optionally specified to the vertex set variable. If the vertex set variable set type is not specified explicitly, the system determines the type implicitly by the vertex set value. The type can be `ANY`, `_` (equivalent to ANY), or any explicit vertex type(s). See the EBNF grammar rule `vertexEdgeType`.

> Declaration syntax difference: vertex set variable vs. base type variable
>
> In a vertex set variable declaration, the optional type specifier follows the variable name and should be surrounded by parentheses: **`vSetName(type)`**
> This is different than a base type variable declaration, where the type specifier is required and comes before the base variable name: **`type varName`**

Assignment

After a vertex set variable is declared, the vertex type of the vertex set variable is immutable. Every assignment (e.g. `SELECT` statement) to this vertex set variable must match the type. The following is an example in which we must declare the vertex set variable type.

Vertex set variable type

```
CREATE QUERY vertexSetVariableTypeExample(vertex<person> m1) FOR GRAPH so
  INT ite = 0;
  S (ANY) = {m1};   # ANY is necessary
  WHILE  ite < 5 DO
    S = SELECT t
        FROM S:s - (ANY:e) -> ANY:t;

    ite = ite + 1;
  END;
  PRINT S;
}
```

In the above example, the query returns the set of vertices after a 5-step traversal from the input `person` vertex. If we declare the vertex set variable `S` without explicitly giving a type, because the type of vertex parameter `m1` is `person`, the GSQL engine will implicitly assign S to be `person` type. However, if `S` is assigned to `person` type, the `SELECT` statement inside the `WHILE` loop causes a type-checking error, because the `SELECT` block will generate all connected vertices, including non-person vertices. Therefore, `S` must be declared as an ANY-type vertex set variable.

`FILE` Object Declaration

A `FILE` object is a sequential text storage object, associated with a text file on the local machine.

EBNF for FILE object declaration

```
fileDeclStmt := FILE fileVar "(" filePath ")"
fileVar := name
```

When a `FILE` object is declared, associated with a particular text file, any existing content in the text file will be erased. During the execution of the query, content written to or printed to the `FILE` object will be appended to the `FILE` object. When the query where the `FILE` object is declared finishes running, the content of the `FILE` object is saved to the text file.

Example:

File object query example

```
CREATE QUERY getUSWorkerInterests (STRING fileLocation) FOR GRAPH workNet
    // Declare FILE object f1
    FILE f1 (fileLocation);
    // Initialize a seed set of all person vertices
    P = {person.*};

    PRINT "header" TO_CSV f1;

    // Select workers located in the US and print their interests onto
    // the FILE object
    USWorkers = SELECT v FROM P:v
              WHERE v.locationId == "us"
              ACCUM f1.println(v.id, v.interestList);
    PRINT "footer" TO_CSV f1;
}
INSTALL QUERY getUSWorkers
RUN QUERY getUSWorkerInterests("/home/tigergraph/fileEx.txt")
```

Assignment and Accumulate Statements

Assignment statements are used to set or update the value of a variable after it has been declared. This applies to base type variables, vertex set variables, and accumulators. Accumulators also have the special += accumulate statement, which was discussed in the Accumulator section. Assignment statements can use expressions to define the new value of the variable.

EBNF for Assignment Statements

```
## Assignment Statement ##
assignStmt := name "=" expr         # baseType variable, vertex set varia
            | name "." name "=" expr      # attribute of a vertex or edge

attrAccumStmt := name "." attrName "+=" expr

lAccumAssignStmt := vertexAlias "." localAccumName ("+="| "=") expr

gAccumAssignStmt :=  globalAccumName ("+=" | "=") expr

loadAccumStmt := globalAccumName "=" "{" "LOADACCUM" loadAccumParam
                                  ["," "LOADACCUM" loadAccumParams]* "}"
```

Vertex and edge (non-accumulator) attributes can use the += operator in an ACCUM or POST-ACCUM clause to perform parallel accumulation.

```
attrAccumStmt := name "."attrName "+=" expr
```

Restrictions on Assignment Statements

In general, assignment statements can take place anywhere after the variable has been declared. However, there are some restrictions. These restrictions apply to "inner level" statements which are within the body of a higher-level statement:

- The `ACCUM` or `POST-ACCUM` clause of a `SELECT` statement
- The `SET` clause of an `UPDATE` statement
- The body of a `FOREACH` statement

- Global accumulator assignment is not permitted within the body of `SELECT` or `UPDATE` statements
- Base type variable assignment is permitted in `ACCUM` or `POST-ACCUM` clauses, but the change in value will not take place until exiting the clause. Therefore, if there are multiple assignment statements for the same variable, only the final one will take effect.
- Vertex attribute assignment is not permitted in an `ACCUM` clause. However, edge attribute assignment is permitted. This is because the `ACCUM` clause iterates over an edge set.

- There are additional restrictions within `FOREACH` loops for the loop variable. See the Data Modification section.

`LOADACCUM` Statement

```
loadAccumStmt := globalAccumName "=" "{" LOADACCUM loadAccumParams
                                   ["," LOADACCUM loadAccumParams]* "}"

loadAccumParams := "(" filePath "," columnId ["," [columnId]* ","
          stringLiteral "," (TRUE | FALSE) ")" ["."FILTER "(" condition "
columnId := "$"(integer | stringLiteral)
```

`LOADACCUM()` can initialize a global accumulator by loading data from a file. `LOADACCUM()` has 3+n parameters explained in the table below, where n is the number of fields in the accumulator.

Any accumulator using generic `VERTEX` as an element type cannot be initialized by `LOADACCUM()`.

Parameters:

Name	Type	Description
`filePath`	String	The absolute file path of the input file to be read. A relative path is not supported.
`columnId`	String or number	The column position(s) or column name(s) of the data file that supply data values to each field of the accumulator.
`separator`	Single-character string	The separator of columns.
`header`	Boolean	Whether this file has a header.

One assignment statement can have multiple `LOADACCUM()` function calls. However, every `LOADACCUM()` referring to the same file in the same assignment statement must use the same separator and header parameter values.

Example:

loadAccumInput.csv

```
person1,1,"test1",3
person5,2,"test2",4
person6,3,"test3",5
```

LoadAccum example

```
CREATE QUERY loadAccumEx(STRING filename) FOR GRAPH socialNet {
  TYPEDEF TUPLE<STRING aaa, VERTEX<post> ddd> yourTuple;
  MapAccum<VERTEX<person>, MapAccum<INT, yourTuple>> @@testMap;
  GroupByAccum<STRING a, STRING b, MapAccum<STRING, STRING> strList> @@te

  @@testMap = { LOADACCUM (filename, $0, $1, $2, $3, ",", false)};
  @@testGroupBy = { LOADACCUM ( filename, $1, $2, $3, $3, ",", true) };

  PRINT @@testMap, @@testGroupBy;
}
```

Results of Query loadAccumEx

```
GSQL > RUN QUERY loadAccumEx("/file_directory/loadAccumInput.csv")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "@@testGroupBy": [
      {
        "a": "3",
        "b": "\"test3\"",
        "strList": {"5": "5"}
      },
      {
        "a": "2",
        "b": "\"test2\"",
        "strList": {"4": "4"}
      }
    ],
    "@@testMap": {
      "person1": {"1": {
        "aaa": "\"test1\"",
        "ddd": "3"
      }},
      "person6": {"3": {
        "aaa": "\"test3\"",
        "ddd": "5"
      }},
      "person5": {"2": {
        "aaa": "\"test2\"",
        "ddd": "4"
      }}
    }
  }]
}
```

Function Call Statements

```
funcCallStmt := name ["<" type ["," type]* ">"] "(" [argList] ")"
              | globalAccumName ("." funcName "(" [argList] ")")+
              | "reset_collection_accum" "(" accumName ")"

argList := expr ["," expr]*
```

Typically, a function call returns a value and so is part of an expression. In some cases, however, the function does not return a value (i.e., returns `VOID`) or the return value can be ignored, so the function call can be used as an entire statement. This is a Function Call Statement.

Examples of Function Call statements

```
ListAccum<STRING> @@listAcc;
BagAccum<INT> @@bagAcc;
...
# examples of function call statements
@@listAcc.clear();
@@bagAcc.removeAll(0);
```

Clear Collection Accumulators

Collection accumulators (e.g., `ListAccum`, `SetAccum`, `MapAccum`) grow in size as data is added. Particularly for vertex-attached accumulators, if the number of vertices is large, their memory consumption can be significant. It can improve system performance to clear or reset collection accumulators during a query as soon as their data is no longer needed. Running the `reset_collection_accum(accumName)` function resets the collection(s) to be zero-length (empty). If the argument is a vertex-attached accumulator, then the entire set of accumulators is reset.

```
"reset_collection_accum" "(" accumName ")"
```

`reset_collection_accum` only works in DISTRIBUTED mode queries. If the query is not in distributed mode, the reset does not take place.

```
CREATE DISTRIBUTED QUERY reset_accum()
FOR GRAPH workNet SYNTAX v2 {
  ListAccum<STRING> @stuff;
  ListAccum<STRING> @@allStuff;

  Comp = SELECT c
    FROM    person:p -(worksFor:w)- company:c
    ACCUM   c.@stuff += p.id,
            @@allStuff += p.id,
            c.@stuff += p.locationId,
            @@allStuff += p.locationId,
            FOREACH interest IN p.interestList DO
              c.@stuff += interest,
              @@allStuff += interest
            END
  ;
  // display accum size: should be full
  PRINT Comp[Comp.@stuff.size()] AS stuffCount;
  PRINT @@allStuff.size() AS allStuffCount;

  reset_collection_accum(@stuff);
  reset_collection_accum(@@allStuff);
  // display accum size: should be empty
  PRINT Comp[Comp.@stuff.size()] AS stuffClear;
  PRINT @@allStuff.size() AS allStuffClear;
}
```

```
[{
    "stuffCount": [
      {"attributes": {"Comp.@stuff.size()": 23},
        "v_id": "company2",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 7},
        "v_id": "company4",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 12},
        "v_id": "company3",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 21},
        "v_id": "company1",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 4},
        "v_id": "company5",
        "v_type": "company"
      }]
  },
  {
    "allStuffCount": 67
  },
  {
    "stuffClear": [
      {"attributes": {"Comp.@stuff.size()": 0},
        "v_id": "company2",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 0},
        "v_id": "company4",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 0},
        "v_id": "company3",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 0},
        "v_id": "company1",
        "v_type": "company"
      },
      {"attributes": {"Comp.@stuff.size()": 0},
        "v_id": "company5",
        "v_type": "company"
      }]
```

```
  },
  {
    "allStuffClear": 0
  }]
```

SELECT Statement

GSQL Syntax Versions

NOTE: Starting with TigerGraph 2.6, there are now two versions of the GSQL Query Language. Both are available in the product. The original version is called GSQL Syntax V1. GSQL Syntax V2 was introduced to provide more flexible and powerful Pattern Matching. A query indicates which grammar to use in the SYNTAX clause of the CREATE QUERY header. V1 is the default.

As new features are being added to GSQL, some are available only V2. In the future, we plan to transition so that Syntax V2 will be the default.

The key differences between the two grammars:

1. SQL-like Tabular SELECT is only in Syntax V2 (since TigerGraph 3.1)
2. Multi-hop Pattern Matching is only in Syntax V2 (since TigerGraph 2.6)
   - In V2, each SELECT statement can traverse a multi-hop path.
     - The traversal direction is under the control of the query writer, with arrowheads on each edge set to show the direction.
     - There is no arrowhead outside the parentheses:
       `Start:s -( (ForwardEdge> | <BackwardEdge):e )- Target:t`
     - Users can write paths that explicitly show multiple hops, and they can use a Kleene star (*) to indicate repetition.
       `Start:s -(Edge1>:e1)- Middle:m -(<Edge2:e2)- Target:t` or
       `Start:s -(Edge*1..3)- Target:t`
   - In V1 (default), each SELECT statement can traverse one hop (from a set of vertices to their adjacent vertices).
     - To write a multi-hop query, you write a sequence of SELECT statements.
     - The traversal action is from left to right, and the notation uses "ASCII art" to depict a connection, either with a rightward facing arrowhead or no arrowhead:
       `Start:s -( Edges:e )-> Target:t` or
       `Start:s -( Edges:e )- Target:t`

This section discusses the SELECT statement in-depth and covers the following EBNF syntax. Note that there are now two forms for a SELECT statement, the classic GSQL form `gsqlSelectBlock`, and the new SQL-like form `sqlSelectBlock`.

Since GSQL has always been similar to SQL in many ways, there are only a few differences to know about.

EBNF for Select Statement

```
selectStmt  := gsqlSelectBlock
             | sqlSelectBlock

gsqlSelectBlock := gsqlSelectClause
               fromClause
               [sampleClause]
               [whereClause]
               [accumClause]
               [postAccumClause]*
               [havingClause]
               [orderClause]
               [limitClause]

sqlSelectBlock := sqlSelectClause
               fromClause
               [whereClause]
               [groupByClause]
               [havingClause]
               [orderClause]
               [limitClause]

gsqlSelectClause := ertexSetName "=" SELECT vertexAlias
sqlSelectClause := SELECT [DISTINCT] columnExpr ("," columnExpr)*
                   INTO tableName
columnExpr := expr [AS columnName]
            | aggregator "("[DISTINCT] expr ")" [AS columnName]
columnName := name
tableName := name

fromClause := FROM (step | stepV2 | pathPattern ["," pathPattern]*)

step   :=  stepSourceSet ["-" "(" stepEdgeSet ")" ("-"|"->") stepVertexSe
stepV2 :=  stepVertexSet ["-" "(" stepEdgeSet ")" "-" stepVertexSet]

stepSourceSet := vertexSetName [":" vertexAlias]
stepEdgeSet := [stepEdgeTypes] [":" edgeAlias]
stepVertexSet := [stepVertexTypes] [":" vertexAlias]
alias := (vertexAlias | edgeAlias)
vertexAlias := name
edgeAlias := name

stepEdgeTypes := atomicEdgeType | "(" edgeSetType ["|" edgeSetType]* ")"
atomicEdgeType := "_" | ANY | edgeSetType
edgeSetType := edgeType | paramName | globalAccumName

stepVertexTypes := atomicVertexType | "(" vertexSetType ["|" vertexSetTyp
atomicVertexType := "_" | ANY | vertexSetType
vertexSetType := vertexType | paramName | globalAccumName
```

```
#----------# Pattern Matching #----------#
pathPattern := stepVertexSet ["-" "(" pathEdgePattern ")" "-" stepVertexS

pathEdgePattern := atomicEdgePattern
                 | "(" pathEdgePattern ")"
                 | pathEdgePattern "." pathEdgePattern
                 | disjPattern
                 | starPattern

atomicEdgePattern  := atomicEdgeType
                    | atomicEdgeType ">"
                    | "<" atomicEdgeType

disjPattern := atomicEdgePattern ("|" atomicEdgePattern)*

starPattern := ([atomicEdgePattern] | "(" disjPattern ")") "*" [starBound

starBounds := CONST_INT ".." CONST_INT
            | CONST_INT ".."
            | ".." CONST_INT
            | CONST_INT
#----------------------------------------#
sampleClause := SAMPLE ( expr | expr "%" ) EDGE WHEN condition
              | SAMPLE expr TARGET WHEN condition
              | SAMPLE expr "%" TARGET PINNED WHEN condition

whereClause := WHERE condition

accumClause := [perClauseV2] ACCUM dmlSubStmtList

perClauseV2 := PER "(" alias ["," alias] ")"

postAccumClause := "POST-ACCUM" dmlSubStmtList

dmlSubStmtList := dmlSubStmt ["," dmlSubStmt]*

dmlSubStmt := assignStmt           // Assignment
            | funcCallStmt         // Function Call
            | gAccumAccumStmt      // Assignment
            | lAccumAccumStmt      // Assignment
            | attrAccumStmt        // Assignment
            | vAccumFuncCall       // Function Call
            | localVarDeclStmt     // Declaration
            | dmlSubCaseStmt       // Control Flow
            | dmlSubIfStmt         // Control Flow
            | dmlSubWhileStmt      // Control Flow
            | dmlSubForEachStmt    // Control Flow
```

GSQL SELECT Statement

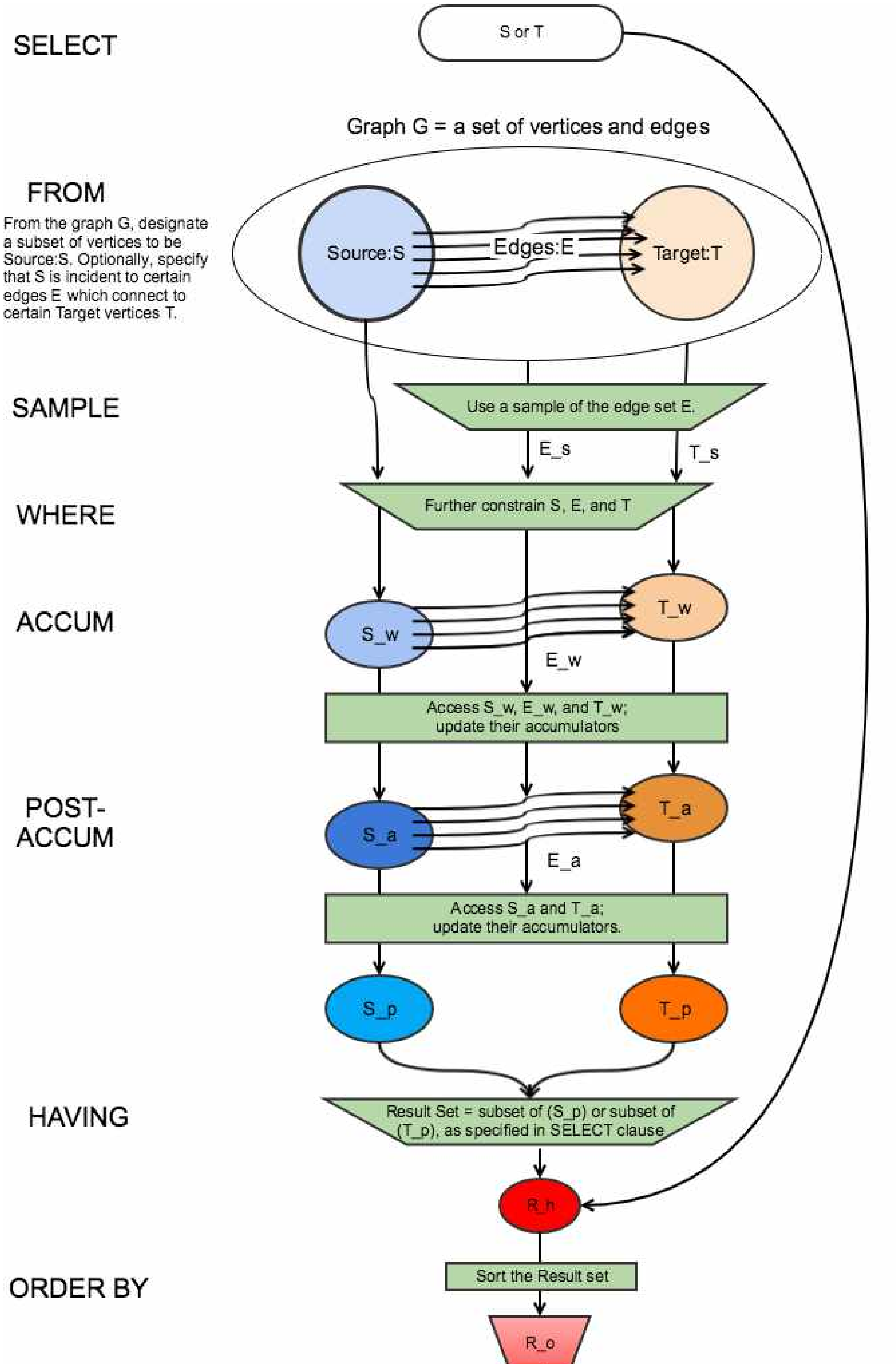
EBNF for GSQL Select Statement

```
gsqlSelectBlock := gsqlSelectClause
                   fromClause
                   [sampleClause]
                   [whereClause]
                   [accumClause]
                   [postAccumClause]*
                   [havingClause]
                   [orderClause]
                   [limitClause]

gsqlSelectClause := vertexSetName "=" SELECT vertexAlias
havingClause := HAVING condition

orderClause := ORDER BY expr [ASC | DESC] ["," expr [ASC | DESC]]*

limitClause := LIMIT ( expr | expr "," expr | expr OFFSET expr )
```

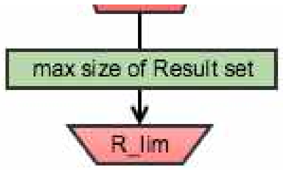
is to become the output. The classic SELECT clause may contain only one item: a vertex alias defined in the FROM clause. As of v3.1, the vertex alias may be from anywhere in a multi-hop pattern, not only an endpoint. GSQL now also supports SQL-like SELECT statements with tabular output.

The fromClause defines a path pattern to traverse in the graph, and each vertex in the path pattern can be given a vertexAlias name. Thus, the SELECT clause picks the set of vertices at one of these points in the pattern -- the source vertices, the target vertices, or those from an interior point in a multi-hop path -- to be the output vertices.

The SELECT block has many optional clauses, which fit together in a logical flow. Overall, the SELECT block starts from a source set of vertices and returns a result set that is either a subset of the source vertices or a subset of their neighboring vertices. Along the way, computations can be performed on the selected vertices and edges. The figure below graphically depicts the overall SELECT data flow. While the ACCUM and POST-ACCUM clauses do not directly affect which vertices are included in the result set, they affect the data (accumulators) which are attached to those vertices.


SELECT
S or T
Graph G = a set of vertices and edges
FROM
From the graph G, designate a subset of vertices to be Source:S. Optionally, specify that S is incident to certain edges E which connect to certain Target vertices T.
Source:S
Edges:E
Target:T
SAMPLE
Use a sample of the edge set E.
E_s
T_s
WHERE
Further constrain S, E, and T
ACCUM
S_w
T_w
E_w
Access S_w, E_w, and T_w; update their accumulators
POST-ACCUM
S_a
T_a
E_a
Access S_a and T_a; update their accumulators.
S_p
T_p
HAVING
Result Set = subset of (S_p) or subset of (T_p), as specified in SELECT clause
R_h
ORDER BY
Sort the Result set
R_o

LIMIT



Basic data flow for a classic 1-hop GSQL SELECT statement

FROM Clause: Vertex and Edge Sets

In classic (Syntax v1) GSQL, the FROM clause described one step or hop pattern. In Pattern Matching (Syntax v2), the pattern can be multiple hops long. Path patterns also have many other options for finer control and greater flexibility.

FROM clause

```
fromClause := FROM (step | stepV2 | pathPattern ["," pathPattern]*)
```

A hop or step consists of going from a starting set of vertices, crossing over a set of their edges, to an ending set of vertices. We typically use the names Source and Target for the starting and ending vertex sets: `Source -(Edges)-> Target`

The step pattern defines constraints for the Source set, the Edge set, and the Target set. The result of the FROM clause can be interpreted as a 3-column virtual table called the match table. Each row is a 3-element tuple: (source vertex, connected edge, target vertex).

Source Vertex Set (SYNTAX v1)

Notice that the edge set and target set are optional: a step can be just source vertices (stepSourceSet).

EBNF for source-only pattern (SYNTAX v1)

```
step   :=  stepSourceSet ["-" "(" stepEdgeSet ")" ("-"|"->") stepVertexSe
```

ⓘ Rules for Source Vertex Set in Syntax V1:

1. The source set may only be a vertexSetName. If this the first SELECT statement in the query, then the vertexSetName is generally created as a seedsSet:
2. The vertexSetName is optionally followed by an alias, which used in subsequent clauses to refer to the source set:

```
stepSourceSet := vertexSetName [":" vertexAlias]
vertexAlias := name
```

For example:

```
resultSet = SELECT s FROM Source:s;
```

This statement can be interpreted as "*Select all vertices s, from the vertex set Source* ." The result is a vertex set. Below is a simple example of a vertex selection.

Vertex SELECT example

```
# displays all 'post'-type vertices
CREATE QUERY printAllPosts() FOR GRAPH socialNet
{
  start =   {post.*};              # initialized with all vertices of typ
  results = SELECT s FROM start:s; # select these vertices
  PRINT results;
}
```

Results of Query printAllPosts

```
GSQL > RUN QUERY printAllPosts()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"results": [
    {
      "v_id": "0",
      "attributes": {
        "postTime": "2010-01-12 11:22:05",
        "subject": "Graphs"
      },
      "v_type": "post"
    },
    {
      "v_id": "10",
      "attributes": {
        "postTime": "2011-02-04 03:02:31",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "2",
      "attributes": {
        "postTime": "2011-02-03 01:02:42",
        "subject": "query languages"
      },
      "v_type": "post"
    },
    {
      "v_id": "4",
      "attributes": {
        "postTime": "2011-02-07 05:02:51",
        "subject": "coffee"
      },
      "v_type": "post"
    },
    {
      "v_id": "9",
      "attributes": {
        "postTime": "2011-02-05 23:12:42",
        "subject": "cats"
      },
```

```
        "v_type": "post"
      },
      {
        "v_id": "3",
        "attributes": {
          "postTime": "2011-02-05 01:02:44",
          "subject": "cats"
        },
        "v_type": "post"
      },
      {
        "v_id": "5",
        "attributes": {
          "postTime": "2011-02-06 01:02:02",
          "subject": "tigergraph"
        },
        "v_type": "post"
      },
      {
```


1-Hop Step (SYNTAX v1)

```
step   :=  stepSourceSet ["-" "(" stepEdgeSet ")" ("-"|"->") stepVertexSe
```


```
          "postTime": "2011-02-04 17:02:41",
          "subject": "Graphs"
        },
        "v_type": "post"
      },
      {
```


```
stepSourceSet := vertexSetName [":" vertexAlias]
stepEdgeSet := [setEdgeTypes] [":" edgeAlias]
stepVertexSet := [setVertexTypes] [":" vertexAlias]
```


```
        },
        "v_type": "post"
      },
```


```
Person:s -( (Bought|Rented):e )- (Product|Service):t
```


```
        "attributes": {
          "postTime": "2011-02-03 01:02:21",
          "subject": "cats"
        },
        "v_type": "post"
      },
      {
        "v_id": "8",
        "attributes": {
          "postTime": "2011-02-03 17:05:52",
          "subject": "cats"
        },
```

```
      "v_type": "post"
    },
    {
      "v_id": "6",
      "attributes": {
        "postTime": "2011-02-05 02:02:05",
        "subject": "tigergraph"
      },
```

Edge Set and Target Vertex Set Options

```
      "v_type": "post"
    }
  ]}]
}
```

1. The edge set (`stepEdgeSet`) and target vertex set (`stepVertexSet`) obey very similar rules. Each may be a set specifier (`stepEdgeTypes` or `stepVertexTypes`) followed optionally by an alias.
2. The set specifier can be any of the following:
   - `"_"` or `"ANY"` or <blank>, which means any edge/vertex.
   - a named type (`edgeType` or `vertexType`)
   - a global SetAcum accumulator containing a set of edges or vertices (`"@@"accumName`)
   - a string or string set parameter which names one or more edge or vertex types (`paramName`) This parameterized type is one aspect of [Dynamic Querying](#).
   - A list of types, string parameters, or global accumulators, e.g., `"(" edgeSetType ["|" edgeSetType]* ")"`
   - The list of types must contain the same type of specifiers.

```
stepEdgeSet := [setEdgeTypes] [":" edgeAlias]
stepVertexSet := [setVertexTypes] [":" vertexAlias]
alias := (vertexAlias | edgeAlias)
vertexAlias := name
edgeAlias := name

stepEdgeTypes := atomicEdgeType
               | "(" edgeSetType ["|" edgeSetType]* ")"
atomicEdgeType := "_" | ANY | edgeSetType
edgeSetType := edgeType | paramName | globalAccumName

stepVertexTypes := atomicVertexType
                 | "(" vertexSetType ["|" vertexSetType]* "
atomicVertexType := "_" | ANY | vertexSetType
vertexSetType := vertexType | paramName | globalAccumName
```

notation	accepted vertex/edge types
(empty)	any type
_	any type
ANY	any type
vertex/edge type name	given type
string parameter holding a vertex/edge type name	given type
global SetAccum containing vertices or edges	given set
(name | name ...)	UNION of two or more collections

Parentheses are always needed around the edge type in a FROM clause:
FROM Source:s -(eType:e)- Target:t

Parentheses are also needed if a vertexEdgeType is the union of more than one or more individual edge types or vertex types:
FROM Source:s -((eType1 | eType2):e) - Target:t

Note the double set of parentheses for the edge specifier. If these is an edge alias, these parentheses are needed. If however there is no edge alias, it is legal to have just a single set of parentheses:
FROM Source:s -(eType1 | eType2) - Target:t

Either the source vertex set (s) or target vertex set (t) can be used as the SELECT argument, which determines the result of the SELECT statement. Note the small difference in the two SELECT statements below.

Selecting source or target vertices from edge-induced selection

```
resultSet1 = SELECT s FROM source:s-(eType:e)-tType:t;   //select from th
resultSet2 = SELECT t FROM source:s-(eType:e)-tType:t;   //select from th
```

resultSet1 is based on the source end of the edges. resultSet2 is based on the target end of the selected edges. However, resultSet1 is NOT identical to the Source vertex set. It is only those members of Source which connect to an eType edge and then to a tType vertex. Other clauses (presented later in this "SELECT Statement" section, can do additional filtering of the Source set.

> ⚠ We strongly suggest that an alias should be declared with every vertex and edge in the FROM clause, as there are several functions and features which are only available to vertex and edge aliases.

If is legal to declare an alias without explicitly stating an edge/target type. See the examples below.

Target vertex type inference

```
resultSet3 = SELECT v FROM Source:v-(eType:e)->(V1|V2):t;
resultSet4 = SELECT v FROM Source:v-(eType:e)->:t;
resultSet5 = SELECT v FROM Source:v-(eType:e)->ANY:t;
resultSet6 = SELECT v FROM Source:v-(eType:e)->_:t;
```

Edge type inference

```
resultSet7 = SELECT v FROM Source:v-((E1|E2|E3):e)->tType:t;
resultSet8 = SELECT v FROM Source:v-(:e)->tType:t;
resultSet9 = SELECT v FROM Source:v-(_:e)->tType:t;
resultSet10 = SELECT v FROM Source:v-(ANY:e)->tType:t;
```

The following are a set of queries that demonstrate edge-induced SELECT blocks. The allPostsLiked and allPostsMade queries show how the target vertex type can be omitted. The allPostsLikedOrMade query uses the "|" operator to select multiple types of edges.

Edge induced SELECT example

```
# uses various SELECT statements (some of which are equivalent) to print
# either the posts made by the given user, the posts liked by the given
# user, or the posts made or liked by the given user.
CREATE QUERY printAllPosts2(vertex<person> seed) FOR GRAPH socialNet
{
    start = {seed}; # initialize starting set of vertices


    # --- statements produce equivalent results
    # select all 'post' vertices which can be reached from 'start' in one
    #   using an edge of type 'liked'
    allPostsLiked = SELECT targetVertex FROM start -(liked:e)-> post:targ

    # select all vertices of any type which can be reached from 'start' i
    #   using an edge of type 'liked'
    allPostsLiked = SELECT targetVertex FROM start -(liked:e)-> :targetVe
    # ----


    # --- statements produce equivalent results
    # start with the vertex set from above, and traverse all edges of typ
    #   (locally those edges are just given a name 'e' in case they need
    #   and return all vertices of type 'post' which can be reached withi
    allPostsMade = SELECT targetVertex FROM start -(posted:e)-> post:targ

    # start with the vertex set from above, and traverse all edges of typ
    #   (locally those edges are just given a name 'e' in case they need
    #   and return all vertices of any type which can be reached within o
    allPostsMade = SELECT targetVertex FROM start -(posted:e)-> :targetVe
    # ----


    # --- statements produce equivalent results
    # select all vertices of type 'post' which can be reached from 'start
    #   using an edge of any type
    # not equivalent to any statement. because it doesn't restrict the ed
    #   this will include any vertex connected by 'liked' or 'posted' edg
    allPostsLikedOrMade = SELECT t FROM start -(:e)-> t;

    # select all vertices of type 'post' which can be reached from 'start
    #   using an edge of type either 'posted' or 'liked'
    allPostsLikedOrMade = SELECT t FROM start -((posted|liked):e)-> post:

    # select all vertices of any type which can be reached from 'start' i
    #   using an edge of type either 'posted' or 'liked/
    allPostsLikedOrMade = SELECT t FROM start -((posted|liked):e)-> :t;

    #option for simplified parentheses in edge pattern:
```

```
    allPostsLikedOrMade = SELECT t FROM start - (posted|liked)-> :t
    # ----

    PRINT allPostsLiked;
    PRINT allPostsMade;
    PRINT allPostsLikedOrMade;
}
```

Results of Query printAllPosts2

```
GSQL > RUN QUERY printAllPosts2("person2")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"allPostsLiked": [
      {
        "v_id": "0",
        "attributes": {
          "postTime": "2010-01-12 11:22:05",
          "subject": "Graphs"
        },
        "v_type": "post"
      },
      {
        "v_id": "3",
        "attributes": {
          "postTime": "2011-02-05 01:02:44",
          "subject": "cats"
        },
        "v_type": "post"
      }
    ]},
    {"allPostsMade": [{
      "v_id": "1",
      "attributes": {
        "postTime": "2011-03-03 23:02:00",
        "subject": "tigergraph"
      },
      "v_type": "post"
    }]},
    {"allPostsLikedOrMade": [
      {
        "v_id": "0",
        "attributes": {
          "postTime": "2010-01-12 11:22:05",
          "subject": "Graphs"
        },
        "v_type": "post"
      },
      {
        "v_id": "3",
        "attributes": {
```

```
          "postTime": "2011-02-05 01:02:44",
          "subject": "cats"
        },
        "v_type": "post"
      },
      {
        "v_id": "1",
        "attributes": {
          "postTime": "2011-03-03 23:02:00",
          "subject": "tigergraph"
        },
        "v_type": "post"
      }
    ]}
  ]
}
GSQL > RUN QUERY printAllPosts2("person6")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"allPostsLiked": [{
      "v_id": "8",
      "attributes": {
        "postTime": "2011-02-03 17:05:52",
        "subject": "cats"
      },
      "v_type": "post"
    }]},
    {"allPostsMade": [
      {
```

Edge induced SELECT example

```
        "attributes": {
          "postTime": "2011-02-04 03:02:31",
          "subject": "cats"
        },
        "v_type": "post"
      },
      {
        "v_id": "5",
        "attributes": {
          "postTime": "2011-02-06 01:02:02",
          "subject": "tigergraph"
```

```
# uses a SELECT statement to print out everything related to a given user
#   this includes posts that the user liked, posts that the user made, an
#   of the user
CREATE QUERY printAllRelatedItems(vertex<person> seed) FOR GRAPH socialNe
{
    sourceVertex = {seed};

    # -- statements produce equivalent output
    # returns all vertices of type either 'person' or 'post' that can be
    #   from the sourceVertex set using one edge of any type
    everythingRelated = SELECT v FROM sourceVertex -(:e)-> (person|post):

    # returns all vertices of any type that can be reached from the sourc
    #   using one edge of any type
    # this statement is equivalent to the above one because the graph sch
    #   has vertex types of either 'person' or 'post'. if there were more
    #   types present, these would not be equivalent.
    everythingRelated = SELECT v FROM sourceVertex -(:e)-> :v;
    # --

    PRINT everythingRelated;
}
```

Results


```
        "attributes": {
          "postTime": "2011-02-03 17:05:52",
          "subject": "cats"
        },
        "v_type": "post"
      }
    ]}
  ]
}
```

```
GSQL > RUN QUERY printAllRelatedItems("person2")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"everythingRelated": [
    {
      "v_id": "0",
      "attributes": {
        "postTime": "2010-01-12 11:22:05",
        "subject": "Graphs"
      },
      "v_type": "post"
    },
    {
      "v_id": "person3",
      "attributes": {
        "gender": "Male",
        "id": "person3"
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {
        "gender": "Male",
        "id": "person1"
      },
      "v_type": "person"
    },
    {
      "v_id": "3",
      "attributes": {
        "postTime": "2011-02-05 01:02:44",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "1",
      "attributes": {
        "postTime": "2011-03-03 23:02:00",
        "subject": "tigergraph"
      },
```

```
      "v_type": "post"
    }
  ]}]
}
GSQL > RUN QUERY printAllRelatedItems("person6")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"everythingRelated": [
    {
      "v_id": "person4",
      "attributes": {
        "gender": "Female",
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "10",
      "attributes": {
        "postTime": "2011-02-04 03:02:31",
        "subject": "cats"
      },
      "v_type": "post"
```

```
      "attributes": {
        "postTime": "2011-02-06 01:02:02",
        "subject": "tigergraph"
      },
      "v_type": "post"
    },
```

```
      "v_type": "person"
    },
    {
      "v_id": "8",
```

Vertex and Edge Aliases

Vertex variables

```
results = SELECT v FROM allVertices:v;
results = SELECT t FROM allVertices:s -()-> :t;
```

Edge variables

```
results = SELECT v
          FROM allVertices:s -(:e)-> :t
          ACCUM VERTEX v = t, EDGE eg = e;
```

```
        "attributes": {
          "postTime": "2011-02-03 17:05:52",
          "subject": "cats"
        },
        "v_type": "post"
      }
    ]}]
  }
```


Pattern Matching (SYNTAX v2)

clause, from the perspective of the formal notation and how the rules can be inferred from that. For a more practical explanation of Pattern Matching, we recommend the [GSQL102 Pattern Matching](#) tutorial.

There are three options for the syntax of the pattern in a FROM clause:

```
fromClause := FROM (step | stepV2 | pathPattern [","  pathPattern]*)
```

1. `step` pattern, SYNTAX v1. This is the default syntax from classic GSQL.
2. `stepV2` pattern, SYNTAX v2. The query or the GSQL session must [specify that SYNTAX v2 is to be used](#). This is still a 1-Hop query, but there are two differences:
   - The source vertex set now has the same flexibility as the target vertex. It is not necessary to create a seedSet. Instead, the source can be one or more vertex types, given either statically or dynamically as string parameters, or the special symbol `ANY` or `_`.
   - The arrowhead `-( )->` should not be used between the edge set and the target set. Simple use a dash `-( )-`.
3. One or more `pathPattern` s.

A pathPattern begins with a `stepVertexSet` and then has one or more hops across and edge set to a target set: `-(pathEdgePattern)-setVertexSet.`

```
pathPattern := stepVertexSet ["-" "(" pathEdgePattern ")" "-" stepVertexS
```

We have already studied stepVertexSet. Now we will look at the options for `pathEdgePattern` and their meanings.

```
pathEdgePattern := atomicEdgePattern
                 | "(" pathEdgePattern ")"
                 | pathEdgePattern "." pathEdgePattern
                 | disjPattern
                 | starPattern

atomicEdgePattern  := atomicEdgeType
                    | atomicEdgeType ">"
                    | "<" atomicEdgeType

atomicEdgeType := "_" | ANY | edgeSetType
edgeSetType := edgeType | paramName | globalAccumName
```

The most basic form for a `pathEdgePattern` is an `atomicEdgePattern`. This in turn can be, similar to `stepVertexSet`, one of the following:

- "_" or "ANY"
- an edgeType, a string parameter, or a global SetAccum accumulator.

Edge Direction

Moreover, an `atomicEdgePattern` can have either a left pointer "<" on the left or a right pointer ">" on the right. These indicate edge direction, of course. If no pointer is used, then the edge is undirected. Suppose we have 3 edge types or parameters called A, B, C.

- A> is a rightward facing A edge
- <B is a leftward facing B edge
- C is an undirected C edge. If C is actually a directed edge type, then there is no match.

Disjunction, Repeats, and Dot Concatenation

Looking at the remaining options for pathEdgePattern, we see that we can have parentheses around it, we can use a dot "." between two pathEdgePatterns, or we can have a disjPattern or starPattern.

disjPattern is how we write "either this edge pattern or that edge pattern""

```
disjPattern := atomicEdgePattern ("|" atomicEdgePattern)*
```

For example, we can combine the three direction-specfic examples from above: `(A> | <B | C)`

starPattern explains how the Kleene star and min..max range specifiers can be used to say "repeat this edge pattern from min to max times."

```
starPattern := ([atomicEdgePattern] | "(" disjPattern ")") "*" [starBound

starBounds := CONST_INT ".." CONST_INT
            | CONST_INT ".."
            | ".." CONST_INT
            | CONST_INT
```

For example, `(A> | <B | C)*2..4` means "a series of 2 to 4 adjacent edges, where each edge has type A>, <B, or C. Note that we do not have to use the same edge type for all the repeats. Either iteration can select an option from `disjPattern`.

The dot operator means concatenate the two edge patterns into one. Naturally, there must be a vertex joining the two edges, but it is omitted from the syntax. The dot operator is a shorthand, when you don't care about the type of that intermediate vertex.
`(A>.<B.C)` means a series of 3 edges, having the specifying types and directions.

Conjunctive Pattern Matching

The optional repeating phrase `["," pathPattern]*` allows you to have multiple pathPatterns. They form a conjunction, meaning all of them must be satisfied in order to have a valid match result.

```
fromClause := FROM (step | stepV2 | pathPattern ["," pathPattern]*)
```

Recall that each step pattern or path pattern forms a match table, one row per matching path in the graph. Each vertex alias or edge alias is one column in the table. When we have a conjunctive path, each path must share at least one vertex alias with another path. This enables the two path sets (and match tables) to be joined. Formally, we make the natural join of the two tables.

This explains all of the syntax for the FROM clause with SYNTAX v2 (Pattern Matching). The other significant area to consider for SYNTAX v2 is the ACCUM and POST-ACCUM clauses.

SAMPLE Clause

The SAMPLE clause is an optional clause that selects a uniform random sample from the population of edges or target vertices specified in the FROM argument.

