2.5

TigerGraph Documentation

Deprecated Features scheduled to be Dropped in next Major Release:

Some features which are currently in the Deprecated State are scheduled to be removed from the product in the next Major Release (e.g., 3.0). Please see the detailed list on the following page: v3.0 Removal of Previously Deprecated Features

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 Server 2.5

Major Sections	Quick Links	
Portal	Developer Portal ↗	
Algorithm Library	GSQL Graph Algorithm Library	
Release Notes	Release Notes-TigerGraph 2.5	
Get Started	HW & SW Requirements Platform Installation GSQL 101 , GSQL 102 , Accumulators	
	Knowledge Base and FAQs	

FAQs & Cheatsheets	GSQL Cheatsheets Comparing TigerGraph Editions	
GSQL	GSQL 101 GSQL 102 Pattern Matching Accumulators Tutorial GSQL Demo Examples GSQL Spec - Data Definition & Loading GSQL Spec - Querying Interpreted GSQL	
GraphStudio	GraphStudio 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 and Cluster Expansion User Privileges and Authentication MultiGraph LDAP and Single Sign-On Data Encryption Admin Portal Backup and Restore	

TigerGraph Cloud

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

TigerGraph Cloud is built on the same TigerGraph Enterprise Server engine that is delivering the fastest and most scalable graph database.

Generally speaking, most of the documentation for TigerGraph Enterprise Server also applies to TigerGraph Cloud. However, since TigerGraph Cloud does not have a command line interface, any command line features are not supported in the same way. In many cases, an alternative GUI-based method has been introduced.

See the TigerGraph Cloud FAQs for answers to common questions.

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 v1 and TigerGraph Distributed Cloud is based on TigerGraph Server 2.5
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	TigerGraph Distributed Cloud does not support MultiGraph.
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	TigerGraph Distributed Cloud supports only a single user.
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 and 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	N/A. The operations are 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

TigerGraph Cloud FAQs

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

We welcome your feedback for TigerGraph Cloud 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. In addition, you can now provision a distributed database, as well as a replica cluster for high availability. 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 a 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: 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 credit 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. Please note, however, that 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 30 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 regions does TigerGraph operate?

A: 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). More regions will be added

soon.

Q: Does TigerGraph Cloud support distributed databases?

A: Yes. In the latest version of TigerGraph Cloud, you can provision a distributed TigerGraph cluster. You may select a partitioning factor up to 10. If you need more than more than 10 instances, please contact TigerGraph at sales@tigergraph.com ↗

Q: Does TigerGraph Cloud support HA?

A: Yes. In the latest version of TigerGraph Cloud, you can provision a highly available TigerGraph cluster by entering 2 for the replication factor during the provisioning process. Future versions of TigerGraph Distributed Cloud will support higher replication factors. 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 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 services are included?

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 built-in queries directly after my Solution has been 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. Currently, TigerGraph Cloud offers instances on the AWS platform. More cloud platforms are coming soon. 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. See the Support Policy terms at https://www.tigergraph.com/support-policy/ ↗

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.

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 at docs.tigergraph.com/intro/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: TigerGraph's MultiGraph feature, available now in the on-premises TigerGraph Enterprise product, will be coming to TigerGraph Cloud soon.

Q: What methods do you support for importing data?

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

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 credit 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 credit 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 credit 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 data will not be preserved. Please see the "Provisioning, Backup and Restore" section for a detailed explanation of 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.

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: I signed up for a TigerGraph Cloud account early, before distributed and HA services were available. Can I still use my earlier solutions? Can I upgrade my solutions to distributed or HA?

A: Our charter TigerGraph Cloud users can access their earlier solutions at our original URL: tgcloud.us ↗. The latest release TigerGraph Distributed Cloud can be accessed at tgcloud.io ↗. We will also be posting additional FAQs in that portal especially for our chapter members.

Pricing

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

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: What will I have to pay if I delete my instance before the end of the month?

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: How do I contact you if I have a billing problem that I can't solve online?

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.

Sizing and Scaling

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

TigerGraph Distributed Cloud offers seven different instances, ranging from seven vCPUs with 15 GiB RAM, to 96 vCPUs with 768 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 considerable. 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.

Contact support@tigergraph.com ↗ for more help.

Q: How do I expand the disk size?

Please contact support@tigergraph.com ↗.

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: Distributed database support is not yet available on TigerGraph Cloud, but it is available through our Managed Services or a deployment of TigerGraph Enterprise on your own cloud account.

Hardware

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

A: A Root Disk (EBS based) is attached to TigerGraph Instances. External EBS drives will be available soon.

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. 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?

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 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. 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) or two. Higher replication factors will be supported in the future.

Limitation: In TigerGraph Distributed Cloud, if replication is used, the total number of instances must be at least 3. If replication factor is 2, then the partition factor must be at least 2, for a total of 2×2 = 4 instances.

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 and Security

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

A: By default, you will be given your own VPC(s) for your TigerGraph Cloud account. Your instances are separated from other accounts by different VPCs. Within your own account, you have different VPCs 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: Can I use TigerGraph Cloud to do VPC peering?

A: The current version of TigerGraph Cloud does not support VPC peering. The feature is coming soon. If you want VPC peering, contact sales@tigergraph.com ↗ for managed services.

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

A: This is mandatory for your security. The default username/password is tigergraph/tigergraph. Please change the password to protect your system. The ability to change a solution's password is provided through the Admin Portal. Go to GraphStudio > Admin Portal > User 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 data 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 are 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.tgcloud.io:9000/graph/MyGraph/vertices/Account?limit=3'`

Q: Does TigerGraph Cloud offer Role Based Access Control?

TigerGraph's role-based access control with MultiGraph will be available in a future update.

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 integrate TigerGraph Cloud into an SSO system will be provided at a future date.

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.

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. ↗

TigerGraph Cloud V1 to TigerGraph Distributed Cloud FAQs

Q: What's the difference between TigerGraph Cloud V1 and TigerGraph Distributed Cloud?

TigerGraph Cloud V1 was released in September 25th, 2019, it supports single server TigerGraph solutions.

TigerGraph Distributed Cloud is released on January 8th, 2020. It is the latest version of TigerGraph Database-as-a-Service product with the additional capability to provision distributed and highly available TigerGraph Clusters. TigerGraph Distributed Cloud is the next generation service to replace TigerGraph Cloud V1.

TigerGraph Cloud V1 and TigerGraph Distributed Cloud have two different backend systems. They will operate concurrently until TigerGraph Cloud V1 is deprecated at the end of Q1, 2020. The detailed deprecation policy for TigerGraph Cloud V1 will be sent to users in January 2020.

Q: What’s the URL for TigerGraph Distributed Cloud?

A: The latest TigerGraph Distributed Cloud is tgcloud.io↗

The charter TigerGraph Cloud V1 is tgcloud.us↗

Q: What are the new features in TigerGraph Distributed Cloud?

- Option to provision a Highly Availability Solution via replication
- Option to provision a Distributed TigerGraph Solution at Provisioning
- Enhanced Cloud Portal Solution Creation Workflow
- EFS as backup storage
- More and better Starter Kits

Q: I already have a login for TigerGraph Cloud V1. Do I need to create a new login for the latest version TigerGraph Distributed Cloud?

A: Yes. TigerGraph Cloud V1 and TigerGraph Distributed Cloud have two different backend systems. You need to register for a new account in TigerGraph Distributed Cloud via tgcloud.io ↗.

In TigerGraph Distributed Cloud, the login system is managed by AUTH0. Therefore, you can login directly through your Google account or your Linkedin account. If you use another email address, you need to register again.

Q: Will my solutions in V1 show up in the latest TigerGraph Distributed Cloud?

A: No. TigerGraph Cloud V1 and TigerGraph Distributed Cloud have two different backend systems. Your solutions in V1 platform will not automatically show up in the new Distributed Cloud. You can export your solutions (which saves your graph and queries) to your local system, and import into your solutions provisioned in the new account in TigerGraph Distributed Cloud. Please refer to the guide described here ↗. To transfer data, you can write GSQL queries which print all your data to files.

Q: How do I move my solutions in TigerGraph Cloud V1 to the latest TigerGraph Distributed Cloud?

A: You can export your solutions (which saves your graph and queries) to your local system, and import into your solutions provisioned in the new account in TigerGraph Distributed Cloud. Please refer to the guide described here ↗. To transfer data, you can write GSQL queries which print all your data to files.

Q: Will you deprecate TigerGraph Cloud V1?

A: Yes. Please see the following key dates for the deprecation process:

- **January 10, 2020:** TigerGraph Distributed Cloud is available.

- **January 31, 2020:** If you have not already created a new account in TigerGraph Distributed Cloud, your legacy TigerGraph Cloud v1 account will be automatically imported into TigerGraph Distributed Cloud. We will send a password reset email for you to set up a new password for TigerGraph Distributed Cloud. Please let us know if you want to opt out of new account import from TigerGraph Cloud V1 to TigerGraph Distributed Cloud. Note that your solutions will not be automatically migrated into the newly released TigerGraph Distributed Cloud.
- **February 24, 2020:** TigerGraph Cloud V1 stops accepting new instances.
- **March 31, 2020:** Cloud V1 ends.

Q: I can see Credit Card information pre-loaded in my account in TigerGraph Distributed Cloud. Why?

A: We are using Stripe for the payment system. If you are using the same V1 email address as the new login email address in TigerGraph Distributed Cloud, Stripe has a record of your credit card from TigerGraph V1. Hence your credit card information will be populated in the account information. You can enter a different credit card first and delete the old credit card if you prefer.

Comparing TigerGraph Editions

Updated 4 Feb 2020

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

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

Overview

OVERVIEW	**Developer Edition**	TigerGraph Cloud	Enterprise Edition
Licensing	Free for non-production, research, or educational use	Pay as you go by the minute; Annual contracts available	Free 30-day trial
Includes	"TigerGraph For One" single graph, single machine	The power of TigerGraph, as a Service + Instant Deployment + Automatic backups + Scale Out & Replication + Security + Pay for what you use	All TigerGraph Database and Enterprise features, including + Distributed Graph + MultiGraph + Security + User Management
Support	Community forum ↗	✅	✅
How to Get It	www.tigergraph.com/developer ↗	www.tigergraph.com/cloud ↗	www.tigergraph.com/free-trial ↗

Database Features

DATABASE FEATURES	Developer Edition	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	500 billion edges	Unlimited	Unlimited
Compressed Data Store	✅	✅	✅
In-Memory Processing; ACID Transactions	✅	✅	✅
Distributed, Auto-Partitioned Graph	✅	✅	✅
MultiGraph	Coming soon	Coming soon	✅
Dynamic Schema Change	Coming soon	✅	✅

* = optional

Graphical User Interface

GRAPHICAL USER INTERFACE	Developer Edition	TigerGraph Cloud	Enterprise Edition

GraphStudio Visual SDK and UI Design, Load, Explore, Query, Visualize	✅	✅	✅ *
Admin Portal	• Monitor system operation	• Monitor system operation • Manage credentials • Backup & Restore	• Monitor system operation • Manage licenses

Enterprise Features

ENTERPRISE FEATURES	Developer Edition	TigerGraph Cloud	Enterprise Edition
Automated Deployment	No	✅	No
Automated Backups	No	✅	No
Backup and Restore	Coming soon	✅	✅
Multiple Users	No	✅	✅
HA Replication	No	✅	✅

Security Features

ENTERPRISE FEATURES	Developer Edition	TigerGraph Cloud	Enterprise Edition
Data Encryption At Rest and In Motion	No	✅	✅

Enterprise User Management LDAP and SSO	No	Coming soon	✅
Role-Based Access Control	No	Coming soon	✅
Audit Compliance	No	✅ SOC 2 type 1	Coming soon
Cloud Security: VPC for each account	n/a	✅	n/a

GSQL Graph Algorithm Library

Updated May 11, 2020

Graph algorithms are functions for measuring characteristics of graphs, vertices, or relationships. Graph algorithms can provide insights into the role or relevance of individual entities in a graph. For example: How centrally located is this vertex? How much influence does this vertex exert over the others?

Some graph algorithms measure or identify global characteristics: What are the natural community groupings in the graph? What is the density of connections?

Using the GSQL Graph Algorithm Library

The GSQL Graph Algorithm 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.

GSQL running on the TigerGraph platform is particularly well-suited for graph algorithms for the 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-algorithms ↗

The library contains two main sections: algorithms and tests. The algorithms folder contains template algorithms and scripts to help you customize and install them. The tests 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.

Installing an Algorithm

Remember that GSQL graph algorithms are simply GSQL queries. However, since we do not know what vertices or edges you want to analyze, or how you want to receive output, the algorithms are in a template format. You need to run a script to personalize your algorithm and then to install it.

> Make sure that the install.sh is owned by the tigergraph user.

1. Within the Algorithms folder is a script **install.sh**. When you run the script, it will first ask you which graph schema you wish to work on. (The TigerGraph platform supports multiple concurrent graphs.)
2. It then asks you to choose from a menu of available algorithms.
3. After knowing your graph schema and your algorithm, the installer will ask you some questions for that particular algorithm:
    a. the installer will guide you in selecting appropriate vertex types and edges types.

    Note this does not have to be all the vertex or edge types in your graph. For example, you may have a social graph with three categories of persons and five types of relationships. You might decide to compute PageRank using Member and Guest vertices and Recommended edges.

    b. Some algorithms use edge weights as input information (such as Shortest Path where each edge has a weight meaning the "length" of that edge. The installer will ask for the name of that edge attribute.
4. Single Node Mode or Distributed Mode? Queries which will analyze the entire graph (such PageRank and Community Detection) will run better in Distributed Mode, if you have a cluster of machines.

5. It will then ask you what type of output you would like. It will proceed to create up to three versions of your algorithm, based on the three ways of receiving the algorithm's output:
    a. Stream the output in JSON format, the default behavior for most GSQL queries.
    b. Save the output value(s) in CSV format to a file. For some algorithms, this option will add an input parameter to the query, to let the user specify how many total values to output.
    c. Store the results as vertex or edge attribute values. The attributes must already exist in the graph schema, and the installer will ask you which attributes to use.
6. After creating queries for one algorithm, the installer will loop back to let you choose another algorithm (returning to step 2 above).
7. If you choose to exit, the installer makes a last request: Do you want to install your queries? Installation is when the code is compiled and bound into the query engine. It takes a few minutes, so it is best to create all your personalized queries at once and then install them as a group.

Example:

```
$ bash install.sh
*** GSQL Graph Algorithm Installer ***
Available graphs:
  - Graph social(Person:v, Friend:e, Also_Friend:e, Coworker:e)
Graph name? social

Please enter the number of the algorithm to install:
 1) EXIT
 2) Closeness Centrality
 3) Connected Components
 4) Strongly Connected Components
 5) Label Propagation
 6) Louvain Method with Parallelism and Refinement
 7) PageRank
 8) Weighted PageRank
 9) Personalized PageRank
10) Shortest Path, Single-Source, No Weight
11) Shortest Path, Single-Source, Positive Weight
12) Shortest Path, Single-Source, Any Weight
13) Minimal Spanning Tree (MST)
14) Cycle Detection
15) Triangle Counting(minimal memory)
16) Triangle Counting(fast, more memory)
17) Cosine Neighbor Similarity (single vertex)
18) Cosine Neighbor Similarity (all vertices)
19) Jaccard Neighbor Similarity (single vertex)
20) Jaccard Neighbor Similarity (all vertices)
21) k-Nearest Neighbors (Cosine Neighbor Similarity, single vertex)
22) k-Nearest Neighbors (Cosine Neighbor Similarity, batch)
23) k-Nearest Neighbors Cross Validation (Cosine Neighbor Similarity)
#? 7
  pageRank() works on directed edges

Available vertex and edge types:
  - VERTEX Person(PRIMARY_ID id STRING, name STRING, score FLOAT, tag STRI
  - DIRECTED EDGE Friend(FROM Person, TO Person, weight FLOAT, tag STRING)
  - DIRECTED EDGE Also_Friend(FROM Person, TO Person, weight FLOAT, tag ST
  - UNDIRECTED EDGE Coworker(FROM Person, TO Person, weight FLOAT, tag STR

Please enter the vertex type(s) and edge type(s) for running PageRank.
   Use commas to separate multiple types [ex: type1, type2]
   Leaving this blank will select all available types
 Similarity algorithms only take single vertex type

Vertex types: Person
Edge types: Friend
The query pageRank is dropped.
The query pageRank_file is dropped.
```

```
The query pageRank_attr is dropped.

Please choose query mode:
1) Single Node Mode
2) Distributed Mode
#? 1
```

```
GSQL > ls
...
Queries:
  - cc_subquery(vertex v, int numVert, int maxHops) (installed v2)
  - closeness_cent(bool display, int outputLimit) (installed v2)
  - closeness_cent_attr() (installed v2)
  - closeness_cent_file(file f) (installed v2)
  - conn_comp() (installed v2)
  - conn_comp_attr() (installed v2)
  - conn_comp_file(file f) (installed v2)
  - label_prop(int maxIter) (installed v2)
  - label_prop_attr(int maxIter) (installed v2)
  - label_prop_file(int maxIter, file f) (installed v2)
  - louvain() (installed v2)
  - louvain_attr() (installed v2)
  - louvain_file(file f) (installed v2)
  - pageRank(float maxChange, int maxIter, float damping, bool display, ir
  - pageRank_attr(float maxChange, int maxIter, float damping, bool displa
  - pageRank_file(float maxChange, int maxIter, float damping, bool displa
  - tri_count() (installed v2)
  - tri_count_fast() (installed v2)
```

Running an Algorithm

```
Please enter the number of the algorithm to install:
1) EXIT
2) Closeness Centrality
3) Connected Components
4) Label Propagation
5) Community detection: Louvain
6) PageRank
7) Shortest Path, Single-Source, Any Weight
8) Triangle Counting(minimal memory)
9) Triangle Counting(fast, more memory)
#? 1
Exiting
```

```
GSQL > RUN QUERY pageRank(0.001, 25, 0.85, 10)
```

```
c
pageRank query: curl -X GET 'http://127.0.0.1:9000/query/social/pageRank?r
pageRank_file query: curl -X GET 'http://127.0.0.1:9000/query/social/pageR
pageRank_attr query: curl -X GET 'http://127.0.0.1:9000/query/social/pageR
```

```
curl -X GET 'http://127.0.0.1:9000/query/alg_graph/pageRank?maxChange=0.00
```

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.

Library Overview

The following algorithms are currently available. The algorithms are grouped into five classes:

- Path
- Centrality
- Community
- Similarity
- Classification (NEW)

Moreover, each algorithm may or may be be currently available for a graph with Undirected Edges, Directed Edges, and Weighted Edges.

- **Coming soon** means that TigerGraph plans to release this variant of the algorithm soon.
- **n/a** means that this variant of the algorithm is typically not used

Algorithm	Class	Undirected Edges	Directed Edges	Weighted Edges
Single-Source Shortest Path	Path	Yes	Yes	Yes
All Pairs Shortest Path	Path	Yes	Yes	Yes
Minimum Spanning Tree or Forest	Path	Yes	n/a	Yes

Cycle Detection	Path	no	Yes	n/a
PageRank	Centrality	n/a	Yes	Yes
Personalized PageRank	Centrality	n/a	Yes	Coming soon
Closeness Centrality	Centrality	Yes	n/a	Coming soon
Betweenness Centrality	Centrality	Yes (NEW)	n/a	Coming soon
(Weakly) Connected Components	Community	Yes (Improved)	n/a	n/a
Strongly Connected Components	Community	n/a	Yes, with reverse edges (NEW)	n/a
Label Propagation	Community	Yes	n/a	n/a
Louvain Modularity	Community	Yes	n/a	n/a
Triangle Counting	Community	Yes	n/a	n/a
Cosine Similarity of Neighborhoods (single-source and all-pairs)	Similarity	Yes	Yes	Yes
Jaccard Similarity of Neighborhoods (single-source and all-pairs)	Similarity	Yes	Yes	No
K-Nearest Neighbors (with cosine	Classification	Yes	Yes	Yes

Computational Complexity

Computational Complexity is a formal mathematical term, referring to how an algorithm's time requirements scale according to the size of the data or other key parameters. For graphs, there are two key data parameters:

- V (or sometimes n), the number of vertices
- E (or sometimes m), the number of edges

The notation O(V^2) (read "big O V squared") means that when V is large, the computational time is proportional to V^2.

Path Algorithms

These algorithms help find the shortest path or evaluate the availability and quality of routes.

Single-Source Shortest Path, Unweighted

> The algorithm we are discussing here finds an unweighted shortest path from one source vertex to each possible destination vertex in the graph. That is, it finds n paths.
>
> If you just want to know the shortest path between two particular vertices, S and T in a graph with unweighted edges, we have described that query in detail in our tutorial document GSQL Demo Examples ↗.
>
> If your graph has weighted edges, see the next algorithm.

Description and Uses

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.

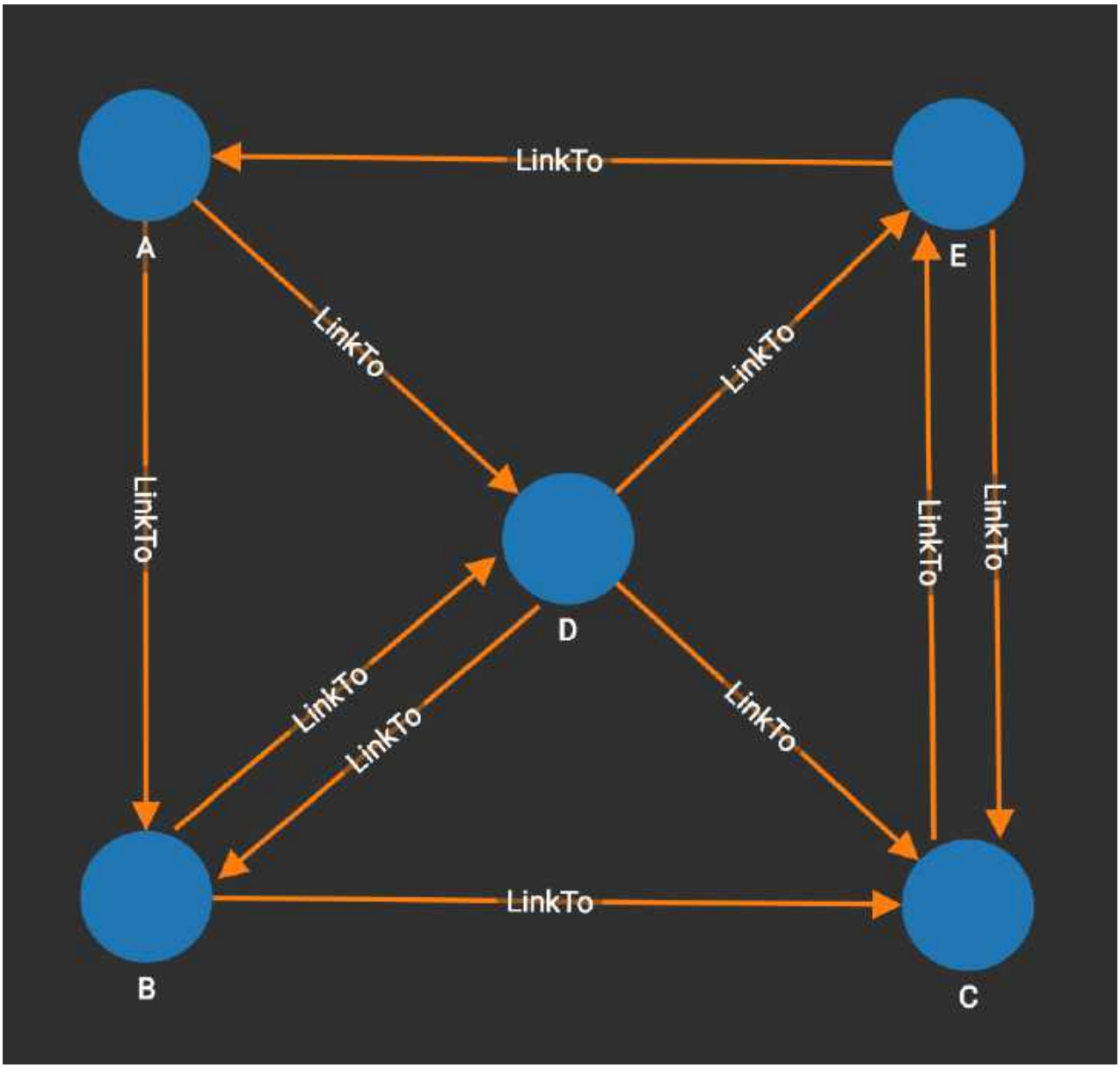
Specifications

```
shortest_ss_no_wt(VERTEX v, BOOL display)
shortest_ss_no_wt_file(VERTEX v, BOOL display, STRING filepath)
shortest_ss_no_wt_attr(VERTEX v, BOOL display)
```

Characteristic	Value
Result	Computes a shortest distance (INT) and shortest path (STRING) from vertex v to each other vertex T. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as two vertex attribute values.
Input Parameters	• **v:** id of the source vertex • **display**: If true, include the graph's edges in the JSON output, so that the full graph can be displayed. • **filepath** (for file output only): the path to the output file
Result Size	V = number of vertices
Computational Complexity	O(E), E = number of edges
Graph Types	Directed or Undirected edges, Unweighted edges

Example

In the below graph, we do not consider the weight on edge. 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 shortest_pos5 choice, not considering weight

```
[
  {
    "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

Description and Uses

Finding shortest paths in a graph with weighted edges is algorithmically harder than in an unweighted graph because just because 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

```
shortest_path_pos_wt(VERTEX v, BOOL display)
shortest_path_pos_wt_file(VERTEX v, BOOL display, STRING filepath)
shortest_path_pos_wt_attr(VERTEX v, BOOL display)
```

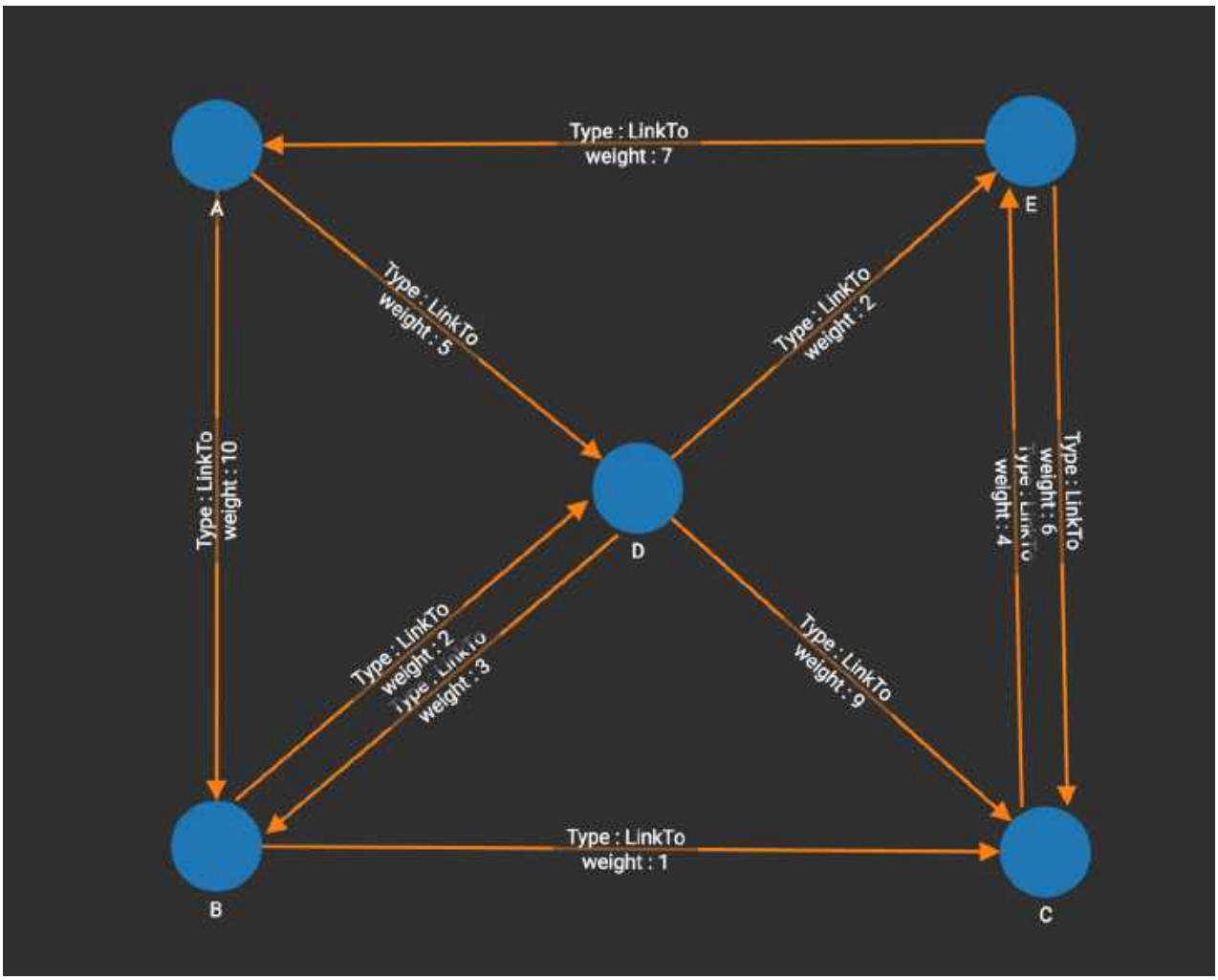
```
shortest_path_neg_wt(VERTEX v, BOOL display)
shortest_path_neg_wt_file(VERTEX v, BOOL display, STRING filepath)
shortest_path_neg_wt_attr(VERTEX v, BOOL display)
```

Characteristic	Value
Result	Computes a shortest distance (INT) and shortest path (STRING) from vertex v to each other vertex T. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as two vertex attribute values.
Input Parameters	• **v:** id of the source vertex • **display**: If true, include the graph's edges in the JSON output, so that the full graph can be displayed. • **filepath** (for file output only): the path to the output file
Result Size	V = number of vertices
Computational Complexity	O(V*E), V = number of vertices, E = number of edges
Graph Types	Directed or Undirected edges, Weighted edges

The shortest_path_neg_wt library 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.

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.

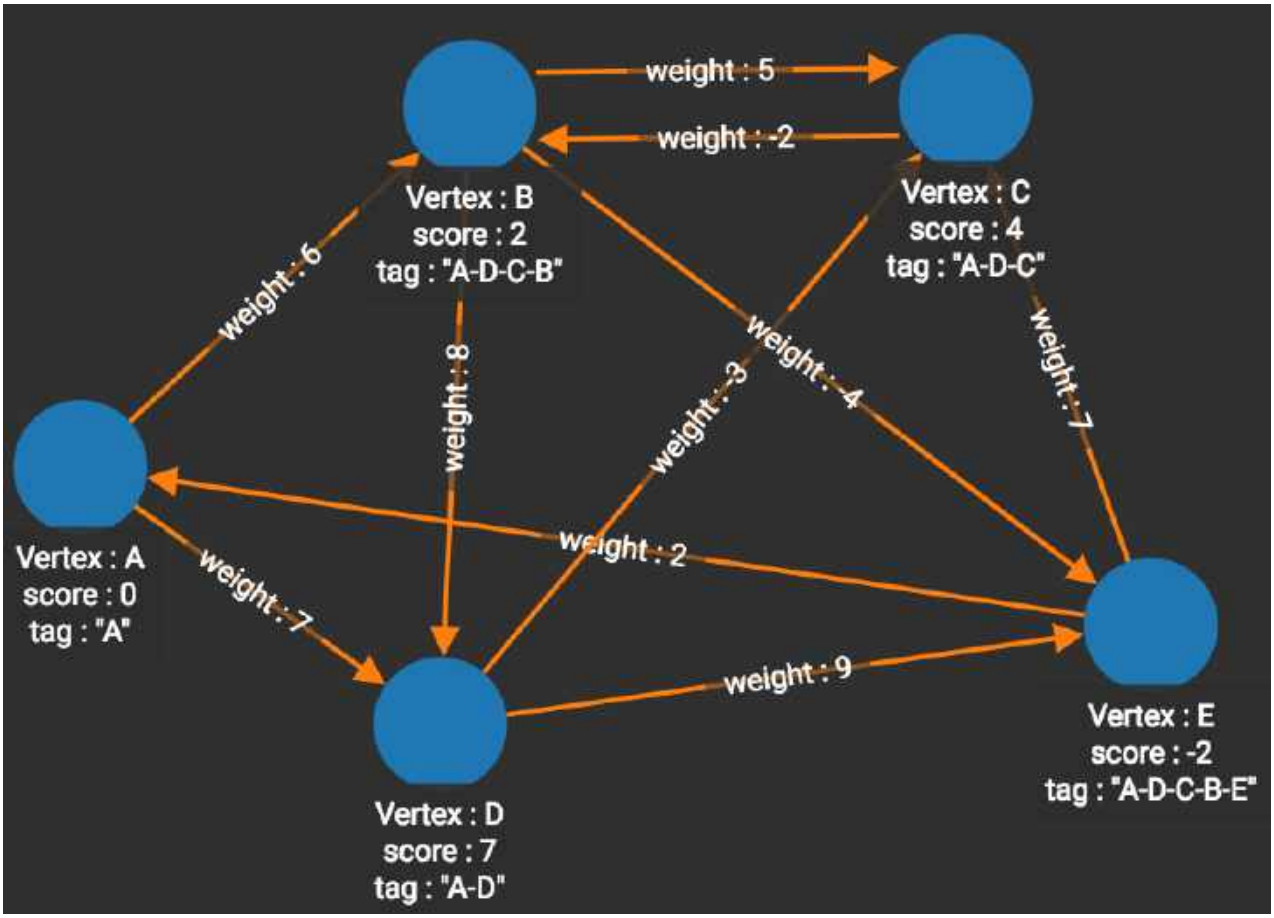


generic graph with shortest_pos5 choice

```
[
  {
    "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.



shortest_path_wt(A, -1, true, "json") on shortest_neg5 graph

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 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 all_pairs_shortest(INT maxDepth, BOOL display, STRING fileBas
{
  Start = {Node.*};
  Result = SELECT s FROM Start:s
        POST-ACCUM
          shortest_ss_any_wt_file(s, maxDepth, display, fileBase+s);
}
```

This example highlights one of the strengths of GSQL: treating queries as stored procedures which can be called from within other queries.

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.

Minimum Spanning Tree (MST)

Description and Uses

Given an undirected and connected graph, a minimum spanning tree is a set of edges which can connect all the vertices in the graph with the minimum sum of edge weights. A parallel version of Prim's algorithm is implemented in the library:

1. Start with a set A = { an arbitrary root vertex r }
2. Select a vertex adjacent to A which has the lowest weight edge connecting it to A. That is, find a vertex y where (1) y is not in A, and (2) y is connected to a vertex x in A such that the weight on the edge e(x,y) is the smallest among all such edges from a vertex in A to a vertex not in A. Add y to A, and add the edge (x,y) to MST.
3. Repeat 2 until A has all vertices in the graph.

> If the graph contains some vertices which are separated from others (that is, there is no path from one to the other), then the MST solution applies only to the part of the graph that it happens to start with. It does not guarantee to work on the largest component. To find a set of minimal spanning trees which cover all the components of a separated graph, use the Minimum Spanning Forest (MSF) algorithm, described after MST.

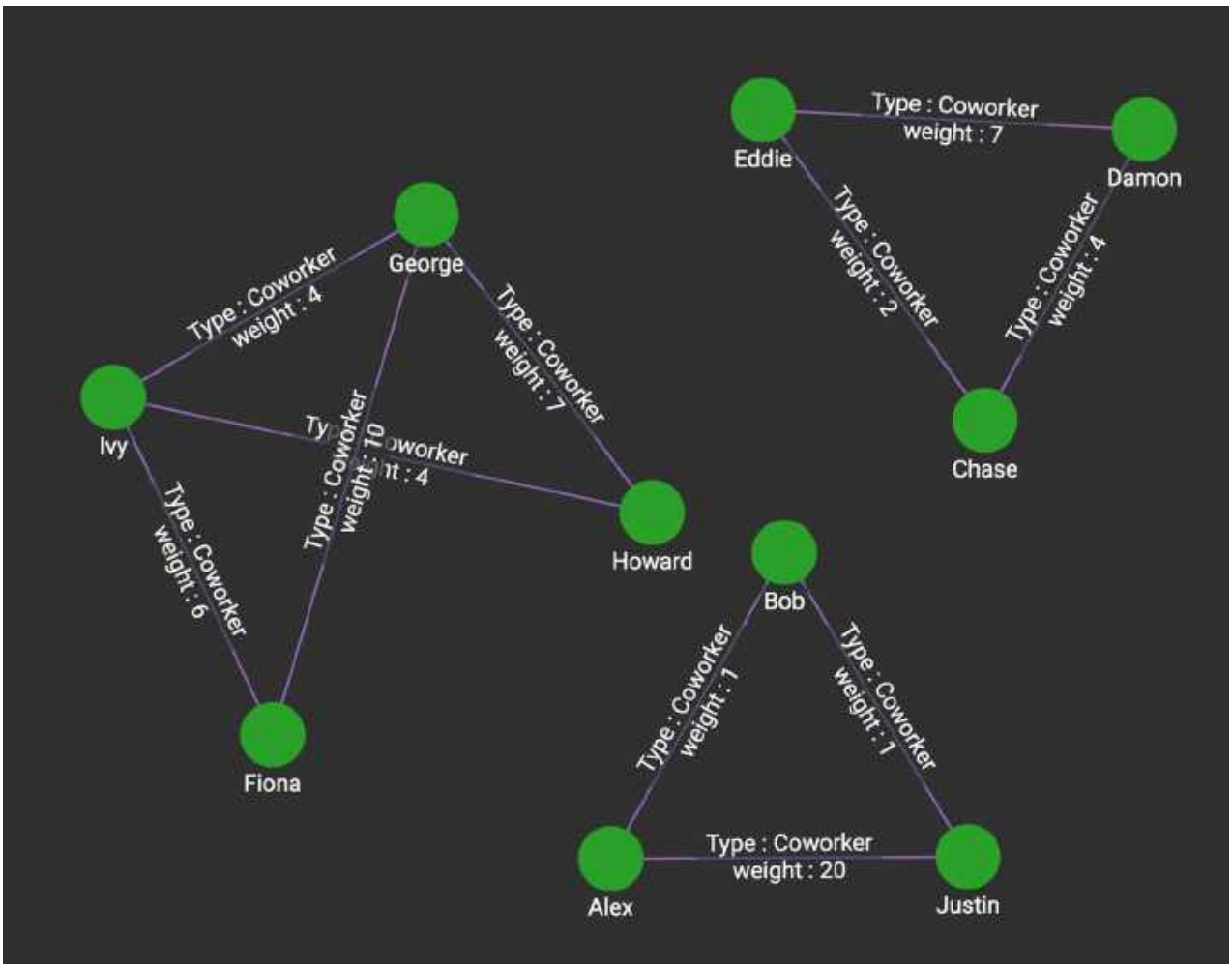
Specifications

```
mst (VERTEX source)
mst_file (VERTEX source, FILE f)
mst_attr (VERTEX source)
```

Characteristic	Value
Result	Returns a set of edges which form a MST. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as an edge attribute value (marking the members of the MST).
Input Parameters	• **source:** id of the source vertex • **filepath** (for file output only): the path to the output file
Result Size	V - 1 = number of vertices - 1
Computational Complexity	O(E * log V), E = number of edges, V = number of vertices
Graph Types	Undirected weighted edges and connected graph

Example

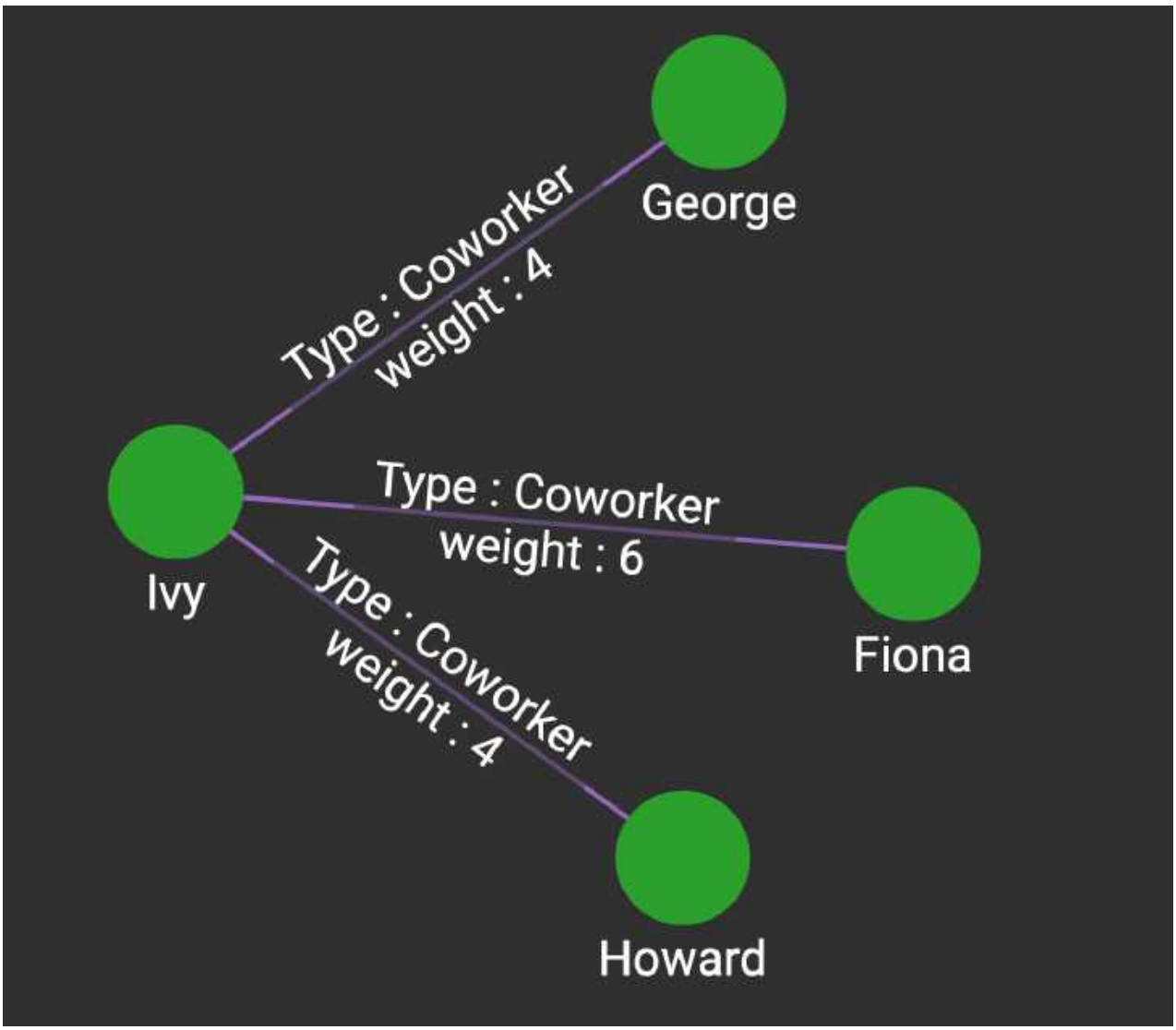
In social10 graph, we consider only the undirected Coworker edges.



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 mst(("Ivy", "Person")). 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.

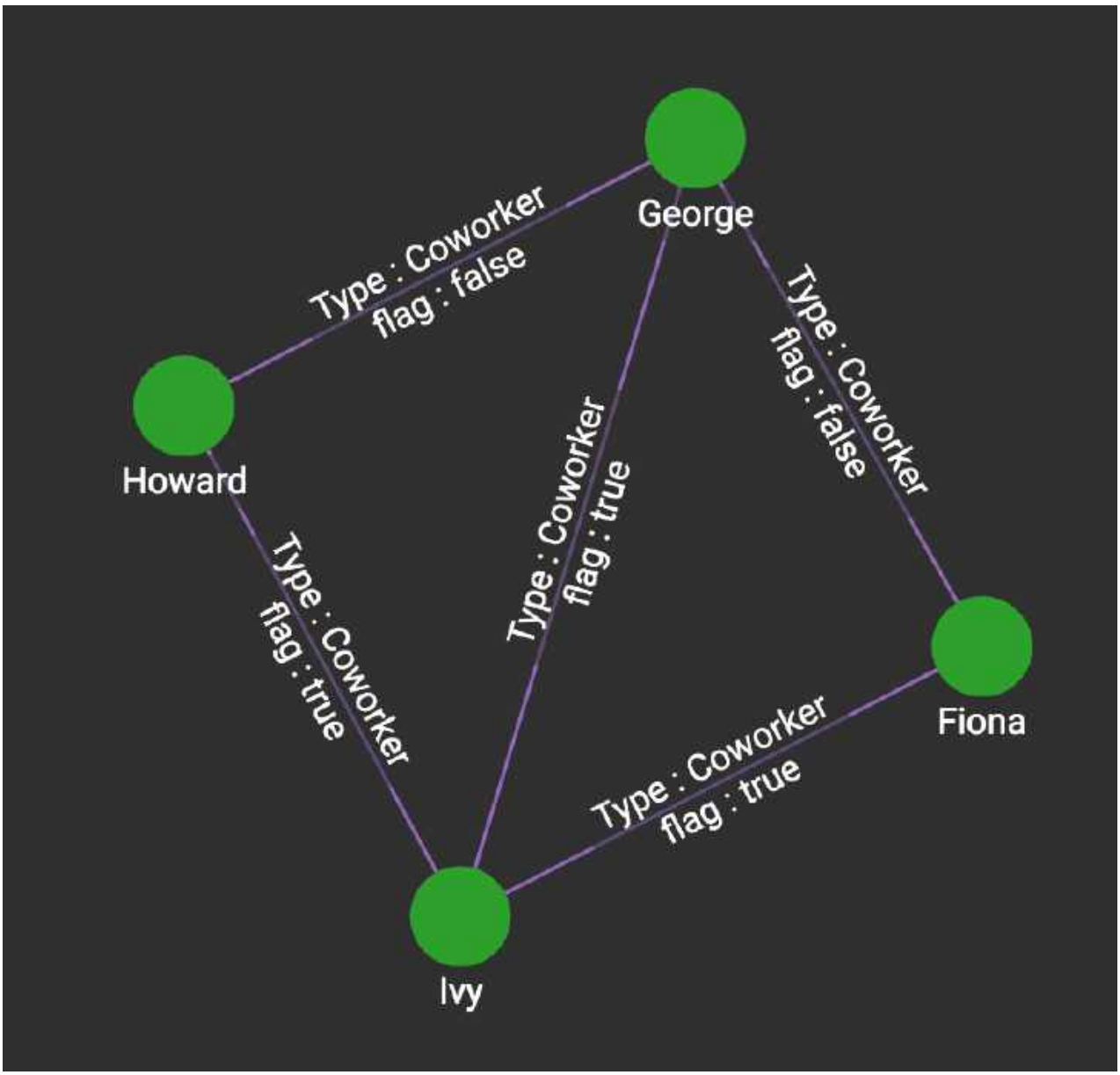


mst(("Ivy","Person")) on social10 graph, with Coworker edges

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:



mst_attr(("Ivy","Person")) on social10 graph, with Coworker edges & edge attribute "flag"

Minimum Spanning Forest (MSF)

Description and Uses

In graph theory terminology, 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. If it has only one component, we say the graph is *connected*. Given an undirected and connected graph, a minimum spanning tree (MST) is a set of edges which can connect all the vertices in one component with the minimal sum of edge weights. If the graph consists of multiple components, then

a collection of MSTs, one for each component, is known as a minimum spanning forest (MSF).

While Prim's algorithm is good when the graph has only one component, if the graph might have more components, then an MSF algorithm should be used. Our library mplements a parallel version of Boruvka's/Sollin's Algorithm [1]:

1. Consider each vertex in the graph as an independent subcomponent.
2. For each subcomponent, select the lowest-weighted edge that leaves the subcomponent and add it to the MST.
3. Merge newly-connected subcomponents into larger subcomponents.
4. Repeat 2 and 3 until only a single subcomponent remains for the graph.

Not only will the MSF algorithm generate the forest by adding multiple edges concurrently, but it is also not limited to a single component, so it can create the complete minimum spanning forest for the graph.

[1] Qin, Lu, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang and Xuemin Lin. "Scalable big graph processing in MapReduce." SIGMOD Conference (2014).

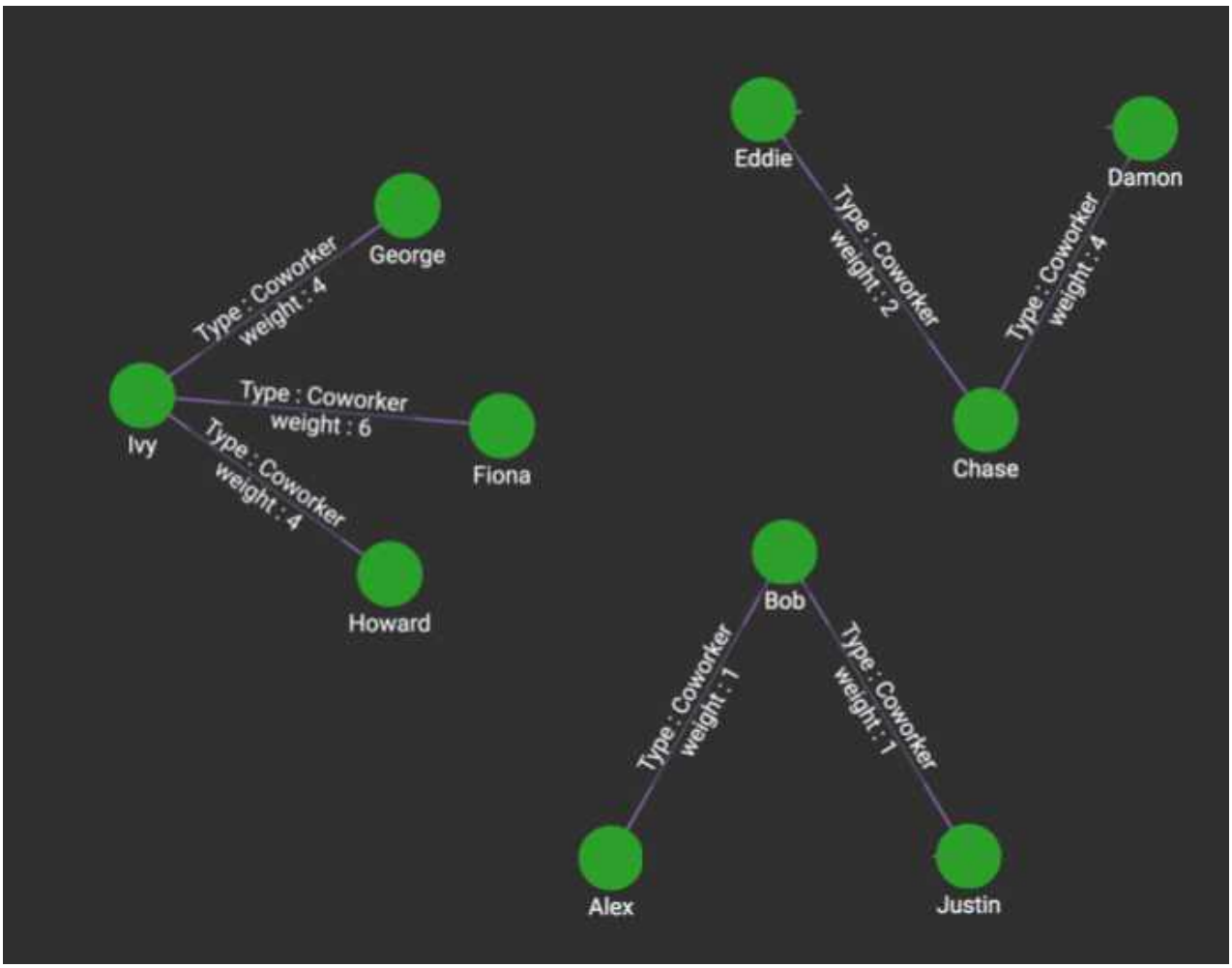
Specifications

```
msf ()
msf_file (FILE f)
msf_attr ()
```

Other details of the MSF algorithm are very much the same as that of the MST algorithm. Please refer to the Minimum Spanning Tree description for usage details and an example.

Example

The Minimum Spanning Tree example introduced the social10 graph, containing three components. The MSF algorithm will compute an MSF consisting of the following 3 MSTs:



Cycle Detection

Description and Uses

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:
Fo 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