> If you want to sample from a set of vertices directly, not from edges or from neighboring (target) vertices, then the following technique is simpler and faster:

Select k random vertices from a vertex set S

```
random = SELECT s
         FROM S:s
         LIMIT k;
```

The SAMPLE clause draws from the edge population consisting of those edges which satisfy all three parts – source set, edge type, and target type – of the FROM clause. The SAMPLE clause is intended to provide a representative sample of the distribution of edges (or vertices) connected to *hub* vertices, instead of dealing with all edges. A *hub* vertex is a vertex with a relatively high degree. (The *degree* of a vertex is the number of edges which connect to it. If edges are directional, one can distinguish between indegree and outdegree.)

EBNF for Sample Clause

```
sampleClause := SAMPLE ( expr | expr "%" ) EDGE WHEN condition # sample a
              | SAMPLE expr TARGET WHEN condition              # sample a
              | SAMPLE expr "%" TARGET PINNED WHEN condition   # sample a
```

The expression following SAMPLE specifies the sample size, either an absolute number or a percentage of the population. The expression in sampleClause must evaluate to a positive integer. There are two sampling methods. One is sampling based on edge id. The other is based on target vertex id: if a target vertex id is sampled, all edges from this source vertex to the sampled target vertex are sampled.

Note: Currently, the WHEN condition that can be used with a SAMPLE clause is limited strictly to checking if the result of a function call on a vertex is greater than or greater than/equal to some number.

Given that the sampling is random, some of the details of each of the example queries may change each time they are run.

The following query displays two modes of sampling: an absolute number of edges from a source vertex and a percentage of edges fro a source vertex. We use the computerNet graph (see Appendix D). In computerNet, there are 31 vertices and 43 edges, but only 7 vertices are source vertices. Moreover, c1, c12, and c23 are hub nodes, with at least 10 outgoing edges each. For the absolute count case, we set the size to 1 edge per source vertex, which is equivalent to a random walk. We expect exactly 7 edges to be selected. For the percentage sampling case, we sample 33% of the edges for vertices which have 3 or more outgoing edges. We expect about 15 edges, but the number may vary.

sampleEx3: SAMPLE based on edges per source vertex

```
CREATE QUERY sampleEx3() FOR GRAPH computerNet
{
    MapAccum<STRING,ListAccum<STRING>> @@absEdges; // record each selecte
    SumAccum<INT> @@totalAbs;
    MapAccum<STRING,ListAccum<STRING>> @@pctEdges; // record each selecte
    SumAccum<INT> @@totalPct;

    start = {computer.*};

    # Sample one outgoing edge per source vertex = Random Walk
    absSample = SELECT v FROM start:s -(:e)-> :v
             SAMPLE 1 EDGE WHEN s.outdegree() >= 1    # sample 1 target v
             ACCUM @@absEdges += (s.id -> v.id),
                   @@totalAbs += 1;
    PRINT @@totalAbs, @@absEdges;

    pctSample = SELECT v FROM start:s -(:e)-> :v
             SAMPLE 33% EDGE WHEN s.outdegree() >= 3  # select ~1/3 of ed
             ACCUM @@pctEdges += (s.id -> v.id),
                   @@totalPct += 1;
    PRINT @@totalPct, @@pctEdges;
}
```

sampleEx3.json

```
GSQL > RUN QUERY sampleEx3()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@totalAbs": 7,
      "@@absEdges": {
        "c4": ["c23"],
        "c11": ["c12"],
        "c10": ["c11"],
        "c12": ["c14"],
        "c23": ["c26"],
        "c14": ["c24"],
        "c1": ["c10"]
      }
    },
    {
      "@@totalPct": 13,
      "@@pctEdges": {
        "c4": ["c23"],
        "c11": ["c12"],
        "c10": ["c11"],
        "c12": [
          "c14",
          "c15",
          "c19"
        ],
        "c23": [
          "c29",
          "c25"
        ],
        "c14": [
          "c24",
          "c23"
        ],
        "c1": [
          "c3",
          "c8",
          "c2"
        ]
      }
    }
```

```
  ]
}
```

Below is an example of using SELECT to only traverse one edge for each source vertex. The vertex-attached accumulators @timesTraversedNoSample and @timesTraversedWithSample are used to keep track of the number of times an edge is traversed to reach the target vertex. Without using sampling, this occurs once for each edge; thus @timesTraversedNoSample has the same number as the in-degree of the vertex. With sampling edges, the number of edges is restricted. This is reflected in the @timesTraversedWithSample accumulator. Notice the difference in the result set. Because only one edge per source vertex is traversed when the SAMPLE clause is used, not all target vertices are reached. The vertex **company3** has 3 incident edges, but in one instance of the query execution, it is never reached. Additionally, **company2** has 6 incident edges, but only 4 source vertices sampled an edge incident to **company2** .

example of SAMPLE using an absolute number of edges

```
CREATE QUERY sampleEx1() FOR GRAPH workNet
{
    SumAccum<INT> @timesTraversedNoSample;
    SumAccum<INT> @timesTraversedWithSample;
    workers = {person.*};

    # the 'beforeSample' result set encapsulates the normal functionality
    # a SELECT statement, where 'timesTraversedNoSample' vertex accumulat
    # each edge incident to the vertex.
    beforeSample = SELECT v FROM workers:t -(:e)-> :v
               ACCUM v.@timesTraversedNoSample += 1;

    # The 'afterSample' result set is formed by those vertices which can
    # reached when for each source vertex, only one edge is used for trav
    # This is demonstrated by the values of 'timesTraversedWithSample' ve
    # is increased for each edge incident to the vertex which is used in
    # sample.
    afterSample = SELECT v FROM workers:t -(:e)-> :v
              SAMPLE 1 EDGE WHEN t.outdegree() >= 1     # only use 1 edge
              ACCUM v.@timesTraversedWithSample += 1;

    PRINT beforeSample;
    PRINT afterSample;
}
```

sampleEx1.json

```
GSQL > RUN QUERY sampleEx1()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"beforeSample": [
      {
        "v_id": "company4",
        "attributes": {
          "country": "us",
          "@timesTraversedNoSample": 1,
          "@timesTraversedWithSample": 1,
          "id": "company4"
        },
        "v_type": "company"
      },
      {
        "v_id": "company5",
        "attributes": {
          "country": "can",
          "@timesTraversedNoSample": 1,
          "@timesTraversedWithSample": 1,
          "id": "company5"
        },
        "v_type": "company"
      },
      {
        "v_id": "company3",
        "attributes": {
          "country": "jp",
          "@timesTraversedNoSample": 3,
          "@timesTraversedWithSample": 3,
          "id": "company3"
        },
        "v_type": "company"
      },
      {
        "v_id": "company2",
        "attributes": {
          "country": "chn",
          "@timesTraversedNoSample": 6,
          "@timesTraversedWithSample": 4,
          "id": "company2"
```

```
      },
      "v_type": "company"
    },
    {
      "v_id": "company1",
      "attributes": {
        "country": "us",
        "@timesTraversedNoSample": 6,
        "@timesTraversedWithSample": 3,
        "id": "company1"
      },
      "v_type": "company"
    }
  ]},
  {"afterSample": [
    {
      "v_id": "company4",
      "attributes": {
        "country": "us",
        "@timesTraversedNoSample": 1,
        "@timesTraversedWithSample": 1,
        "id": "company4"
      },
      "v_type": "company"
    },
    {
      "v_id": "company5",
      "attributes": {
        "country": "can",
        "@timesTraversedNoSample": 1,
        "@timesTraversedWithSample": 1,
        "id": "company5"
      },
      "v_type": "company"
    },
    {
      "v_id": "company3",
      "attributes": {
        "@timesTraversedWithSample": 3,
        "id": "company3"
      },
      "v_type": "company"
    },
    {
      "v_id": "company2",
      "attributes": {
```


WHERE Clause

EBNF for Where Clause

```
whereClause := WHERE condition
```

```
            "country": "chn",
            "@timesTraversedNoSample": 6,
            "@timesTraversedWithSample": 4,
            "id": "company2"
          },
          "v_type": "company"
        },
        {
          "v_id": "company1",
          "attributes": {
            "country": "us",
            "@timesTraversedNoSample": 6,
            "@timesTraversedWithSample": 3,
            "id": "company1"
          },
          "v_type": "company"
        }
      ]}
    ]
```


WHERE used as a filter

```
resultSet1 = SELECT v FROM S:v-((E1|E2|E3):e)->(V1|V2):t;
resultSet2 = SELECT v FROM S:v-(:e)->:t
                     WHERE t.type IN ("V1", "V2") AND
                           t IN v.neighbors("E1|E2|E3")
```

The following examples demonstrate using the WHERE clause to limit the resulting vertex set based on a vertex attribute.

Basic SELECT WHERE

```
CREATE QUERY printCatPosts() FOR GRAPH socialNet {
    posts = {post.*};
    catPosts = SELECT v FROM posts:v        # select only those post vert
                WHERE v.subject == "cats";  # which have a subset of 'cat
    PRINT catPosts;
}
```

Results for Query printCatPosts

```
GSQL > RUN QUERY printCatPosts()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"catPosts": [
    {
      "v_id": "10",
      "attributes": {
        "postTime": "2011-02-04 03:02:31",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "9",
      "attributes": {
        "postTime": "2011-02-05 23:12:42",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "3",
      "attributes": {
        "postTime": "2011-02-05 01:02:44",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "11",
      "attributes": {
        "postTime": "2011-02-03 01:02:21",
        "subject": "cats"
      },
      "v_type": "post"
    },
    {
      "v_id": "8",
      "attributes": {
        "postTime": "2011-02-03 17:05:52",
        "subject": "cats"
      },
```

```
      "v_type": "post"
    }
  ]}]
}
```

SELECT WHERE using IN operator

```
CREATE QUERY findGraphFocusedPosts() FOR GRAPH socialNet
{
    posts = {post.*};
    results = SELECT v FROM posts:v                  # select only post ve
        WHERE v.subject IN ("Graph", "tigergraph"); # which have a subjec
    PRINT results;
}
```

Results for Query findGraphFocusedPosts

```
GSQL > RUN QUERY findGraphFocusedPosts()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"results": [
    {
      "v_id": "5",
      "attributes": {
        "postTime": "2011-02-06 01:02:02",
        "subject": "tigergraph"
      },
      "v_type": "post"
    },
    {
      "v_id": "1",
      "attributes": {
        "postTime": "2011-03-03 23:02:00",
        "subject": "tigergraph"
      },
      "v_type": "post"
    },
    {
      "v_id": "6",
      "attributes": {
        "postTime": "2011-02-05 02:02:05",
        "subject": "tigergraph"
      },
      "v_type": "post"
    }
  ]}]
}
```

WHERE NOT limitations

The NOT operator may not be used in combination with the .type attribute selector. To check if an edge or vertex type is not equal to a given type, use the != operator. See the example below.

The following example shows the equivalence of using WHERE as a type filter as well as its limitations.

SELECT WHERE using AND/OR

```
# finds female person in the social network. all of the following stateme
# are equivalent (i.e., produce the same results)
CREATE QUERY findFemaleMembers() FOR GRAPH socialNet
{
    allVertices = {ANY}; # includes all posts and person
    females = SELECT v FROM allVertices:v
          WHERE v.type   == "person" AND
                v.gender != "Male";

    females = SELECT v FROM allVertices:v
          WHERE v.type   == "person" AND
                v.gender == "Female";

    females = SELECT v FROM allVertices:v
          WHERE v.type       == "person" AND
                NOT v.gender == "Male";

    females = SELECT v FROM allVertices:v
          WHERE v.type       != "post" AND
                NOT v.gender == "Male";

    # does not compile. cannot use NOT operator in combination with type
    #females = SELECT v FROM allVertices:v
    #     WHERE NOT v.type   != "person" AND
    #           NOT v.gender == "Male";

    # does not compile. cannot use NOT operator in combination with type
    #females = SELECT v FROM allVertices:v
    #     WHERE NOT v.type   == "post" AND
    #           NOT v.gender == "Male";

    personVertices = {person.*};
    females = SELECT v FROM personVertices:v
           WHERE NOT v.gender == "Male";

    females = SELECT v FROM personVertices:v
           WHERE v.gender != "Male";

    females = SELECT v FROM personVertices:v
           WHERE v.gender != "Male" AND true;

    females = SELECT v FROM personVertices:v
           WHERE v.gender != "Male" OR false;

    PRINT females;
}
```

Results for Query findFemaleMembers

```
GSQL > RUN QUERY findFemaleMembers()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"females": [
    {
      "v_id": "person4",
      "attributes": {
        "gender": "Female",
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {
        "gender": "Female",
        "id": "person5"
      },
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {
        "gender": "Female",
        "id": "person2"
      },
      "v_type": "person"
    }
  ]}]
}
```

The following example uses edge attributes to determine which workers are registered as full time for some company.

WHERE using edge attributes

```
# find all workers who are full time at some company
CREATE QUERY fullTimeWorkers() FOR GRAPH workNet
{
    start = {person.*};
    fullTimeWorkers = SELECT v FROM start:v -(worksFor:e)-> company:t
            WHERE e.fullTime;   # fullTime is a boolean attribute on the

    PRINT fullTimeWorkers;
}
```

fullTimeWorkers Results

```
GSQL > RUN QUERY fullTimeWorkers()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"fullTimeWorkers": [
    {
      "v_id": "person4",
      "attributes": {
        "interestList": ["football"],
        "skillSet": [ 10, 1, 4 ],
        "skillList": [ 4, 1, 10 ],
        "locationId": "us",
        "interestSet": ["football"],
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "person11",
      "attributes": {
        "interestList": [ "sport", "football" ],
        "skillSet": [10],
        "skillList": [10],
        "locationId": "can",
        "interestSet": [ "football", "sport" ],
        "id": "person11"
      },
      "v_type": "person"
    },
    {
      "v_id": "person10",
      "attributes": {
        "interestList": [ "football", "sport" ],
        "skillSet": [3],
        "skillList": [3],
        "locationId": "us",
        "interestSet": [ "sport", "football" ],
        "id": "person10"
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
```

```
    "attributes": {
      "interestList": [ "management", "financial" ],
      "skillSet": [ 3, 2, 1 ],
      "skillList": [ 1, 2, 3 ],
      "locationId": "us",
      "interestSet": [ "financial", "management" ],
      "id": "person1"
    },
    "v_type": "person"
  },
  {
    "v_id": "person6",
    "attributes": {
      "interestList": [ "music", "art" ],
      "skillSet": [ 10, 7 ],
      "skillList": [ 7, 10 ],
      "locationId": "jp",
      "interestSet": [ "art", "music" ],
      "id": "person6"
    },
    "v_type": "person"
  },
  {
    "v_id": "person2",
    "attributes": {
      "interestList": ["engineering"],
      "skillSet": [ 6, 5, 3, 2 ],
      "skillList": [ 2, 3, 5, 6 ],
      "locationId": "chn",
      "interestSet": ["engineering"],
      "id": "person2"
    },
    "v_type": "person"
  },
  {
    "v_id": "person8",
    "attributes": {
      "interestList": ["management"],
      "skillSet": [ 2, 5, 1 ],
      "skillList": [ 1, 5, 2 ],
      "locationId": "chn",
      "interestSet": ["management"],
      "id": "person8"
    },
    "v_type": "person"
  },
Multiple Edge Type WHERE clause
    "v_id": "person12",
```

```
CREATE QUERY multipleEdgeTypeWhereEx(vertex<person> m1) FOR GRAPH
  allUser = {m1};
  FilteredUser = SELECT s
      FROM allUser:s - ((posted|liked|friend):e) -> (post|person):
      # WHERE e.actionTime > epoch_to_datetime(1) AND t.gender ==
      WHERE ( e.type == "liked" AND e.actionTime > epoch_to_dateti
            ( e.type == "friend" AND t.gender == "Male" )
            ;
  PRINT FilteredUser;
}
```


```
        "interestSet": [ "teaching", "engineering", "music" ],
        "id": "person12"
      },
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {
        "interestList": ["teaching"],
        "skillSet": [ 6, 1, 4 ],
        "skillList": [ 4, 1, 6 ],
        "locationId": "jp",
        "interestSet": ["teaching"],
        "id": "person3"
      },
      "v_type": "person"
    },
    {
      "v_id": "person9",
      "attributes": {
        "interestList": [ "financial", "teaching" ],
        "skillSet": [ 2, 7, 4 ],
        "skillList": [ 4, 7, 2 ],
        "locationId": "us",
        "interestSet": [ "teaching", "financial" ],
        "id": "person9"
      },
      "v_type": "person"
    }
  ]}]
}
```


ACCUM and POST-ACCUM Clauses

⚠ The statements within the ACCUM clause are executed sequentially for a given vertex or edge. However, there is no fixed order in which a vertex set or edge set is processed.

The optional POST-ACCUM clause enables aggregation and other computations across the set of vertices (but not edges) selected by the preceding clauses. POST-ACCUM can be used without ACCUM. If it is preceded by an ACCUM clause, then it can be used for 2-stage accumulative computation: a first stage in ACCUM followed by a second stage in POST-ACCUM.

⚠ Each statement within the POST-ACCUM clause can refer to either source vertices or target vertices but not both.

Since the ACCUM clause iterates over edges, and often two edges will connect to the same source vertex or to the same target vertex, the ACCUM clause can be repeated multiple times for one vertex.

⚠ Operations that are to be performed exactly once per vertex should be performed in the POST-ACCUM clause.

The primary purpose of the ACCUM or POST-ACCUM clause is to collect information about the graph by updating accumulators (via += or =). See the "Accumulator" section for details on the += operation. However, other kinds of statements (e.g., branching, iteration, local assignments) are permitted to support more complex computations or to log activity. The EBNF syntax below defines the allowable kinds of statements that can occur within an ACCUM or POST-ACCUM. The **dmlSubStmt** list is similar to the **queryBodyStmt** list which applies to statements outside of a SELECT block; it is important to note the differences. Each of these statement types is discussed in one of the main sections of this reference document.

EBNF for ACCUM and POST-ACCUM Clauses

```
accumClause := [perClauseV2] ACCUM dmlSubStmtList

perClauseV2 := PER "(" alias ["," alias] ")"

postAccumClause := POST-ACCUM dmlSubStmtList

dmlSubStmtList := dmlSubStmt ["," dmlSubStmt]*

dmlSubStmt := assignStmt           // Assignment
            | funcCallStmt         // Function Call
            | gAccumAccumStmt      // Assignment
            | lAccumAccumStmt      // Assignment
            | attrAccumStmt        // Assignment
            | vAccumFuncCall       // Function Call
            | localVarDeclStmt     // Declaration
            | dmlSubCaseStmt       // Control Flow
            | dmlSubIfStmt         // Control Flow
            | dmlSubWhileStmt      // Control Flow
            | dmlSubForEachStmt    // Control Flow
            | BREAK                // Control Flow
            | CONTINUE             // Control Flow
            | insertStmt           // Data Modification
            | dmlSubDeleteStmt     // Data Modification
            | printlnStmt          // Output
            | logStmt              // Output
```

Note that dml-sub-statements do not include global accumulator assignment statement (gAccumAssignStmt) but global accumulator accumulation statement (gAccumAccumStmt). Global accumulators may perform accumulation += but not assignment "=" within these clauses.

There are additional restrictions on dml-sub level statements:

- Global variable assignment is permitted in ACCUM or POST-ACCUM clauses, but the change in value will not take place until the query completes. Therefore, if there are multiple assignment statements for the same variable, only the final one will take effect.
- Vertex attribute assignment "=" is not permitted in an ACCUM clause. However, edge attribute assignment is permitted. This is because the ACCUM clause iterates over an edge set. Vertex attribute attribute assignment is permitted in the POST-ACCUM clause. Like all updates, the change in value does not take place until the query completes.

Aliases and ACCUM/POST-ACCUM Iteration Model

To reference each element of the selected set, use the aliases defined in the FROM clause. For example, assume that we have the following aliases:

Example of vertex and edge aliases

```
FROM source:s -(edgeTypes:e)-> targetTypes:t # edge-induced selection
FROM source:v                                # vertex-induced selection
```

Let (V1, V2,... Vn) be the vertices in the vertex-induced selection . The following pseudocode emulates ACCUM clause behavior.

Model for ACCUM behavior in vertex-induced selection

```
FOREACH v in (V1,V2,...Vn) DO # iterations may occur in parallel, in unkn
  dmlSubStmts referencing v
DONE
```

Let E = (E1, E2,... En) be the edges in the edge-induced selected set. Further, let S = (S1,S1,...Sn) and T= (T1,T2,...Tn) be the multisets (bags) of source vertices and target vertices which correspond to the edge set. S and T are bags, because they can contain repeated elements.

Model for ACCUM behavior in edge-induced selection

```
FOREACH i in (1..n) DO # iterations may occur in parallel, in unknown ord
  dmlSubStmts referencing e, s, t, which really means e_i, s_i, t_i
DONE
```

Note that any reference to the source alias s or target alias t is for the endpoint vertices of the current edge.

Similarly, the POST-ACCUM clause acts like a FOREACH loop on the vertex result set specified in the SELECT clause (e.g., either S or T).

Pattern Matching ACCUM and POST-ACCUM

In Pattern Matching (SYNTAX v2), we may have a multi-step pattern, with several vertex and edge aliases. Each statement (dml-sub-statement) can refer to one or more aliases. There are no restrictions, though complex expressions and accessing multiple aliases may degrade performance.

The PER clause is available only in SYNTAX v2. It is an optional prefix to the ACCUM clause, affecting only that clause. The PER clause allows the user to specify that they wish to aggregate the match table, so that there is one row PER <alias>. For more information see the PER Clause section in the Pattern Matching tutorial.

Multiple POST-ACCUM clauses are supported in TG 3.0+. In SYNTAX v2, Each POST-ACCUM may refer to only one vertex alias. See the POST-ACCUM section in the Pattern Matching tutorial.

Edge/Vertex Type Inference and Conflict

If multiple edge types are specified in edge-induced selection, each ACCUM statement in ACCUM clause checks whether edge types are conflicted. If only a subset of edge types are effective in an ACCUM statement , this statement is not executed on other edge types. For example:

Multiple Edge Type ACCUM statement check

```
CREATE QUERY multipleEdgeTypeCheckEx(vertex<person> m1) FOR GRAPH socialN
  ListAccum<STRING> @@testList1, @@testList2, @@testList3;
  allUser = {m1};
  allUser = SELECT s
         FROM allUser:s - ((posted|liked|friend):e) -> (post|person):t
         ACCUM @@testList1 += to_string(datetime_to_epoch(e.actionTime))
              ,@@testList2 += t.gender
             #,@@testList3 += to_string(datetime_to_epoch(e.actionTime))
               ;
  PRINT @@testList1, @@testList2, @@testList3;
}
```

In the above example, line 6 is only executed on "liked" edges, because "actionTime" is the attribute of "liked" edge only. Similarly, line 7 is only executed on "friend" edges, because "gender" is the attribute of "person" only, and only "friend" edge uses "person" as target vertex. However, line 8 causes a compilation error, because it uses multiple edges where some edges cannot be supported in a part of the statement, i.e., "liked" edges doesn't have t.gender, "friend" edges doesn't have e.actionTime.

> We strongly suggest that if multiple edge types are specified in edge-induced selection, ACCUM clauses should uses CASE statement (see Section "Control Flow Statements" for more details) to separate the operation on each edge type or each target vertex type (or combination of target vertex type and edge type). The edge-type conflict checking then checks the ACCUM statement inside each THEN/ELSE blocks based on the condition. For example,
>
> Multiple Edge Type ACCUM statement check 2
>
> ```
> CREATE QUERY multipleEdgeTypeCheckEx2(vertex<person> m1) FOR GRAPH
>   ListAccum<STRING> @@testList1;
>   allUser = {m1};
>   allUser = SELECT s
>             FROM allUser:s - ((posted|liked|friend):e) -> (post|pe
>             ACCUM CASE
>                     WHEN e.type == "liked" THEN    # for liked edg
>                       @@testList1 += to_string(datetime_to_epoch(e
>                     WHEN e.type == "friend" THEN   # for friend ec
>                       @@testList1 += t.gender
>                     ELSE      # For the remained edge type, which
>                       @@testList1 += to_string(datetime_to_epoch(t
>                   END
>             ;
>   PRINT @@testList1;
> }
> ```
>
> The above query is compilable. However, if we switch line 8 and line 10, the edge-type conflict checking generates errors because "liked" edges doesn't support t.gender and "friend" edges doesn't support e.actionTime.

Similar to the ACCUM clause, if multiple source/target vertex types are specified in edge-induced selection and the POST-ACCUM clauses accesses source/target vertex, each ACCUM statement in POST-ACCUM clause checks whether source/target vertex types are conflicted. If only a subset of source/target vertex types are effective in a POST-ACCUM statement, this statement is not executed on other source/target vertex types.

Similar to ACCUM clause, we strongly suggest that if multiple source/target vertex types are specified in edge-induced selection and the POST-ACCUM clauses accesses source/target vertex, POST-ACCUM clauses should uses CASE statement (see Section "Control Flow Statements" for more details) to separate the operation on each source/target vertex type. The vertex type conflict checking then checks the ACCUM statement inside each THEN/ELSE blocks based on the condition.

Rules for Updating Vertex-Attached Accumulators

Prior to v1.0, a vertex-attached accumulator could only be updated in an ACCUM or POST-ACCUM clause and only if its vertex was selected for by the preceding FROM-SAMPLE-WHERE clauses.

Beginning in v1.0, there are additional circumstances where a vertex-attached accumulator may be updated. Vertices which are *referenced via a vertex-attached accumulator of a selected vertex* may have their vertex-attached accumulators updated in the ACCUM clause (but not in the POST-ACCUM clause). That is, a vertex referenced by an selected vertex can be updated, with some limitations explained below. Some examples will help to illustrate this more complex condition.

- Suppose a query declares a vertex-attached *accumulator which holds vertex information* . We call this a **vertex-holding accumulator** . This could take several forms:
  - A scalar accumulator, e.g., MaxAccum< **VERTEX** > @maxV;
  - A collection accumulator: e.g., ListAccum< **VERTEX** > @listV;
  - An accumulator containing tuple(s), where the tuple type contains a **VERTEX** field.
- If a vertex V is selected, then not only can V's accumulators be updated, but the vertices stored in its vertex-holding accumulators can also be updated, in the ACCUM clause.
- Before these indirectly referenced vertices can be used, they need to be **activated** . There are two ways to activate an indirect vertex:
  - A vertex from a vertex-holding accumulator is first assigned to a local vertex variable. The vertex can now be updated through the local vertex variable.

```
ACCUM
  VERTEX<person> mx = tgt.@maxV,   # assign to local variable
          mx.@curId += src.id      # access via local variable
```

- A FOREACH loop can iterate on a vertex-holding collection accumulator. The vertices can now be updated through the loop variable.

```
ACCUM
  FOREACH vtx IN src.@setIds DO   # iterate on collection accumulator
      vtx.@curId += tgt.id        # access via loop variable
  END
```

The following uses are NOT supported by the new rules:

- Indirectly activated vertices may not be updated in the POST-ACCUM clause or outside of a SELECT statement.
- Passing a vertex into the query as an input parameter is not a route to activation.
- Using a global vertex-holding accumulator is not a route to activation.
- If a vertex is being indirectly activated by assigning it to a local variable (e.g., a variable declaring in ACCUM or POST-ACCUM), note the following rule, which always applies to all local variables:
    - A local variable can be declared and initialized in an ACCUM block once. It cannot be redeclared or reassigned later in the ACCUM block.

The following query demonstrates updates to indirectly activated vertices.

Updating an Indirectly-Referenced Vertex

```
CREATE QUERY vUpdateIndirectAccum() FOR GRAPH socialNet {

  SetAccum<VERTEX<person>> @posters;
  SetAccum<VERTEX<person>> @fellows;

   Persons = {person.*};
   # To each post, attach a list of persons who liked the post
   likedPosts = SELECT p
       FROM Persons:src -(liked:e)-> post:p
       ACCUM
        p.@posters += src;

    # To each person who liked a post, attach a list of everyone
    # who also liked one of this person's liked posts.
    likedPosts = SELECT src
        FROM likedPosts:src
        ACCUM
          FOREACH v IN src.@posters DO
            v.@fellows += src.@posters
          END
        ORDER BY src.subject;

    PRINT Persons[Persons.@fellows];
}
```

Results from Query vUpdateIndirectAccums

```
GSQL > RUN QUERY vUpdateIndirectAccess()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Persons": [
    {
      "v_id": "person4",
      "attributes": {"Persons.@fellows": [
        "person8",
        "person4"
      ]},
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {"Persons.@fellows": [ "person2", "person1", "person3
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {"Persons.@fellows": ["person7"]},
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {"Persons.@fellows": [ "person2", "person1", "person3
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {"Persons.@fellows": ["person5"]},
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {"Persons.@fellows": ["person6"]},
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {"Persons.@fellows": [ "person2", "person1", "person3
      "v_type": "person"
    },
```

```
    },
    {
      "v_id": "person8",
      "attributes": {"Persons.@fellows": [ "person8", "person4" ]},
      "v_type": "person"
    }
  ]}]
}
```

ACCUM and POST-ACCUM Examples

We now show several examples. This example demonstrates how ACCUM or POST-ACCUM can be used to count the number of vertices in the given set.

Accum and PostAccum Semantics

```
#Show Accum PostAccum Behavior
CREATE QUERY accumPostAccumSemantics() FOR GRAPH workNet {

  SumAccum<INT> @@vertexOnlyAccum;
  SumAccum<INT> @@vertexOnlyPostAccum;

  SumAccum<INT> @@vertexOnlyWhereAccum;
  SumAccum<INT> @@vertexOnlyWherePostAccum;

  SumAccum<INT> @@sourceWithEdgeAccum;
  SumAccum<INT> @@sourceWithEdgePostAccum;

  SumAccum<INT> @@targetWithEdgeAccum;
  SumAccum<INT> @@targetWithEdgePostAccum;

  #Seed start set with all company vertices
  start = {company.*};

  #Select all vertices in source set start
  selectVertexSet = SELECT v from start:v
                    #Happens once for each vertex discovered
                    ACCUM @@vertexOnlyAccum += 1

                    #Happens once for each vertex in the result set "v"
                    POST-ACCUM @@vertexOnlyPostAccum += 1;

  #Select all vertices in source set start with a where constraint
  selectVertexSetWhere = SELECT v from start:v WHERE (v.country == "us")
                        #Happens once for each vertex discovered that als
                        # meets the constraint condition
                        ACCUM @@vertexOnlyWhereAccum += 1

                        #Happens once for each vertex in the result set "
                        POST-ACCUM @@vertexOnlyWherePostAccum += 1;

  #Select all source "s" vertices in set start and explore all "worksFor"
  selectSourceWithEdge = SELECT s from start:s -(worksFor)-> :t
                         #Happens once for each "worksFor" edge discovere
                         ACCUM @@sourceWithEdgeAccum += 1

                        #Happens once for each vertex in result set "s" (
                        POST-ACCUM @@sourceWithEdgePostAccum += 1;

  #Select all target "t" vertices found from exploring all "worksFor" edg
  selectTargetWithEdge = SELECT t from start:s -(worksFor)-> :t
                         #Happens once for each "worksFor" edge discovere
                         ACCUM @@targetWithEdgeAccum += 1
```

```
                        #Happens once for each vertex in result set "t"
                        POST-ACCUM @@targetWithEdgePostAccum += 1;

  PRINT @@vertexOnlyAccum;
  PRINT @@vertexOnlyPostAccum;

  PRINT @@vertexOnlyWhereAccum;
  PRINT @@vertexOnlyWherePostAccum;

  PRINT @@sourceWithEdgeAccum;
  PRINT @@sourceWithEdgePostAccum;

  PRINT @@targetWithEdgeAccum;
  PRINT @@targetWithEdgePostAccum;
}
```

accumPostAccumSemantics Result

```
GSQL > RUN QUERY accumPostAccumSemantics()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@vertexOnlyAccum": 5},
    {"@@vertexOnlyPostAccum": 5},
    {"@@vertexOnlyWhereAccum": 2},
    {"@@vertexOnlyWherePostAccum": 2},
    {"@@sourceWithEdgeAccum": 17},
    {"@@sourceWithEdgePostAccum": 5},
    {"@@targetWithEdgeAccum": 17},
    {"@@targetWithEdgePostAccum": 12}
  ]
}
```

This example uses ACCUM to find all the subjects a user posted about.

Vertex ACCUM Example

```
# For each person, make a list of all their post subjects
CREATE QUERY userPosts() FOR GRAPH socialNet {
  ListAccum<STRING> @personPosts;
  start = {person.*};

  # Find all user post topics and append them to the vertex list accum
  userPostings = SELECT s FROM start:s -(posted)-> :g
                 ACCUM s.@personPosts += g.subject;

  PRINT userPostings;
}
```

Results for Query userPosts

```
GSQL > RUN QUERY userPosts()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"userPostings": [
    {
      "v_id": "person4",
      "attributes": {
        "gender": "Female",
        "@personPosts": ["cats"],
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {
        "gender": "Male",
        "@personPosts": ["query languages"],
        "id": "person3"
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "@personPosts": [ "cats", "tigergraph" ],
        "id": "person7"
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {
        "gender": "Male",
        "@personPosts": ["Graphs"],
        "id": "person1"
      },
      "v_type": "person"
    },
/*** other vertices omitted ***/
  ]}]
```

```
  ...
}
```

This example shows each person's posted vertices and each person's like behaviors (liked edges).

ACCUM<VERTEX> and ACCUM<EDGE> Example

```
# Show each user's post and liked post time
CREATE QUERY userPosts2() FOR GRAPH socialNet {
  ListAccum<VERTEX> @personPosts;
  ListAccum<EDGE> @personLikedInfo;
  start = {person.*};

  # Find all user post topics and append them to the vertex list accum
  userPostings = SELECT s FROM start:s -(posted)-> :g
                 ACCUM s.@personPosts += g;

  userPostings = SELECT s from start:s -(liked:e)-> :g
                 ACCUM s.@personLikedInfo += e;

  PRINT start;
}
```

Results from Query userPosts2

```
GSQL > RUN QUERY userPosts2()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"start": [
    {
      "v_id": "person4",
      "attributes": {
        "gender": "Female",
        "@personPosts": ["3"],
        "id": "person4",
        "@personLikedInfo": [{
          "from_type": "person",
          "to_type": "post",
          "directed": true,
          "from_id": "person4",
          "to_id": "4",
          "attributes": {"actionTime": "2010-01-13 03:16:05"},
          "e_type": "liked"
        }]
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "@personPosts": [ "9", "6" ],
        "id": "person7",
        "@personLikedInfo": [{
          "from_type": "person",
          "to_type": "post",
          "directed": true,
          "from_id": "person7",
          "to_id": "10",
          "attributes": {"actionTime": "2010-01-12 11:22:05"},
          "e_type": "liked"
        }]
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
```

```
      "attributes": {
        "gender": "Male",
        "@personPosts": ["0"],
        "id": "person1",
        "@personLikedInfo": [{
          "from_type": "person",
          "to_type": "post",
          "directed": true,
          "from_id": "person1",
          "to_id": "0",
          "attributes": {"actionTime": "2010-01-11 11:32:00"},
          "e_type": "liked"
        }]
      },
      "v_type": "person"
    },
/*** other vertices omitted ***/
  ]}]
}
```


This example counts the total number of times each topic is used.

Global ACCUM Example

```
# Show number of total posts by topic
CREATE QUERY userPostsByTopic() FOR GRAPH socialNet {
  MapAccum<STRING, INT> @@postTopicCounts;
  start = {person.*};

  # Append subject and update the appearance count in the global map accu
  posts = SELECT g FROM start -(posted)-> :g
          ACCUM @@postTopicCounts += (g.subject -> 1);

  PRINT @@postTopicCounts;
}
```

Results for Query userPostsByTopic

```
GSQL > RUN QUERY userPostsByTopic()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@postTopicCounts": {
    "cats": 5,
    "coffee": 1,
    "query languages": 1,
    "Graphs": 2,
    "tigergraph": 3
  }}]
}
```

This is an example of using ACCUM and POST-ACCUM in conjunction. The ACCUM traverses the graph and finds all people who live and work in the same country. After this is determined, POST-ACCUM examines each vertex (person) to see if they work where they live.

Vertex POST-ACCUM Example

```
#Show all person who both work and live in the same country
CREATE QUERY residentEmployees() FOR GRAPH workNet {

  ListAccum<STRING> @company;
  OrAccum @worksAndLives;

  start = {person.*};


  employees = SELECT s FROM start:s -(worksFor)-> :c
              #If a person works for a company in the same country where
              # add the company to the list
              ACCUM CASE WHEN (s.locationId == c.country) THEN
                           s.@company += c.id
                         END

              #Check each vertex and see if a person works where they liv
              POST-ACCUM CASE WHEN (s.@company.size() > 0) THEN
                           s.@worksAndLives += True
                         ELSE
                           s.@worksAndLives += False
                         END;

  PRINT employees WHERE (employees.@worksAndLives == True);
}
```

residentEmployees Result

```
GSQL > RUN QUERY residentEmployees()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"employees": [
    {
      "v_id": "person11",
      "attributes": {
        "interestList": [
          "sport",
          "football"
        ],
        "skillSet": [10],
        "skillList": [10],
        "@worksAndLives": true,
        "locationId": "can",
        "interestSet": [ "football", "sport" ],
        "id": "person11",
        "@company": ["company5"]
      },
      "v_type": "person"
    },
    {
      "v_id": "person10",
      "attributes": {
        "interestList": [ "football", "sport" ],
        "skillSet": [3],
        "skillList": [3],
        "@worksAndLives": true,
        "locationId": "us",
        "interestSet": [ "sport", "football" ],
        "id": "person10",
        "@company": ["company1"]
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {
        "interestList": [ "management", "financial" ],
        "skillSet": [ 3, 2, 1 ],
        "skillList": [ 1, 2, 3 ],
        "@worksAndLives": true,
```

```
          "locationId": "us",
          "interestSet": [ "financial", "management" ],
          "id": "person1",
          "@company": ["company1"]
        },
        "v_type": "person"
      },
      {
        "v_id": "person2",
        "attributes": {
          "interestList": ["engineering"],
          "skillSet": [ 6, 5, 3, 2 ],
          "skillList": [ 2, 3, 5, 6 ],
          "@worksAndLives": true,
          "locationId": "chn",
          "interestSet": ["engineering"],
          "id": "person2",
          "@company": ["company2"]
        },
        "v_type": "person"
      }
    ]}]
  }
```


This is an example of a POST-ACCUM only that counts the number people with a particular gender.

Global POST-ACCUM Example

```
#Count the number of person of  a given gender
CREATE QUERY personGender(STRING gender) FOR GRAPH socialNet {

  SumAccum<INT> @@genderCount;

  start = {ANY};

  # Select all person vertices and check the gender attribute
  friends = SELECT v FROM start:v
            WHERE v.type == "person"

            POST-ACCUM CASE WHEN (start.gender == gender) THEN
                         @@genderCount += 1
                       END;

  PRINT @@genderCount;
}
```

Results for Query personGender

```
GSQL > RUN QUERY personGender("Female")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@genderCount": 3}]
}
```

HAVING Clause

The optional HAVING clause provides constraints on the result set of the SELECT. The constraints are applied **after** ACCUM and POST-ACCUM actions. This differs from the WHERE clause, which is applied **before** the ACCUM and POST-ACCUM actions.

EBNF for HAVING Clause

```
havingClause := HAVING condition
```

The condition in the `HAVING` clause is applied to each vertex in the SELECT set (either source or target vertices) which also fulfilled the FROM and WHERE conditions. The HAVING clause is intended to test one or more of the accumulator variables that were updated in the ACCUM or POST-ACCUM clause, though the condition may be anything that equates to a boolean value. If the condition is false for a particular vertex, then that vertex is excluded from the result set.

The following example demonstrates using the HAVING clause to constrain a result set based on the vertex accumulator variable which was updated during the ACCUM clause.

Example 1. HAVING

```
# find all persons meeting a given activityThreshold, based on how many p
CREATE QUERY activeMembers(int activityThreshold) FOR GRAPH socialNet
{
        SumAccum<int> @activityAmount;
        start = {person.*};
        result = SELECT v FROM start:v -(:e)-> post:tgt
                          ACCUM v.@activityAmount +=1
                          HAVING v.@activityAmount >= activityThreshold;
        PRINT result;
}
```

If the activityThreshold parameter is set to 3, the query returns 5 vertices:

Example 1 Results

```
GSQL > RUN QUERY activeMembers(3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"result": [
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "@activityAmount": 3,
        "id": "person7"
      },
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {
        "gender": "Female",
        "@activityAmount": 3,
        "id": "person5"
      },
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {
        "gender": "Male",
        "@activityAmount": 3,
        "id": "person6"
      },
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {
        "gender": "Female",
        "@activityAmount": 3,
        "id": "person2"
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
```

```
      "attributes": {
        "gender": "Male",
        "@activityAmount": 3,
        "id": "person8"
      },
      "v_type": "person"
    }
  ]}]
}
```

If the activityThreshold parameter is set to 2, the query would return 8 vertices. With activityThreshold = 4, the query would return no vertices.

The following example demonstrates the equivalence of a SELECT statement in which the condition for the HAVING clause is always true.

Example 2. HAVING with literal condition

```
# find all person meeting a given activityThreshold, based on how many po
CREATE QUERY printMemberActivity() FOR GRAPH socialNet
{
        SumAccum<int> @activityAmount;
        start = {person.*};

        #### --- equivalent statements -----
        result = SELECT v FROM start:v -(:e)-> post:tgt
                          ACCUM v.@activityAmount +=1
                          HAVING true;

        result = SELECT v FROM start:v -(:e)-> post:tgt
                          ACCUM v.@activityAmount +=1;
        #### -----

        PRINT result;
}
```

Results from Query printMemberActivity

```
GSQL > RUN QUERY printMemberActivity()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"result": [
    {
      "v_id": "person4",
      "attributes": {
        "gender": "Female",
        "@activityAmount": 4,
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {
        "gender": "Male",
        "@activityAmount": 4,
        "id": "person3"
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "gender": "Male",
        "@activityAmount": 6,
        "id": "person7"
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {
        "gender": "Male",
        "@activityAmount": 4,
        "id": "person1"
      },
      "v_type": "person"
    },
    {
      "v_id": "person5",
```

```
        "attributes": {
          "gender": "Female",
          "@activityAmount": 6,
          "id": "person5"
        },
        "v_type": "person"
      },
      {
        "v_id": "person6",
        "attributes": {
          "gender": "Male",
          "@activityAmount": 6,
          "id": "person6"
        },
        "v_type": "person"
      },
      {
        "v_id": "person2",
        "attributes": {
          "gender": "Female",
          "@activityAmount": 6,
          "id": "person2"
        },
        "v_type": "person"
      },
      {
        "v_id": "person8",
        "attributes": {
          "gender": "Male",
          "@activityAmount": 6,
          "id": "person8"
        },
        "v_type": "person"
      }
    ]}]
  }
```


The following shows an example of equivalent result sets from using WHERE vs. HAVING. Recall that the WHERE clause is evaluated before the ACCUM and that the HAVING clause is evaluated after the ACCUM. Both constrain the result set based on a condition that vertices must meet.