```
cycle_detection (INT depth)
cycle_detection_file (INT depth, FILE f)
```

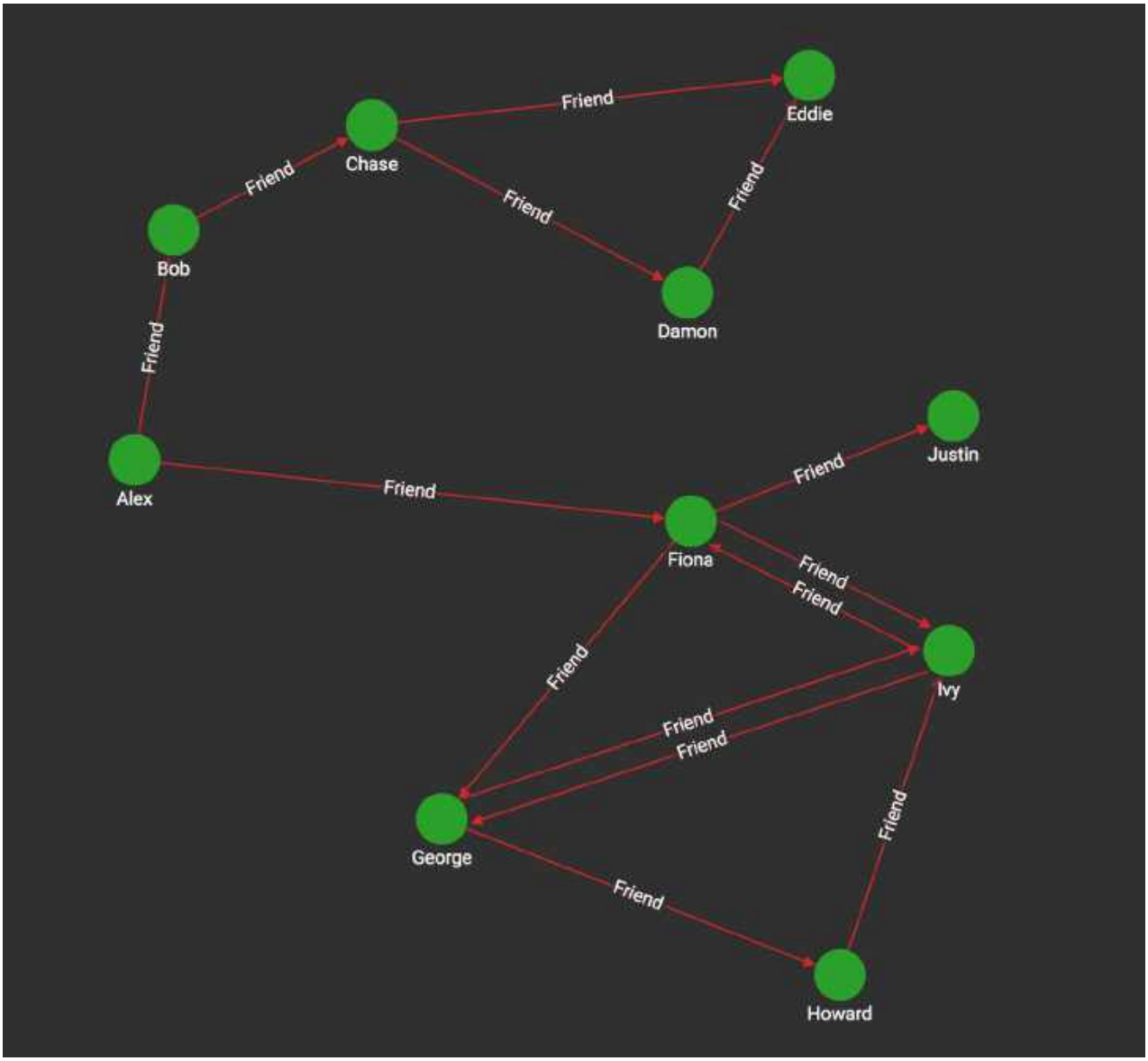
The algorithm traverses all edges. A user could modify the GSQL algorithm so that it traverse only edges of a certain type.

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	• **depth:** the maximum cycle length to search for = maximum number of iterations • **filepath** (for file output only): the path to the output file
Result Size	Number of cycles * average cycle length Both of these measures are not known in advance.
Computational Complexity	O(E *k), E = number of edges. k = min(max. cycle length, depth paramteter)

Example

In the social10 graph, there are 5 cycles, all with the Fiona-George-Howard-Ivy cluster.



cycle_detection(10) on social10 graph

```
[
  {
    "@@cycles": [
      [
        "Fiona",
        "Ivy"
      ],
      [
        "George",
        "Ivy"
      ],
      [
        "Fiona",
        "George",
        "Ivy"
      ],
      [
        "George",
        "Howard",
        "Ivy"
      ],
      [
        "Fiona",
        "George",
        "Howard",
        "Ivy"
      ]
    ]
  }
]
```


Centrality Algorithms

Centrality algorithms determine the importance of each vertex within a network. Typical applications:

PageRank is designed for directed edges. The classic interpretation is to find the most "important" web pages, based on hyperlink referrals, but it can be used for another network where entities make positive referrals of one another.

Closeness Centrality and Betweenness Centrality both deal with the idea of "centrally located."

PageRank

Description and Uses

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.

Specifications

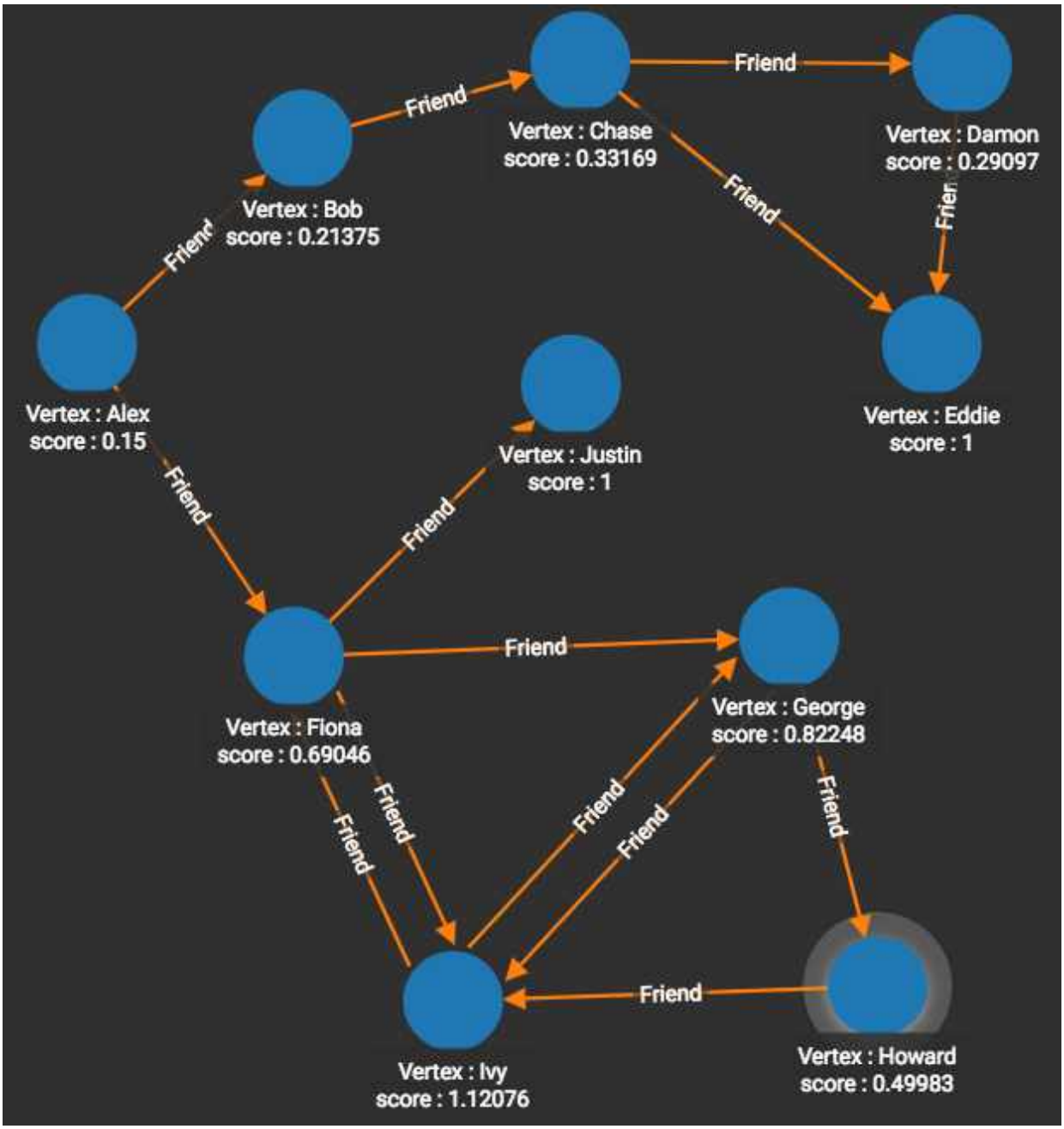
```
pageRank(FLOAT maxChange, INT maxIter, FLOAT damping, BOOL display, INT ou
pageRank_file(FLOAT maxChange, INT maxIter, FLOAT damping, FILE f)
pageRank_attr(FLOAT maxChange, INT maxIter, FLOAT damping)
```

Characteristic	Value

Result	Computes a PageRank value (FLOAT type) for each vertex. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• **maxChange**: PageRank will stop iterating when the largest difference between any vertex's current score and its previous score ≤ maxChange. That is, the scores have become very stable and are changing by less that maxChange from one iteration to the next. Suggested value: 0.001 or less. • **maxIter**: maximum number of iterations. Suggested value: between 10 and 100. • **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. Suggested value: 0.85. • **f** (for file output only): the path to the output file • **display** (for JSON output only): If true, include the graph's edges in the JSON output, so that the full graph can be displayed. • **outputLimit** (for JSON output only): maximum number of vertex values to output. Values will be sorted with highest value first.
Result Size	V = number of vertices
Computational 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

We ran pageRank on our test10 graph (using Friend edges) with the following parameter values: damping=0.85, maxChange=0.001, and maxIter=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 have 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.



pageRank_attr(0.001, 25, 0.85,"json",10) on social10 graph, with Friend edges

Personalized PageRank

Description and Uses

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

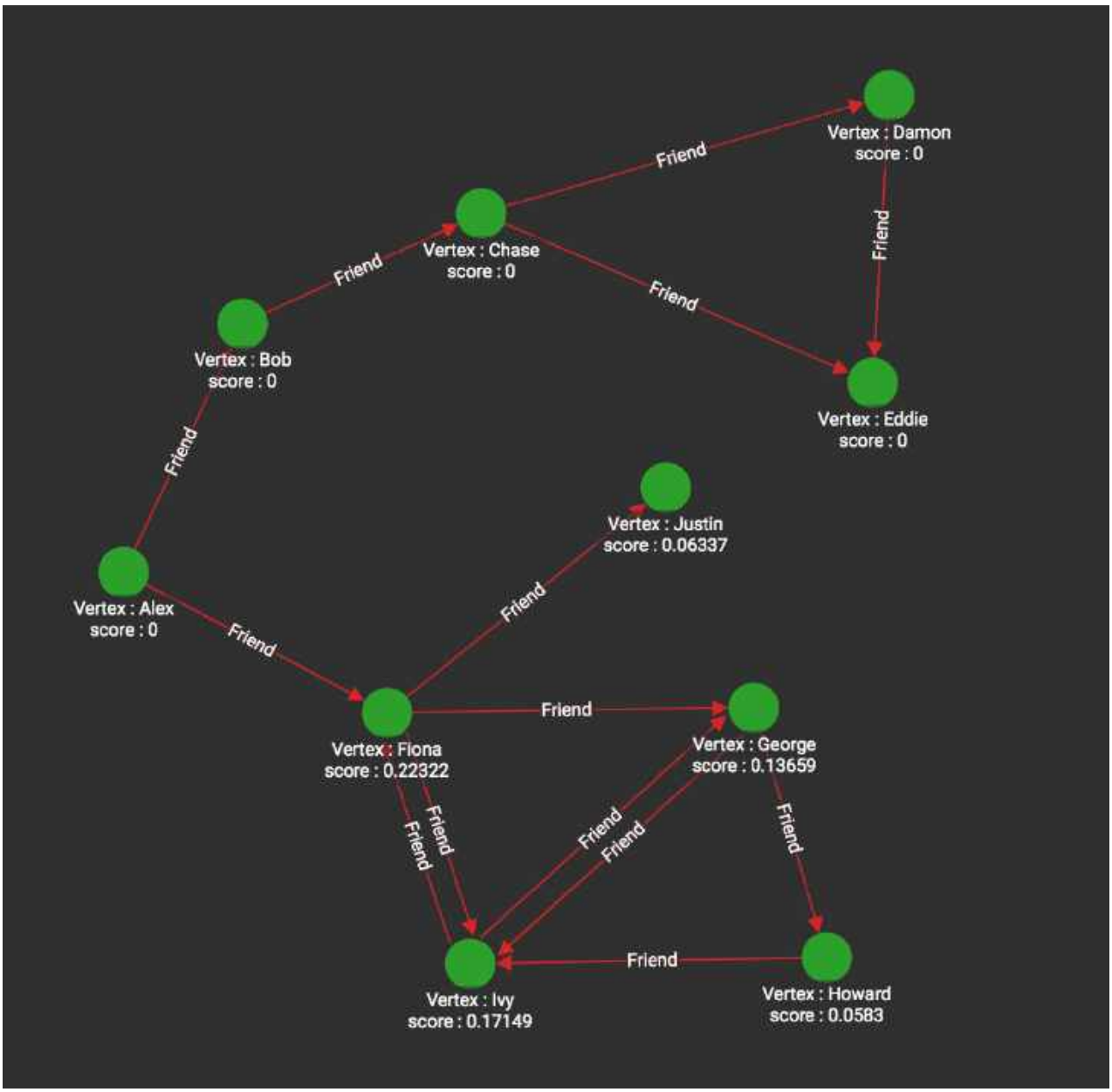
```
pageRank_pers(Set<Vertex> source, FLOAT maxChange, INT maxIter, FLOAT damp
pageRank_pers_file(Set<Vertex> source, FLOAT maxChange, INT maxIter, FLOAT
pageRank_pers_attr(Set<Vertex> source, FLOAT maxChange, INT maxIter, FLOAT
```

Characteristic	Value
Result	Computes a personalized PageRank value (FLOAT type) for each vertex. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• **source**: a set of source vertices • **maxChange**: personalized PageRank will stop iterating when the largest difference between any vertex's current score and its previous score ≤ maxChange. That is, the scores have become very stable and are changing by less that maxChange from one iteration to the next. Suggested value: 0.001 or less. • **maxIter**: maximum number of iterations. Suggested value: between 10 and 100. • **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. Suggested value: 0.85. • **f** (for file output only): the path to the output file • **outputLimit** (for JSON output only): maximum number of vertex values to output. Values will be sorted with highest value first.
Result Size	V = number of vertices
Computational 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 our test10 graph using Friend edges with the following parameter values: damping=0.85, maxChange=0.001, maxIter=25, and source="Fiona". 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.



pageRank_pers_attr([("Fiona","Person")],0.001,25,0.85) on social10 graph, with Friend edges

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" is a vertex. The steps below show the steps for one vertex v.

Description of Steps	Mathematical Formulation
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	$CC(v) = 1/d_{avg}(v)$

These steps are repeated for every vertex in the graph.

Specifications

```
closeness_cent(BOOL display, INT outputLimit)
closeness_cent_file(STRING filepath)
closeness_cent_attr()
```

Parameters

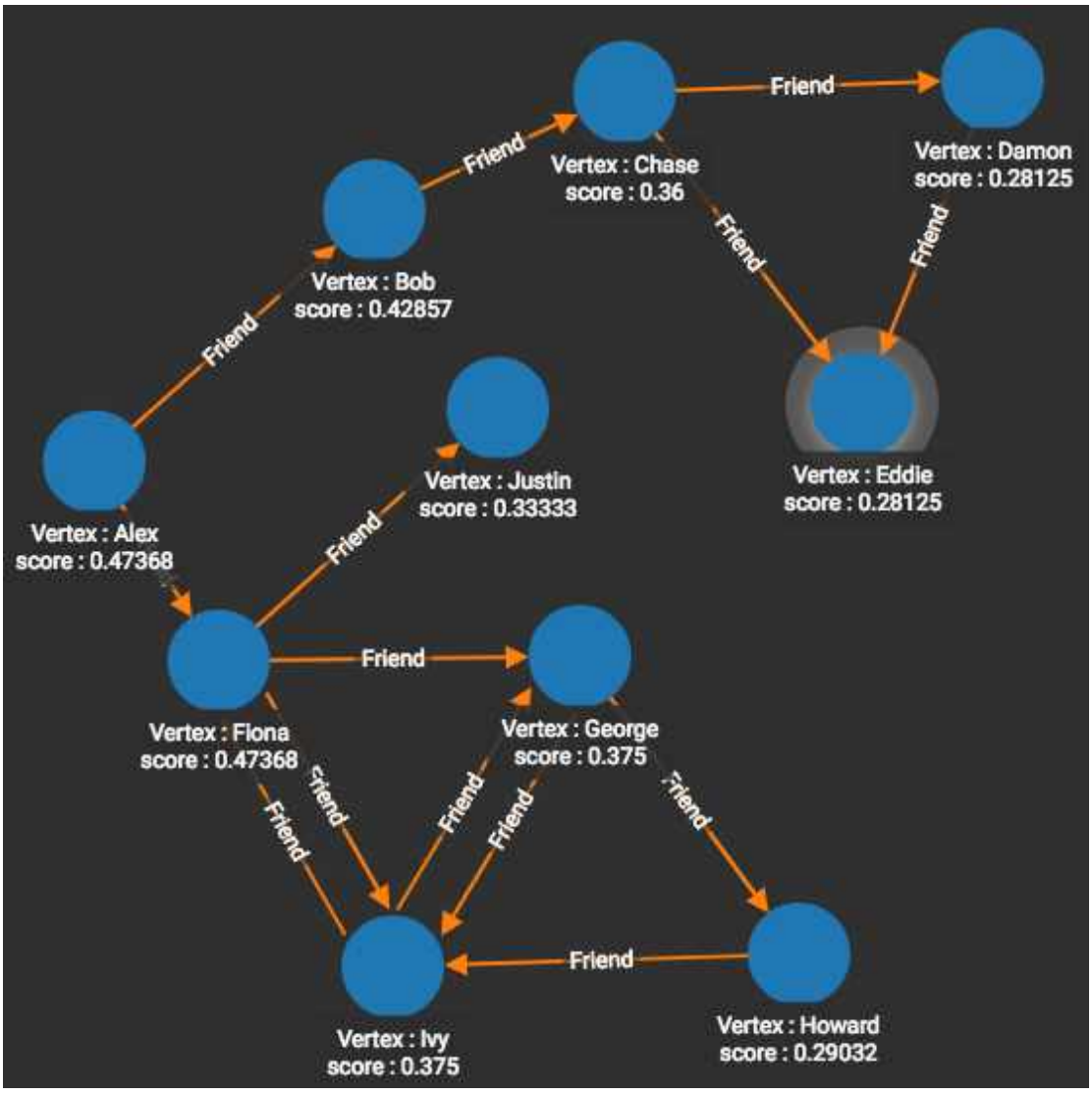
Characteristic	Value
Result	Computes a Closeness Centrality value (FLOAT type) for each vertex. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Required Input Parameters	• **display**: If true, include the graph's edges in the JSON output, so that the full graph can be displayed. **filepath** (for file output only): the path to the output file • **outputLimit** (for JSON output only): maximum number of vertex values to output. Values will be sorted with highest value first.
Result Size	V = number of vertices
Computational 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 or Undirected edges, Unweighted

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.



closeness_cent("json",10) on social10 graph, with Friend and Also_Friend edges

Betweenness Centrality

The Betweenness Centrality of a vertex is defined as the number of shortest paths which pass through this vertex, divided by the total number of shortest paths. That is

$$BC(v) = \sum_{s \neq v \neq t} \{PD_{st}(v)/2\} = \sum_{s \neq v \neq t} \{SP_{st}(v)/(2 \cdot SP_{st})\},$$

where $PD$is called the pair dependency, $SP_{st}$is the total number of shortest paths between nodes s and t, and $SP_{st}(v)$is the number of those paths that pass through v. The betweenness centrality above is defined for an undirected graph where the two directions of a path are counted as one by dividing the pair dependency by 2.

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

$$PD_{s*}(v) = \sum_{w:v \in P_s(w)} \{SP_{sv}/SP_{sw} \cdot (1 + PD_{s*}(w))\},$$

where the set of predecessors of vertex w on shortest paths from s $P_s(w)$is defined as

$$P_s(w) = \{u \in V : \{u, w\} \in E, dist(s, w) = dist(s, u) + dist(u, w)\}.$$

For each single 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.

Specifications

```
betweenness_cent(INT maxHops)
betweenness_cent_file(STRING filepath, INT maxHops)
betweenness_cent_attr(INT maxHops)
```

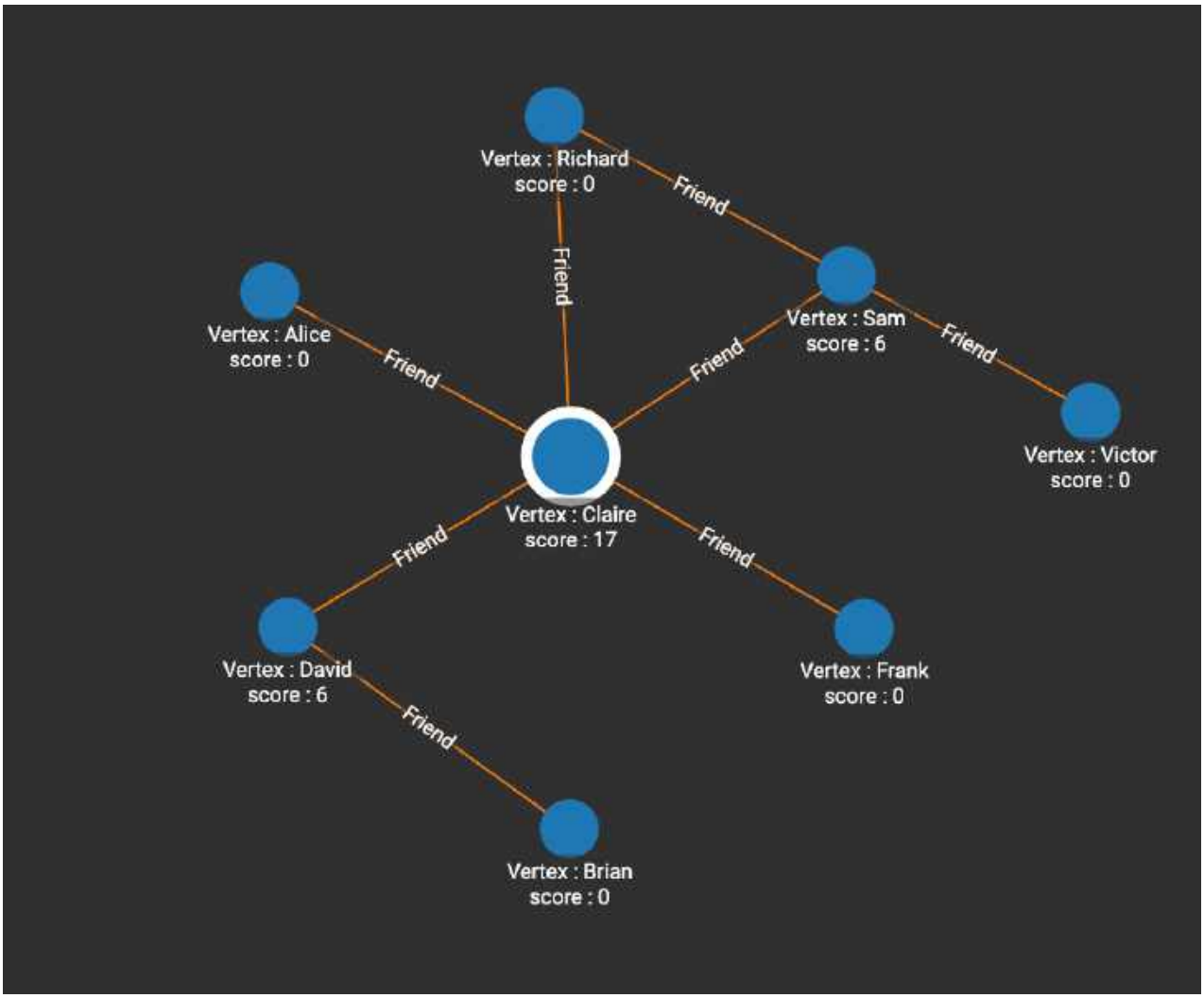
Parameters

Characteristic	Value
Result	Computes a Betweenness Centrality value (FLOAT type) for each vertex. The result is available in 3 forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Required Input Parameters	• **filepath** (for file output only): the path to the output file • **maxHops:** maximum number of iterations
Result Size	V = number of vertices
Computational 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 social 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

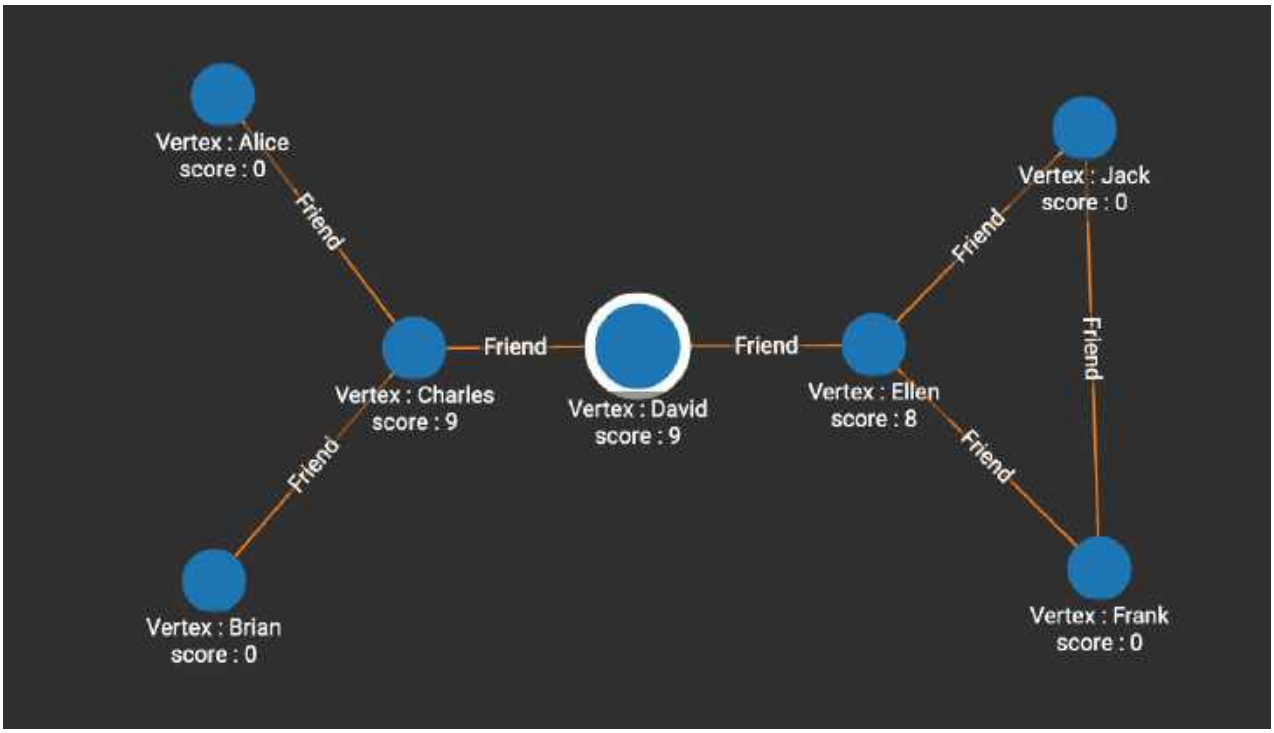
6. David also has a score of 6, since Brian has 6 paths to other people that pass through David.



betweenness_cent_attr(10) 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.



betweenness_cent_attr(10) on a social graph with undirected edges Friend

```
[
  {
    "@@BC": {
      "Alice": 0,
      "Frank": 0,
      "Charles": 9,
      "Ellen": 8,
      "Brian": 0,
      "David": 9,
      "Jack": 0
    }
  }
]
```

Community Algorithms

These algorithms evaluate how a group is clustered or partitioned, as well as its tendency to strengthen or break apart.

(Weakly) Connected Components

Description and Uses

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 task 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

conn_comp is deprecated. Use wcc_fast instead.

The implementation uses the following algorithm maintaining pointers from each vertex to a parent vertex, which may or may not be itself:

1. Initialize parent pointers by linking each vertex to the minimum ID from among itself and its neighbors. Each set of vertices that are connected via parent pointers can be described as a “family”.
2. Merge families of vertices if they are connected to other families via graph edges.
3. Decrease the depth, or “number of generations,” of each family by making each vertex’s grandparent its new parent.
4. Repeat 2 and 3 until each connected component is wholly represented by a single family with a single parent for all vertices in the family.

```
wcc_fast()
wcc_fast_file(STRING filepath)
wcc_fast_attr()
```

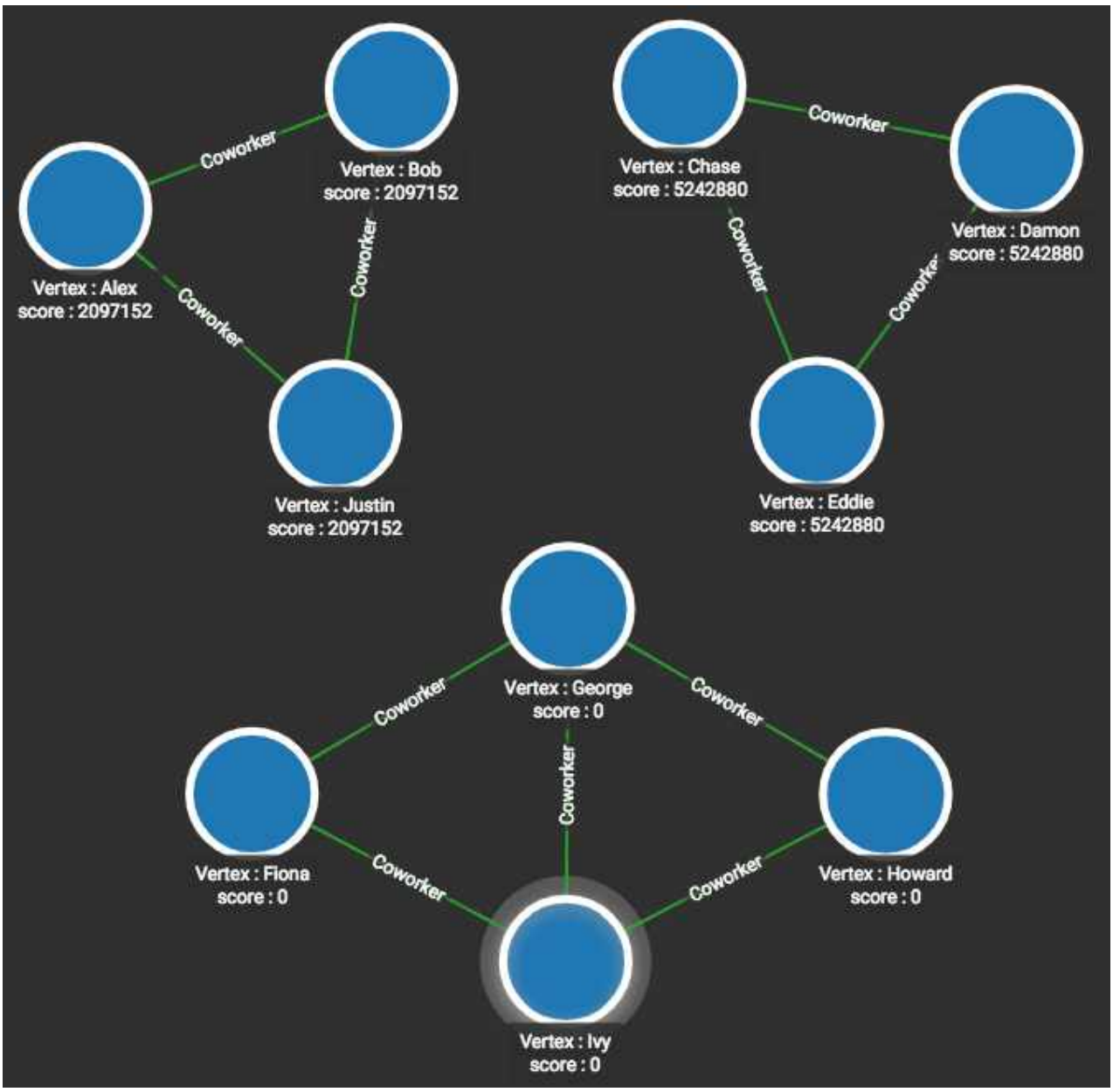
Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value. The result is available in three forms: • streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• **filepath** (for file output only): the path to the output file
Result Size	V = number of vertices
Computational Complexity	O(E * log V), E = number of edges, V = number of vertices d = max(diameter of components)
Graph Types	Undirected edges

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.



conn_comp(true, "json") on social10 graph with Coworker edges

Strongly Connected Components

Description and Uses

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 to 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

```
scc(INT iter = 500, INT iter_wcc = 5, INT top_k_dist)
scc_file(INT iter = 500, INT iter_wcc = 5, INT top_k_dist, FILE f)
scc_attr(INT iter = 500, INT iter_wcc = 5, INT top_k_dist)
```

Characteristic	Value
	Assigns a component id (INT) to each vertex, such that members of the same

Result	component have the same id value. 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	• **iter**: number of maximum iteration of the algorithm • **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) • **top_k_dist**: top k result in SCC distribution • **f** (for file output only): the path to the output file
Result Size	V = number of vertices
Computational Complexity	O(iter*d), d = max(diameter of components)
Graph Types	Directed edges with reverse direction edges as well

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.

Label Propagation

Description and Uses

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 they 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

```
label_prop(INT maxIter)
label_prop_file(INT maxIter, FILE filepath)
label_prop_attr(INT maxIter)
```

Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value. 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	• **maxIter**: the maximum number of update iterations. • **filepath** (for file output only): the path to the output file
Result Size	V = number of vertices
Computational Complexity	O(E*k), E = number of edges, k = number of iterations.
Graph Types	Undirected edges

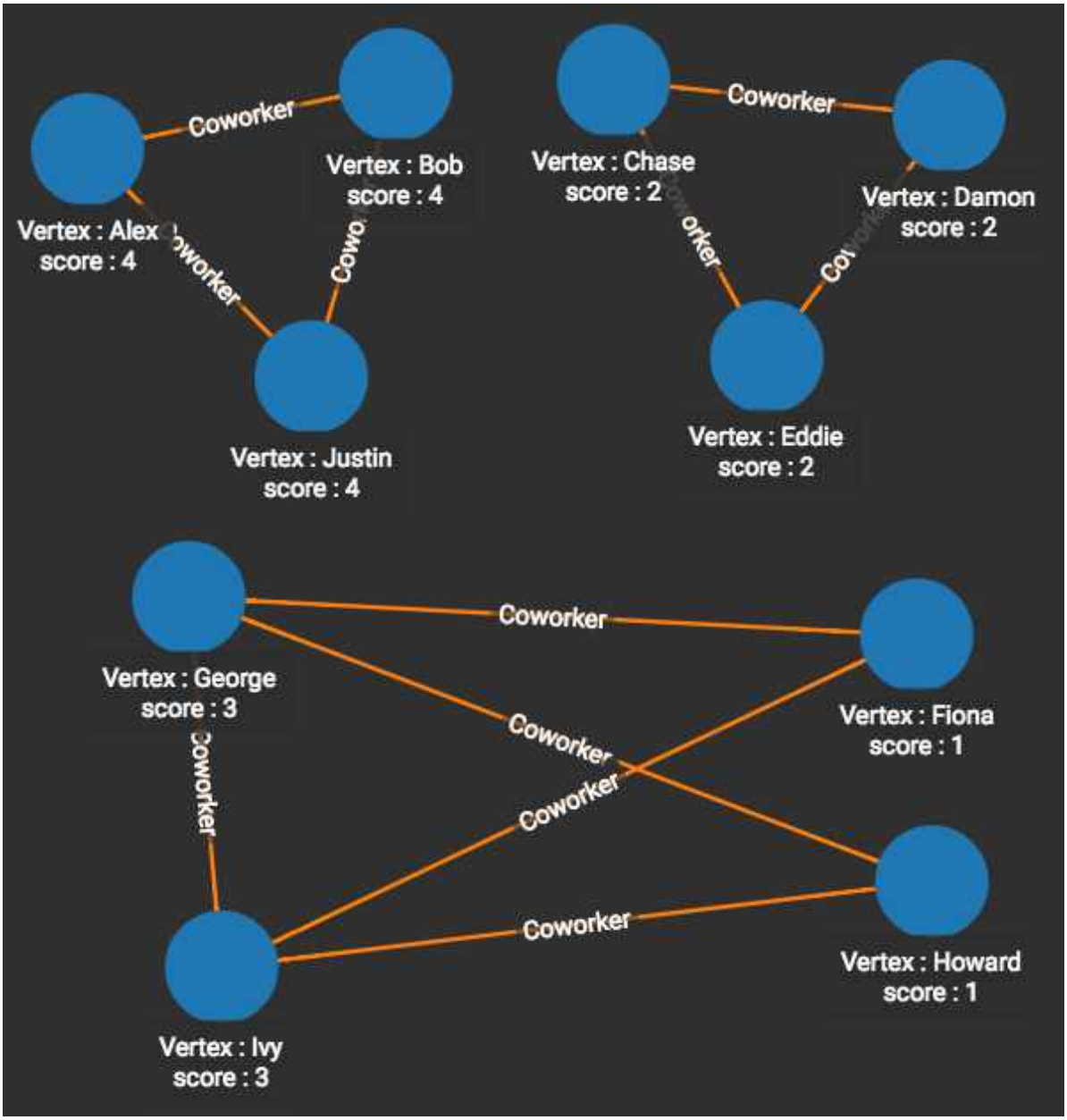
Example

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. See can see the symmetry:

- (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 maxIter to 10, but the algorithm reached steady state after 3 iterations.



label_prop(10) on social10 graph with Coworker edges

Louvain Modularity for Community Detection (Deprecated)

This algorithm is deprecated because a much higher performance algorithm (louvain_parallel) should be used instead of the original louvain algorithm.

Description and Uses

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.

Also, we use changes in modularity to guide optimization of the partitioning. That is, we begin with a candidate partitioning and measure its modularity. Then we make an incremental change and confirm that the modularity has improved.

The most 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.

Specifications

```
louvain()
louvain_file(FILE filepath)
louvain_attr()
```

Louvain Method with Parallelism and Refinement

Description and Uses

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 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 same labels into one vertex, and calculates the aggregated edge weights between new vertices. For the coarsened graph, the algorithm conducts 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

```
louvain_parallel(INT iter1 = 10, INT iter2 = 10, INT iter3 = 10, INT split
louvain_parallel_file(INT iter1 = 10, INT iter2 = 10, INT iter3 = 10, INT
louvain_parallel_attr(INT iter1 = 10, INT iter2 = 10, INT iter3 = 10, INT
```

Characteristic	Value
Result	Assigns a component id (INT) to each vertex, such that members of the same component have the same id value. 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	• **iter1**: the max number of iterations for the first phase. Default value is 10 • **iter2**: the max number of iterations for the second phase. Default value is 10 • **iter3**: the max number of iterations for the refinement phase. Default value is 10 • **split**: the number of splits in phase 1. Increase the number to save memory, at the expense of having longer running time. Default value is 10. • **outputLevel**: different detail level of community distribution shown in the result. Choice "0" only lists number of communities grouped by community

	size, while choice "1" also lists the members • **fComm**(for file output only): the path to the output file for community labels • **fDist**(for file output only): the path to the output file for community number distribution
Result Size	V = number of vertices
Computational Complexity	O(V^2*L), V = number of vertices, L = (iter1 * iter2 + iter3) = total number of iterations
Graph Types	Undirected, weighted edges A edge weight attribute is required.

Example

If we use louvain_parallel for social10 graph, it will give the same result as the result of as the connected components algorithm. The social26 graph is a connected graph which is quite dense. 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
      }
    ]
```

Triangle Counting

Description and Uses

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.

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.

```
tri_count()
tri_count_file(FILE filepath)
tri_count_attr()
```

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.

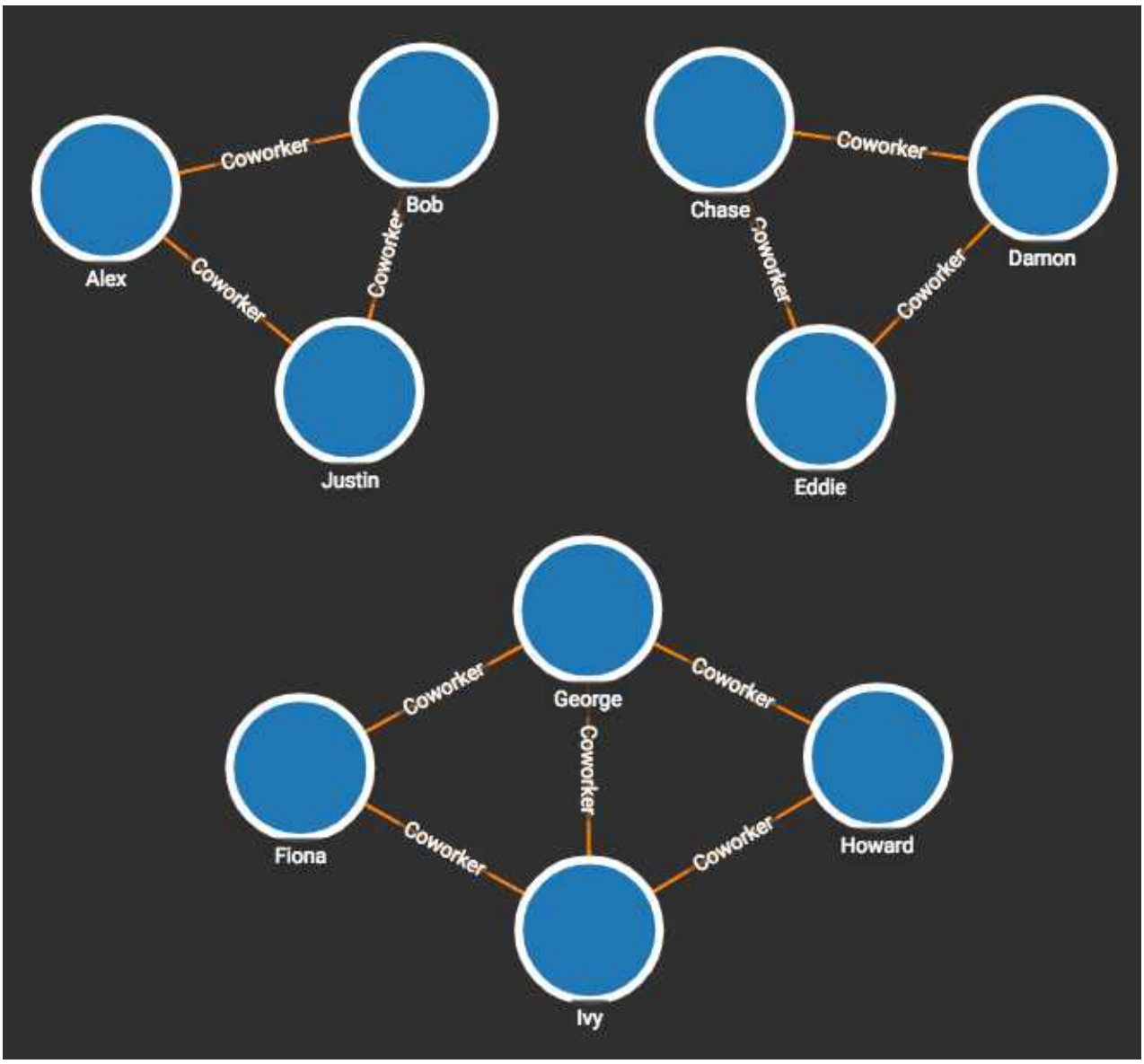
tri_count_fast() is a smarter algorithm which does two passes over the edges. In the first pass we mark which of the two endpoint vertices has fewer neighbors. In the second pass, we count the overlap only between marked vertices. The result is that we eliminate 1/3 of the neighborhood matching, the slowest 1/3, but at the cost of some additional memory.

```
tri_count_fast()
tri_count_fast_file(FILE filepath)
tri_count_fast_attr()
```

Characteristic	Value
Result	Returns the number of triangles in the graph.
Input Parameters	None
Result Size	1 integer
Computational 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}
  ]
}
```


Similarity Algorithms

There are many ways to measure the similarity between two vertices in a graph, but all of them compare either (1) the features of the vertices themselves, (2) the relationships of each of the two vertices, or (3) both. We use a graph called movie to demonstrate our similarity algorithms.

Cosine Similarity of Neighborhoods, Single Source

Description and Uses

To compare two vertices by cosine similarity, first selected properties of each vertex are represented as a vector. For example, a property vector for a Person vertex could have the elements (age, height, weight). Then the cosine function is applied to the two vectors.

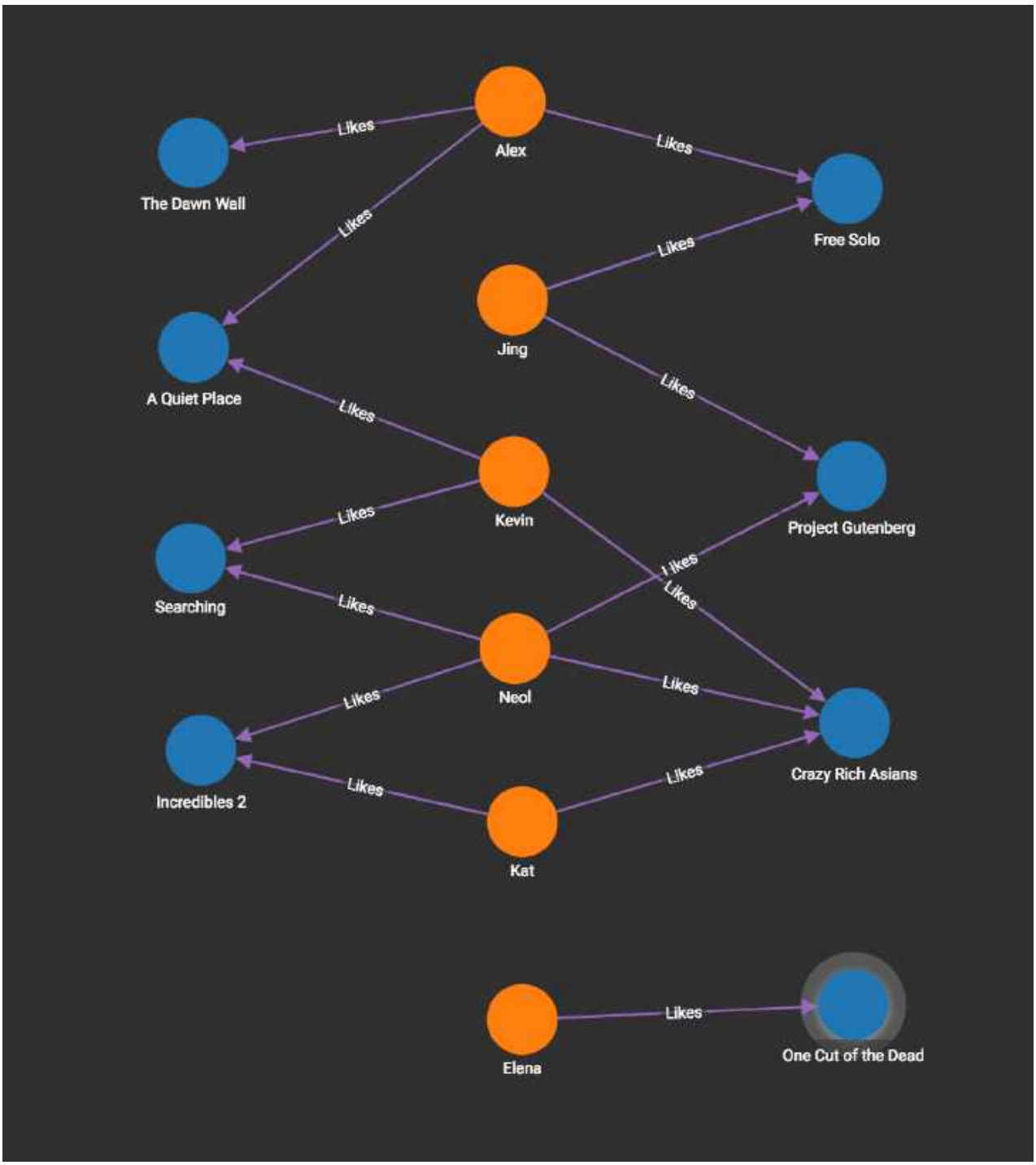
The cosine similarity of two vectors A and B is defined as follows:

$$cos(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_i A_i B_i}{\sqrt{\sum_i A_i^2}\sqrt{\sum_i B_i^2}}$$

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 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 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 give a rating of 10 to the movie "Free Solo".



movie graph

Specifications

```
cosine_nbor_ss(VERTEX source, INT topK)
cosine_nbor_ss_file(VERTEX source, INT topK, FILE filepath)
cosine_nbor_ss_attr(VERTEX source)
```

Characteristic	Value

Result	the topK vertices 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	• **source**: the source vertex • **topK**: the number of vertices • **filepath** (for file output only): the path to the output file
Result Size	topK
Computational 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 chose 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..

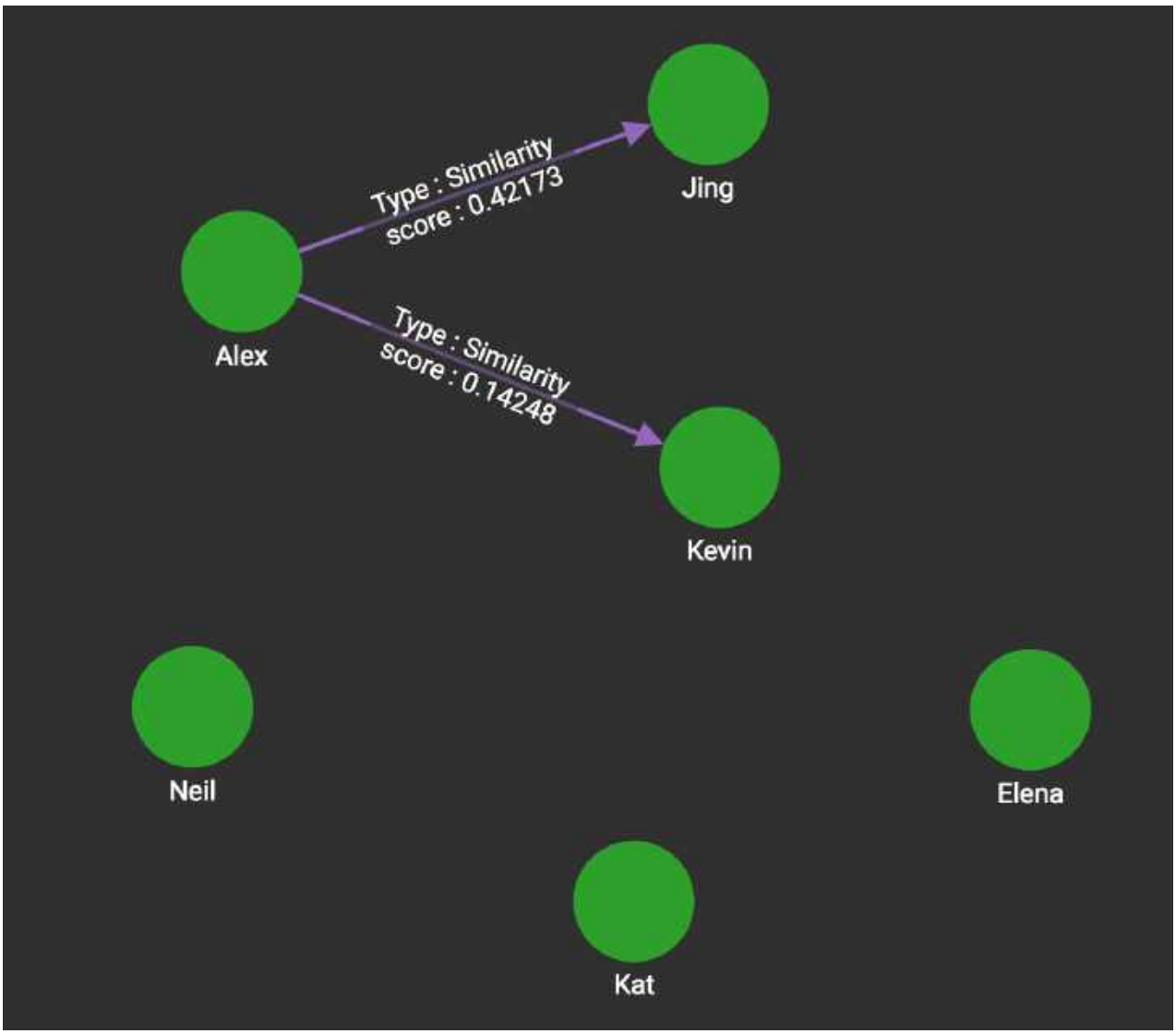
In the previous example, if the input is Alex, and topK is set to 5, then we calculate the cosine similarity between him and two other persons, Jing and Kevin. The JSON output shows the top k similar vertices and their similarity score in descending order. The output limit is 5 persons, but we have only 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:



Cosine Similarity of Neighborhoods, All Pairs

Description and Uses

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