Example 3. HAVING vs. WHERE

```
# Compute the total post activity for each male person.
# Because the gender of the vertex does not change, evaluating whether th
# is male before (WHERE) the ACCUM clause or after (HAVING) the ACCUM cla
# change the result. However, if the condition in the HAVING clause could
# the ACCUM clause, these statements would produce different results.

CREATE QUERY activeMaleMembers() FOR GRAPH socialNet
{
    SumAccum<INT> @activityAmount;
    start = {person.*};

    #### --- statements produce equivalent results
    result1 = SELECT v FROM start:v -(:e)-> post:tgt
                      WHERE v.gender == "Male"
                      ACCUM v.@activityAmount +=1;

    result2 = SELECT v FROM start:v -(:e)-> post:tgt
                      ACCUM v.@activityAmount +=1
                      HAVING v.gender == "Male";

    PRINT result2[result2.@activityAmount];
    PRINT result2[result2.@activityAmount];
}
```

Results from Query ActiveMaleMembers

```
GSQL > RUN QUERY activeMaleMembers()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"result2": [
      {
        "v_id": "person3",
        "attributes": {"result2.@activityAmount": 4},
        "v_type": "person"
      },
      {
        "v_id": "person7",
        "attributes": {"result2.@activityAmount": 6},
        "v_type": "person"
      },
      {
        "v_id": "person1",
        "attributes": {"result2.@activityAmount": 4},
        "v_type": "person"
      },
      {
        "v_id": "person6",
        "attributes": {"result2.@activityAmount": 6},
        "v_type": "person"
      },
      {
        "v_id": "person8",
        "attributes": {"result2.@activityAmount": 6},
        "v_type": "person"
      }
    ]},
    {"result2": [
      {
        "v_id": "person3",
        "attributes": {"result2.@activityAmount": 4},
        "v_type": "person"
      },
      {
        "v_id": "person7",
        "attributes": {"result2.@activityAmount": 6},
        "v_type": "person"
      },
```

```
      {
        "v_id": "person1",
        "attributes": {"result2.@activityAmount": 4},
        "v_type": "person"
      },
      {
        "v_id": "person6",
        "attributes": {"result2.@activityAmount": 6},
        "v_type": "person"
      },
      {
        "v_id": "person8",
        "attributes": {"result2.@activityAmount": 6},
        "v_type": "person"
      }
    ]}
  ]
}
```

The following example has a compilation error because the result set is taken from the source vertices, but the HAVING condition is checking the target vertices.

Example 4. HAVING the wrong vertex set

```
# find all person having a post subject about cats
# This query is illegal because the having condition is testing the wrong
CREATE QUERY printMemberAboutCats() FOR GRAPH socialNet
{
        start = {person.*};

        result = SELECT v FROM start:v -(:e)-> post:tgt
                          HAVING tgt.subject == "cats";
        PRINT result;
}
```

Compilation Error for printMemberAboutCats

```
> gsql printMemberAboutCats.gsql
Semantic Check Error in query printMemberAboutCats (SEM-50): line 8, col
The SELECT block selects src, but the HAVING clause uses tgt
```

ORDER BY Clause

The optional ORDER BY clause sorts the result set.

EBNF for ORDER BY Clause

```
orderClause := ORDER BY expr [ASC | DESC] ["," expr [ASC | DESC]]*
```

ASC specifies ascending order (least value first), and DESC specifies descending order (greatest value first). If neither is specified, then ascending order is used. Each expr must refer to the attributes or accumulators of a member of the result set, and the expr must evaluate to a sortable value (e.g., a number or a string). ORDER BY offers hierarchical sorting by allowing a comma-separated list of expressions, sorting first by the leftmost expr. It uses the next expression only to sort items where the current sort expr results in identical values. Any items in the result set which cannot be sorted (because the sort expressions do not pertain to them) will appear at the end of the set, after the sorted items.

In tabular SELECT queries, the ORDER BY expressions may only be SELECT column aliases.

The following example demonstrates the use of ORDER BY with multiple expressions. The returned vertex set is first ordered by the number of friends of the vertex, and then ordered by the number of coworkers of that vertex.

topPopular.gsql: ORDER BY Descending

```
# find the most popular people, sorting first based on the number as frie
# and then in case of a tie by the number of coworkers
CREATE QUERY topPopular() FOR GRAPH friendNet
{
    SumAccum<INT> @numFriends;
    SumAccum<INT> @numCoworkers;
    start = {person.*};

    result = SELECT v FROM start -((friend|coworker):e)-> person:v
             ACCUM CASE WHEN e.type == "friend" THEN v.@numFriends += 1
                        WHEN e.type == "coworker" THEN v.@numCoworkers += 1
               END
         ORDER BY v.@numFriends DESC, v.@numCoworkers DESC;

    PRINT result;
}
```

topPopular.json

```
GSQL > RUN QUERY topPopular()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"result": [
    {
      "v_id": "person9",
      "attributes": {
        "@numCoworkers": 3,
        "@numFriends": 5,
        "id": "person9"
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {
        "@numCoworkers": 1,
        "@numFriends": 4,
        "id": "person8"
      },
      "v_type": "person"
    },
    {
      "v_id": "person12",
      "attributes": {
        "@numCoworkers": 1,
        "@numFriends": 4,
        "id": "person12"
      },
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {
        "@numCoworkers": 4,
        "@numFriends": 3,
        "id": "person6"
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
```

```
        "attributes": {
          "@numCoworkers": 3,
          "@numFriends": 3,
          "id": "person1"
        },
        "v_type": "person"
      },
      {
        "v_id": "person4",
        "attributes": {
          "@numCoworkers": 5,
          "@numFriends": 2,
          "id": "person4"
        },
        "v_type": "person"
      },
      {
        "v_id": "person3",
        "attributes": {
          "@numCoworkers": 3,
          "@numFriends": 2,
          "id": "person3"
        },
        "v_type": "person"
      },
      {
        "v_id": "person2",
        "attributes": {
          "@numCoworkers": 3,
          "@numFriends": 2,
          "id": "person2"
        },
        "v_type": "person"
```

LIMIT Clause

EBNF for LIMIT Clause

```
limitClause := LIMIT ( expr | expr "," expr | expr OFFSET expr )
```

```
        "attributes": {
          "@numCoworkers": 1,
          "@numFriends": 2,
          "id": "person10"
        },
        "v_type": "person"
      },
      {
        "v_id": "person7",
        "attributes": {
          "@numCoworkers": 6,
          "@numFriends": 1,
```

LIMIT scenarios

```
result = SELECT v FROM S -(:e)-> :v LIMIT k;          # case 1: k = Count
result = SELECT v FROM S -(:e)-> :v LIMIT j, k;       # case 2: j = Offse
result = SELECT v FROM S -(:e)-> :v LIMIT k OFFSET j; # case 3: k = Count
```


```
: "person7"
        "v_id": "person5",
        "attributes": {
          "@numCoworkers": 5,
          "@numFriends": 1,
          "id": "person5"
        },
        "v_type": "person"
      },
      {
        "v_id": "person11",
        "attributes": {
          "@numCoworkers": 1,
          "@numFriends": 1,
          "id": "person11"
        },
        "v_type": "person"
      }
    ]}]
  }
```


- The behavior of Case 3 is the same as that of Case 2, except that the syntax is different. The keyword OFFSET separates the two expressions, and the count comes before the offset, rather than vice versa. If the list has at least 7 items, then LIMIT 5 OFFSET 2 would return [v_3 , v_4 , v_5, v_6 , v_7].

If any of the expressions evaluate to a negative integer, the results are undefined.

> OFFSET is intended for result sets which are in a known order. It is a compile time error to use OFFSET without the ORDER BY clause.

The following examples demonstrate the various forms of the LIMIT clause.

The first example shows the LIMIT clause when used as an upper limit. It returns a result set with a maximum size of 4 elements in the set.

limitEx1.gsql: LIMIT by some number

```
CREATE QUERY limitEx1(INT k) FOR GRAPH friendNet
{
        start = {person.*};

        result1 = SELECT v FROM start:v
                ORDER BY v.id
                LIMIT k;

        PRINT result1[result1.id]; // api v2
}
```

limit1Ex.json Results

```
GSQL > RUN QUERY limitEx1(4)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"result1": [
    {
      "v_id": "person1",
      "attributes": {"result1.id": "person1"},
      "v_type": "person"
    },
    {
      "v_id": "person10",
      "attributes": {"result1.id": "person10"},
      "v_type": "person"
    },
    {
      "v_id": "person11",
      "attributes": {"result1.id": "person11"},
      "v_type": "person"
    },
    {
      "v_id": "person12",
      "attributes": {"result1.id": "person12"},
      "v_type": "person"
    }
  ]}]
}
```

The following example shows how to use the LIMIT clause with an offset.

limit2Ex.gsql: LIMIT with lower-bound and size

```
CREATE QUERY limitEx2(INT j, INT k) FOR GRAPH friendNet
{
        start = {person.*};
        result2 = SELECT v FROM start:v
                ORDER BY v.id
                LIMIT j, k;

        PRINT result2[result2.id]; // api v2
}
```

limit2Ex.json Results

```
GSQL > RUN QUERY limitEx2(2,3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"result2": [
    {
      "v_id": "person11",
      "attributes": {"result2.id": "person11"},
      "v_type": "person"
    },
    {
      "v_id": "person12",
      "attributes": {"result2.id": "person12"},
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {"result2.id": "person2"},
      "v_type": "person"
    }
  ]}]
}
```

The following example shows the alternative syntax for a result size limit with an offset. This time we try larger values for offset and size. In a large data set, limitTest(5,20) might return 20 vertices, but since we don't have 25 vertices in the original data, the output was fewer than 20 vertices.

limit3Ex.gsql: LIMIT with OFFSET

```
CREATE QUERY limitEx3(INT j, INT k) FOR GRAPH friendNet
{
        start = {person.*};

        result3 = SELECT v FROM start:v
                ORDER BY v.id
                LIMIT k OFFSET j;

        PRINT result3[result3.id]; // api v2
}
```

limit3Ex.json Results

```
GSQL > RUN QUERY limitEx3(5,20)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"result3": [
    {
      "v_id": "person3",
      "attributes": {"result3.id": "person3"},
      "v_type": "person"
    },
    {
      "v_id": "person4",
      "attributes": {"result3.id": "person4"},
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {"result3.id": "person5"},
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {"result3.id": "person6"},
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {"result3.id": "person7"},
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {"result3.id": "person8"},
      "v_type": "person"
    },
    {
      "v_id": "person9",
      "attributes": {"result3.id": "person9"},
      "v_type": "person"
    }
  ]}]
}
```

Loading...

Control Flow Statements

The GSQL Query Language includes a comprehensive set of control flow statements to empower sophisticated graph traversal and data computation: IF/ELSE, CASE, WHILE, and FOREACH.

Differences in Block Syntax

Note that any of these statements can be used as a query-body statement or as a DML-sub level statement.

If the control flow statement is at the query-body level, then its block(s) of statements are query-body statements (*queryBodyStmts*). In a *queryBodyStmts* block , each individual statement ends with a semicolon, so there is always semicolon at the end.

If the control flow statement is at the DML-sub level, then its block(s) of statements are DML-sub statements (*dmlSubStmtList*). In a *dmlSubStmtList* block, a comma separates statements, but there is no punctuation at the end.

The "Statement Types" subsection in the Chapter on "CREATE / INSTALL / RUN / SHOW / DROP QUERY" has a more detailed general example of the difference between queryBodyStmts and dmlSubStmts.

IF Statement

The IF statement provides conditional branching: execute a block of statements (*queryBodyStmts* or *dmlSubStmtList*) only if a given *condition* is true. The IF statement allows for zero or more ELSE-IF clauses, followed by an optional ELSE clause. The IF statement can be used either at the query-body level or at the DML-sub-statement level. (See the note about differences in block syntax .)

IF syntax

```
queryBodyIfStmt := IF condition THEN queryBodyStmts
                  [ELSE IF condition THEN queryBodyStmts ]*
                  [ELSE queryBodyStmts ] END
dmlSubIfStmt :=    IF condition THEN dmlSubStmtList
                  [ELSE IF condition THEN dmlSubStmtList ]*
                  [ELSE dmlSubStmtList ] END
```

If a particular IF condition is not true, then the flow proceeds to the next ELSE IF condition. When a true condition is encountered, its corresponding block of statements is executed, and then the IF statement terminates (skipping any remaining ELSE-IF or ELSE clauses). If an ELSE-clause is present, its block of statements are executed if none of the preceding conditions are true. Overall, the functionality can be summarized as "execute the first block of statements whose conditional test is true."

IF semantics

```
# if then
IF x == 5 THEN y = 10; END;      # y is assigned to 10 only if x is 5.

# if then else
IF x == 5 THEN y = 10;         # y is 10 only if x is 5.
ELSE y = 20; END;              # y is 20 only if x is NOT 5.

#if with ELSE IF
IF x == 5 THEN y = 10;         # y is 10 only if x is 5.
ELSE IF x == 7 THEN y = 5;     # y is 5  only if x is 7.
ELSE y = 20; END;              # y is 20 only if x is NOT 5 and NOT 7.
```

Example 1. countFriendsOf2.gsql : Simple IF-ELSE at query-body level

```
# count the number of friends a person has, and optionally include cowork
CREATE QUERY countFriendsOf2(vertex<person> seed, BOOL includeCoworkers)
{
       SumAccum<INT> @@numFriends = 0;
       start = {seed};

       IF includeCoworkers THEN
          friends = SELECT v FROM start -((friend | coworker):e)-> :v
                ACCUM @@numFriends +=1;
       ELSE
          friends = SELECT v FROM start -(friend:e)-> :v
                ACCUM @@numFriends +=1;
       END;
       PRINT @@numFriends, includeCoworkers;
}
```

Example 1 Results

```
GSQL > RUN QUERY countFriendsOf2("person2", true)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "@@numFriends": 5,
    "includeCoworkers": true
  }]
}
GSQL > RUN QUERY countFriendsOf2("person2", false)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "@@numFriends": 2,
    "includeCoworkers": false
  }]
}
```

Example 2. IF-ELSE IF-ELSE at query-body level

```
# determine if a user is active in terms of social networking (i.e., post
CREATE QUERY calculateActivity(vertex<person> seed) FOR GRAPH socialNet
{
    SumAccum<INT> @@numberPosts = 0;
    start = {seed};
    result = SELECT postVertex FROM start -(posted:e)-> :postVertex
           ACCUM @@numberPosts += 1;

    IF  @@numberPosts < 2 THEN
       PRINT "Not very active";
    ELSE IF @@numberPosts < 3 THEN
       PRINT "Semi-active";
    ELSE
       PRINT "Very active";
    END;
}
```

Example 2 Results for Query calculateActivity

```
GSQL > RUN QUERY calculateActivity("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Not very active": "Not very active"}]
}
GSQL > RUN QUERY calculateActivity("person5")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Semi-active": "Semi-active"}]
}
```

Example 3. Nested IF at query-body level

```
# use a more advanced activity calculation, taking into account number of
# and number of likes that a user made
CREATE QUERY calculateInDepthActivity(vertex<person> seed) FOR GRAPH soci
{
    SumAccum<INT> @@numberPosts = 0;
    SumAccum<INT> @@numberLikes = 0;
    start = {seed};
    result = SELECT postVertex FROM start -(posted:e)-> :postVertex
           ACCUM @@numberPosts += 1;
    result = SELECT likedPost FROM start -(liked:e)-> :likedPost
           ACCUM @@numberLikes += 1;

    IF @@numberPosts < 2 THEN
        IF @@numberLikes < 1 THEN
            PRINT "Not very active";
        ELSE
            PRINT "Semi-active";
        END;
    ELSE IF @@numberPosts < 3 THEN
        IF @@numberLikes < 2 THEN
            PRINT "Semi-active";
        ELSE
            PRINT "Active";
        END;
    ELSE
        PRINT "Very active";
    END;
}
```

Example 3 Results for Query calculateInDepthActivity

```
GSQL > RUN QUERY calculateInDepthActivity("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Semi-active": "Semi-active"}]
}
```

Example 4. Nested IF at DML-sub level

```
# give each user post an accumulated rating based on the subject and how
# This query is equivalent to the query ratePosts shown above
CREATE QUERY ratePosts2() FOR GRAPH socialNet {
    SumAccum<INT> @rating = 0;
    allPeople = {person.*};

    results = SELECT v FROM allPeople -(:e)-> post:v
        ACCUM IF e.type == "posted" THEN
                IF v.subject == "cats" THEN
                  v.@rating += -1     # -1 if post is about cats
                ELSE IF v.subject == "Graphs" THEN
                  v.@rating += 2      # +2 if post is about graphs
                ELSE IF v.subject == "tigergraph" THEN
                  v.@rating += 10     # +10 if post is about tigergraph
                END
              ELSE IF e.type == "liked" THEN
                v.@rating += 3                            # +3 each time
            END
        ORDER BY v.@rating DESC
        LIMIT 5;
    PRINT results;
}
```

Example 4 Results for Query ratePosts2

```
GSQL > RUN QUERY ratePosts2()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"results": [
    {
      "v_id": "6",
      "attributes": {
        "postTime": "2011-02-05 02:02:05",
        "subject": "tigergraph",
        "@rating": 13
      },
      "v_type": "post"
    },
    {
      "v_id": "0",
      "attributes": {
        "postTime": "2010-01-12 11:22:05",
        "subject": "Graphs",
        "@rating": 11
      },
      "v_type": "post"
    },
    {
      "v_id": "1",
      "attributes": {
        "postTime": "2011-03-03 23:02:00",
        "subject": "tigergraph",
        "@rating": 10
      },
      "v_type": "post"
    },
    {
      "v_id": "5",
      "attributes": {
        "postTime": "2011-02-06 01:02:02",
        "subject": "tigergraph",
        "@rating": 10
      },
      "v_type": "post"
    },
    {
      "v_id": "4",
```

```
        "attributes": {
          "postTime": "2011-02-07 05:02:51",
          "subject": "coffee",
          "@rating": 6
        },
        "v_type": "post"
      }
    ]}]
  }
```

CASE Statement

The CASE statement provides conditional branching: execute a block of statements only if a given condition is true. CASE statements can be used as query-body statements or DML-sub-statements. (See the note about differences in block syntax ↗.)

CASE syntax

```
queryBodyCaseStmt := CASE   (WHEN condition THEN queryBodyStmts)+ [ELSE qu
                   | CASE expr (WHEN constant  THEN queryBodyStmts)+ [ELSE qu
dmlSubCaseStmt := CASE      (WHEN condition THEN dmlSubStmtList)+ [ELSE dm
                   | CASE expr (WHEN constant  THEN dmlSubStmtList)+ [ELSE dm
```

One CASE statement contains one or more WHEN-THEN clauses, each WHEN presenting one expression. The CASE statement may also have one ELSE clause whose statements are executed if none of the preceding conditions are true.

There are two syntaxes of the CASE statement: one equivalent to an if-else statement, and the other is structured like a switch statement. The if-else version evaluates the boolean *condition* within each WHEN-clause and executes the first block of statements whose *condition* is true. The optional concluding ELSE-clause is executed only if all WHEN-clause conditions are false.

The switch version evaluates the expression following the keyword WHEN and compares its value to the expression immediately following the keyword CASE. These expressions do not need to be boolean; the CASE statement compares pairs of expressions to see if their values are equal. The first WHEN-THEN clause to have an expression value equal to the CASE expression value is executed; the remaining clauses are skipped. The optional ELSE-clause is executed only if no WHEN-clause expression has a value matching the CASE value.

CASE Semantics

```
STRING drink = "Juice";

# CASE statement: if-else version
CASE
  WHEN drink == "Juice" THEN @@calories += 50
  WHEN drink == "Soda"  THEN @@calories += 120
  ...
  ELSE @@calories = 0       # Optional else-clause
END
# Since drink = "Juice", 50 will be added to calories

# CASE statement: switch version
CASE drink
  WHEN "Juice" THEN @@calories += 50
  WHEN "Soda"  THEN @@calories += 120
  ...
  ELSE  @@calories = 0    # Optional else-clause
END
# Since drink = "Juice", 50 will be added to calories
```

Example 1. CASE as IF-ELSE

```
# Display the total number times connected users posted about a certain s
CREATE QUERY userNetworkPosts (vertex<person> seedUser, STRING subjectNam
    SumAccum<INT> @@topicSum = 0;
    OrAccum @visited;
    reachableVertices = {};            # empty vertex set
    visitedVertices (ANY) = {seedUser};  # set that can contain ANY type

    WHILE visitedVertices.size() !=0 DO        # loop terminates when all
        visitedVertices = SELECT s              # s is all neighbors of v
            FROM visitedVertices-(:e)->:s
            WHERE s.@visited == false
            ACCUM s.@visited = true,
                CASE
                    WHEN s.type == "post" and s.subject == subjectName TH
                END;
    END;
    PRINT @@topicSum;
}
```

Example 1 Results for Query userNetworkPosts

```
GSQL > RUN QUERY userNetworkPosts("person1", "Graphs")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@topicSum": 3}]
}
```

Example 2. CASE as switch

```
# tally male and female friends of the starting vertex
CREATE QUERY countGenderOfFriends(vertex<person> seed) FOR GRAPH socialNe
       SumAccum<INT> @@males = 0;
       SumAccum<INT> @@females = 0;
       SumAccum<INT> @@unknown = 0;
       startingVertex = {seed};

       people = SELECT v FROM startingVertex -(friend:e)->:v
                ACCUM CASE v.gender
                  WHEN "Male" THEN @@males += 1
                  WHEN "Female" THEN @@females +=1
                  ELSE @@unknown += 1
                END;
    PRINT @@males, @@females, @@unknown;
}
```

Example 2 Results for Query countGenderOfFriends

```
GSQL > RUN QUERY countGenderOfFriends("person4")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "@@males": 2,
    "@@unknown": 0,
    "@@females": 1
  }]
}
```

Example 3. Multiple CASE statements

```
# give each social network user a social impact score which accumulates
# based on how many friends and posts they have
CREATE QUERY scoreSocialImpact() FOR GRAPH socialNet api("v2") {
       SumAccum<INT> @socialImpact = 0;
       allPeople = {person.*};
       people = SELECT v FROM allPeople:v
              ACCUM CASE WHEN v.outdegree("friend") > 1 THEN v.@socialImp
                    CASE WHEN v.outdegree("friend") > 2 THEN v.@socialImp
                    CASE WHEN v.outdegree("posted") > 1 THEN v.@socialImp
                    CASE WHEN v.outdegree("posted") > 3 THEN v.@socialImp
       #PRINT people.@socialImpact;       // api v1
       PRINT people[people.@socialImpact]; // api v2
}
```

Example 3 Results for Query scoreSocialImpact

```
GSQL > RUN QUERY scoreSocialImpact()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"people": [
    {
      "v_id": "person4",
      "attributes": {"people.@socialImpact": 2},
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {"people.@socialImpact": 1},
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {"people.@socialImpact": 2},
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {"people.@socialImpact": 1},
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {"people.@socialImpact": 2},
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {"people.@socialImpact": 2},
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {"people.@socialImpact": 1},
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {"people.@socialImpact": 3},
```

```
      "v_type": "person"
    }
  ]}]
}
```

Example 4. Nested CASE statements

```
# give each user post a rating based on the subject and how many likes it
CREATE QUERY ratePosts() FOR GRAPH socialNet api("v2") {
  SumAccum<INT> @rating = 0;
  allPeople = {person.*};

  results = SELECT v FROM allPeople -(:e)-> post:v
    ACCUM CASE e.type
      WHEN "posted" THEN
        CASE
          WHEN v.subject == "cats" THEN v.@rating += -1    # -1 if post a
          WHEN v.subject == "Graphs" THEN v.@rating += 2   # +2 if post a
          WHEN v.subject == "tigergraph" THEN v.@rating += 10 # +10 if po
          END
      WHEN "liked" THEN v.@rating += 3                 # +3 each time pos
      END;
  #PRINT results.@rating; // api v1
  PRINT results[results.@rating]; // api v2
}
```

Example 4 Results for Query ratePosts

```
GSQL > RUN QUERY ratePosts()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"results": [
    {
      "v_id": "0",
      "attributes": {"results.@rating": 11},
      "v_type": "post"
    },
    {
      "v_id": "10",
      "attributes": {"results.@rating": 2},
      "v_type": "post"
    },
    {
      "v_id": "2",
      "attributes": {"results.@rating": 0},
      "v_type": "post"
    },
    {
      "v_id": "4",
      "attributes": {"results.@rating": 6},
      "v_type": "post"
    },
    {
      "v_id": "9",
      "attributes": {"results.@rating": -1},
      "v_type": "post"
    },
    {
      "v_id": "3",
      "attributes": {"results.@rating": 2},
      "v_type": "post"
    },
    {
      "v_id": "5",
      "attributes": {"results.@rating": 10},
      "v_type": "post"
    },
    {
      "v_id": "7",
      "attributes": {"results.@rating": 2},
```

```
        "v_type": "post"
      },
      {
        "v_id": "1",
        "attributes": {"results.@rating": 10},
        "v_type": "post"
      },
      {
        "v_id": "11",
        "attributes": {"results.@rating": -1},
        "v_type": "post"
      },
      {
        "v_id": "8",
        "attributes": {"results.@rating": 2},
        "v_type": "post"
      },
      {
        "v_id": "6",
        "attributes": {"results.@rating": 13},
        "v_type": "post"
      }
    ]}]
  }
```


WHILE Statement

The WHILE statement provides unbounded iteration over a block of statements. WHILE statements can be used as query-body statements or DML-sub-statements. (See the note about differences in block syntax .)

WHILE syntax

```
queryBodyWhileStmt := WHILE condition [LIMIT simpleSize] DO queryBodyStmt
dmlSubWhileStmt :=    WHILE condition [LIMIT simpleSize] DO dmlSubStmtLis
simpleSize := integer | varName | paramName
```

The WHILE statement iterates over its body (*queryBodyStmts* or *dmlSubStmtList*) until the *condition* evaluates to false or until the iteration limit is met. A *condition* is any expression that evaluates to a boolean. The condition is evaluated before each iteration. `CONTINUE` statements can be used to change the control flow within the while block. `BREAK` statements can be used to exit the while loop.

A WHILE statement may have an optional LIMIT clause. LIMIT clauses has a constant positive integer value or integer variable to constrain the maximum number of loop iterations. The example below demonstrates how the LIMIT behaves.

> ⓘ If a limit value is not specified, it is possible for a WHILE loop to iterate infinitely. It is the responsibility of the query author to design the condition logic so that it is guaranteed to eventually be true (or to set a limit).

WHILE LIMIT semantics

```
# These three WHILE statements behave the same.  Each terminates when
# (v.size == 0) or after 5 iterations of the loop.
WHILE v.size() !=0 LIMIT 5 DO
    # Some statements
END;

INT iter = 0;
WHILE (v.size() !=0) AND (iter < 5) DO
    # Some statements
    iter = iter + 1;
END;

INT iter = 0;
WHILE v.size() !=0 DO
    IF iter == 5 THEN  BREAK;  END;
    # Some statements
    iter = iter + 1;
END;
```

Below are a number of examples that demonstrate the use of WHILE statements.

Example 1. Simple WHILE loop

```
# find all vertices which are reachable from a starting seed vertex (i.e.
CREATE QUERY reachable(vertex<person> seed) FOR GRAPH workNet
{
    OrAccum @visited;
    reachableVertices = {};        # empty vertex set
    visitedVertices (ANY) = {seed};  # set that can contain ANY type of v

    WHILE visitedVertices.size() !=0 DO        # loop terminates when all
        visitedVertices = SELECT s              # s is all neighbors of v
                FROM visitedVertices-(:e)->:s
                WHERE s.@visited == false
                POST-ACCUM s.@visited = true;
        reachableVertices = reachableVertices UNION visitedVertices;
    END;
    PRINT reachableVertices;
}
```

reachable Results

```
GSQL > RUN QUERY reachable("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"reachableVertices": [
    {
      "v_id": "person3",
      "attributes": {
        "interestList": ["teaching"],
        "skillSet": [ 6, 1, 4 ],
        "skillList": [ 4, 1, 6 ],
        "locationId": "jp",
        "interestSet": ["teaching"],
        "@visited": true,
        "id": "person3"
      },
      "v_type": "person"
    },
    {
      "v_id": "person9",
      "attributes": {
        "interestList": [ "financial", "teaching" ],
        "skillSet": [ 2, 7, 4 ],
        "skillList": [ 4, 7, 2 ],
        "locationId": "us",
        "interestSet": [ "teaching", "financial" ],
        "@visited": true,
        "id": "person9"
      },
      "v_type": "person"
    },
    {
      "v_id": "person4",
      "attributes": {
        "interestList": ["football"],
        "skillSet": [ 10, 1, 4 ],
        "skillList": [ 4, 1, 10 ],
        "locationId": "us",
        "interestSet": ["football"],
        "@visited": true,
        "id": "person4"
      },
      "v_type": "person"
```

```
    },
    {
      "v_id": "person7",
      "attributes": {
        "interestList": [ "art", "sport" ],
        "skillSet": [ 6, 8 ],
        "skillList": [ 8, 6 ],
        "locationId": "us",
        "interestSet": [ "sport", "art" ],
        "@visited": true,
        "id": "person7"
      },
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {
        "interestList": [ "management", "financial" ],
        "skillSet": [ 3, 2, 1 ],
        "skillList": [ 1, 2, 3 ],
        "locationId": "us",
        "interestSet": [ "financial", "management" ],
        "@visited": true,
        "id": "person1"
      },
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {
        "interestList": [ "sport", "financial", "engineering" ],
        "skillSet": [ 5, 2, 8 ],
        "skillList": [ 8, 2, 5 ],
        "locationId": "can",
        "interestSet": [ "engineering", "financial", "sport" ],
        "@visited": true,
        "id": "person5"
      },
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {
        "interestList": [ "music", "art" ],
        "skillSet": [ 10, 7 ],
        "skillList": [ 7, 10 ],
        "locationId": "jp",
        "interestSet": [ "art", "music" ],
```

```
        "@visited": true,
        "id": "person6"
      },
      "v_type": "person"
    },
    {
      "v_id": "person2",
      "attributes": {
        "interestList": ["engineering"],
        "skillSet": [ 6, 5, 3, 2 ],
        "skillList": [ 2, 3, 5, 6 ],
        "locationId": "chn",
        "interestSet": ["engineering"],
        "@visited": true,
        "id": "person2"
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {
        "interestList": ["management"],
        "skillSet": [ 2, 5, 1 ],
        "skillList": [ 1, 5, 2 ],
        "locationId": "chn",
        "interestSet": ["management"],
```


Example 2. WHILE loop using a LIMIT

```
# find all vertices which are reachable within two hops from a starting s
CREATE QUERY reachableWithinTwo(vertex<person> seed) FOR GRAPH workNet
{
    OrAccum @visited;
    reachableVertices = {};        # empty vertex set
    visitedVertices (ANY) = {seed};  # set that can contain ANY type of v

    WHILE visitedVertices.size() !=0 LIMIT 2 DO # loop terminates when al
        visitedVertices = SELECT s               # s is all neighbors of
                FROM visitedVertices-(:e)->:s
                WHERE s.@visited == false
                POST-ACCUM s.@visited = true;
        reachableVertices = reachableVertices UNION visitedVertices;
    END;
    PRINT reachableVertices;
}
```


```
        "@visited": true,
                   ny2"
```


reachableWithinTwo Results


```
      },
      "v_type": "company"
```

```
GSQL > RUN QUERY reachableWithinTwo("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"reachableVertices": [
    {
      "v_id": "person4",
      "attributes": {
        "interestList": ["football"],
        "skillSet": [ 10, 1, 4 ],
        "skillList": [ 4, 1, 10 ],
        "locationId": "us",
        "interestSet": ["football"],
        "@visited": true,
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "person3",
      "attributes": {
        "interestList": ["teaching"],
        "skillSet": [ 6, 1, 4 ],
        "skillList": [ 4, 1, 6 ],
        "locationId": "jp",
        "interestSet": ["teaching"],
        "@visited": true,
        "id": "person3"
      },
      "v_type": "person"
    },
    {
      "v_id": "person9",
      "attributes": {
        "interestList": [ "financial", "teaching" ],
        "skillSet": [ 2, 7, 4 ],
        "skillList": [ 4, 7, 2 ],
        "locationId": "us",
        "interestSet": [ "teaching", "financial" ],
        "@visited": true,
        "id": "person9"
      },
      "v_type": "person"
```

```
    },
    {
      "v_id": "person5",
      "attributes": {
        "interestList": [ "sport", "financial", "engineering" ],
        "skillSet": [ 5, 2, 8 ],
        "skillList": [ 8, 2, 5 ],
        "locationId": "can",
        "interestSet": [ "engineering", "financial", "sport" ],
        "@visited": true,
        "id": "person5"
      },
      "v_type": "person"
    },
    {
      "v_id": "person6",
      "attributes": {
        "interestList": [ "music", "art" ],
        "skillSet": [ 10, 7 ],
        "skillList": [ 7, 10 ],
        "locationId": "jp",
        "interestSet": [ "art", "music" ],
        "@visited": true,
        "id": "person6"
      },
      "v_type": "person"
    },
    {
      "v_id": "person10",
      "attributes": {
        "interestList": [ "football", "sport" ],
        "skillSet": [3],
        "skillList": [3],
        "locationId": "us",
        "interestSet": [ "sport", "football" ],
        "@visited": true,
        "id": "person10"
      },
      "v_type": "person"
    },
    {
      "v_id": "person8",
      "attributes": {
        "interestList": ["management"],
        "skillSet": [ 2, 5, 1 ],
        "skillList": [ 1, 5, 2 ],
        "locationId": "chn",
        "interestSet": ["management"],
```

```
          "@visited": true,
          "id": "person8"
        },
        "v_type": "person"
      },
      {
        "v_id": "company1",
        "attributes": {
          "country": "us",
          "@visited": true,
          "id": "company1"
        },
        "v_type": "company"
      },
      {
        "v_id": "person2",
        "attributes": {
          "interestList": ["engineering"],
          "skillSet": [ 6, 5, 3, 2 ],
          "skillList": [ 2, 3, 5, 6 ],
          "locationId": "chn",
          "interestSet": ["engineering"],
          "@visited": true,
          "id": "person2"
        },
        "v_type": "person"
```


FOREACH Statement

FOREACH syntax

```
queryBodyForEachStmt := FOREACH forEachControl DO queryBodyStmts END
dmlSubForEachStmt :=    FOREACH forEachControl DO dmlSubStmtList END

forEachControl := ( iterationVar | "(" keyVar ("," valueVar)+ ")") (IN |
                | iterationVar IN RANGE "[" expr "," expr"]" ["." STEP("
iterationVar := name
keyVar := name
valueVar := name
```


```
      {
        "v_id": "person7",
        "attributes": {
          "interestList": [ "art", "sport" ],
          "skillSet": [ 6, 8 ],
          "skillList": [ 8, 6 ],
          "locationId": "us",
          "interestSet": [ "sport", "art" ],
          "@visited": true,
          "id": "person7"
        },
        "v_type": "person"
```

```
    },
    {
      "v_id": "person1",
      "attributes": {
        "interestList": [ "management", "financial" ],
        "skillSet": [ 3, 2, 1 ],
        "skillList": [ 1, 2, 3 ],
        "locationId": "us",
        "interestSet": [ "financial", "management" ],
        "@visited": true,
        "id": "person1"
      },
      "v_type": "person"
    }
  ]}]
}
```

- A query-body-level FOREACH cannot iterate over a set or bag of constants. For example, FOREACH i in (1,2,3) is not supported. However, DML-sub FOREACH does support this.

FOREACH ... IN RANGE

The FOREACH statement has an optional RANGE clause RANGE[expr, expr], which can be used to define the iteration collection. Optionally, the range may specify a step size:
RANGE[expr, expr].STEP(expr)

Each expr must evaluate to an integer. Any of the integers may be negative, but the step expr may not be 0.

The clause RANGE[a,b].STEP(c) produces the sequence of integers from a to b, inclusive, with step size c. That is,
(a, a+c, a+2*c, a+3*c, ... a+k*c), where k = the largest integer such that |k*c| ≤ |b-a|.

If the .STEP method is not given, then the step size c = 1.

Nested FOREACH IN RANGE with MapAccum

```
CREATE QUERY foreachRangeEx() FOR GRAPH socialNet {
  ListAccum<INT> @@t;
  Start = {person.*};
  FOREACH i IN RANGE[0, 2] DO
    @@t += i;
    L = SELECT Start
        FROM Start
        WHERE Start.id == "person1"
        ACCUM
          FOREACH j IN RANGE[0, i] DO
            @@t += j
          END
        ;
  END;
  PRINT @@t;
}
```

Results for Query foreachRangeEx

```
GSQL > RUN QUERY foreachRangeEx()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@t": [ 0, 0, 1, 0, 1, 2, 0, 1, 2 ]}]
}
```

FOREACH IN RANGE with step

```
CREATE QUERY foreachRangeStep(INT a, INT b, INT c) FOR GRAPH minimalNet {
  ListAccum<INT> @@t;
  FOREACH i IN RANGE[a,b].step(c) DO
    @@t += i;
  END;
  PRINT @@t;
}
```

The step value can be positive for an ascending range or negative for a descending range. If the step has the wrong polarity, then the loop has zero iterations; that is, the exit condition is already satisfied.

foreachRangeStep.json Results

```
GSQL > RUN QUERY foreachRangeStep(100,0,-9)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@t": [
    100,
    91,
    82,
    73,
    64,
    55,
    46,
    37,
    28,
    19,
    10,
    1
  ]}]
}
GSQL > RUN QUERY foreachRangeStep(-100,100,-9)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@t": []}]
}
```

Query-body-level FOREACH Examples

Example 1 - FOREACH with ListAccum

```
# Count the number of companies whose country matches the provided string
CREATE QUERY companyCount(STRING countryName) FOR GRAPH workNet {
  ListAccum<STRING> @@companyList;
  INT countryCount;
  start = {ANY};                       # start will have a set of all ve

  s = SELECT v FROM start:v            # get all vertices
      WHERE v.type == "company"        # that have a type of "company"
      ACCUM @@companyList += v.country; # append the country attribute fr

  # Iterate the ListAccum and compare each element to the countryName par
  FOREACH item in @@companyList DO
    IF item == countryName THEN
       countryCount = countryCount + 1;
    END;
  END;
  PRINT countryCount;
}
```

companyCount Results

```
GSQL > RUN QUERY companyCount("us")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"countryCount": 2}]
}
GSQL > RUN QUERY companyCount("can")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"countryCount": 1}]
}
```

Example 2 - FOREACH with a seed set

```
#Find all company person who live in a given country
CREATE QUERY employeesByCompany(STRING country) FOR GRAPH workNet {
  ListAccum<VERTEX<company>> @@companyList;
  start = {ANY};

  # Build a list of all company vertices
  #  (these are vertex IDs only)
  s = SELECT v FROM start:v
      WHERE v.type == "company"
      ACCUM @@companyList += v;

  # Use the vertex IDs as Seeds for vertex sets
  FOREACH item IN @@companyList DO
    companyItem = {item};
    employees = SELECT t FROM companyItem -(worksFor)-> :t
                WHERE (t.locationId == country);
    PRINT employees;
  END;
}
```

employeesByCompany Results

```
GSQL > RUN QUERY employeesByCompany("us")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [ {"employees": []},
    {"employees": []},
    {"employees": [
      {
        "v_id": "person9",
        "attributes": {
          "interestList": [
            "financial",
            "teaching"
          ],
          "skillSet": [ 2, 7, 4 ],
          "skillList": [ 4, 7, 2 ],
          "locationId": "us",
          "interestSet": [ "teaching", "financial" ],
          "id": "person9"
        },
        "v_type": "person"
      },
      {
        "v_id": "person10",
        "attributes": {
          "interestList": [ "football", "sport" ],
          "skillSet": [3],
          "skillList": [3],
          "locationId": "us",
          "interestSet": [ "sport", "football" ],
          "id": "person10"
        },
        "v_type": "person"
      },
      {
        "v_id": "person7",
        "attributes": {
          "interestList": [ "art", "sport" ],
          "skillSet": [ 6, 8 ],
          "skillList": [ 8, 6 ],
          "locationId": "us",
          "interestSet": [ "sport", "art" ],
          "id": "person7"
```

```
      },
      "v_type": "person"
    }
  ]},
  {"employees": [
    {
      "v_id": "person4",
      "attributes": {
        "interestList": ["football"],
        "skillSet": [ 10, 1, 4 ],
        "skillList": [ 4, 1, 10 ],
        "locationId": "us",
        "interestSet": ["football"],
        "id": "person4"
      },
      "v_type": "person"
    },
    {
      "v_id": "person9",
      "attributes": {
        "interestList": [ "financial", "teaching" ],
        "skillSet": [ 2, 7, 4 ],
        "skillList": [ 4, 7, 2 ],
        "locationId": "us",
        "interestSet": [ "teaching", "financial" ],
        "id": "person9"
      },
      "v_type": "person"
    },
    {
      "v_id": "person7",
      "attributes": {
        "interestList": [ "art", "sport" ],
        "skillSet": [ 6, 8 ],
        "skillList": [ 8, 6 ],
        "locationId": "us",
        "interestSet": [ "sport", "art" ],
        "id": "person7"
```

Example 3 - Nested FOREACH with MapAccum

```
      "v_type": "person"
    },
    {
      "v_id": "person1",
      "attributes": {
        "interestList": [ "management", "financial" ],
        "skillSet": [ 3, 2, 1 ],
        "skillList": [ 1, 2, 3 ],
        "locationId": "us",
```

```
# Count the number of employees from a given country and list their ids
CREATE QUERY employeeByCountry(STRING countryName) FOR GRAPH workNet {
  MapAccum <STRING, ListAccum<STRING>> @@employees;

  # start will have a set of all person type vertices
  start = {person.*};

  # Build a map using person locationId as a key and a list of strings to
  s = SELECT v FROM start:v
      ACCUM @@employees += (v.locationId -> v.id);

  # Iterate the map using (key,value) pairs
  FOREACH (key,val) in @@employees DO
    IF key == countryName THEN
      PRINT val.size();

      # Nested foreach to iterate over the list of person ids
      FOREACH employee in val DO
        PRINT employee;
      END;

      # MapAccum keys are unique so we can BREAK out of the loop
      BREAK;
    END;
  END;
}
```

employeeByCountry Results


```
          "skillList": [ 1, 2, 3 ],
                    d": "us",
           interestSet": [ "financial", "management" ],
          "id": "person1"
        },
        "v_type": "person"
      }
    ]}
  ]
}
```

```
GSQL > RUN QUERY employeeByCountry("us")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"val.size()": 5},
    {"employee": "person4"},
    {"employee": "person10"},
    {"employee": "person7"},
    {"employee": "person1"},
    {"employee": "person9"}
  ]
}
```

DML-sub FOREACH Examples

ACCUM FOREACH

```
# Show post topics liked by users and show total likes per topic
CREATE QUERY topicLikes() FOR GRAPH socialNet {
 SetAccum<STRING> @@personPosts;
 SumAccum<INT> @postLikes;
 MapAccum<STRING,INT> @@likesByTopic;


  start = {person.*};

  # Find all user posts and generate a set of post topics
  # (set has no duplicates)
  posts = SELECT g FROM start - (posted) -> :g
          ACCUM @@personPosts += g.subject;

  # Use set of topics to increment how many times a specfic
  #  post is liked by other users
  likedPosts = SELECT f FROM start - (liked) -> :f
               ACCUM FOREACH x in @@personPosts DO
                         CASE WHEN (f.subject == x) THEN
                           f.@postLikes += 1
                         END
                     END
               # Aggregate all liked totals by topic
               POST-ACCUM @@likesByTopic += (f.subject -> f.@postLikes);

  # Display the number of likes per topic
  PRINT @@likesByTopic;
}
```

Results for Query topicLikes

```
GSQL > RUN QUERY topicLikes()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@likesByTopic": {
    "cats": 3,
    "coffee": 2,
    "Graphs": 3,
    "tigergraph": 1
  }}]
}
```

Example 1 - POST-ACCUM FOREACH

```
#Show a summary of the number of friends all persons have by gender
CREATE QUERY friendGender() FOR GRAPH socialNet {
  ListAccum<STRING> @friendGender;
  SumAccum<INT> @@maleGenderCount;
  SumAccum<INT> @@femaleGenderCount;

  start = {person.*};

  # Record a list showing each friend's gender
  socialMembers = SELECT s from start:s -(friend)-> :g
              ACCUM s.@friendGender += (g.gender)

              # Loop over each list of genders and total them
              POST-ACCUM FOREACH x in s.@friendGender DO
                           CASE WHEN (x == "Male") THEN
                             @@maleGenderCount += 1
                           ELSE
                             @@femaleGenderCount += 1
                           END
                         END;

  PRINT @@maleGenderCount;
  PRINT @@femaleGenderCount;
}
```

Results for Query friendGender

```
GSQL > RUN QUERY friendGender()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@maleGenderCount": 11},
    {"@@femaleGenderCount": 7}
  ]
}
```

CONTINUE and BREAK Statements

The CONTINUE and BREAK statements can only be used within a block of a WHILE or FOREACH statement. The CONTINUE statement branches control flow to the end of the loop, skipping any remaining statements in the current iteration, and proceeding to the next iteration. That is, everything in the loop block after the CONTINUE statement will be skipped, and then the loop will continue as normal.