```
cosine_nbor_ap(INT topK)
cosine_nbor_ap_file(INT topK, FILE filepath)
cosine_nbor_ap_attr(INT topK)
```

Characteristic	Value
Result	the topK 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	• **topK**: the number of vertex pairs • **filepath** (for file output only): the path to the output file
Result Size	topK
Computational Complexity	O(E^2 / V), V = number of vertices, E = number of edges
Graph Types	Undirected or directed edges, weighted edges

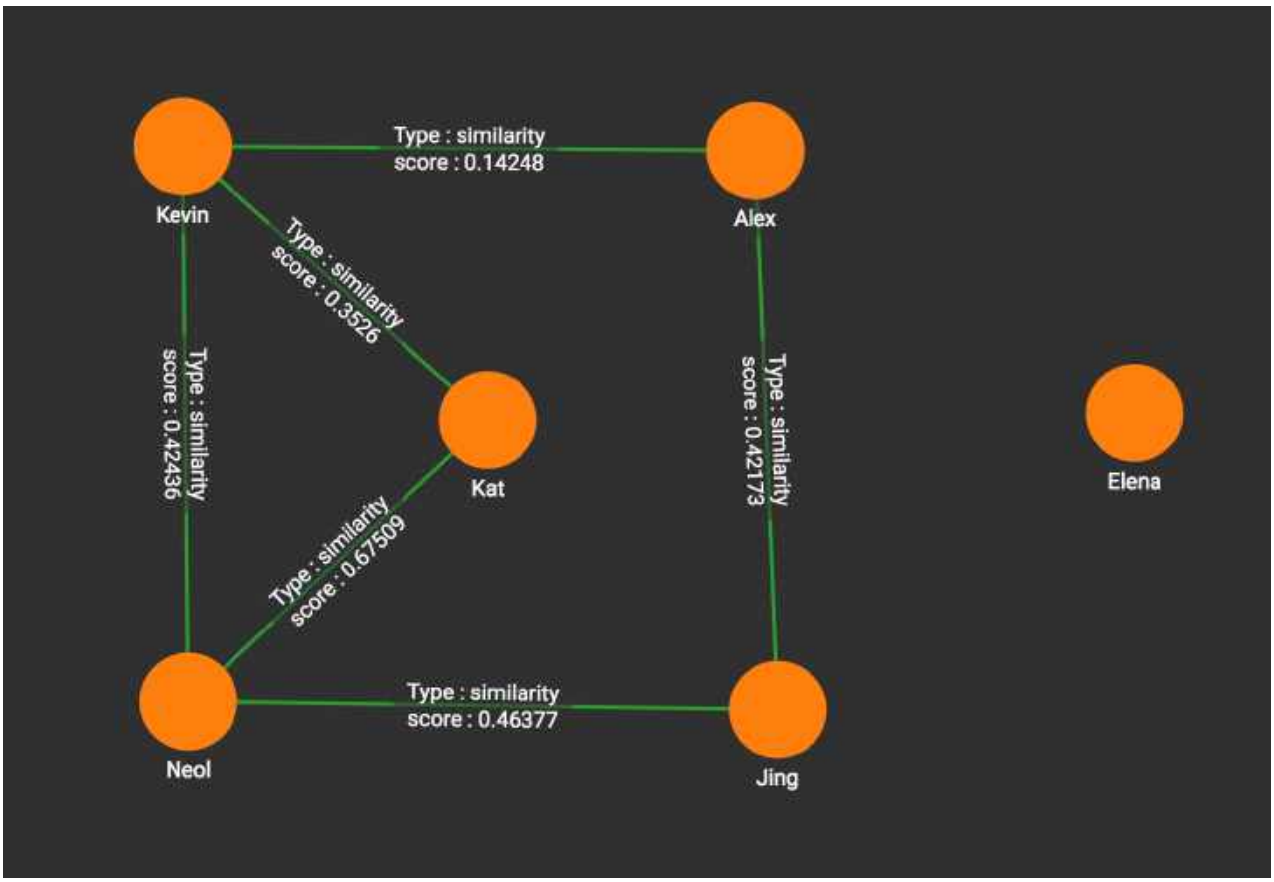
Example

Using the movie graph, calculate the cosine similarity between all pairs and show the top 5 pairs: cosine_nbor_ap(5). 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:



Jaccard Similarity of Neighborhoods, Single Source

Description and Uses

The Jaccard index measures the relative overlap between two sets. To compare two vertices by Jaccard similarity, first select a set of values for each vertex. For example, a set of values for a Person could be the cities the Person has lived in. Then the Jaccard index is computed for the two vectors.

The Jaccard index of two sets A and B is defined as follows:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The value ranges from 0 to 1. If A and B are identical, then Jaccard(A, B) = 1. If both A and B are empty, we define the value to be 0.

Specifications

In the current

```
jaccard_nbor_ss(VERTEX source, INT topK)
jaccard_nbor_ss_file(VERTEX source, INT topK, FILE filepath)
jaccard_nbor_ss_attr(VERTEX source, INT topK)
```

Characteristic	Value
Result	the topK vertices 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	• **source**: the source vertex • **topK**: the number of vertices • **filepath** (for file output only): the path to the output file
Result Size	topK
Computational 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 chose 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

Description and Uses

This algorithm computes the same similarity scores as the Jaccard similarity of neighborhoods, single source algorithm (jaccard_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 algorithm may not be a practical choice

Specifications

```
jaccard_nbor_ap(INT topK)
jaccard_nbor_ap_file(INT topK, FILE filepath)
jaccard_nbor_ap_attr(INT topK)
```

Characteristic	Value
Result	the topK 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	• **topK**: the number of vertices • **filepath** (for file output only): the path to the output file
Result Size	topK
Computational 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
      }
    ]
  }
]
```


Classification Algorithms

Classification algorithms, or classifiers, are one of the simplest forms of machine learning. They seek to predict the classification of a given entity, based on the evidence of previously classified entities. Classification is closely related to similarity and clustering; all of them deal with finding and using the commonalities among entities.

k-Nearest Neighbors, Cosine Neighbor Similarity, single vertex

Description and Uses

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 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 which 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 a similarity score, in which case "nearest" means "most similar". In our algorithm, the distance is the reciprocal of cosine neighborhood 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

```
knn_cosine_ss(VERTEX source, INT topK)
knn_cosine_ss_file(VERTEX source, INT topK, FILE f)
knn_cosine_ss_attr(VERTEX source, INT topK)
```

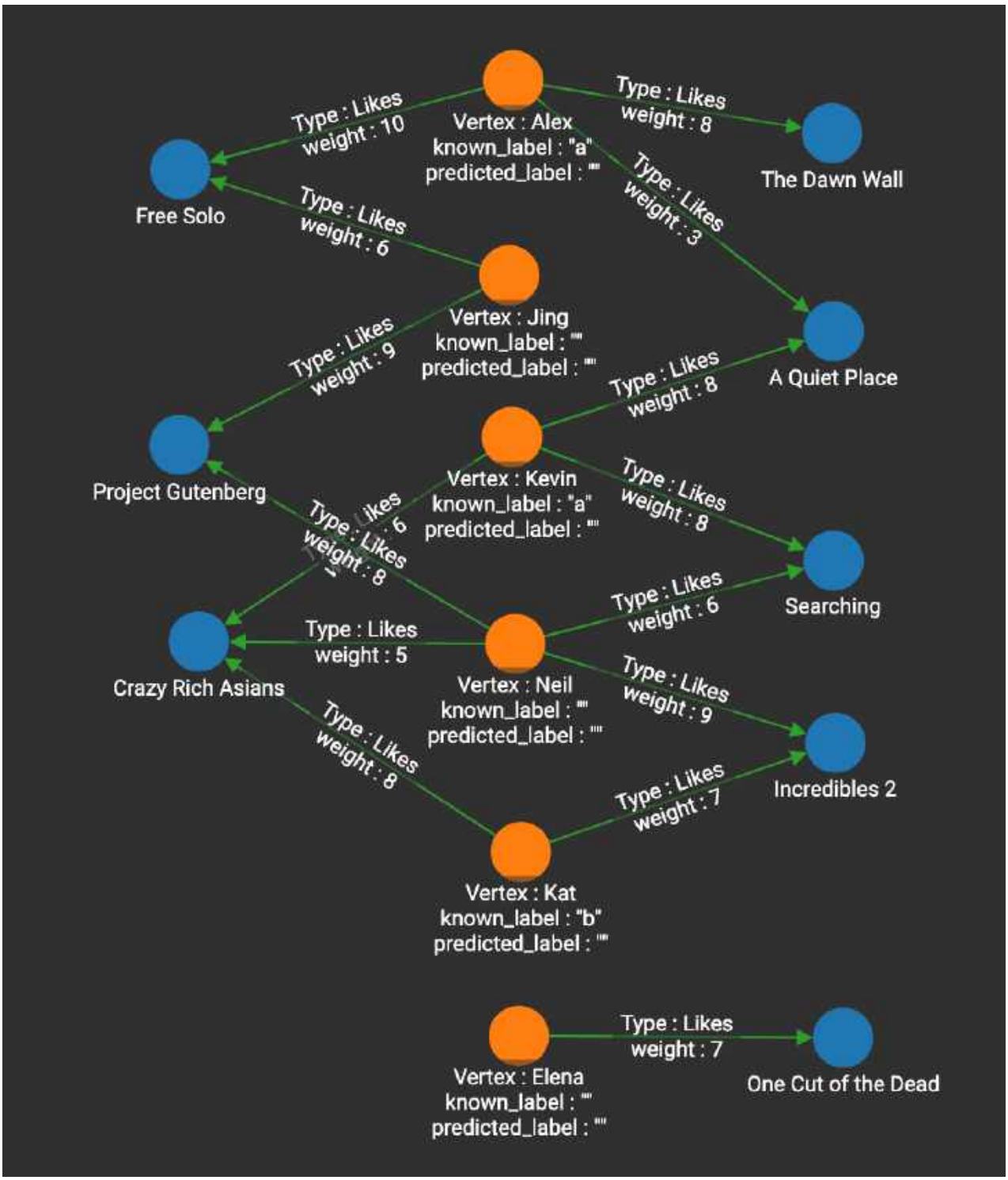
Characteristic	Value
	The predicted label for the source vertex. The result is available in three forms:

Result	• streamed out in JSON format • written to a file in tabular format, or • stored as a vertex attribute value.
Input Parameters	• **source**: the vertex for which you want to predict the label • **topK**: number of nearest neighbors to consider • **filepath** (for file output only): the path to the output file
Result Size	V = number of vertices
Computational 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 chose 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" is 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, Cosine Neighbor Similarity, All Vertices Batch

Description and Uses

This algorithm is a batch version of the k-Nearest Neighbors, Cosine Neighbor Similarity, single vertex ↗. It make 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

```
knn_cosine_all(INT topK)
knn_cosine_all_file(INT topK, FILE f)
knn_cosine_all_attr(INT topK)
```

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	• **topK**: number of nearest neighbors to consider • **filepath** (for file output only): the path to the output file
Result Size	V = number of vertices

Computational Complexity	O(E^2 / V), V = number of vertices, E = number of 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, Cosine Neighbor Similarity, cross validation

Description and Uses

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 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

```
knn_cosine_cv(INT min_k, INT max_k)
```

Characteristic	Value
Result	A list of prediction accuracy for every k value in the give range, and the value of k with highest predicting accuracy in the given range. The result is available in JSON format
Input Parameters	• min_k: lower bound of k (included) • max_k: upper bound of k (included)
Result Size	max_k-min_k+1
Computational 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
  }
]
```

Release Notes, Change Log

Problem with Microsoft Internet Explorer and Edge browsers.

The ID and Edge browsers are not displaying the tables on several of our our documentation web pages.

The Google Chrome, Mozilla Firefox, and Apple Safari browsers have been validated.

For documentation of TigerGraph versions prior to 2.2, please see doc-archive.tigergraph.com ↗.

- Release Notes - TigerGraph Server 2.5
- Release Notes - TigerGraph 2.4 ↗
- Release Notes - TigerGraph 2.3 ↗
- Release Notes - TigerGraph 2.2 ↗
- Change Log

Release Notes - TigerGraph Server 2.5

Release Date: September 20, 2019

Release Notes for Previous Versions:

- 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.

New Features

GSQL Enhancements

- ✅ MinAccum and MaxAccum now support STRING and TUPLE data types.
- ✅ Added more STRING manipulation built-in token functions.
  See Token Functions for Attribute Expressions ↗ in the GSQL Language Reference, Part 1
- ✅ Add DELETE TUPLE, to delete a user-defined tuple typ
- ✅ MultiGraph independence: Each graph can name vertex types and edge types, independently of other graphs.

GraphStudio Enhancements

See the relevant sections of the GraphStudio UI Guide

- **Design Schema**:

    - ✅ Users can upload their own vertex icons.
    - ✅ More built-in vertex icons.
- **Map Data to Graph**:
    - ✅ Allow use of user-defined token functions.
- **Explore Graph**:
    - ✅ Persist the graph exploration result on the "Explore Graph" page.
    - ✅ Can save/load an exploration result.
- **Write Query**:
    - ✅ Persist query result on “Write Query” page.
    - ✅ Can save/load a query result.

REST Enhancements

See RESTPP API User Guide

RESTPP Server

- ✅ Add header option to limit the size of the output response.
- ✅ Add built-in endpoints to REFRESH or DROP tokens
- ✅ Add built-in endpoint to get query ID
- ✅ Add built-in endpoint to abort query using queryID
- ✅ Replace 6 shortest path query endpoints with 2 parameterized ones

GSQL Server

- ✅ Add GSQL server endpoint to get the graph schema.

License Key

- ℹ️ Rename the **System-Specific License Activation** page to be **Advanced License Issues**

- ✅ Add **Limited Capacity mode**: If the platform reaches a graph size or memory limit established by the license key or a configuration parameter, then the system will refuse to add new data. The graph may still be queried, and data may be deleted. See the **Usage Limits Controlled by License Key** section in the Advanced License Issues page.

System Integration

JDBC Driver

- ✅ Added support for GSQL interpreted mode, Spark, and Python.

Modified Features

- ℹ️ **GSQL**: To select pattern matching support in a query, the punctuation around the version string has been dropped. The syntax is now
  `CREATE QUERY ... SYNTAX v2`
  instead of
  `CREATE QUERY ... SYNTAX("v2")`

V3.0 Removal of Previously Deprecated Features

TigerGraph 2.x contains some features which are 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 will streamline 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 count(*) FROM ... // count is always current

Change Log

This page will document all the changes to TigerGraph product including New Features and Bug Fixes.

> Distributed Graph support, MultiGraph, and certain enterprise features are available in the Enterprise Edition only. They do not pertain to the Developer Edition.

Documentation Changes

- 07/02/20 RESTPP Built-in Endpoints - Corrected the URL for the GET Schema REST endpoint
- 07/02/20 Vertex Functions - Clarified the behavior of outdegree()

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

Fixes:

- 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 needed.**

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

Fixes

- 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

Introduction and Overview

GET STARTED with TigerGraph

Welcome to the TigerGraph™ Platform - the first real-time, Native Parallel Graph data analytics platform. We have a quick **Overview** and **Glossary** to help you understand the TigerGraph environment and its GraphStudio UI.

Quick Start Guide for New Users

Installation Checklist

1. **CHECK** Hardware and Software Requirements
2. **DOWNLOAD** the TigerGraph platform: www.tigergraph.com/download ↗
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 enterprise edition
   sudo ./install.sh -s

   # to install developer edition
   sudo ./install.sh
   ```

   b. For additional options, see TigerGraph Platform Installation Guide
4. Enterprise Edition: Consider other System Administration issues, such as Security.
   Also, if you have a GraphStudio license, activate GraphStudio.

You're ready to go!

- **BUILD** your first graph application and start to learn the GSQL language with GSQL 101 .
- **GET ANSWERS** to basic questions from the Knowledge Base and FAQs .

- **DISCUSS** and share with your fellow TigerGraph users: https://groups.google.com/a/opengsql.org/forum/#!forum/gsql-users ↗
- **LEARN** more GSQL through additional use cases: GSQL Demo Examples .
  - Demo scripts are located in `<your_install_folder>/document/examples`
  - Sample use cases, used in our Test Drive demo systems, are located in `<your_install_folder>/document/examples/test_drive`
- Do you want an architectural overview? TigerGraph Platform Overview
- The new Admin Portal dashboard lets you see how your system is running.
- **The full documentation is at docs.tigergraph.com .**

GraphStudio

Our GraphStudio UI lets beginners and pros alike set up and perform analytics with a TigerGraph database, all from a graphical user interface. The only code you'll need to write is for queries themselves; everything else is managed graphically.

- **Schema Designer** - Describe your graph data model.
- **Loading Builder** - Select your input files, then drag-and-drop to link input data to vertex and edge fields.
- **Graph Explorer** - display and explore your graph data, in an intuitive and visual way.
- **Query Editor** - view, edit, and run queries. Display the results graphically.

Note: GraphStudio is included in your Developer Edition but is licensed separately.

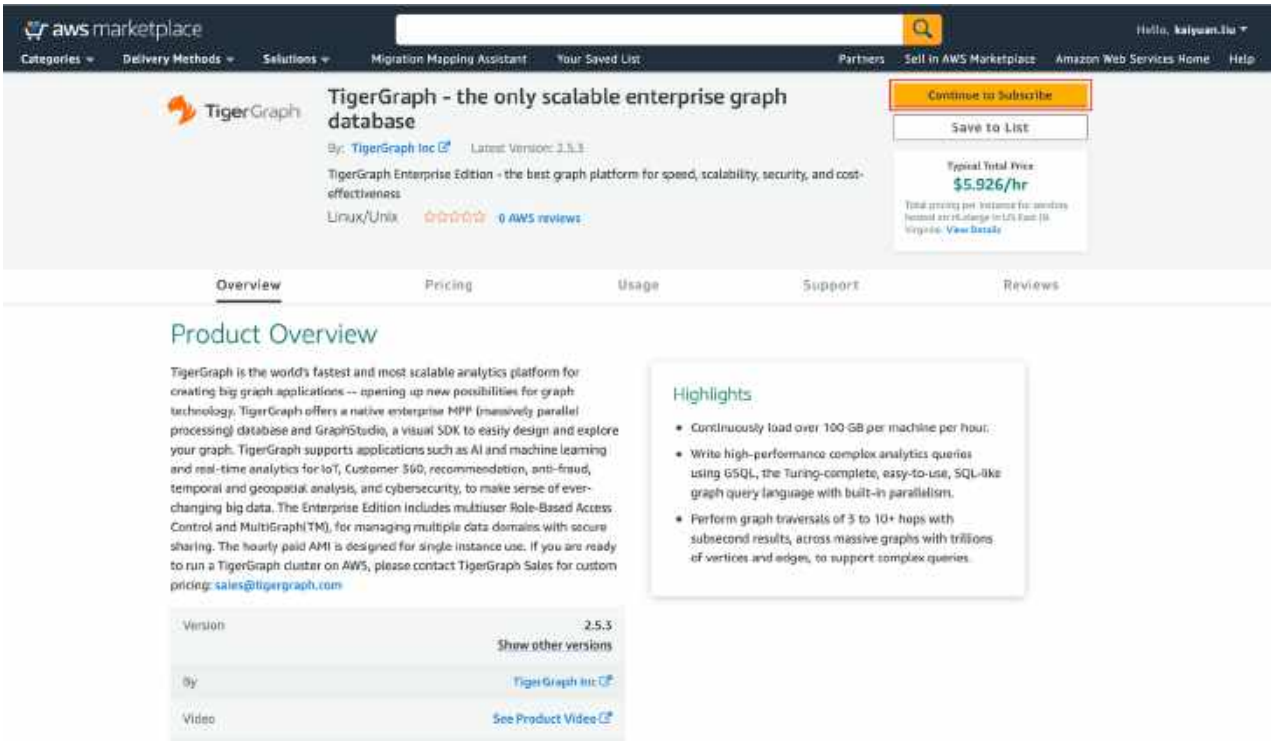
If you have any questions or suggestions, please contact us at **tigergraph.freshdesk.com** ↗

GET STARTED on AWS

Deploying Your Instance

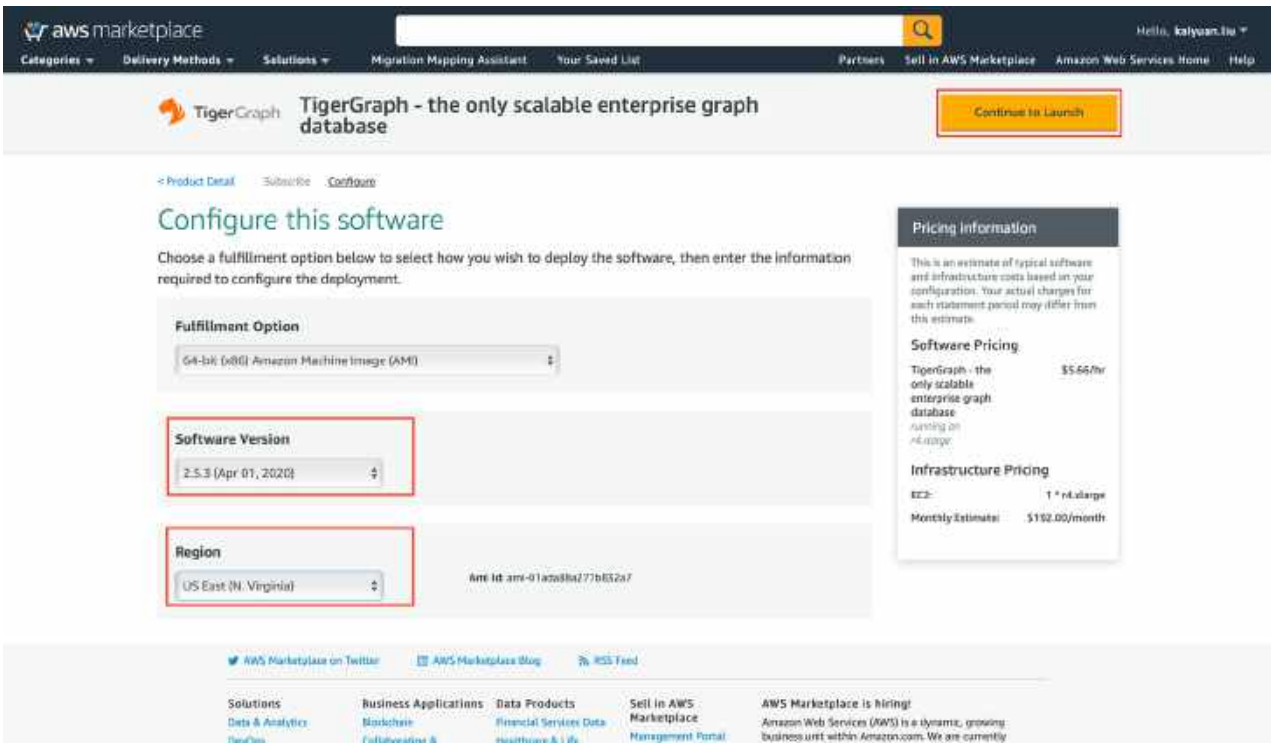
1. Go to https://aws.amazon.com/marketplace/ ↗ and search TigerGraph

2. Click on "Continue to Subscribe"



3. Click on "Continue to Configuration"

4. Select the Software Version and Region. We recommend selecting the latest version for the most up to date features. When complete, click on "Continue to Launch"



5. Select the instance type, security group settings and other settings. The default settings should work for you, but feel free to modify them. Click "Launch" when

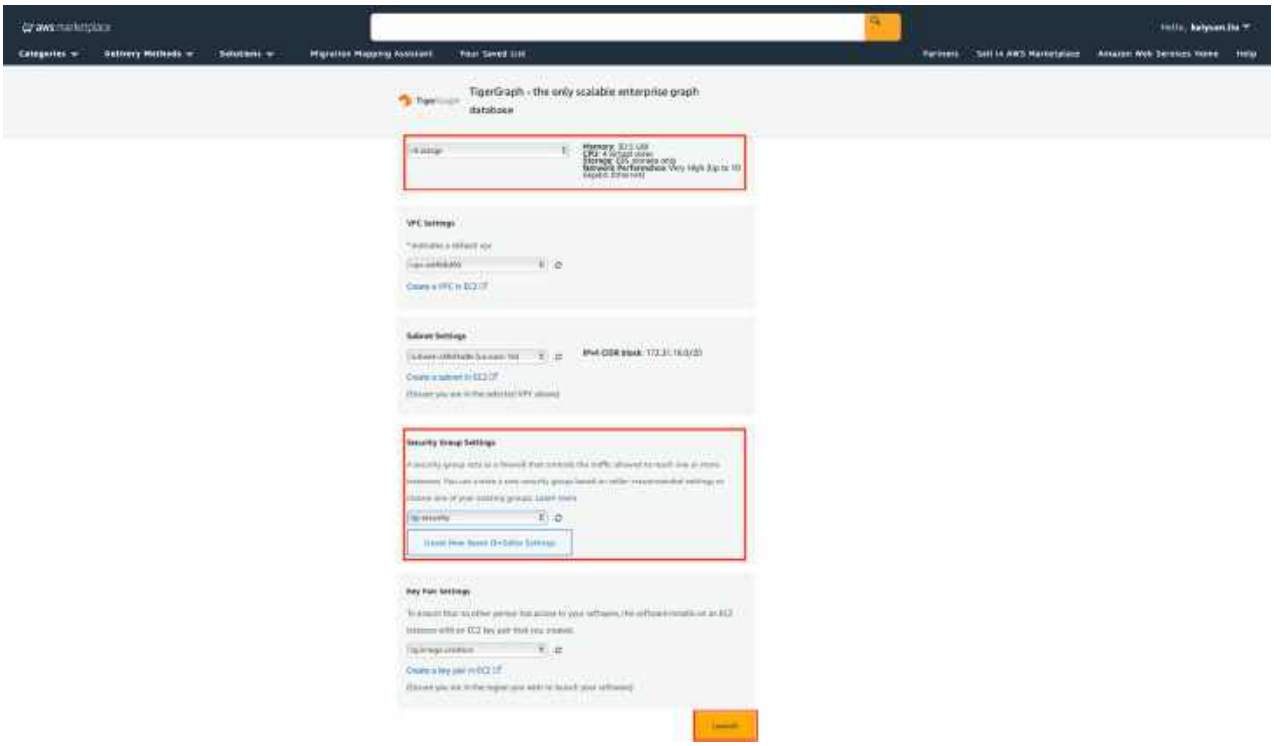
finished.

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

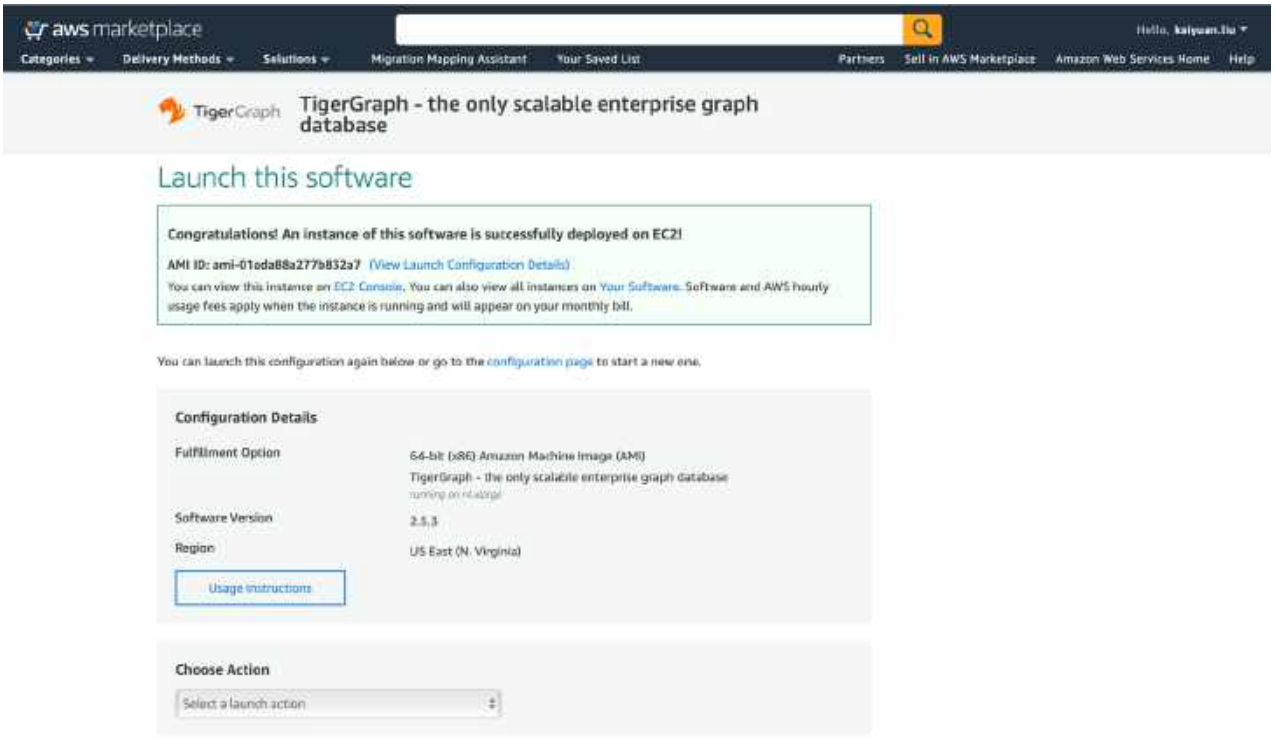
The security group must allow inbound TCP traffic to port 14240 if you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the GraphStudio UI Guide.

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.



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



Starting TigerGraph on Your Instance

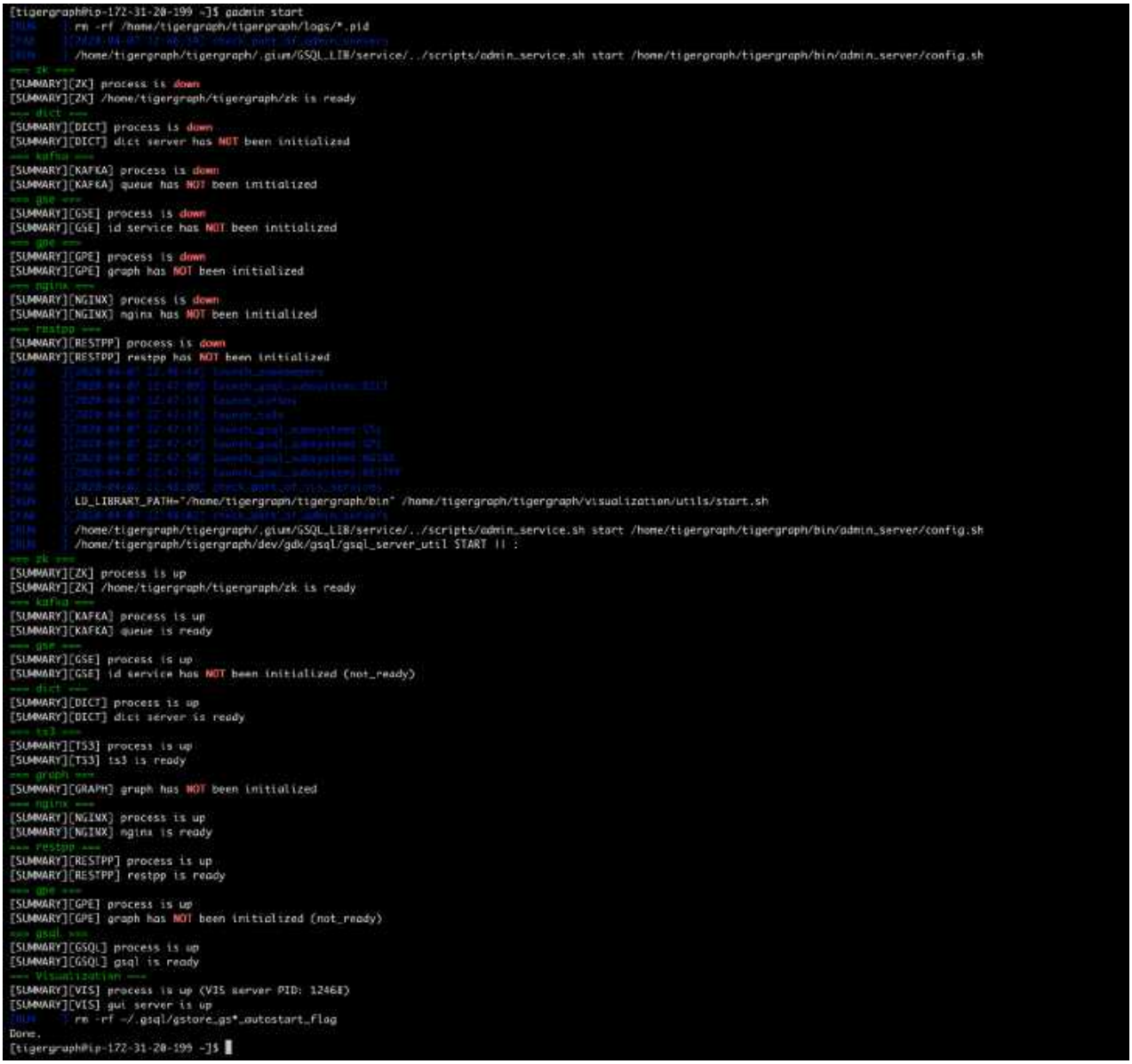
1. Log on to the instance and switch to the tigergraph user using the following command:

```
sudo su - tigergraph
```

```
[ec2-user@ip-172-31-20-199 ~]$ sudo su - tigergraph
Last login: Tue Apr  7 22:44:37 UTC 2020 on pts/0
[tigergraph@ip-172-31-20-199 ~]$
```

2. Run the following command to start TigerGraph

```
gadmin start
```



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

Next Steps

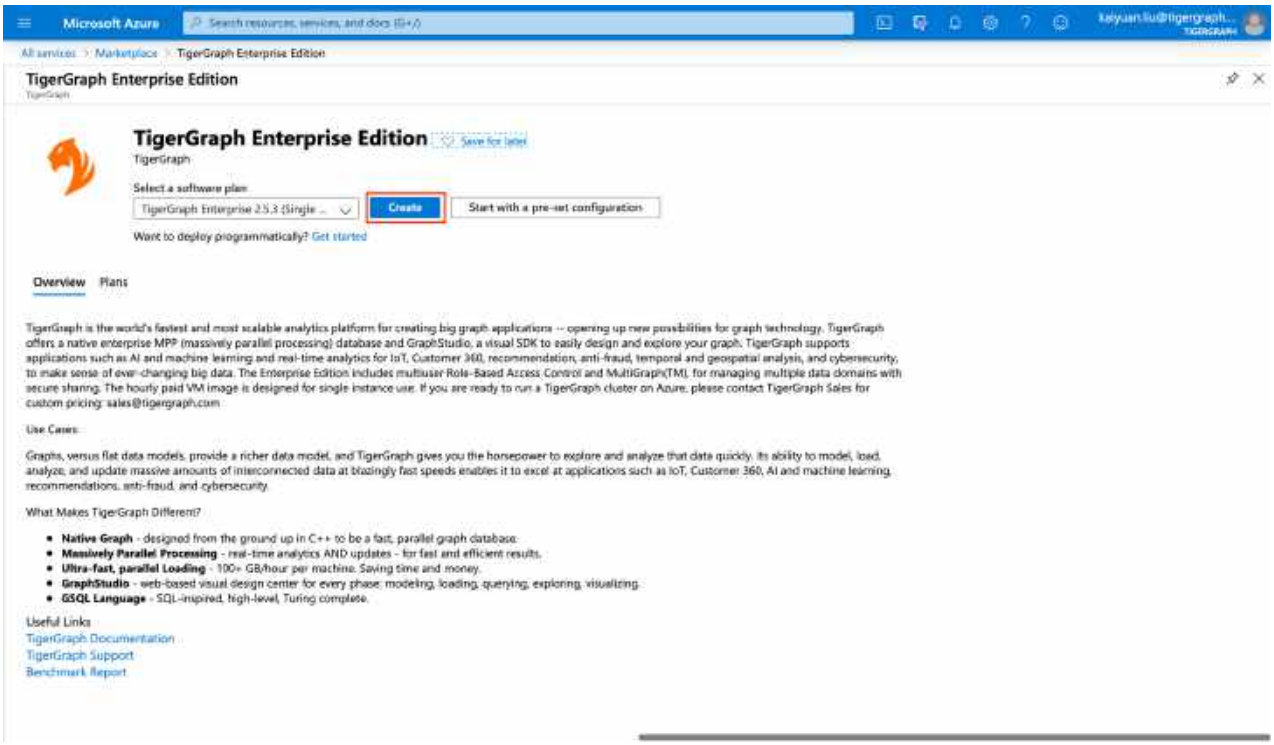
If you're new to TigerGraph, see the section "You're Ready to go ↗" for suggested next steps.

GET STARTED on Microsoft Azure

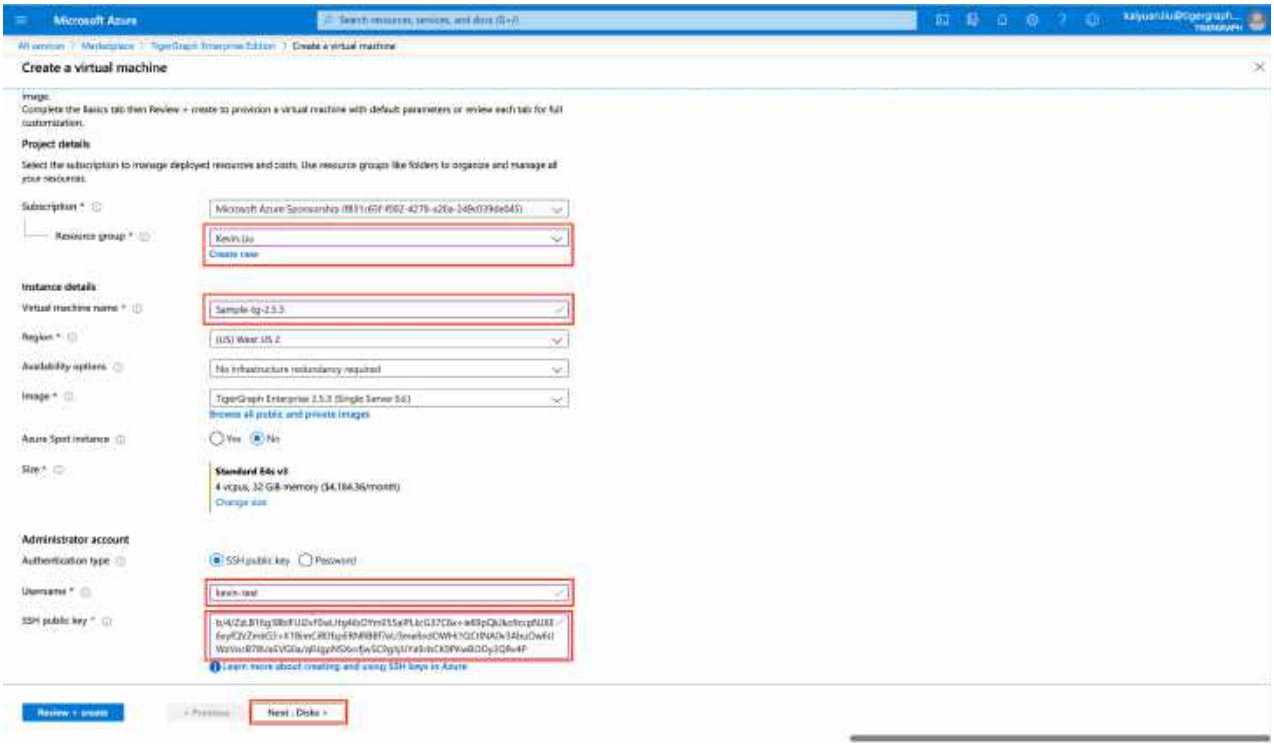
Deploying Your Instance

1. Go to https://portal.azure.com/#blade/Microsoft_Azure_Marketplace/MarketplaceOffersBlade/selectedMenuItemId/home ↗ and search "TigerGraph Enterprise Edition"

2. Click on "Create"



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



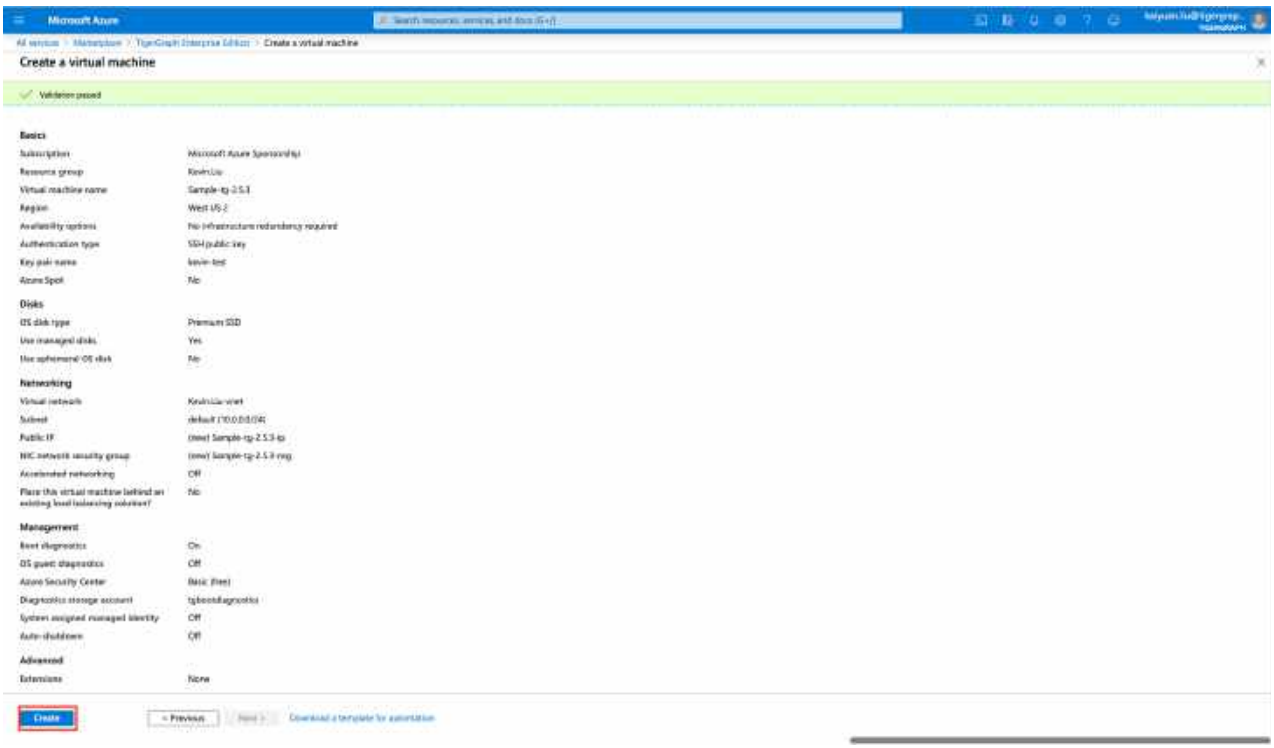
4. Keep the default values for all other settings and click "Next" until you see the "Review + Create" page below. Check all your settings and click "Create" when your satisfied.

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

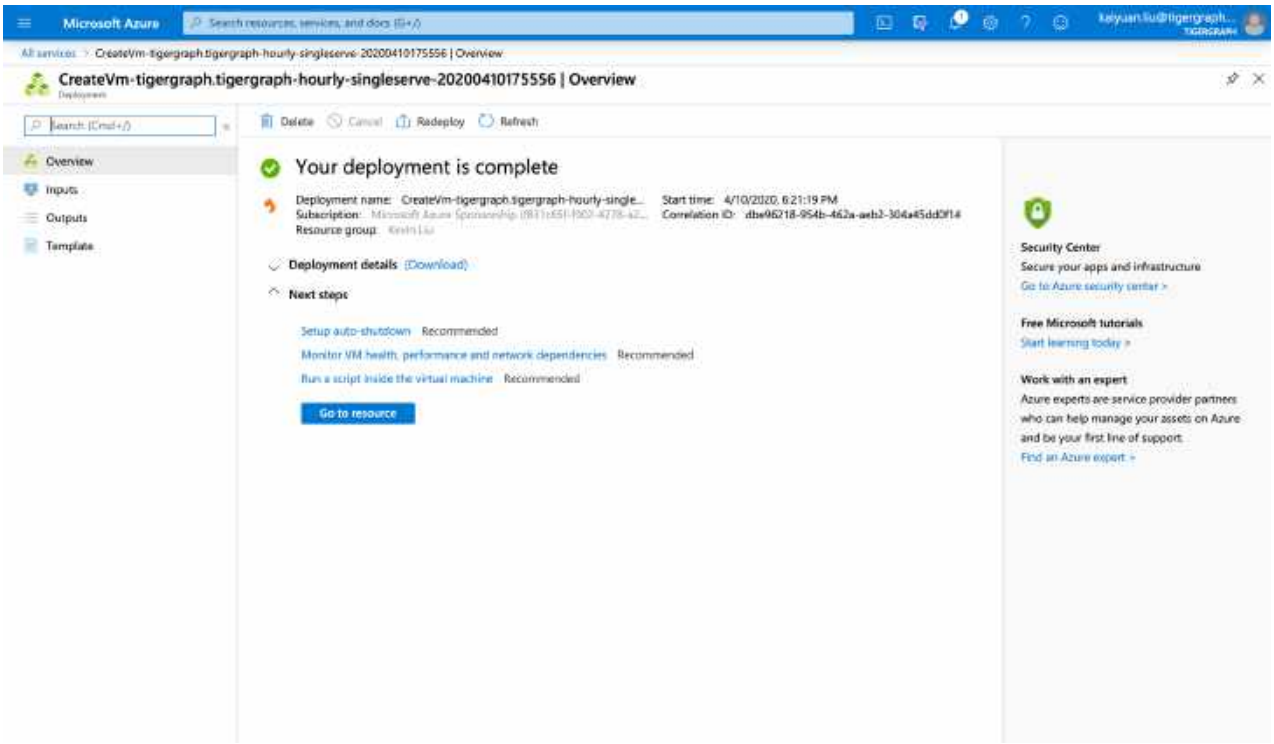
The "NIC network security group" must allow inbound TCP traffic to port 14240 if you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the GraphStudio UI Guide.

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.



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



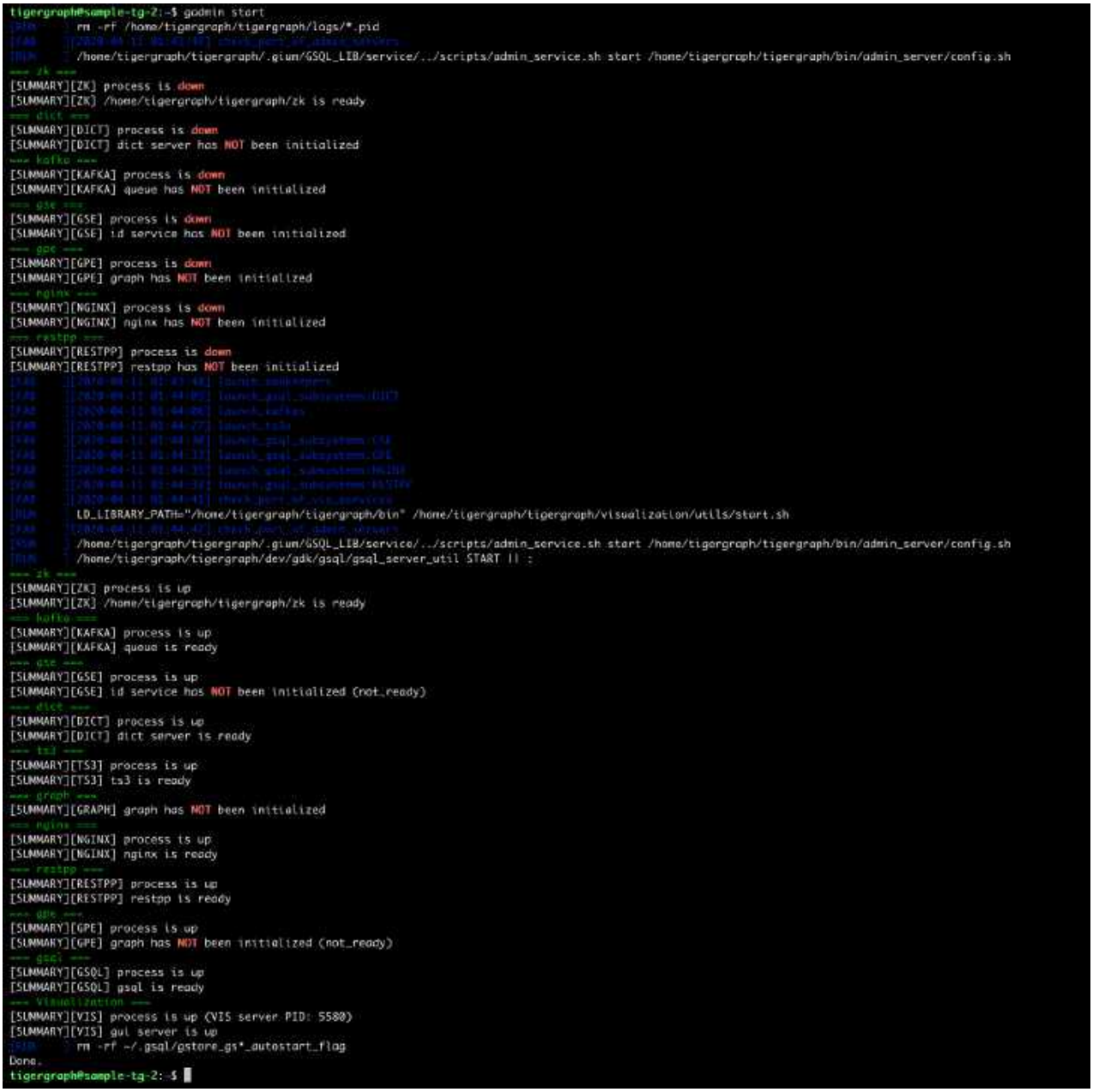
Starting TigerGraph on Your Instance

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

```
sudo su - tigergraph
```

2. Run the following command to start TigerGraph

```
gadmin start
```



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

Next Steps

If you're new to TigerGraph, see the section "You're Ready to go ↗" for suggested next steps.

GET STARTED on Google Cloud

Welcome to the getting started on Google Cloud tutorial. In this tutorial we will show you how to start TigerGraph from an image on Google Cloud. Please select your editon from below:

GET STARTED With TigerGraph Developer Edition on Google Cloud >

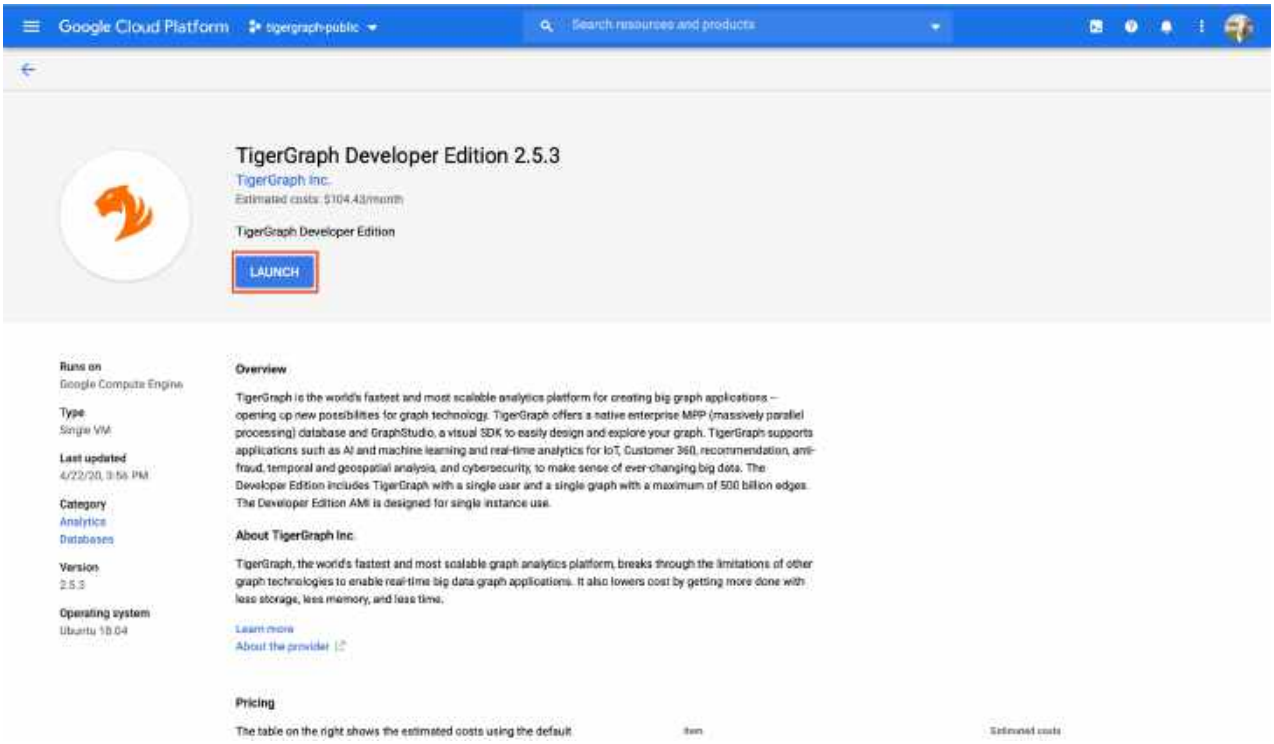
GET STARTED With TigerGraph Enterprise Edition on Google Cloud >

GET STARTED With TigerGraph Developer Edition on Google Cloud

Deploying Your Instance

1. Go to https://console.cloud.google.com/marketplace ↗ and search "Tigergraph Developer Edition". Choosing the latest edition (2.5.3) is recommended.

2. Click on "Launch"



3. The default setting should work for you, but feel free to modify them. When complete, click on "Deploy"

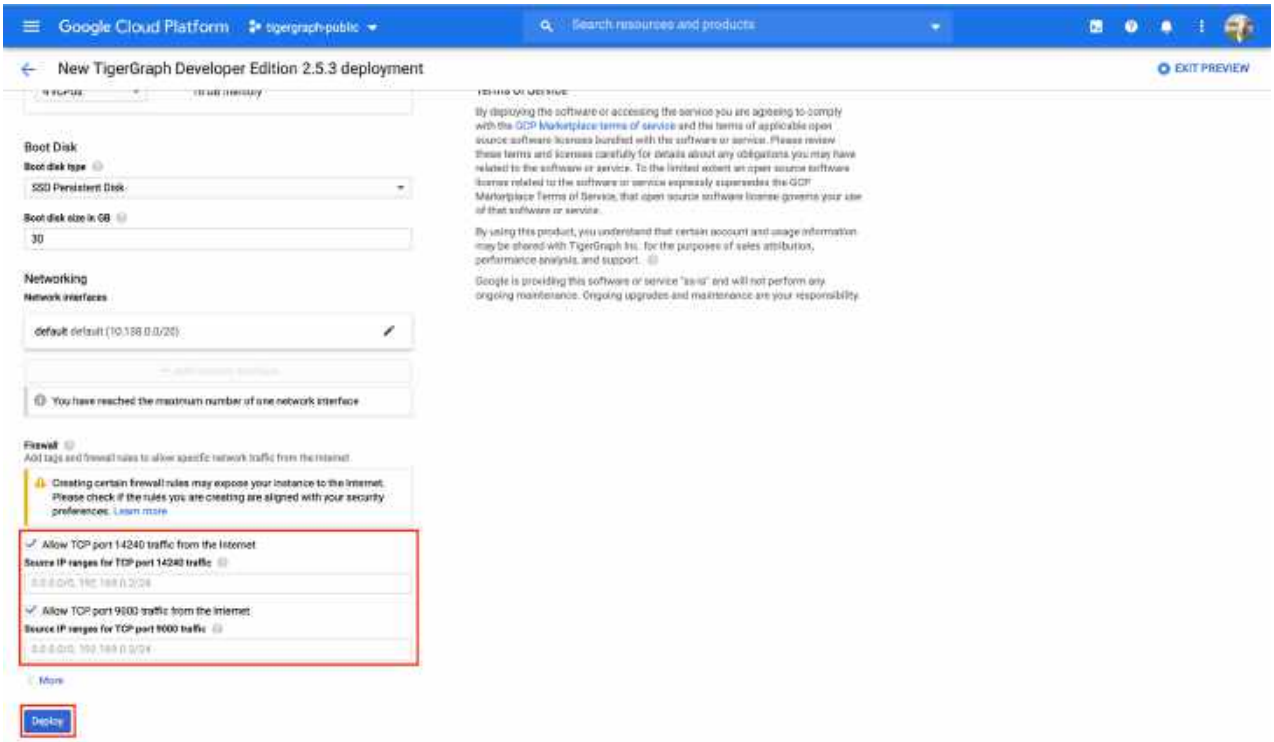
Notes:
The instance type needs to have at least 4 CPU and 16GB RAM for TigerGraph to work properly.

The "Allow TCP port 14240 traffic from the Internet" checkbox must be checked if

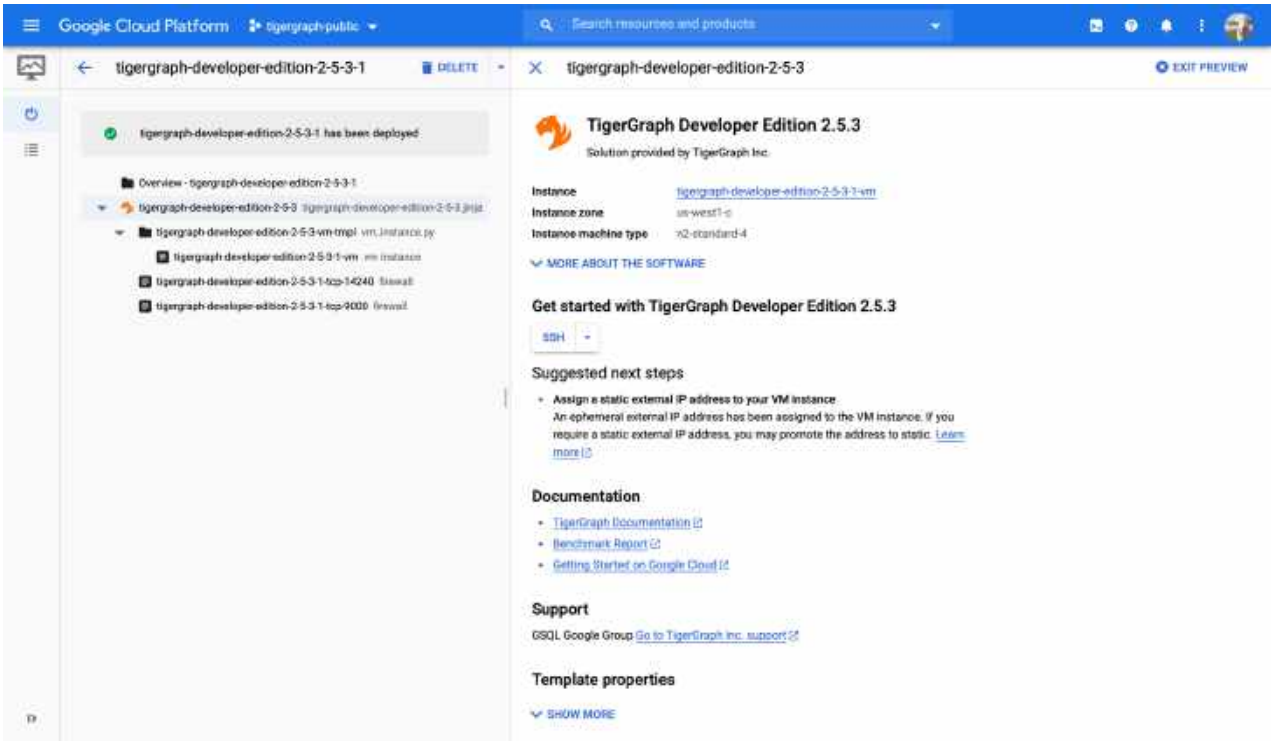
you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the GraphStudio UI Guide.

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.



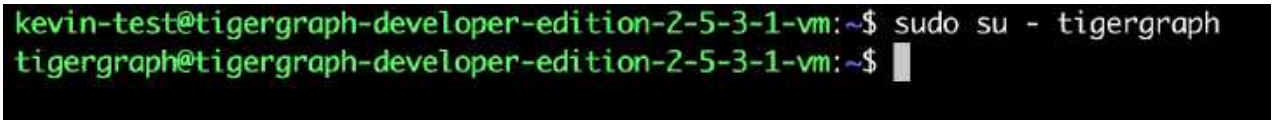
4. That's it! The TigerGraph instance has been successfully deployed on Google Cloud.



Starting TigerGraph on Your Instance

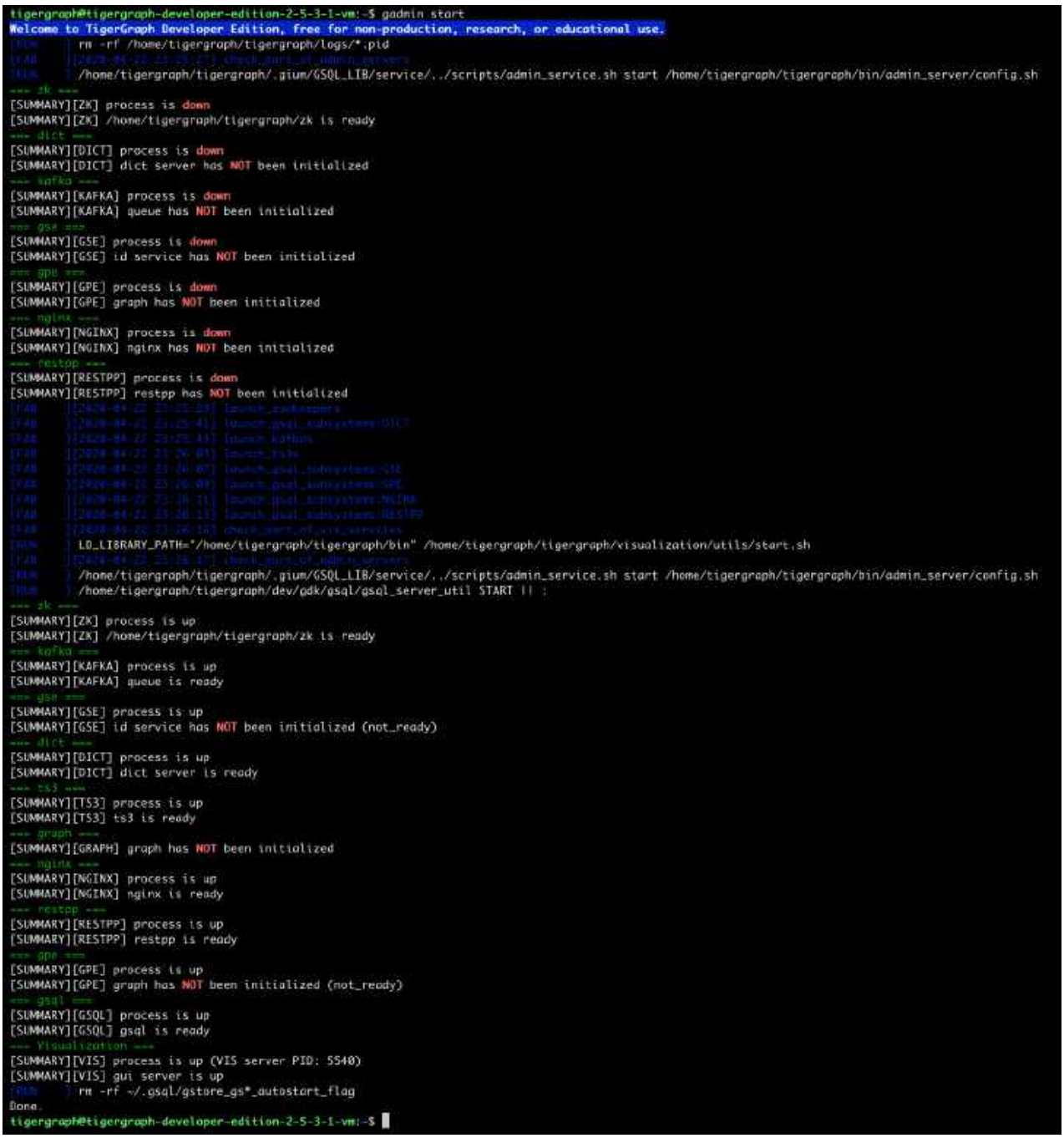
1. Log on to the instance and switch to the tigergraph user using the following command:

```
sudo su - tigergraph
```



2. Run the following command to start TigerGraph

```
gadmin start
```



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

Next Steps

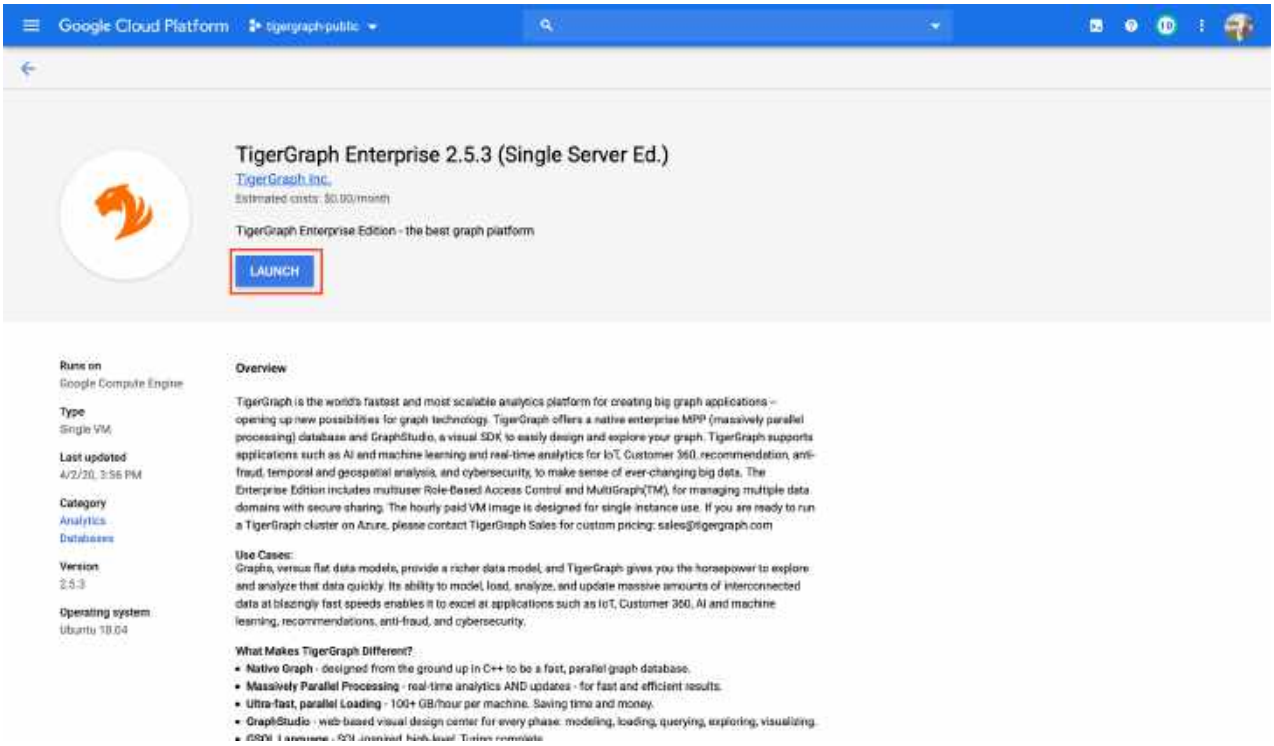
If you're new to TigerGraph, see the section "You're Ready to go ↗" for suggested next steps.

GET STARTED With TigerGraph Enterprise Edition on Google Cloud

Deploying Your Instance

1. Go to https://console.cloud.google.com/marketplace ↗ and search "Tigergraph Enterprise Edition". Choosing the latest edition (2.5.3) is recommended.

2. Click on "Launch"



3. The default setting should work for you, but feel free to modify them. When complete, click on "Deploy"

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

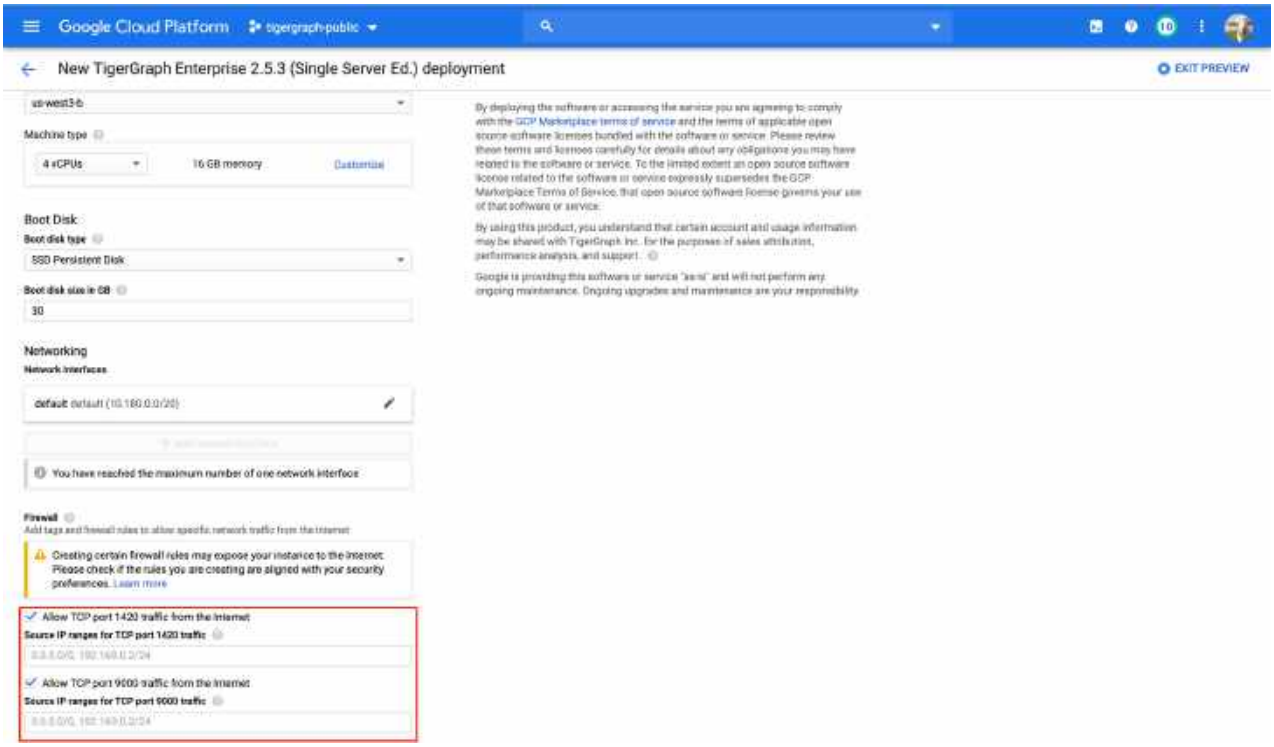
The "Allow TCP port 14240 traffic from the Internet" checkbox must be checked if

you want to access GraphStudio (TigerGraph's visualization platform). For more about GraphStudio, see the GraphStudio UI Guide.

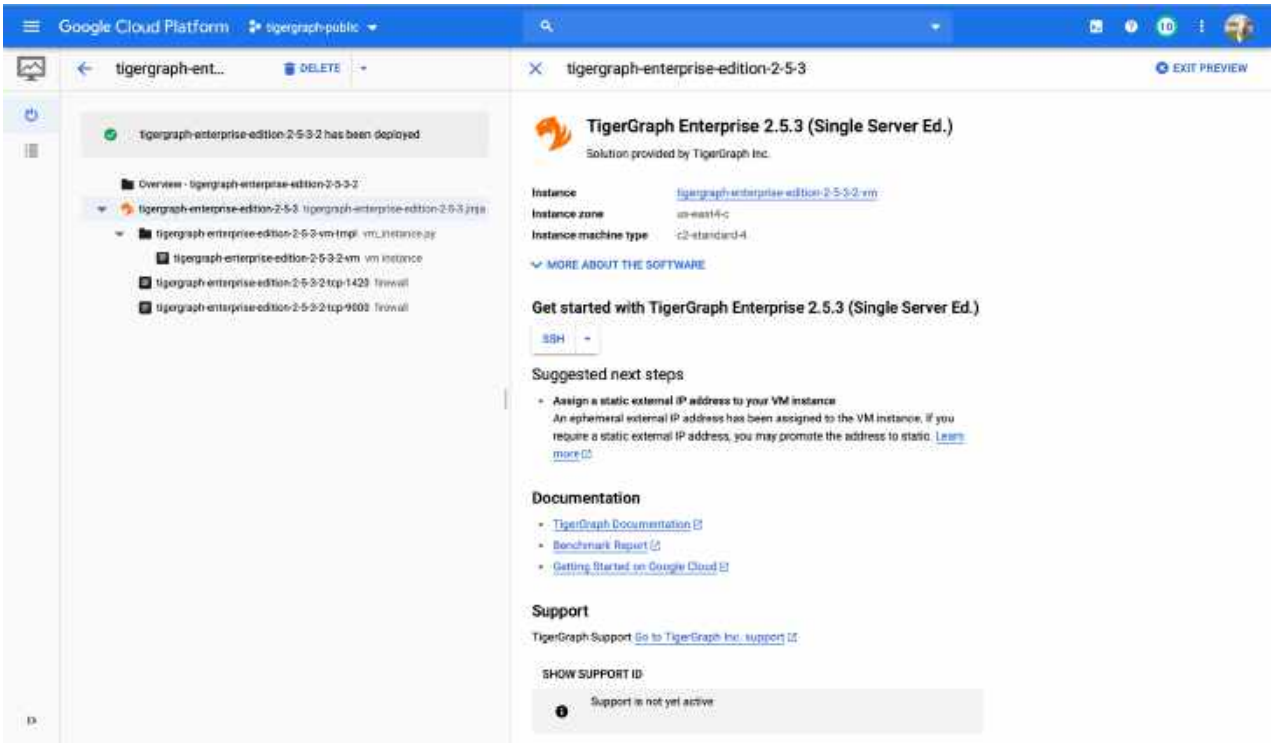
For information on how to set up authentication please see User access management

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.



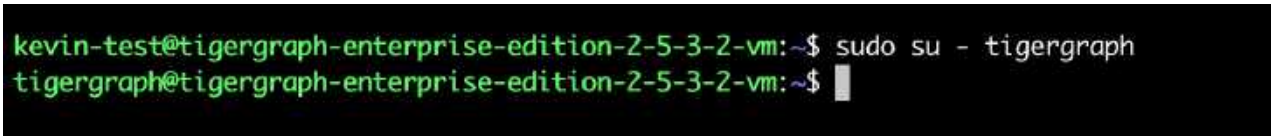
4. That's it! The TigerGraph instance has been successfully deployed on Google Cloud.



Starting TigerGraph on Your Instance

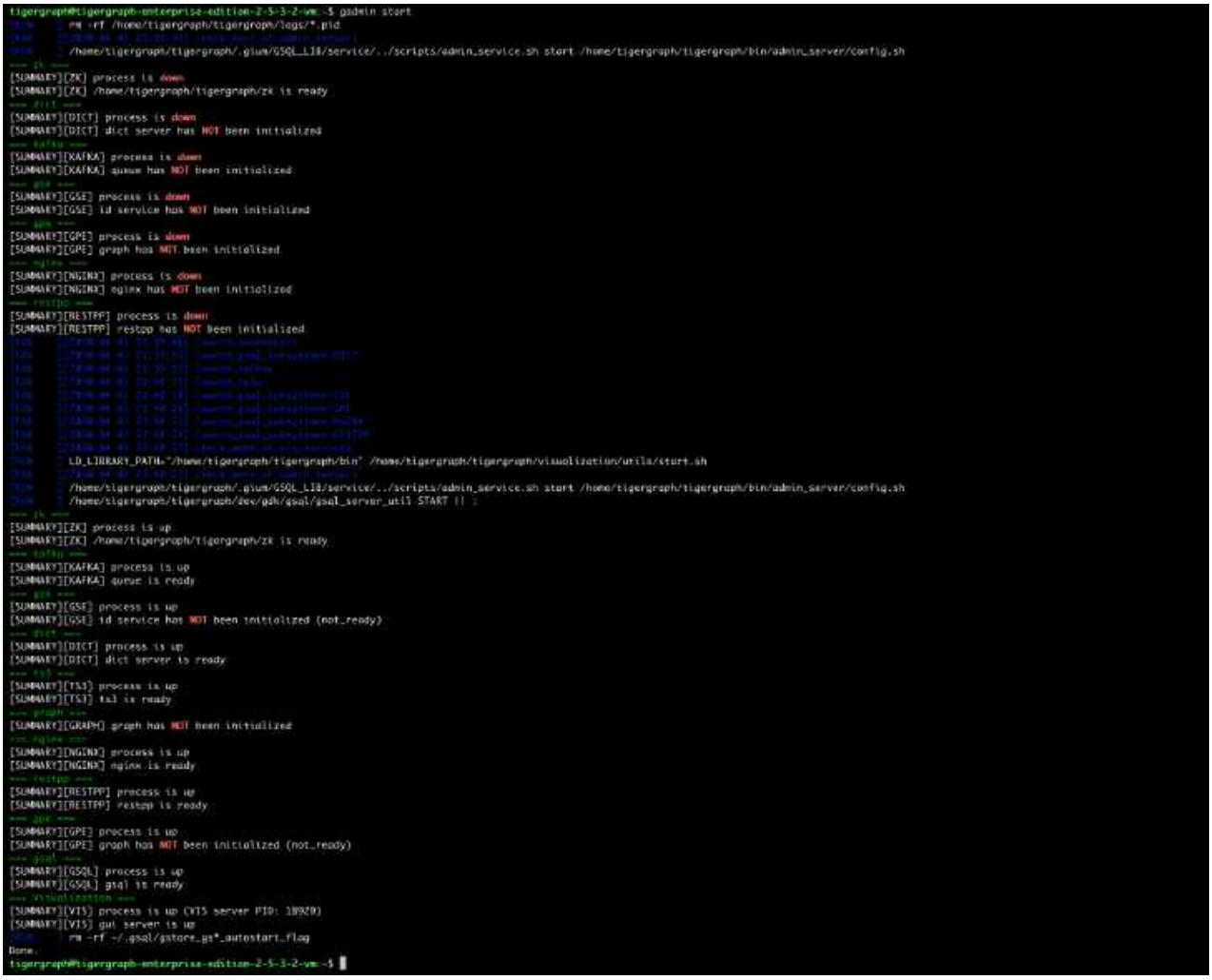
1. Log on to the instance and switch to the tigergraph user using the following command:

```
sudo su - tigergraph
```



2. Run the following command to start TigerGraph

```
gadmin start
```



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

Next Steps

If you're new to TigerGraph, see the section "You're Ready to go ↗" for suggested next steps.

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

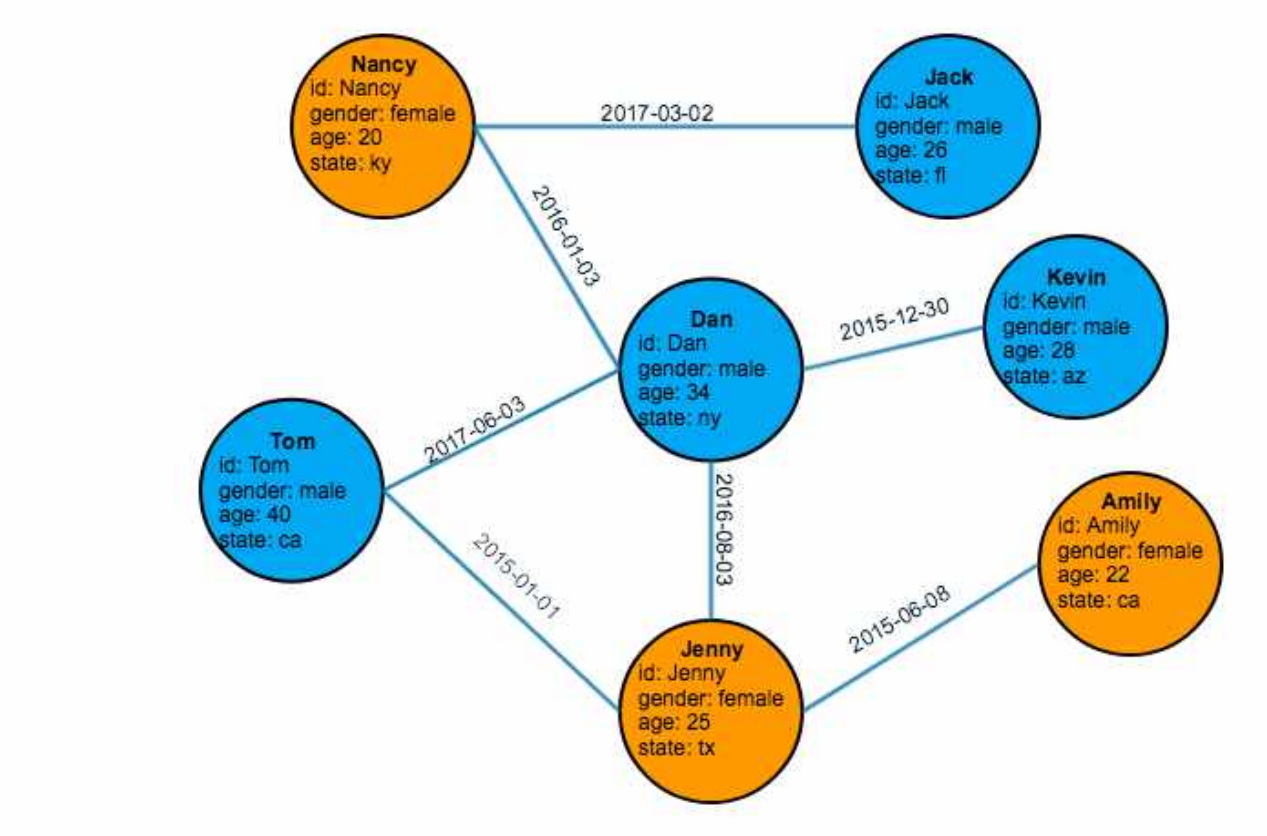
Introduction

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.

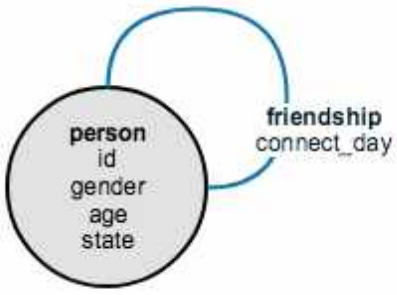
Many people call a data entity a **node** ; at TigerGraph we called it a **vertex**. The plural is **vertices**. We call a connection an edge. Both vertices and edges can have properties or attributes. The figure below is a visual representation of a graph containing 7 vertices (shown as circles) and 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". If you need further help, please see the 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 -fy
```

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 DAT
```

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 STR
Edge Types:
  - UNDIRECTED EDGE friendship(FROM person, TO person, connect_day DATETIM

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 us
[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_logs

Job "social_m1.1528095850854" loading status
[FINISHED] m1 ( Finished: 2 / Total: 2 )
  [LOADED]
  +-------------------------------------------------------------------------
  |                       FILENAME |   LOADED LINES |   AVG SPEED |   DURA
  |/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 simple built-in queries to inspect the data.

Select Vertices

The following GSQL command reports the total number of person vertices. The person.csv data file had 7 lines after the header.

GSQL command

```
SELECT count(*) FROM person
```

Similarly, the following GSQL command reports the total number of friendship edges. The friendship.csv file also had 7 lines after the header.

GSQL command

```
SELECT count(*) FROM person-(friendship)->person
```

The results are illustrated below.

GSQL shell

```
GSQL > SELECT count(*) FROM person
[{
  "count": 7,
  "v_type": "person"
}]
GSQL > SELECT count(*) FROM person-(friendship)->person
[{
  "count": 7,
  "e_type": "friendship"
}]
GSQL >
```

Counting Undirected Edges

> As of TG 2.4, undirected edges are counted once per edge. Previously, they were counted once per endpoint. E.g., the example above now returns 7 instead of 14.

If you want to see the details about a particular set of vertices, you can use "SELECT *" and the WHERE clause to specify a predicate condition. Here are some statements to try:

GSQL command

```
SELECT * FROM person WHERE primary_id=="Tom"
SELECT name FROM person WHERE state=="ca"
SELECT name, age FROM person WHERE age > 30
```

The result is in JSON format as shown below.

GSQL shell

```
GSQL > SELECT * FROM person WHERE primary_id=="Tom"
[{
  "v_id": "Tom",
  "attributes": {
    "gender": "male",
    "name": "Tom",
    "state": "ca",
    "age": 40
  },
  "v_type": "person"
}]
GSQL > SELECT name FROM person WHERE state=="ca"
[
  {
    "v_id": "Amily",
    "attributes": {"name": "Amily"},
    "v_type": "person"
  },
  {
    "v_id": "Tom",
    "attributes": {"name": "Tom"},
    "v_type": "person"
  }
]
GSQL > SELECT name, age FROM person WHERE age > 30
[
  {
    "v_id": "Tom",
    "attributes": {
      "name": "Tom",
      "age": 40
    },
    "v_type": "person"
  },
  {
    "v_id": "Dan",
    "attributes": {
      "name": "Dan",
      "age": 34
    },
    "v_type": "person"
  }
]
```

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 three parts, with some added punctuation to represent the traversal direction:

GSQL syntax

```
source_type -(edge_type)-> target_type
```

Note that the arrow → is always used, whether it's an undirected or directed edge. That is because we are describing the direction of the query's traversal (search) through the graph, not the direction of the edge itself.

We can use the from_id predicate in the WHERE clause to select all friendship edges starting from the vertex identified by the "from_id". The keyword ANY to indicate that any edge type or any target vertex type is allowed. The following two queries have the same result

GSQL command

```
SELECT * FROM person-(friendship)->person WHERE from_id =="Tom"
SELECT * FROM person-(ANY)->ANY WHERE from_id =="Tom"
```

⚠ Restrictions on built-in edge select queries

To prevent queries which might return an excessive number of output items, built-in edge queries have the following restrictions:

1. The source vertex type must be specified.
2. The from_id condition must be specified.

There is no such restriction for user-defined queries.

The result is shown below.

GSQL

```
GSQL > SELECT * FROM person-(friendship)->person WHERE from_id =="Tom"
[
  {
    "from_type": "person",
    "to_type": "person",
    "directed": false,
    "from_id": "Tom",
    "to_id": "Dan",
    "attributes": {"connect_day": "2017-06-03 00:00:00"},
    "e_type": "friendship"
  },
  {
    "from_type": "person",
    "to_type": "person",
    "directed": false,
    "from_id": "Tom",
    "to_id": "Jenny",
    "attributes": {"connect_day": "2015-01-01 00:00:00"},
    "e_type": "friendship"
  }
]
```

Another way to check the graph's size is using one of the options of the administrator tool, `gadmin`. From a Linux shell, enter the command

`gadmin status graph -v`

Linux shell

```
[tigergraph@localhost ~]$ gadmin status graph -v
verbose is ON
=== graph ===
[m1     ][GRAPH][MSG ] Graph was loaded (/usr/local/tigergraph/gstore/0/pa
[m1     ][GRAPH][INIT] True
[INFO   ][GRAPH][MSG ] Above vertex and edge counts are for internal use w
[SUMMARY][GRAPH] graph is ready
```

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) FOR GRAPH social{
  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 STR
Edge Types:
  - UNDIRECTED EDGE friendship(from person, to person, connect_day DATETIM

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", $"gende
      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=VALUE

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

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.

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) FOR GRAPH social{
  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
```

The result is shown as below.

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.
- 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.
- 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 will give you an 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 ↗.

GSQL 102 - Pattern Matching

Get Set

Introduction

In this tutorial, we will show you how to write and run Pattern Matching queries. Pattern Matching is available in TigerGraph 2.4+.

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

What is a Graph Pattern?

Pattern is a traversal trace on the graph schema. For repetitive traversal on the schema, we can use some regular expression to represent the repeating step(s). 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 conforms to a given query pattern.

Prepare Your TigerGraph Environment

We're assuming you are running Developer Edition 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. There are also links to download files at various points in the tutorial. 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 2.4 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 developer version from https://www.tigergraph.com/download/ ↗

   Linux Shell

   ```
   $ gsql
   GSQL > version
   GSQL version: 2.4
   ```

4. If the GSQL shell does not launch, try resetting the system with "gadmin start". This will take some time to launch each service if they have not been started yet. If you need further help, please see the 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 -fy	Force all TigerGraph services to restart
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 twitter-like social forum. It comes with a data generator, which allows you to generate data at different scale factors. Scale factor 1 generates roughly 1GB raw data, scale factor 10 generates roughly 10GB raw data, etc.

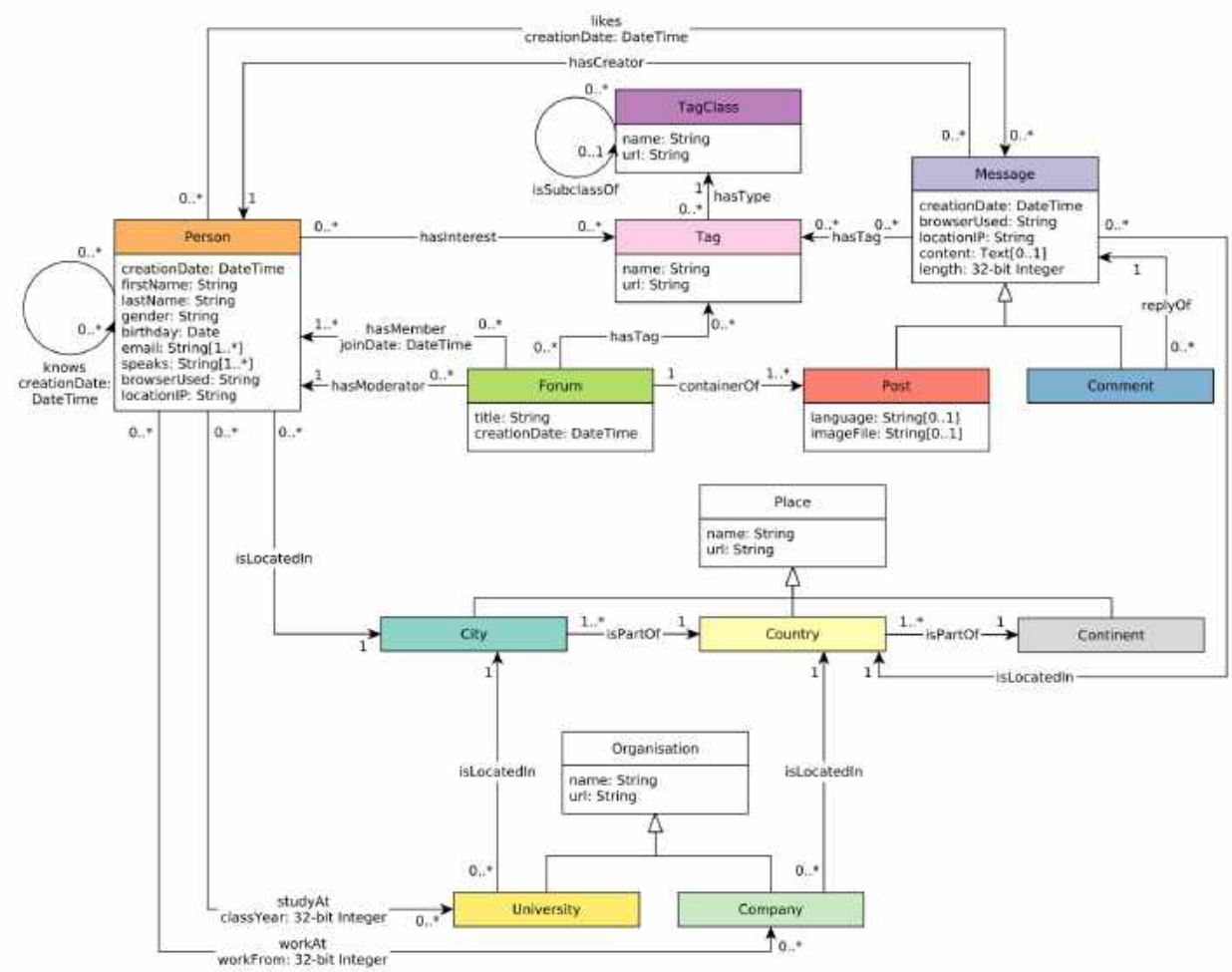


Figure 1. LDBC SNB Schema

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. Tags can be used to classify a Forum and a Person's interests. Tags can belong to a TagClass. The relationships between entities are modeled as directed edges. 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 entity type 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, we create two edge types to Country:

- **Post**_IS_LOCATED_IN_Country
- **Comment**_IS_LOCATED_IN_Country

Schema Naming Conventions

Vertex Type

For each entity in Figure 1 (the rectangular boxes), we create a vertex type with the entity's name.

- **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 related to a person's affiliation.
- **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 Type

For each relationship in Figure 1, we create an edge type whose name consists of the source entity name, the edge name **(all capitalized)**, and the target entity name. The three parts are connected by underscores.

- SourceEntityName_**EDGENAME**_TargetEntityName

For example,

- **Person_KNOWS_Person**: Person is the source and target entity names, and Knows is the edge name.
- **Person_LIKES_Comment:** Person is the source entity name, Comment is the target entity name, and Likes is the edge name.

When the edge name has two or more words, we separate words by an underscore as well. For example:

- **Tag_HAS_TYPE_TagClass:** Tag is the source entity name, TagClass is the target entity name, and hasType is the edge name (which is written as HAS_TYPE).
- **Forum_HAS_MODERATOR_Person:** Forum is the source entity name, Person is the target entity name, and hasModerator is the edge name (which is written as HAS_MODERATOR).

GSQL Schema DDL

The GSQL script below can be downloaded from this link ↗.

GSQL script

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

//vertex types
CREATE VERTEX Comment (PRIMARY_ID id UINT, id UINT, creationDate DATETIME,
CREATE VERTEX Post (PRIMARY_ID id UINT, id UINT, imageFile STRING, creatio
CREATE VERTEX Company (PRIMARY_ID id UINT, id UINT, name STRING, url STRIN
CREATE VERTEX University (PRIMARY_ID id UINT, id UINT, name STRING, url ST
CREATE VERTEX City (PRIMARY_ID id UINT, id UINT, name STRING, url STRING)
CREATE VERTEX Country (PRIMARY_ID id UINT, id UINT, name STRING, url STRIN
CREATE VERTEX Continent (PRIMARY_ID id UINT, id UINT, name STRING, url STR
CREATE VERTEX Forum (PRIMARY_ID id UINT, id UINT, title STRING, creationDa
CREATE VERTEX Person (PRIMARY_ID id UINT, id UINT, firstName STRING, lastN
CREATE VERTEX Tag (PRIMARY_ID id UINT, id UINT, name STRING, url STRING)
CREATE VERTEX TagClass (PRIMARY_ID id UINT, id UINT, name STRING, url STRI

//edge types
CREATE DIRECTED EDGE Forum_CONTAINER_OF_Post (FROM Forum, TO Post) WITH RE
CREATE DIRECTED EDGE Comment_HAS_CREATOR_Person (FROM Comment, TO Person)
CREATE DIRECTED EDGE Post_HAS_CREATOR_Person (FROM Post, TO Person) WITH F
CREATE DIRECTED EDGE Person_HAS_INTEREST_Tag (FROM Person, TO Tag) WITH RE
CREATE DIRECTED EDGE Forum_HAS_MEMBER_Person (FROM Forum, TO Person, joinD
CREATE DIRECTED EDGE Forum_HAS_MODERATOR_Person (FROM Forum, TO Person) WI
CREATE DIRECTED EDGE Comment_HAS_TAG_Tag (FROM Comment, TO Tag) WITH REVER
CREATE DIRECTED EDGE Post_HAS_TAG_Tag (FROM Post, TO Tag) WITH REVERSE_EDG
CREATE DIRECTED EDGE Forum_HAS_TAG_Tag (FROM Forum, TO Tag) WITH REVERSE_E
CREATE DIRECTED EDGE Tag_HAS_TYPE_TagClass (FROM Tag, TO TagClass) WITH RE
CREATE DIRECTED EDGE Company_IS_LOCATED_IN_Country (FROM Company, TO Count
CREATE DIRECTED EDGE Comment_IS_LOCATED_IN_Country (FROM Comment, TO Count
CREATE DIRECTED EDGE Post_IS_LOCATED_IN_Country (FROM Post, TO Country) WI
CREATE DIRECTED EDGE Person_IS_LOCATED_IN_City (FROM Person, TO City) WITH
CREATE DIRECTED EDGE University_IS_LOCATED_IN_City (FROM University, TO Ci
CREATE DIRECTED EDGE City_IS_PART_OF_Country (FROM City, TO Country) WITH
CREATE DIRECTED EDGE Country_IS_PART_OF_Continent (FROM Country, TO Contin
CREATE DIRECTED EDGE TagClass_IS_SUBCLASS_OF_TagClass (FROM TagClass, TO T
CREATE DIRECTED EDGE Person_KNOWS_Person (FROM Person, TO Person, creation
CREATE DIRECTED EDGE Person_LIKES_Comment (FROM Person, TO Comment, creati
CREATE DIRECTED EDGE Person_LIKES_Post (FROM Person, TO Post, creationDate
CREATE DIRECTED EDGE Comment_REPLY_OF_Comment (FROM Comment, TO Comment) W
CREATE DIRECTED EDGE Comment_REPLY_OF_Post (FROM Comment, TO Post) WITH RE
CREATE DIRECTED EDGE Person_STUDY_AT_University (FROM Person, TO Universit
CREATE DIRECTED EDGE Person_WORK_AT_Company (FROM Person, TO Company, work

//LDBC SNB graph schema
CREATE GRAPH ldbc_snb (*)
```

Load Data

Define the Loading Job

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

You can download the below loading script from here ↗.

GSQL Loading Script

```
USE GRAPH ldbc_snb
CREATE LOADING JOB load_ldbc_snb FOR GRAPH ldbc_snb {
  // define vertex source files
  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 source files
  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, $0, $1, $2, $3, $4, $5) USING header="tr
  LOAD v_post_file
    TO VERTEX Post VALUES ($0, $0, $1, $2, $3, $4, $5, $6, $7) USING heade
  LOAD v_organisation_file
    TO VERTEX Company VALUES ($0, $0, $2, $3) WHERE $1=="company",
    TO VERTEX University VALUES ($0, $0, $2, $3) WHERE $1=="university" US
  LOAD v_place_file
    TO VERTEX City VALUES ($0, $0, $1, $2) WHERE $3=="city",
    TO VERTEX Country VALUES ($0, $0, $1, $2) WHERE $3=="country",
```

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

// load edge
LOAD forum_containerOf_post_file
    TO EDGE Forum_CONTAINER_OF_Post VALUES ($0, $1) USING header="true", s
LOAD comment_hasCreator_person_file
    TO EDGE Comment_HAS_CREATOR_Person VALUES ($0, $1) USING header="true"
LOAD post_hasCreator_person_file
    TO EDGE Post_HAS_CREATOR_Person VALUES ($0, $1) USING header="true", s
LOAD person_hasInterest_tag_file
    TO EDGE Person_HAS_INTEREST_Tag VALUES ($0, $1) USING header="true", s
LOAD forum_hasMember_person_file
    TO EDGE Forum_HAS_MEMBER_Person VALUES ($0, $1, $2) USING header="true
LOAD forum_hasModerator_person_file
    TO EDGE Forum_HAS_MODERATOR_Person VALUES ($0, $1) USING header="true"
```

Prepare The Raw Data

Linux Bash

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

```
LOAD post_hasTag_tag_file
    TO EDGE Post_HAS_TAG_Tag VALUES ($0, $1) USING header="true", separato
LOAD forum_hasTag_tag_file
    TO EDGE Forum_HAS_TAG_Tag VALUES ($0, $1) USING header="true", separat
LOAD tag_hasType_tagclass_file
    TO EDGE Tag_HAS_TYPE_TagClass VALUES ($0, $1) USING header="true", sep
```

Linux Bash

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

```
    TO EDGE University_IS_LOCATED_IN_City VALUES ($0, $1) WHERE to_int($1)
LOAD comment_isLocatedIn_place_file
    TO EDGE Comment_IS_LOCATED_IN_Country VALUES ($0, $1) USING header="tr
LOAD post_isLocatedIn_place_file
    TO EDGE Post_IS_LOCATED_IN_Country VALUES ($0, $1) USING header="true"
LOAD person_isLocatedIn_place_file
    TO EDGE Person_IS_LOCATED_IN_City VALUES ($0, $1) USING header="true",
LOAD place_isPartOf_place_file
    TO EDGE Country_IS_PART_OF_Continent VALUES ($0, $1) WHERE to_int($0)
    TO EDGE City_IS_PART_OF_Country VALUES ($0, $1) WHERE to_int($0) > 110
LOAD tagclass_isSubclassOf_tagclass_file
    TO EDGE TagClass_IS_SUBCLASS_OF_TagClass VALUES ($0, $1) USING header=
LOAD person_knows_person_file
    TO EDGE Person_KNOWS_Person VALUES ($0, $1, $2) USING header="true", s
LOAD person_likes_comment_file
    TO EDGE Person_LIKES_Comment VALUES ($0, $1, $2) USING header="true",
```

Run The Loading Job

```
  LOAD person_likes_post_file
    TO EDGE Person_LIKES_Post VALUES ($0, $1, $2) USING header="true", sep
  LOAD comment_replyOf_comment_file
    TO EDGE Comment_REPLY_OF_Comment VALUES ($0, $1) USING header="true",
  LOAD comment_replyOf_post_file
    TO EDGE Person_STUDY_AT_University VALUES ($0, $1, $2) USING header="t
  LOAD person_workAt_organisation_file
    TO EDGE Person_WORK_AT_Company VALUES ($0, $1, $2) USING header="true"
}
```

Linux Bash

```
gsql setup_schema.gsql
```

/home/tigergraph/ldbc_snb_data/social_network. Note, the folder should have the name **social_network**.

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

#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/gsql102$ ./load_data.sh
[Tip: Use "CTRL + C" to stop displaying the loading status update, then us
[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_s

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_hasCreator
  |            /mnt/data/download/ldbc_snb_data/social_network/comment_has
  |     /mnt/data/download/ldbc_snb_data/social_network/comment_isLocatedI
  |       /mnt/data/download/ldbc_snb_data/social_network/comment_replyOf_
  |          /mnt/data/download/ldbc_snb_data/social_network/comment_reply
  |                           /mnt/data/download/ldbc_snb_data/social_networ
  |        /mnt/data/download/ldbc_snb_data/social_network/forum_container
  |        /mnt/data/download/ldbc_snb_data/social_network/forum_hasMember
  |     /mnt/data/download/ldbc_snb_data/social_network/forum_hasModerator
  |              /mnt/data/download/ldbc_snb_data/social_network/forum_has
  |                  /mnt/data/download/ldbc_snb_data/social_network/organ
  |/mnt/data/download/ldbc_snb_data/social_network/organisation_isLocatedI
  |                          /mnt/data/download/ldbc_snb_data/social_network
  |        /mnt/data/download/ldbc_snb_data/social_network/person_hasInter
  |      /mnt/data/download/ldbc_snb_data/social_network/person_isLocatedI
  |           /mnt/data/download/ldbc_snb_data/social_network/person_knows
  |          /mnt/data/download/ldbc_snb_data/social_network/person_likes_
  |             /mnt/data/download/ldbc_snb_data/social_network/person_lik
  |   /mnt/data/download/ldbc_snb_data/social_network/person_studyAt_organ
  |    /mnt/data/download/ldbc_snb_data/social_network/person_workAt_organ
  |                           /mnt/data/download/ldbc_snb_data/social_networ
  |          /mnt/data/download/ldbc_snb_data/social_network/place_isPartO
  |                            /mnt/data/download/ldbc_snb_data/social_netwo
  |        /mnt/data/download/ldbc_snb_data/social_network/post_hasCreator
  |               /mnt/data/download/ldbc_snb_data/social_network/post_has
  |        /mnt/data/download/ldbc_snb_data/social_network/post_isLocatedI
  |                             /mnt/data/download/ldbc_snb_data/social_netw
  |          /mnt/data/download/ldbc_snb_data/social_network/tag_hasType_t
  |                        /mnt/data/download/ldbc_snb_data/social_network/t
  |/mnt/data/download/ldbc_snb_data/social_network/tagclass_isSubclassOf_t
  +-------------------------------------------------------------------------
```

After loading, you can check the graph's size using one of the options of the administrator tool, `gadmin`. From a Linux shell, enter the command

`gadmin status graph -v`

Linux shell

```
gadmin status graph -v
verbose is ON
=== graph ===
[m1     ][GRAPH][MSG ] Graph was loaded (/mnt/data/tigergraph/gstore/0/pa
[m1     ][GRAPH][INIT] True
[INFO   ][GRAPH][MSG ] Above vertex and edge counts are for internal use w
[SUMMARY][GRAPH] graph is ready
```

You should see VertexCount: **3,181,724** and EdgeCount **34,512,076**.