The BREAK statement branches control flow out of the loop, i.e., it will exit the loop and stop iteration.

Below are a number of examples that demonstrate the use of BREAK and CONTINUE.

Continue and Break Semantics

```
# While with a continue
INT i = 0;
INT nCount = 0;
WHILE i < 10 DO
  i = i + 1;
  IF (i % 2 == 0) { CONTINUE; }
  nCount = nCount + 1;
END;
# i is 10, nCount is 5 (skips the increment for every even i).

# While with a break
i = 0;
WHILE i < 10 DO
  IF (i == 5) { BREAK; }  # When i is 5 the loop is exited
  i = i + 1;
END;
# i is now 5
```

Example 1. Break

```
# find posts of a given person, and post of friends of that person, frien
# until a post about cats is found. The number of friend-hops to reach is
CREATE QUERY findDegreeOfCats(vertex<person> seed) FOR GRAPH socialNet
{
    SumAccum<INT> @@degree = 0;
    OrAccum @@foundCatPost = false;
    OrAccum @visited = false;

    friends (ANY) = {seed};
    WHILE @@foundCatPost != true AND friends.size() > 0 DO
          posts = SELECT v FROM friends-(posted:e)->:v
                  ACCUM CASE WHEN v.subject == "cats" THEN @@foundCatPost

          IF @@foundCatPost THEN
            BREAK;
          END;

          friends = SELECT v FROM friends-(friend:e)->:v
                  WHERE v.@visited == false
                  ACCUM v.@visited = true;
          @@degree += 1;
    END;
    PRINT @@degree;
}
```

Results of Query findDegreeOfCats

```
GSQL > RUN QUERY findDegreeOfCats("person2")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@degree": 2}]
}
GSQL > RUN QUERY findDegreeOfCats("person4")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@degree": 0}]
}
```

Example 2. findEnoughFriends.gsql: While loop using continue statement

```
# find all 3-hop friends of a starting vertex. count coworkers as friends
# if there are not enough friends
CREATE QUERY findEnoughFriends(vertex<person> seed) FOR GRAPH friendNet
{
    SumAccum<INT> @@distance = 0;   # keep track of the distance from the
    OrAccum @visited = false;
    visitedVertices = {seed};
    WHILE true LIMIT 3 DO
        @@distance += 1;
        # traverse from visitedVertices to its friends
        friends = SELECT v
            FROM visitedVertices -(friend:e)-> :v
            WHERE v.@visited == false
            POST-ACCUM v.@visited = true;
        PRINT @@distance, friends;

        # if number of friends at this level is sufficient, finish this i
        IF visitedVertices.size() >= 2 THEN
            visitedVertices = friends;
            CONTINUE;
        END;
        # if fewer than 4 friends, add in coworkers
        coworkers = SELECT v
            FROM visitedVertices -(coworker:e)-> :v
            WHERE v.@visited == false
            POST-ACCUM v.@visited = true;
        visitedVertices = friends UNION coworkers;
        PRINT @@distance, coworkers;
    END;
}
```

findEnoughFriends.json Example 2 Results

```
GSQL > RUN QUERY findEnoughFriends("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "@@distance": 1,
      "friends": [
        {
          "v_id": "person4",
          "attributes": {
            "@visited": true,
            "id": "person4"
          },
          "v_type": "person"
        },
        {
          "v_id": "person2",
          "attributes": {
            "@visited": true,
            "id": "person2"
          },
          "v_type": "person"
        },
        {
          "v_id": "person3",
          "attributes": {
            "@visited": true,
            "id": "person3"
          },
          "v_type": "person"
        }
      ]
    },
    {
      "coworkers": [
        {
          "v_id": "person5",
          "attributes": {
            "@visited": true,
            "id": "person5"
          },
          "v_type": "person"
```

```
        },
        {
          "v_id": "person6",
          "attributes": {
            "@visited": true,
            "id": "person6"
          },
          "v_type": "person"
        }
      ],
      "@@distance": 1
    },
    {
      "@@distance": 2,
      "friends": [
        {
          "v_id": "person9",
          "attributes": {
            "@visited": true,
            "id": "person9"
          },
          "v_type": "person"
        },
        {
          "v_id": "person1",
```


Example 3. While loop using break statement


```
            "@visited": true,
            "id": "person1"
          },
          "v_type": "person"
        },
        {
          "v_id": "person8",
          "attributes": {
            "@visited": true,
            "id": "person8"
          },
          "v_type": "person"
        }
      ]
    },
    {
      "@@distance": 3,
      "friends": [
        {
          "v_id": "person12",
          "attributes": {
            "@visited": true,
```

```
# find at least the top-k companies closest to a given seed vertex, if th
CREATE QUERY topkCompanies(vertex<person> seed, INT k) FOR GRAPH workNet
{
    SetAccum<vertex<company>> @@companyList;
    OrAccum @visited = false;
    visitedVertices (ANY) = {seed};
    WHILE true DO
        visitedVertices = SELECT v                  # traverse from x to
                FROM visitedVertices -(:e)-> :v
                WHERE v.@visited == false
                ACCUM CASE
                    WHEN (v.type == "company") THEN # count the number of
                        @@companyList += v
                    END
                POST-ACCUM v.@visited += true;      # mark vertices as vi

        # exit loop when at least k companies have been counted
        IF @@companyList.size() >= k OR visitedVertices.size() == 0 THEN
           BREAK;
        END;
    END;
    PRINT @@companyList;
}
```

Example 3. topkCompanies Results

```
GSQL > RUN QUERY topkCompanies("person1", 2)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@companyList": [
    "company2",
    "company1"
  ]}]
}
GSQL > RUN QUERY topkCompanies("person2", 3)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@companyList": [
    "company3",
    "company2",
    "company1"
  ]}]
}
```

Example 4 - Usage of CONTINUE in FOREACH

```
#List out all companies from a given country
CREATE QUERY companyByCountry(STRING countryName) FOR GRAPH workNet {
  MapAccum <STRING, ListAccum<STRING>> @@companies;
  start = {company.*};                    # start will have a set of all c

  #Build a map using company country as a key and a list of strings to ho
  s = SELECT v FROM start:v
      ACCUM @@companies += (v.country -> v.id);

  #Iterate the map using (key,value) pairs
  FOREACH (key,val) IN @@companies DO
    IF key != countryName THEN
      CONTINUE;
    END;

    PRINT val.size();

    #Nested foreach to iterate over the list of company ids
    FOREACH comp IN val DO
      PRINT comp;
    END;
  END;
}
```

companyByCountry Results

```
GSQL > RUN QUERY companyByCountry("us")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"val.size()": 2},
    {"comp": "company1"},
    {"comp": "company4"}
  ]
}
```

Example 5 - Usage of BREAK in FOREACH

```
#List all the persons located in the specified country
CREATE QUERY employmentByCountry(STRING countryName) FOR GRAPH workNet {
  MapAccum < STRING, ListAccum<STRING> > @@employees;
  start = {person.*};                    # start will have a set of all pe

  #Build a map using person locationId as a key and a list of strings to
  s = SELECT v FROM start:v
      ACCUM @@employees += (v.locationId -> v.id);

  #Iterate the map using (key,value) pairs
  FOREACH (key,val) IN @@employees DO
    IF key == countryName THEN
      PRINT val.size();

      #Nested foreach to iterate over the list of person ids
      FOREACH employee IN val DO
        PRINT employee;
      END;

      BREAK;
    END;
  END;
}
```

employmentByCountry Result

```
GSQL > RUN QUERY employmentByCountry("us")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"val.size()": 5},
    {"employee": "person1"},
    {"employee": "person4"},
    {"employee": "person7"},
    {"employee": "person9"},
    {"employee": "person10"}
  ]
}
```

Data Modification Statements

The GSQL language provides full support for vertex and edge insertion, deletion, and attribute update is provided. Therefore, the language is more than just a "query" language.

Each query is considered one transaction. Therefore, modifications to the graph data do not take effect until the entire query is completed (committed). Accordingly, any modification statement does not affect any other statements inside the same query.

Query-body DELETE Statement

The query-body DELETE statement deletes a given set of edges or vertices. This statement can only be used as a query-body statement. (Deletion at the DML-sub level is served by the DML-sub DELETE statement, described next.

> The GSQL DELETE operation is a cascading deletion. If a vertex is deleted, then all of the edges which connect to it are automatically deleted as well.

EBNF

```
queryBodyDeleteStmt := DELETE alias FROM pattern [whereClause]
pattern := (vertexPattern | edgePattern)
```

The vertexPattern and edgePattern terms in the FROM clause follow the same rules as those in the FROM clause in a SELECT statement. The WHERE clause can filter the items in the vertexPattern or edgePattern. Below are two examples, one for deleting vertices and one for deleting edges.

DELETE statement example

```
# Delete all "person" vertices with location equal to "us"
CREATE QUERY deleteEx() FOR GRAPH workNet {
  S = {person.*};
  DELETE s FROM S:s
    WHERE s.locationId == "us";
}
```

DELETE statement example 2

```
# Delete all "worksFor" edges where the person's location is "us"
CREATE QUERY deleteEx2() FOR GRAPH workNet {
  S = {person.*};
  DELETE e FROM S:s -(worksFor:e)-> company:t
    WHERE s.locationId == "us";
}
```

The following query can be used to observe the effect of the delete statements. This query counts the person vertices who work in the US ("us") and the worksFor edges for persons in the US. When the initial workNet test data loaded, there are 5 persons and 9 worksFor edges for locationId = "us". If query deleteEx2 is run, the worksAtUS query will then find the 5 persons but 0 worksFor edges. Next, if the deleteEx query is run, the worksAtUS query will then find 0 persons and 0 worksFor edges.

Query to check the results of deleteEx and deleteEx2

```
CREATE QUERY countAtLocation(STRING loc) FOR GRAPH workNet {
  SetAccum<EDGE> @@selEdge;
  Start = {person.*};

  SV = SELECT s FROM Start:s
    WHERE s.locationId == loc;
  PRINT SV.size() AS numVertices;

  SE = SELECT s FROM Start:s -(worksFor:e)-> company:t
    WHERE s.locationId == loc
    ACCUM @@selEdge += e;
  PRINT @@selEdge.size() AS numEdges;
}
```

For example, the following sequence of countAtLocation, deleteEx2, and deleteEx queries deleteEx.run

```
RUN QUERY countAtLocation("us")
RUN QUERY deleteEx2()
RUN QUERY countAtLocation("us")
RUN QUERY deleteEx()
RUN QUERY countAtLocation("us")
```

will produce the following result:

Results from DeleteEx Example

```
# Before any deletions
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"numVertices": 5},
    {"numEdges": 9}
  ]
}
# Delete edges
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": []
}
# After deleting edges
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"numVertices": 5},
    {"numEdges": 0}
  ]
}
# Delete vertices
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
```

```
  "results": []
}
# After deleting vertices
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"numVertices": 0},
    {"numEdges": 0}
  ]
}
```

DML-sub DELETE Statement

DML-sub DELETE is a DML-substatement which deletes one vertex or edge each time it is called. (Deletion at the query-body level is served by the Query-body DELETE statement described above.) In practice, this statement resides within the body of a SELECT...ACCUM/POST-ACCUM clause, so it is called once for each member of a selected vertex set or edge set.

The GSQL DELETE operation is a cascading deletion. If a vertex is deleted, then all of the edges which connect to it are automatically deleted as well.

The ACCUM clause iterates over an edge set, which can encounter the same vertex multiple times. If you wish to delete a vertex, it is best practice to place the DML-sub DELETE statement in the POST-ACCUM clause rather than in the ACCUM clause.

EBNF

```
dmlSubDeleteStmt := DELETE "(" alias ")"
```

The following example uses and modifies the graph data for socialNet.

DELETE within ACCUM vs. POST-ACCUM

```
# Remove any post vertices posted by the given user
CREATE QUERY deletePosts(vertex<person> seed) FOR GRAPH socialNet {
    start = {seed};

    # Best practice is to delete a vertex in a POST-ACCUM, which only
    # occurs once for each vertex v, guaranteeing that a vertex is not
    # deleted more than once
    postAccumDeletedPosts = SELECT v FROM start -(posted:e)-> post:v
             POST-ACCUM DELETE (v);

    # Possible, but not recommended as the DML-sub DELETE statement occur
    # once for each edge of the vertex v
    accumDeletedPosts = SELECT v FROM start -(posted:e)-> post:v
             ACCUM DELETE (v);
}

# Need a separate query to display the results, because deletions don't t
CREATE QUERY selectUserPosts(vertex<person> seed) FOR GRAPH socialNet {
    start = {seed};

    userPosts = SELECT v FROM start -(posted:e)-> post:v;
    PRINT userPosts;
}
```

For example, the following sequence of selectUserPosts and deletePosts queries

deletePosts.run

```
RUN QUERY selectUserPosts("person3")
RUN QUERY deletePosts("person3")
RUN QUERY selectUserPosts("person3")
```

will produce the following result:

Results from DeletePosts Example

```
# Before the deletion
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"userPosts": [{
    "v_id": "2",
    "attributes": {
      "postTime": "2011-02-03 01:02:42",
      "subject": "query languages"
    },
    "v_type": "post"
  }]}]
}
# Deletion; no output results requested at this point
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": []
}
# After the deletion
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"userPosts": []}]
}
```

`INSERT INTO` Statement

The `INSERT INTO` statement adds edges or vertices to the graph. When the ID value(s) for the inserted vertex/edge match those of an existing vertex/edge, then the new values will overwrite the old values. To insert an edge, its endpoint vertices must already exist, either before running the query or inserted earlier in that query. The `INSERT INTO` statement can be used as a query-body-level statement or a DML-sub statement.

Like any other data modification in a query, the insertion does not take effect until the entire query is completed.

EBNF

```
insertStmt := insertVertexStmt | insertEdgeStmt
insertVertexStmt := INSERT INTO (vertexType | name)
                    ["(" PRIMARY_ID ["," attrName]* ")"]
                    VALUES "(" ( "_" | expr ) ["," ("_" | expr)]*] ")"

insertEdgeStmt   := INSERT INTO (edgeType | EDGE name)
                    ["(" FROM "," TO ["," attrName]* ")"]
                    VALUES "(" ( "_" | expr ) [vertexType]
                    ["," ( "_" | expr ) [vertexType] ["," ("_" | expr)]*] ")
```

Dynamic Query Support
As of TigerGraph 3.0, the vertex or edge type in an `INSERT` statement can either be set statically (`vertexType` or `edgeType`), or it can be written as a string variable (`name`), with the value being set at run time, to make a Dynamic DML query.
`INSERT INTO (vertexType | name) ...`
Note that to INSERT an edge type dynamically, the keyword EDGE is required:
`INSERT INTO (edgeType | EDGE name) ...`

There are two options for specifying the attributes of the vertex or edge type for the values provided:

- Provide a value for the ID(s) and then each attribute, in the canonical order for the vertex or edge type. In this case, it is not necessary to explicitly name the attributes, since it is assumed that every attribute is being referenced, in order.

  INSERT with implicit attribute names

  ```
  INSERT INTO vertex_or_edge_type VALUES (full_list_of_parameter_values)
  ```

- Name the specific attributes to be set, and then provide a corresponding list of values. The attributes can be in any order, with the exception that the IDs must come first. That is, to insert a vertex, the first attribute name must be `PRIMARY_ID`. To insert an edge, the first two attribute names must be `FROM` and `TO`.

  INSERT with explicit attribute names

  ```
  INSERT INTO vertex_type (PRIMARY_ID, specified_attributes)
  VALUES (ID, values_for_specified_attributes)

  INSERT INTO edge_type (FROM, TO, specified_attributes)
  VALUES (
  ```

For each attribute value, provide either an expression *expr* or `_`, which means the default value for that attribute type. The optional *name* which follows the first two (id) values is to specify the source vertex type and target vertex type, if the edge type had been defined with wildcard vertex types.

Query-Body INSERT

The query `insertEx` illustrates query-body level `INSERT` statements: insert new `company` vertices and `worksFor` edges into the `workNet` graph.

INSERT statement

```
CREATE QUERY insertEx(STRING name, STRING name2, STRING name3, STRING com
  # Vertex insertion
    # Adds 2 'company' vertices. One is located in the USA, and a sister
    # company:
    # company(PRIMARY_ID clientId STRING, id STRING, country STRING)
    INSERT INTO company VALUES ( comp, comp, "us" );
    INSERT INTO company (PRIMARY_ID, country) VALUES ( comp + "_jp", "jp"

  # Edge insertion
    # Adds a 'worksFor' edge from person 'name' to the company 'comp', fi
    # values for startYear (0), startMonth (0), and fullTime (false).
    # worksFor:
    # worksFor(FROM person, TO company, startYear INT, startMonth INT, fu
    INSERT INTO worksFor VALUES (name person, comp company, _, _, _);

    # Adds a 'worksFor' edge from person 'name2' to the company 'comp', f
    # values for startMonth (0), but specifying values for startYear and
    INSERT INTO worksFor (FROM, TO, startYear, fullTime) VALUES (name2 pe

    # Adds a 'worksFor' edge from person 'name3' to the company 'comp', f
    # values for startMonth (0), and fullTime (false) but specifying a va
    INSERT INTO worksFor (FROM, TO, startYear) VALUES (name3 person, comp
}
```

The query `whoWorksForCompany` can be used to check the effect of query `insertEx`. Prior to running `insertEx`, running `whoWorksForCompany("gsql")` will find 0 `companies` called `"gsql"` and 0 `worksFor` edges for company `"gsql"`. If we then run the query `insertEx("tic", "tac", "toe", "gsql")`, then `insertEx("gsql")` will find a company called `"gsql"` and another one called `"gsql_jp"`. Moreover, it will find 3 edges, tic, tac, and toe, with different values for the `startMonth`, `startYear`, and fullTime parameters.

Query to check the results of insertEx

```
CREATE QUERY whoWorksForCompany(STRING comp) FOR GRAPH workNet {
  SetAccum<EDGE> @@setEdge;

  Comps = {company.*};
  PRINT Comps[Comps.id];   # output api v2

  Pers = {person.*};
  S = SELECT s
    FROM Pers:s -(worksFor:e)-> :t
    WHERE t.id == comp
    ACCUM @@setEdge += e;
  PRINT @@setEdge;
}
```

DML-sub INSERT

The following example shows a DML-sub level INSERT. Because the statement applies to all companies, several vertices will be inserted.

DML-sub INSERT statement

```
# Add a child company of a given company name. The new child company is i
CREATE QUERY addNewChildCompany(STRING name) FOR GRAPH workNet {
  allCompanies = {company.*};
  X = SELECT s
      FROM allCompanies:s
      WHERE s.id == name
      ACCUM  INSERT INTO company VALUES ( name + "_jp", name + "_jp", "jp
}

# Add separate query to list the companies, before and after the insertio
CREATE QUERY listCompanyNames(STRING countryFilter) FOR GRAPH workNet {
  allCompanies = {company.*};
  C = SELECT s
      FROM allCompanies:s
      WHERE s.country == countryFilter;

  PRINT C.size() AS numCompanies;
  PRINT C;
}
```

Example: Add a child company in Japan to the US-based company `company3`. List all the Japan-based companies before and after the insertion.

addNewChildCompany.run

```
RUN QUERY listCompanyNames("jp")
RUN QUERY addNewChildCompany("company4")
RUN QUERY listCompanyNames("jp")
```

Results from addNewChildCompany Example

```
# Before insertion
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"numCompanies": 1},
    {"C": [{
      "v_id": "company3",
      "attributes": {
        "country": "jp",
        "id": "company3"
      },
      "v_type": "company"
    }]}
  ]
}
# insert company "company4_jp"
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": []
}
# after insertion
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"numCompanies": 2},
    {"C": [
      {
        "v_id": "company3",
        "attributes": {
          "country": "jp",
```

```
          "id": "company3"
        },
        "v_type": "company"
      },
      {
        "v_id": "company4_jp",
        "attributes": {
          "country": "jp",
          "id": "company4_jp"
        },
        "v_type": "company"
      }
    ]}
  ]
}
```


UPDATE Statement

The `UPDATE` statement updates the attributes of vertices or edges.

EBNF

```
updateStmt := UPDATE alias FROM pattern SET dmlSubStmtList [whereClause]
pattern := (vertexPattern | edgePattern)
```

`UPDATE` statements are not supported in syntax V2. To update vertex or edge attributes in syntax v2, use assignment statements in the `ACCUM` and `POST-ACCUM` clauses of a `SELECT` statement. For more details, see Data Modification in Syntax V2.

The set of vertices or edges to update is described in the `FROM` clause, following the same rules as the `FROM` clause in a `SELECT` statement. In the `SET` clause, the `dmlSubStmtList` contains assignment statements to update the attributes of a vertex or edge. Both simple base type attributes and collection type attributes can be updated. These assignment statements use the vertex or edge aliases declared in the `FROM` clause. The optional `WHERE` clause supports boolean conditions to filter the items in the vertex set or edge set.

UPDATE statement example

```
# Change all "person" vertices with location equal to "us" to "USA"
CREATE QUERY updateEx() FOR GRAPH workNet  {
  S = {person.*};

  UPDATE s FROM S:s
  SET s.locationId = "USA",  # simple base type attribute
      s.skillList = [1,2,3]  # collection-type attribute
  WHERE s.locationId == "us";

  # The update cannot become effective within this query, so PRINT S stil
  PRINT S;
}
```

The `UPDATE` statement can only be used as a query-body-level statement. However, DML-sub level updates are still possible by using other statement types. A vertex attribute's value can be updated within the `POST-ACCUM` clause of a `SELECT` block by using the assignment operator (`=`); An edge attribute's value can be updated within the `ACCUM` clause of a `SELECT` block by using the assignment operator. In fact, the `UPDATE` statement is equivalent to a `SELECT` statement with `ACCUM` and/or `POST-ACCUM` to update the vertex or edge attribute values.

> Updating a vertex's attribute value in an `ACCUM` clause is not allowed, because the update can occur multiple times in parallel, and possibly result in a non-deterministic value. If the vertex attribute value update depends on an edge attribute value, use the vertex-attached accumulators to save the value and update the vertex attribute's value in the `POST-ACCUM` clause.

The query below uses a `SELECT` statement instead of an `UPDATE` statement and performs the same update as the query above. Query `updateEx2` reverses the `locationId` change made by `updateEx` .

UPDATE statement example 2

```
# The second example is equivalent to the above updateEx
CREATE QUERY updateEx2() FOR GRAPH workNet  {
  S = {person.*};

  X = SELECT s
      FROM S:s
      WHERE S.locationId == "USA"
      POST-ACCUM S.locationId = "us",
                 S.skillList = [3,2,1];
  PRINT S;
}
```

Below is an example of an edge update with two attribute changes, including an incremental change:

UPDATE statement example 3

```
CREATE QUERY updateEx3() FOR GRAPH workNet{
  S = {person.*};

  # update edge and target vertices' attribute
  UPDATE e FROM S:s - (worksFor:e) -> :t
  SET e.startYear = e.startYear + 1, // Incremental change
      e.fullTime = false
  WHERE s.locationId == "us";

  PRINT S;
}
```

Other Update Methods

In addition to `UPDATE` statements and `SELECT` statements, a simple assignment statement at the query-body level can be used to update the attribute value of a single vertex or edge, if the vertex or edge has been assigned to a variable or parameter.

update by assignment

```
# change the given person's new location
CREATE QUERY updateByAssignment(VERTEX<person> v, STRING newLocation) FOR
  v.locationId = newLocation;
}
```

Output Statements and FILE Objects

`PRINT` Statement (API v2)

The PRINT statement specifies output data. Each execution of a PRINT statement adds a JSON object to the results array which will be part of the query output. A PRINT statement can appear anywhere that query-body statements are permitted.

> A PRINT statement does not trigger immediate output. The full set of data from all PRINT statements is delivered at one time, when the query concludes.
>
> A query can print a maximum of 2GB of data.
>
> If the output is to a FILE object, then the size limit does not apply.

EBNF

```
printStmt := PRINT printExpr ("," printExpr)* [WHERE condition] [TO_CSV (
printExpr := (expr | vExprSet) [ AS jsonKey]
           | tableName
vExprSet  := expr "[" vSetProj ("," vSetProj)* "]"
vSetProj  := expr [ AS jsonKey]
jsonKey := name
```

Each PRINT statement contains a list of expressions for output data. The optional WHERE clause filters the output. If the *condition* is false for any items, then those items are excluded from the output.

Each `printExpr` contributes one key-value pair to the PRINT statement's JSON object result. The optional AS clause sets the JSON key for the expression, overriding the default key (explained below).

> If the query includes one more tabular SELECT statements, the PRINT statement can include table names. Both tabular and non-tabular `printExpr` expressions can be including in one PRINT statement.

Simple Example Showing JSON Output Format

```
STRING str = "first statement";
INT number = 5;
PRINT str, number;

str = "second statement";
number = number + 1;
PRINT str, number;

# The statements above produce the following output
{
  "version": {"edition": "developer","api": "v2","schema": 0},
  "error": false,
  "message": "",
  "results": [
    {
      "str": "first statement",
      "number": 5
    },
    {
      "str": "second statement",
      "number": 6
    }
  ]
}
```

`PRINT` Expressions

Each `printExpr` may be one of the following:

- A literal value
- A global or local variable (including `VERTEX` and `EDGE` variables)
- An attribute of a vertex variable, e.g., Person.name
- A global accumulator
- An expression whose terms are among the types above. The following operators may be used:

Data type	Operators
String	concatenation: `+`
Set	`UNION INTERSECT MINUS`

Numeric	Arithmetic: `+ - * / . %` Bit: `<< >> & |`

Parentheses can be used for controlling order of precedence.

1. A vertex set variable
2. A vertex expression set `vExprSet` (only available if the output API is set to `"v2"`. Vertex expression sets are explained in a separate section below.

JSON Format: Keys

If a `printExpr` includes the optional **AS *name*** clause, then the *name* sets the key for that expression in the JSON output. Otherwise, the following rules determine the key: If the expression is simply a single variable (local variable, global variable, global accumulator, or vertex set variable), then the key is the variable name. Also, for a vertex expression set, the key is the vertex set variable name. Otherwise, the key is the entire expression, represented as a string.

JSON Format: Values

Each data type has a distinct output format.

- Simple numeric, string, and boolean data types follow JSON standards.
- Lists, sets, bags, and arrays are printed as JSON **arrays** (i.e., a list enclosed in square brackets).
- Maps and tuples are printed as JSON **objects** (i.e., a list of key:value pairs enclosed in curly braces).
- Vertices and edges have a custom JSON object, shown below.
- A vertex set variable is treated as a list of vertices.
- Accumulator output format is determined by the accumulator's return type. For example, an `AvgAccum` outputs a `DOUBLE` value, and a `BitwiseAndAccum` outputs an `INT` value. For container accumulators, simply consider whether the output is a list, set, bag, or map.
    - `ListAccum`, `SetAccum`, `BagAccum`, `ArrayAccum`: list

- `MapAccum`: map
- `HeapAccum`, `GroupByAccum`: list of tuples

Full details of vertices are printed only when part of a vertex set variable or vertex expression set. When a single vertex is printed (from a variable or accumulator whose data type happens to be VERTEX), only the vertex id is printed.

Cases where only the vertex id will be printed

```
ListAccum<VERTEX> @@vList;  // not a vertex set variable
VERTEX v;                   // not a vertex set variable
...
PRINT @@vList, v;           // output will contain only vertex ids
```

Vertex (when not part of a vertex set variable)

The output is just the vertex id as a string:

Output Format for a Value which is a Vertex, not part of a Vertex Set Variable

```
"<vertex_id>"
```

Vertex (as part of a vertex set variable)

Output Format for a Vertex as part of a Vertex Set Variable

```
{
  "v_id":   "<vertex_id>",
  "v_type": "<vertex_type>",
  "attributes": {
    <list of key:value pairs,
     one for each attribute
     or vertex-attached accumulator>
  }
}
```

Edge

Output Format for a Value which is an Edge

```
{
  "e_type":    "<edge_type>",
  "directed":  <boolean_value>,
  "from_id":   "<source_vertex_id>",
  "from_type": "<source_vertex_type>",
  "to_id":     "<target_vertex_id>",
  "to_type":   "<target_vertex_type>",
  "attributes": {
    <list of key:value pairs,
     one for each attribute>
  }
}
```

List, Set or Bag

Output format for a Value which is a List, Set, or Bag

```
[
  <value1>,
  <value2>,
  ...,
  <valueN>
]
```

Map

Output Format for a Value which is a Map

```
{
  <key1>: <value1>,
  <key2>: <value2>,
  ...,
  <keyN>: <valueN>
}
```

Tuple

Output Format for a Value which is a Tuple

```
{
  <fieldName1>: <value1>,
  <fieldName2>: <value2>,
  ...,
  <fieldNameN>: <valueN>
}
```

Vertex Set Variable

Output Format for a Value which is a Vertex Set Variable

```
[
  <vertex1>,
  <vertex2>,
  ...,
  <vertexN>
]
```

Vertex Expression Set

A vertex expression set is a list of expressions which is applied to each vertex in a vertex set variable. The expression list is used to compute an alternative set of values to display in the "attributes" field of each vertex.

The easiest way to understand this is to consider examples containing only one term and then consider combinations. Consider the following example query. C is a vertex set variable containing the set of all company vertices. Furthermore, each vertex has a vertex-attached accumulator @count.

Example Query for Vertex Expression Set

```
# CREATE VERTEX company(PRIMARY_ID clientId STRING, id STRING, country ST

CREATE QUERY vExprSet () FOR GRAPH workNet {
  SumAccum<INT> @count;
  C = {company.*};

  # include some print statements here
}
```

If we print the full vertex set, the "attributes" field of each vertex will contain 3 fields: "id", "country", and "@count". Now consider some simple vertex expression sets:

- `PRINT C[C.country]` prints the vertex set variable C, except that the "attributes" field will contain only "country", instead of 3 fields.
- `PRINT C[C.@count]` prints the vertex set variable C, except that the "attributes" field will contain only "@count", instead of 3 fields.
- `PRINT C[C.@count AS company_count]` prints the same as above, except that the "@count" accumulator is is aliased as "company_count".
- `PRINT C[C.id, C.@count]` prints the vertex set variable C, except that the "attributes" field will contain only "id" and "@count".
- `PRINT C[C.id+"_ex", C.@count+1]` prints the vertex set variable C, except that the "attributes" field contains the following:
    - One field consists of each vertex's id value, with the string "_ex" appended to it.
    - Another field consists of the @count value incremented by 1. Note: the value of @count itself has not changed, only the displayed value is incremented.

The last example illustrates the general format for a vertex expression set:

Syntax for Vertex Expression Set

```
vExprSet  := expr "[" vSetProj {, vSetProj} "]"
vSetProj  := expr [ AS name]
```

The vertex expression set begins with the name of a vertex set variable. It is followed by a list of attribute expressions, enclosed in square brackets. Each attribute expression follows the same rules described earlier in the Print Expressions section. That is, each attribute expression may refer to one or more attributes or vertex-attached accumulators of the current vertices, as well as literals, local or global variables, and global accumulators. The allowed operators (for numeric, string, or set operations) are the same ones mentioned above.

The *key* for the vertex expression set is the vertex set variable name.

The *value* for the vertex expression set is a modified vertex set variable, where the regular "attributes" value for each vertex is replaced with a set of key:value pairs corresponding to the set of attribute expressions given in the print expression.

An example which shows all of the cases described above, in combination, is shown below.

Print Basic Example

```
CREATE QUERY printExampleV2(VERTEX<person> v) FOR GRAPH socialNet {

  SetAccum<VERTEX> @@setOfVertices;
  SetAccum<EDGE> @postedSet;
  MapAccum<VERTEX,ListAccum<VERTEX>> @@testMap;
  FLOAT paperWidth = 8.5;
  INT paperHeight = 11;
  STRING Alpha = "ABC";

  Seed = person.*;
  A = SELECT s
      FROM Seed:s
      WHERE s.gender == "Female"
      ACCUM @@setOfVertices += s;

  B = SELECT t
      FROM Seed:s - (posted:e) -> post:t
      ACCUM s.@postedSet += e,
        @@testMap += (s -> t);

# Numeric, String, and Boolean expressions, with renamed keys:
  PRINT paperHeight*paperWidth AS PaperSize, Alpha+"XYZ" AS Letters,
    A.size() > 10 AS AsizeMoreThan10;
# Note how an expression is named if "AS" is not used:
  PRINT A.size() > 10;

# Vertex variables.  Only the vertex id is included (no attributes):
  PRINT v, @@setOfVertices;

# Map of Person -> Posts posted by that person:
  PRINT @@testMap;

# Vertex Set Variable. Each vertex has a vertex-attached accumulator, whi
# happens to be a set of edges (SetAccum<EDGE>), so edge format is shown
  PRINT A AS VSetVarWomen;

# Vertex Set Expression. The same set of vertices as above, but with only
# one attribute plus one computed attribute:
  PRINT A[A.gender, A.@postedSet.size()] AS VSetExpr;
}
```

Note how the results of the six PRINT statements are grouped in the JSON "results" field below:

1. Each of the six PRINT statements is represented as one JSON object with the "results" array.

2. When a PRINT statement has more than one expression (like the first one), the expressions may appear in the output in a different order than on the PRINT statement.
3. The 2nd PRINT statement shows a key that is generated from the expression itself.
4. The 3rd and 4th PRINT statements show a set of vertices (different than a vertex set variable) and a map, respectively.
5. The 5th PRINT statement shows the vertex set variable A, including its vertex-attached accumulators (PRINT A).
6. The 6th PRINT statement shows a vertex set expression for A, customized to include only one static attribute plus a newly computed attribute.

Results from Query printExampleV2 (WITH COMMENTS ADDED)

```
GSQL > RUN QUERY printExampleV2("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "AsizeMoreThan10": false,
      "Letters": "ABCXYZ",
      "PaperSize": 93.5
    },
    {"A.size()>10": false},
    {
      "v": "person1",
      "@@setOfVertices": [ "person4", "person5", "person2" ]
    },
    {"@@testMap": {
      "person4": ["3"],
      "person3": ["2"],
      "person2": ["1"],
      "person1": ["0"],
      "person8": [ "7", "8" ],
      "person7": [ "9", "6" ],
      "person6": [ "10", "5" ],
      "person5": [ "4", "11" ]
    }},
    {"VSetVarWomen": [
      {
        "v_id": "person4",
        "attributes": {
          "gender": "Female",
          "id": "person4",
          "@postedSet": [{
            "from_type": "person",
            "to_type": "post",
            "directed": true,
            "from_id": "person4",
            "to_id": "3",
            "attributes": {},
            "e_type": "posted"
          }]
        },
        "v_type": "person"
      },
```

```
        {
          "v_id": "person5",
          "attributes": {
            "gender": "Female",
            "id": "person5",
            "@postedSet": [
              {
                "from_type": "person",
                "to_type": "post",
                "directed": true,
                "from_id": "person5",
                "to_id": "11",
                "attributes": {},
                "e_type": "posted"
              },
              {
                "from_type": "person",
                "to_type": "post",
                "directed": true,
                "from_id": "person5",
                "to_id": "4",
                "attributes": {},
                "e_type": "posted"
              }
            ]
          },
          "v_type": "person"
        },
```

Printing CSV to a FILE Object

```
        {
          "v_id": "person2",
          "attributes": {
            "gender": "Female",
            "id": "person2",
            "@postedSet": [{
              "from_type": "person",
              "to_type": "post",
              "directed": true,
              "from_id": "person2",
```

PRINT to CSV FILE syntax example

```
PRINT @@setOfVertices TO_CSV file1;
```

```
              "e_type": "posted"
            }]
          },
          "v_type": "person"
        }
      ]},
      {"VSetExpr": [
        {
```

```
        "v_id": "person4",
        "attributes": {
          "A.@postedSet.size()": 1,
          "A.gender": "Female"
        },
        "v_type": "person"
      },
      {
        "v_id": "person5",
        "attributes": {
          "A.@postedSet.size()": 2,
          "A.gender": "Female"
        },
        "v_type": "person"
      },
      {
        "v_id": "person2",
        "attributes": {
          "A.@postedSet.size()": 1,
          "A.gender": "Female"
        },
        "v_type": "person"
      }
    ]}
  ]
}
```


2. Each line of data to print to a file includes a label which can be used to identify the data.

PRINT WHERE and PRINT TO_CSV FILE Object Example

```
CREATE QUERY printExampleFile() FOR GRAPH socialNet {
  SetAccum<VERTEX> @@testSet, @@testSet2;
  ListAccum<STRING> @@strList;
  int x = 3;
  FILE file1 ("/home/tigergraph/printExampleFile.txt");

  Seed = person.*;
  A = SELECT s
      FROM Seed:s
      WHERE s.gender == "Female"
      ACCUM @@testSet += s, @@strList += s.gender;
  A = SELECT s
      FROM Seed:s
      WHERE s.gender == "Male"
      ACCUM @@testSet2 += s;

  PRINT @@testSet, @@testSet2 TO_CSV file1;  # 1st line: 2 4 5, 1 3 6 7 8
  PRINT x WHERE x < 0 TO_CSV file1;   # 2nd line: <skipped because no con
  PRINT x WHERE x > 0 TO_CSV file1;   # 3rd line: 3
  PRINT @@strList TO_CSV file1;       # 4th line: Female Female Female
  PRINT A.gender TO_CSV file1;     # 5th line: Male\n Male\n Male\n Male\
}
```

`FILE println` statement

The `FILE println` statement writes data to a `FILE` object. Unlike the `PRINT` statement, which is a query-body level statement, the `FILE println` statement can be either a query-body level statement or a DML-sub-statement.

EBNF for FILE println statement

```
printlnStmt := fileVar".println" "(" expr ("," expr)* ")"
```

`println` is a method of a FILE object variable. The `println` statement can be used either at the query-body level or a DML-sub-statement, e.g., within the ACCUM clause of a SELECT block. Each time `println` is called, it adds one new line of values to the `FILE` object, and then to the corresponding file.

The `println` function can print any expression that can be printed by a `PRINT` statement with the exception of vertex set variables. Vertex expression sets are also not applicable to the `println` function.

If the `println` statement has a list of expressions to print, it will produce a comma-separated list of values. If an expression refers to a list or set, then the output will be a list of values separated by spaces.

> The data from query-body level `FILE` print statements (either `TO_CSV` or `println`) will appear in their original order. However, due to the parallel processing of statements in an ACCUM block, the order in which `println` statements at the DML-sub-statement level are processed cannot be guaranteed.

Example

File object query example

```
CREATE QUERY fileEx (STRING fileLocation) FOR GRAPH workNet {

    FILE f1 (fileLocation);
    P = {person.*};

    PRINT "header" TO_CSV f1;

    USWorkers = SELECT v FROM P:v
              WHERE v.locationId == "us"
              ACCUM f1.println(v.id, v.interestList);

    PRINT "footer" TO_CSV f1;
}
INSTALL QUERY fileEx
RUN QUERY fileEx("/home/tigergraph/files")
```

All of the `PRINT` statements in this example use the `TO_CSV` option, so there is no JSON output to the console.

Results from Query fileEx

```
GSQL > RUN QUERY fileEx("/home/tigergraph/fileEx.txt")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": []
}
```

All the output in this case goes to the `FILE` object. In the query definition, the line `"header"` is printed first, followed by the `println` statements in the `ACCUM` clause, and `"footer"` is printed last. The output in the file follows this order because the order of query-body level statements is maintained in the output.

File contents produced by fileEx example

```
[tigergraph@localhost]$ more /home/tigergraph/fileEx.txt
header
person7,art sport
person10,football sport
person4,football
person9,financial teaching
person1,management financial
footer
```

However, within the `ACCUM` clause itself, the order of the `println` statements is not guranteed.

Passing a FILE Object as a Parameter

A FILE Object can be passed from one query to a subquery. The subquery can then also write to the FILE object.

Example: query passing a FILE object to another query

```
CREATE QUERY fileParamSub(FILE f, STRING label, INT num) FOR GRAPH social
    f.println(label, "header");
    FOREACH i IN RANGE [1,2] DO
        f.println(label, num+i);
    END;
    f.println(label, "footer");
}

CREATE QUERY fileParamMain(STRING mainlabel) FOR GRAPH socialNet {
    FILE f ("/home/tigergraph/fileParam.txt");
    f.println(mainlabel, "header");
    FOREACH i IN RANGE [1,2] DO
        f.println(mainlabel, i);
        fileParamSub(f, " sub", 10*i);
    END;
    f.println(mainlabel, "footer");
}
```

```
GSQL > RUN QUERY fileParamMain("main")
GSQL > EXIT

$ cat /home/tigergraph/fileParam.txt
main,header
main,1
 sub,header
 sub,11
 sub,12
 sub,footer
main,2
 sub,header
 sub,21
 sub,22
 sub,footer
main,footer
```

`LOG` Statement

The LOG statement is another means to output data. It works as a function that outputs information to a log file.

EBNF for LOG statement

```
logStmt := LOG "(" condition "," argList ")"
```

The first argument of the LOG statement is a boolean condition that enables or disables logging. This allows logging to be easily turned on/off, for uses such as debugging. After the condition, LOG takes one or more expressions (separated by commas). These expressions are evaluated and output to the log file.

Unlike the PRINT statement, which can only be used as a query-body statement, the LOG statement can be used as both a query-body statement and a DML-sub-statement.

The values will be recorded in the GPE log. To find the log file after the query has completed, open a Linux shell and use the command "gadmin log gpe". It may show you more than one log file name; use the one ending in "INFO". Search this file for "UDF_".

Examples

```
BOOLEAN debug = TRUE;
INT x = 10;

LOG(debug, 20);
LOG(debug, 10, x);
```

`RETURN` Statement

EBNF for RETURN statement

```
returnStmt := RETURN expr
```

The RETURN statement specifies data that a *subquery* passes back to an outer query that called the subquery. The return type for a `RETURN` statement can be any base type or accumulator type, but **must be the same type** as indicated by the `RETURNS` clause of the subquery.

Subquery Returning HeapAccum Example

```
TYPEDEF tuple<name string, friends int> myTuple
TYPEDEF HeapAccum<myTuple>(3, friends DESC) myHeap

CREATE QUERY subquery1() FOR GRAPH socialNet RETURNS (myHeap){
    myHeap @@heap;      // Define the heap accumulator at the global leve
    SumAccum<int> @friends;
    Start = {person.*};
    Start = select s from Start:s-(friend:e)-:t
            accum s.@friends += 1
            post-accum @@heap += myTuple(s.id,s.@friends);
    RETURN @@heap;
}

CREATE QUERY query1() FOR GRAPH socialNet {
    PRINT subquery1();
}
```

Exception Statements

This section describes how the GSQL language responds to exceptions and supports user-defined exception handling . An exception is a run-time error. The GSQL language supports both built-in system exceptions and user-defined exceptions. Built-in exceptions include GSQL language exceptions (such as out-of-range value, wrong data type, and illegal operation), and errors arising in other TigerGraph components or from the operation system.

The GSQL query language also supports user-defined exception responses, also known as exception handling. This section covers the following syntax for user-defined exception behavior:

```
#########################################################################
## Exception Statements ##

declExceptStmt := EXCEPTION exceptVarName "(" errorCode ")"
exceptVarName  := name
errorCode      := integer

raiseStmt        := RAISE exceptVarName [errorMsg]
errorMsg         := "(" expr ")"

tryStmt          := TRY queryBodyStmts EXCEPTION caseExceptBlock+
                    [elseExceptBlock] END ";"
caseExceptBlock := WHEN exceptVarName THEN queryBodyStmts
elseExceptBlock := ELSE queryBodyStmts
```

Default Exception Response

When an exception occurs during the execution of a query, the default response is the following:

- The query will not execute any more statements; it will exit.
- If the query was run using the RUN QUERY command, then an error message will be displayed.

- If the query was run by invoking the GET /query REST++ endpoint, then the output will be a simple JSON object. Some errors have a error "code" field; others do no t:

Output of Unhandled Exception (query run as REST Endpoint)

```
{
  "error": true,
  "message": "<errorMsg>"
  "code": "<errType><errorCode>"
}
```

The example below show two common errors: wrong data type and divide-by-zero. First we define a simple query that divides 100.0 by the query's input parameter.

Example: query excpBuiltin

```
CREATE QUERY excpBuiltin(INT n1) FOR GRAPH minimalNet {
  PRINT 100.0/n1;
}
```

We then test three cases:

1. A valid input (such as n1 = 7)
2. Wrong data type (n1 = "A")
3. Divide by zero (n1 = 0)

First we test using the GSQL interface. When the query runs without error, the output is in JSON format. Where there is a built-in exception, however, only an error message is displayed.

Exception response for RUN QUERY

```
GSQL > RUN QUERY excpBuiltin(7)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"100.0/n1": 14.28571}]
}

GSQL > RUN QUERY excpBuiltin("a")
Values of parameter n1 must be INT64 type, invalid value [a] provided.

GSQL > RUN QUERY excpBuiltin(0)
Runtime Error: divider is zero.
```

The situation is a little different when running the query as a REST++ endpoint. The output is always in JSON format.

> As of TigerGraph v1.2, the format for the GET /query endpoint has changed. The graph name must now be specified after /query: `/query/{graph_name}/{query_name}`

Exception response for GET /query request

```
$ curl -X GET "http://localhost:9000/query/minimalNet/excpBuiltin?n1=7"
{
    "error": false,
    "message": "",
    "results": [
        {
            "100.0/n1": 14.28571
        }
    ],
    "version": {
        "edition": "developer",
        "api": "v2",
        "schema": 0
    }
}

$ curl -X GET "http://localhost:9000/query/minimalNet/excpBuiltin?n1=a"
{
    "code": "REST-30000",
    "error": true,
    "message": "Values of parameter n1 must be INT64 type, invalid value
    "version": {
        "edition": "developer",
        "api": "v2",
        "schema": 0
    }
}

$ curl -X GET "http://localhost:9000/query/minimalNet/excpBuiltin?n1=0"
{
    "error": true,
    "message": "Runtime Error: divider is zero.",
    "version": {
        "edition": "developer",
        "api": "v2",
        "schema": 0
    }
}
```

User-Defined Exception Behavior

A query author can specify what should be the response if a particular type of exception occurs within a particular specified block of statements.

The following statement types are available to specify a user-defined exception condition or a user-defined exception response.

- The EXCEPTION Declaration Statement names a user-defined exception.
- The RAISE Statement indicates that one of the user-defined exceptions has occurred.
- The TRY...EXCEPTION Statement is used to define and apply user-defined exception handling to a block of query-body statements. This can be used with or without preceding user-defined EXCEPTION and RAISE statements.

> Built-in exceptions always take precedence over user-defined exceptions. Therefore, user-defined exceptions can only be used to catch conditions that would not be caught by a built-in exception. This means that built–in exceptions are best used to capture situations which are legal according to the general syntax and semantics of the GSQL query language, but which are illegal or undesirable for a particular user application.

EXCEPTION Declaration Statement

```
declExceptStmt := EXCEPTION exceptVarName "(" errorCode ")"
exceptVarName := name
errorCode     := integer
```

To use a user-defined exception, it must first be declared. An exception declaration statement declares a user-defined exception type, assigning a name and identification number. The id number errorCode must be greater than 40,000. Numbers 40,000 and lower are reserved for system exceptions. Exception statements must be placed before any query-body statements, after accumulator declaration statements . A query can declare multiple exception types.

RAISE Statement

```
raiseStmt := RAISE exceptVarName [errorMsg]
errorMsg := "(" expr ")"
```

The RAISE statement announces that a user-defined exception has just occurred. The exceptVarName must match one of the exceptions that was previously declared. An optional error message can be specified. Once the RAISE statement is executed, the flow of execution changes. If the RAISE statement is not within a TRY clause, then the query ends with the default exception response, using the error code and error message defined by the exception type and RAISE statements. If the RAISE is within a TRY statement, then execution jumps to the EXCEPTION handling clause of the TRY statement.

A RAISE statement itself does not include the conditions that define the exception. Typically, the user will use an IF...THEN statement and place the RAISE statement within the THEN clause.

> ⚠ In the current version, a RAISE statement can only be used as a query-body-statement. It cannot be used as a DML-sub-statement. In particular, you cannot RAISE an exception inside a SELECT statement.

The example below defines and checks for two types of exceptions: an empty input set (40001) and no matching edges (40002). Remember that the minimum allowed code number is 40001.

Example: Unhandled User-Defined Exceptions

```
CREATE QUERY excpCountActivity(SET<VERTEX<person>> vSet, STRING eType) FO
  # Count how many edges there are from each member of the input person s
  # along the specified edge type.

  MapAccum<STRING,INT> @@allCount;
  EXCEPTION emptyList (40001);
  EXCEPTION noEdges   (40002);

  IF ISEMPTY(vSet) THEN ## Raise 40001
    RAISE emptyList ("Error: Input parameter 'vSet' (type SET<VERTEX<pers
  END;

  Start = vSet;
  Results = SELECT s
    FROM Start:s -(:e)-> post:t
    WHERE e.type == eType
    ACCUM @@allCount += (t.subject -> 1);

  IF Results.size() == 0 THEN ## Raise 40002
    RAISE noEdges ("Error: No '" + eType + "' edges from the vertex set")
  END;
  PRINT @@allCount;
}
```

Results

```
// Valid input: no exceptions
$ curl -X GET "http://localhost:9000/query/socialNet/excpCountActivity?vS
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "@@allCount": {
      "cats": 1,
      "tigergraph": 2
    }
  }]
}

// empty input set (due to spelling error in parameter name)
$ curl -X GET "http://localhost:9000/query/socialNet/excpCountActivity?vs
{
  "code": "40001",
  "error": true,
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "message": "Error: Input parameter 'vSet' (type SET<VERTEX<person>>) is
}


// no edges (due to unknown edge type)
$ curl -X GET "http://localhost:9000/query/socialNet/excpCountActivity?vS
{
  "code": "40002",
  "error": true,
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "message": "Error: No 'commented' edges from the vertex set"
}
```

TRY...EXCEPTION Statement for Custom Error Handling

```
tryStmt          := TRY queryBodyStmts EXCEPTION caseExceptBlock [elseExce
caseExceptBlock := WHEN exceptVarName THEN queryBodyStmts+
elseExceptBlock := ELSE queryBodyStmts
```

The TRY...EXCEPTION Statement is used to define and apply user-defined exception handling to a block of query-body statements. A TRY...EXCEPTION statement can be nested within a TRY block or EXCEPTION block.

> ⚠ The current version of GSQL does not support custom handling of built-in exceptions. Therefore, if a built-in exception occurs, it ignores the TRY..EXCEPTION blocks and simply applies the default handling, and the query aborts. In future updates, we plan to support custom handling of both custom exceptions (RAISE) and built-in exception with the TRY...EXCEPTION block.

The TRY...EXCEPTION Statement is a compound statement containing two blocks. The first block (TRY) consists of the query-body statements for which custom error handling should be applied. The second block (EXCEPTION) contains a series of WHEN...THEN exception handling clauses. Each exception handling clause names an exception type and specifies what actions to take in the event of the exception. An optional ELSE clause contains handling statements for all other exceptions. The following text and visual flowchart details how the TRY... EXCEPTION block handles an exception.

When an exception occurs within a TRY block, the flow of execution skips the remainder of the TRY block and jumps to the EXCEPTION block. The GSQL flow now seeks to match the exception type with a handler. After executing the handling statements in the THEN or ELSE clause, the flow skips the remainder of the EXCEPTION block and continues with the statement following the END statement. However, if there is no matching WHEN or ELSE handler, then the exception is propagated. That is, the RAISE state is maintained after exiting the EXCEPTION block. If the TRY...EXCEPTION block is nested inside another TRY block, then the handling process is repeated at this upper level. This repeats until either the exception is handled or there are no more TRY...EXCEPTION blocks.

Finally, if the unhandled exception is not within a TRY block, then the the query is aborted, and the default exception response is the output.

```
TRY
    stmts
    IF cond1 THEN RAISE A; END;
    TRY
        stmts
        IF cond2 THEN RAISE A; END;
        IF cond3 THEN RAISE B; END;
    EXCEPTION
        WHEN A THEN handStmtsX;
    END;
    stmts
EXCEPTION
    WHEN B THEN handStmtsY;
    ELSE handStmtsZ;
END;
```

Case 1: If cond1 is true in the outer TRY block,

- RAISE A and jump to the output EXCEPTION block.

Handled by ELSE HandStmtsZ.

Case 2: If cond2 is true in the inner TRY block,

- RAISE A and jump to the inner EXCEPTION block.

Handled by handStmtsX;

Case 3: If cond3 is true in the inner TRY block,

- RAISE B and jump to the inner EXCEPTION block. There is no matching handler here, so propagate the exception. Jump to the outer EXCEPTION block. Handled by handStmtsY.

Custom Handling Example:

The following example is a modified shortest path query. It looks for all paths from a source to a target in a computer network. It uses breadth-first search and stops at depth N when it has found at least one path at depth N, or it has searched the entire graph. There are three conditions which will cause it to RAISE an exception and abort the search:

1. Seeing an edge with a negative connection speed (because the graph has bad data).
2. Seeing an edge with a very slow connection speed (again because the graph has bad data).
3. If no path was found in the graph (the search is already over, but we skip printing results).

Note that cases 1 and 2 do NOT mean that a negative or slow speed edge is actually on a shortest path, only that the query noticed a bad edge during its search. Also, because we cannot RAISE within the SELECT block, we use a workaround: set an integer variable with an error code. Immediately after the SELECT block, test the integer variable and RAISE exceptions as needed.

Example: Path Search with Exceptions

```
CREATE QUERY compPathValid (vertex<computer> src, vertex<computer> tgt, B
FOR GRAPH computerNet {
# Find valid paths in a computer network from a source to a target.
# Stop search once you have found some paths.
# 3 Exceptions: (1) Negative connection speed, (2) Slow connection speed,
# Set enExcp=true to raise exceptions. enExcp=false will find paths, good

    OrAccum @@reached, @visited;
    ListAccum<STRING> @paths;
    DOUBLE minSpeed = 0.4;
    INT err;

    EXCEPTION negSpeed (40001);
    EXCEPTION slowSpeed (40002);
    EXCEPTION notReached (40003);

    TRY
        Start = {src};
        # Initialize: path to src is itself.
        Start = SELECT s
            FROM Start:s
            ACCUM s.@paths = s.id;

        WHILE Start.size() != 0 AND NOT @@reached DO
            Start = SELECT t
                FROM Start:s -(:e)-> :t
                WHERE t.@visited == false
                ACCUM CASE
                    WHEN e.connectionSpeed < 0 THEN err = 1
                    WHEN e.connectionSpeed < minSpeed THEN err = 2
                    WHEN t == tgt THEN @@reached += true
                    END,
            # List1 * List2 -> List(each elem of List1 concat w/each elem
                    t.@paths += (s.@paths * ["~"]) * [t.id]
                POST-ACCUM t.@visited = true;
            IF err == 1 AND enExcp THEN
                RAISE negSpeed ("Negative Speed");
            ELSE IF err == 2 AND enExcp THEN
                RAISE slowSpeed ("Slow Speed");
            END;
        END; # WHILE

        IF NOT @@reached AND enExcp THEN
            RAISE notReached ("No path to target");
        ELSE
            Result = {tgt};
            PRINT Result[Result.@paths]; // api v2
        END:
```

```
    EXCEPTION
        WHEN negSpeed THEN PRINT "bad path: negative speed";
        WHEN slowSpeed THEN PRINT "bad path: slow speed";
        WHEN notReached THEN PRINT "no path from source to target";
    END;
}
```

As the data in Appendix D show:

- Any search passing through c1 will see negative edges.
- Any search passing through c12 will see negative and slow edges.
- Any search passing through c14 will see negative edges.

The results for 5 cases are shown: 1 valid search plus each of the 3 exception conditions. The 5th case is the same as the 4th, but exception handling is not enabled.

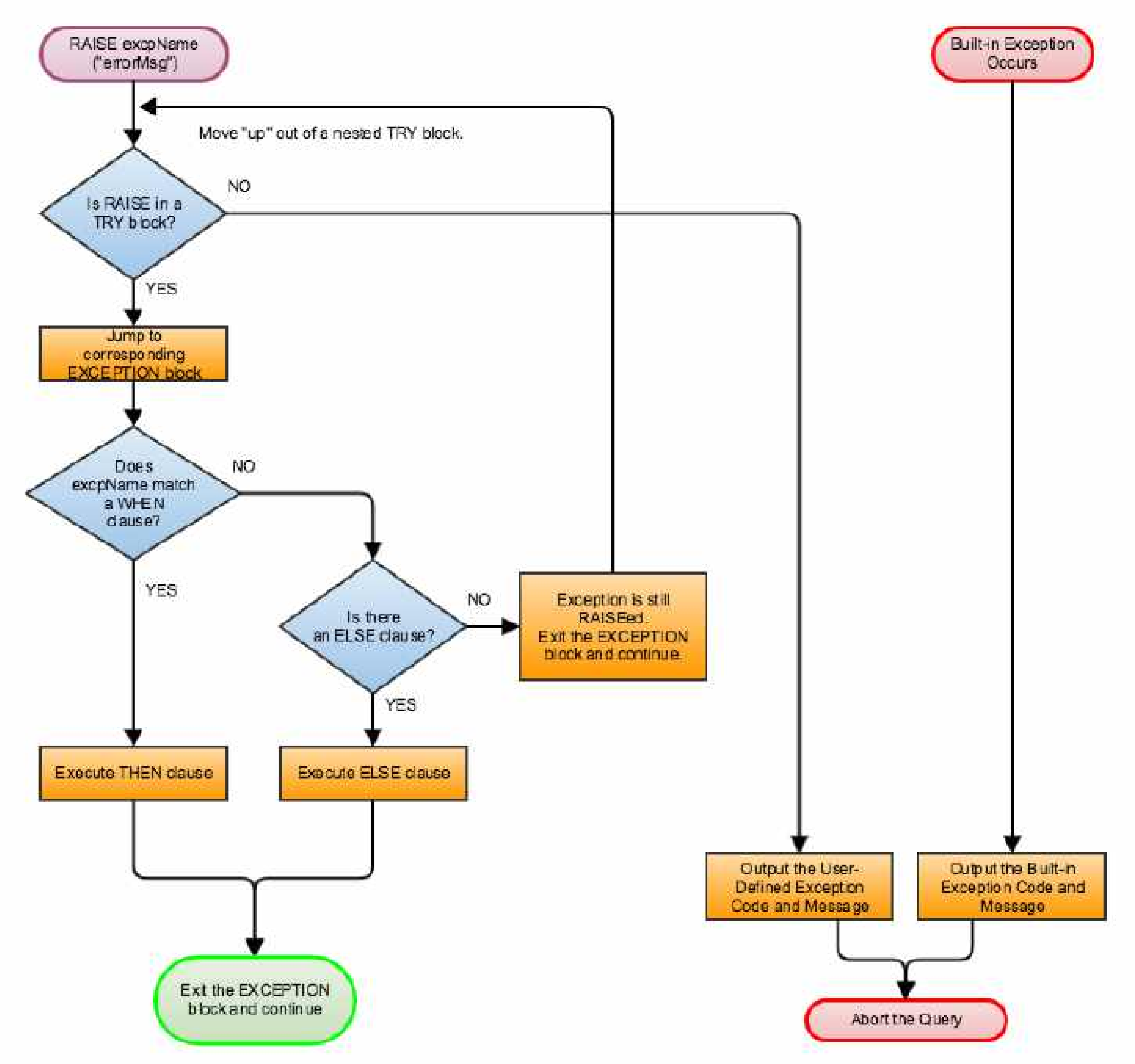
compPathValid.json

```
GSQL > RUN QUERY compPathValid("c10","c12",true)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "c12",
    "attributes": {"Result.@paths": ["c10~c11~c12"]},
    "v_type": "computer"
  }]}]
}
GSQL > RUN QUERY compPathValid("c1","c12",true)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"bad path: negative speed": "bad path: negative speed"}]
}
GSQL > RUN QUERY compPathValid("c10","c13",true)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"bad path: slow speed": "bad path: slow speed"}]
}
GSQL > RUN QUERY compPathValid("c24","c25",true)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"no path from source to target": "no path from source to t
}
```

```
}
GSQL > RUN QUERY compPathValid("c24","c25",false)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "c25",
    "attributes": {"Result.@paths": []},
    "v_type": "computer"
  }]}]
}
```

Exception Handling Flowchart

The flowchart below summarizes all the cases for triggering and handling exceptions, both user-defined and built-in.


RAISE excpName ("errorMsg")
Built-in Exception Occurs
Move "up" out of a nested TRY block.
Is RAISE in a TRY block?
NO
YES
Jump to corresponding EXCEPTION block
Does excpName match a WHEN clause?
NO
YES
Is there an ELSE clause?
NO
Exception is still RAISEed. Exit the EXCEPTION block and continue
YES
Execute THEN clause
Execute ELSE clause
Output the User-Defined Exception Code and Message
Output the Built-in Exception Code and Message
Exit the EXCEPTION block and continue
Abort the Query

Loading...

Appendix

Common Errors and Problems

Floating Point Precision Limits

No computer can store all floating point numbers (i.e., non-integers) with perfect precision. The **float** data type offers about 7 decimal digits of precision; the **double** data type offers about 15 decimal digits of precision. Comparing two float or double values by using operators involving exact equality (==, <=, >=, BETWEEN ... AND ...) might lead to unexpected behavior. If the GSQL language parser detects that the user is attempting an exact equivalence test with float or double data types, it will display a warning message and suggestion. For example, if there are two float variables v and v2, the expression v == v2 causes the following warning message:

```
The comparison 'v==v2' may lead to unexpected behavior because it involve
equality test between float/double numeric values. We suggest to do such
comparison with an error margin, e.g. 'abs((v) - (v2)) < epsilon', where
is a very small positive value of your choice, such as 0.0001.
```

Response to Non-existent vertex ID

If a query has a vertex parameter (VERTEX or VERTEX<vType>), and if the ID for a nonexistent vertex is given when running the query, an error message is shown, and the query won't run. This is also the response when calling a function to convert a single vertex ID string to a vertex:

- to_vertex(): See Section "Miscellaneous Functions".

However, if the parameter is a vertex set (SET<VERTEX> or SET<VERTEX<vType>>), and one or more nonexistent IDs are given when running the query, a warning message is shown, but the query still runs, ignoring those nonexistent IDs. Therefore, if all given IDs are nonexistent, the parameter becomes an empty set. T his is also the response when calling a function to convert a set of vertex IDs to a set of vertices :

- to_vertex_set(): See Section " Miscellaneous Functions ".
- SelectVertex(): See Section " Miscellaneous Functions ".

Loading...

Loading...

Example Graphs

Below is the listing of the graph create&load command files and data files to generate the six example graphs used in this document: **workNet** , **socialNet** , **friendNet** , **computerNet** , **minimalNet** , and**investmentNet** . The tar-gzip file **gsql_ref_examples_2.0.gz** contains all of these files. Each graph has its own folder. To create a particular graph, go in its folder and run the following command:
gsql graph_create.gsql

66KB gsql_ref_examples_2.0.gz

gsql_ref_examples_2.0.gz

workNet

graph_create.gsql for workNet

```
# Updated 5/1/18 for v2.0
DROP ALL
CREATE VERTEX person(PRIMARY_ID personId STRING, id STRING, locationId ST
CREATE VERTEX company(PRIMARY_ID clientId STRING, id STRING, country STRI
CREATE UNDIRECTED EDGE worksFor(FROM person, TO company, startYear INT, s
CREATE GRAPH workNet(*)

USE GRAPH workNet // v1.2
CREATE LOADING JOB loadMember FOR GRAPH workNet {
  DEFINE FILENAME f;
  LOAD f
    TO VERTEX person VALUES($0, $0, $1, _, _, SPLIT($3,"|"), SPLIT($3,"|"
    TO TEMP_TABLE t2(id, skill) VALUES ($0, flatten($2,"|",1));

  LOAD TEMP_TABLE t2
    TO VERTEX person VALUES($0, _, _, $"skill", $"skill", _, _);
}

CREATE LOADING JOB loadCompany FOR GRAPH workNet {
  DEFINE FILENAME f;
  LOAD f TO VERTEX company VALUES($0, $0, $1);
}

CREATE LOADING JOB loadMemberCompany FOR GRAPH workNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE worksFor VALUES($0, $1, $2, $3, $4);
}

RUN LOADING JOB loadMember USING f="./persons"
RUN LOADING JOB loadCompany USING f="./companies"
RUN LOADING JOB loadMemberCompany USING f="./person_company"
```

file: persons (vertices)

```
person1,us,1|2|3,management|financial
person2,chn,2|3|5|6,engineering
person3,jp,4|1|6,teaching
person4,us,4|1|10,football
person5,can,|8|2|5,sport|financial|engineering
person6,jp,7|10,music|art
person7,us,8|6,art|sport
person8,chn,1|5|2,management
person9,us,4|7|2,financial|teaching
person10,us,3,football|sport
person11,can,10,sport|football
person12,jp,1|5|2|2|2,music|engineering|teaching|teaching|teaching
```

file: company (vertices)

```
company1,us
company2,chn
company3,jp
company4,us
company5,can
```

file: person_company (edges)

```
person1,company1,2016,1,1
person1,company2,2014,3,0
person2,company1,2015,7,1
person2,company2,2012,6,0
person3,company1,2016,6,1
person4,company2,2013,2,1
person5,company2,2016,4,0
person6,company1,2015,1,1
person7,company2,2016,3,0
person7,company3,2014,1,0
person8,company1,2013,2,1
person9,company2,2015,12,1
person9,company3,2016,11,1
person10,company1,2016,2,1
person10,company3,2014,5,0
person11,company5,2016,5,1
person12,company4,2014,1,1
```

socialNet

graph_create.gsql for socialNet

```
# Updated 5/1/18 for v2.0
DROP ALL
CREATE VERTEX person(PRIMARY_ID personId UINT, id STRING, gender STRING)
CREATE UNDIRECTED EDGE friend(FROM person, TO person)
CREATE VERTEX post(PRIMARY_ID postId UINT, subject STRING, postTime DATET
CREATE DIRECTED EDGE posted(FROM person, TO post)
CREATE DIRECTED EDGE liked(FROM person, TO post, actionTime DATETIME)
CREATE GRAPH socialNet(*)

USE GRAPH socialNet // v1.2
CREATE LOADING JOB loadMember FOR GRAPH socialNet {
  DEFINE FILENAME f;
  LOAD f TO VERTEX person VALUES($0, $0, $1) ;
}

CREATE LOADING JOB loadFriend FOR GRAPH socialNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE friend VALUES($0, $1) ;
}

CREATE LOADING JOB loadPost FOR GRAPH socialNet {
  DEFINE FILENAME f;
  LOAD f TO VERTEX post VALUES($0, $1, $2);
}

CREATE LOADING JOB loadPosted FOR GRAPH socialNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE posted VALUES($0, $1) ;
}

CREATE LOADING JOB loadLiked FOR GRAPH socialNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE liked VALUES($0, $1, $2) ;
}

RUN LOADING JOB loadMember USING f="./persons"
RUN LOADING JOB loadFriend USING f="./friends"
RUN LOADING JOB loadPost USING f="./posts"
RUN LOADING JOB loadPosted USING f="./posted"
RUN LOADING JOB loadLiked USING f="./liked"
```

file: persons (vertices)

```
person1,Male
person2,Female
person3,Male
person4,Female
person5,Female
person6,Male
person7,Male
person8,Male
```

file: friends (edges)

```
person1,person2
person2,person3
person3,person4
person4,person5
person4,person6
person5,person7
person6,person8
person7,person8
person8,person1
```

file: posts (vertices)

```
0,Graphs,2010-01-12 11:22:05
1,tigergraph,2011-03-03 23:02:00
2,query languages,2011-02-03 01:02:42
3,cats,2011-02-05 01:02:44
4,coffee,2011-02-07 05:02:51
5,tigergraph,2011-02-06 01:02:02
6,tigergraph,2011-02-05 02:02:05
7,Graphs,2011-02-04 17:02:41
8,cats,2011-02-03 17:05:52
9,cats,2011-02-05 23:12:42
10,cats,2011-02-04 03:02:31
11,cats,2011-02-03 01:02:21
```

file: posted (edges)

```
person1,0
person2,1
person3,2
person4,3
person5,4
person5,11
person6,5
person6,10
person7,6
person7,9
person8,7
person8,8
```

file: liked (edges)

```
person1,0,2010-01-11 11:32:00
person2,0,2010-01-12 10:52:15
person2,3,2010-01-11 16:02:26
person3,0,2010-01-16 05:15:53
person4,4,2010-01-13 03:16:05
person5,6,2010-01-12 21:12:05
person6,8,2010-01-14 11:23:05
person7,10,2010-01-12 11:22:05
person8,4,2010-01-11 03:26:05
```

friendNet

graph_create.gsql for friendNet

```
# Updated 5/1/18 for v2.0
DROP ALL
CREATE VERTEX person(PRIMARY_ID personId UINT, id STRING)
CREATE UNDIRECTED EDGE friend(FROM person, TO person)
CREATE UNDIRECTED EDGE coworker(FROM person, TO person)
CREATE GRAPH friendNet(*)

USE GRAPH friendNet // v1.2
CREATE LOADING JOB loadMember FOR GRAPH friendNet {
  DEFINE FILENAME f;
  LOAD f TO VERTEX person VALUES($0, $0);
}

CREATE LOADING JOB loadFriend FOR GRAPH friendNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE friend VALUES($0, $1);
}

CREATE LOADING JOB loadCoworker FOR GRAPH friendNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE coworker VALUES($0, $1);
}

RUN LOADING JOB loadMember USING f="./persons"
RUN LOADING JOB loadFriend USING f="./friends"
RUN LOADING JOB loadCoworker USING f="./coworkers"
```

file: persons (vertices)

```
person1
person2
person3
person4
person5
person6
person7
person8
person9
person10
person11
person12
```

file: friends (edges)

```
person1,person2
person1,person3
person1,person4
person2,person8
person3,person9
person4,person6
person5,person6
person6,person9
person7,person9
person8,person10
person9,person8
person10,person12
person11,person12
person12,person8
person12,person9
```

file: coworkers (edges)

```
person1,person4
person1,person5
person1,person6
person2,person3
person2,person4
person3,person5
person3,person6
person4,person5
person4,person6
person5,person6
person6,person5
person7,person9
person7,person5
person7,person4
person8,person9
person9,person2
person10,person7
person11,person7
person12,person7
```

computerNet

graph_create.gsql for computerNet

```
# Updated 5/1/18 for v2.0
DROP ALL
CREATE VERTEX computer(PRIMARY_ID compID UINT, id STRING)
CREATE DIRECTED EDGE connected(FROM computer, TO computer, connectionSpee
CREATE GRAPH computerNet(*)

USE GRAPH computerNet // v1.2
CREATE LOADING JOB loadComputer FOR GRAPH computerNet {
  DEFINE FILENAME f;
  LOAD f TO VERTEX computer VALUES($0, $0);
}
CREATE LOADING JOB loadConnection FOR GRAPH computerNet {
  DEFINE FILENAME f;
  LOAD f TO EDGE connected VALUES($0, $1, $2, $3);
}

RUN LOADING JOB loadComputer USING f="./computers"
RUN LOADING JOB loadConnection USING f="./connections"
```

file: computers (vertices)

```
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12
c13
c14
c15
c16
c17
c18
c19
c20
c21
c22
c23
c24
c25
c26
c27
c28
c29
c30
c31
```

file: connections (edges)

```
c1,c2,16.0,3
c1,c3,64.0,3
c1,c4,64.0,2
c1,c5,16.5,3
c1,c6,64.3,3
c1,c7,3.2,3
c1,c8,-3.5,3
c1,c9,-5.1,1
c1,c10,15.5,3
c1,c10,.5,1
c1,c10,126,3
c10,c11,16,3
c11,c12,.5,3
c12,c13,-0.5,3
c12,c14,0.16,4
c12,c15,1e2,3
c12,c16,3.516e3,3
c12,c17,5.12e-3,2
c12,c18,-2.34e-5,1
c12,c19,-0.000000000234,5
c12,c20,0.000123e-5,4
c12,c21,1000e3,1
c12,c22,0.000123e10,1
c14,c23,123456e-6,1
c14,c24,123456e5,3
c23,c24,64,2
c23,c25,16,2
c23,c26,32,2
c23,c27,16,2
c23,c28,3,1
c23,c29,32,2
c23,c30,16,2
c23,c25,3,2
c23,c26,3,2
c23,c27,64,2
c23,c28,32,2
c23,c29,3,2
c23,c30,3,2
c23,c31,32,2
c4,c23,16,2
c4,c23,32,2
c4,c23,64,2
c4,c23,3,2
```

minimalNet

graph_create.gsql for minimalNet

```
DROP ALL
CREATE VERTEX testV(PRIMARY_ID id STRING)
CREATE UNDIRECTED EDGE testE(FROM testV, TO testV)
CREATE GRAPH minimalNet(*)
```

There is no loading job or data for minimalNet (hence, "minimal.")

investmentNet

graph_create.gsql for investmentNet

```
# Updated 5/1/18 for v2.0
DROP ALL
TYPEDEF TUPLE <age UINT (4), mothersName STRING(20) > SECRET_INFO
CREATE VERTEX person(PRIMARY_ID personId STRING, portfolio MAP<STRING, DO
CREATE VERTEX stockOrder(PRIMARY_ID orderId STRING, ticker STRING, orderS
CREATE UNDIRECTED EDGE makeOrder(FROM person, TO stockOrder, orderTime DA
CREATE GRAPH investmentNet (*)

USE GRAPH investmentNet // v1.2
CREATE LOADING JOB loadPerson FOR GRAPH investmentNet {
  DEFINE FILENAME f;
    LOAD f
    TO VERTEX person VALUES($0, SPLIT($1, ":", ";"), SECRET_INFO( $2, $3
}

CREATE LOADING JOB loadOrder FOR GRAPH investmentNet {
  DEFINE FILENAME f;
    LOAD f
    TO VERTEX stockOrder VALUES($1, $3, $4, $5),
    TO EDGE makeOrder VALUES($0, $1, $2);
}

RUN LOADING JOB loadPerson USING f="./persons"
RUN LOADING JOB loadOrder USING f="./orders"
```

file: persons (vertices)

```
person1,AAPL:3142.24;G:6112.23;MS:5000.00,25,JAMES
person2,A:5242.62;GCI:5331.21;BAH:3200.00,67,SMITH
person3,AA:5223.73;P:7935.00;BAK:6923.52,45,WILLIAMS
person4,ACH:3542.62;S:6521.55;BABA:4030.52,51,ANTHONY
```

file: orders (vertices and edges)

```
person1,0,1488566548,AAPL,500,34.42
person1,1,1488566549,A,210,50.55
person1,2,1488566550,B,211,202.32
person2,3,1488566555,S,2,42.44
person3,4,1488566155,ABC,2,52.44
person4,5,1488566255,Z,2,62.34
person4,6,1488566655,S,2,10.01
```

Interpreted GSQL Limitations

Currently, a query defined with `CREATE DISTRIBUTED QUERY` cannot be run in Interpreted Mode. However, interpreted queries can still run on a distributed graph with a regular, non-distributed execution model.

The list below includes additional limitations. These limitations are expected to be temporary. We are continuing to expand the capabilities of Interpreted Mode.

Unsupported Features in Interpreted Mode

- Distributed Query Mode
- Any operation involving files
  - `FILE` objects
  - `PRINT TO_CSV`
  - `LoadAccum()` function
  - `SelectVertex()` function
- Statements
  - Exception statements:
    - `RAISE` statements
    - `TRY...EXCEPTION` statements
  - `RETURN` statement
    - Returning a vertex set variable
  - `SELECT` statement
    - `ORDER BY` clause
  - Assigning value to a global variable at the statement level
- Attributes and accumulators
  - `ArrayAccum`
  - Accessing the previous value of a vertex-attached accumulator with the ' operator, e.g., `src.@acc'`
  - `STRING COMPRESS` as accumulator type

    - Accessing the attribute of a vertex variable that is not a vertex alias defined in a `FROM` clause
- Functions and Operators
    - Aggregate functions: `AVG()`, `MIN()`, `MAX()`, `COUNT()`, `ISEMPTY()`, `SUM()`
    - `neighbor()`, `neighborAttribute()`
    - `COALESCE()`
    - `IS NULL`, `IS NOT NULL`
    - `evaluate()`
    - `datetime_format()`
    - `parse_json_object()`, `parse_json_array()`
    - User-Defined Functions
- Data types
    - `JSONOBJECT`, `JSONARRAY`
    - `BAG` type parameters
- Other
    - In `INSERT`: Optional vertex type information for from- and to- vertices when inserting edges
    - Using `_` or `ANY` to specify any source vertex type in a `FROM` clause
    - Having the same name for global accumulators, global variables, and query parameters
        - For example, there cannot be a query parameter named `param` and a global accumulator named `@@param` within the same query

RESTful API User Guide

Introduction

TigerGraph's REST API endpoints exist on the REST++ and the GSQL server. REST++ (or RESTPP) is the TigerGraph customized REST ↗ server. Our API accepts URL-encoded query string parameters, JSON ↗-encoded request bodies and returns JSON encoded responses. This user guide provides information on how to engage with our REST APIs: the introduction section explains how to send requests, pass parameters, and format request bodies, while the subsequent sections describe in detail each endpoint and its input and output.

To submit a request, send an HTTP request to the REST++ server or the GSQL server. By default, the REST++ server listens for requests at port 9000 and the GSQL server listens on port 14240. A request needs to specify five things:

- The request method (`GET`, `POST`, `PUT`, or `DELETE`)
- The endpoint address
- Any required or optional request parameters in URL encoding
- For `POST` requests, a data payload (i.e., request body) in JSON encoding unless otherwise specified
- In some cases, request header parameters

For requests that are sent to the GSQL server, the sender needs to provide TigerGraph user credentials for the request to be accepted. If authentication is enabled on the RESTPP server, a request token needs to be included in the request header as well.

> In a test or development environment, the requester may be on the same server as REST++. In this case, the server IP is **localhost**.

Query String Parameters

TigerGraph's API endpoints accept parameters in URL encoding, which is straightforward in the case of string, number, and boolean values. However, some parameters are more complex and require specific formatting. The list below describes how to format the complex type parameter values when executing a query.

Parameter type	Description	Example
`SET` or `BAG` of primitives	Assign multiple values to the same parameter name.	A `SET<INT>` parameter named `p1` is assigned 3 integers: **`p1=1&p1=5&p1=10`**
`VERTEX` with a defined type	Use the primary key of the vertex.	A `VERTEX<person>` parameter named `vp` is assigned a vertex whose ID is `"person2"`: **`vp=person2`**
`VERTEX` without a defined type	Use `<parameter_name>.<parameter_type>` to specify the type of the parameter, and also provide the primary key of the vertex.	A `VERTEX` parameter named `va` is assigned a `person` type vertex whose ID is `"person1"` **`va=person1&va.type=person`**
`SET` or `BAG` of vertices with a defined type	Same as a `SET` or `BAG` of primitives, where the value for each element is the primary key of the vertex.	A `SET<VERTEX<person>>` parameter named `vp` is assigned two vertices with IDs `person3` and `person4`: **`vp=person3&vp=person4`**
`SET` or `BAG` of vertices without a defined type	The `SET` or `BAG` must be treated as an array, specifying the order of the elements with indices `[0]`, `[1]`, etc.	A `SET<VERTEX>` parameter named `vp` is assigned a `person` type vertex with an ID of `person1` and a `post` type vertex with an ID of `11` : **`vp[0]=person1&vp[0].type=person&vp[1]=11&vp[1].type=post`**

Input Data for `POST` requests

Input data for `POST` requests should be in JSON format, unless the endpoint specifically accepts data in other formats. There are two ways to supply the data: inline or in a separate file.

Inline Data

The data should be formatted as a single string without linebreaks. If using curl, use the `-d` option, followed by the JSON string.

Syntax for a POST request with Inline Data Payload

```
curl -X POST -d 'json_string' "http://server_ip:9000/path_to_endpoint?req
```

The following example uses the `POST /graph` endpoint to insert one User type vertex whose ID is `"id6`" into the graph called `"socialNet"`.

Example using inline input data

```
curl -X POST -d '{"vertices":{"User":{"id6":{"id":{"value":"id6"}}}}}' "h
```

Data File

Often it will be more convenient for the input data to be in a separate file, especially if the data is large.