Basic Pattern Concepts

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 focus on the linear pattern.

> Currently, pattern matching may only be used in read-only 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:

```
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.

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. e1 is the alias of E1.
2. `FROM 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 `(<_|_>|_)`

How To Enter Pattern Match Syntax Mode

To use the pattern match syntax, 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 new SYNTAX option 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.

CHANGE ADVISORY

The punctuation used with the SYNTAX keyword was streamlined, from

```
CREATE QUERY <query_name><parameters> FOR GRAPH <graph_name> SYNTAX ("v2")
# original version, TigerGraph 2.4.0
```

to

```
CREATE QUERY <query_name><parameters> FOR GRAPH <graph_name> SYNTAX v2
# final version, since TigerGraph 2.4.1
```

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 the new Interpreted Mode for GSQL, also introduced in TigerGraph 2.4. Interpreted mode lets us skip the INSTALL step, and even to 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 send an anonymous query to the interpret engine, 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 milli-second
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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
   #start with all persons.
   Seed = {Person.*};
   #1-hop pattern.
   friends = SELECT p
             FROM Seed:s - (<Person_KNOWS_Person: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];
}
```

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. Do the same as Example 1, but use a right-directed edge pattern.

Example 2. Right Directed Edge Pattern

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
   #start with all persons.
   Seed = {Person.*};
   #1-hop pattern.
   friends = SELECT s
             FROM Seed:s - (Person_KNOWS_Person>:e) - Person:p
             WHERE p.firstName == "Viktor" AND p.lastName == "Akhiezer"
             ORDER BY s.birthday ASC
             LIMIT 3;

    PRINT  friends[friends.firstName, friends.lastName, friends.birthday];
}
```

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
```

The output should be the same as example1's output.

Example 3. Find Viktor Akhiezer's total number of comments, total number of posts, and total number of persons he knows. A Person can reach Comments, Posts and other Persons via a directed edge.

Example 3. Right Directed Any Edge Pattern.

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
   SumAccum<int> @commentCnt= 0;
   SumAccum<int> @postCnt= 0;
   SumAccum<int> @personCnt= 0;

   #start with all persons.
   Seed = {Person.*};
   #1-hop pattern.
   Result = SELECT s
            FROM Seed:s - (_>:e) - :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
                      WHEN tgt.type == "Person" THEN
                           s.@personCnt += 1
                  END;

    PRINT  Result[Result.@commentCnt, Result.@postCnt, Result.@personCnt];
}
```

You can copy the above GSQL script to a file named example3.gsql, and invoke this script file in Linux.

Linux Bash

```
gsql example3.gsql
```

Output of Example 3.

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"Result": [{
    "v_id": "28587302323577",
    "attributes": {
      "Result.@personCnt": 25,
      "Result.@commentCnt": 152,
      "Result.@postCnt": 96
    },
    "v_type": "Person"
  }]}]
}
```

Example 4. Do the same as Example 3, but use a left-directed edge pattern.

Note below (line 10) that the Seed is now {Person.*, Comment.*, Post.* }, the three types of entities that are targets of edges from a Person.

> In the current version, the vertex set on the left side of the pattern must be defined in a previous statement (e.g., a seed statement), the same requirement as in v1 syntax FROM clauses. In the example below, the current version of pattern matching would not permit
> FROM _:s -(<:e) - Person:tgt

Example 4. Left Directed Any Edge Pattern

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
   SumAccum<int> @commentCnt= 0;
   SumAccum<int> @postCnt= 0;
   SumAccum<int> @personCnt= 0;

   #start with all persons, comments, and posts
   Seed = {Person.*, Comment.*, Post.*};
   #1-hop pattern.
   Result = SELECT tgt
            FROM Seed:s - (<_:e) - Person:tgt
            WHERE tgt.firstName == "Viktor" AND tgt.lastName == "Akhiezer"
            ACCUM  CASE WHEN s.type == "Comment" THEN
                          tgt.@commentCnt += 1
                        WHEN s.type == "Post" THEN
                          tgt.@postCnt += 1
                       WHEN s.type == "Person" THEN
                          tgt.@personCnt += 1
                    END;

    PRINT  Result[Result.@commentCnt, Result.@postCnt, Result.@personCnt];
}
```

You can copy the above GSQL script to a file named example4.gsql, and invoke this script file in linux command line. The output should be the same as in Example 3.

Example 5. Find the two oldest persons who either know "Viktor Akhiezer" or are known by "Vicktor Akhiezer". KNOWS is a directed relationship, so we need to include both directions in the pattern.

Example 5. Disjunctive 1-hop edge pattern.

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
   #start with all persons.
   Seed = {Person.*};
   #1-hop pattern.
   friends = SELECT p
             FROM Seed:s - ((<Person_KNOWS_Person|Person_KNOWS_Person>):e)
             WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
             ORDER BY p.birthday ASC
             LIMIT 2;

   PRINT friends;
}
```

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": [{"friends": [
    {
      "v_id": "10995116279461",
      "attributes": {
        "birthday": "1980-05-13 00:00:00",
        "firstName": "Gregorio",
        "lastName": "Cajes",
        "gender": "male",
        "speaks": [
          "en",
          "tl"
        ],
        "browserUsed": "Firefox",
        "locationIP": "110.55.251.62",
        "id": 10995116279461,
        "creationDate": "2010-12-16 18:12:57",
        "email": ["Gregorio10995116279461@gmail.com"],
        "@multPropagAcc_1": 0
      },
      "v_type": "Person"
    },
    {
      "v_id": "4398046517846",
      "attributes": {
        "birthday": "1980-04-24 00:00:00",
        "firstName": "Abdul-Malik",
        "lastName": "Glosca",
        "gender": "male",
        "speaks": [
          "ar",
          "en"
        ],
        "browserUsed": "Chrome",
        "locationIP": "109.200.168.137",
        "id": 4398046517846,
        "creationDate": "2010-05-21 00:07:05",
        "email": [
          "Abdul-Malik4398046517846@gmail.com",
          "Abdul-Malik4398046517846@gmx.com",
```

```
            "Abdul-Malik4398046517846@land.ru"
          ],
          "@multPropagAcc_1": 0
        },
        "v_type": "Person"
      }
    ]}]
}
```

Example 6. Find the total comments or posts created by "Viktor Akhiezer". Again, we include two types of edges, but in this case, we count them together.

Example 6. Disjunctive 1-hop edge pattern.

```
USE GRAPH ldbc_snb
#pattern match syntax version is v2
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
   SumAccum<int> @@cnt = 0;
   Seed = {Person.*};

   friends = SELECT t
             FROM Seed:s-((<Comment_HAS_CREATOR_Person|<Post_HAS_CREATOR_P
             WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
             ACCUM  @@cnt += 1 ;

  PRINT @@cnt;
}
```

You can copy the above GSQL script to a file named example6.gsql, and invoke this script file in Linux:

Linux Bash

```
gsql example6.gsql
```

Output of Example 6.

```
Using graph 'ldbc_snb'
{
  "error": false,
  "message": "",
  "version": {
    "schema": 0,
    "edition": "enterprise",
    "api": "v2"
  },
  "results": [{"@@cnt": 89}]
}
```

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 - (E1*) - Y:y`
  2. `FROM X:x - (E2>*) - Y:y`
  3. `FROM X:x - (<E3*) - 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 - ((E1|E2>|<E3)*) - Y:y`
     - Either E1, E2> or <E3 can be chosen at each repetition.
- **1-hop star pattern with bounds**
  1. `FROM X:x - (E1*2..) - Y:y`
     - Lower bounds only. There is a chain of at least 2 E1 edges.
  2. `FROM X:x - (E2>*..3) - Y:y`
     - Upper bounds only. There is a chain of between 0 and 3 E2 edges.

3. `FROM X:x - (<E3*3..5) - Y:y`
    - Both Lower and Upper bounds. There is a chain of 3 to 5 E3 edges.
4. `FROM X:x - ((E1|E2>|<E3)*3) - Y:y`
    - Exact bound. There is a chain of exactly 3 edges, where each edge is either E1, E2>, or <E3.

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:

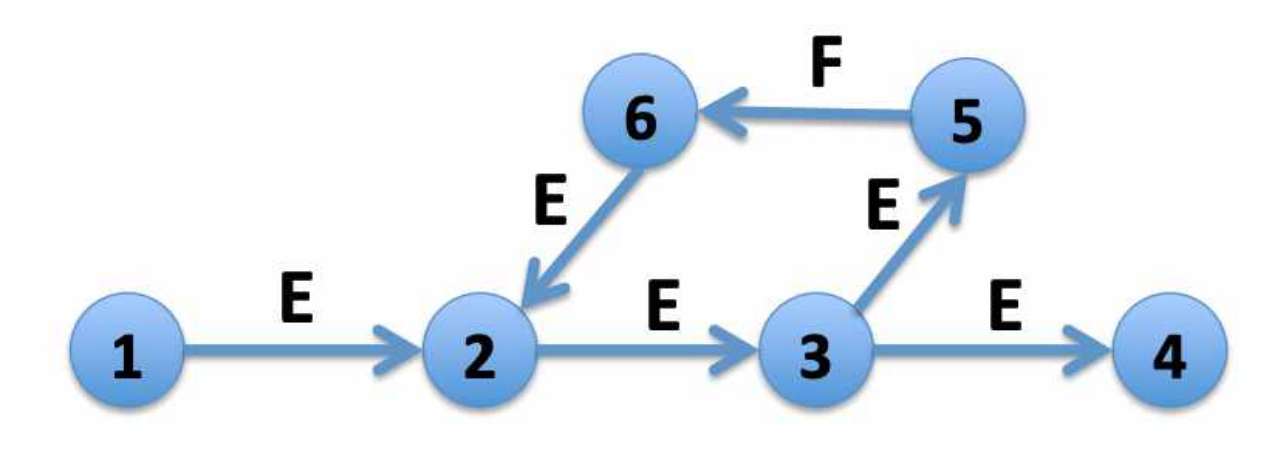


Figure 2 Shortest Path Illustration.

In Figure 2, for 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 new Interpreted Mode for GSQL, introduced in TigerGraph 2.4. Interpreted mode lets us skip the INSTALL step, and even to 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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  TagClass1 = {TagClass.*};

  TagClass2 =  SELECT t
               FROM TagClass1:s - (TagClass_IS_SUBCLASS_OF_TagClass>*)-Tag
               WHERE s.name == "TennisPlayer";

    PRINT  TagClass2;
}
```

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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  TagClass1 = {TagClass.*};

  TagClass2 =  SELECT t
               FROM TagClass1:s - (TagClass_IS_SUBCLASS_OF_TagClass>*1)-Ta
               WHERE s.name == "TennisPlayer";

    PRINT  TagClass2;
}
```

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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  TagClass1 = {TagClass.*};

  TagClass2 =  SELECT t
               FROM TagClass1:s - (TagClass_IS_SUBCLASS_OF_TagClass>*1..2)
               WHERE s.name == "TennisPlayer";

  PRINT  TagClass2;
}
```

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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  TagClass1 = {TagClass.*};

  TagClass2 =  SELECT t
               FROM TagClass1:s - (TagClass_IS_SUBCLASS_OF_TagClass>*..2)-
               WHERE s.name == "TennisPlayer";

  PRINT  TagClass2;
}
```

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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  TagClass1 = {TagClass.*};

  TagClass2 =SELECT t
             FROM TagClass1:s - (TagClass_IS_SUBCLASS_OF_TagClass>*1..)-Ta
             WHERE s.name == "TennisPlayer";

  PRINT  TagClass2;
}
```

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
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
  SumAccum<int> @@commentCnt = 0;

  #start with all persons.
  Seed = {Person.*};

  # find top 3 latest comments that is liked or created by Viktor Akhiezer
  # and the total number of comments related to Viktor Akhiezer
  Top3Comments = SELECT p
    FROM Seed:s - ((<Comment_HAS_CREATOR_Person|Person_LIKES_Comment>)*1)
    WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
    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 far
        },
        "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, fiv
        },
        "v_type": "Comment"
      }
    ]},
```

```
      {"@@commentCnt": 152}
    ]
  }
```

Multiple Hop Patterns

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 did not need to specify that the intermediate vertex type is Y, nor need to refer to that vertex set 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:e1.E2>:e2) - Z:z
```

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, 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.

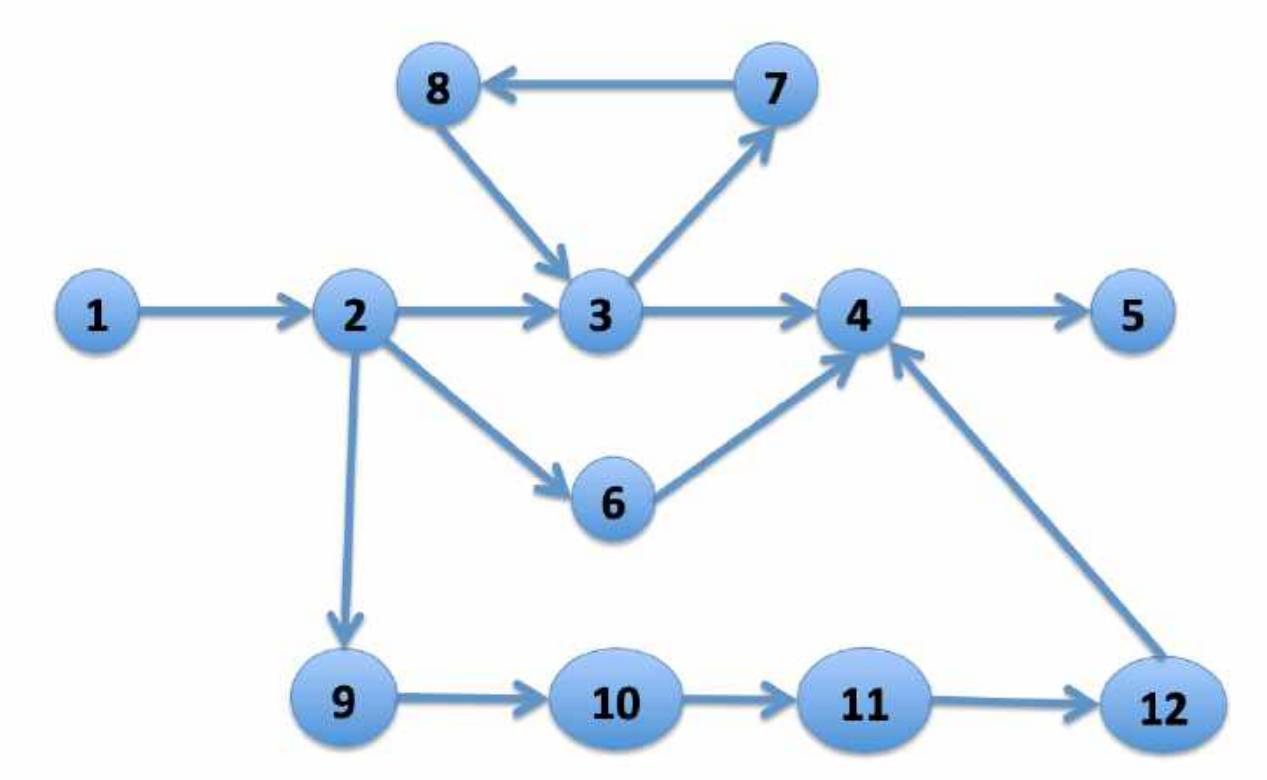


Figure 3. Shortest Path Illustration

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 number of paths to consider is NP.
- 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 infinite matches, because we can go around the 3-7-8-3 cycle any number of times.

Additional Details about Pattern Matching

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.

TigerGraph 2.4 has certain restrictions on how accumulators and aliases can be used. Some or all of these restrictions will be lifted in future releases. We elaborate on them below.

SELECT Clause

The SELECT clause specifies the output vertex set of a SELECT statement. For a multiple-hop pattern, we can only select one of the two endpoints of the pattern. None of the intermediate aliases can be selected. The example below shows the two possible choices for the given pattern:

SELECT Clause Can Select End Points Only

```
#select starting end point x
SELECT x
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

```
#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

In a multiple-hop pattern, the WHERE clause is a conjunction of local hop predicates (conditions), with the exception that the starting end point can appear in the last hop local predicate.

Consider a pattern

```
X1:x1-(E1:e1)-X2:x2-(E2:e2)-X3:x3-(E3:e3)-X4:x4
```

- **Local hop predicate** means a predicate can refer to a single hop's alias (x_i, e_i, x_(i+1)) only.
- **Last hop local predicate** can refer to the pattern's starting end point. I.e. (x1, x3, e3, x4).
- **A WHERE clause** is a conjunction (AND) of local hop predicates.

WHERE Clause Support "AND" of Local Predicate

```
# (x, e2, y) belongs to the 1-hop
# (y, e3, z) belongs to the 2-hop
# (x, z, e4, u) is the last hop local predicate
SELECT x
FROM X:x-(E2>:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u
WHERE x.age > y.age AND y.name != z.name AND (x.salary + z.salary) < u.sal
```

- **Kleene Star breaks local hop predicate.** When a local hop's edge has kleene star, we cannot compose a local predicate using the local alias'.

Kleene Star Break Local Predicate

```
# (x,y) belongs to the 1-hop
# which has *, then semantic error will be given
SELECT x
FROM X:x-(E2>*:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u
WHERE x.age > y.age AND y.name != z.name AND (x.salary + z.salary) < u.sal
```

ACCUM and POST-ACCUM Clauses

In the current version of GSQL, only certain parts of a pattern (and their corresponding aliases) are available in ACCUM and POST-ACCUM clauses. Refer to the example pattern and its highlighted parts in Figure 4 below.

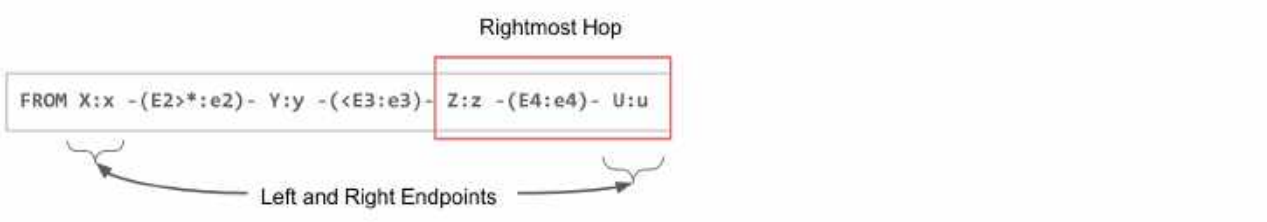


Figure 4. Example pattern to show ACCUM behavior

1. Accumulators can only be attached to the pattern's endpoints (x and u in the figure). The accumulation statements may only access data from either the left endpoint (x) or the rightmost hop (z, e4, and u). Below is an example of a valid ACCUM clause.
2. In the POST-ACCUM clause, only the pattern's endpoints can be accessed.
3. For queries in Distributed mode, accumulators may only be on the right endpoint (x in the figure).

The example below shows a valid ACCUM clause:

ACCUM To The Two End Points of A Pattern

```
# (z,e4, u) belongs to the last-hop aliass
# (x, u) are the end points of the pattern
SELECT x
FROM X:x-(E2>*:e2)-Y:y-(<E3:e3)-Z:z-(E4:e4)-U:u
ACCUM x.@cnt += z.id, u.@cnt += e4.id
```

Examples of Multiple Hop Pattern Match

Example 1. Find the 3rd superclass of the Tag class whose name is "TennisPlayer".

Example1. Succict Representation Of Multiple-hop Pattern

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  TagClass1 = {TagClass.*};

  TagClass2 =  SELECT t
    FROM TagClass1:s - (TagClass_IS_SUBCLASS_OF_TagClass>.TagClass_IS_SUBC
    WHERE s.name == "TennisPlayer";

  PRINT  TagClass2;
}
```

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 Succict Representation Of Multiple-hop Pattern

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {

  SumAccum<String> @continentName;

  msg = {Comment.*, Post.*};

  accMsgContinent =
    SELECT s
    FROM msg:s-((Post_IS_LOCATED_IN_Country>|Comment_IS_LOCATED_IN_Country
    WHERE  year(s.creationDate) == 2011 AND month(s.creationDate) == 1
    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, iews. His writings and methods wer
```

```
      },
      "v_type": "Post"
    }
  ]}]
}
```

Example 3. Find Viktor Akhiezer's favorite author of 2012 whose last name begins with character 'S', and how many LIKES Viktor give to the author's post or comment.

Example 3. Multiple-hop Pattern With Accumulator Applied To All Matched Paths

```
USE GRAPH ldbc_snb
SET syntax_version="v2"

INTERPRET QUERY () FOR GRAPH ldbc_snb {
  SumAccum<int> @likesCnt;
  PersonAll = {Person.*};

  FavoriteAuthors =  SELECT t
              FROM PersonAll:s-((Person_LIKES_Comment>|Person_LIKES_Post>
              WHERE s.firstName == "Viktor" AND s.lastName == "Akhiezer"
              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"
    }
  ]}]
}
```

Application And Benchmark Queries

We have demonstrated the basic pattern match syntax. You should have mastered the basics by this point. The next step is to see more examples and practice more.

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 syntax_version="v2"
set query_timeout=60000

DROP QUERY RecommendMessage

CREATE QUERY RecommendMessage (String fn, String ln) FOR GRAPH ldbc_snb {

  SumAccum<int> @TagInCommon;
  SumAccum<float> @SimilarityScore;
  SumAccum<float> @Rank;
  OrAccum @Liked = false;

  Seed = {Person.*};

  #mark messages liked by Viktor
    MessageLiked =
       SELECT msg
       FROM Seed:s-((Person_LIKES_Comment>|Person_LIKES_Post>))-:msg
       WHERE s.firstName == fn AND s.lastName == ln
       ACCUM msg.@Liked = true;

   #calculate log similarity score for each persons share the same interes
    Others   =
       SELECT p
       FROM Seed:s-((Person_LIKES_Comment>|Person_LIKES_Post>).(Comment_HA
        - ((<Comment_HAS_TAG_Tag|<Post_HAS_TAG_Tag).(<Person_LIKES_Comment
       WHERE s.firstName == fn AND s.lastName == ln
       ACCUM 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-((Person_LIKES_Comment>|Person_LIKES_Post>)) -
              WHERE  msg.@Liked == false
              ACCUM msg.@Rank +=o.@SimilarityScore
              ORDER BY msg.@Rank DESC
              LIMIT 2;

  PRINT   RecommendedMessage[RecommendedMessage.content, RecommendedMessag
}


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 establi
      },
      "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 syntax_version="v2"
SET query_timeout=60000

DROP QUERY RecommendMessage

CREATE QUERY RecommendMessage (String fn, String ln) FOR GRAPH ldbc_snb {

  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;

  Seed = {Person.*};

   #mark messages liked by input user
    InputPerson =
       SELECT s
       FROM Seed:s-((Person_LIKES_Comment>|Person_LIKES_Post>))-:msg
       WHERE s.firstName == fn AND s.lastName == ln
       ACCUM msg.@Liked = true, @@InputPersonMsgCnt += 1;

    sqrtOfInputPersonMsgCnt = sqrt(@@InputPersonMsgCnt);

   #find common msg between input user and other persons
    Others   =
       SELECT p
       FROM InputPerson:s-((Person_LIKES_Comment>|Person_LIKES_Post>))-:ms
                       -((<Person_LIKES_Comment|<Person_LIKES_Post))-:p
       ACCUM p.@MsgInCommon += 1;

    #calculate cosine similarity score.
    #|AxB|/(sqrt(Sum(A_i^2)) * sqrt(Sum(B_i^2)))
    Others  =
        SELECT o
        FROM Others:o-((Person_LIKES_Comment>|Person_LIKES_Post>))-:msg
        ACCUM o.@MsgCnt += 1
        POST-ACCUM o.@SimilarityScore = o.@MsgInCommon/(sqrtOfInputPersonM

   #recommend new messages to input user that have not liked by him.
    RecommendedMessage =
             SELECT msg
             FROM Others:o-((Person_LIKES_Comment>|Person_LIKES_Post>)) -
             WHERE  msg.@Liked == false
```

```
              ACCUM msg.@Rank +=o.@SimilarityScore
              ORDER BY msg.@Rank DESC
              LIMIT 2;

  PRINT   RecommendedMessage[RecommendedMessage.content, RecommendedMessag
}

INTERPRET query RecommendMessage("Viktor", "Akhiezer")
```

LDBC SNB Benchmark Queries

We have made available all LDBC SNB benchmark queries translated into GSQL pattern matching syntax. The benchmark queries are described in Sections 4 and 5 of the LDBC Social Network Benchmark Reference ↗.

You can find our GSQL pattern match translations of these queries on Github: https://github.com/tigergraph/ecosys/tree/master/tools/ldbc_benchmark/tigergraph/queries_pattern_match ↗

Note we use CREATE/INSTALL/RUN QUERY instead of INTERPRET QUERY so that these queries can be optimized and installed as REST services. There are three sets of queries:

- Interactive Short Queries ↗
- Complex Short Queries ↗
- Business Intelligence Queries ↗

Also, you may want to use the GraphStudio UI to help you visualize and explore the graph and to try your hand at writing your own queries.

Last but not least, join GSQL community forum ↗, discuss and get help from fellow GSQL users and GSQL developers.

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.

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.

TigerGraph Platform Overview

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:

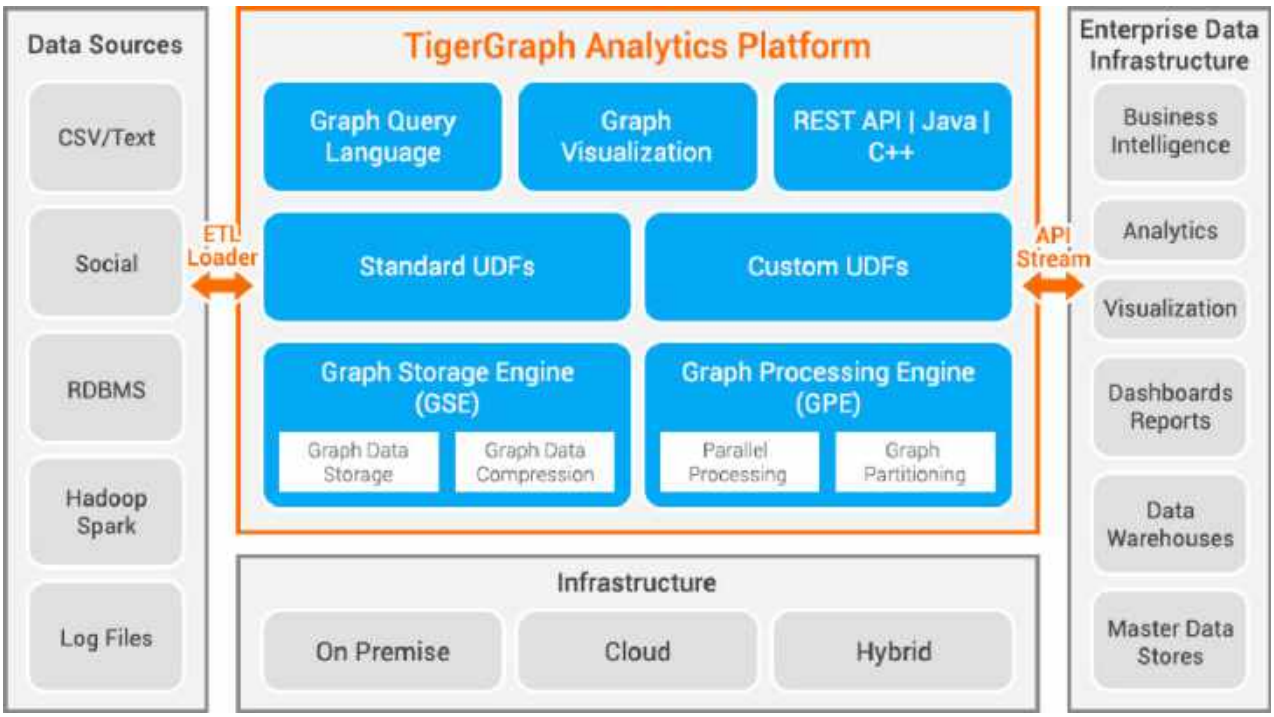
- Fast data loading speed to build graphs - able to load 50 to 150 GB of data per hour, per machine
- Fast execution of parallel graph algorithms - able to traverse hundreds of million of vertices/edges per second per machine
- Real-time updates and inserts using REST - able to stream 2B+ daily events 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.

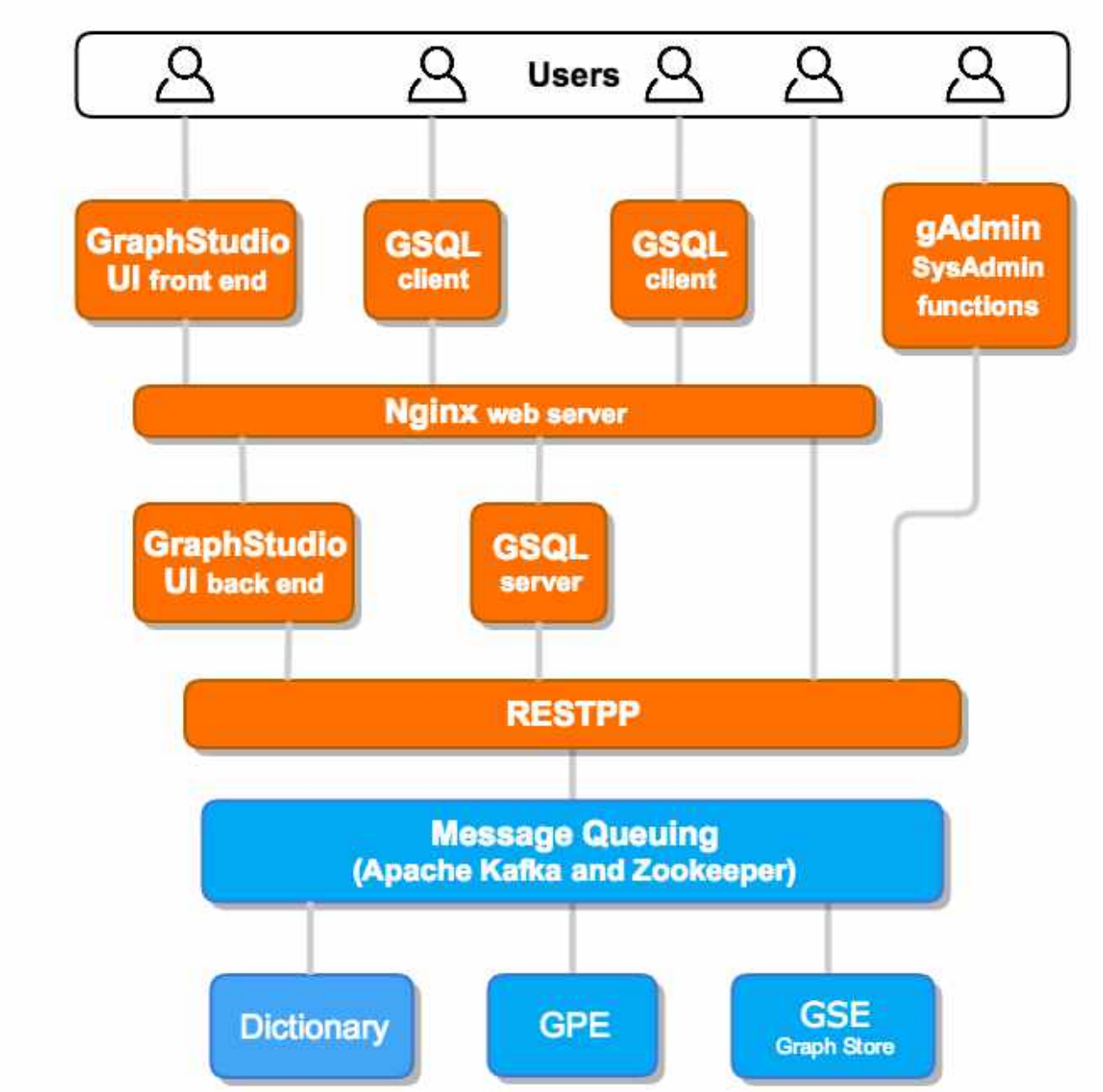
System Overview

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 c loser loo k 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.
IUM	Installation, Upgrade, and Maintenance (generic term). These functions are handled by gadmin.
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.
	A free open-source program from the

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. Contact support@tigergraph.com ↗for help in your specific situation. As of this writing (May 2018), the TigerGraph Platform is releasing version 2.0, which is a major revision.

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 -h`

`$ man gadmin`

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, post
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?

Starting with v1.2, the TigerGraph MultiGraph service, an add-on option, supports logical partitions of one unified master graph. Each partition is treated as an independent graph, with its own set of user privileges. 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?

See the ***GSQL Demo Examples*** for introductory examples. See ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data*** for the complete specifications. We also have a cheatsheet; go to doc.tigergraph.com.

- Which loading method should I use?

Created 01 May 2018

Beginning with v2.0, the TigerGraph platform introduces an extended syntax for defining and running loading jobs which 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.

The syntax for pre-v2.0 online loading and offline loading will be supported through v2.x, but they are deprecated.

- Why has Offline loading been deprecated?

Online loading is preferred. Online loading can do everything that offline loading can do, plus it has the following advantages:

1. Can run while other graph operations are in progress.
2. Uses multithreaded execution for faster performance.
3. Does not need to turn the GPE off, which saves time.
4. Its data source is specified at run time rather than at compile time.
5. Can add data to an existing graph.

What are the syntax differences between v2.0 loading and older online & offline loading?

Offline loading is deprecated and is now being emulated by online loading. Therefore, there is no performance difference.

Here are the main differences between the styles. Note that v2.0 is superficially similar to old offline loading, but it also supports run-time filenames. The actual behavior and performance of v2.0 loading is online loading with concurrency.

Syntax Detail	v2.0 Loading	Online	Offline
CREATE JOB statement	Keyword LOADING is used: `CREATE LOADING JOB job_name FOR GRAPH graph_name {...`	Keyword ONLINE_POST is used: `CREATE ONLINE_POST JOB job_name FOR GRAPH graph_name {...`	Keyword LOADING is used: `CREATE LOADING JOB job_name FOR GRAPH graph_name {...`
input source filename	LOAD statement must refer to a valid filepath or to a file variable. `LOAD "myFile1" TO ... LOAD fileVar2 TO ...` Filepath can specify one or more machines in a cluster. Optional	Filename appears in the USING clause in the RUN statement, so one JOB can handle only one	Filename appears in the LOAD statement, so one JOB can handle multiple input files if it has multiple LOAD

	DEFINE FILENAME statement to define a file variable. File variables can be set/overridden at run time: `RUN myJob USING f1="myFile1", f2="myFile2"`	input file: `RUN myJob USING FILENAME="myFile"`	statements: `LOAD "myFile1" TO ... LOAD "myFile2" TO ...`
HEADER, SEPARATOR, EOL, QUOTE parameters	These parameters appear at the end of the LOAD statement: `LOAD "myFile" TO ... USING SEPARATOR=",", QUOTE="double"`	These parameters appear at the end of the RUN statement: `RUN myJob USING FILENAME="myFile", SEPARATOR=",", QUOTE="double"`	These parameters appear at the end of the LOAD statement: `LOAD "myFile" TO ... USING SEPARATOR=",", QUOTE="double"`

How do I convert an offline loading job to an online loading job?

See the offline2online command, described in ***GSQL Language Reference, Part 1: Defining Graphs and Loading Data .***

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?*).

What is the format of an 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_DEFIN
}
```

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 a
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="t
```

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:adventu
102|"The Three-Body Problem"|FIC:fiction,SF:science fiction,CHN:Chines
```

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, flatt

  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="\r
```

Can the TigerGraph system load data from a streaming source?

Yes. Use online loading. Specifically, online loading 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 (opt
  ...,
  TO EDGE edge_typeN VALUES (attr_expr...) [WHERE conditions] [OPTION (opt
  [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
GroupByAccum	key: baseType, TUPLE, STRING COMPRESS accumulator: ListAccum, SetAccum, BagAccum, MapAccum

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 for
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

> Please check out the Troubleshooting Guide for more information regarding TigerGraph system log files, and what to look for.

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_root_dir>/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 capacit
```

In Limited Capacity mode, additional data may not be inserted. Data may be queried and deleted.

MultiGraph Overview

- 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 not a 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.)

MultiGraph service is available in the Enterprise Edition only.

Beginning with Version 1.2, 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 Enterprise Edition of the TigerGraph platform. MultiGraph enables several powerful use cases:

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.

While the system has the inherent capability of managing multiple graphs, the ability of users to create more than one graph may depend on your license key.

To support the new MultiGraph capabilities, a few changes to the previous specifications are necessary. These changes affect all users, even if only a single

graph is deployed. We advise all users to read the Concepts and Modified Specifications below.

There are also several New Commands .

Concepts

Graphs and Graph Domains

A graph is a 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 and edge types>)
```

NEW

- It is possible to define multiple graphs. The domains of two graphs may be completely separate, may overlap, or may coincide exactly.
- A TigerGraph instance with a basic license key can have one graph. A TigerGraph instance with a MultiGraph license key can create multiple graphs.
- A vertex type or edge type created by a superuser is a global type.
- A superuser can include a global vertex or edge type in one or more graphs. Global types can be shared among multiple graphs.
- The admin users or designer users for a particular graph can add local vertex types and edge types to their own graph.

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.

CHANGES

- 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.
- A user may be granted different roles on different graphs .
- A new top-level role is added: superuser . The superuser automatically has admin privilege on every graph, and has additional global privileges.

Setting a Working Graph

Previously, there was only one graph, and so all users were automatically able to use that graph.

NEW

- A user must set their working graph in order to access that graph.
- 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 have always required that the graph name be specified, even when there was support for only one graph. Now, it is clear that these definitions are graph-specific.

New and Modified Specifications

Modified Specifications

If you are a user of an earlier TigerGraph system (v1.1 or earlier), please note the following specifications have changed.

1. Set the working graph in GSQL: You must always set the working graph, either using the -g flag with the gsql command, or by using the USE GRAPH command.

2. RESTPP Endpoint changes: Endpoints which pertain to the graph data have been modified to include the name of the graph in the request URL.
   See RESTPP API User Guide .
3. User Authentication secrets and tokens: The way in which secrets and tokens are created and used has changed, in order to follow OAuth standards more closely.
   See Managing User Privileges and Authentication.
4. Changes to privileges of certain roles: If you had been using only the single default user with admin privilege, you will not notice any difference. That user has been promoted to superuser status. If you are making use of users with different roles, note the following changes in privileges:
   - A new top-level role, superuser , is defined. The superuser has admin privilege on all graphs, and is the only role who can create and modify shared vertex types, shared edge types, and graphs.
   - The architect role is renamed designer .
   - The public role is renamed observer .
   - The following commands are now shifted from admin and designer roles to the superuser role:
     - CREATE / DROP VERTEX|EDGE|GRAPH
     - CLEAR GRAPH STORE
     - DROP ALL
   - Newly created users no longer automatically have the observer role. They have no role until explicitly granted one.
5. In the CREATE VERTEX statement, the WITH STATS option "outdegree" is no longer available. "outdegree_by_edgetype" is still supported and is the default.

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

New Commands

The following commands are new. This section provides only a summary list. For full details and examples, see the main documentation for the relevant topics.

USE GRAPH <graph_name>

- For all users
- Sets the given graph as the user's working graph.

USE GLOBAL

- For superusers
- Must be set to have privilege to create and assign global vertex and edge types.

CREATE GLOBAL SCHEMA_CHANGE JOB

- For superusers

Troubleshooting Guide

Introduction

The Troubleshooting Guide teaches you how check on the status of your TigerGraph system, and when needed, how to 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 spac

$ free -g              (Make sure all servers have enough memory.)

$ tsar                 (Make sure there is no irregular memory usage on th

$ dmesg -T | tail      (Make sure there are no Out of Memory, or any other
```

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.

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
```

Setting automated collection of Logs

Customers can setup automated collection of logs as well as restart of services. This will ensure that all the logs related to critical process crash are collected in a timely fashion. This will minimize the need for access to customer environment to diagnose issues remotely. This will also avoid delayed restart of services.

Please follow the up to date instructions from TigerGraph Ecosystem Github Site ↗
Here are the two steps:

1. Users can save the following script as "~/.gsql/dump_log_auto_start.sh"

```
DUMP_LOG_AUTO_START_LOCK="/tmp/.dump_log_auto_start_lock"

if [[ -f ${DUMP_LOG_AUTO_START_LOCK} ]]; then
    exit 0
fi

if ~/.gium/gadmin status -v gpe gse | grep PROC | grep False; then
    touch ${DUMP_LOG_AUTO_START_LOCK}
    # dump 600 seconds logs before gpe/gse is down
    ~/.gium/gcollect -t 600 -o /tmp/dumplog-`date +"%Y%m%d-%T"` collec
    ~/.gium/gadmin start gpe gse
    rm ${DUMP_LOG_AUTO_START_LOCK}
fi
```

> **Note**: By default, when GPE or GSE goes down the script will dump logs for the preceding 10 minutes to the folder "/tmp/dumplog-$timestamp" (e.g., /tmp/dumplog-20200620-03:18:03) and restart GPE automatically. The time for log collection is configurable and user may change it accordingly.

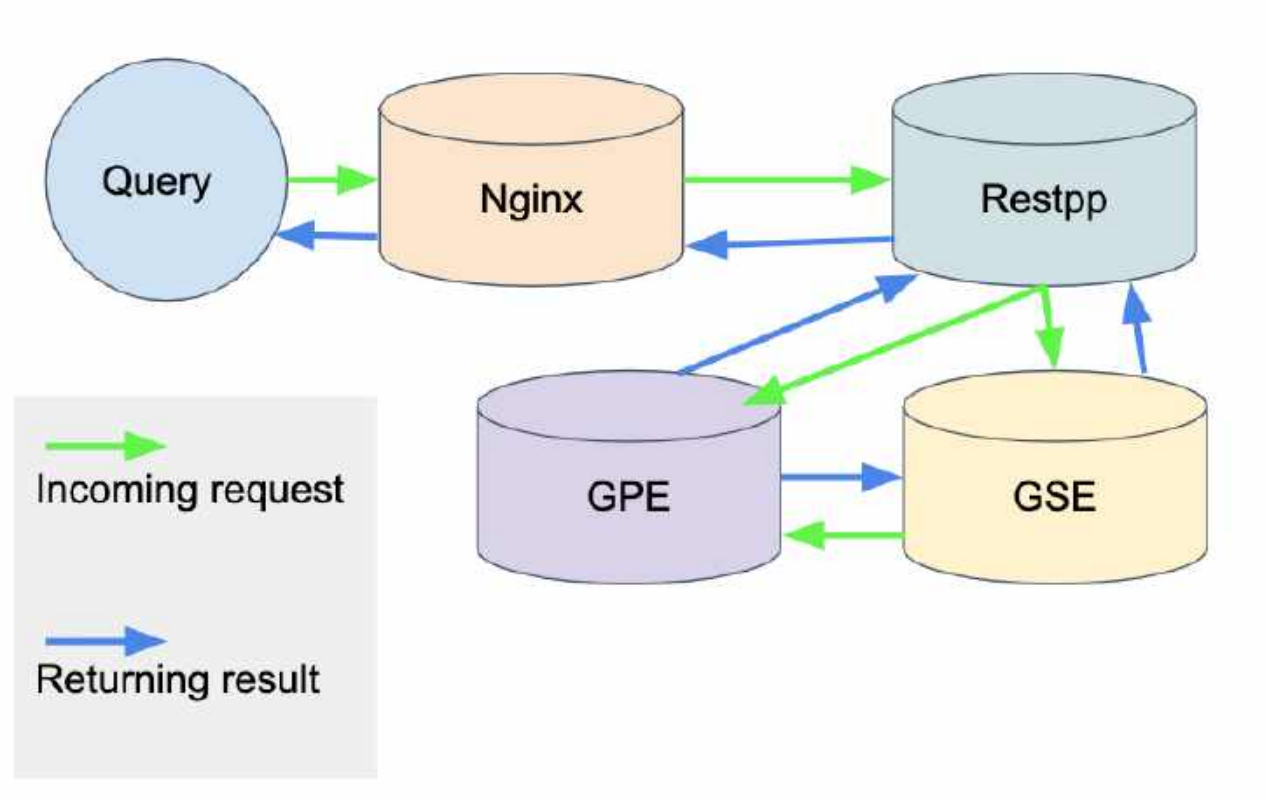
2. After configuring the script, add a cron job by running "crontab -e" and add the following line at the end:

```
* * * * * /home/tigergraph/.gsql/dump_log_auto_start.sh >/dev/null 2>&1
```

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.l
```

ⓘ You can click on the image below to expand it.

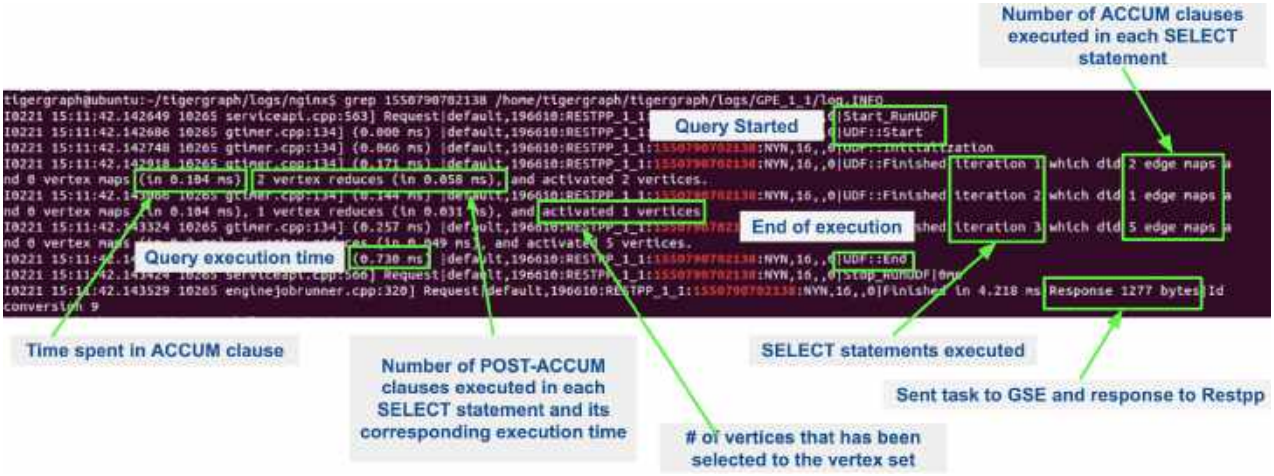
2. Nginx sends the request to Restpp.

```
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.lo
```

Other Useful Commands for Query Debugging

```
Check recently executed query:
$ grep UDF:: /home/tigergraph/tigergraph/logs/GPE_1_1/log.INFO | tail -n 5

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 | t


Grep logs from all servers:
$ grun all “grep UDF:: /home/tigergraph/tigergraph/logs/GPE_*/log.INFO | t
```

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 Li command: `free -g` to check available memory and swap status. To combat this, you can either 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 ??? , this 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 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 still running, it is just 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 a 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.

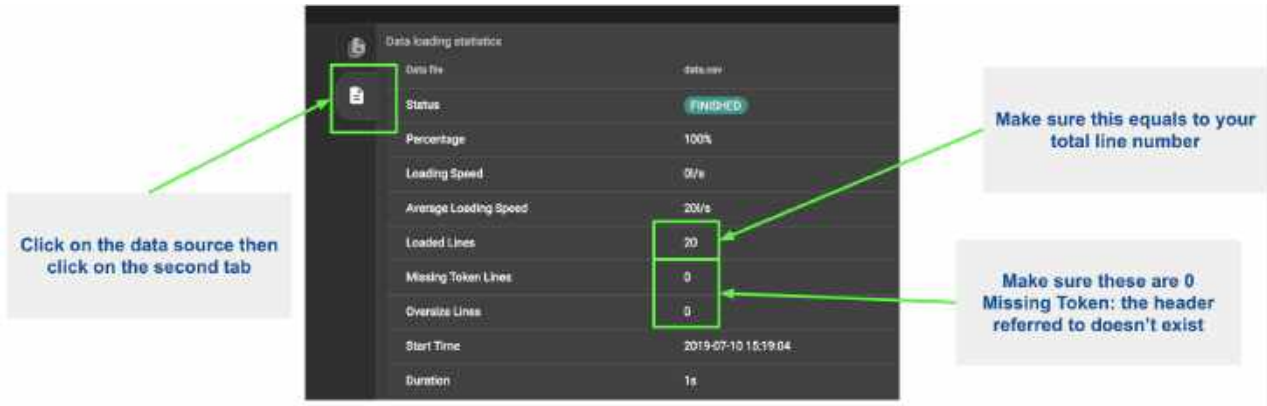
⚠ If you have a c++ user-defined function error, your query will fail to install, even if it does not utilize the UDF.

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.)

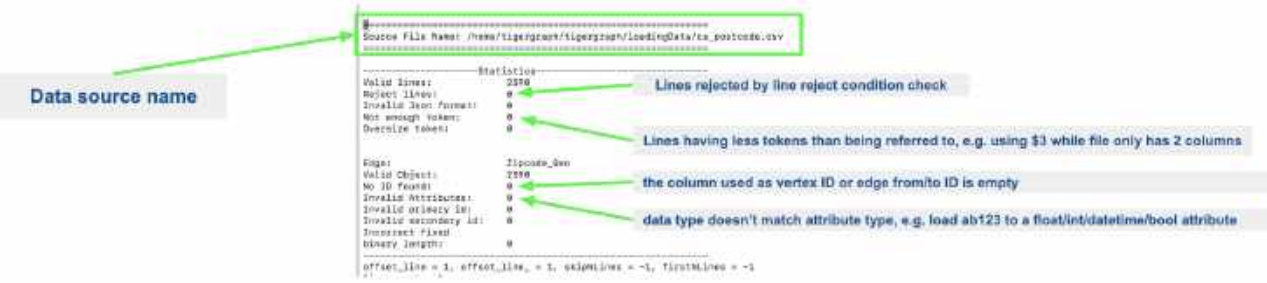


Command Line

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 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.