For curl, use `--data-binary <path_to_file>` as in the example below:

Syntax for a POST request with Payload Data File

```
curl -X POST --data-binary <path_to_file> "http://server_ip:9000/path_to_
```

Formatting Data in JSON

Data of primitive types, including `INT, UINT, FLOAT, DOUBLE, STRING, BOOL`, and `DATETIME`, as well as arrays and objects, follow the standard JSON Data Interchange Syntax. This subsection describes how to format complex data types.

Data type	Description	Example
`SET`, `LIST` or `BAG` of primitives	Use a JSON array of primitive values.	A set of primitive values: `[0.5, 0,6, 0.7]`
`VERTEX`	Use a JSON object that has an `id` field whose value is the primary key of the vertex and a `"type"` field whose value is the type of the vertex.	A `person` vertex with an ID of `"Tom"`: `{"id": "Tom", "type": "person"}`

`MAP`	Use a JSON object that has two JSON arrays with keys `keyList` and `valueList`, each containing the keys and the values of the map respectively.
	
	The order of items in the valueList should correspond to the order of items in the keyList.
		A map of nations and their capitals:
		`{`
		`"keyList": ["England", "Germany"],`
		`"valueList": ["London", "Berlin"]`
		`}`

User-Defined Type (UDT)	Use a JSON object that has two JSON arrays with keys `keyList` and `valueList`. The `keyList` array contains the field names of the tuple, and the `valueList` array contains the values of the fields. The order of items in the valueList should	Tuple schema definition: `TYPEDEF TUPLE <name STRING, age INT> person` A `person` tuple written in JSON: `{"keyList: ["name", "age"], "valueList": ["Sam", 24]}`

correspond to the order of items in the keyList

Example: UDT Definition

```
TYPEDEF TUPLE <field1 INT(1), field2 UINT, field3 STRING(10), field4
```

Vertices with Composite Keys

If a vertex has a composite key composed of N attributes, then N values must be provided for the `"id"` . The values can be presented either as a JSON object with N key-value pairs, or as a JSON array with a list of N values in the same order as defined in the schema.

The example below shows the two methods for a vertex `v` having a composite primary key composed of the three attributes `id`, `name`, and `label.`

Option1

Vertex v with composite key as JSON object

```
{
  "v": {
    "id": {
      "id": 3,
      "name": "c",
      "label": 300
    },
    "type": "v3"
  }
}
```

Option2

Vertex v with composite key as JSON array

```
{
  "v": {
    "id": {
      "id": [3, "c", 300] // The values in the array must be in the s
                          // order as they are defined in the schema
    },
    "type": "v3"
  }
}
```

`SET` or `BAG` of Vertices

To describe a `SET` or `BAG` of vertices in JSON, use a JSON array with *vertex objects* nested in the `SET` or `BAG` array.

Output Responses

All TigerGraph REST responses are in JSON format. The output JSON object has four fields: `"version"`, `"error"`, `"message"`, and `"result"`.

- `"version"` - this field describes the version of the running TigerGraph instance.
- "`error"` - a boolean value to indicate if there is an error in processing the request. If there is an error, the `"error"` field will be `true`.
- `"message"` - the error message when there is an error. If a request is successful, the field will be an empty string or a brief message conveying the result of the request.
- `"results"` - this field contains the resulting data from the request. Details about the result of each built-in endpoint are described in the Built-in Endpoints section.

```
// Example response
{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "id1",
      "v_type": "User",
      "attributes": {}
    }
  ]
}
```

To make the JSON output more human-readable in the terminal, use the `jq` ↗ command or Python json library built into most Linux installations:

```
curl -X method "http://server_ip:9000/path_to_endpoint?request_par
curl -X method "http://server_ip:9000/path_to_endpoint?request_par
```

Size and Time limits

The maximum length for the request URL is 8K bytes, including the query string. Requests with a large parameter size should use a data payload file instead of inline data.

Request Body Size

The maximum size for a request body, including the payload file, is set by the system parameter `Nginx.ClientMaxBodySize`. The default value is 200 (in MB). To increase this limit, use the following `gadmin` command:

```
gadmin config set Nginx.ClientMaxBodySize NNN
```

The upper limit of this setting is 1024 MB. Raising the size limit for the data payload buffer reduces the memory available for other operations, so be cautious about increasing this limit.

GSQL Query Timeout

By default, an HTTP request in the TigerGraph system times out after 16 seconds. to customize this timeout limit for a particular query instance, you can set the GSQL-TIMEOUT parameter in the request header. If you are using curl to submit your RESTPP request, the syntax would be the following:

```
curl -X <GET/POST> -H "GSQL-TIMEOUT: <timeout value in ms>" '<request_URL
```

Response Size

You can specify the response size limit of an HTTP request with the following header:

```
curl -X <GET/POST> -H "RESPONSE-LIMIT: <size limit in byte>" '<request_UR
```

If the response size is larger than the given limit, an error message will be returned instead of the actual query result:

```
{
  "error": true,
  "message": "The query response size is 256MB, which exceeds limit 32MB.
  "results": [],
  "code": "REST-4000"
}
```

`curl` options

Request examples in this guide are made using `curl`. Below is a list of `curl` options used in our code examples:

- `-d <data>`
  - Sends the specified data in a `POST` request to the HTTP server in the same way that a browser does when a user has filled in an HTML form and presses the submit button. This will cause curl to pass the data to the server using the content-type `application/x-www-form-urlencoded`.
  - If you start the data with the character `@`, the rest should be a filepath from which to read the data. The command `curl -d @foobar` will read data from a file named `foobar`.
- `--data-binary <data>`
  - Sends data with a `POST` request exactly as specified with no extra processing.
- `--fail`
  - Makes curl fail silently (no output at all) on server errors.
  - This is mostly done to enable scripts etc. to better deal with failed attempts. In normal cases when an HTTP server fails to deliver a document, it returns an HTML document stating so (which often also describes why and more). This flag will prevent curl from outputting that and return error 22.
- `-H <header>`
  - Extra header to include in the request when sending HTTP to a server. You may specify any number of extra headers.
  - TigerGraph APIs use headers to specify size and time limits, as well as to provide RESTPP authentication tokens.
- `-s`
  - Silent or quiet mode. Don't show a progress meter or error messages. It will still output the data you ask for, potentially even to the terminal/stdout unless you redirect it.
- `-u <user:password>`
  - Submits the specified user name and password for server authentication.
- `-X <request_method>`
  - Specifies a custom request method to use when communicating with the HTTP server. If this option is not used, curl will make a `GET` request by default.

Authentication

TigerGraph's RESTful APIs communicate with either the REST++ server on port 9000 or the GSQL server on port 14240. Each server uses different methods of authentication.

REST++ Server Requests

By default, TigerGraph REST++ endpoints are public: anyone with access to the HTTP ports of the TigerGraph server can run your endpoints. When REST++ authentication is enabled, then a valid authorization token must be included in the header.

The REST++ server implements OAuth 2.0-style authorization as follows:

1. Each user can create one or more **secrets** (unique pseudorandom strings). Each secret is associated with a particular user and the user's privileges for a particular graph.
2. Anyone who has this secret can invoke a special REST endpoint to generate authorization **tokens** (other pseudorandom strings).
3. An authorization token can then be used to perform TigerGraph database operations via other REST endpoints. Each token will expire after a certain period of time. The TigerGraph default lifetime for a token is 1 month.

Using Tokens

Once REST++ authentication is enabled, a valid token should always be included in the HTTP header. If you are using curl to format and submit your REST++ requests, then use the following syntax:

curl GSQL request, with authorization token in header

```
curl -X GET -H "Authorization: Bearer <token>" '<request_URL>'
```

Here is a sample request:

```
curl -X GET -H "Authorization: Bearer 01234567abcdefgh01234567abcdefgh" "
```

GSQL Server Requests

All requests made to the GSQL Server must be accompanied by your user credentials. You can use the `curl -u` option to pass in the username and password of the TigerGraph user:

```
curl -X GET -u <username>:<password> "http://localhost:14240/<path_to_end
```

To keep your credentials more secure, one way to avoid having the user name and password on the command line is to instead use a .netrc file ↗ or a config file ↗. If you are in interactive mode, you can also use the `-u` option without specifying the password, and then curl will prompt for the password.

Built-in Endpoints

System Utilities

Echo (public)

`GET /echo` and `POST /echo`

These endpoints provide simple diagnostic utilities, which respond with the following message if the RESTPP server is up and running. Neither `GET /echo` or `POST /echo` require authentication, even when RESTPP authentication is enabled.

Sample request:

GET echo/ Request and Response

```
curl -X GET "http://localhost:9000/echo"
{
    "error": false,
    "message": "Hello GSQL"
}
```

`POST /echo` has the same response as `GET /echo`.

Parameters

Name	Required	Description
`sleep`	No	Integer that indicates the number of seconds for which the response will be delayed.

Health check (public)

This endpoint performs a simple server health check. It listens on port 14240 and does not require authentication. If you ping it and the server is running, it will respond with the message "pong".

Endpoint:

`GET /api/ping`

Sample request:

```
$ curl 'http://localhost:14240/api/ping'

{
  "error": "false",
  "message": "pong",
  "results": {}
}
```

Parameters:

No parameters.

List all endpoints

`GET /endpoints/{graph_name}`

This endpoint returns a list of the installed endpoints and their parameters. There are three types of endpoints:

- *Built-in endpoints* which are preinstalled in the TigerGraph system
- *Dynamic endpoints* which are generated when compiling GSQL queries
- *Static endpoints* which are user-installed endpoints

To include one or more of the endpoint types in the output, include the endpoint type in the parameter query string and set its value to `true`. If no type parameters are provided, all endpoints are returned.

Example: Report on all built-in endpoints

```
curl -X GET "http://localhost:9000/endpoints?builtin=true" | jq .
```

Response

There are over a dozen built-in endpoints, and some have several parameters, so the formatted JSON output of all built-in endpoints is over 300 lines long. It is listed in full in Appendix A. Below is a small excerpt of the output:

Subset of GET /endpoints output

```
    "GET /endpoints": {
        "parameters": {
            "builtin": {
                "default": "false",
                "max_count": 1,
                "min_count": 0,
                "type": "BOOL"
            },
            "dynamic": {
                "default": "false",
                "max_count": 1,
                "min_count": 0,
                "type": "BOOL"
            },
            "static": {
                "default": "false",
                "max_count": 1,
                "min_count": 0,
                "type": "BOOL"
            }
        }
    }
```

Parameters

Name	Required	Description
`builtin`	No	Takes a boolean value. Returns built-in endpoints if true.

`dynamic`	No	Takes a boolean value. Returns dynamic endpoints if true.
`static`	No	Takes a boolean value. Returns user-installed endpoints if true

Show component versions

`GET /version`

This endpoint returns the GIT versions of all components of the system.

Sample request:

```
curl -X GET "http://localhost:9000/version"
{"error":"false", "message":"TigerGraph RESTPP:
 --- Version ---
product              release_2.6.0_05-09-2020 ab1e3d0da6237c27468d6cabb90
olgp                 release_2.6.0_05-09-2020 046c745088106b69920b9bdb3bd
topology             release_2.6.0_05-09-2020 c028af100117f2051b619436c3a
gpe                  release_2.6.0_05-09-2020 34b9e86ef7b5fdaa106637e7db1
gse                  release_2.6.0_05-09-2020 ed2c2351357aa9077fa4dee7ea7
third_party          release_2.6.0_05-09-2020 4bce6990bae5be2b91e9201693c
utility              release_2.6.0_05-09-2020 2ce197d3edb3557bdd66ed1a419
realtime             release_2.6.0_05-09-2020 52a82b454437c73b47d846acd58
er                   release_2.6.0_05-09-2020 a3e6cb7606fb74984c75cae9bbd
gle                  release_2.6.0_05-09-2020 d8bdbd1cf346e181aa9a317c704
bigtest              release_2.6.0_05-09-2020 2f64c47b7a5ac1834ead9a22eef
document             release_2.6.0_05-09-2020 6327094bd76b2dbc8f4625108d5
glive                release_2.6.0_05-09-2020 93f61ea06fe42759c808fc58ff6
gap                  release_2.6.0_05-09-2020 e798efb595545bf91c449034566
gst                  release_2.6.0_05-09-2020 1b695c02f277efad0ddfb2deab7
gus                  release_2.6.0_05-09-2020 eee784502b5387844e462305bae
blue_features        release_2.6.0_05-09-2020 5d7a4e8d806519f529274b33149
blue_commons         release_2.6.0_05-09-2020 432763afc49bf986aed4731e502
"}
```

Parameters

This endpoint does not take any parameters.

Monitor system metrics

`POST /ts3/api/datapoints`

TigerGraph System State Service (TS3) is TigerGraph's managed monitoring service that collects system metrics and events. Many TigerGraph components will report metrics such as CPU usage, memory usage, disk usage, and network traffic to TS3 at regular intervals. You can use this endpoint to read from TS3, filtering for the data points you need by time (`when`, `from`, and `to`), component(`who`), metric(`what`) and location(`where`). Visualization of such metrics are available in Admin Portal - Dashboard - Cluster Monitoring.

On a TigerGraph cluster, this endpoint is only present on the `m1` node.

Parameters

Name	Required	Description
`from`	No	Epoch timestamp that indicates the start of the time filter. Only data points reported after the timestamp will be included in the return results.
`to`	No	Epoch timestamp that indicates the end of the time filter. Only data points reported before the timestamp will be included in the return results.
`latest`	No	Number of latest data points to return. If provided, the endpoint will return the latest data points that satisfy the `what`, `who` and

		`where` filters and ignore other time-related filters.
`what`	No	Name of the metric to filter for. Possible values are: • `cpu`: Percentage of CPU usage by component • `mem`: Memory usage in megabytes by component • `diskspace`: Disk usage in megabytes by directory • `network`: Network traffic in bytes since the service started • `qps`: Number of requests per second by endpoint • `servicestate`: Whether or not the service is online. A value of `0` indicates that the service is offline while a value of `1` means the service is online • `connection`: Number of open TCP connections
`who`	No	Name of the component that reported the datapoint
`where`	No	Name of the node that the datapoint is reported for

Sample requests

In the sample request below, the filters in the query string include a timeframe starting at `1618957536` and ending at `1619023346`, and specifying that the response should only include CPU information:

```
$ curl -X GET
"https://crunch.i.tgcloud.io:14240/ts3/api/datapoints?from=1618957536&to=

# Three data points returned
[
  {
    "detail": 0,  # GPE is using 0 percent CPU
    "when": 1619023346,
    "where": "m1",
    "who": "GPE_1#1",
    "what": "cpu"
  },
  {
    "detail": 0,
    "when": 1619023346,
    "where": "m1",
    "who": "GSE_1#1",
    "what": "cpu"
  },
  {
    "detail": 0,
    "when": 1619023346,
    "where": "m1",
    "who": "RESTPP#1",
    "what": "cpu"
  }
]
```

In the below example, the request asks for the 10 latest data points regarding memory usage:

```
$ curl -X GET
"https://crunch.i.tgcloud.io:14240/ts3/api/datapoints?what=mem&latest=10"

[
  {
    "detail": 159,
    "when": 1620076473,
    "where": "m1",
    "who": "RESTPP#1",
    "what": "mem"
  },
  {
    "detail": 211,
    "when": 1620076533,
    "where": "m1",
    "who": "GPE_1#1",
    "what": "mem"
  },
  {
    "detail": 436,
    "when": 1620076533,
    "where": "m1",
    "who": "GSE_1#1",
    "what": "mem"
  },
  {
    "detail": 159,
    "when": 1620076533,
    "where": "m1",
    "who": "RESTPP#1",
    "what": "mem"
  },
  {
    "detail": 211,
    "when": 1620076593,
    "where": "m1",
    "who": "GPE_1#1",
    "what": "mem"
  },
  {
    "detail": 436,
    "when": 1620076593,
    "where": "m1",
    "who": "GSE_1#1",
    "what": "mem"
  },
  {
    "detail": 159,
```

```
      "when": 1620076593,
      "where": "m1",
      "who": "RESTPP#1",
      "what": "mem"
    },
    {
      "detail": 210,
      "when": 1620076653,
      "where": "m1",
      "who": "GPE_1#1",
      "what": "mem"
    },
    {
      "detail": 436,
      "when": 1620076653,
      "where": "m1",
      "who": "GSE_1#1",
      "what": "mem"
    },
    {
      "detail": 159,
      "when": 1620076653,
      "where": "m1",
      "who": "RESTPP#1",
      "what": "mem"
    }
  ]
```

Show query performance

`GET /statistics/{graph_name}`

This endpoint returns real-time query performance statistics over the given time period, as specified by the `seconds` ****parameter. The `seconds` parameter must be a positive integer less than or equal to 60.

Sample request:

The return object is a hash of the endpoints and their performance data:

```
# The example shows two endpoints (/graph/vertex and
# /statistics) called during the past 60 seconds.
curl -X GET "http://localhost:9000/statistics/poc_graph?seconds=60" | jq

{
  "GET /graph/vertices/{vertex_type}/{vertex_id}": {
    "CompletedRequests": 8,
    "QPS": 0.08,
    "TimeoutRequests": 0,
    "AverageLatency": 130,
    "MaxLatency": 133,
    "MinLatency": 128,
    "LatencyPercentile": [
      200,
      200,
      200,
      200,
      200,
      200,
      200,
      200,
      200,
      200
    ]
  },
  "GET /statistics": {
    "CompletedRequests": 4226,
    "QPS": 42.26,
    "TimeoutRequests": 0,
    "AverageLatency": 2,
    "MaxLatency": 125,
    "MinLatency": 0,
    "LatencyPercentile": [
      10,
      10,
      10,
      10,
      10,
      10,
      10,
      10,
      10,
      200
    ]
  }
}
```

Each endpoint has the following attributes:

- `CompletedRequests` - the number of completed requests.
- `QPS` - query per second.
- `TimeoutRequests` - the number of requests not returning before the system-configured timeout limit. Timeout requests are not included in the calculation of QPS.
- `AverageLatency` - the average latency of completed requests.
- `MaxLatency` - the maximum latency of completed requests.
- `MinLatency` - the minimum latency of completed requests.
- `LatencyPercentile` - The latency distribution. The number of elements in this array depends on the `segments` ****parameter of this endpoint whose default value is 10, meaning the percentile range 0-100% will be divided into ten equal segments: 0%-10%, 11%-20%, etc. `Segments` ****must be [1, 100].

If there is no query sent in the past given seconds, an empty json will be returned.

Parameters

Name	Required	Description
`seconds`	Yes	Positive integer less than 60 that indicates how many seconds back from the current time the statistics report will cover.
`segments`	No	Integer that indicates the number of segments that `LatencyPercentile` array in the response will be split into. The value for this endpoint must be between 1 and 100 and has a default value of 10.

Rebuild graph engine

`GET /rebuildnow/{graph_name}` or `POST /rebuildnow/{graph_name}`

In TigerGraph, when new data is being loaded into the graph (such as new vertices or edges), data is first stored in memory before it is saved to disk permanently. TigerGraph runs a rebuild of the Graph Processing Engine (GPE) to commit the data in memory to disk every 30 seconds, but you can also call this endpoint to trigger a rebuild immediately.

Parameters:

Name	Required	Description
`threadnum`	No	Number of threads used to execute the rebuild. If not specified, the number specified in line 185 of the `.tg.cfg` file (`"RebuildThreadNumber"`) in the home directory of the server on which TigerGraph is running will be used; it is set to 3 by default. The maximum value for this parameter is the number of vCPUs per node in your distributed system. If you are running a single-node server, the maximum is the number of vCPUs on that node. You can run `lscpu` in the command line of your Linux server and look in the `CPU(s)` column to view the number of vCPUs.
`vertextype`	No	Vertex type to perform the rebuild for. If not provided, the rebuild will be run for all the vertex types.
		Segment ID of the segments to rebuild. If not

`segid`	No	provided, all segments will be rebuilt. In general, it is recommended not to provide this parameter and rebuild all segments.
`path`	No	Path to save the summary of the rebuild to. If not provided, the default path is `/tmp/rebuildnow`
`force`	No	Boolean value that indicates whether to perform rebuilds for segments for which there are no records of new data. Normally, a rebuild would skip such segments, but if `force` is set true, the segments will not be skipped.

Example:

```
$ curl -X GET 'http://localhost:9000/rebuildnow/social'

# JSON response
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "RebuildNow finished, please check details in the folder: /t
  "results": [],
  "code": "REST-0000"
}

# Example summary file
$ cat finished.summary.txt

[SELECTED]  Segment id: 106, vertextype: 0, vertexsubtypeid: 0, vertexcou
[SKIPPED]   Segment id: 6, vertextype: 0, vertexsubtypeid: 0, vertexcount
```

Check deleted vertices

~~`GET /deleted_vertex_check`~~

In certain rare cases, TigerGraph's Graph Processing Engine (GPE) and Graph Storage Engine (GSE) might be out of sync on vertex deletion information. When this happens, some vertices might exist on one of the components, but not the other. Even though these errors are exceedingly rare, TigerGraph provides an endpoint that allows you to check the deleted vertices on GSE and GPE and see if they out of sync.

The check passes if there are no discrepancies between the GSE and GPE in terms of deleted vertices. If there is a discrepancy, the check fails and the return result will contain the IDs of the deleted vertices that are not synced properly. If you are running TigerGraph on a distributed cluster, the check will be performed on each node of the cluster, and the endpoint will return a list containing the results of the check for every node.

Parameters:

Name	Required	Description
`threadnum`	No	Integer that indicates the number of threads used to execute the deleted vertex check jobs. This parameter is optional and the default value is 6 if none is provided
`segid`	No	IDs of segments to perform the deleted vertex check for. If none is provided, the check will be performed on all segments.
`vertextype`	No	Vertex types to perform the deleted vertex check for. If none is provided, the check

		will be performed on all vertex types.
`verbose`	No	Integer that indicates the level of detail in the return results. Here is a list of accepted values and their corresponding level of detail: • `0` (default) : Only return whether the check passed and the list of unsynced vertex IDs • `1`: In addition to the previous level, also return vertex count information • `2`: In addition to the previous level, return vertex count information for every segment • `4`: In addition to the previous level, also return the IDs of deleted vertices for every segment
`log`	No	Integer that indicates the log level of the deleted vertex check. This log is not returned in the endpoint's HTTP response, but is printed to the logs of the GPE component at `/tigergraph/log/gpe/log.INFO`: • `0` (default): Report brief log for the check as a whole • `1`: Report logs for each segment

- `2`: Report additional logs on the obtained

Example:

```
# Passing check performed on a single-node database
$ curl -X GET "http://localhost:9000/deleted_vertex_check?threadnum=10&ve

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "check passed",
  "results": [
    {
      "GPE": "GPE_1_1",
      "PassCheck": true,
      "UnSyncList": []
    }
  ],
  "code": "REST-0000"
}

# Failed check performed on a distributed cluster

$ curl -X GET 'http://localhost:9000/deleted_vertex_check?threadnum=10&ve
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "check failed",
  "results": [
    {
      "GPE": "GPE_2_1",
      "PassCheck": false,
      "UnSyncList": [
        {
          "Segid": 193,
          "IsRemote": false,
          "VertexType": "region",
          "GPEDelHash": 7013042118817697000,
          "IDSDelHash": 202375168
        }
      ]
    },
    {
      "GPE": "GPE_3_1",
```

```
      "PassCheck": false,
      "UnSyncList": [
        {
          "Segid": 193,
          "IsRemote": true,
          "VertexType": "region",
          "GPEDelHash": 7013042118817697000,
          "IDSDelHash": 202375168
        }
      ]
    },
    {
      "GPE": "GPE_1_1",
      "PassCheck": false,
      "UnSyncList": [
        {
          "Segid": 193,
          "IsRemote": true,
          "VertexType": "region",
          "GPEDelHash": 7013042118817697000,
          "IDSDelHash": 202375168
        }
      ]
    }
  ],
  "code": "REST-0000"
}
```


Authentication

The endpoints in this subsection allow users to create, refresh and delete authentication tokens for requests made to the REST++ server. **These endpoints only exist when user authentication is enabled on RESTPP endpoints.**

Request a token (`GET`)

`GET /requesttoken`

If authentication is enabled on RESTPP endpoints, a token needs to be included in the request header for all requests sent to the RESTPP server. A user can generate a token using either

- A secret, which is a random string generated in GSQL (see Managing User Privileges and Authentication)
- Their username and password in their request header as well as specifying the graph

You may also use a `POST` request to generate your token, see Request a token (POST).

Sample request:

```
curl -X GET "http://localhost:9000/requesttoken?secret=jiokmfqqfu2f95qs6u
{
  "code": "REST-0000",
  "expiration": 1616042814,
  "error": false,
  "message": "Generate new token successfully.",
  "token": "tohvf6khjqju8jf0r0l1cohhlm8gi5fq"
}

curl --user example_username:example_password -X GET "localhost:9000/requ
```

Parameters:

Name	Required	Description
`secret`	Yes if `graph` is not supplied	User's secret to generate the token.
`lifetime`	No	Period of time for which the token is valid measured in seconds. The default value is about 2.6 million (about a month).
`graph`	Yes if `secret` is not supplied	Name of the graph that the token will be valid for.

Users can use either `secret` or their username and password to generate a token. If the user does not supply a secret and chooses to use their username and password instead, then the parameter `graph` becomes required.

Request a token (`POST`)

`POST /requesttoken`

If authentication is enabled on RESTPP endpoints, a token needs to be included in the request header for all requests sent to the RESTPP server. A user can generate a token using either

- A secret, which is a random string generated in GSQL (see Managing User Privileges and Authentication)
- Their username and password in their request header as well as specifying the graph

You may also use a `GET` request to generate your token, see Request a token (GET).

Sample requests

Request

```
curl -d <path_to_secret> -X POST \
"http://localhost:9000/requesttoken?lifetime=1000000"
```

Response

```
{
  "code": "REST-0000",
  "expiration": 0,
  "error": false,
  "message": "Refresh token successfully.",
  "token": "tohvf6khjqju8jf0r0l1cohhlm8gi5fq"
}
```

Replace `path_to_secret` with the path to the file containing your secret. The file should only include a single line, which is your secret.

Request

```
curl --user tigergraph:tigergraph \
-X POST '127.0.0.1:9000/requesttoken?graph=gsql_demo'
```

Response

```
{
    "error":false,
    "message":"Request token successfully.",
    "results":{"token":"o5pn931drmppjasi2vjidrlf6rp4n4po"}
}
```

You can also use a username-password pair to generate a token. In this case, you must also specify the graph you want to generate the token for.

Parameters:

Name	Required	Description
`lifetime`	No	Period of time for which the token is valid in seconds. The default value is about 2.6 million (about a month).

	Yes if using username-	The graph for which to

Refresh a token

`PUT /requesttoken`

This endpoint takes a token and its associated secret and refreshes the lifetime of the token. The token itself remains unchanged.

Parameters:

Name	Required	Description
`token`	Yes	Token to refresh.
`secret`	Yes	User's secret used to generate the token.
`lifetime`	Yes	Period of time for which the token is valid measured in seconds.

Sample request

```
curl -X PUT "http://localhost:9000/requesttoken?lifetime=15&secret=ksdoil
{
  "code": "REST-0000",
  "expiration": 15,
  "error": false,
  "message": "Refresh token successfully.",
  "token": "0mq98l9pderkaivndf820gudg923p3l0"
}
```

Delete a token

`DELETE /requesttoken`

This endpoint takes a token and its associated secret, and deletes the token.

Parameters:

Name	Required	Description
`token`	Yes	Token to delete.
`secret`	Yes	User's secret used to generate the token.

Loading jobs

Run a Loading Job

`POST /ddl/{graph_name}`

This endpoint is for loading data into a graph. It submits data as an HTTP request payload, to be loaded into the graph by the DDL Loader. The data payload can be formatted as generic CSV or JSON. For more details, please see GSQL Language Reference Part 1 - Defining Graphs and Loading Data.

If the loading job references multiple files, multiple HTTP requests are needed to complete the loading job since you can only provide data for one filename varibale at a time. The loading job will skip the `LOAD` statements referencing filename variables that the request didn't provide data for. To provide data for a filename variable, put the data in the request body and use the `filename` parameter (explained in the parameter table below) to match the variable name defined in the loading job.

If a `LOAD` statement is written using a filepath string instead of a file variable, even though the filepath is already provided in the loading job, you still need to provide data in the request body for the `LOAD` statement to run. Since there isn't a file variable in this case, use a position-based file identifier to identify the filepath string you are providing data for in the `filename` parameter.

Request body:

The request body is the data to be loaded (either in CSV or JSON format).

Curl allows you to read the data from an input file by using the @ symbol:

```
curl -X POST --data-binary @./company.csv "http://…"
```

Sample request:

In this example, the loading job is dependent on three filename variables (`f1` and `f3`) and one filepath string. Therefore, three HTTP requests are needed to complete the loading job.

```
# Loading job
CREATE LOADING JOB load_data for GRAPH poc_graph {

    DEFINE FILENAME f1;
    DEFINE FILENAME f3;

    LOAD f1 to VERTEX person VALUES ($0, $0);
    LOAD "/home/data/company.csv" to VERTEX company VALUES ($0, $0);

    LOAD f3 to EDGE work_at VALUES ($0, $1, $3, $4, $5);
}

# Provide data for for the second LOAD statement
curl -X POST --data-binary @./another_company.csv \
"http://localhost:9000/ddl/poc_graph?tag=load_data&filename=__GSQL_FILENA
```


```
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "sourceFileName": "Online_POST",
      "statistics": {
        "validLine": 7927,
        "rejectLine": 0,
        "failedConditionLine": 0,
        "notEnoughToken": 0,
        "invalidJson": 0,
        "oversizeToken": 0,
        "vertex": [
          {
            "typeName": "company",
            "validObject": 7,
            "noIdFound": 0,
            "invalidAttribute": 0,
            "invalidPrimaryId": 0,
            "invalidSecondaryId": 0,
            "incorrectFixedBinaryLength": 0
          }
        ],
        "edge": [],
        "deleteVertex": [],
        "deleteEdge": []
```

```
        }
      }
    ],
    "code": "REST-0000"
  }

# Provide data for filename f1 for the first LOAD statement
curl -X POST --data-binary @./person.csv \
"http://localhost:9000/ddl/poc_graph?tag=load_data&filename=f1"

# Provide data for filename f3 for the third LOAD statement
curl -X POST --data-binary @./work_at.csv \
"http://localhost:9000/ddl/poc_graph?tag=load_data&filename=f3"
```

Parameters:

Name	Required	Description
`tag`	Yes	Loading job name defined in your DDL loading job
`filename`	Yes	File variable name or file path for the file containing the data
`sep`	No	Separator of CSV data. If your data is JSON, you do not need to specify this parameter. The default separator is a comma `","`
`eol`	No	End-of-line character. Only one or two characters are allowed, except for the special case "\r\n". The default value is `"\n"`
`ack`	No	`"all"`: request will return after all GPE instances have acknowledged the `POST` request. `"none"`: request will return immediately after RESTPP processed the `POST` request.

`timeout`	No	Timeout in seconds. If set to 0, use system-wide endpoint timeout setting.
`concise`	No	Boolean value that indicates whether to return concise results of the data loading request. Concise results will only include the number of vertices and edges added or deleted, and will omit information such as the number of valid and invalid lines in the default response

If there are special characters in your parameter values, the special characters should use URL encoding ↗. To avoid confusion about whether you should you one or two backslashes, we do not support backslash escapes for the `eol` or `sep` parameter.

The maximum size of data you can upload via this endpoint is controlled by the `Nginx.ClientMaxBodySize` configuration parameter (default is 200 MB).

Graphs

Run built-in functions on graph

`POST /builtins/{graph_name}`

This endpoint runs a set of built-in functions and returns relevant statistics about a graph.

Request body:

This endpoint expects a data payload in the request body that specifies which function to run on the graph. Depending on the function being run, different fields may also be expected in the request body.

Here is a list of functions supported by this endpoint and their corresponding data payload format.

- `stat_vertex_attr`
  - Returns the minimum, maximum, and average values of the given vertex type's `int`, `uint`, `float` and `double` attributes, and the count of `true` and `false` of a boolean attribute.
  - Data payload fields:
    - `"function": "stat_vertex_attr"`: This specifies that the function to run is `stat_vertex_attr`.
    - `"type"`: The vertex type whose attribute values to report on. Required field. It also accepts the value `*` (wild card), in which case, all vertex types are included.
- `stat_edge_attr`
  - Returns the minimum, maximum, and average values of the given edge type's `int`, `uint`, `float` and `double` attributes, and the count of `true` and `false` of a boolean attribute.
  - Data payload fields:
    - `"function": stat_edge_attr`
    - `"type"`: The edge type whose attribute values to report on. Required field. It also accepts the value `*` , in which case all edge types are included.
    - `"from_type"`: Optional. The source vertex type of the edges to report on.
    - `"to_type"`: Optional. The target vertex type of the edges to report on.
- `stat_vertex_number`
  - Returns the number of vertices of the given vertex type.
  - Data payload fields:
    - `"function"`: `"stat_vertex_number"`
    - `"type"`: Required field. The vertex type of the vertices to count. It also accepts the value `*` (wild card), in which case, all vertex types are included.

- `stat_edge_number`
  - Returns the number of edges of the given edge type
  - Data payload fields:
    - `"function": "stat_edge_number"`
    - `"type"`: Required field. The edge type of the edges to count. It also accepts the value `*` , in which case all edge types are included.
    - `"from_type"`: Optional. The source vertex type of the edges to report on.
    - `"to_type"`: Optional. The target vertex type of the edges to report on.

Sample requests:

Below is an example request running `stat_vertex_attr` on `socialNet` and its output. The vertex type `"Person"` has a `uint` attribute `"age"`.

```
curl -X POST "http://localhost:9000/builtins/socialNet" \
-d  '{"function":"stat_vertex_attr","type":"Person"}' | jq .

{
  "version": {
      "api": "v2",
      "schema": 0
   },
  "error": false,
  "message": "",
  "results": [
    {
      "vertexName": "Person",
      "attributeStat": [
        {
          "vattrName": "age",
          "MAX": 64,
          "MIN": 15,
          "AVG": 36.5
        }
      ]
    }
  ]
}
```

Here is an example request running `stat_edge_attr` on `socialNet` and its output. The edge type `"Liked"` has a float attribute `"strength"`.

```
curl -X POST "http://localhost:9000/builtins/socialNet" \
-d  '{"function":"stat_edge_attr","type":"Liked", "from_type":"*", "to_ty

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "Liked",
      "attributes": {
        "weight": {
          "MAX": 2.5,
          "MIN": 1,
          "AVG": 1.375
        }
      }
    }
  ]
}
```

Here is an example request running `stat_vertex_number` and its output.

```
curl -X POST "http://localhost:9000/builtins/socialNet" \
-d  '{"function":"stat_vertex_number","type":"*"}' | jq .

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_type": "User",
      "count": 4
    },
    {
      "v_type": "Page",
      "count": 4
    },
    {
      "v_type": "Product",
      "count": 7
    },
    {
      "v_type": "DescWord",
      "count": 7
    },
    {
      "v_type": "NameUser",
      "count": 9
    },
    {
      "v_type": "VidUser",
      "count": 4
    },
    {
      "v_type": "Video",
      "count": 5
    },
    {
      "v_type": "AttributeTag",
      "count": 4
    }
  ]
}
```

Parameters:

No parameters.

Show graph schema metadata

`GET /gsqlserver/gsql/schema`

Returns schema details about a vertex type, an edge type, or the entire graph schema. This is a GSQL Server request sent to port 14240, and authentication credentials need to be provided.

Sample request:

```
$ curl -u tigergraph:tigergraph \
"localhost:14240/gsqlserver/gsql/schema?graph=workNet&type=company"

{
  "error": false,
  "message": "",
  "results": {
    "Config": {
      "STATS": "OUTDEGREE_BY_EDGETYPE",
      "PRIMARY_ID_AS_ATTRIBUTE": false
    },
    "Attributes": [
      {
        "AttributeType": {
          "Name": "STRING"
        },
        "IsPartOfCompositeKey": false,
        "PrimaryIdAsAttribute": false,
        "AttributeName": "id",
        "HasIndex": false,
        "IsPrimaryKey": false
      },
      {
        "AttributeType": {
          "Name": "STRING"
        },
        "IsPartOfCompositeKey": false,
        "PrimaryIdAsAttribute": false,
        "AttributeName": "country",
        "HasIndex": false,
        "IsPrimaryKey": false
      }
    ],
    "PrimaryId": {
      "AttributeType": {
        "Name": "STRING"
      },
      "IsPartOfCompositeKey": false,
      "PrimaryIdAsAttribute": false,
      "AttributeName": "clientId",
      "HasIndex": false,
      "IsPrimaryKey": false
    },
    "Name": "company"
  }
}
```

***Vertex schema object* fields:**

- `Name` : the vertex type name, same as the input parameter "type"
- `PrimaryId` : details about the primary id
- `Attributes` : details about each attribute, listed in order
- `Config` : details about global properties of the vertex type

***Edge schema object* fields:**

- `Name` : the edge type name, same as the input parameter "type"
- `FromVertexTypeName` : source vertex type name
- `ToVertexTypeName` : target vertex type name
- `Attributes` : details about each attribute, listed in order
- `IsDirected` : whether the edge is directed
- `Config` : additional details about global properties of the edge type

***Graph schema object* fields:**

- `GraphName` : the graph name, same as the input parameter "graph"
- `VertexTypes` : an array of *vertex schema objects*. Each vertex schema object is exactly the JSON output if that specific vertex type had been specified.
- `EdgeTypes` : an array of *edge schema objects*. Each edge schema object is exactly the JSON output if that specific edge type had been specified.

```
{
  "error": false,
  "message": "",
  "results": {
    "GraphName": "workNet",
    "VertexTypes": [
      {
        "Config": {...},
        "Attributes": [...],
        "PrimaryId": {...},
        "Name": "person"},
      {
        "Config": {...},
        "Attributes": [...],
        "PrimaryId": {...},
        "Name": "company"}
    ],
    "EdgeTypes": [
      {
        "IsDirected": false,
        "ToVertexTypeName": "company",
        "Config": {},
        "Attributes": [...],
        "FromVertexTypeName": "person",
        "Name": "worksFor"
      }
    ]
  }
}
```


Parameters:

Name	Required	Description
`graph`	Yes	The name of the graph whose schema to retrieve.
`type`	No	The vertex or edge type whose details to retrieve. If not provided, the endpoint will provide a *graph schema object* containing the schema details of the entire graph.

Upsert data to graph

`POST /graph/{graph_name}`

This endpoint upserts vertices and/or edges into a graph. To upsert means that if a vertex or edge does not exist, it is inserted, and if it does exist, it is updated.

Atomic upsert transaction

By default, the `POST /graph/{graph_name}` endpoint is not atomic. If something goes wrong during the process of the request, the request data can be partially consumed by the database.

You can append a query string parameter `atomic_post` to the URL of the request and set its value to true to make the request an atomic transaction, which means that updates to the database contained in the request are all-or-nothing. Either all changes are successful, or none is successful.

For example, suppose we have the following request to upsert two vertices:

```
curl --data-binary @vertices.json http://localhost:9000/graph/social
```

And the content of `vertices.json` is:

```
{
 "vertices": {
    "person": {
      "Velma": {
        "age": {
           "value": 30
        }
      },
      "Kelly": {
        "age": {
           "value": 22
        }
      }
    }
  }
}
```

With the request above, if the vertex `Kelly` fails to be upserted due to a machine failure, it is still possible that the vertex `Velma` is upserted to the database.

If you add the `atomic_post` parameter to the request URL and set its value to true, the request becomes atomic and if any part of the request body fails to be upserted, nothing will be upserted:

```
# This is an atomic request
curl --data-binary @vertices.json http://localhost:9000/graph/social?auto
```

Parameters

Name	Required	Description
`ack`	No	The value of this parameter can either be `"all"` or `"none"`. `"all"`: request will return after all GPE instances have acknowledged the POST `"none"`: request will return immediately after RESTPP processed the POST.

`new_vertex_only`	No	Boolean value that indicates whether or not to update existing vertices. If the value is true, it will only insert new vertices and not update existing ones.
`vertex_must_exist`	No	Boolean value that indicates whether or not to insert new edges when the `FROM` or `TO` vertices don't exist. If the value is true, the request will only insert edge if both the `FROM` and `TO` vertices of the edge already exist. If the value is false, the request will always insert new edges, and create the necessary vertices with default values for their attributes.
`atomic_post`	No	Boolean value that indicates whether or not this request is an atomic transaction.

The response is the number of vertices and edges that were accepted. The API uses JSON format to describe the vertices and edges to be upserted. The JSON code can be stored in a text file or specified directly in a command line. There is a maximum size for a `POST` data payload (see the **Size Limits** ****section). The JSON format for describing a vertex set or edge set is summarized below.

Request body

The payload data should be in JSON according to the schema shown below:

Request bodyd schema

```
"vertices": {
   "<vertex_type>": {
      "<vertex_id>": {
         "<attribute>": {
            "value": <value>,
            "op": <opcode>
         }
      }
   }
},
"edges": {
   "<source_vertex_type>": {
      "<source_vertex_id>": {
         "<edge_type>": {
            "<target_vertex_type>": {
               "<target_vertex_id>": {
                  "<attribute>": {
                     "value": <value>,
                     "op": <opcode>
                  }
               }
            }
         }
      }
   }
}
```

The fields in angle brackets (`<>`) are placeholder names or values, to be replaced with actual values. The keys in angle brackets, such as `<vertex_type>`, can be repeated to form a list of items. The keys which are not in angle brackets are exact texts that must be used as they are. The nested hierarchy means that vertices are grouped by type. Edges, on the other hand, are first grouped by source vertex type, then vertex ID, then edge type.

The first example below shows two `User` vertices having an attribute called `age`:

Upsert Example Data 1: Two User vertices

```
  "vertices": {
     "User": {
       "id6": {
         "age": {
            "value": 30
          }
       },
       "id1": {
         "age": {
            "value": 22
          }
       }
     }
   }
 }
```

The second example starts with one `User` vertex. If `id6` already exists, it is not changed. If it doesn't yet exist, it is created with default attribute values. Then two edges are created: a `Liked` edge from `id1` to `id6`, and then a `Liked_By` edge from `id6` to `id1`.