Further Debugging

If after taking these actions you cannot solve the issue, please reach out to support@tigergraph.com to request assistance.

System Administration

TigerGraph Administrators Guide

Hardware and Software Requirements

Actual hardware requirements will vary based on your data size, workload and features you choose to install.

Hardware Requirements

Component	Minimum	Recommended
CPU	1.8 GHz (64-bit processor) or faster multi-core	Dual-socket multi-core, 2.0 GHz (64-bit processors) or faster
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 depend on data size. 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.
- The platform works excellently as a single node. For high availability or scaling, a multi-node configuration is possible.

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. When a range of versions is given, it has been tested on the two endpoints, oldest and newest. 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.

	On-Premises hosting	Java JDK version	GCC version (C/C++)
RedHat 6.5 to 6.9 (x64)	Yes	1.8.0_141	4.8.2
RedHat 7.0 to 7.4 (x64)	Yes	1.8.0_141	4.8.2
Centos 6.5 to 6.9 (x64)	Yes	1.8.0_141	4.8.2
Centos 7.0 to 7.4 (x64)	Yes	1.8.0_141	4.8.2
Ubuntu 14.04 LTS Ubuntu 16.04 LTS Ubuntu 18.04 LTS (x64)	Yes	1.8.0_141	4.8.4
Debian 8 (jessie)	Yes	1.8.0_141	4.8.4

Additionally, we offer Amazon Machine Images (AMI) to run on Amazon EC2. Please contact us regarding recommended configurations.

Prerequisite Software

Utilities

Before offline installation, the TigerGraph system needs a few basic software packages to be present.

1. tar, to extract files from the offline package
2. curl, an alternative way to send query request to TigerGraph
3. crontab, a basic OS software module which TigerGraph relies on
4. uuidgen, a tool to creates an universally unique identifier of the server
5. ip, to configure the network
6. ssh/sshd, to connect to the server
7. more, a tool to display the License Agreement
8. netstat, a basic OS tool to check the network status
9. semanage, to manage SELinux context of ssh
10. 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 coreutils

# Ubuntu or Debian:
sudo apt install tar curl cron iproute util-linux uuid-runtime net-tools c
```

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

Browser	Chrome	Safari	Firefox	Opera
Supported version	54.0+	11.1+	59.0+	52.0+

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:

Installation and Configuration

Installation, Cluster Configuration and Scale-out, License Activation

Getting Super Powers

Becoming a super hero is a fairly straight forward process:

```
$ give me super-powers
```

> Super-powers are granted randomly so please submit an issue if you're not happy with yours.

Once you're strong enough, save the world:

hello.sh

```
# Ain't no code for that yet, sorry
echo 'You got to trust me on this, I saved the world'
```

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. Please use the Table of Contents to go to the appropriate section of this guide.

- Single Node Installation
- Cluster Installation and Configuration

> If you are installing the **Developer Edition**, you can also install a Docker image or a virtual machine (VirtualBox) image. Your welcome email message will direct you to the appropriate resources.

Preparation

> This section is for **New Installations**. If you are updating from a previous version of the TigerGraph platform, first read the section below on Upgrading an Existing Installation .

Before you can install the TigerGraph system, you need the following:

1. One or more servers that meets the minimum *Hardware and Software Requirements* with regard to operating system, memory and hard disk space, as well as enough memory and storage to store your graph data.
2. sudo or root privilege.
3. A license key provided by TigerGraph (not applicable to Developer Edition)
4. A TigerGraph system package .
5. If your package is a *tar.gz file, you may need to install some software prerequisites.

> Use a BASH shell, otherwise there may be installation issues.

Obtaining a TigerGraph Package

If you do not yet have a TigerGraph system package, you can request one at www.tigergraph.com/download/ ↗.

Software Prerequisites for *.tar.gz Packages

If your package is a *tar.gz file, you also need to insure your machine has the following software prerequisites.

1. Pre-install these basic Linux utilities on your server, if necessary:
   - tar
   - curl
   - ip
   - more
   - uuidgen
   - crontab
   - ssh/sshd
   - netstat
   - semanage
2. I f you are installing a cluster, you also need the following:
   - ntpd
   - iptables/firewalld
3. If you will use the password login method (P method) instead of ssh key login method (K method) to install the TigerGraph platform, you will also need the following:
   - sshpass

Single Node Installation

The name of your package may vary, depending on the product edition (e.g., developer or enterprise) and the version (e.g., 2.0.1). For the examples here, we will assume the name is tigergraph-x.y.z.tar.gz. Substitute the name of your actual package file.

1. Extract the package:

Example: extract for <version> = x.y.z

```
tar -xzf tigergraph-x.y.z.tar.gz
```

2. A folder named **`tigergraph-<version>-offline (or tigergraph-<version>-developer)`** will be created. Change into this folder. To Install with default settings, run the install.sh script with commands:

Example: Default single-server installation for <version> = 2.0.0

```
# to install enterprise edition
cd tigergraph-*/
sudo ./install.sh -s

# to install developer edition
cd tigergraph-*/
sudo ./install.sh
```

The installer will ask you a few questions:

- Do you agree to the License Terms and Conditions?
- What is your license key? (not applicable to Developer Edition)
- Do you want to use the default TigerGraph user name or select/create your own?
- Do you want to use the default TigerGraph user password or create your own?
- Do you want to use the default installation folder or select/create your own?

To see what are the default settings, and to see how customize the installation, read the Installation Options section below.

⚠ Some 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.

3. The installer concludes by using 'su' to switch to the tigergraph user account. To confirm correct operation:

    1. Try the command `gadmin status`

    If the system installed correctly, the command should report that **zk** , **kafka** , **dict, ts3, nginx, gsql,** and **Visualization** are up and ready. Since there is no graph data loaded yet, **gse** , **gpe** , and **restpp** are not initialized.

    2. Try the command `gsql --version`

4. Basic installation is now finished! Please see Post-Installation Notes below.

Installation Options

The following default settings will be applied if no parameters specified:

- The installer will create a user called **tigergraph** , with password **tigergraph** .
- 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:

```
# Installation options of enterprise edition
Usage:
./install.sh (-s|-c) [-u <user>] [-p <password>] [-r <tigergraph_root_dir>
./install.sh -c -n
./install.sh -h
Options:
  -h  --  show the help
  -u  --  TigerGraph user [default: tigergraph]
  -p  --  TigerGraph password [default: tigergraph]
  -r  --  TigerGraph.Root.Dir [default: <tigergraph_user_home>/tigergraph]
  -l  --  TigerGraph license key
  -s  --  Single server option: install the tigergraph platform on a sing]
  -c  --  Cluster option: install the tigergraph platform on a cluster
  -n  --  Non-interactive option: suppress prompts, and continue installat

[NOTE    ]: Using option '-c -n' together will non-interactively install t
on a cluster with all configurations from config file "cluster_config.jso
In this case, the config file should be modified before installation, and

# Installation options of developer edition
Usage:
./install.sh [-u <user>] [-p <password>] [-r <tigergraph_root_dir>]
./install.sh -h
Options:
  -h  --  show the help
  -u  --  TigerGraph user [default: tigergraph]
  -p  --  TigerGraph password [default: tigergraph]
  -r  --  TigerGraph.Root.Dir [default: <tigergraph_user_home>/tigergraph]
  -n  --  Non-interactive option: suppress prompts, and continue installat
```

Cluster Installation and Configuration (Enterprise Edition Only)

TigerGraph cluster configuration enables the graph database to be partitioned and distributed across multiple server nodes in a local network (not available in the Developer Edition). The cluster can either be a physical cluster or a network virtual cluster from a cloud service such as Amazon EC2 or Microsoft Azure.

⚠ 1. The installation of TigerGraph 2.x has been validated on Amazon EC2 and Microsoft Azure and on a physical on-premises cluster. For Amazon EC2, please

make sure all tcp ports are open among all cluster nodes, otherwise service may not start.

2. In TigerGraph 2.x, the installation machine can be within or outside the cluster. If outside the cluster, the installation machine should be a Linux machine.
3. Currently, every machine in the cluster must have a sudo user with the same username and password or SSH key .
4. To install a high-availability cluster (with at least 3 nodes), please set HA.option to be "true" for non-interactive installation or answer "yes" to HA question for interactive installation.
5. Do not run the cluster installation script in sudo mode.

During cluster configuration, the user provides the following information:

- The IP address for each server node, e.g., 172.30.3.2
- The login credentials for the nodes.

Cluster Installation Overview

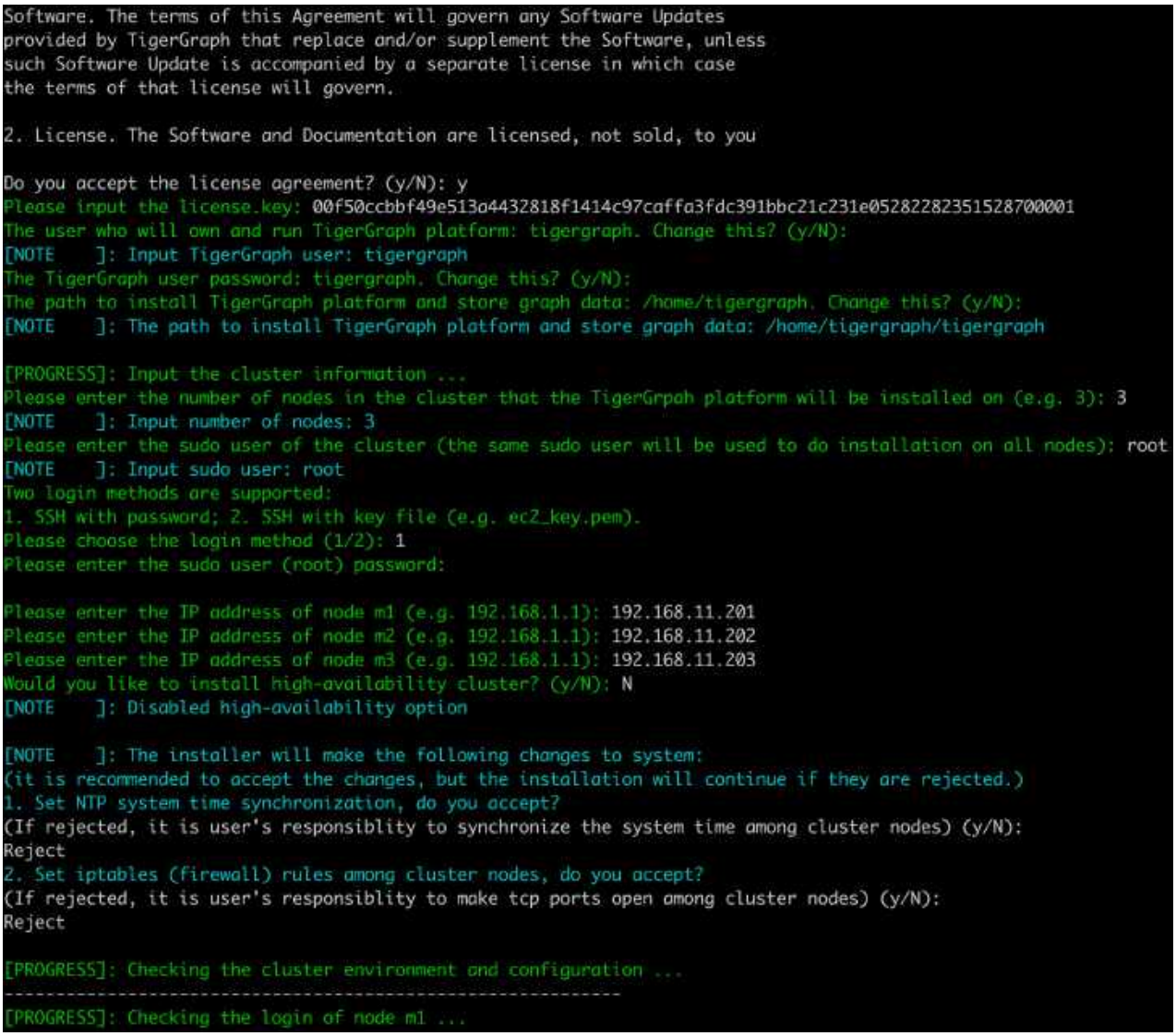
Cluster installation begins by the user downloading the TigerGraph software package to any Linux machine in the cluster or with access to the cluster nodes (see notes above). When the user runs the installation script with the cluster option, it will either prompt the user for cluster configuration information described above, or if the user requests non-interactive installation, it will read the configuration information from a file `cluster_config.json` located in the same folder with the platform package. The installer then proceeds to install the product on each of the cluster nodes and to configure the cluster.

The two installation methods, interactive and non-interactive, are described below.

Interactive Mode Installation

In interactive mode, the installer will first ask the same basic questions it asks for single-node installation. It will then ask how many machines are in your cluster. Then it will prompt for the IP addresses of the machines, assigning each machine an alias m1, m2, m3, etc. Next it will ask for sudo user name and credentials

information. Last, it will ask the user if they accept some changes to the system. (See non-interactive mode installation below for details about user credentials.) A screenshot of interactive installation is shown below.



Non-Interactive Mode Installation

For non-interactive mode installation, the user must put all the settings into the file `cluster_config.json` before running the installer. This file is in the folder with your install.sh file and other TigerGraph package files.

the two key parameters to set are the following:

1. **nodes.ip**
   Each machine in the cluster is defined as a key:value pair, where the key is a

machine alias m1, m2, m3, etc. **NOTE: If you chose names other than m1, m2, etc., be sure to list them in alphanumeric order in the config file. The first machine ("m1") has a special role in some cases.** Use as many key:value pairs as you need, placing the public IP addresses next to each key. The installer will auto detect the local IP addresses and use them to configure the system. If the installer detects more than one local IP address, it will ask the user to select one for configuration.

2. **nodes.login**
   Two login methods are supported:
   - SSH with password
   - SSH with key file

   For SSH with password, you must input the sudo/root user and its password. For SSH with key file, you may specify the AWS ec2 key file or other key file. If nothing provided, the installer will use default ssh key file such as `~/.ssh/id_rsa`.
3. **HA.option**
   If enable.HA is set to "true", then the system will be configured for a replication factor of 2. For example, if your cluster has 6 machines, 3 will be used for one copy of the data, and 3 will be used for a replica copy of the data. More advanced configuration is possible after initial setup. See Configuring a High Availability Cluster v2.1

Below is a sample **`cluster_config.json`** file.

⚠ **The node names (e.g., m1, m2, etc.) MUST be given in alphanumeric order, because the first machine has a special role in some situations. In our documentation we will refer to this machine as m1.**

cluster_config.json example

```
{
  "tigergraph.user.name": "tigergraph",
  "tigergraph.user.password": "tigergraph",
  "tigergraph.root.dir": "/home/tigergraph/tigergraph",
  "license.key": "91583b19abf850cee381168e0e0cd41fcaceba2d734cd3a9e6f5
  "nodes.ip": {
    "m1": "172.30.3.2",
    "m2": "172.30.3.3",
    "m3": "172.30.3.4",
    "m4": "172.30.3.5"
  },
  "nodes.login": {
    "supported.methods (this is a comment)": "P. SSH with password; K.
    "notes (this is a comment)": "All nodes must use the same sudo use
    "chosen.method": "K",
    "P": {
        "sudo.user.name": "sudoUserName",
        "sudo.user.password": "sudoUserPassword"
    },
    "K": {
      "sudo.user.name": "centos",
      "ssh.key.file": "/home/centos/.ssh/gsql_east.pem"
    }
  },
  "HA.option": {
    "Notes of HA.option (this is a comment)": "option to install high-
    "enable.HA": "false"
  }
}
```

Cluster Installation Commands

Advanced Configuration

Sometimes you may want further control over configuration details, such as replication factor of individual components, security settings, and others. You may also want to install a new TigerGraph system to match your existing TigerGraph system's setup. TigerGraph supports advanced configuration with the `-a` option. This option can be used in either interactive mode or non-interactive mode.

> ⓘ Advanced configuration will override the default configuration, and the related configuration in **`cluster_config.json`**.

First, create a configuration file named **`adv_config.cfg`**. You can manually create this file, or if you have an existing TigerGraph system, you can generate a file representing its configuration, with the following command: `gadmin --dump-config |grep replicas >> adv_config.cfg)`

If you manually create it, make sure it's a valid YAML file.

For example, the `adv_config.cfg` file below sets up TigerGraph as a 3-node cluster with HA replications factor of 3.

```
dictserver.servers: m1,m2,m3
gpe.servers: m1,m2,m3
gse.servers: m1,m2,m3
kafka-loader.servers: m1,m2,m3
kafka.servers: m1,m2,m3
restpp-loader.servers: m1,m2,m3
restpp.servers: m1,m2,m3
zk.servers: m1,m2,m3
gpe.replicas: 3
gse.replicas: 3
kafka.num.replicas: 3
```

Second, in the installation command, add the `-a` option. Once the installation is done, verify the system has the configuration as specified.

Cluster Installation Commands

> ⚠ **Do not** run the cluster installation script with sudo permission.

After you have planned out your cluster configuration, you are ready to run the installer.

1. Extract the package.

Example: extract for <version> = 2.0.0

```
tar -xzf tigergraph-x.y.z.tar.gz
```

2. A folder named `tigergraph-<version>-offline` will be created. Change into this folder. To run cluster installation in interactive mode, use the -c option:

Example: Installation with interactive cluster configuration for <version> = 2.0.0

```
cd tigergraph-*/

## Method 1. Interactive mode
./install.sh -c
```

To run cluster installation in non-interactive mode, using the settings in the `cluster_config.json` file, use the -c and -n (or merged -cn) options:

Example: Installation with non-interactive cluster configuration <version> = 2.0.0

```
cd tigergraph-*/

## Method 2. non-interactive
 # step 1: modify the config file "cluster_config.json"
 # step 2:
 ./install.sh -cn
```

3. The installer concludes by prompting the user to login to node m1 of the cluster and use 'su' to switch to the tigergraph user account.
To confirm correct operation:

1. Try the command `gadmin status` from any machine in the cluster.

   If the system installed correctly, the command should report that **zk** , **kafka** , **dict, nginx, gsql,** and **Visualization** are up and ready. Since there is no graph data loaded yet, **gse** , **gpe** , and **restpp** are not
2. Try the command `gsql --version`

The `gsql` command must be run on node m1 of the cluster because the gsql server is installed on m1 only.

4. Basic installation is now finished! Please see Post-Installation Notes below.

Post-Installation Notes

Change Your Password

If you installed with the default password, we recommend that you change it now.

Additional Customization

To perform additional customization, run `gadmin --configure` (must be on node m1 if it is cluster), followed by `gadmin config-apply`. The ' `gadmin config-apply` ' command must be run on node m1 if it is cluster, since only node m1 contians *pkg_pool*resources. If you configured one or more items of gpe.servers, gse.servers, restpp.servers, kafka.servers, zk.servers, dictserver.servers, gpe.replicas, or gse.replicas, you must reinstall the package by running command `gadmin pkg-install reset` on node m1.

see the appropriate sections of the ***TigerGraph System Administrators Guide v2.1***.

Learning To Use TigerGraph

If you are a first-time user:

- See our GSQL language tutorial for first-timer users: ***GSQL 101***
- Start designing, using our visual interface. see the ***TigerGraph GraphStudio UI Guide***.
- To see more GSQL examples, see ***GSQL Demo Examples***.
- To get answers to common questions, see ***TigerGraph Knowledge Base and FAQs***.

Upgrading an Existing Installation

> ⚠ Developer Edition upgrade is not supported
>
> The Developer Edition is not designed for upgrade from one version to another
> It is not possible to upgrade a Developer Edition installation to Enterprise Edition.

> ⓘ If you have written User-Defined Functions for your queries, be sure to make a backup of these files :
> <tigergraph.root.dir>/dev/gdk/gsql/src/QueryUdf/ExprFunctions.hpp
> <tigergraph.root.dir>/dev/gdk/gsql/src/QueryUdf/ExprUtil.hpp

If your specific versions are not listed below, please upgrade by :

1. Download the latest version of TigerGraph to your system.
2. Extract the tarball.
3. Run the TigerGraph.bin file that was extracted from the tarball.

Updating from v2.1.7 to v2.2.x

These steps are assuming that v2.1.7 is installed. To upgrade to v2.2 from a version older than v2.1.7 , please upgrade to v2.1.7 first. If the tigergraph username and password have been changed, please have them ready as you will need them in order to update the system.

1. Download tigergraph-2.2.x-offline.tar.gz with user “tigergraph” and extract the tarball file.
2. Download the post_upgrade.sh script that is attached here ↗.
3. Run tigergraph.bin under the same folder to upgrade to 2.2.x
4. Run the post-upgrade script that was downloaded in step 2 : **post_upgrade.sh** -u <sudoUser> [-P <sudoPass> | -K <sshKey>] -p <tigergraphUserPass>

Updating from v2.0 to v2.1

v2.0 can be upgraded to v2.1 Enterprise Edition. The data store format and GSQL language scripts in v2.0 are forward compatible to v2.1.

Upgrading from v1.x to v2.x

The data store format between 1.x and 2.x for single servers is forward compatible but not backward compatible. For a single server platform, users can upgrade from 1.x to 2.x without reloading data or recreating the graph schema. Some details of the GSQL language have changed, so some loading jobs and queries will need to be revised and reinstalled.

For a cluster configuration, direct upgrade from 1.x to 2.x is not supported at this time. Users interested in migrating from 1.x to 2.x need to export their data and metadata, install v2.x, and then reload data and metadata, with some small modifications. Please contact support@tigergraph.com ↗for assistance.

Please consult the Release Notes for all the versions between your current version and your target version (e.g., v2.1) to see a summary of specification changes. Contain support@tigergraph.com ↗for assistance.

Workflow for Direct Upgrade

1. Verify that your data store is compatible and is eligible for direct update / upgrade.
2. Review the specification changes and how they may affect your applications (loading jobs and queries).
3. Stop issuing new commands to your TigerGraph system and allow any operations to complete.
4. (Recommended) Backup your data, as a precaution.
5. Follow the procedure at the beginning of this document for installing a new system. The installer will automatically shut down your system and start it again.

⚠ Be sure to specify the same username as your current installation. Otherwise, if you use a different user name, it will be treated as a new installation, with an empty graph.

1. Pay attention to output messages during the installation process which may alert you to additional tasks or checks you should perform.
2. Run the command **gsql** to start the GSQL shell. The first time after an update, gsql performs two important operations:
    a. Copies your catalog from your old installation to the new installation .
    b. Compares the files in the backup /dev_<datetime>/gdk/gsql/src folder to the new /dev/gdk/gsql/src folder. Pay attention to any files residing in the old folder but not in the new folder. Review them and copy them to the new folder if appropriate. See the example below.
3. Revise and reinstall loading jobs, user-defined functions, and queries as needed.

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 loading balancing when all components are operational, as well as automatic failover in the event of a service disruption. One TigerGraph server consists of several components (e.g., GSE, GPE, RESTPP). The default HA configuration has a replication factor of 2, meaning that a fully-functioning system maintains two copies of the data, stored on separate machines. In advanced HA setup, users can set a higher replication factor.

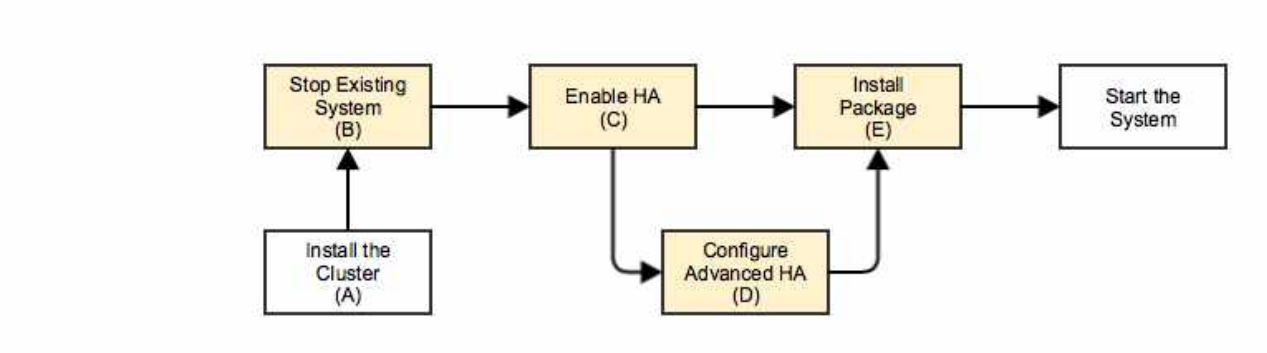
System Requirements

- An HA cluster needs at least 3 server machines . Machines can be physical or virtual. This is true even the system only has one graph partition.
- For a distributed system with N partitions (where N > 1), the system must have at least 2N machines.
- The same version of the TigerGraph software package is installed on each machine.

Limitations

1. HA configuration should be done immediately after system installation and before deploying the system for database use.
2. To convert a non-HA system to an HA system, the current version of TigerGraph requires that all the data and metadata be cleared, and all TigerGraph services be stopped. This limitation will be removed in a future release.

Workflow



Starting from version 2.1, configuring a HA cluster is integrated into platform installation, please check the document TigerGraph Platform Installation Guide for detail.

(A) Install TigerGraph

Follow the instructions in the document TigerGraph Platform Installation Guide to install the TigerGraph system in your cluster.

> In the instructions below, all the commands need to be run as the *tigergraph* OS user, on the machine designated "m1" during the cluster installation.

(B) Stop the TigerGraph Service

Be sure you are logged in as the *tigergraph* OS user on machine "m1". Before setting up HA or changing HA configuration, the current TigerGraph system must be fully stopped. If the system has any graph data, clear out the data (e.g., with "gsql DROP ALL").

Stopping all TigerGraph services

```
gadmin stop ts3 -fy
gadmin stop all -fy
gadmin stop admin -fy
```

(C) Enable HA

After the cluster installation, create an HA configuration using the following command:

```
gadmin --enable ha
```

This command will automatically generate a configuration for a distributed (partitioned) database with an HA system replication factor of 2. Some individual components may have a higher replication factor .

Sample output:

Successful HA configuration

```
tigergraph@m1$ gadmin --enable ha
[FAB    ][m3,m2] mkdir -p ~/.gium
[FAB    ][m3,m2] scp -r -P 22 ~/.gium ~/
[FAB    ][m3,m2] mkdir -p ~/.gsql
[FAB    ][m3,m2] scp -r -P 22 ~/.gsql ~/
[FAB    ][m3,m2] mkdir -p ~/.venv
[FAB    ][m3,m2] scp -r -P 22 ~/.venv ~/
[FAB    ][m3,m2] cd ~/.gium; ./add_to_path.sh
[RUN    ] /home/tigergraph/.gsql/gpe_auto_start_add2cron.sh
[FAB    ][m3,m2] mkdir -p /home/tigergraph/.gsql/
[FAB    ][m3,m2] scp -r -P 22 /home/tigergraph/.gsql/all_log_cleanup /home
[FAB    ][m3,m2] mkdir -p /home/tigergraph/.gsql/
[FAB    ][m3,m2] scp -r -P 22 /home/tigergraph/.gsql/all_log_cleanup_add2c
[FAB    ][m1,m3,m2] /home/tigergraph/.gsql/all_log_cleanup_add2cron.sh
[FAB    ][m1,m3,m2] rm -rf /home/tigergraph/tigergraph_coredump
[FAB    ][m1,m3,m2] mkdir -p /home/tigergraph/tigergraph/logs/coredump
[FAB    ][m1,m3,m2] ln -s /home/tigergraph/tigergraph/logs/coredump /home/
```

If the HA configuration fails, e.g, if the cluster doesn't satisfy the HA requirements, then the command will stop running with a warning.

HA configuration failure

```
tigergraph@m1$ gadmin --enable ha
Detect config change. Please run 'gadmin config-apply' to apply.
ERROR:root: To enable HA configuration, you need at least 3 machines.
Enable HA configuration failed.
```

(D) [Optional] Configure Advanced HA

In this optional additional step, advanced users can run several "gadmin --set" commands to control the replication factor and manually specify the host machine for each TigerGraph component. The table below shows the recommended settings for each component. See the later example section for different configuration cases.

Component	Configuration Key	Suggested Number of Hosts	Suggested Number of Replicas
ZooKeeper	zk.servers	3 or 5	-
Dictionary Server	dictserver.servers	3 or 5	-
Kafka	kafka.servers	same as GPE	same as GPE
	kafka.num.replicas	2 or 3	2 or 3
GSE	gse.servers	every host	every host
	gse.replicas	2	2
GPE	gpe.servers	every host	every host
	gpe.replicas	2	2
REST	restpp.servers	every host	every host

Example: There is a 3-machine cluster m1, m2 and m3. Kafka, GPE, GSE and RESTPP are all on m1 and m2, with replication factor 2. This is a non-distributed graph HA setup.

Example: 3-machine non-distributed HA cluster

```
gadmin --set zk.servers m1,m2,m3
gadmin --set dictserver.servers m1,m2,m3
gadmin --set dictserver.base_ports 17797,17797,17797
gadmin --set kafka.servers m1,m2
gadmin --set kafka.num.replicas 2
gadmin --set gse.replicas 2
gadmin --set gpe.replicas 2
gadmin --set gse.servers m1,m2
gadmin --set gpe.servers m1,m2
gadmin --set restpp.servers m1,m2
```

(E) Install Package

Once the HA configuration is done, proceed to install the package from the first machine (named “m1” in the cluster installation configuration).

```
gadmin pkg-install reset -fy
```

Examples

The table below shows how to setup for the common setups. Note if convert the system from another configuration, must stop the old TigerGraph system first.

System Goal	Cluster Configuration (number of servers in cluster is X)	How to A,B,C, etc. refer to the Steps in the section above.
Non-distributed graph with HA	Each server machine holds the complete graph.	• For both initial installation and reconfiguration, (A) → B → C → D → E. While in D, set all replicas to *X*, e.g, gpe.replicas = *X* gse.replicas = *X* restpp.replicas = *X* ...

		• Note: (A) means A is needed only in initial installation
Distributed graph without HA	Graph is partitioned among all the cluster servers.	• Note: no HA is equivalent to replication factor 1 • For initial installation, skip B, C, D and E. • For reconfiguration, B → C → D → E. While in D, set all replicas to 1, e.g., gpe.replicas = 1 gse.replicas = 1 restpp.replicas = 1 ...
Distributed graph with HA	Graph is partitioned with replica factor N. Number of partitions Y equals X/N.	• For both initial installation and reconfiguration, (A) → B → C → D → E. While in D, set all replicas to N , *e.g.,* gpe.replicas = N gse.replicas = N ... • Note: (A) means A is needed only in initial installation

Cluster Scale-Out

Adding machines to a TigerGraph cluster, for distributed data and/or HA

Introduction

Cluster expansion allows the user to add new machine nodes to an existing cluster and to redistribute data, while the entire system is offline.

Prerequisites

1. The current TigerGraph system must be installed in cluster mode, not single-node mode.
2. The total graph data storage space for the expanded cluster should be at least 3 times as large as the current Gstore disk usage.
   a. During the expansion process, a backup copy of all the graph data files is created, plus additional working space is needed.
   b. To check your existing gstore disk space:
3. The new nodes are available.

Configure GBAR

Cluster Expansion Workflow

The GBAR utility is used for cluster expansion. If this is your first time using GBAR, you must first run `gbar config`. See the Backup and Restore guide. For a large system one of the key parameters is `backup_core_timeout`. The default value is 5 hours. The config script gives guidance on estimating an appropriate value.

Set Up Environment in New Nodes

From the command line, switch to the `<tigergraph_root_dir>/pkg_pool/syspre_pkg` directory under the TigerGraph root directory (`~/tigergraph/pkg_pool/syspre_pkg` by default). In this directory, a utility script `set_syspre.sh` is used to setup environment:

Run `./set_syspre.sh -h` to see the usage:

```
./set_syspre.sh  -h

Usage:
./set_syspre.sh -i <IP address/host name> -u <sudo user> (-P <password> |
./set_syspre.sh -h
Options:
  -h  --  show the help
  -i  --  the IP address of the new machine
  -u  --  sudo user [default: $USER]
  -P  --  sudo user password [default: empty]
  -K  --  sudo user ssh key [default: empty]
  -p  --  tigergraph user password [default: tigergraph]

[NOTE    ]: This script must be run under tigergraph user.
```

For example, to set up the environment on a new node `192.168.1.6` with sudo user called `ubuntu` and login key `ubuntu_rsa`, run the following command:

Set Environment in New Nodes

```
./set_syspre.sh -i 192.168.1.6 -u ubuntu -K ~/.ssh/ubuntu_rsa
```

Firewall check

The firewall configuration on new node must be the same as that on existing nodes. Otherwise, the TigerGraph instances on new nodes may not work properly.

For users using TigerGraph 2.2 with Ubuntu, you must comment out the following block at the beginning of `.bashrc` in the tigergraph user's home directory, on every node.

```
# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac
```

When done, the environment including system-prerequisites and ssh keys for the TigerGraph system will be set up on the new nodes.

Add New Nodes to Cluster

To expand the cluster, run `gbar expand` with a list of new nodes in the following format:

```
gbar expand <node_alias_1>:<ip_1>,<node_alias_1>:<ip_2>,...,<node_alias_n>
```

For example, the following command adds two nodes to the cluster:

```
gbar expand m6:192.168.1.6,m7:192.168.1.7
```

The command above will redistribute the data on all nodes including m6 and m7, so that each node has about the same amount of data.

GBAR will run the following checks for each new node:

1. The number of new nodes must be an integer multiple of **max**(gpe.replicas, gse.replicas).
2. Each new node alias must be a valid identifier.
3. Each new node's IP address must be accessible via ssh from the node where `gbar expand` is being run.

Error Handling

If the system does not have a schema or data, it will report a data integrity check error. You may ignore this warning.

Advanced Expansion Mode

Advanced expansion configuration options are possible. Contact TigerGraph Support for guidance.

Should any errors occur, GBAR will roll back to the state before node expansion started. As a failsafe, a backup copy of the data is kept, until expansion either succeeds or finishes rollback.

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 administration 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 (so-called 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 which 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

Note: Before beginning the license activation process, the TigerGraph package must be installed on each server, and the TigerGraph system must be configured with gadmin.

- Collect the fingerprint of the whole TigerGraph system using the command tg_ lic_seed , which can be executed on any machine in the system. The command tg_lic_seed packs all the collected data into a local file (named tigergraph_seed). When tg_lic_seed has completed successfully, it outputs the path of the collected data to the console.

Collect Fingerprint of TigerGraph System

```
$ tg_lic_seed
seed file is ready at /home/tigergraph/tigergraph/tigergraph_seed
```

- Send the tigergraph_seed file to TigerGraph , either through our license activation web portal (preferred) or by email to license@tigergraph.com. ↗If using email, please include the following information:
  - Company/Organization name
  - Contract number. If you do not know you contract number, please contact your sales representative or sales@tigergraph.com. ↗
- If the contract and license seed are in good order, a new license key file will be certificated and sent back to you.
- Copy the license key file to a directory on the TigerGraph system where the TigerGraph linux user has r ead permission .
- To install the license key, run command tg_ lic_install , specifying the path to the license key file.

Install License

```
$ tg_lic_install
Usage: tg_lic_install <license_path>
```

If installation is completed successfully, the message "install license successfully" will be displayed in the console. Otherwise, another message "failed to install license" will be displayed.

Checking License Information

After a license key has been installed successfully on a TigerGraph system, the information of the installed license is available 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 gadmin status -v graph and collect the VertexCount, EdgeCount, and Partition Size. Compare this information to the limits established for your license.

Checking graph limits

```
~/tigergraph/loadingData$ gadmin status -v graph
verbose is ON
[Warning] License will expire in 11 days
=== graph ===
[m1     ][GRAPH][MSG ] Graph was loaded (/home/tigergraph/tigergraph/gsto
[m1     ][GRAPH][INIT] True
[INFO   ][GRAPH][MSG ] Above vertex and edge counts are for internal use w
[SUMMARY][GRAPH] graph is ready
```

The output may include a warning message such such as the following:

```
[Warning] License limit exceeded. The system is running in limited capacit
```

User access management

User Privileges and Authentication, LDAP, Single Sign-on

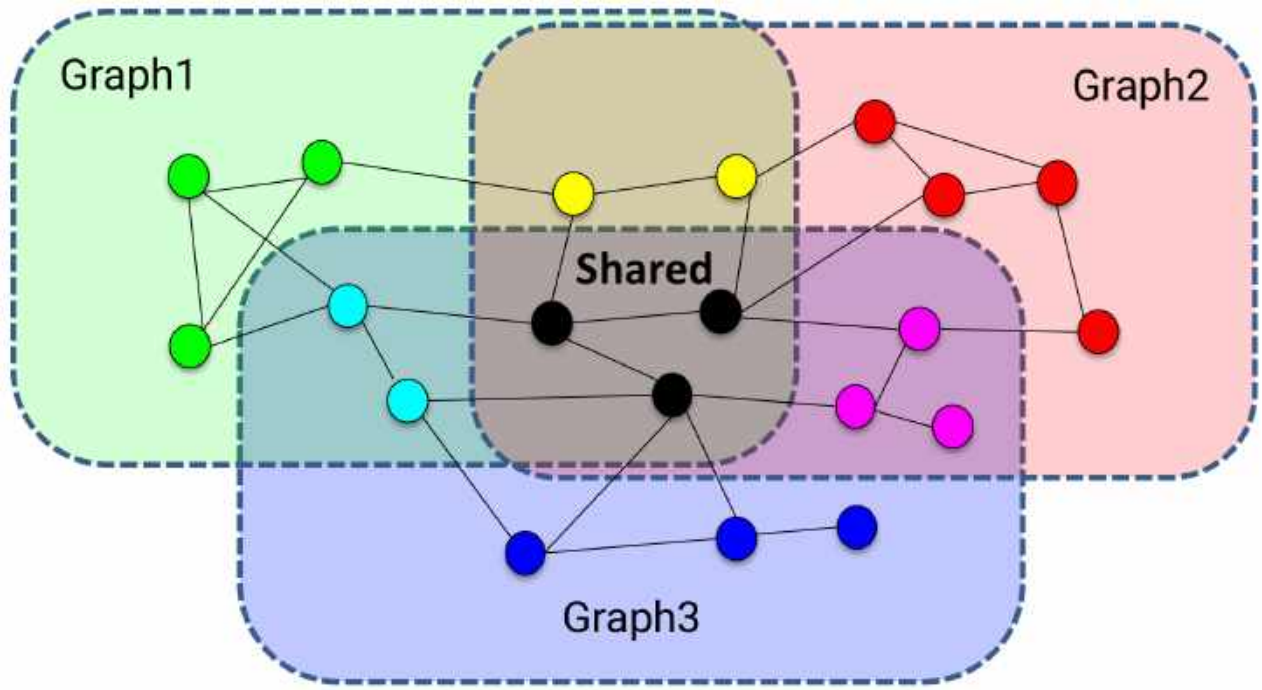
User Privileges and Authentication

Creation and management of multiple users and roles is available in the Enterprise Edition only.

Overview

The TigerGraph platform provides a complete and robust feature set to manage and control user privilege and authentication of GSS operations:

- Creation and management of multiple TigerGraph users
- Granting to each user a role on a particular graph, each role entailing a set of privileges
- Oauth 2.0-style user authentication
- Extensible framework, so that additional security- and user- related capabilities can be added in future releases



Users and Credentials

TigerGraph's role-based access control system naturally extends to a multiple graph system: A user is granted a role on a particular graph. The superuser role (new in TigerGraph 1.2) is defined for administration of the entire unified supergraph.

TigerGraph users exist only with the TigerGraph platform; they are different than operating system users . When the system is first installed, an initial user is automatically created. The default name for this initial user is tigergraph , with password tigergraph . This user has full administrative privilege and can create additional users and can set their privileges (see Roles and Privileges). For simplicity, we will refer to this initial superuser as the tigergraph user.

If user authentication is enabled (see the section Enabling and Using User Authentication), 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.

1. username-password pair
2. a token: a unique 32-character string which can be used for REST++ requests, with an expiration date.

Enabling and Using User Authentication

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 user tigergraph's password is tigergraph, gsql authentication remains disabled. This is designed for user convenience in single-user configurations or installations which do not require security, such as demo and training installations. The behavior is compatible with early TigerGraph versions which did not support multiple roles or multiple graphs.

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 set up a secure system with user authentication for both points of entry:

1. To enable user authentication for GSQL: change the password of the tigergraph user to something other than tigergraph.
2. To enable Oauth 2 authentication for REST++, use the gadmin program to configure the RESTPP.Authentication parameter. See details below.

More details about each of these two steps are below.

⚠ User authorization for the browser-based GraphStudio UI is in development.

GSQL Authentication

To enable user authentication for GSQL: change the password of the tigergraph user to something other than tigergraph. See ALTER PASSWORD below.

To run a single GSQL command or command file, the user must provide their username and password. The graph also needs to be specified. To specify the username in the command line, use the -u option. The user can also provide their password with the -p option. If the password is not provided on the command line, the system will then prompt the user for their password, so this method is only appropriate for interactive use. If -u not used, then the system will assume that the request is coming from the default tigergraph user. It will then prompt for tigergraph's password (assuming GSQL authentication is enabled). Note that if -u is not used and authentication is disabled, then the system simply responses to all requests, as it did in earlier versions (unprotected administrative mode).

Use the -g parameter to specify which graph on which to operate.

Running one GSQL command or command line, with username, graph name, and password

```
$ gsql [-u username] [-p pasword] [-g gname] <commmand>
Password: ********
```

To enter the GSQL interactive shell, simply omit the <command> from the command line. The user does not need to provide credentials again inside the shell. The example below show s two users entering the shell with their passwords. T he user does not need to specify a graph to enter the interactive shell.

Examples: entering with interactive shell

```
$ gsql
Password for tigergraph: ********
Welcome to GSQL Shell version: 1.2

GSQL > SHOW USER
Users:
  - Name: tigergraph
    - Roles: admin, observer

GSQL > EXIT

$ gsql -u frank
Password for frank: ******
Welcome to GSQL Shell version: 1.2

GSQL > SHOW USER
  - Name: frank
    - Secret: jiokmfqqfu2f95qs6ug85o89rpkneib3
    - Roles: designer, observer

GSQL > EXIT
```

REST++ Authentication

The REST++ server implements OAuth 2.0-style authorization as follows: 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. Anyone who has this secret can invoke a special REST endpoint to generate authorization tokens (other pseudorandom strings). An authorization token can then be used to perform TigerGraph database operations via other REST endpoints. According to OAuth 2.0 protocol, each token will expire after a certain period of time. The TigerGraph default lifetime for a token is 1 month.

Each REST++ request should contain an authorization token in the HTTP header. The REST++ server reads the header. If the token is not valid, REST++ will refuse to run the query and instead will return an authentication error.

Enabling REST++ Authentication

The token authentication of REST++ can be turned on by using the following commands:

Enabling REST++ OAuth Authentication

```
gadmin --configure RESTPP.Authentication
gadmin config-apply
gadmin restart restpp nginx vis -y
```

Creating Tokens

A user must have a secret before they create a token. Secrets are generated in GSQL (see CREATE SECRET below). The special endpoint
GET /requesttoken is used to create a token. The endpoint has two parameters:

- secret (required): the user's secret
- lifetime (optional): the lifetime for the token, in seconds. The default is one month, approximately 2.6 million seconds.

Example: REST++ Request to Generate a Token

```
curl -X GET 'localhost:9000/requesttoken?secret=jiokmfqqfu2f95qs6ug85o89rp
```

Using Tokens

Once REST++ authentication is enabled, a 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>" 'http://localhost:9000/quer
```

When you use the RUN QUERY command in the GSQL language, this triggers a curl command within the GSQL system. GSQL will automatically use (and generate, if necessary) a token in the curl request for an authorized user.

> ⚠ Authorization for gadmin
>
> Currently, authorization for the gadmin program comes from Linux, and is not related GSQL authorization. In short, only the Linux TigerGraph user can run gadmin.
>
> Details: During installation, the user selects a name and password for the TigerGraph Linux user. The default user and password are tigergraph and tigergraph, respectively. This user is a Linux user; the installer will create a Linux account if needed. Only the TigerGraph Linux user can run gadmin. This Linux user is unrelated to the TigerGraph default user mentioned in the GSQL Authentication section.

Roles and Privileges

The TigerGraph system includes six predefined roles — superuser, admin, designer, querywriter, queryreader, and observer. Each role has a fixed and logical set of privileges to perform operations. These roles form a hierarchy, with superuser being at the top. Broadly speaking,

- An **observer** (formerly "public") can log on, view the schema and other catalog details for its designated graph, and change their own password.
- A **queryreader** has all observer privileges, and can also run existing loading jobs and queries for its designated graph.
- A **querywriter** has all queryreader privileges, and can also create queries and run data-manipulation commands on its designated graph.
- A **designer** (formerly "architect") has all querywriter privileges, and can modify the schema, create loading jobs for its designated graph.
- An **admin** has all designer privileges, and can also create or drop users and grant or revoke roles for its designated graph. That is, an admin can control the existence and privileges of other users on its graph.
- A **superuser** automatically has admin privileges on all graphs, and can also create global vertex and edge types, create multiple graphs, and clear the database.

The detailed permissions for each role are listed in the following table. Except for the superuser, the scope of privilege is always limited to one's own graph. In some cases, the behavior of the operation depends on one's privilege level. More detailed descriptions of the User Management commands are given later in this

document. For details about the Graph Definition, Loading, Querying, and Modification commands, see the GSQL Language Reference documents.

Command Type	Operations	super user	admin	designer	query-writer	que reac
Status	Ls	x	x	x	x	x
User Management	Create/Drop User	x	x			
	Show User	x	x	x	x	x
	Alter (Change) Password	x	x	x	x	x
	Grant/Revoke Role	x	x			
	Create/Drop/Show Secret,	x	x	x	x	x
	Create/Drop/Show/Refresh Token (Deprecated)	x	x	x	x	x
Schema Design	Create/Drop Vertex/Edge/Graph	x				
	Clear Graph Store	x				
	Drop All	x				
	Use Graph	x	x	x	x	x
	Use Global	x	x	x	x	x

	Create/Run Global Schema_Change Job	x				
	Create/Run Schema_Change Job	x	x	x		
Loading and Querying	Create/Drop Loading Job	x	x	x		
	Create/Interpret/Install/ Drop Query	x	x	x	x	
	Typedef	x	x	x	x	
	Offline to Online Job Translation	x	x	x	x	
	Run Query	x	x	x	x	x
	Run Loading Job	x	x	x	x	x
Data Modification	Upsert/Delete/Select Commands	x	x	x	x	

Creating and Managing Users

The TigerGraph installation process creates one user called tigergraph who has the superuser role. The superuser role has full privilege to perform any action, included

creating or removing other users, and assigning roles to the other users. An superuser can create other superusers, who would also have full privilege.

An admin user is similar to a superuser whose scope is limited to a designated graph. An admin can create and manage other users for that graph alone.

The user tigergraph is permanent. It cannot be dropped by another admin user.

Most of the commands in this section, can be run only by a superuser or an admin user. The exception is SHOW USER. Any user can display their own profile.

User Management Commands

```
CREATE USER
DROP USER <user1>,...<userN>
SHOW USER
ALTER PASSWORD [<user1>]
GRANT ROLE admin [ON GRAPH <gname>] TO <user1>,...<userN>
REVOKE ROLE admin [ON GRAPH <gname>] FROM <user1>,...<userN>
```

CREATE USER

Required privilege: superuser, admin
Create a new user. GSQL will prompt for the user name and password.

Example: Create user

```
tigergraph:GSQL > CREATE USER
User Name : frank
New Password : ************
Re-enter Password : ************
The user "frank" is created.
```

DROP USER

```
DROP USER <user1>,...<userN>
```

Required privilege: superuser, admin
Delete the listed users.

⚠ The command takes effect with no warning and cannot be undone.

Example: Drop two users

```
tigergraph:GSQL > DROP USER hermione, jk
Password: *********
The user "hermione" is dropped.
The user "jk" is dropped.
```

SHOW USER

Required privilege: any
Display user's name, role, secret, and token. Non-admin/superuser users see only their own information. Admin/superuser users see information for all users.

Example: admin user showing profile information for all users

```
tigergraph:GSQL > SHOW USER
Users:
  - Name: tigergraph
    - Roles: admin, observer

  - Name: frank
    - Secret: 3ridhimp5icllq04qgt0r1fgddv1hf9e
      - Token: j13nv837thrr19u0ahjr8m0is2ded6kk expire at: 2017-09-13 15:1
    - Roles: designer, observer

  - Name: jk
    - Roles: observer

  - Name: hermione
    - Roles: observer
```

GRANT/REVOKE ROLE

```
GRANT ROLE <username> [ON GRAPH <gname>] TO <user1>,...<userN>
REVOKE ROLE <username> [ON GRAPH <gname>] FROM <user1>,...<userN>
```

Required privilege: superuser, admin
Grant a role (or revoke a role) for a user, which add s (or removes) privileges.

The ON GRAPH clause is required unless the role being granted/revoked is superuser.

The example below grants the queryreader role to two users, revokes it from one of the them (jk), and then grants the querywriter role to both users.

Example: Granting and Revoking Roles

```
GSQL > GRANT ROLE queryreader ON GRAPH Hogwarts TO jk,hermione
Role "queryreader" is successfully granted to user(s): jk, hermione

GSQL > REVOKE ROLE queryreader ON GRAPH Hogwarts FROM hermione
Role "queryreader" is successfully revoked from user(s): hermione

GSQL > GRANT ROLE querywriter ON GRAPH London TO hermione,jk
Role "querywriter" is successfully granted to user(s): hermione, jk

GSQL > SHOW USER
Users:
* - Name: tigergraph
    - Roles: superuser

  - Name: hermione
    - Roles:
      - GraphName: London
        - Roles: querywriter

  - Name: jk
    - Roles:
      - GraphName: Hogwarts
        - Roles: queryreader
      - GraphName: London
        - Roles: querywriter
```

A user can have more than one role. For example, jk can be a queryreader on the

Hogwarts graph and a querywriter on the London graph.

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.

1. user name and password pair.
2. a token: a unique 32-character string which can be used for REST++ requests. A token expires 3 months from the date of creation.

The following set of commands are used to create and manage passwords, authentication secrets, and authentication tokens.

GSQL Commands for Managing Credentials

```
ALTER PASSWORD [user1]
CREATE SECRET [alias1]
SHOW SECRET
DROP SECRET <secret1>
CREATE TOKEN
SHOW TOKEN
DROP TOKEN <token1>
REFRESH TOKEN <token1>
```

Like any other GSQL commands, the user must supply credentials to run these commands. In order to create a secret or create a token, the user must supply their password.

ALTER PASSWORD

```
ALTER PASSWORD [<user1>]
```

When an admin/superuser user creates a new user, the admin/superuser user sets the user's initial password. Afterward, a user can change their own password.

Example: Admin user changing his/her own password

```
herminone:GSQL > ALTER PASSWORD
Password: *******
New Password : ************
Re-enter Password : ************
Password has been changed.
```

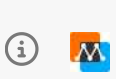
Moreover, an admin/superuser user can revise any user's password. For example, to change hermione's password, the command is ALTER PASSWORD hermione .

Example: Admin user changing another user's password

```
tigergraph:GSQL > ALTER PASSWORD hermione
Password: *******
New Password : ************
Re-enter Password : ************
Password has been changed.
```

CREATE / SHOW / DROP SECRET

These commands create and manage a user's secrets, unique strings which can serve as a user's credentials in certain circumstances. A user can have multiple secret strings. Each time that CREATE SECRET is executed, a new secret string is created. Therefore, when running DROP SECRET, the user must specify which secret is to be dropped. The following example shows a series of commands: log into the GSQL shell with a password, create two secrets, one for each of two graphs, then drop one of the secrets.

 A secret represents more than just the user's identity but also the user's role for a particular graph. If user's role is revoked, the secret becomes invalid.

Example of CREATE / SHOW / DROP SECRET commands

```
$ gsql -u jk -g Hogwarts
Password for jk : *******
Welcome to GSQL Shell version: 1.2

GSQL > CREATE SECRET HH
The secret: 4sjmn1q13vp2klqb7v3t151vac9db2am has been created for user "ti

GSQL > SHOW SECRET
    - Secret: 4sjmn1q13vp2klqb7v3t151vac9db2am
      - Alias: HH
      - GraphName: Hogwarts

GSQL > USE GRAPH London
Using graph 'London'

GSQL > CREATE SECRET LL
The secret: 75j8kf75g545mgc24mefsjm1iic7m9i2 has been created for user "ti

GSQL > SHOW SECRET
    - Secret: 4sjmn1q13vp2klqb7v3t151vac9db2am
      - Alias: HH
      - GraphName: Hogwarts
    - Secret: 75j8kf75g545mgc24mefsjm1iic7m9i2
      - Alias: LL
      - GraphName: London

GSQL > USE GRAPH Hogwarts
Using graph 'Hogwarts'

GSQL > DROP SECRET 4sjmn1q13vp2klqb7v3t151vac9db2am
Secret 4sjmn1q13vp2klqb7v3t151vac9db2am has been removed.

GSQL > SHOW SECRET
    - Secret: 75j8kf75g545mgc24mefsjm1iic7m9i2
      - Alias: LL
      - GraphName: London
```

CREATE / SHOW / DROP / REFRESH TOKEN (deprecated)

⚠ The TOKEN commands in GSQL are deprecated. The recommended procedure to create tokens is to use the REST++ endpoint GET /requesttoken.

These commands create and manage a user's tokens, unique strings which can be used as credentials when making a REST++ request. In fact, tokens are the only credentials that can be used for REST++ requests. In order to create a token, a user must first have a secret. A user can have multiple tokens, but each token is associated with its secret. Each token is given a lifetime and expiration date when it is created; the default lifetime is 3 months. However, the clock can be reset, giving 3 months from the current time, by using the REFRESH TOKEN command.

The following example shows a series of commands: log into the GSQL shell, create a second secret, create a token for one secret, create another token for another secret, and drop one token.

Example of CREATE / SHOW / REFRESH / DROP TOKEN commands

```
$ gsql -u jk -g London
Password for jk : *******

GSQL > CREATE SECRET
The secret: mv88grasoidc7fenk1ffl6hnll8f2apf has been created for user "jk

GSQL > SHOW SECRET
    - Secret: 33bt4o86c33nauhhipaenh9pluun86po
      - GraphName: London
    - Secret: mv88grasoidc7fenk1ffl6hnll8f2apf
      - GraphName: London

GSQL > CREATE TOKEN
Secret : 33bt4o86c33nauhhipaenh9pluun86po
The access token: kn1hlp1a6b9lugq2mkqohsd8i0ht2tt3 is created and it will

GSQL > SHOW TOKEN
    - Secret: 33bt4o86c33nauhhipaenh9pluun86po
      - Token: kn1hlp1a6b9lugq2mkqohsd8i0ht2tt3 expire at: 2018-05-03 18:3
      - GraphName: London
    - Secret: mv88grasoidc7fenk1ffl6hnll8f2apf
      - GraphName: London

GSQL > CREATE TOKEN
Secret : mv88grasoidc7fenk1ffl6hnll8f2apf
The access token: ieec6odigmja01rkt3qmq0nar4iufsvq is created and it will

GSQL > SHOW TOKEN
    - Secret: 33bt4o86c33nauhhipaenh9pluun86po
      - Token: kn1hlp1a6b9lugq2mkqohsd8i0ht2tt3 expire at: 2018-05-03 18:3
      - GraphName: London
    - Secret: mv88grasoidc7fenk1ffl6hnll8f2apf
      - Token: ieec6odigmja01rkt3qmq0nar4iufsvq expire at: 2018-05-03 18:3
      - GraphName: London

GSQL > DROP TOKEN kn1hlp1a6b9lugq2mkqohsd8i0ht2tt3
Token kn1hlp1a6b9lugq2mkqohsd8i0ht2tt3 has been removed.

GSQL > SHOW TOKEN
    - Secret: 33bt4o86c33nauhhipaenh9pluun86po
      - Alias: LL
      - GraphName: London
    - Secret: mv88grasoidc7fenk1ffl6hnll8f2apf
      - Token: ieec6odigmja01rkt3qmq0nar4iufsvq expire at: 2018-05-03 18:3
      - GraphName: London
```

Creating and Managing Proxy Groups

Proxy groups are used for LDAP Authentication. The CREATE / SHOW / DROP GROUP commands require the superuser or admin privilege.

CREATE GROUP

```
CREATE GROUP <groupname> PROXY "<attributename>=<value>"
```

Required privilege: superuser, admin
Create a proxy group whose membership is defined as those users who have an attribute satisfying the rule <attributename>=<value>.

After a group has been defined, roles can be granted to (or revoked from) the group, as in the example below:

CREATE GROUP and GRANT ROLE example

```
CREATE GROUP developers PROXY "role=engineering"
GRANT ROLE querywriter ON GRAPH computerNet TO developers
```

SHOW GROUP

```
SHOW GROUP <groupname>
```

Required privilege: superuser, admin. Display information about a group.

DROP GROUP

```
DROP GROUP <groupname>
```

Required privilege: superuser, admin. Delete the named group definition. The users in the group will lose this proxy association but are otherwise unaffected.

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 are used to provide centralized user authentication service. 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 LDAPv3 protocol. StartTLS/SSL connection is also supported.

SASL authentication is not yet supported. Some LDAP server are configured to require a client certificate upon connection. Client certificate is 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 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 ↗.

Step 1 - Configure the LDAP Connection

To enable and configure LDAP, run three commands.

1. Configure LDAP:

```
gadmin --configure ldap
```

The gadmin program will then prompt the user for the settings for several LDAP configuration parameters.

2.Apply the configuration:

```
gadmin config-apply
```

3.Restart the gsql server:

```
gadmin restart gsql -y
```

An example configuration is shown below.

Example of gadmin --configure ldap

```
$ gadmin --configure ldap
# Enable LDAP authentication: default false
security.ldap.enable [False]: true
True

# Configure LDAP server hostname: default localhost
security.ldap.host [ldap.tigergraph.com]: ldap.tigergraph.com
ldap.tigergraph.com

# Configure LDAP server port: default 389
security.ldap.port [389]: 389
389

# Configure LDAP search base DN, the root node to start the LDAP search fo
security.ldap.base_dn [dc=tigergraph,dc=com]: dc=tigergraph,dc=com
dc=tigergraph,dc=com

# Configure LDAP search base DN, the root node to start the LDAP search fo
security.ldap.search_filter [(objectClass=*)]:
(objectClass=*)

# Configure the username attribute name in LDAP server: default uid
security.ldap.username_attribute [uid]: uid
uid

# Configure the DN of LDAP user who has read access to the base DN specifi
security.ldap.admin_dn [cn=Manager,dc=tigergraph,dc=com]: cn=Manager,dc=ti
cn=Manager,dc=tigergraph,dc=com

# Configure the password of the admin DN specified above. Needed only when
security.ldap.admin_password [secret]: secret
secret

# Enable SSL/StartTLS for LDAP connection [none/ssl/starttls]: default non
security.ldap.secure.protocol [starttls]: none
none

# Configure the truststore path for the certificates used in SSL: default
security.ldap.secure.truststore_path [/tmp/ca_server.pkcs12]:
/tmp/ca_server.pkcs12

# Configure the truststore format  [JKS/PKCS12]: default JKS
security.ldap.secure.truststore_format [pkcs12]:
pkcs12

# Configure the truststore password: default changeit
security.ldap.secure.truststore_password [test]:
test
```

```
# Configure to trust all LDAP servers (unsafe): default false
security.ldap.secure.trust_all [False]: false
```

Below is an explanation of each configuration parameter.

security.ldap.enable

Set to "true" to enable LDAP; "false" to disable LDAP.

security.ldap.host

Hostname of LDAP server.

security.ldap.port

Port of LDAP server.

security.ldap.base_dn

Base DN (Distinguished Name), in order for GSQL to perform the LDAP search.

security.ldap.search_filter

A search filter is optional. When configured, the search is only performed for the LDAP entries that satisfy the filter. The filter must strictly follow LDAP filter format, i.e., the condition must be wrapped by parentheses, etc. A description of the different types of filters is available at https://www.ldap.com/ldap-filters ↗. The official specification for LDAP filters is available at https://docs.ldap.com/specs/rfc4515.txt ↗.

security.ldap.username_attribute

This specifies the LDAP attribute to search when the GSQL server looks up the usernames in the LDAP server upon login. For example, in the configuration shown above, when a user logs in with the "-u john" option, the GSQL server will search the

"uid" attribute in LDAP to find "john" and check the credentials only after "john" is found.

security.ldap.admin_password & security.ldap.admin_dn

These options are needed when the LDAP server is not publicly readable. In this case, the admin DN and corresponding password need to be specified in order for the GSQL server to connect to the LDAP server.

security.ldap.secure.protocol

When set to "none", TigerGraph uses insecure LDAP connection. This can be changed to a secure connection protocol: "starttls" or "ssl".

security.ldap.secure.truststore_path & security.ldap.secure.truststore_password

When starttls or ssl is used, a truststore path as well as its password needs to be configured.

security.ldap.secure.truststore_format

Currently, the TigerGraph system supports two trustore formats: pkcs12 and jks.

security.ldap.secure.trust_all

When specified, the GSQL server will blindly trust any LDAP sever.

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 login 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
```

⚠ Only users with the admin and superuser role can create, show, or drop a group.

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. If your IDP supports SAML 2.0 protocol, you should be able to integrate your identity provider with TigerGraph Single Sign-On .

Currently we have verified following identity providers:

- Okta ↗
- Auth0 ↗

In order to use Single Sign-On , you need 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-hostname: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.

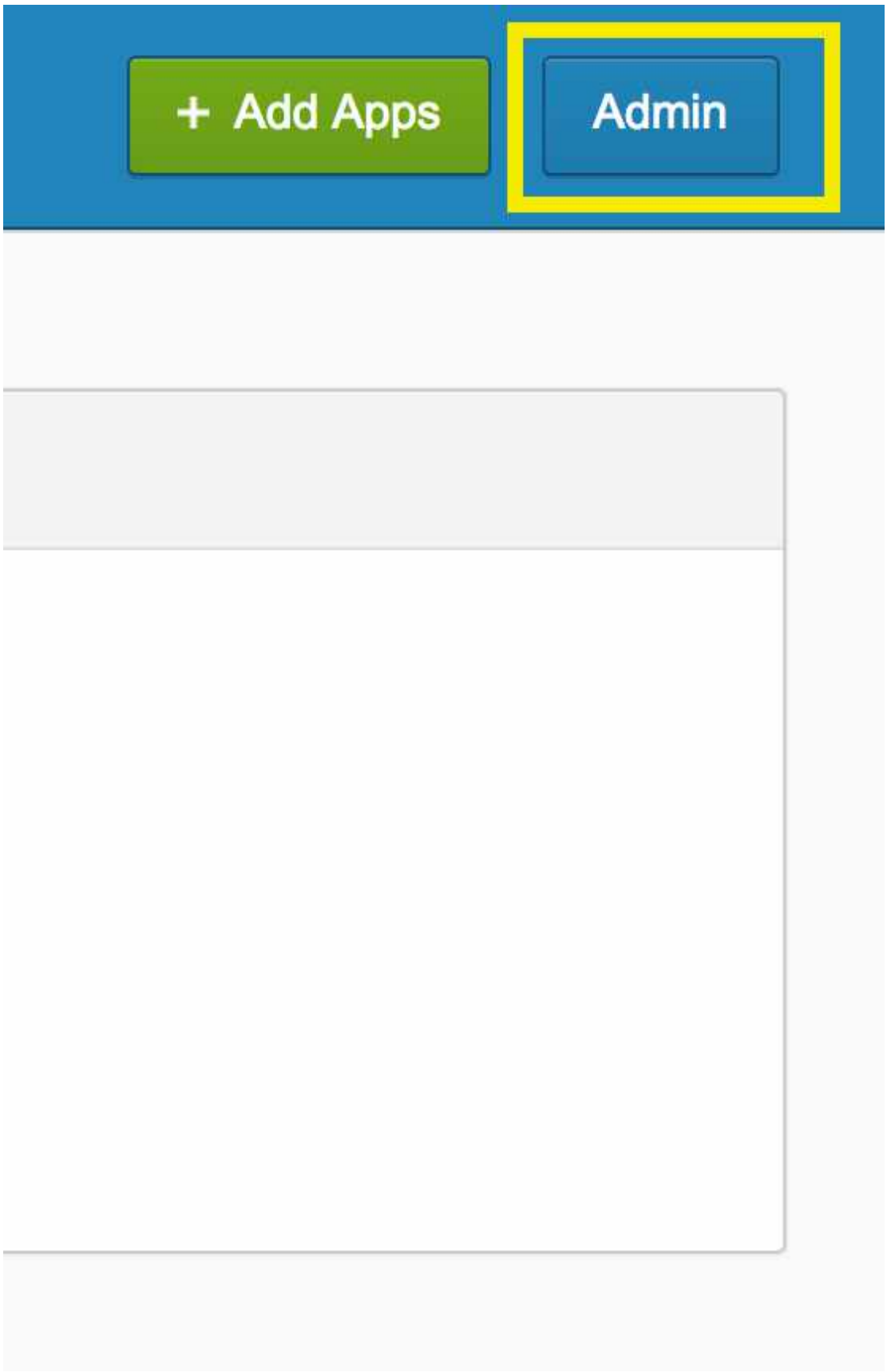
Configure Identity Provider

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.

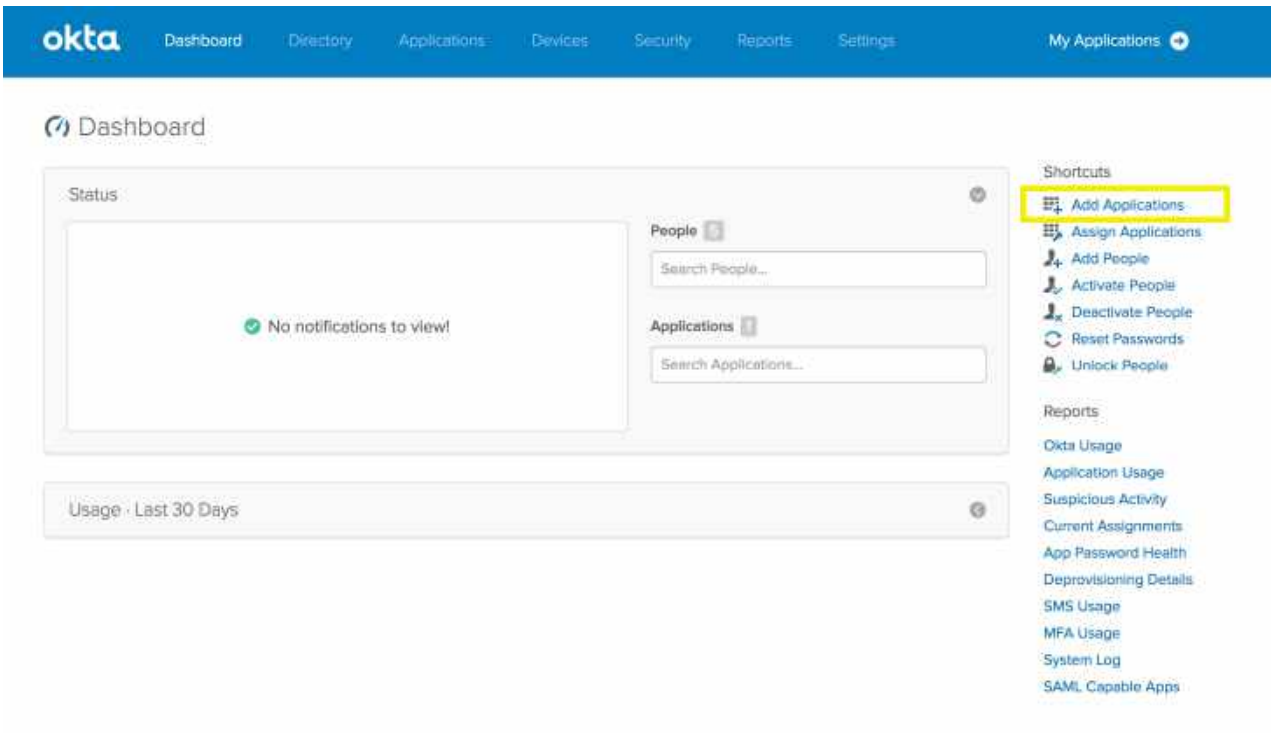
After you finish configuring your identity provider, you will get an Identity Provider Single Sign-On URL , Identity Provider Entity Id , and an X.509 certificate file idp.cert . You need these 3 things to configure TigerGraph next.

Okta

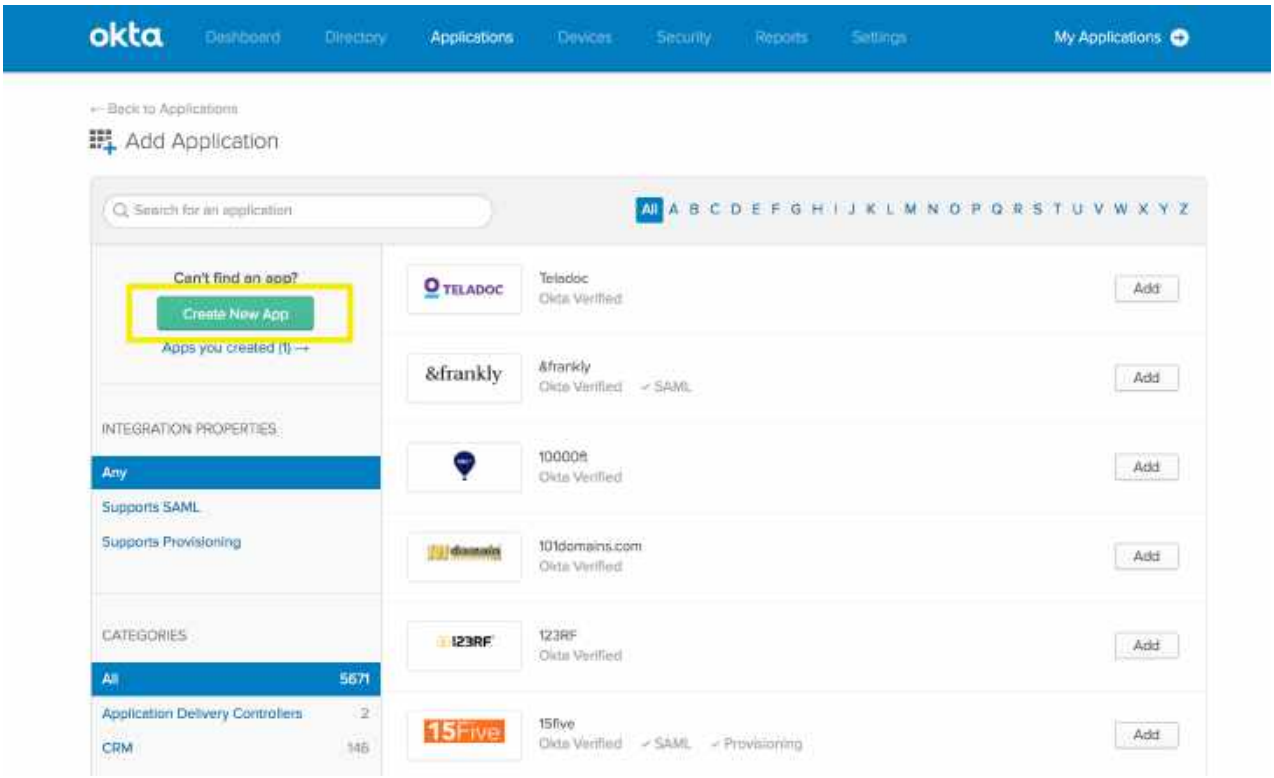
After logging into Okta as the admin user, click Admin button at the top-right corner.



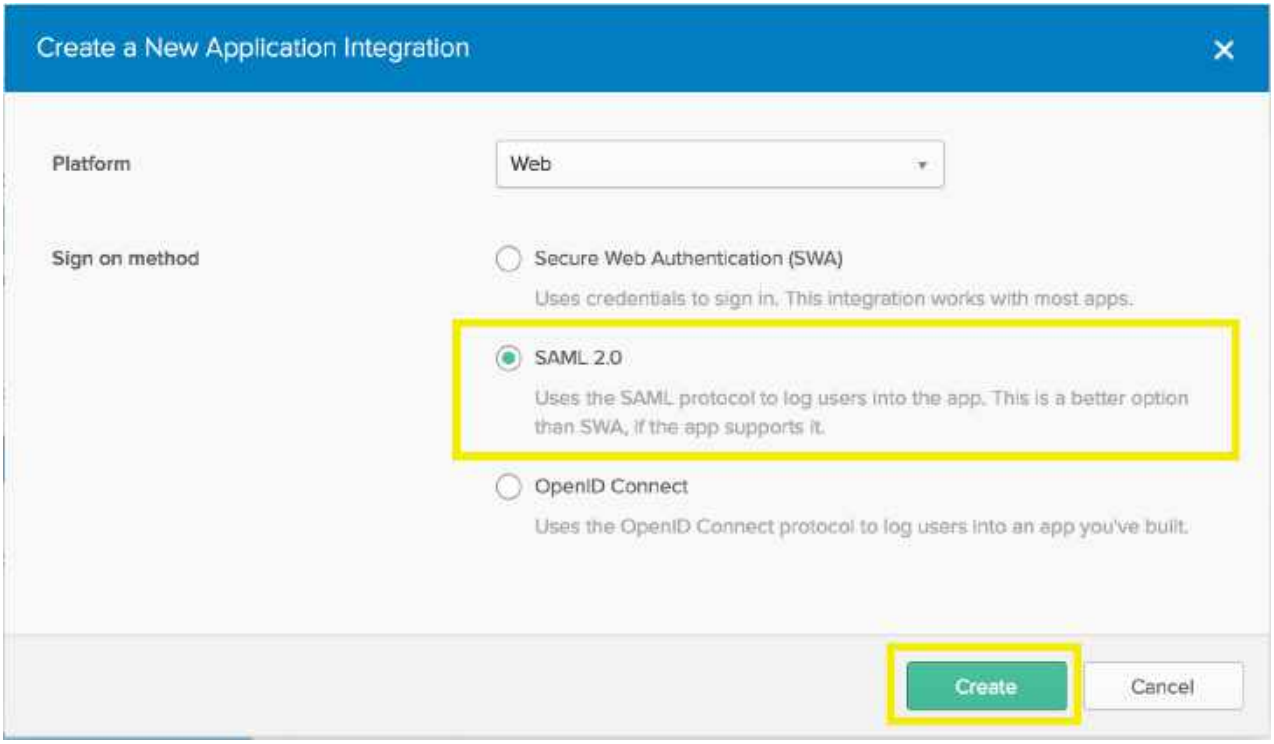
Click Add Applications in the right menu.



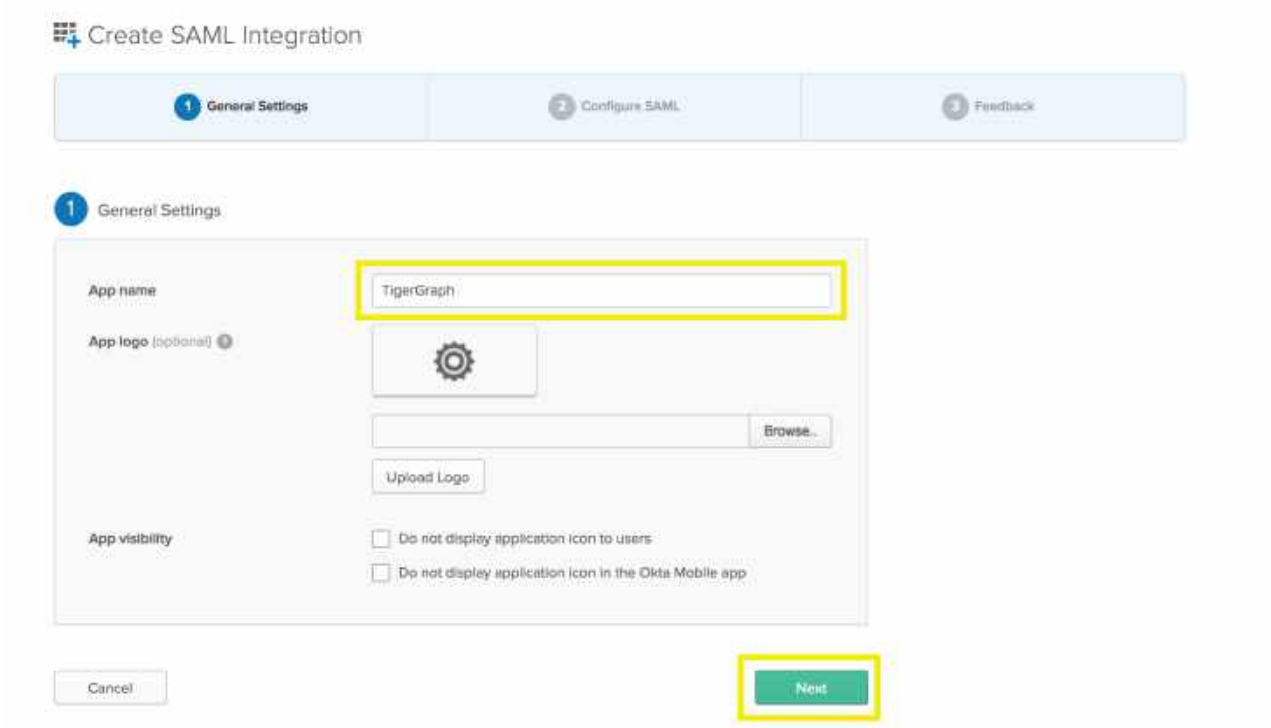
Click Create New App button in the left toolbar.



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 .

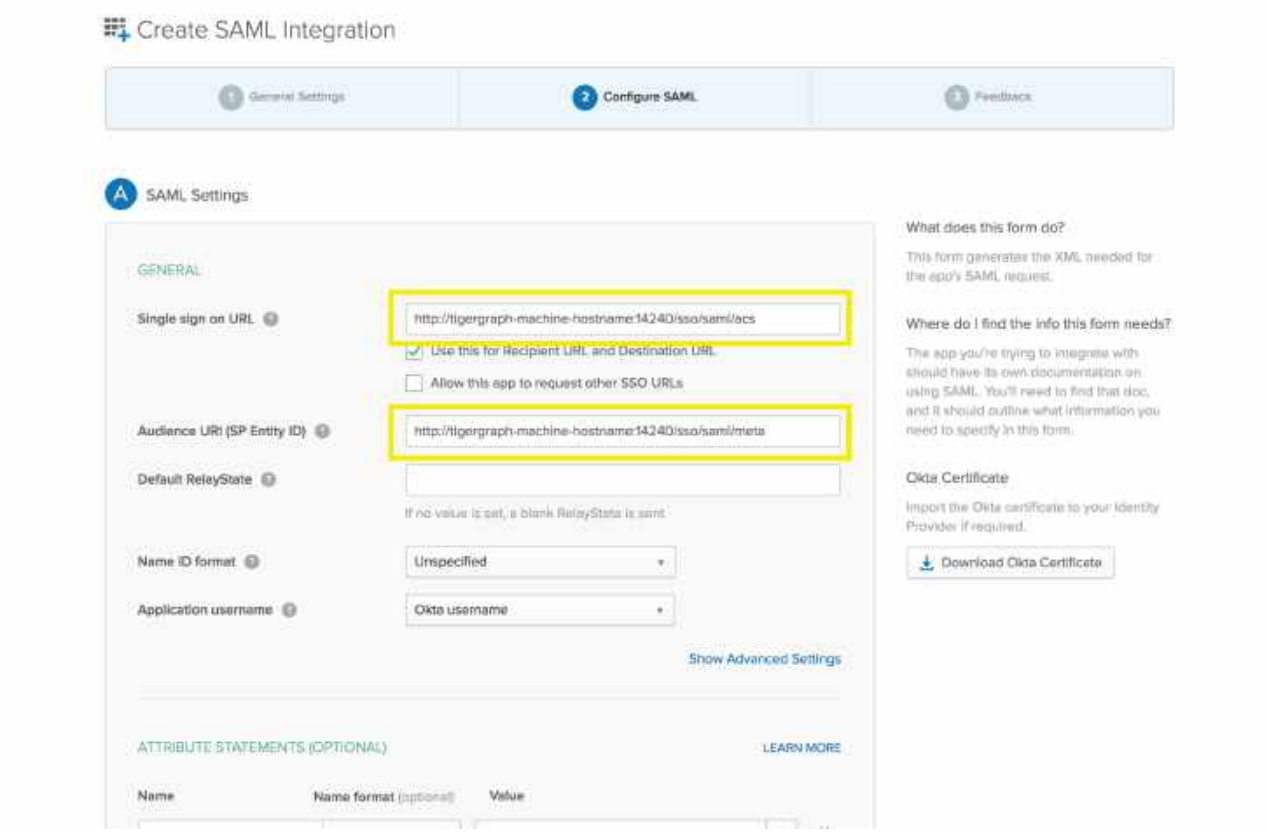
Both are URLs in our case. You need to know the hostname of the TigerGraph machine. If you can visit GraphStudio UI through a browser, the URL contains the hostname. It can be either an IP or a domain name.

The Assertion Consumer Service URL , or Single sign on URL, is

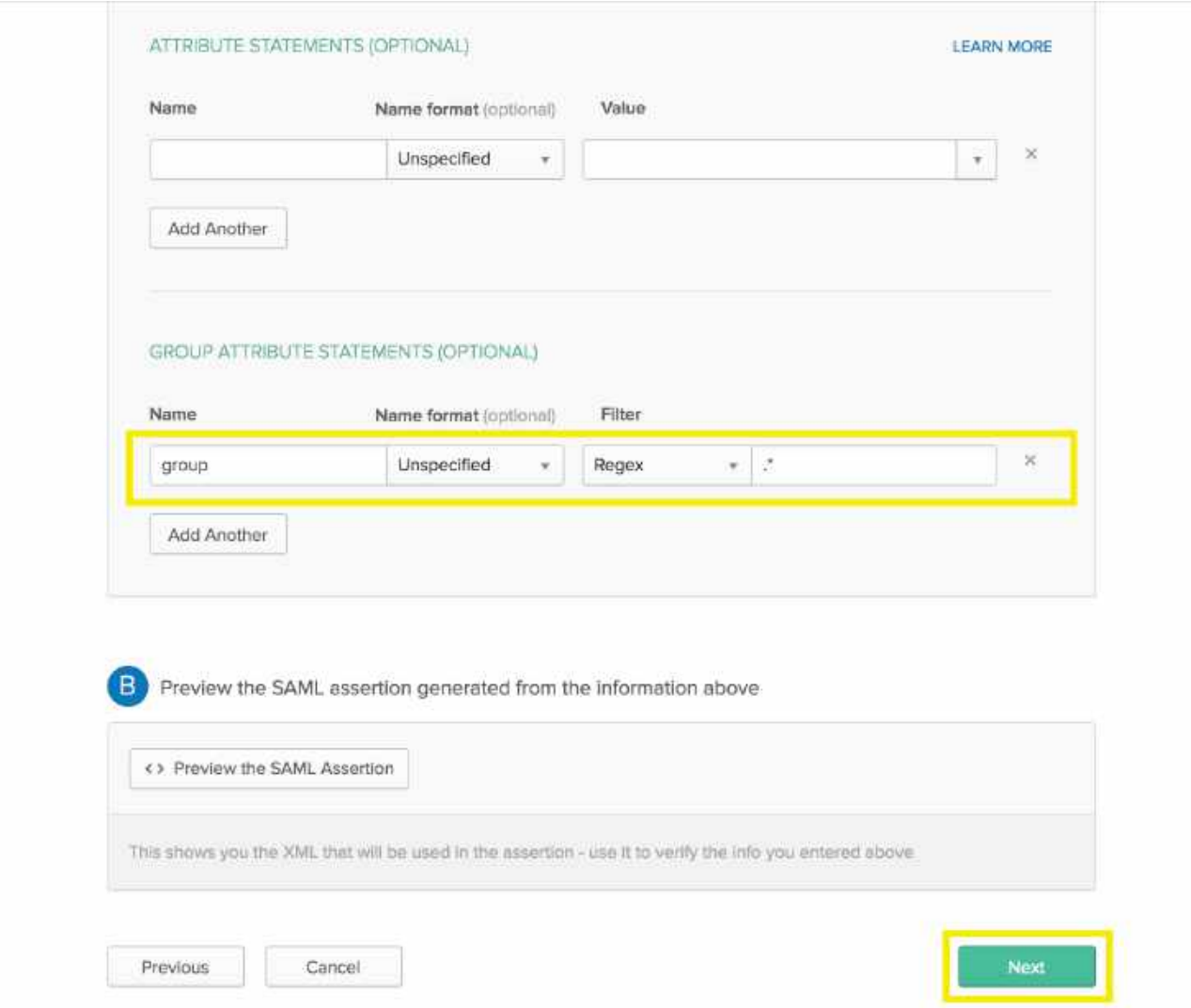
> http://tigergraph-machine-hostname:14240/sso/saml/acs

The SP entity id URL is:

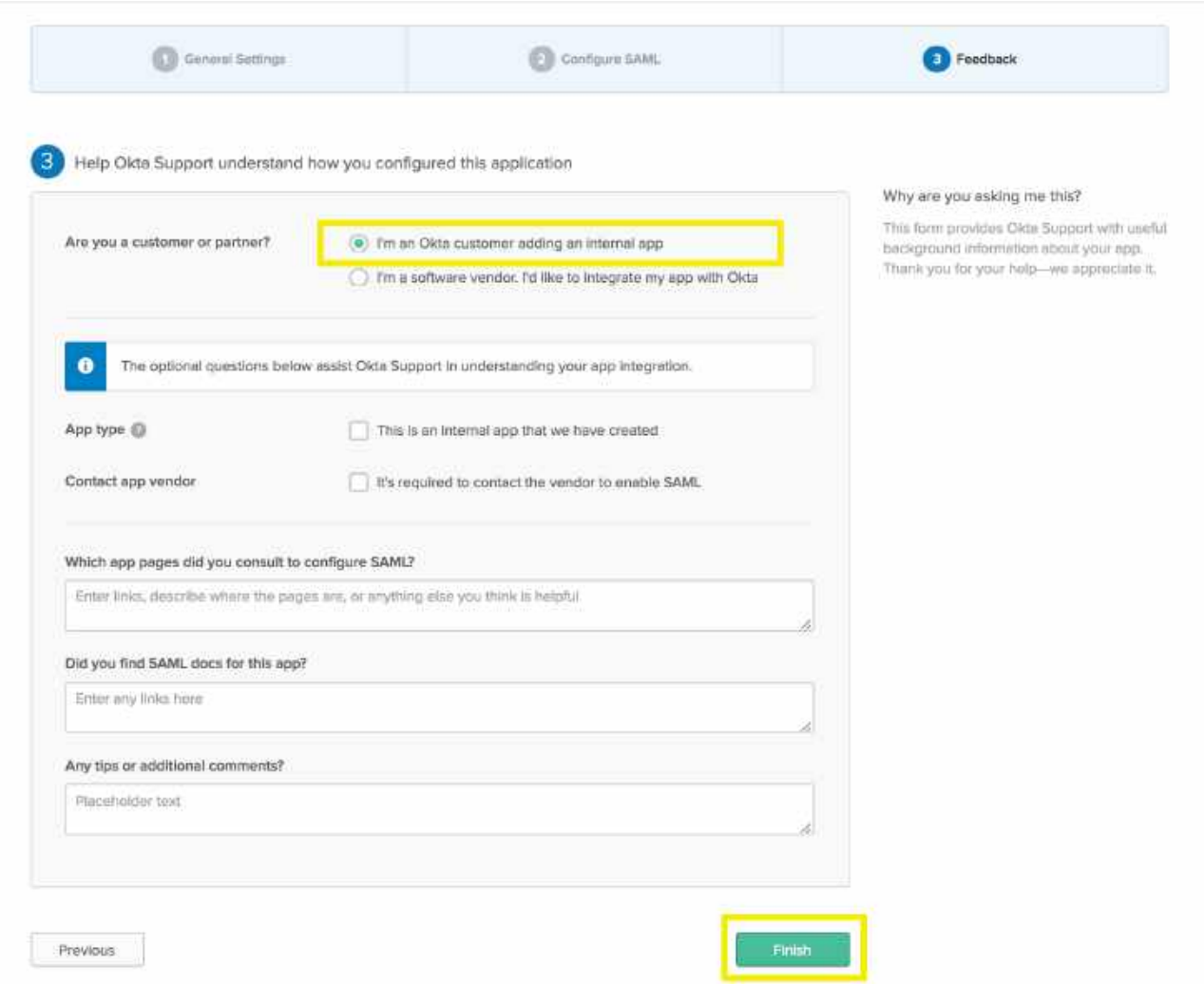
> http://tigergraph-machine-hostname:14240/sso/saml/meta



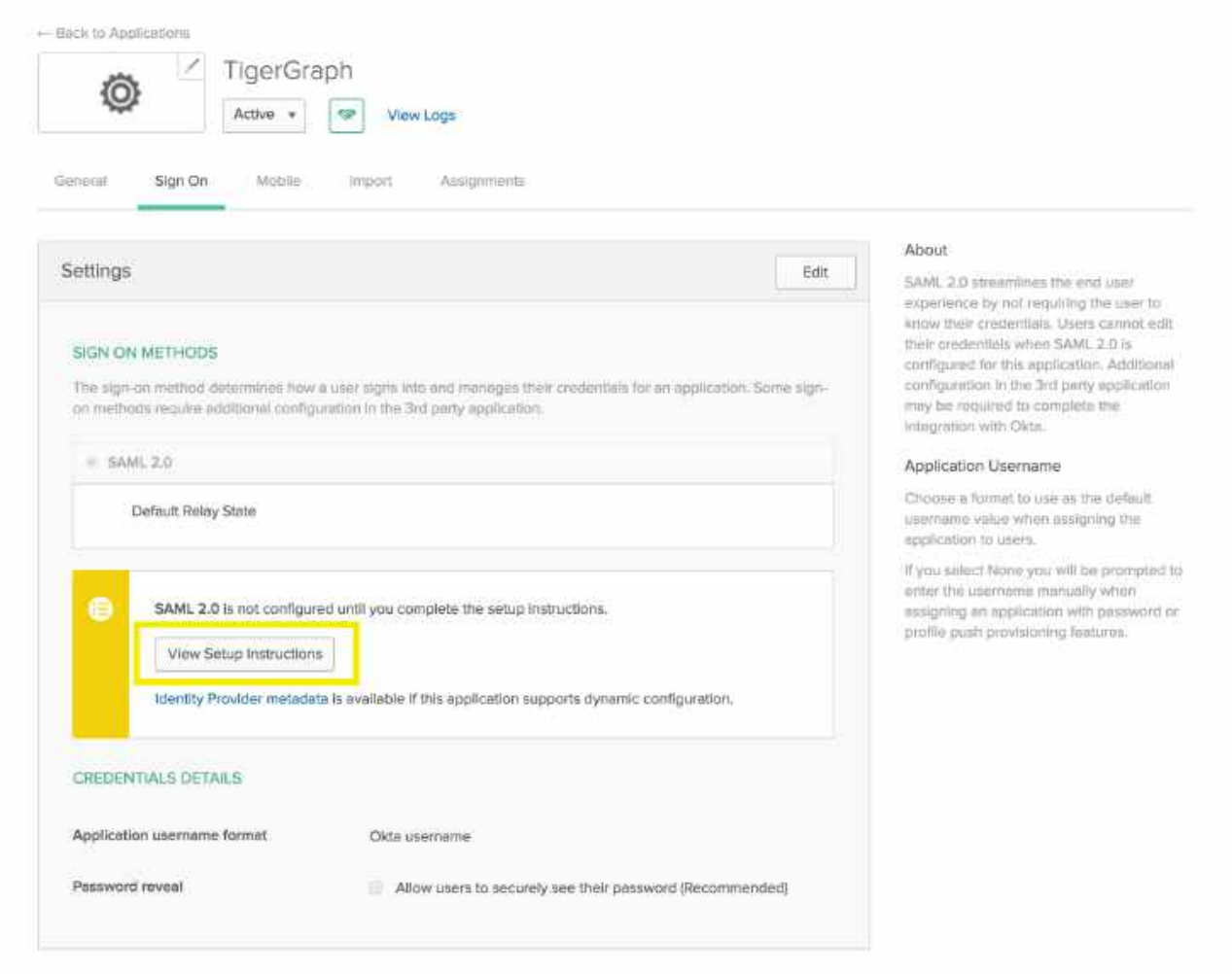
Scroll to the bottom for Group Attribute Statements. Usually you want to grant roles to users based on their user group. You can give a name to your attribute statement; here we use group . For filter, we want to return all group attribute values of all users, so we use Regex .* as the filter. Click Next after you set up everything.



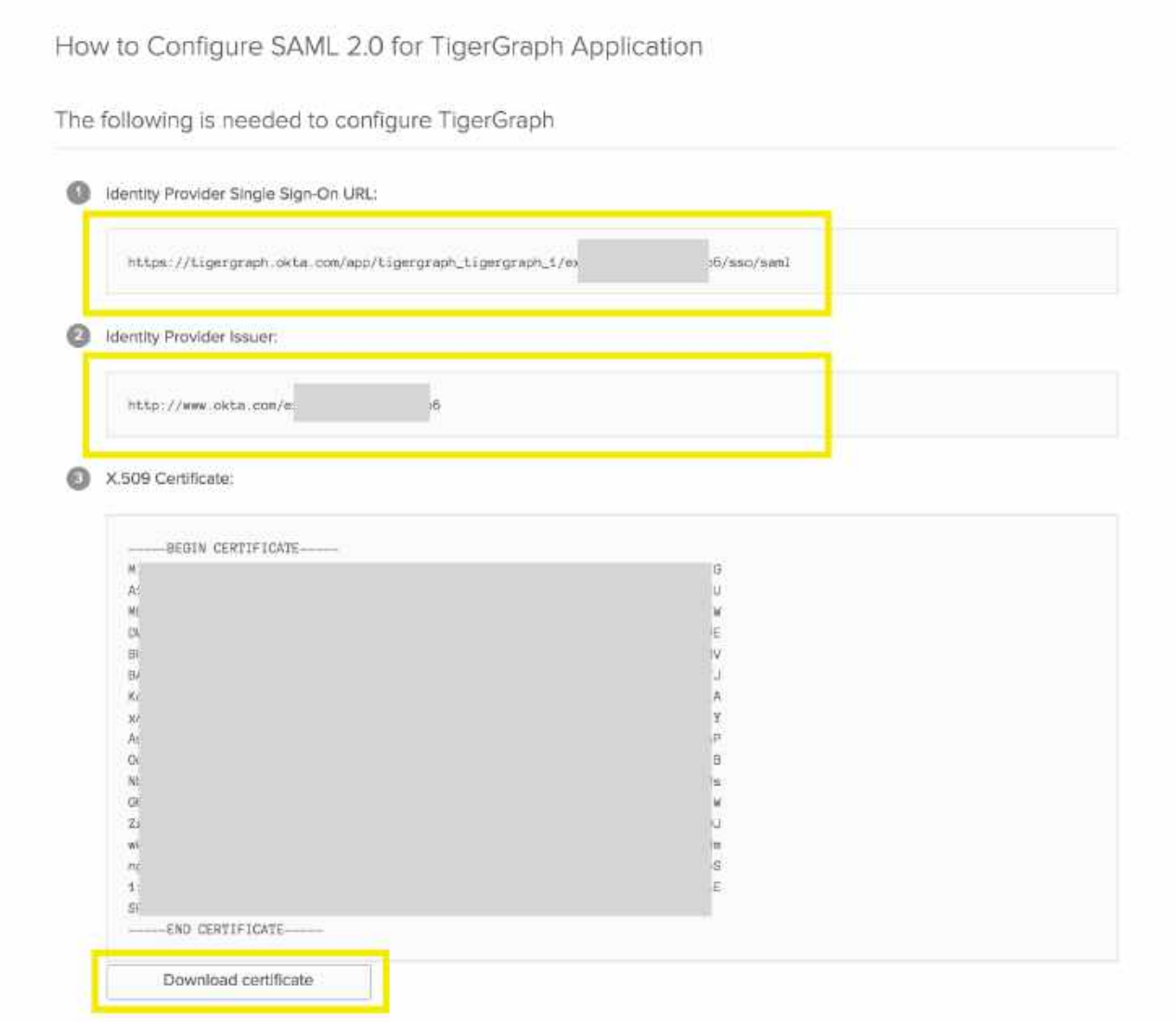
In the final step, choose whether you want to integrate your app with Okta or not. Then click Finish .



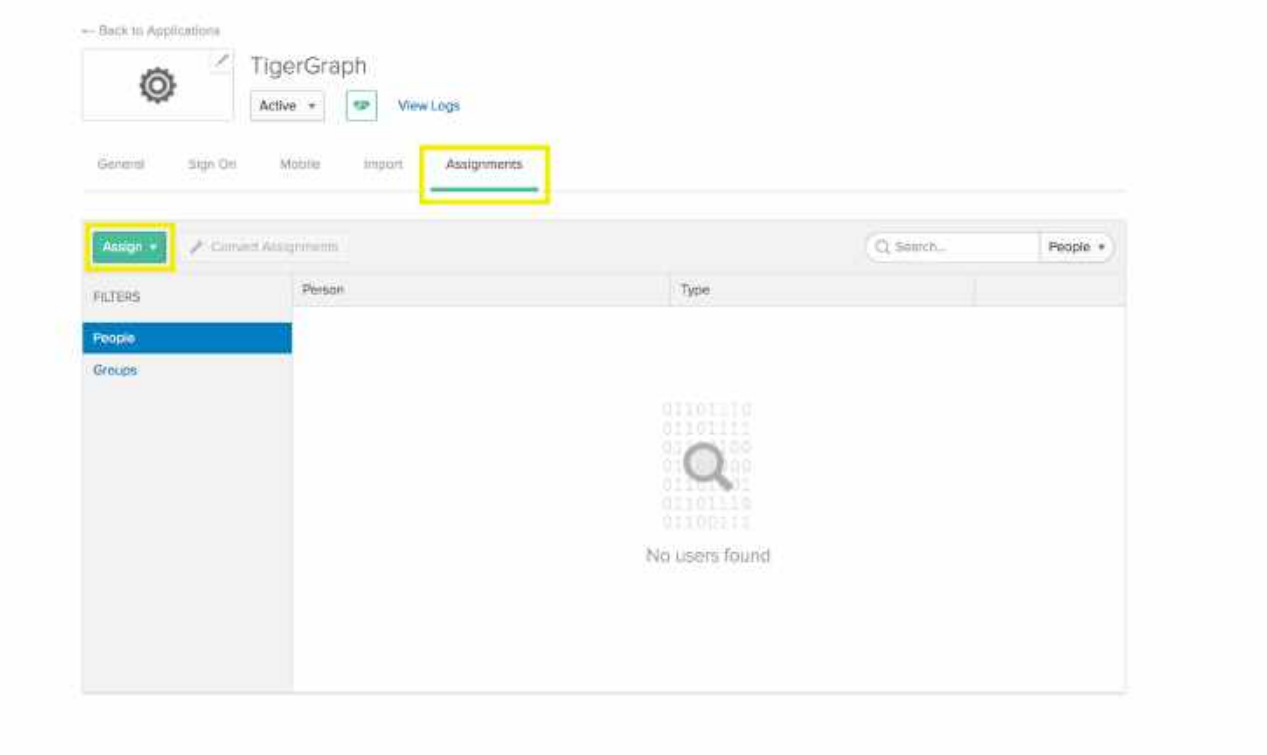
Now your Okta identity provider settings are finished. Click View Setup Instructions button to gather information you will need to setup TigerGraph Single Sign-On.



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).

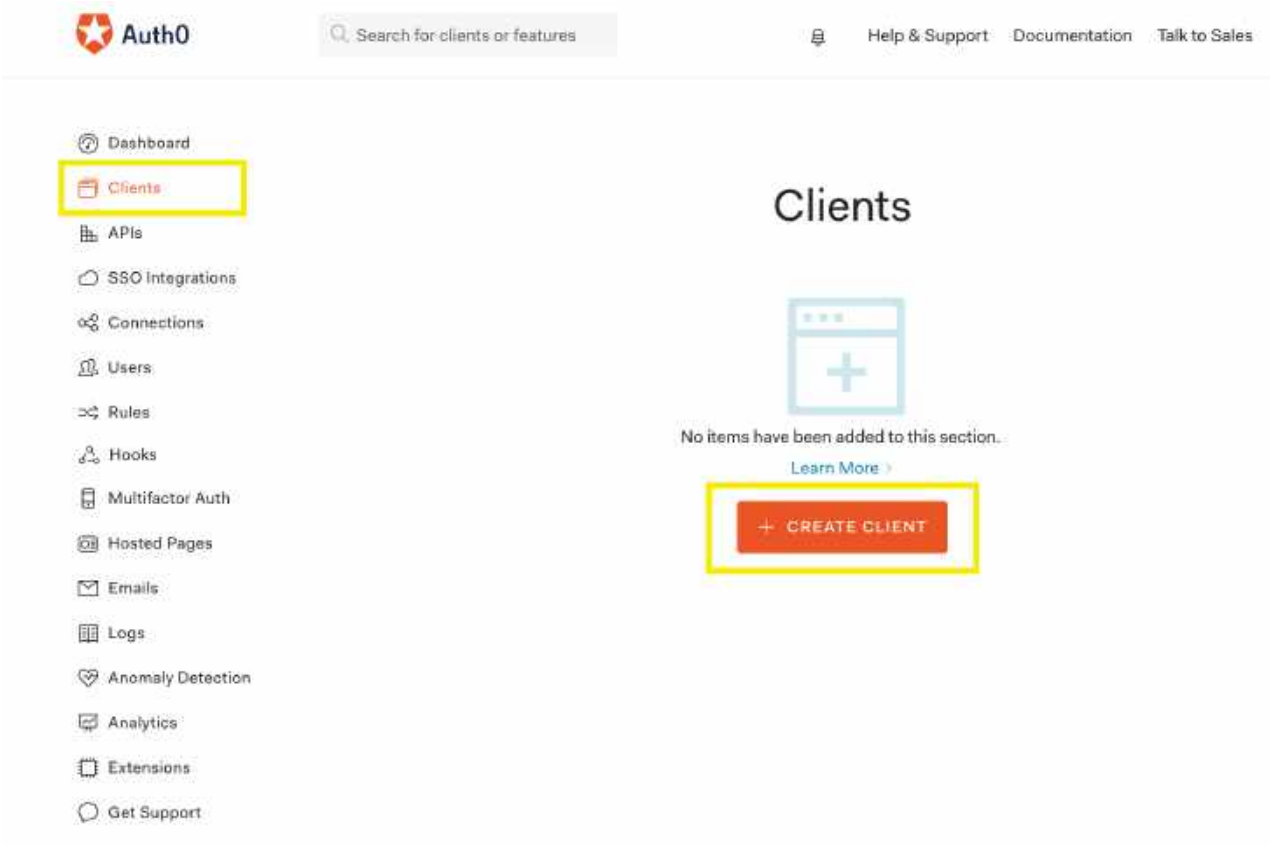


Finally, return to previous page, go to the Assignments tab, click the Assign button, and assign people or groups in your organization to access this application.

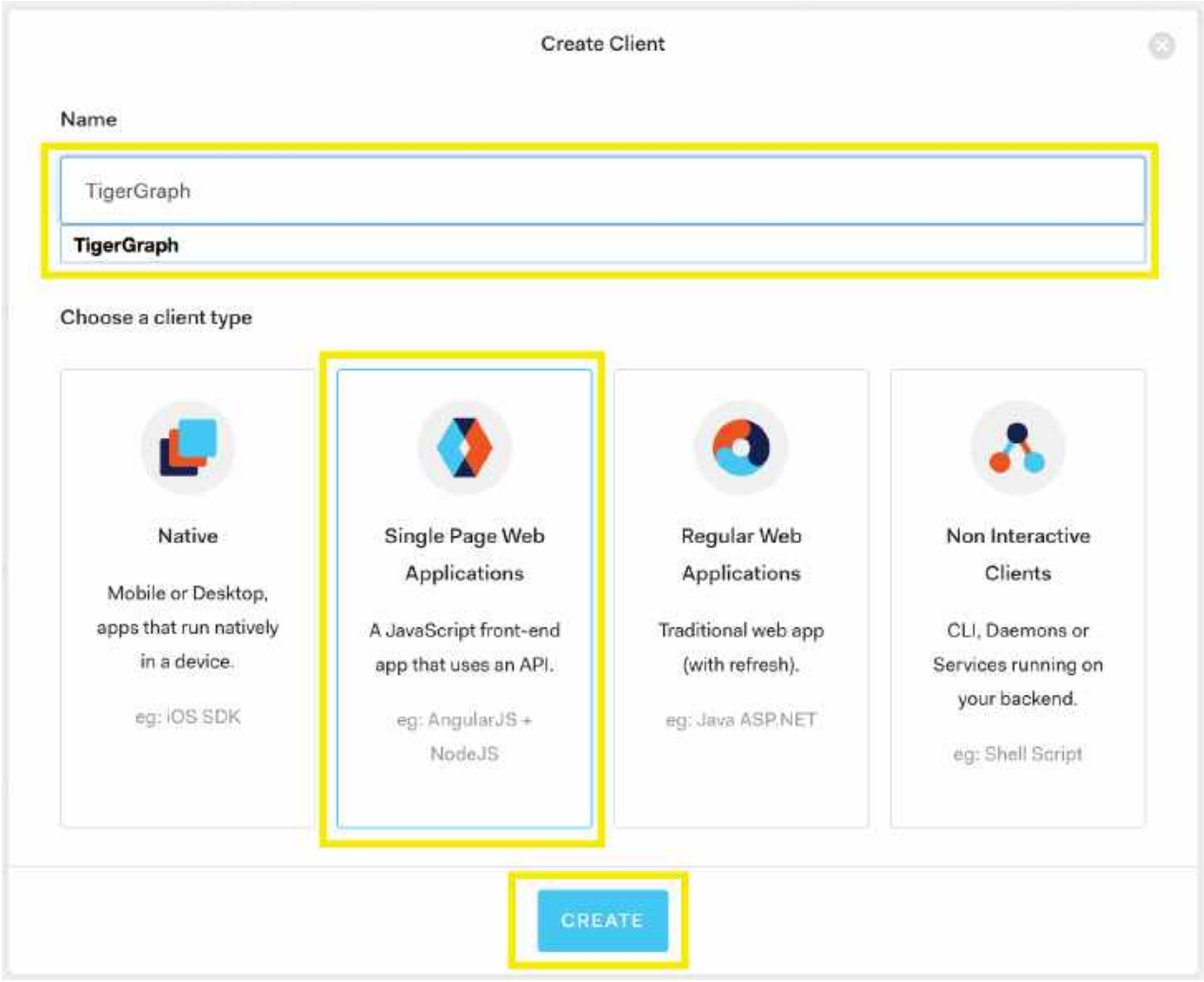


Auth0

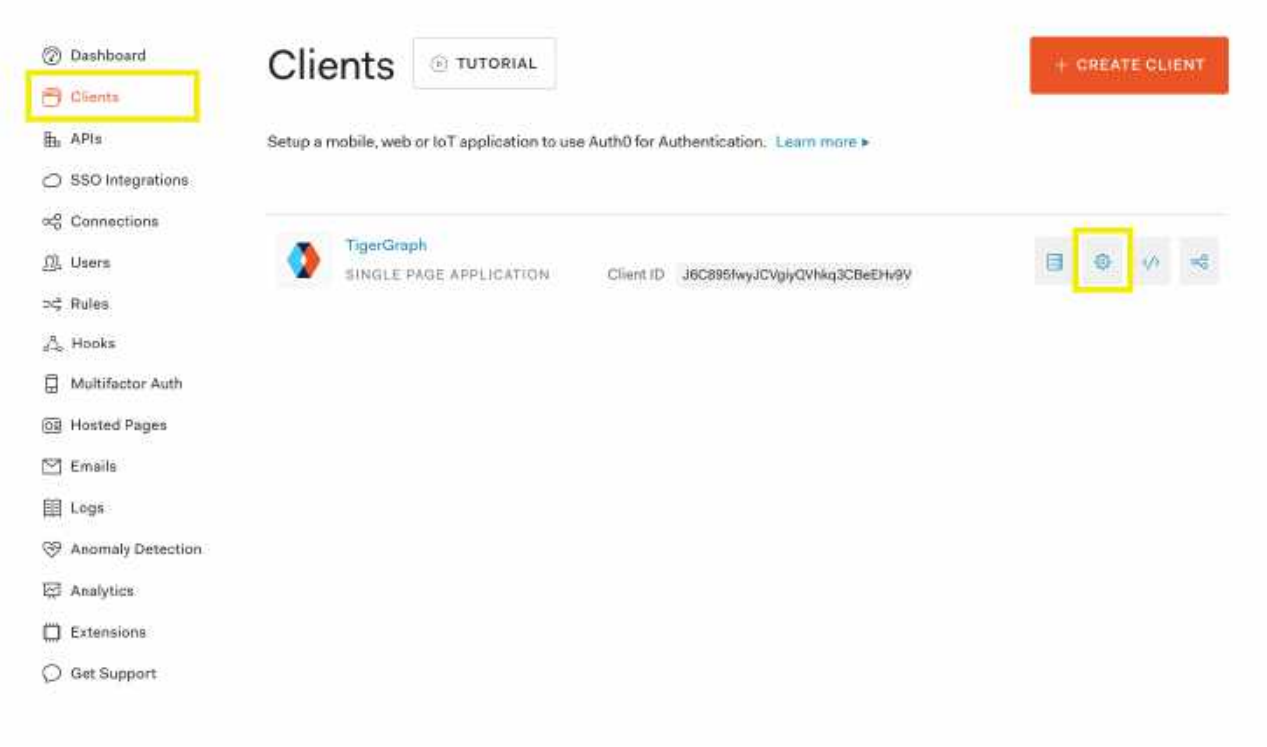
After logging into Auth0, click Clients in the left navigation bar, and then click CREATE CLIENT button.



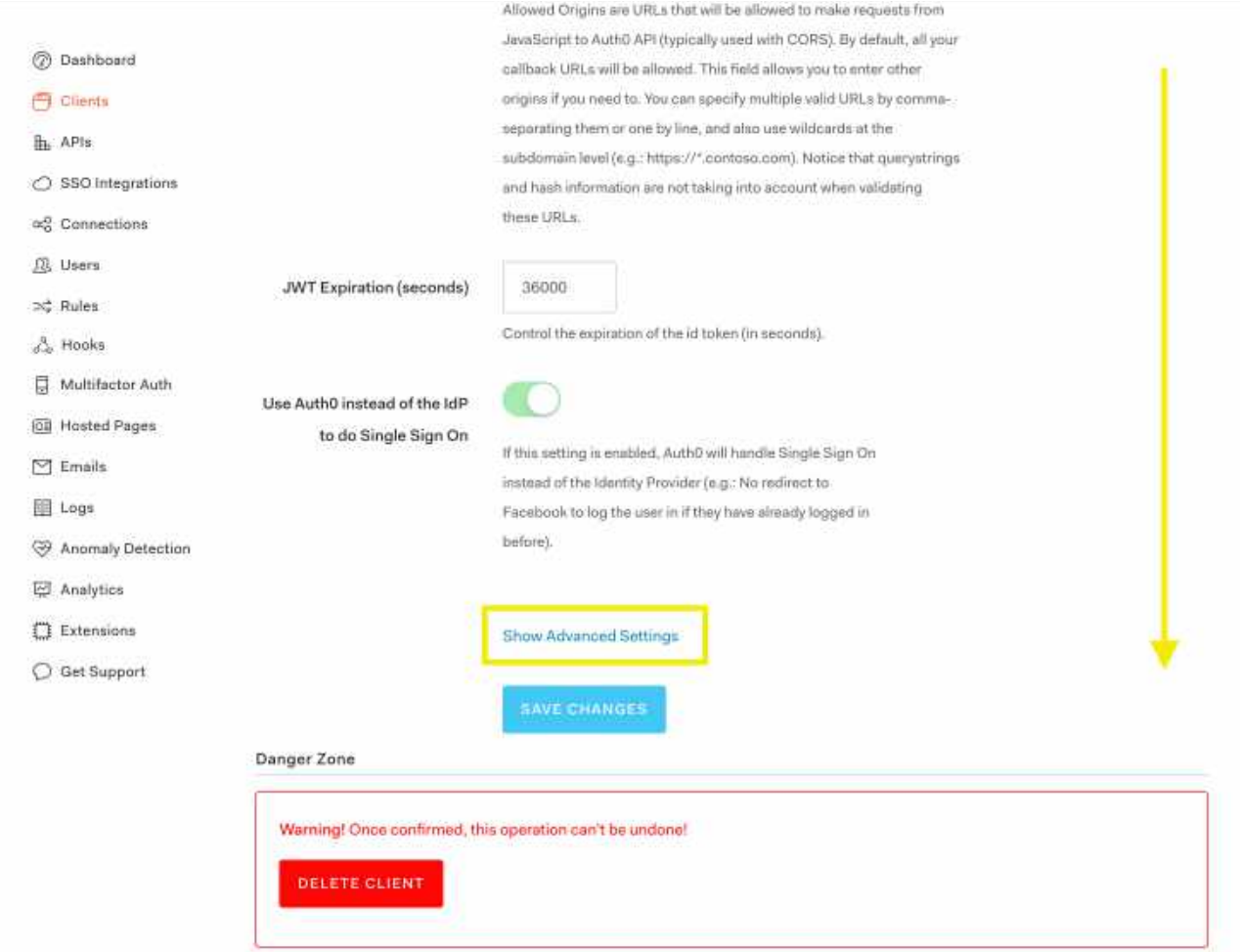
In the pop-up window, enter TigerGraph (or whatever application name you want to use) in the Name input box. Choose Single Page Web Application , and then click the CREATE button.



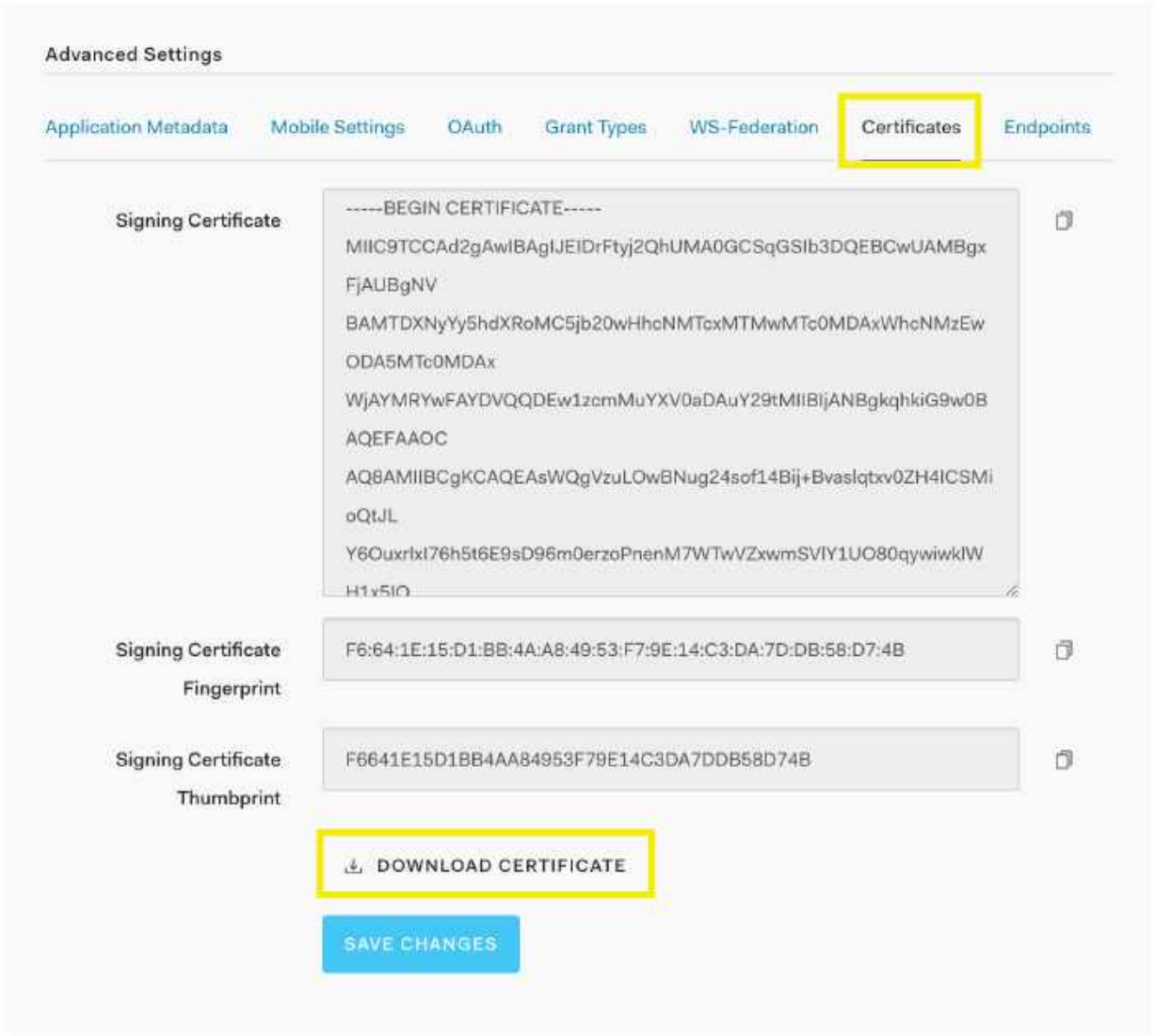
Click Clients again. In the Shown Clients list, click the settings icon of your newly created TigerGraph client.



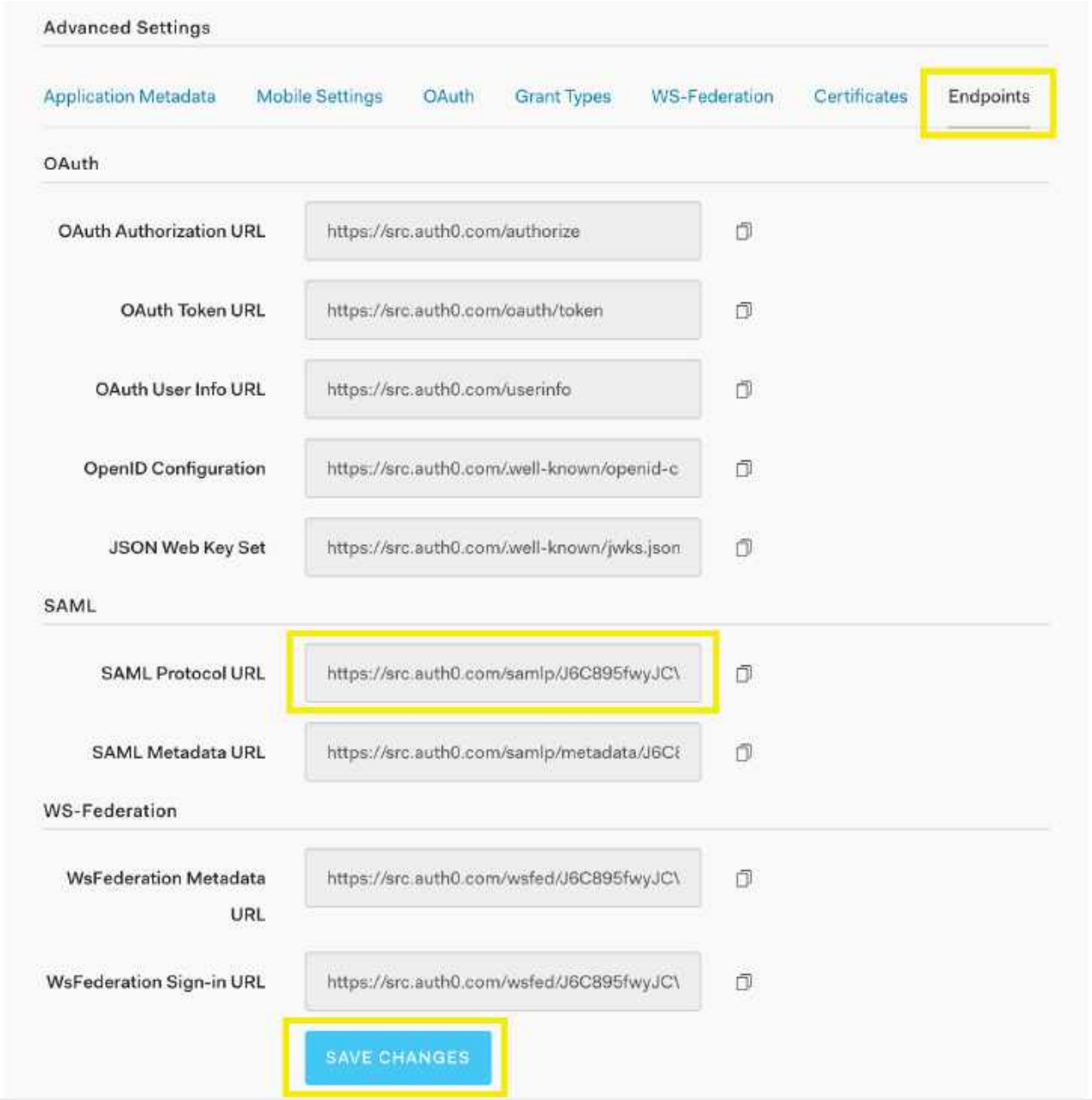
Scroll down to the bottom of the settings section, and click Show Advanced Settings .



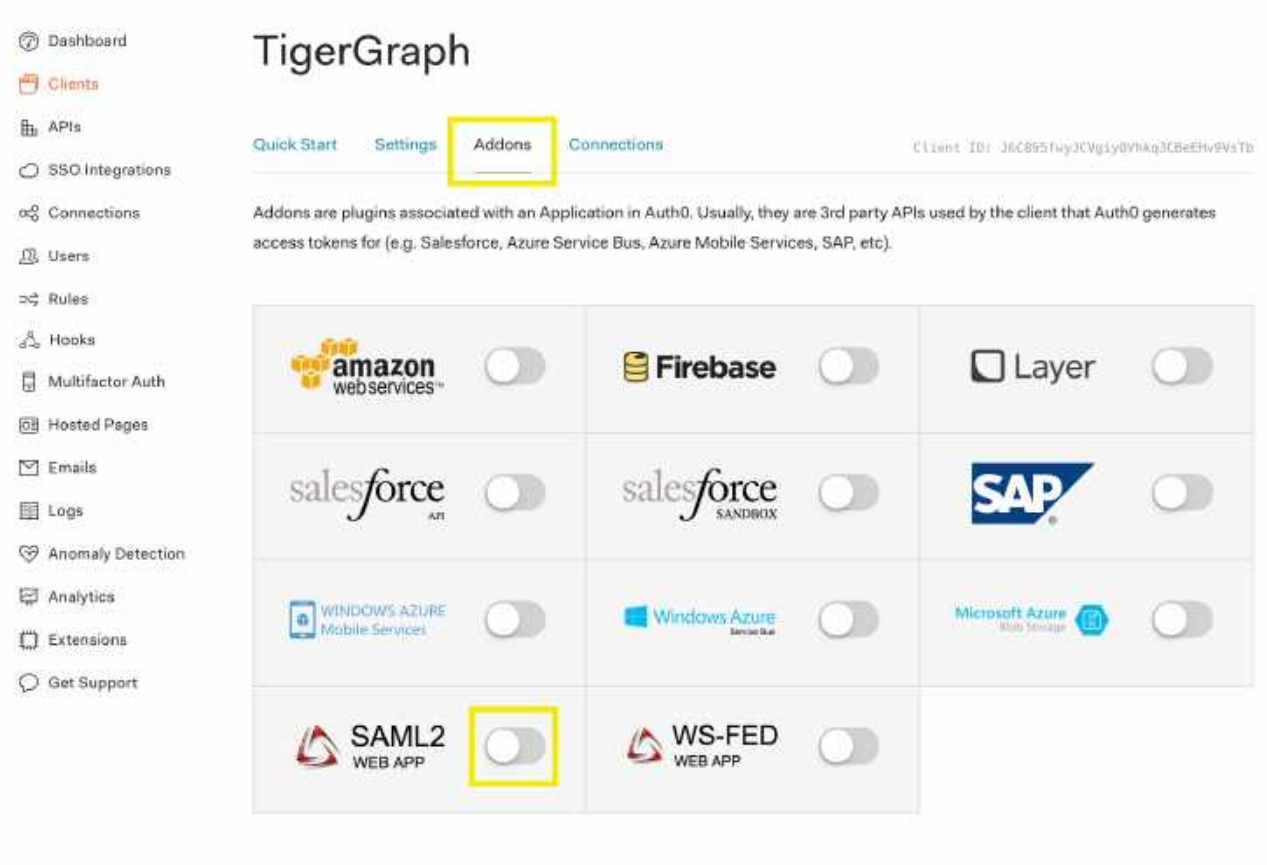
Click the Certificates tab and then click DOWNLOAD CERTIFICATE. In the chooser list, choose CER. Rename the downloaded file as idp.cert , and put it somewhere on the TigerGraph machine. Let's assume you put it under your home folder: /home/tigergraph/idp.cert. If you installed TigerGraph in a cluster, 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.

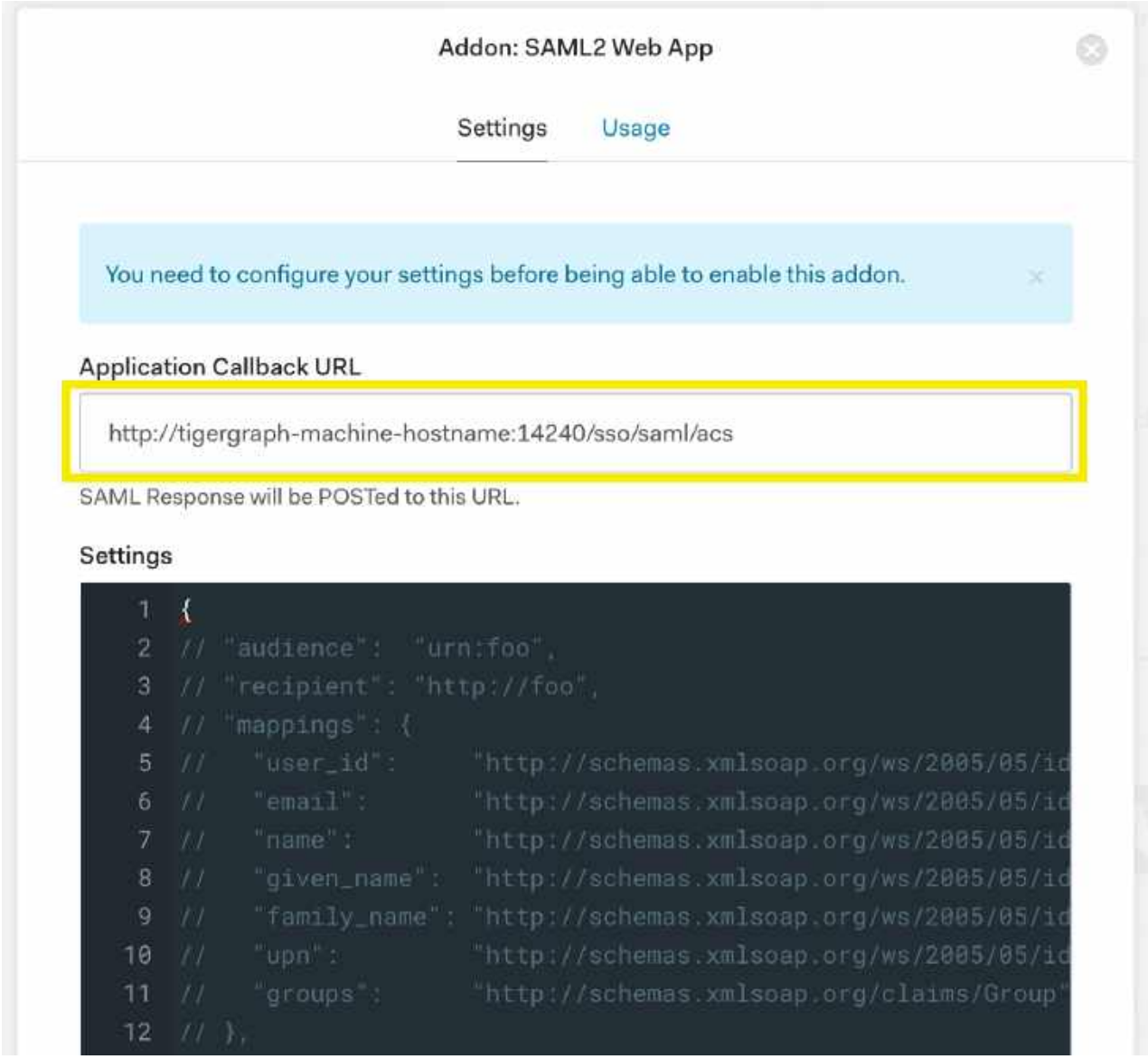


Scroll up to the top of the page, click the Addons tab, and switch on the toggle at the right side of the SAML2 card.

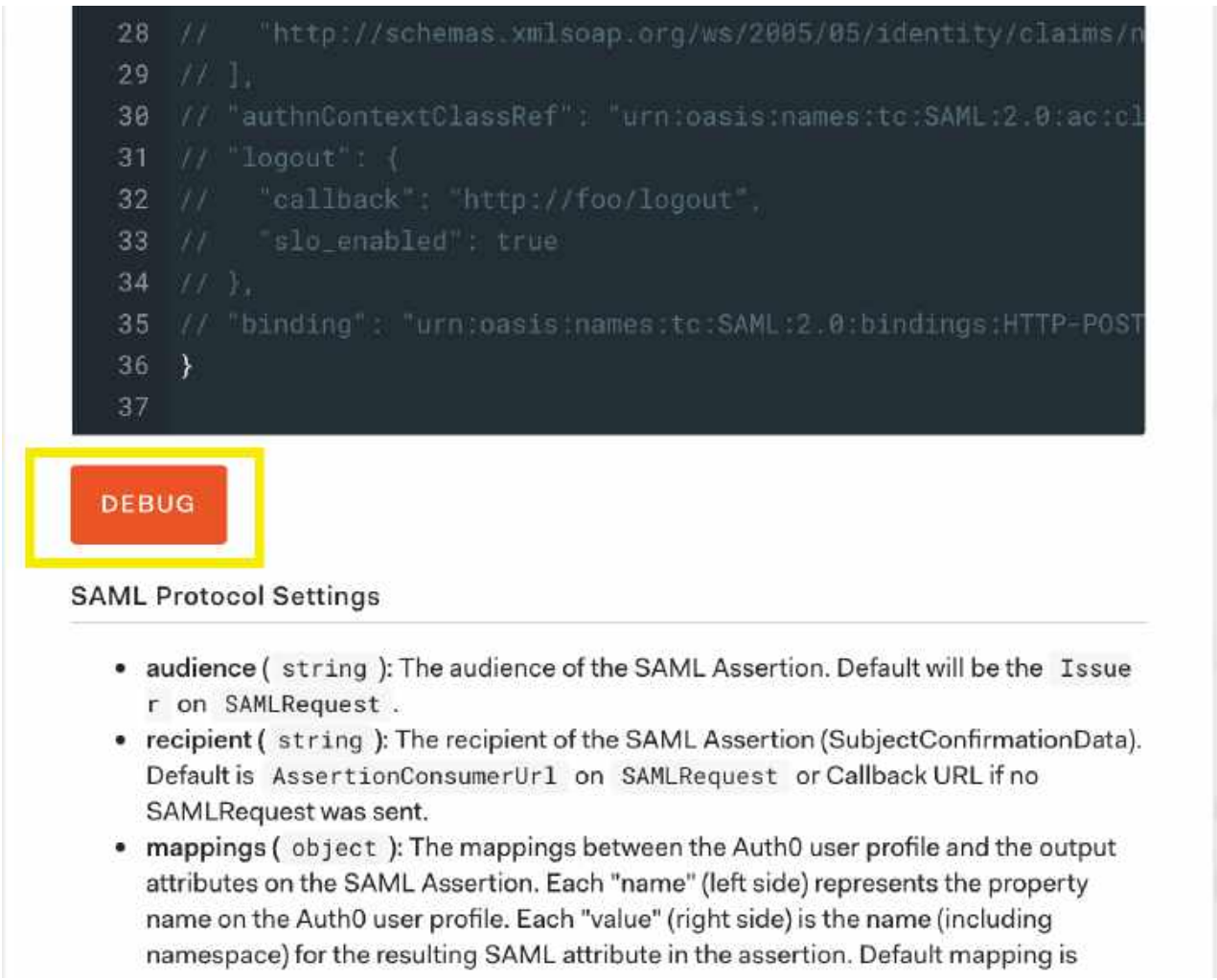


In the pop-up window, enter the Assertion Consumer Service URL in the Application Callback URL input box:

> http://tigergraph-machine-hostname:14240/sso/saml/acs

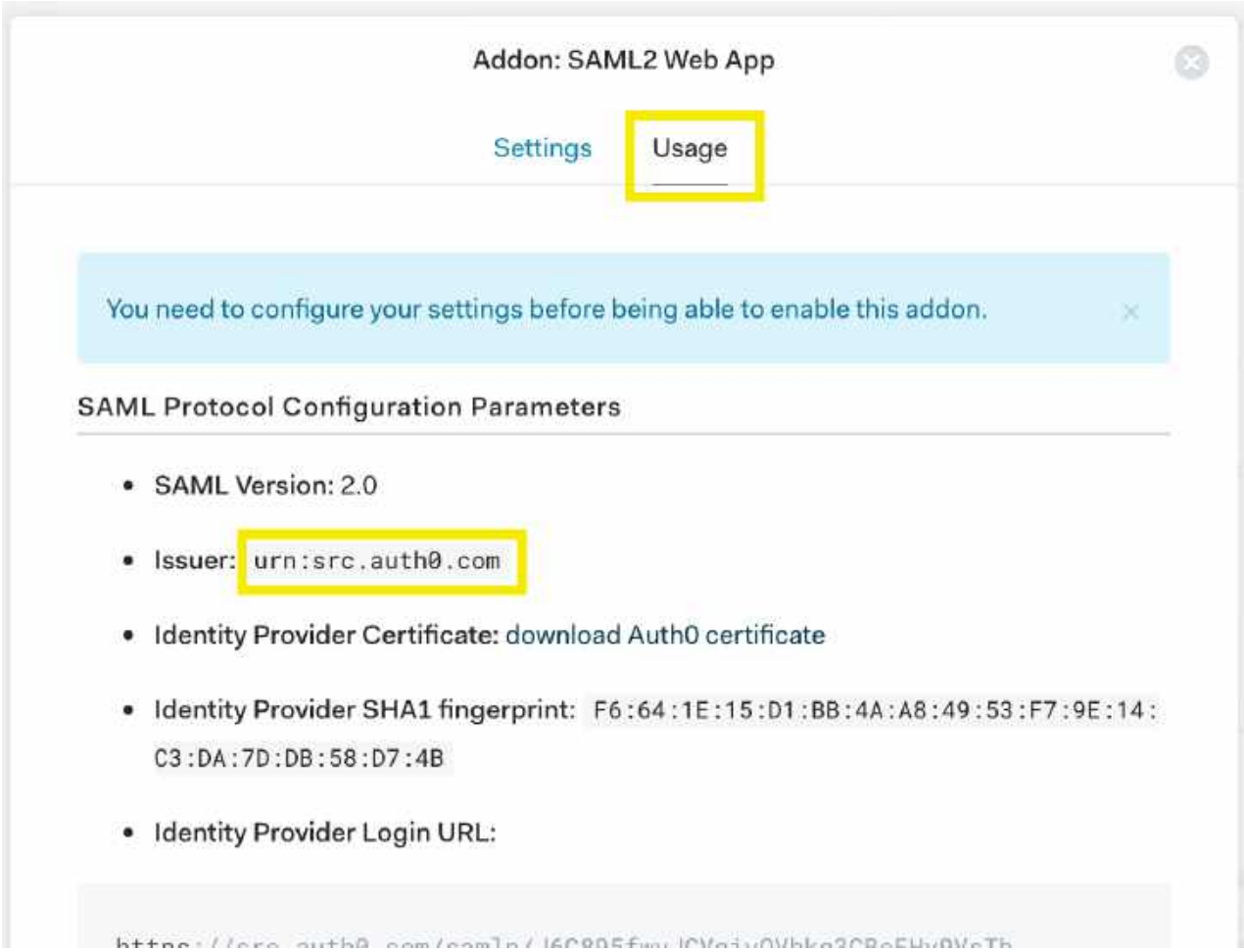


Scroll down to the end of the settings JSON code, click the DEBUG button, and log in as any existing user in your organization in the pop-up login page.

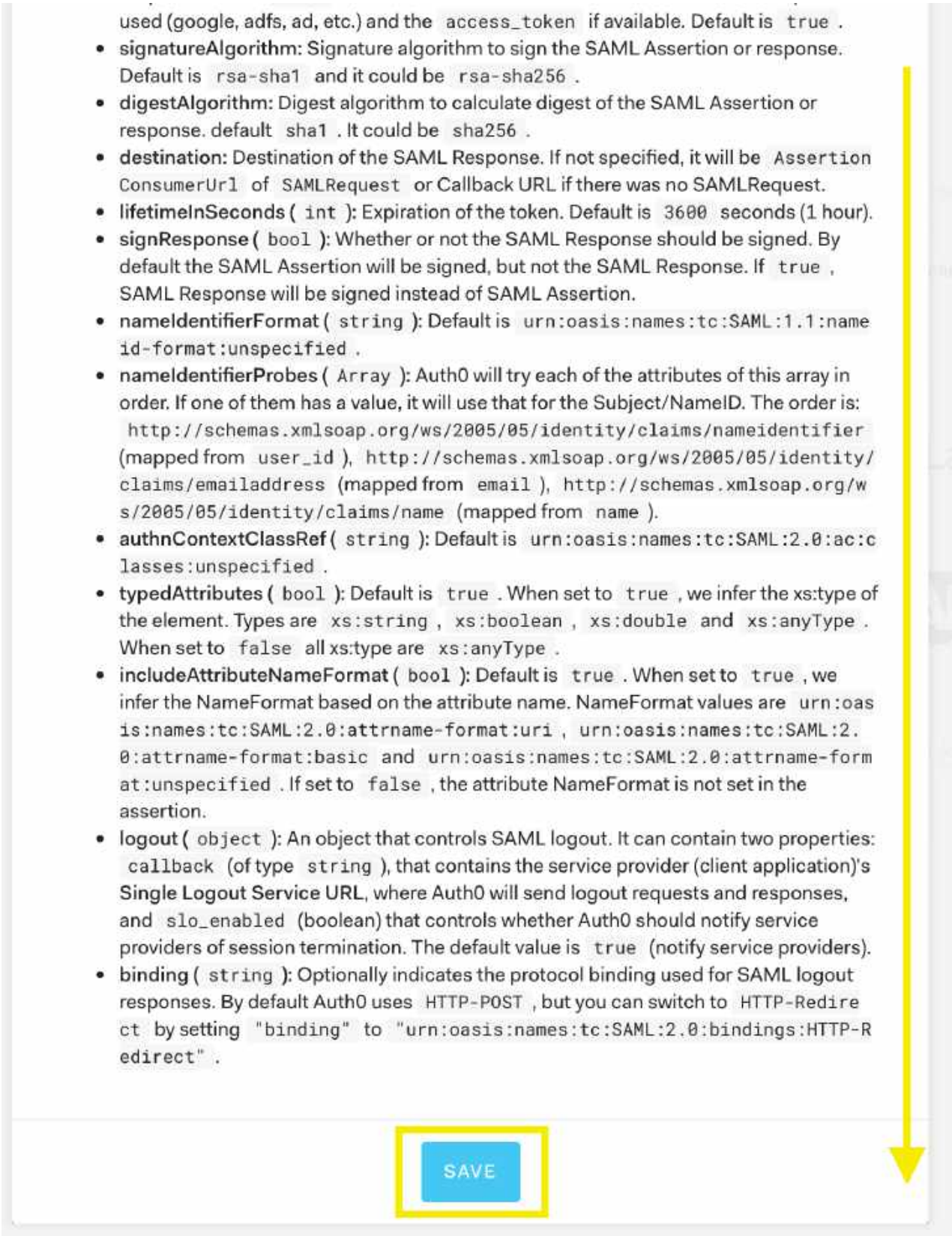


If login in successfully, the SAML response will be shown in decoded XML format. Scroll down to the attributes section. Here you will see some attribute names, which you will use to set proxy rules when creating groups in an upcoming configuration step.

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.



end of Auth0 configuration instructions. Jump to Step 2: [Enable Single Sign-On for TigerGraph↗]

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/tigerg

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 DN
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-hostnam
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.ke
```