Upsert Example Data 2:add:id6.json

```
{
 "vertices": {
    "User": {
      "id6": {
      }
    }
  },
  "edges": {
    "User":{
      "id1": {
        "Liked": {
          "User": {
            "id6" : {
              "weight" : {
                "value": 5.0
              }
            }
          }
        }
      },
      "id6": {
        "Liked_By": {
          "User": {
            "id1" : {
              "weight" : {
                "value": 1.0
              }
            }
          }
        }
      }
    }
  }
}
```


Follow the instructions in the Introduction section to format advanced data types. For example, the following payload is used to upsert two `User` vertices with an attribute `coordinates` of type `LIST` and an attribute `measurements` of type `MAP`:

```
{
 "vertices": {
    "User": {
      "id4": {
        "coordinates": {
           "value": [51.3345, -7.2233]
         },
        "measurements": {
           "value": {
             "keylist": ["chest", "waist", "hip"],
             "valuelist": [35, 30, 35]
           }
         }
      },
      "id5": {
        "coordinates": {
           "value": [31.3245, -17.3292]
         },
        "measurements": {
           "value": {
             "keylist": ["chest", "waist", "hip"],
             "valuelist": [39, 35, 41]
           }
         }
      }
    }
  }
}
```

Operation codes

Each attribute value may be accompanied by an operation (op) code, which provides very sophisticated schemes for data update or insertion:

Type	op	Meaning
1	`"ignore_if_exists"` or `"~"`	If the vertex/edge does not exist, use the payload value to initialize the attribute; but if the vertex/edge already exists, do not change this attribute.

2	`"add"` or `"+"`	Add the payload value to the existing value.
3	`"and"` or `"&"`	Update to the logical AND of the payload value and the existing value.
4	`"or"` or `"	"`
5	`"max"` or `">"`	Update to the higher value between the payload value and the existing value.
6	`"min"` or `"<"`	Update to the lower value between the payload value

If an attribute is not given in the payload, the attribute stays unchanged if the vertex/edge already exists, or if the vertex/edge does not exist, a new vertex/edge is created and assigned the default value for that data type. The default value is 0 for `int/uint`, 0.0 for `float/double`, and `""` (empty string) for string.

Invalid data types cause the request to be rejected

The RESTPP server validates the request before updating the values. The following schema violations will cause the entire request to fail and no change will be made to a graph:

- For vertex upsert
  - Invalid vertex type
  - Invalid attribute data type
- For edge upsert:
  - Invalid source vertex type
  - Invalid edge type
  - Invalid target vertex type
  - Invalid attribute data type.

If an invalid attribute name is given, it is ignored.

Output response

The response is the number of vertices and edges that were accepted. Additionally, if `new_vertex_only` is true, the response will include two more fields:

- `skipped_vertices`: the number of vertices in the input data which already existed in the graph
- `vertices_already_exist`: the id and type of the input vertices which were skipped

If `vertex_must_exist` is true, the response will include two more fields:

- `skipped_edges`: the number of edges in the input data rejected because of missing endpoint vertices
- `miss_vertices`: the id and type of the endpoint vertices which were missing

Examples

The example file `add_id6.json` (shown in the **Request Body** section) upserts one `User` vertex with `id = "id6"`, one `Liked` edge, and one `Liked_By` edge. The `Liked` edge is from `"id1` " to `"id6"`; the `Liked_By` edge is from `"id6"` to `"id1"`.

The following example submits an upsert request by using the payload data stored in `add_id6.json`.

```
$ curl -X POST --data-binary @add_id6.json \
"http://localhost:9000/graph"

{"accepted_vertices":1,"accepted_edges":2}
```

If we set the value of `vertex_must_exist` parameter to `true`, the endpoint will only insert edges whose endpoint vertices both exist. This includes the vertices being inserted in the same request. Therefore, inserting the content of `add_id6.json` to an empty graph would cause the edges to be rejected:

```
$ curl -X POST --data-binary @add_id6.json \
"http://localhost:9000/graph?vertex_must_exist=true"

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "accepted_vertices": 1,
      "accepted_edges": 0,
      "skipped_edges": 2,
      "edge_vertices_not_exist": [
        {
          "v_type": "User",
          "v_id": "id1"
        }
      ]
    }
  ],
  "code": "REST-0003"
}
```

Vertices

> To support multiple graphs within one system, the graph data REST endpoint URLs include an optional graph name.

Insert vertices

To insert vertices or edges, use the Upsert data to graph endpoint.

List vertices

`GET /graph/{graph_name}/vertices/{vertex_type}`

This endpoint returns all vertices having the type *`vertex_type`* in a graph.

Sample request:

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/User"

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "id1",
      "v_type": "User",
      "attributes": {}
    },
    {
      "v_id": "id2",
      "v_type": "User",
      "attributes": {}
    }
    // ... all vertices in graph socialNet of type User
  ]
}
```

Parameters

Name	Required	Description
`count_only`	No	Takes a boolean value. If the value is true, the `results` field will only contain the count of how many vertices were selected. Default is `false`.
		Attributes of the selected vertices to return. The

`select`	No	parameter takes a list, which is a string of comma-separated values, and will only return the attributes that are provided.
`filter`	No	Conditions used to filter the returned vertices. The parameter takes a list of conditions, which is a string of comma-separated values. If any filter conditions are provided, the endpoint will only return the vertices that satisfy the conditions. Six comparison operators are supported for this parameter: `=`, `!=`, `>`, `>=`, `<` and `<=`. If the value on the right side of an operator is a string literal, it should be enclosed in double-quotes.
`limit`	No	Integer value that specifies the total number of vertices to return
`sort`	No	Attributes to sort the results by. The parameter takes a list, which is a string of comma-separated values, and will sort the returned vertices based on the attributes provided in the list in order. Add "-" in front of the attribute to sort in descending order.
`timeout`	No	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to 0 or isn't provided, the system-wide endpoint timeout setting is applied.

Retrieve a vertex

`GET /graph/{graph_name}/vertices/{vertex_type}/{vertex_id}`

This endpoint will return a single vertice by its vertex ID.

Sample request:

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/User/id1"

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "id1",
      "v_type": "User",
      "attributes": {}
    }
  ]
}
```

Parameters:

Name	Required	Description
`select`	No	Attributes of the selected vertices to return. The parameter takes a list, which is a string of comma-separated values, and will only return the attributes that are provided.
		Integer that specifies the number of seconds after which the query will time

`timeout`	No	out. If the parameter is set to 0 or isn't provided, the system-wide endpoint timeout setting is applied.

Delete vertices

`DELETE /graph/{graph_name}/vertices/{vertex_type}`

This endpoint deletes vertices by their vertex type. The delete operation is a cascading deletion. If a vertex is deleted, then all of the edges connected to it are automatically deleted as well.

Sample request:

The response object will contain a `"deleted_vertices"` field that indicates the number of vertices that were deleted

```
curl -X DELETE "http://localhost:9000/graph/socialNet/vertices/User"

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": {
    "v_type": "person",
    "deleted_vertices": 3
  }
}
```

Parameters:

Name	Required	Description

`permanent`	No	Takes a boolean value. If the value is true, the deleted vertex IDs can never be inserted back, unless the graph is dropped or the graph store is cleared.
`filter`	No	Conditions used to filter the vertices to delete. The parameter takes a list of conditions, which is a string of comma-separated values. If any filter conditions are provided, the endpoint will only delete the vertices that satisfy the conditions. Six comparison operators are supported for this parameter: `=`, `!=`, `>`, `>=`, `<` and `<=`. If the value on the right side of an operator is a string literal, it should be enclosed in double-quotes.
`limit`	No	Integer value that specifies the total number of vertices to delete.
`sort`	No	Attributes to sort the vertices by. In delete operations, `sort` should always be used together with `limit`. The endpoint will delete the number of vertices under the limit specified in the order specified. The parameter takes a list of attributes, and the endpoint will sort all vertices based on the attributes provided in the list in order. Add `"-"` in front of the attribute to sort

		by that attribute in descending order.
`timeout`	No	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to 0 or isn't provided, the system-wide

Delete vertices by type

`DELETE /graph/{graph_name}/delete_by_type/vertices/{vertex_type}`

This endpoint deletes all vertices of the given vertex type in a graph.

Sample request:

```
curl -X DELETE "http://localhost:9000/graph/poc_graph/delete_by_type/vert
```

Parameters:

Name	Required	Description
`permanent`	No	Takes a boolean value. If the value is true, the deleted vertex IDs can never be inserted back, unless the graph is dropped or the graph store is cleared.
`ack`	No	If the parameter is set to "none", the delete operation doesn't need to get acknowledgment from any GPE. If it is set to "all" (default), the operation needs to get

		acknowledgment from all

Delete a vertex

`DELETE /graph/{graph_name}/vertices/{vertex_type}/{vertex_id}`

Sample request:

```
curl -X DELETE "http://localhost:9000/graph/socialNet/vertices/User/id1"

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": {
    "v_type": "User",
    "deleted_vertices": 1
  }
}
```

Parameters:

Name	Required	Description
`timeout`	no	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to 0 or isn't provided, the system-wide endpoint timeout setting is applied.

Edges

Insert edges

To insert vertices or edges, use the Upsert data to graph endpoint.

List edges of a vertex

`GET /graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_id}`

This endpoint returns all edges which are connected to a given vertex ID in the graph

Sample request:

```
curl -X GET "http://localhost:9000/graph/socialNet/edges/VidUser/0?limit=

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "2",
      "to_type": "Video",
      "attributes": {
        "rating": 5.2,
        "date_time": 0
      }
    },
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "0",
      "to_type": "Video",
      "attributes": {
        "rating": 6.8,
        "date_time": 0
      }
    }
  ]
}
```

Parameters:

Name	Required	Description
`count_only`	No	Takes a boolean value. If the value is true, the `results` field will only contain the count of how

		many edges were selected. Default is `false`.
`select`	No	Attributes of the selected edges to return. The parameter takes a list, which is a string of comma-separated values. If `select` is provided, the edges returned will only show the attributes provided.
`filter`	No	Conditions used to filter the edges to return. The parameter takes a list of conditions, which is a string of comma-separated values. If any filter conditions are provided, the endpoint will only return the edges that satisfy the conditions. Six comparison operators are supported for this parameter: `=`, `!=`, `>`, `>=`, `<` and `<=`. If the value on the right side of an operator is a string literal, it should be enclosed in double-quotes.
`limit`	No	Integer value that specifies the maximum limit of the total number of edges to return.
`sort`	No	Attributes to sort the results by. The parameter takes a list, which is a string of comma-separated values, and will sort all the edges based on the attributes provided in the list in order. Add `"-"` in front of the attribute to sort in descending order.

`timeout`	No	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to `0` or isn't provided, system-wide endpoint timeout setting is applied.

List edges of a vertex by edge type

`GET`
`/graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_id}/{edge_type}`

This endpoint lists all the edges of a specified type connected to a given vertex ID in the graph

Sample request:

```
curl -X GET "http://localhost:9000/graph/socialNet/edges/VidUser/0/User_V

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "2",
      "to_type": "Video",
      "attributes": {
        "rating": 5.2,
        "date_time": 0
      }
    },
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "0",
      "to_type": "Video",
      "attributes": {
        "rating": 6.8,
        "date_time": 0
      }
    }
  ]
}
```

Parameters:

Name	Required	Description
`count_only`	No	Takes a boolean value. If the value is true, the `results` field will only contain the count of how

		many edges were selected. Default is `false`.
`select`	No	Attributes of the selected edges to return. The parameter takes a list, which is a string of comma-separated values. If `select` is provided, the edges returned will only show the attributes provided.
`filter`	No	Conditions used to filter the edges to return. The parameter takes a list of conditions, which is a string of comma-separated values. If any filter conditions are provided, the endpoint will only return the edges that satisfy the conditions. Six comparison operators are supported for this parameter: `=`, `!=`, `>`, `>=`, `<` and `<=`. If the value on the right side of an operator is a string literal, it should be enclosed in double quotes.
`limit`	No	Integer value that specifies the maximum limit of the total number of edges to return.
`sort`	No	Attributes to sort the results by. The parameter takes a list, which is a string of comma-separated values, and will sort all the edges based on the attributes provided in the list in order. Add `"-"` in front of the attribute to sort in descending order.

`timeout`	No	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to `0` or isn't provided, system-wide endpoint timeout setting is applied.

List edges of a vertex by edge type and target type

```
GET /graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_id}/{ed
```

This endpoint lists edges connected to a given vertex by edge type and target vertex type

> Use `"_"` for `edge_type` in the URL to permit any edge type.

Sample request:

```
curl -X GET "http://localhost:9000/graph/socialNet/edges/VidUser/0/User_V

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "2",
      "to_type": "Video",
      "attributes": {
        "rating": 5.2,
        "date_time": 0
      }
    },
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "0",
      "to_type": "Video",
      "attributes": {
        "rating": 6.8,
        "date_time": 0
      }
    }
  ]
}
```

Parameters:

Name	Required	Description
`count_only`	No	Takes a boolean value. If the value is true, the `results` field will only contain the count of how

		many edges were selected. Default is `false`.
`not_wildcard`	No	Boolean value that indicates whether or not `"_"` supplied in the endpoint URL is a wildcard. If the parameter is true, `"_"` is interpreted literally to select only edges with edge type name equal to underscore.
`select`	No	Attributes of the selected edges to return. The parameter takes a list, which is a string of comma-separated values. If `select` is provided, the edges returned will only show the attributes provided.
`filter`	No	Conditions used to filter the edges to return. The parameter takes a list of conditions, which is a string of comma-separated values. If any filter conditions are provided, the endpoint will only return the edges that satisfy the conditions. Six comparison operators are supported for this parameter: `=`, `!=`, `>`, `>=`, `<` and `<=`. If the value on the right side of an operator is a string literal, it should be enclosed in double-quotes.
`limit`	No	Integer value that specifies the maximum limit of the total number of edges to return.
		Attributes to sort the results by. The parameter takes a

`sort`	No	list, which is a string of comma-separated values, and will sort all the edges based on the attributes provided in the list in order. Add `"-"` in front of the attribute to sort in descending order.
`timeout`	No	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to `0` or isn't provided, the system-wide endpoint timeout setting is applied.

Retrieve edge by source, target, and edge type

```
GET /graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_id}/{ed
```

This endpoint returns the edge of a specified type between a source vertex and a target vertex.

Sample request:

```
curl -X GET "http://localhost:9000/graph/socialNet/edges/VidUser/0/User_V

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "2",
      "to_type": "Video",
      "attributes": {
        "rating": 5.2,
        "date_time": 0
      }
    }
   ]
 }
```

Parameters:

Name	Required	Description
`select`	No	Attributes of the selected edges to return. The parameter takes a list, which is a string of comma-separated values. If `select` is provided, the edges returned will only show the attributes provided.
`timeout`	No	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to `0` or isn't provided, the

		system-wide endpoint timeout setting is applied

Delete an edge

```
DELETE /graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_id}/
```

Deletes an edge by its source vertex type and ID, target vertex type and ID, as well as edge type.

Sample request

```
$ curl -X DELETE "https://crunch.i.tgcloud.io:9000/graph/CrunchBasePre_20

# Response
{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "work_for_company",
      "deleted_edges": 1
    }
  ]
}
```

Parameters:

Name	Required	Description
`timeout`	no	Integer that specifies the number of seconds after which the query will time out. If the parameter is set to 0 or isn't provided, the

		system-wide endpoint timeout setting is applied.

Queries

Get query metadata

`GET /gsqlserver/gsql/queryinfo`

Returns metadata details about a query. In particular, it lists the input parameters and output `PRINT` statement syntax. **This endpoint exists on port 14240 and requests are sent to the GSQL server.** Therefore, you should provide authentication credentials in the request.

Sample request:

```
$ curl -u tigergraph:tigergraph -X GET \
"http://localhost:14240/gsqlserver/gsql/queryinfo?graph=workNet&query=to_

{
  "output": [
    {
      "v": "vertex"
    },
    {
      "@@v2": "SetAccum<vertex>"
    },
    {
      "S2": [
        {
          "v_id": "int",
          "attributes": {
            "interestList": "INT_LIST",
            "skillSet": "INT_SET",
            "skillList": "INT_LIST",
            "locationId": "STRING",
            "interestSet": "INT_SET",
            "id": "STRING"
          },
          "v_type": "person"
        },
        {
          "v_id": "int",
          "attributes": {
            "country": "STRING",
            "id": "STRING"
          },
          "v_type": "company"
        }
      ]
    },
    {
      "SDIFF.size()": "int"
    }
  ],
  "input": {
    "uid": "string",
    "uids": "set<string>",
    "vtype": "string"
  },
  "queryname": "to_vertex_setTest",
  "error": false,
  "message": "",
  "version": {
```

```
      "schema": 0,
      "edition": "DEVELOPER_EDITION",
      "api": "V2"
    }
  }
```

The JSON response object contains three fields:

- `queryname`: name of the query, same as the query input parameter.
- `input`: unordered list of the input parameter names and data types.
- `output`: JSON object that follows the same structure of the query's output. For each key-value pair, the key is the name that appears in the query output, while the values are the data types of the output.

Parameters:

Name	Required	Description
`graph`	Yes	Name of the graph
`query`	Yes	Name of the query

Run an installed query (`GET`)

`GET /query/{graph_name}/{query_name}`

Each time a new TigerGraph query is installed, a dynamic endpoint is generated. This new endpoint enables the user to run the new TigerGraph query through HTTP requests and giving the parameters in URL or in a data payload. In the case of a `GET` request, parameters should be passed in through the query string.

Parameters

Name	Required	Description
		Boolean value that indicates whether to use read-

`read_committed`	No	committed isolation level ↗ for the query. At the read committed level, it is guaranteed that any data read is committed at the moment it is read. By default, it is off.

Query parameter passing

When using a `GET` request to run an installed query, the query parameters are passed in through the query string of the URL.

Parameter type	Query string format	Example
Set or bag of primitives	Assign multiple values to the same parameter name.	A set `p1` of integers: **`p1=1&p1=5&p1=10`**
`VERTEX<type>`	Use the ID of the vertex: `parameterName=vertex_id`	A vertex with parameter name `vp` and an ID of person2: **`vp=person2`**
`VERTEX` (type not pre-specified)	Use two query string parameters: `parameterName=vertex_id` `parameterName.type=vertex_type`	A vertex with parameter name `va` , type `person` and and ID `person1` : `va=person1&va.type=person`
Set or bag of `VERTEX<type>`	Assign multiple vertex IDs to the same `SET` or `BAG` parameter name.	A set parameter named `vp` of vertices of type person: `vp=person3&vp=person4`
Set or bag of `VERTEX` (type not pre-specified)	The `SET` or `BAG` must be treated as an array. A vertex ID and type must be provided for the vertex element at each index.	A set parameter named `vp` of vertices: `vp[0]=person1&vp[0].type=person&vp[1]=11&vp[1].type=post`

Specify replica

If you have a TigerGraph HA cluster, you can specify a query to run on a particular replica with the HTTP header `GSQL-REPLICA`. The value of the header needs to be an integer within the range one to the replication factor of the cluster. If you supply a invalid value for the header, the request will return an error.

Specify thread limit

When running a query through RESTPP, you can specify a limit on the number of threads that the query is allowed to use on each node through the HTTP header `GSQL-THREAD-LIMIT`. The number of threads used by a query means the number of vCPUs used by the query. By default, a query will use all threads that are available on a machine.

For example, if you have a cluster of three nodes, each with 8 vCPUs, then a query will use all 8 threads available on a node in the cluster by default. By providing a thread limit in the request header, you can limit the query to only use a number of threads under the limit.

Sample request:

To run query `hello` on a graph named `social`, and the query parameter is of type `VERTEX<person>` whose ID is `"Tom"`

Running a query via HTTP request

```
curl -X GET "http://localhost:9000/query/social/hello?p=Tom"

# Limiting the query to use under 4 threads
curl -x GET -H "GSQL-THREAD-LIMIT: 4" "http://localhost:9000/query/social

# Specifying the query to run on the first replica
curl -X GET -H "GSQL-REPLICA: 2" "http://localhost:9000/query/social/hell

# Specifying the query to run on the primary cluster
curl -X GET -H "GSQL-REPLICA: 1" "http://localhost:9000/query/social/hell
```

Run an installed query (`POST`)

`POST /query/{graph_name}/{query_name}`

Users can also run queries through a `POST` request, which allows them to pass query parameters in JSON. This is especially helpful when the query takes complex parameters.

Parameters

Name	Required	Description
`read_committed`	No	Boolean value that indicates whether to use read-committed isolation level ↗ for the query. At the read committed level, it is guaranteed that any data read is committed at the moment it is read. By default, it is off.

Query parameter Passing

When using a `POST` request to run an installed query, the query parameters are passed in through the request body and encoded in JSON format. The formatting rules for the JSON payload is the same as using JSON to pass in parameters in the `RUN QUERY` command.

Parameter type	Syntax	Example
`DATETIME`	Use a string formatted as `"YYYY-MM-DD HH-MM-SS"`	`"2019-02-19 19:19:19"`
Set or bag of primitives	Use a JSON array containing the primitive values	`["a", "list", "of", "args"]`
`VERTEX<type>`	Use a JSON object containing a field `"id"` for the vertex ID and a field	`{"id": "person1",` `"type": "person"}`

`VERTEX` (type not specified)	`"type"` for the type of the vertex Use a JSON object containing a field `"id"` for the vertex ID	`{"id": "person1"}`
Set or bag of `VERTEX<type>`	Use a JSON array containing a list of JSON `VERTEX<type>` object	`[{"id": "person1"},` `{"id": "person2"}]`
Set or bag of vertices of unspecified types	Use a JSON array containing a list of JSON `VERTEX`	`[{"id": "person1",` `"type": "person"},{"id":` `"person2",` `"type": "person"}]`

Specify replica

If you have a TigerGraph HA cluster, you can specify a query to run on a particular replica with the HTTP header `GSQL-REPLICA`. The value of the header needs to be an integer within the range one to the replication factor of the cluster. If you supply an invalid value for the header, the request will return an error.

Specify thread limit

When running a query through RESTPP, you can specify a limit on the number of threads that the query is allowed to use on each node through the HTTP header `GSQL-THREAD-LIMIT`. The number of threads used by a query means the number of vCPUs used by the query. By default, a query will use all threads that are available on a machine.

For example, if you have a cluster of three nodes, each with 8 vCPUs, then a query will use all 8 threads available on a node in the cluster by default. By providing a thread limit in the request header, you can limit the query to only use a number of threads under the limit.

Sample request

The query in this sample request takes a parameter of type `VERTEX<person>`:

```
curl -X POST -d '{"p":{"id":"Tom","type":"person"}}' \
"http://localhost:9000/query/social/hello"

# Specify that the query run on the first replica
curl -X POST -d -H "GSQL-REPLICA: 2" '{"p":{"id":"Tom","type":"person"}}'
"http://localhost:9000/query/social/hello"

# Specify that the query run on the primary cluster
curl -X POST -d -H "GSQL-REPLICA: 1" '{"p":{"id":"Tom","type":"person"}}'
"http://localhost:9000/query/social/hello"

# Specify that the query run on
curl -X POST -d -H "GSQL-THREAD-LIMIT: 4" '{"p":{"id":"Tom","type":"perso
"http://localhost:9000/query/social/hello"
```

Installed queries can run in Detached Mode. To do this, use the `GSQL-ASYNC` header and set its value to `true`. The results and status of the queries run in Detached Mode can be retrieved with a query ID, which is returned immediately when queries are executed in Detached Mode.

Run an interpreted query

`POST /gsqlserver/interpreted_query`

This endpoint runs a GSQL query in Interpreted Mode. The query body should be supplied at the data payload, and the query's parameters are supplied as the URL's query string. **This endpoint exists on the GSQL server on port 14240.**

This request goes directly to the GSQL server (port 14240) instead of the RESTPP server (port 9000), so the username and password must be specified in the header. If you are using curl, you can use the `-u` option as shown below.

Request body:

The request body for this endpoint should be the entire `INTERPRET QUERY` statement.

Parameter passing:

When running an interpreted query through this endpoint, the query parameters should be passed in through the URL query string.

Sample request:

```
curl --fail -u <my_username>:<my_password> -X POST \
"http://localhost:14240/gsqlserver/interpreted_query?a=10" \
-d 'INTERPRET QUERY (INT a) FOR GRAPH gsql_demo {
    PRINT a;
 }'
```

List running queries

`GET /showprocesslist/{graph_name}`

This endpoint reports statistics of running queries of a graph: the query's request ID, start time, expiration time, and the REST endpoint's URL.

Sample request:

```
curl -X GET "http://localhost:9000/showprocesslist/poc_graph" | jq .

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "requestid": "65538.RESTPP_1_1.1558386411523.N",
      "startTime": "2019-05-20 14:06:51.523",
      "expirationTime": "2019-05-20 14:15:11.523",
      "url": "/sleepgpe?milliseconds=100001"
    },
    {
      "requestid": "196609.RESTPP_1_1.1558386401478.N",
      "startTime": "2019-05-20 14:06:41.478",
      "expirationTime": "2019-05-20 14:15:01.478",
      "url": "/sleepgpe?milliseconds=100000"
    }
  ],
  "code": "REST-0000"
}
```

Parameters:

No Parameters.

Abort a query

`GET /abortquery/{graph_name}`

This endpoint safely aborts a selected query by ID or all queries of an endpoint by endpoint URL of a graph.

Sample request:

```
curl -X GET "localhost:9000/abortquery/poc_graph?requestid=16842763.RESTP

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "aborted_queries": [
        {
          "requestid": "16842763.RESTPP_1_1.1561401340785.N",
          "url": "/sleepgpe?milliseconds=110000"
        },
        {
          "requestid": "16973833.RESTPP_1_1.1561401288421.N",
          "url": "/sleepgpe?milliseconds=100000"
        }
      ]
    }
  ],
  "code": "REST-0000"
}
```

Parameters:

Name	Required	Description
`requestid`	No	The ID of the query to abort. It can take a single query ID or the string `"all"`. If `requestid` is set to all. It will abort all running queries.
`url`	No	The endpoint whose running queries to abort. You must specify the base of the endpoint's URL, but then use a wildcard to allow for different parameters. For

		example, to abort all running queries for the endpoint `/sleepgpe`, use

Check query status (Detached Mode)

`GET /query_status`

This endpoint allows you to check the status of a query run in detached mode.

Sample request:

```
$ curl -s -X GET "http://localhost:9000/query_status?graph_name=poc_graph

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [{
    "requestid": "4.RESTPP_1_1.1599672031541.N",
    "startTime": "2020-09-09 10:20:31.541",
    "expirationTime": "2020-09-09 10:20:47.541",
    "url": "/query/ldbc_snb/countIndirectFriends?pid=21990232555889",
    "elapsedTime": 19,
    "status": "success"
  }]
}
```

Field	Description
`url`	URL of the given query.
`status`	The status of the given query. Possible values are `“success”`, `“timeout”`, `“aborted”`, or `“running”`.

`startTime`	The timestamp for the start time of the given query.
`requestid`	The query ID associated with the given query status JSON object.
`expirationTime`	The timestamp for when the given query times out. The default timeout limit is 16 seconds and can be set using the `GSQL-TIMEOUT` header.
`elapsedTime`	Elapsed real time of the given query measured in milliseconds. For completed queries, the value shows the total runtime

If one or more of the provided query IDs (`requestid`) are invalid, the return JSON will include an `unknown_requestid` field containing all the invalid query IDs. If a query ID is marked as unknown, it means either the query does not exist or that it was not run in Detached Mode.

Parameters:

Name	Required	Description
`graph_name`	Yes	Name of the graph the query belongs to. Required parameter.
`requestid`	No	String ID of the query. It also accepts the value `"all"`, in which case it will return a status report for every running query. When multiple `requestid` are provided, it will return the status of all corresponding queries. If no `requestid` is provided, or if the value of `requestid` is `"all"`, it will return the status of all queries that are currently running.

		The output will contain one JSON object for each

Check query results (Detached Mode)

`GET /query_result`

This endpoint allows you to check the results of queries run in Detached Mode if they have finished running. If the query is still running, the endpoint will respond with an error and a message saying `"Unable to retrieve result for query <requestid>"`. Ensure that the query is finished before checking its result.

Sample request:

```
$ curl -s -X GET curl -s -X GET "http://localhost:9000/query_result?reque

{
  "error": false,
  "message": "",
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "results": [{"vSet": [{
    "v_id": "21990232555889",
    "attributes": {"vSet.@friendCount": 13677},
    "v_type": "Person"
  }]}]
}
```

Parameters:

Name	Required	Description
`requestid`	Yes	String ID of the query.

Path-Finding Algorithms

The TigerGraph platform comes with two built-in endpoints, `/shortestpath` and `/allpaths`, which return either the shortest or all unweighted paths connecting a set of source vertices to a set of target vertices. The table below summarizes the two path-finding endpoints.

Input Parameters and Output Format for Path-Finding

Each REST endpoint reads a JSON-formatted payload that describes the input parameters. These parameters specify which vertices and edges may be on the paths, additional conditions on the attributes of the vertices and edges, and the maximum length of a path.

Source and target vertices

Each endpoint must have either a **source** or **sources** key and either a **target** or **targets** parameter. The source and target parameters describe a single vertex. The format for a vertex object is as follows: `{"type" : "<vertex_type_name>", "id" : "<vertex_id>"}.` The sources and targets parameters are JSON arrays containing a list of vertex objects.

Filters
The payload may also have an array of filter conditions, to restrict the vertices or edges in the paths. Each individual filter is a JSON object which describes a condition on one vertex type or edge type. A filter object has one or two key-value pairs: `{"type": "<vertex_or_edge_type>", "condition": "<attribute_condition>"}`

- `"type":` the vertex type or edge type to be filtered
- `"condition"` (optional): a boolean expression on one attribute of the given vertex type or edge type. "AND" and "OR" may be used to make compound expressions.

Example of a filter array:

```
[{"type": "bought", "condition": "price < \"100\" and quality == \"good\""
 {"type": "sold",   "condition": "price > \"100\"  or quality != \"good\""
```

Note that all filtering conditions in **`vertexFilters`** and **`edgeFilters`** are combined with the `"OR"` relationship, i.e., if a vertex (or edge) fulfills any one of the filter conditions, then this vertex (or edge) will be included in the resulting paths.

Output

The JSON output is a list of vertices and a list of edges. Each vertex and each edge is listed in full, with all attributes. The collections of vertices and edges are not in path order.

Find shortest path

`POST /shortestpath/{graph_name}`

This endpoint takes a source vertex or a set of source vertices, a target vertex or a set of target vertices, and returns the shortest path between the source and the target. If the source is a set of vertices, the resulting path will begin with one of the vertices in the set. If the target is a set of vertices, the resulting path will end with one of the vertices in the set.

Request body:

This endpoint expects a request body that describes the source and target vertex or vertex set. Below is a table of all the fields in the request body.

Key	Type	Description
`source`	vertex object	Each path must start from this vertex. Mutually exclusive with `sources`.

`sources`	vertex array	Each path must start from one of these vertices. Mutually exclusive with `source`.
`target`	vertex object	Each path must end at this vertex. Mutually exclusive with `targets`.
`targets`	vertex array	Each path must end at one of these vertices. Mutually exclusive with `target`.
`vertexFilters`	filter array	(OPTIONAL) Restrict the paths to those whose vertices satisfy any of the given filters.
`edgeFilters`	filter array	(OPTIONAL) Restrict the paths to those whose edges satisfy any of the given filters. See details of filters

Sample request:

```
curl -s -X POST "http://localhost:9000/shortestpath/movieNet" \
-d '{
  "sources":[{"type":"VidUser","id":"2"}],
  "targets":[{"type":"VidUser","id":"0"}, {"type":"VidUser","id":"3"}],
  "edgeFilters":[{"type":"User_Video","condition":"rating > 5 and date_ti
  "maxLength":4
}'

# Result is an array of vertex json objects and edge json objects,
# describing the subgraph for all found vertices and edges.
{
  "version": { "edition": "developer", "api": "v2", "schema": 0 },
  "error": false,
  "message": "Cannot get 'vertex_filters' filters, use empty filter.",
  "results": [
    {
      "vertices": [
        { "v_id": "3","v_type": "VidUser","attributes": { "name": "Dale"
        { "v_id": "0","v_type": "Video","attributes": { "name": "Solo", "
        { "v_id": "0","v_type": "VidUser","attributes": { "name": "Angel"
      ],
      "edges": [
        {
          "e_type": "User_Video", "from_id": "0", "from_type": "Video",
          "to_id": "0", "to_type": "VidUser", "directed": false,
          "attributes": { "rating": 6.8, "date_time": 15000 }
        },
        {
          "e_type": "User_Video", "from_id": "0", "from_type": "Video",
          "to_id": "3", "to_type": "VidUser",  "directed": false,
          "attributes": { "rating": 6.6, "date_time": 16000 }
        }
      ]
    }
  ]
}
```

Parameters:

Key	Required	Description

`maxLength`	No	Integer that specified the maximum length of a shortest path. The default value is 6.
		If **true**, the endpoint will return all shortest paths

Find all paths

`POST /allpaths/{graph_name}`

This endpoint finds all paths between a source vertex (or vertex set) and target vertex (or vertex set).

Request body:

This endpoint expects a request body that describes the source and target vertex or vertex set. Below is a table of all the fields in the request body.

Key	Type	Description
`source`	vertex object	Each path must start from this vertex. Mutually exclusive with `sources`.
`sources`	vertex array	Each path must start from one of these vertices. Mutually exclusive with `source`.
`target`	vertex object	Each path must end at this vertex. Mutually exclusive with `targets`.
`targets`	vertex array	Each path must end at one of these vertices. Mutually exclusive with `target`.
`vertexFilters`	filter array	(OPTIONAL) Restrict the paths to those whose

		vertices satisfy any of the given filters.
`edgeFilters`	filter array	(OPTIONAL) Restrict the paths to those whose edges satisfy any of the given filters. See details of filters above.

Parameters:

Name	Required	Description
`maxLength`	Yes	Maximum path length.

> ⚠ The current implementation of this endpoint will include paths with loops. Since it is possible to go around a loop an infinite number of times, it is important that you select the smallest value of maxLength which you consider appropriate. Even if there are no loops in your graph, a smaller maxLength will allow your query to run faster.

Sample request:

The example below requests all paths between the source vertex set {Video 0} and the target vertex set {AttributeTag "action"}, up to maximum length 3. The path may only contain Video vertices where `year >= 1984`. The result includes 3 paths:
AttrributeTag "action" -- Video 0
AttrributeTag "action" -- Video 3 -- VidUser 4 -- Video 0
AttrributeTag "action" -- Video 2 -- VidUser 0 -- Video 0

```
curl -s -X POST "http://localhost:9000/allpaths/movieNet" \
-d '{
  "sources":[{"type":"Video","id":"0"}],
  "targets":[{"type": "AttributeTag", "id":"action"}],
  "vertexFilters":[{"type":"Video", "condition":"year >= 1984"}],
  "maxLength": 3
}'

# Result is an array of vertex json objects and edge json objects,
# indicating the subgraph for all found vertices and edges.
{
  "version": { "edition": "developer", "api": "v2", "schema": 0 },
  "error": false,
  "message": "Cannot get 'edge_filters' filters, use empty filter.",
  "results": [
    {
      "vertices": [
        { "v_id": "action","v_type": "AttributeTag","attributes": {}},
        { "v_id": "3","v_type": "VidUser","attributes": { "name": "Dale"
        { "v_id": "0","v_type": "VidUser","attributes": { "name": "Angel"
        { "v_id": "0","v_type": "Video","attributes": { "name": "Solo", "
        { "v_id": "2","v_type": "Video","attributes": { "name": "Thor", "
        { "v_id": "4","v_type": "Video","attributes": { "name": "Ran", "y
      ],
      "edges": [
        {
          "e_type": "Video_AttributeTag", "from_id": "0", "from_type": "V
          "to_id": "action", "to_type": "AttributeTag", "directed": false
          "attributes": { "weight": 1, "date_time": 0 }
        },
        {
          "e_type": "Video_AttributeTag", "from_id": "4", "from_type": "V
          "to_id": "action", "to_type": "AttributeTag", "directed": false
          "attributes": { "weight": 1, "date_time": 11000 }
        },
        {
          "e_type": "User_Video", "from_id": "3", "from_type": "VidUser",
          "to_id": "4", "to_type": "Video", "directed": false,
          "attributes": { "rating": 8.4, "date_time": 12000 }
        },
        {
          "e_type": "User_Video", "from_id": "3", "from_type": "VidUser",
          "to_id": "0", "to_type": "Video", "directed": false,
          "attributes": { "rating": 6.6, "date_time": 16000 }
        },
        {
          "e_type": "Video_AttributeTag", "from_id": "2", "from_type": "V
          "to_id": "action", "to_type": "AttributeTag", "directed": false
```

```
            "attributes": { "weight": 1, "date_time": 0 }
          },
          {
            "e_type": "User_Video", "from_id": "2", "from_type": "VidUser",
            "to_id": "0", "to_type": "Video", "directed": false,
            "attributes": { "rating": 7.4, "date_time": 17000 }
          },
          {
            "e_type": "User_Video", "from_id": "0", "from_type": "Video",
            "to_id": "0", "to_type": "VidUser", "directed": false,
            "attributes": { "rating": 6.8, "date_time": 15000 }
          }
        ]
      }
    ]
  }
```

Other versions of pathfinding algorithms are available in the GSQL Graph Algorithm Library.

Loading...

Data Loader User Guides

Data Loaders are interfaces built in to the TigerGraph system which enable users to use the same high-level GSQL protocol for high-speed parallel data loading, whether the data reside directly on the network file system, or come from one of several other supported data sources. When the data are coming from another data source, some initial configuration is needed. Then you can use the same type of loading jobs described in the GSQL Language Reference: Part 1 - Data Definition and Loading.

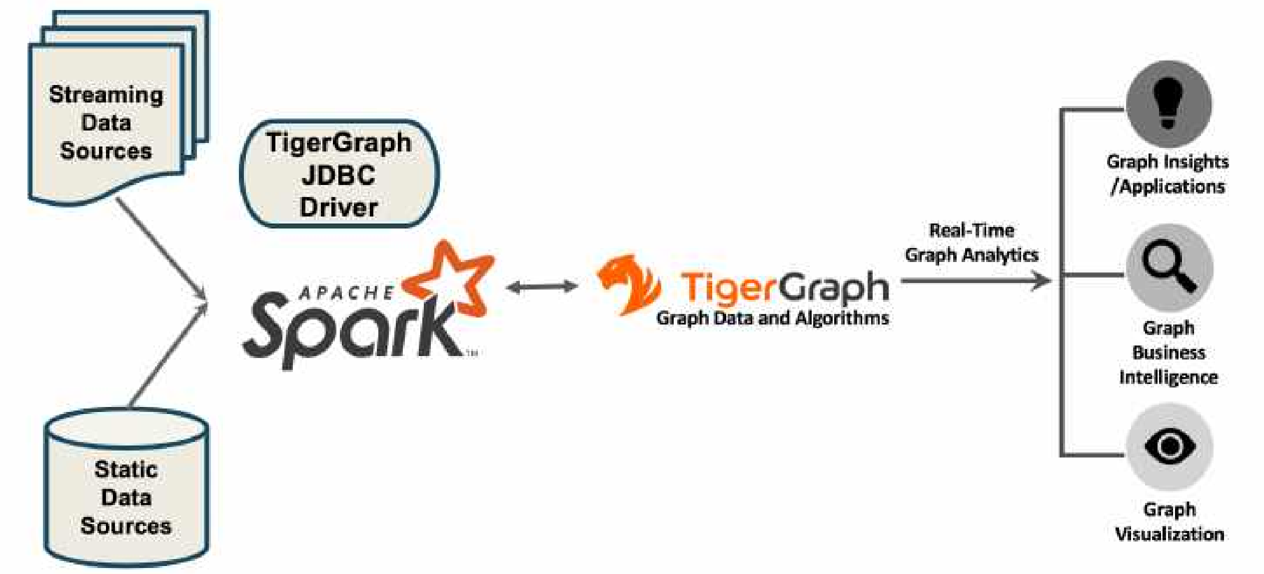
To configure a data source, see the appropriate data loader user guide:

- **AWS S3** Loader User Guide
- **Kafka** Loader User Guide
- **Spark** Connection Via JDBC Driver

Spark Connection Via JDBC Driver

Apache Spark is a popular big data distributed processing system which is frequently used in data management ETL process and Machine Learning applications.

Using the open-source type 4 JDBC Driver for TigerGraph, you can read and write data between Spark and TigerGraph. This is a two-way data connection.



The Github Link to the JDBC Driver is https://github.com/tigergraph/ecosys/tree/master/tools/etl/tg-jdbc-driver ↗
The README file there provides more details.

Note that the TigerGraph JDBC Driver has more use cases than just as a Spark Connection. You can use TigerGraph's JDBC Driver for your Java and Python applications. This is an open source project.

AWS S3 Loader User Guide

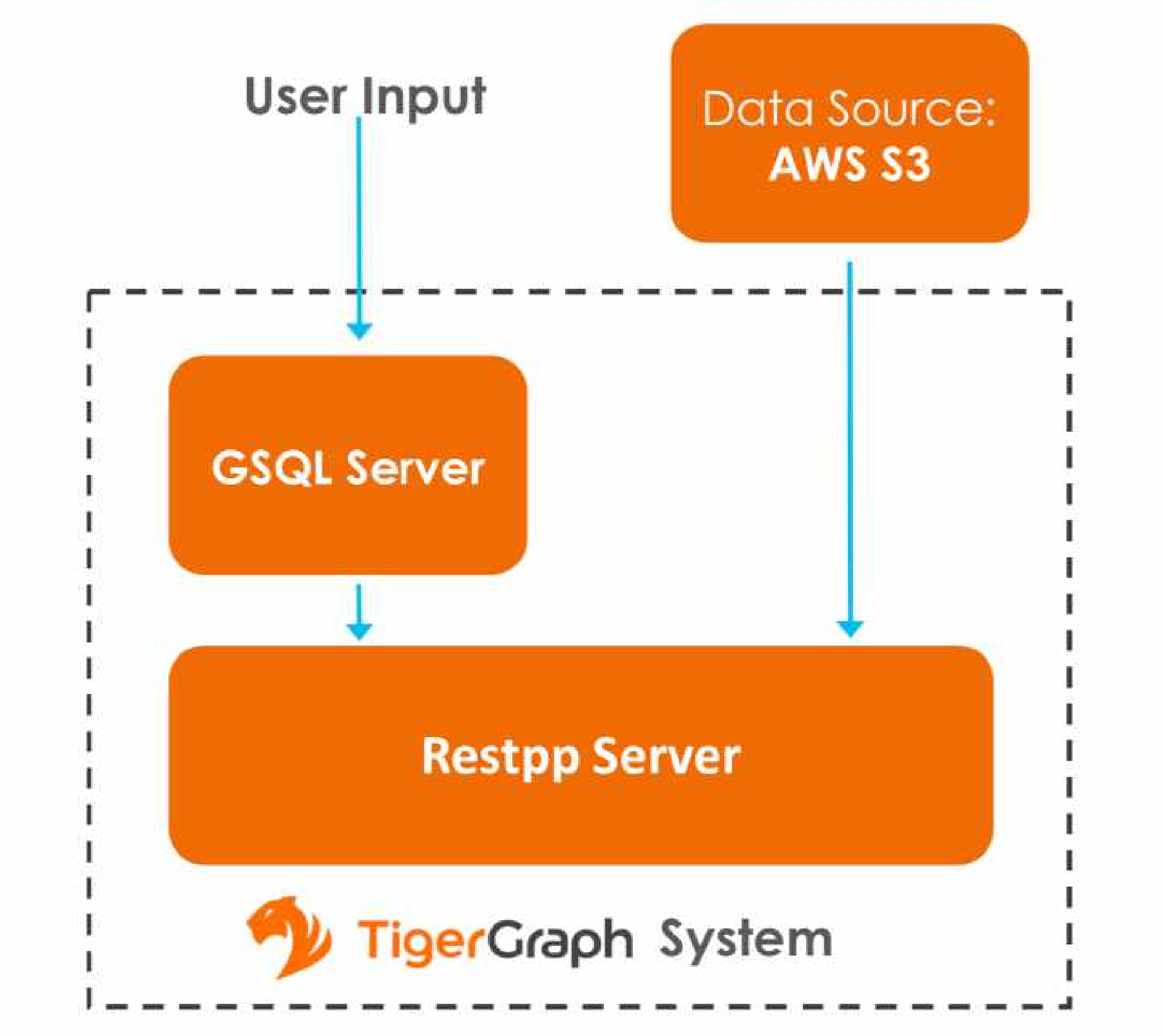
Overview

AWS Simple Storage Service (S3) is a popular destination to store data in the cloud and has become an essential component in the data pipeline of many enterprises. It is an object storage service on the AWS platform which can be accessed through a web service interface.

TigerGraph's S3 Loader makes it easier for you to integrate with an Amazon S3 service and ingest data from S3 buckets either in *realtime* or via *one-time data import* into the TigerGraph System. Your TigerGraph cluster can be deployed either on-premises or in a cloud environment.

Architecture

From a high level, a user provides instructions to the TigerGraph system through GSQL, and the external Amazon S3 data is loaded into TigerGraph's RESTPP server. The following diagram demonstrates the S3 Loader data architecture.



Prerequisites

You should have uploaded your data to Amazon S3 buckets.

Once you have the buckets set up, you need to prepare the following two configuration files and place them in your desired location in the TigerGraph system:

1. S3 data source configuration file: This file includes the credentials for accessing Amazon S3 which consists of *access key* and *secret key*. Through the configuration file, the TigerGraph system acquires the authority to access your buckets. Please see the example in Step 1. Define the Data Source.

2. S3 file configuration file: This file specifies various options for reading data from Amazon S3. Please the see example in Step 2. Create a Loading Job.

Configuring and Using the S3 Loader

There are three basic steps:

1. Define the Data Source
2. Create a Loading Job
3. Run the Loading Job

The GSQL syntax for the S3 Loader is designed to be consistent with the existing GSQL loading syntax.

1. Define the Data Source

CREATE DATA_SOURCE

Before starting a S3 data loading job, you need to define the credentials to connect to Amazons S3. The CREATE DATA_SOURCE statement defines a data_source type variable, with a sub type S3:

```
CREATE DATA_SOURCE S3 data_source_name
```

S3 Data Source Configuration File

After the data source is created, use the `SET` command to specify the path to a configuration file for that data source.

```
SET data_source_name = "/path/to/s3.config";
```

This `SET` command reads, validates, and applies the configuration file, integrating its settings into TigerGraph's dictionary. The data source configuration file's content, structured as a JSON object, describes the S3 credential settings, including the *access key* and *secret key*, as well as the AWS region. If a specific AWS region is not provided, TigerGraph will use the default region `us-west-2`. A sample configuration file is shown in the following example:

s3.config

```
{
    "file.reader.settings.fs.s3a.access.key": "AKIAJ****4YGHQ",
    "file.reader.settings.fs.s3a.secret.key": "R8bli****p+dT4",
    "aws.region":"cn-north-1"
}
```

For simplicity, you can merge the `CREATE DATA_SOURCE` and `SET` statements:

```
CREATE DATA_SOURCE S3 data_source_name = "/path/to/s3.config"
```

- If you have a TigerGraph cluster, the configuration file must be on machine m1, where the GSQL server and GSQL client both reside, and it must be in JSON format. If the configuration file uses a relative path, the path should be relative to the GSQL client working directory.
- Each time when the config file is updated, you must run "SET data_source_name" to update the data source details in the dictionary.

To further simplify, instead of specifying the S3 data source config file path, you can also directly provide the S3 data source configuration as a string argument, as shown below:

```
CREATE DATA_SOURCE S3 data_source_name = "{\"file.reader.settings.fs.s3a.
```

Tip: The above simplified statement is useful for using S3 Data Loader in TigerGraph Cloud when you are not loading S3 data through GraphStudio. In TigerGraph Cloud (tgcloud.io), you can also use GSQL web shell to define and create S3 data sources, without creating the S3 data source configuration file in the file system.

ADVANCED: MultiGraph Support

The S3 Loader supports the TigerGraph MultiGraph feature. In the MultiGraph context, a data source can be either global or local:

1. A global data source can only be created by a superuser, who can grant the global data source to any graph.
2. An admin user can only create a local data source, which cannot be accessed by other graphs.

The following are examples of permitted DATA_SOURCE operations.

1. A **superuser** may create a global level data source without assigning it to a particular graph:

```
CREATE DATA_SOURCE S3 s1 = "/path/to/config"
```

2. A **superuser** may grant/revoke a data source to/from one or more graphs:

```
GRANT DATA_SOURCE s1 TO GRAPH graph1, graph2
REVOKE DATA_SOURCE s1 FROM GRAPH graph1, graph2
```

3. An **admin** user may create a local data source for a specified graph which the admin user administers:

```
CREATE DATA_SOURCE S3 s1 = "/path/to/config" FOR GRAPH test_graph
```

In the above statement, the local data_source s1 is only accessible to graph test_graph. A superuser cannot grant it to another graph.

DROP DATA_SOURCE

A data_source variable can be dropped by a user who has the privilege. A global data source can only be dropped by a superuser. A local data_source can only be dropped by an admin for the relevant graph or by a superuser. The syntax for the DROP command is as follows:

```
DROP DATA_SOURCE <source1>[<source2>...] | * | ALL
```

Below is an example with a few legal s3 data_source create and drop commands.

```
CREATE DATA_SOURCE S3 s1 = "/home/tigergraph/s3.config"
CREATE DATA_SOURCE S3 s2 = "/home/tigergraph/s3_2.config"

DROP DATA_SOURCE s1, s2
DROP DATA_SOURCE *
DROP DATA_SOURCE ALL
```

SHOW DATA_SOURCE

The SHOW DATA_SOURCE command will display a summary of all existing data_sources for which the user has privilege:

```
$ GSQL SHOW DATA_SOURCE *

# The sample output:
Data Source:
  - S3 s1 ("file.reader.settings.fs.s3a.access.key": "AKIAJ****4YGHQ", "f
# The global data source will be shown in global scope.
# The graph scope will only show the data source it has access to.
```

2. Create a Loading Job

The S3 Loader uses the same basic CREATE LOADING JOB ↗ syntax used for standard GSQL loading jobs. A DEFINE FILENAME statement should be used to assign a loader FILENAME variable to a S3 data source name and the path to its config file.

In addition, the filename can be specified in the RUN LOADING JOB statement with the USING clause. The filename value set by a RUN statement overrides the value set in the CREATE LOADING JOB.

Below is the syntax for DEFINE FILENAME when using the S3 Loader. In the syntax, $DATA_SOURCE_NAME is the S3 data source name, and the path points to a configuration file *which provides information about how to read an Amazon S3 file*. The S3 file configuration file must be in JSON format.

```
DEFINE FILENAME filevar "=" [filepath_string | data_source_string];
data_source_string = $DATA_SOURCE_NAME":"<path_to_configfile>
```

Example: Load a S3 Data Source ***s1***, where the path to the file configuration file is "~/files.conf":

```
DEFINE FILENAME f1 = "$s1:~/files.config";
```

S3 File Configuration File

The S3 file configuration file tells the TigerGraph system exactly which Amazon S3 files to read and how to read them. Similar to the data source configuration file described above, the contents are in JSON object format. An example file is shown below.

files.config

```
{
    "file.uris": "s3://my-bucket/data.csv"
}
```

The "file.uris" key is required. It specifies one or more paths on your Amazon S3 bucket. Each path is either to an individual file or to a directory. If it is a directory, then each file directly under that directory is included. You can specify multiple paths by using a comma-separated list. An example with multiple paths is show below:

files.config

```
{
    "file.uris": "s3://my-bucket1/data1.csv,s3://my-bucket1/data2.csv,s3:
}
```

Instead of specifying the config file path, you can also directly provide the S3 file configuration as a string argument, as shown below:

```
DEFINE FILENAME f1 = "$s1:~/files.config";
DEFINE FILENAME f1 = "$s1:{\"file.uris\":\"s3://my-bucket/data.csv\"}";
```

ADVANCED: Configure How to Read S3 File

Besides the required "file.uris" key, you can further configure the S3 loader. A sample full configuration is shown below:

files.config

```
{
    "tasks.max": 1,
    "file.uris": "s3://my-bucket/data.csv",
    "file.regexp": ".*",
    "file.recursive": false,
    "file.scan.interval.ms": 60000,
    "file.reader.type": "text",
    "file.reader.batch.size": 10000,
    "file.reader.text.archive.type": "auto",
    "file.reader.text.archive.extensions.tar": "tar",
    "file.reader.text.archive.extensions.zip": "zip",
    "file.reader.text.archive.extensions.gzip": "tar.gz,tgz"
}
```

Following is a detailed explanation of each option:

- "**tasks.max**" (default is **1**): specifies the maximum number of tasks which can run in parallel. E.g. if there are 2 files and 2 tasks, each task will handle 1 file. If there are 2 files and 1 task, the single task will handle 2 files. If there is 1 file and 2 tasks, one of the tasks will handle the file.
- "**file.uris**": specifies the path(s) to the data files on Amazon S3. The path can also be dynamic by using expressions to modify the URIs at runtime. These expressions have the form `${XX}` where XX represents a pattern from `DateTimeFormatter` ↗ Java class.

> ⓘ if you want to ingest data dynamically, i.e. directories/files created every day and avoid adding new URIs every time, you can include expressions in URIs to do that. For example, for the URI `s3://my-bucket/${yyyy}`, it is converted to `s3://my-bucket/2019` when running the loader. You can use as many as you like in the URIs, for instance: `s3://my-bucket/${yyyy}/${MM}/${DD}/${HH}-${mm}`

- "**file.regexp**" (default is **.*** which matches all files): the regular expression to filter which files to read.
- "**file.recursive**" (default is **false**): whether to recursively access all files in a directory.
- "**file.scan.interval.ms**" (default is **60000**): the wait time in ms before starting another scan of the file directory after finishing the current scan. Only applicable in **stream** mode.
- "**file.reader.type**" (default is **text**): the type of file reader to use. If **text**, read the file line by line as pure text. If **parquet**, read the file as parquet format.
- "**file.reader.batch.size**" (default is **1000**): maximum number of lines to include in a single batch.
- "**file.reader.text.archive.type**" (default is **auto**): the archive type of the file to be read. If **auto**, determine the archive type automatically. If **tar**, read the file with tar format. if **zip**, read the file with zip format. If **gzip**, read the file with gzip format. If **none**, read the file normally.
- "**file.reader.text.archive.extensions.tar**" (default is **tar**): the list of file extensions to be read with tar format.
- "**file.reader.text.archive.extensions.zip**" (default is **zip**): the list of file extensions to be read with zip format.
- "**file.reader.text.archive.extensions.gzip**" (default is **gzip**): the list of file extensions to be read with gzip format.

> The archive type is applied to all files in "file.uris" when loading. If you have different archive type files to be read at the same time, set **auto** for "file.reader.text.archive.type" and configure how to detect each archive extensions by providing the extensions list. Currently we support **tar**, **zip** and **gzip** archive types.

3. Run the Loading Job

The S3 Loader uses the same RUN LOADING JOB statement that is used for GSQL loading from files. Each filename variable can be assigned a string "DATA_SOURCE Var:file configure", which will override the value defined in the loading job. In the example below, the config files for f2 and f3 are being set by the RUN command, whereas f1 is using the config which was specified in the CREATE LOADING JOB statement.

```
RUN LOADING JOB job1 USING f1, f2="$s1:~/files1.config", f3="$s2:~/files2
```

A RUN LOADING JOB instance may only use one type of data source. E.g., you may not mix both S3 data sources and regular file data sources in one loading job.

All filename variables in one loading job statement must refer to the same DATA_SOURCE variable.

There are two modes for the S3 Loader: **streaming** mode and **EOF** mode. The default mode is **streaming** mode. In **streaming** mode, loading will never stop until the job is aborted. In **EOF** mode, loading will stop after consuming the provided Amazon S3 file objects.

To set **EOF** mode, an optional parameter is added to the RUN LOADING JOB syntax:

```
RUN LOADING JOB [-noprint] [-dryrun] [-n [i],j] jobname
   [ USING filevar [="filepath_string"][, filevar [="filepath_string"]]*
   [, CONCURRENCY="cnum"][,BATCH_SIZE="bnum"]][, EOF="true"]
```

Manage Loading Jobs

S3 Loader loading jobs are managed the same way as native loader jobs. The three key commands are

- SHOW LOADING STATUS
- ABORT LOADING JOB
- RESUME LOADING JOB

For example, the syntax for the SHOW LOADING STATUS command is as follows:

```
SHOW LOADING STATUS job_id|ALL
```

To refer to a specific job instance, use the job_id which is provided when RUN LOADING JOB is executed. For each loading job, the above command reports the following information :

1. current loaded lines
2. average loading speed
3. loaded size
4. duration

See Inspecting and Managing Loading Jobs ↗ for more details.

S3 Loader Example

Here is an example code for loading data through the S3 Loader:

```
USE GRAPH test_graph
DROP JOB load_person
DROP DATA_SOURCE s1

# Create data_source s3 s1 = "s3_config.json" for graph test_graph.
CREATE DATA_SOURCE S3 s1 FOR GRAPH test_graph
SET s1 = "s3_config.json"

# Define the loading jobs.
CREATE LOADING JOB load_person FOR GRAPH test_graph {
  DEFINE FILENAME f1 = "$s1:s3_file_config.json";
  LOAD f1
      TO VERTEX Person VALUES ($2, $0, $1),
      TO EDGE Person2Comp VALUES ($0, $1, $2)
      USING SEPARATOR=",";
}

# load the data
RUN LOADING JOB load_person
```

Loading...

GSQL Cheat Sheets

GSQL Language quick reference manuals.

This page provides two cheat sheets for GSQL:

1. GSQL Cheat Sheet for Data Definition and Loading
2. GSQL Cheat Sheet for Querying

The cheat sheets are updated for TigerGraph v3.1.

63KB
DDL Language Cheatsheet.pdf
pdf

110KB
Query Language Cheatsheet.pdf
pdf

Using a Remote GSQL Client

Installation

When the TigerGraph platform is installed, the GSQL client and server are on the same machine. The client is packaged as a Java jar file, `gsql_client.jar` located in the folder `<TigerGraph_root_dir>/app/<VERSION_NUM>/dev/gdk/gsql/lib/` . Installation consists of copying the file `gsql_client.jar` to the client machine and storing it anywhere the user finds appropriate. The client machine needs to have Java 7.0 or higher.

Running the Client

To run the client, execute the jar file each time that you would run gsql if you were local to the GSQL server. That is, the command

```
java -jar <path>gsql_client.jar
```

takes the place of gsql. For example, the commands

```
gsql DROP ALL
gsql create_my_schema.gsql
gsql LS
```

would become

```
java -jar gsql_client.jar DROP ALL
java -jar gsql_client.jar create_my_schema.gsql
java -jar gsql_client.jar LS
```

Therefore, it may be useful to define a Unix alias:

```
alias gsql="java -jar <path>/gsql_client.jar"
```

The java operation alone is not sufficient, however, because it does not tell the client where to find the GSQL server. In addition, the client needs to satisfy three conditions:

1. It must know the IP address of the GSQL server.
2. It must have the authorization to access the server in general and to execute the requested GSQL commands in particular.
3. If a SSL/TLS encrypted connection (e.g., HTTPS) is used, then it must provide the certificate chain.

Specifying Server IP Address

There are two ways to provide the IP address of the GSQL Server.

- Method 1: Store the address in a file. Create a one-line file called **gsql_server_ip_config** containing the ip address of the GSQL server. **This file needs to be in the same directory where you run GSQL.**
- Method 2 : Every time you run the client jar, provide the ip address on the command line, e.g., " *gsql -ip 192.168.55.46* "

Client Authorization and Authentication

The GSQL server applies the same user authentication procedures to remote GSQL users that it applies to local GSQL users. That is, if user authentication has been enabled, then each gsql command line must include valid user credentials.

The client addresses the server at server port 14240. You need to make sure your security policy allows the access to this port. If you want to use another port, use the `-ip` option and specify the port at the end of the IP address.

```
$ java -jar gsql.jar -ip ip_address:alternative_port
```

⚠ It is strongly recommended that you enable user authentication. See the document ***Managing User Privileges and Authentication v2.1#GSQLAuthentication*** for more details.

HTTPS Connection

If SSL/TLS is enabled for TigerGraph, to connect a GSQL remote client to the GSQL server, each GSQL command line should provide the certificate chain file via the -cacert option. This certificate file should be exactly the same as the file of entry Nginx.SSL.Cert setting SSL for Nginx. See Encrypting Connections: Step 2. Configure SSL with Gadmin. For example:

```
gsql -cacert /path/to/certificate -ip hostname:port
```

File Path Semantics

Data loading jobs always specify an input file location; logically the data should be on the server side, not on the client side. Because the command request comes from one machine and the target data file is on another machine, it no longer makes sense to use a relative path.

Rule: If a remote GSQL client invokes an instruction containing a relative path, the GSQL server considers the starting point of the path to be `<tigergraph_rootdir>/app/<VERSION_NUM>/dev/gdk/gsql` on the GSQL server.

It is strongly recommended that remotely-run GSQL commands use absolute paths only.

For example, if the data file cf_data.csv is in the folder /home/tigergraph/example/cf/, then the command to run the loading job might look like this:

```
java -jar gsql_client.jar 'RUN JOB load_cf USING FILENAME="/home/tigergra
```

Example: Modifying a Bash Script for a Remote GSQL Client

The GSQL Tutorials employ both GSQL and bash scripts to run the examples. Typically, each example case contains 3 GSQL command files (for schema creation, data loading, and querying) and one bash script to run all the parts together and to display status information. Below is a simplified version of the Collaborative Filtering (cf) bash script:

RUN_cf.sh: Bash script for Collaborative Filtering (cf) example

```
#!/bin/bash
test='cf'
####
gsql 'DROP ALL'
gsql ../${test}/${test}_model.gsql
gsql 'CREATE GRAPH gsql_demo(*)'

# Loading
gsql -g gsql_demo ../${test}/${test}_load.gsql
## loading script contains this line:
## RUN JOB load_cf USING FILENAME="../cf/data/cf_data.csv", SEPARATOR=","

# Querying
gsql -g gsql_demo ../${test}/${test}_query.gsql
gsql -g gsql_demo INSTALL QUERY ALL
gsql -g gsql_demo 'RUN QUERY topCoLiked("id1", 10)'
```

The bash script will not run from a remote GSQL client unless a few changes are made: We need to invoke "java -jar gsql_client.jar" instead of "gsql", and need to specify the server ip address. If we use the gsql_server_ip_config file, this file must be in the same folder as the command file. The GSQL Tutorial has several different folders, one for each example, so that suggests making several config files. Below is an approach that minimizes the changes required and maximizes standardization.

A. Do initial client setup. This is done only once.

1. Store gsql_client.jar in a standard location, say *~/gsql_client/gsql_client.jar* (e.g,, */home/tigergraph/gsql_client/ gsql_client.jar*)

2. Create a file called gsql_server_ip_config containing the GSQL server's IP address, and store it a standard location, say *~/gsql_client/gsql_server_ip_config* .

Sample config file: /home/tigergraph/gsql_client/gsql_server_ip_config

```
123.45.67.255
```

3. In the .bashrc file in your home directory, add an alias for gsql which points to the standard location:

```
alias gsql='java -jar ~/gsql_client/gsql_client.jar'
```

B. Add a standard header to each bash script.

standard which makes 'gsql' work on remote clients

```
alias gsql='java -jar gsql_client.jar'
shopt -s expand_aliases
ln -s ~/gsql_client/gsql_client.jar gsql_client.jar
ln -s ~/gsql_client/gsql_server_ip_config gsql_server_ip_config
```

This header does the following:

1. Repeat the alias definition for the gsql command. The definition in .bashrc may not be visible here.
2. By default, bash scripts ignore aliases. Instruct the script to use aliases.
3. Define softlinks from the current folder to the locations of the client jar and config file.

C. Change any relative paths to absolute paths. This is the only step that must be customized for each script.

Here is the resulting script. Four standard lines were added to the beginning, and one line was edited in the cf_load.gsql file.

RUN_cf_remote.sh: Modified bash script for Collaborative Filtering (cf) example

```
#!/bin/bash
alias gsql='java -jar gsql_client.jar'
shopt -s expand_aliases
ln -s ~/gsql_client/gsql_client.jar gsql_client.jar
ln -s ~/gsql_client/gsql_server_ip_config gsql_server_ip_config
test='cf'
####
gsql 'DROP ALL'
gsql ../${test}/${test}_model.gsql
gsql 'CREATE GRAPH gsql_demo(*)'

# Loading
gsql -g gsql_demo ../${test}/${test}_load.gsql
## loading script contains this line:
## RUN JOB load_cf USING FILENAME="/home/tigergraph/tigergraph/document/e

# Querying
gsql -g gsql_demo ../${test}/${test}_query.gsql
gsql -g gsql_demo INSTALL QUERY ALL
gsql -g gsql_demo 'RUN QUERY topCoLiked("id1", 10)'
```

Legal

Patents and Third Party Software

Patent and Third Party Notice for TigerGraph Platform v 3.2 Sep 2021

U.S. Pat. No. 9953106, 9977837, 10120956.
Additional Patents pending.

The TigerGraph software program uses some third-party software components that are licensed under their own terms. The attached notices are provided for information only.

- Section 1. List of third-party software in the TigerGraph platform.
- Section 2. List of additional third-party software in the TigerGraph Cloud Service interface.
- Section 3. Table of license types.

A separate third-party disclosure is available for the GraphStudio Visual SDK. See https://docs.tigergraph.com/ui/graphstudio/patent-and-third-party-notice

==

Section 1. List of third-party software in the TigerGraph platform

==

antlr Copyright (c) 2014 antlr. https://www.antlr.org/ Licensed under BSD3

commons-cli Copyright (c) 2002-2017 The Apache Software Foundation. https://commons.apache.org/proper/commons-cli/ Licensed under Apache2

commons-codec Copyright (c) 2002-2017 The Apache Software Foundation. https://commons.apache.org/proper/commons-codec/ Licensed under Apache2 commons-lang3 Copyright (c) 2001-2018 The Apache Software Foundation. https://commons.apache.org/proper/commons-lang/ ↗ Licensed under Apache2

diffutils Copyright (c) 2010 Free Software Foundation. https://github.com/java-diff-utils/java-diff-utils ↗ Licensed under Apache2

grpc Copyright (c) 2018 The gRPC authors. https://grpc.io/ ↗ Licensed under Apache2

guava Copyright (c) 2010-2016 OneLogin, Inc. https://github.com/google/guava ↗ Licensed under Apache2

java-saml Copyright (c) 2010-2016 OneLogin, Inc. https://github.com/onelogin/java-saml ↗ Licensed under MIT

jline Copyright (c) 2002-2018 Guillaume Nodet. https://github.com/jline/jline3 ↗ Licensed under BSD3

joda-time Copyright (c) 2002-2018 Joda. https://www.joda.org/joda-time/ ↗ Licensed under Apache2

json Copyright (c) 2002 JSON. https://github.com/stleary/JSON-java ↗ Licensed under JSON

LDAP SDK Copyright (c) 2009 Ping Identity Corporation. https://github.com/pingidentity/ldapsdk/blob/master/README.md ↗ Licensed under LDAPSDK

log4j Copyright (c) 1999-2018 The Apache Software Foundation. https://logging.apache.org/log4j/2.x/ ↗ Licensed under Apache2

netty Copyright (c) 2018 The Netty Project. https://netty.io/ ↗ Licensed under Apache2

protobuf-java Copyright (c) 2008 Google. https://developers.google.com/protocol-buffers/ ↗ Licensed under BSD3 slf4j Copyright (c) 2004-2017 QOS.ch. https://www.slf4j.org/ ↗ Licensed under MIT

stax2-api Copyright (c) 2013-2019 Tatu Saloranta. https://github.com/FasterXML/stax2-api ↗ Licensed under BSD

woodstox-core-asl Copyright (c) 2014-2019 Tatu Saloranta. https://github.com/FasterXML/woodstox ↗ Licensed under Apache2

xmlsec Copyright (c) 2002-2016 Aleksey Sanin. https://www.aleksey.com/xmlsec/ ↗ Licensed under MIT

yamlbeans Copyright (c) 2008 Nathan Sweet. https://github.com/EsotericSoftware/yamlbeans ↗ Licensed under MIT

zookeeper Copyright (c) 2010-2018 The Apache Software Foundation. https://zookeeper.apache.org/ ↗ Licensed under Apache2

gson Copyright (c) 2008-2019 Google. https://github.com/google/gson ↗ Licensed under Apache2

javacc Copyright (c) 2006 Sun Microsystems. https://javacc.org/ ↗ Licensed under BSD3

javacpp Copyright (c) 1989 Ty Coon. https://github.com/bytedeco/javacpp ↗ Licensed under Apache

murmurhash Copyright (c) 2010-2019 aapleby. https://github.com/aappleby/smhasher ↗ Licensed under MIT

kafka Copyright (c) 2017 The Apache Software Foundation. https://kafka.apache.org/ ↗ Licensed under Apache2

librdkafka Copyright (c) 2012-2018 Magnus Edenhill. https://github.com/edenhill/librdkafka ↗ Licensed under BSD2

libfcgi Copyright (c) 2012-2019 toshic. https://github.com/toshic/libfcgi ↗ Licensed under FCGI zlib Copyright (c) 1995-2017 Jean-loup Gailly. https://zlib.net/ Licensed under ZLIB

libcurl Copyright (c) 1996-2019 Daniel Stenberg. https://curl.haxx.se/libcurl/ Licensed under CURL

zeromq Copyright (c) 2007-2016 various copyright holders. http://zeromq.org/ Licensed under LGPL3

libhttp_parser Copyright (c) 2009-2019 Joyent. https://github.com/nodejs/http-parser Licensed under MIT

yaml-cpp Copyright (c) 2008-2015 Jesse Beder. https://github.com/jbeder/yaml-cpp Licensed under MIT

cryptopp Copyright (c) 1995-2016 Wei Dai. https://www.cryptopp.com/ Licensed under BOOST

boost Copyright (c) 1998-2005 Beman Dawes, David Abrahams. https://www.boost.org/ Licensed under BOOST

snappy Copyright (c) 2011 Google, Inc. https://github.com/google/snappy Licensed under BSD3

jsoncpp Copyright (c) 2007-2010 Baptiste Lepilleur. https://github.com/open-source-parsers/jsoncpp Licensed under MIT

cereal Copyright (c) 2014 Randolph Voorhies, Shane Grant. https://github.com/USCiLab/cereal Licensed under BSD3

nginx Copyright (c) 2011-2018 nginx. http://nginx.org/ Licensed under BSD2

jemalloc Copyright (c) 2002-2018 Jason Evans, (c) 2007-2012 Mozilla Foundation, (c) 2009-2018 Facebook, Inc. http://jemalloc.net/ Licensed under BSD2

sparsehash Copyright (c) 2005 Google. https://github.com/sparsehash/sparsehash Licensed under BSD3 rapidjson Copyright (c) 2006-2013 Alexander Chemeris. https://github.com/Tencent/rapidjson Licensed under MIT

openjdk Copyright (c) 2019 Oracle. https://openjdk.java.net/ Licensed under GPL2+CE GPL2 with Classpath Exception: Clarification that the using the licensed code with other code does not require that other code to be GPL2 compatible.

gcc Copyright (c) 1992-2017 Free Software Foundation. https://gcc.gnu.org/ (gcc) Licensed under GPL3, with Runtime Library Exception The GCC license provides an exception to GPLv3 to allow compilation of non- GPL (including proprietary) programs to use, in this way, the header files and runtime libraries covered by this Exception. The complete terms are available at https://www.gnu.org/licenses/gcc-exception.html

tsar Copyright (c) 2004 Apache. https://github.com/alibaba/tsar Licensed under Apache2

gperftools Copyright (c) 2005 Google. https://github.com/gperftools/gperftools (we use its tcmalloc) Licensed under BSD3

python 2.7 Copyright (c) 2001-2019 Python. https://www.python.org/download/releases/2.7/license/ Licensed under Python2

Go github.com/GeertJohan/go.rice/embedded Licensed under BSD 2-clause "Simplified" License

Gabs github.com/Jeffail/gabs Licensed under MIT License

Sarama github.com/Shopify/sarama Licensed under MIT License

Go Meta Linter github.com/alecthomas/gometalinter Licensed under MIT License

Units github.com/alecthomas/units Licensed under MIT License

Quantile github.com/beorn7/perks/quantile Licensed under MIT License

Readline github.com/chzyer/readline Licensed under MIT License

Misspell github.com/client9/misspell/cmd/misspell ↗ Licensed under MIT License

Client V3 github.com/coreos/etcd/clientv3 ↗ Licensed under Apache License 2.0

Journal github.com/coreos/go-systemd/journal ↗ Licensed under Apache License 2.0

Capnslog github.com/coreos/pkg/capnslog ↗ Licensed under Apache License 2.0

Zip github.com/daaku/go.zipexe ↗ Licensed under MIT License

Spew github.com/davecgh/go-spew/spew ↗ Licensed under ISC License

JWT github.com/dgrijalva/jwt-go ↗ Licensed under MIT License

Breaker github.com/eapache/go-resiliency/breaker ↗ Licensed under MIT License

Go xerial snappy github.com/eapache/go-xerial-snappy ↗ Licensed under MIT License

Queue github.com/eapache/queue ↗ Licensed under MIT License

Color github.com/fatih/color ↗ Licensed under MIT License

Gorp github.com/go-gorp/gorp ↗ Licensed under MIT License

Oleutil github.com/go-ole/go-ole/oleutil ↗ Licensed under MIT License

Protobuf github.com/gogo/protobuf ↗ Licensed under BSD 3-clause "New" or "Revised" License

Go Lint github.com/golang/lint/golint ↗ Licensed under BSD 3-clause "New" or "Revised" License

Snappy github.com/golang/snappy ↗ Licensed under BSD 3-clause "New" or "Revised" License

Renameio github.com/google/renameio ↗ Licensed under Apache License 2.0

RocksDB https://github.com/facebook/rocksdb/ ↗ Licensed under Apache License 2.0

Shlex github.com/google/shlex ↗ Licensed under Apache License 2.0

UUID github.com/google/uuid ↗ Licensed under BSD 3-clause "New" or "Revised" License

Ineffassign github.com/gordonklaus/ineffassign ↗ Licensed under MIT License

Backoffutils github.com/grpc-ecosystem/go-grpc-middleware/util/backoffutils ↗ Licensed under Apache License 2.0

Prometheus github.com/grpc-ecosystem/go-grpc-prometheus ↗ Licensed under Apache License 2.0

GRPC-gateway github.com/grpc-ecosystem/grpc-gateway ↗ Licensed under BSD 3-clause "New" or "Revised" License

Go-uuid github.com/hashicorp/go-uuid ↗ Licensed under Mozilla Public License 2.0

Go-fork github.com/jcmturner/gofork ↗ Licensed under BSD 3-clause "New" or "Revised" License

Ansiterm github.com/juju/ansiterm ↗ Licensed under GNU Lesser General Public License v3.0

Tabwriter github.com/juju/ansiterm/tabwriter ↗ Licensed under BSD 3-clause "New" or "Revised" License

Compress github.com/klauspost/compress ↗ Licensed under BSD 3-clause "New" or "Revised" License

Xxhash github.com/klauspost/compress/zstd/internal/xxhash ↗ Licensed under MIT License

fs github.com/kr/fs ↗ Licensed under BSD 3-clause "New" or "Revised" License

Pretty github.com/kr/pretty ↗ Licensed under MIT License

Text github.com/kr/text ↗ Licensed under MIT License

Vtclean github.com/lunixbochs/vtclean ↗ Licensed under MIT License

Go-colorable github.com/mattn/go-colorable ↗ Licensed under MIT License

Go-isatty github.com/mattn/go-isatty ↗ Licensed under MIT License

Go-sqlite3 github.com/mattn/go-sqlite3 ↗ Licensed under MIT License

Pbutil github.com/matttproud/golang_protobuf_extensions/pbutil ↗ Licensed under Apache License 2.0

i18n github.com/nicksnyder/go-i18n/i18n ↗ Licensed under MIT License

Go-toml github.com/pelletier/go-toml ↗ Licensed under MIT License

xxh32 github.com/pierrec/lz4/internal/xxh32 ↗ Licensed under BSD 3-clause "New" or "Revised" License

errors github.com/pkg/errors ↗ Licensed under BSD 2-clause "Simplified" License

sftp github.com/pkg/sftp ↗ Licensed under BSD 2-clause "Simplified" License

difflib github.com/pmezard/go-difflib/difflib ↗ Licensed under BSD 3-clause "New" or "Revised" License

Go github.com/prometheus/client_model/go ↗ Licensed under Apache License 2.0

Common github.com/prometheus/common ↗ Licensed under Apache License 2.0

Procfs github.com/prometheus/procfs ↗ Licensed under Apache License 2.0

Go-metrics github.com/rcrowley/go-metrics ↗ Licensed under BSD 2-clause "Simplified" License

Cron github.com/robfig/cron ↗ Licensed under MIT License

Gopsutil github.com/shirou/gopsutil ↗ Licensed under BSD 3-clause "New" or "Revised" License

W32 github.com/shirou/w32 ↗ Licensed under BSD 3-clause "New" or "Revised" License

Cobra github.com/spf13/cobra ↗ Licensed under Apache License 2.0

Pflag github.com/spf13/pflag ↗ Licensed under BSD 3-clause "New" or "Revised" License

Testify github.com/stretchr/testify ↗ Licensed under MIT License

Promptui github.com/tigergraph/promptui ↗ Licensed under BSD 3-clause "New" or "Revised" License

Deadcode github.com/tsenart/deadcode ↗ Licensed under BSD 3-clause "New" or "Revised" License

Etcd go.etcd.io/etcd ↗ Licensed under Apache License 2.0

Atomic go.uber.org/atomic ↗ Licensed under MIT License

Multierr go.uber.org/multierr ↗ Licensed under MIT License

Zap go.uber.org/zap ↗ Licensed under MIT License

Crypto https://pkg.go.dev/golang.org/x/crypto ↗ Licensed under BSD 3-clause "New" or "Revised" License

Golint https://pkg.go.dev/golang.org/x/lint/golint ↗ Licensed under BSD 3-clause "New" or "Revised" License

Net https://pkg.go.dev/golang.org/x/text ↗ Licensed under BSD 3-clause "New" or "Revised" License

Errgroup https://pkg.go.dev/golang.org/x/sync/errgroup ↗ Licensed under BSD 3-clause "New" or "Revised" License

Sys https://pkg.go.dev/golang.org/x/sys ↗ Licensed under BSD 3-clause "New" or "Revised" License

Text https://pkg.go.dev/golang.org/x/text ↗ Licensed under BSD 3-clause "New" or "Revised" License

Genproto https://pkg.go.dev/google.golang.org/genproto ↗ Licensed under Apache License 2.0

Grpc https://pkg.go.dev/google.golang.org/grpc ↗ Licensed under Apache License 2.0

Check.v1 gopkg.in/check.v1 ↗ Licensed under BSD 2-clause "Simplified" License

Aescts.v1 gopkg.in/jcmturner/aescts.v1 ↗ Licensed under Apache License 2.0

dnsutils.v1 gopkg.in/jcmturner/dnsutils.v1 ↗ Licensed under Apache License 2.0

gokrb5.v7 gopkg.in/jcmturner/gokrb5.v7 ↗ Licensed under Apache License 2.0

rpc.v1 gopkg.in/jcmturner/rpc.v1 ↗ Licensed under Apache License 2.0

jack.v2 gopkg.in/natefinch/lumberjack.v2 ↗ Licensed under MIT License

yaml.v2 gopkg.in/yaml.v2 ↗ Licensed under Apache License 2.0

Go Programming Language cloud.google.com/go ↗ Licensed under Apache2

go-semver github.com/coreos/go-semver ↗ Licensed under Apache2

Resty github.com/go-resty/resty ↗ Licensed under MIT license

flock github.com/gofrs/flock ↗ Licensed under BSD 3-clause "New" or "Revised" License protobuf github.com/golang/protobuf ↗ Licensed under BSD 3-clause "New" or "Revised" License

client_golang github.com/prometheus/client_golang ↗ Licensed under Apache2

net golang.org/x/net ↗ Licensed under BSD 3-clause "New" or "Revised" License

Section 2. List of additional third-party software in the TigerGraph Cloud Service interface

The TigerGraph Cloud Service permits users to use the TigerGraph graph database and analytics platform via a web interface. **TigerGraph does not copy or distribute the TigerGraph Cloud software to the end user. **

In additional to having the third-party components of the TigerGraph Platform, the TigerGraph Cloud Service interface includes the following additional third-party software.

Netdata Copyright (c) 2016-2018, Costa Tsaousis. Copyright (c) 2018, Netdata Inc. https://github.com/netdata/netdata ↗ Licensed under GPL v3 or later ↗.

Section 3. Table of license types.

The following table explains the license abbreviations used in the list of TigerGraph Third Party Software. A link is provided to an official source for each license.

Abbreviation	License Name and Source
Apache2	Apache License version 2.0 https://www.apache.org/licenses/LICENSE-2.0 ↗
	Boost Software License

BOOST	http://www.boost.org/LICENSE_1_0.txt ↗
BSD2	2-Clause BSD (Berkeley Standard Distribution) License https://opensource.org/licenses/BSD-2-Clause ↗
BSD3	3-Clause BSD (Berkeley Standard Distribution) License https://opensource.org/licenses/BSD-3-Clause ↗
CURL	Curl License https://curl.haxx.se/docs/copyright.html ↗
FCGI	FastCGI2 License https://github.com/FastCGI-Archives/fcgi2/blob/master/LICENSE.TERMS ↗
GPL2	GNU General Public License version 2.0 https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html ↗
GPL2+CE	GNU General Public License, version 2, with the Classpath Exception https://openjdk.java.net/legal/gplv2+ce.html ↗
GNU	General Public License version 3.0 https://www.gnu.org/licenses/gpl-3.0.en.html ↗
ISC	ISC License https://opensource.org/licenses/ISC ↗
JSON	JSON License http://www.json.org/license.html ↗
LDAPSDK	UnboundID LDAP SDK Free Use License https://docs.ldap.com/ldap-sdk/docs/LICENSE-UnboundID-LDAPSDK.txt ↗

LGPL3	GNU Lesser General Public License version 3.0 https://www.gnu.org/licenses/lgpl-3.0.en.html
MIT	MIT (Massachusetts Institute of Technology) License https://opensource.org/licenses/MIT
Mozilla	Mozilla Public License 2.0 https://www.mozilla.org/en-US/MPL/2.0/
MPICH	MPICH License http://git.mpich.org/mpich.git/blob/HEAD:/COPYRIGHT
OPENSSL	OpenSSL License https://www.openssl.org/source/license.html
Python2	Python 2.7 License https://www.python.org/download/releases/2.7/license/
SLI_OFL1.1	SIL Open Font License version 1.1 http://scripts.sil.org/cms/scripts/page.php?item_id=OFL_web
ZLIB	zlib License

Workshop

Connected Data London 2019

Graphs, Machine Learning, and Explainable AI

CDL19 - Graphs, ML, and Explainable AI.pdf
pdf
2MB

CDL'19 - Graphs, ML, and Exp

- Computer Setup
  - You will be using a Linux virtual server on the cloud.
  - You will need a browser, other than Microsoft IE or Edge. Google Chrome has been tested the most thoroughly.
  - You need to remote login to the Linux server and use a shell console. If you have a Linux or Mac, you can use your native shells and ssh. If you have a Windows machine, you need to install a terminal emulator like PuTTY: https://linuxacademy.com/guide/17385-use-putty-to-access-ec2-linux-instances-via-ssh-from-windows/
  - The instructor will tell you how to log into your particular machine.
- Exercise 1: Unsupervised Learning - Community Detection Graph Algorithm
- Exercise 2: Graph Database as a Neural Network
- Exercise 3: Extracting Graph Features to Train an Explainable AI Model