Finally, change the certificate and private key file to have permission 600 or less. (The tigergraph user can read or write the file; no other user has any permission.)

```
chmod 600 /home/tigergraph/sp.*
```

Enable and configure Single Sign-On Using Gadmin

From a TigerGraph machine, run the following command: gadmin --configure sso.saml

Answering the questions is straightforward; an example is shown below.

> In v2.3, the requirements for the security.sso.saml.sp.url 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 --configure sso.saml
Enter new values or accept defaults in brackets with Enter.

Enable SAML2-based SSO: default false
security.sso.saml.enable [False]: true
True

TigerGraph Service Provider URL: default http://127.0.0.1:14240
security.sso.saml.sp.url [http://127.0.0.1:14240]: tigergraph-machine-grap
tigergraph-machine-graphstudio-url

Path to host machine's x509 Certificate filepath: default empty
security.sso.saml.sp.x509cert: /home/tigergraph/sp.cert
/home/tigergraph/sp.cert

Path to host machine's private key filepath. Require PKCS#8 format (start
security.sso.saml.sp.private_key: /home/tigergraph/sp.pem
/home/tigergraph/sp.pem

Identity Provider Entity ID: default http://idp.example.com
security.sso.saml.idp.entityid [http://idp.example.com]: http://identity.p
http://identity.provider.entity.id

Single Sign-On URL: default http://idp.example.com/sso/saml
security.sso.saml.idp.sso.url [http://idp.example.com/sso/saml]: http://id
http://identity.provider.single-sign-on.url

Identity Provider's x509 Certificate filepath: default empty
security.sso.saml.idp.x509cert: /home/tigergraph/idp.cert
/home/tigergraph/idp.cert

Sign AuthnRequests before sending to Identity Provider: default true
security.sso.saml.advanced.authn_request.signed [True]:
True

Require Identity Provider to sign assertions: default true
security.sso.saml.advanced.assertions.signed [True]:
True

Require Identity Provider to sign SAML responses: default true
security.sso.saml.advanced.responses.signed [True]: false
false

Sign Metadata: default true
security.sso.saml.advanced.metadata.signed [True]:
True

Signiture algorithm [rsa-sha1/rsa-sha256/rsa-sha384/rsa-sha512]: default r
```

```
security.sso.saml.advanced.signature_algorithm [rsa-sha256]:
rsa-sha256

Authentication context (comma separate multiple values)
security.sso.saml.advanced.requested_authn_context [urn:oasis:names:tc:SAM
urn:oasis:names:tc:SAML:2.0:ac:classes:Password

...

Test servers with supplied settings? [Y/n] y

...

Success. All settings are valid
Save settings? [y/N] y

...

Done.
```

The reason we change security.sso.saml.advanced.responses.signed 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
$ 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. The first type is nameid equations; the second type is 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.

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.

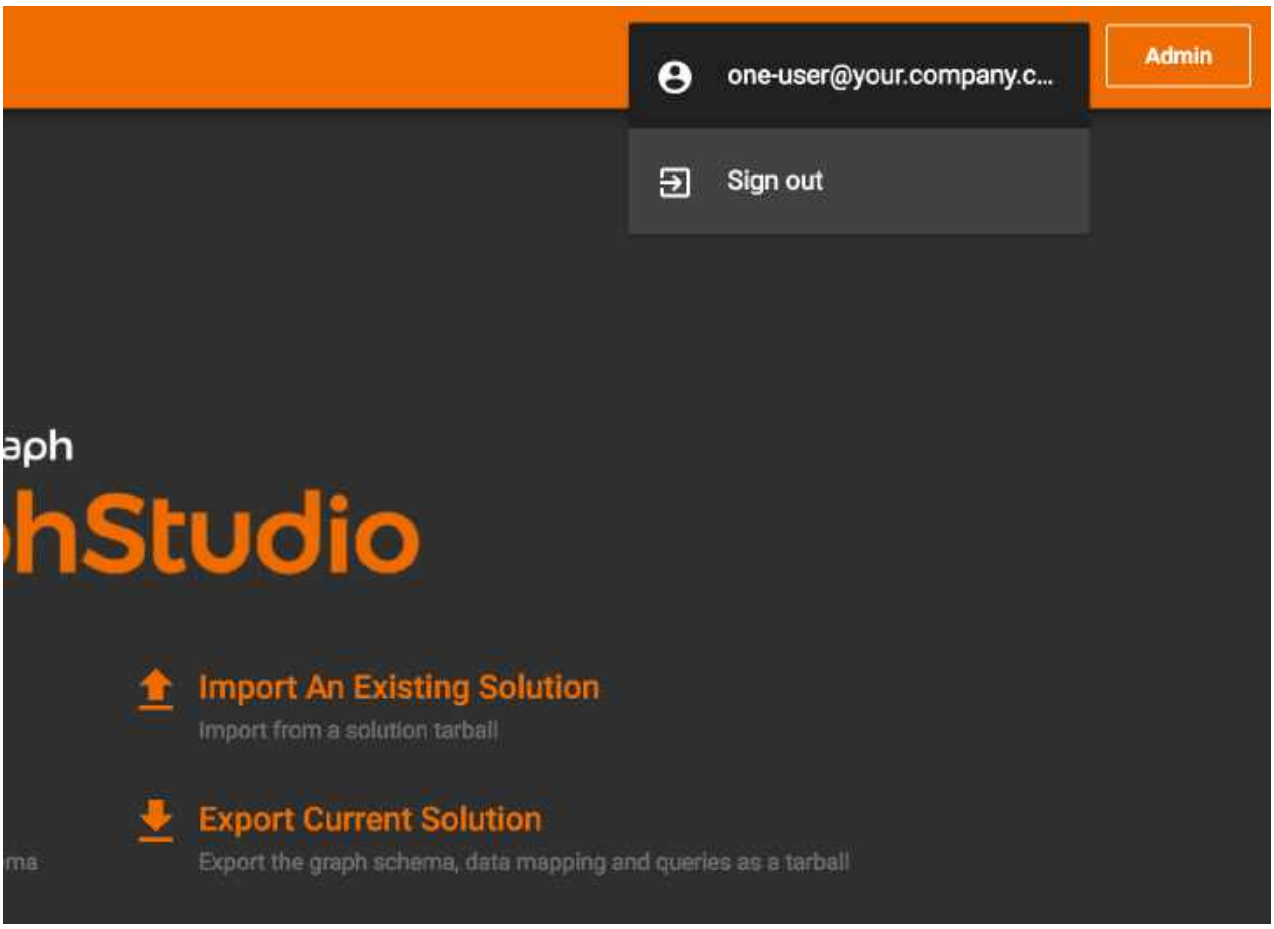
```
GSQL > change password
New Password : ********
Re-enter Password : ********
Password has been changed.
GSQL > exit
```

Testing Single Sign-On

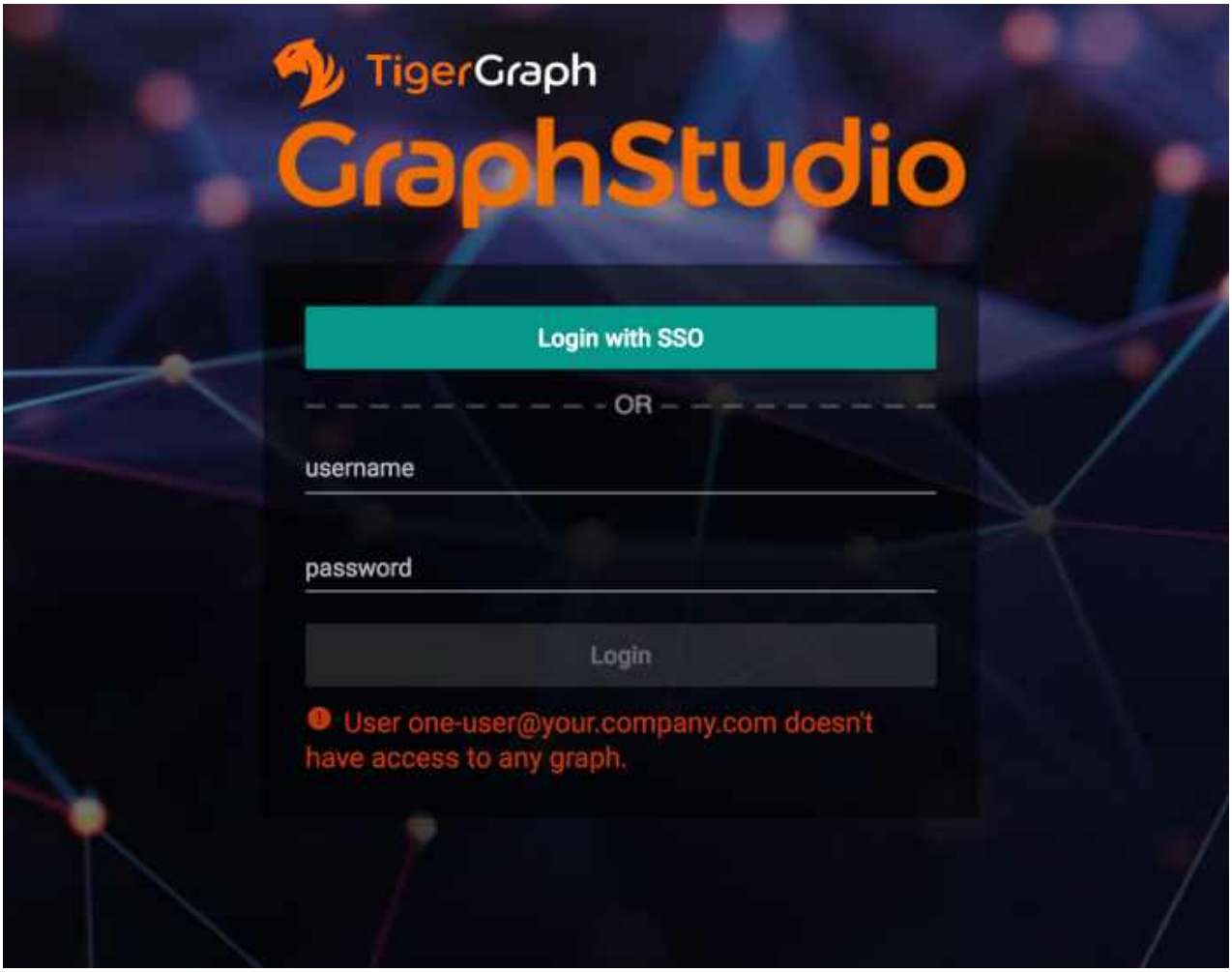
Now you have finished all configurations for Single Sign-On. Let's test it.

Visit the GraphStudio UI in your browser. You should see a Login with SSO button appear on top of the login panel:

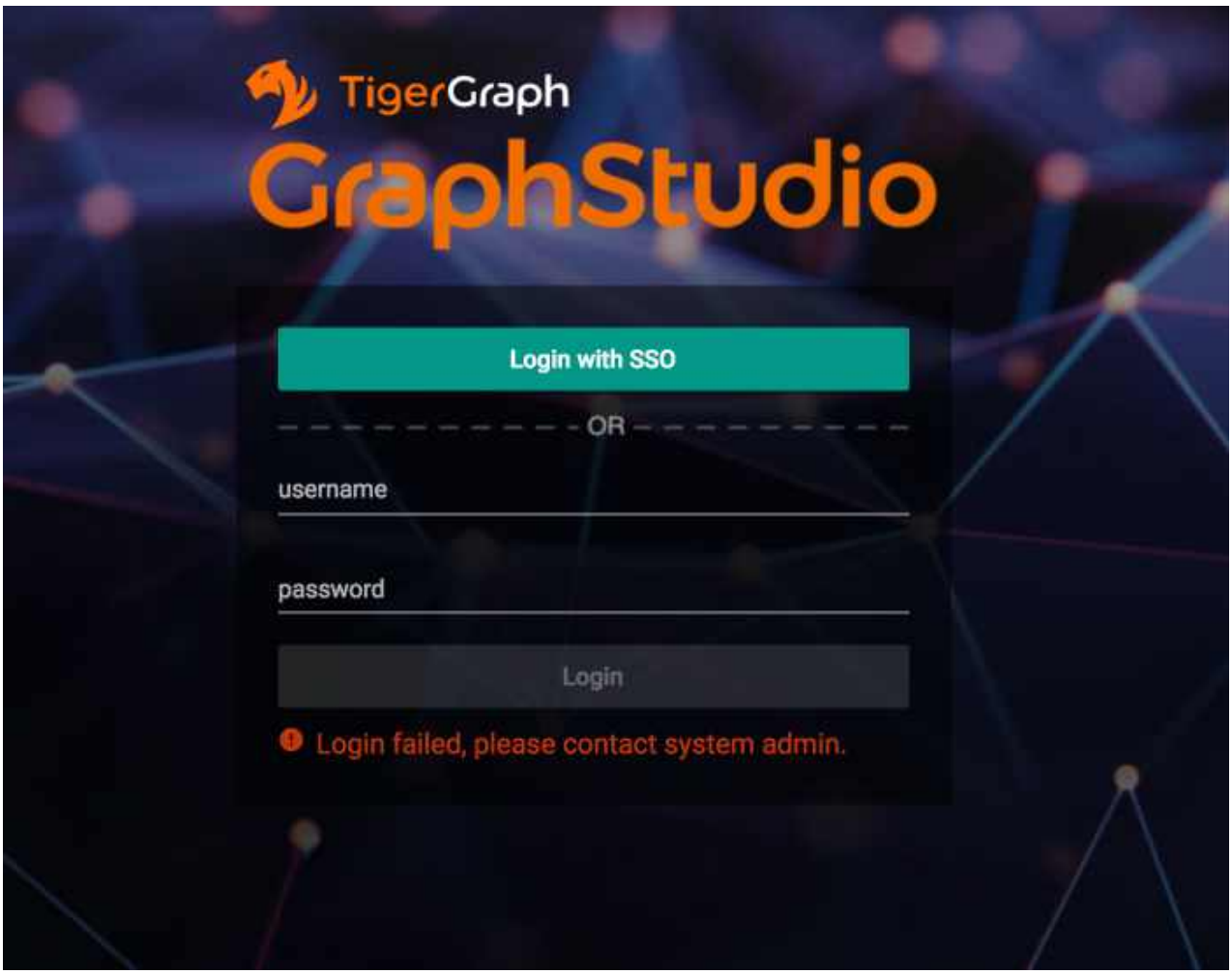
Clicking the button will navigate to your identity provider's login portal. If you have already logged in there, you will be redirected back to GraphStudio immediately. After about 10 seconds, the verification should finish, and you are authorized to use GraphStudio. If you haven't login at your identity 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 | grep GSQL_LOG
GSQL : /home/tigergraph/tigergraph/dev/gdk/gsql/logs/GSQL_LOG
```

Then, grep the SAML authentication-related logs:

```
cat /home/tigergraph/tigergraph/dev/gdk/gsql/logs/GSQL_LOG | 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 ↗.

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 a SSL certificate is bound to a domain
3. How a 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 a SSL Certificate

The two main options for obtaining a 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 use a configure your TigerGraph system to use the certificate to enable SSL.

Option 1: Using a Certificate From A Trusted Agent

First, obtain a 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 --configure 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-seli

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 DN
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.

```
$ gadmin --configure ssl

Enter new values or accept defaults in brackets with Enter.

Enable SSL with all HTTP responses (SSL Cert required): default False
Nginx.SSL.Enable [False]: True
True

Path to SSL cert bundle (domain cert, intermediate cert and root cert)
Nginx.SSL.Cert []: /home/tigergraph/nginx-selfsigned.crt
/home/tigergraph/nginx-selfsigned.crt

Path to SSL key
Nginx.SSL.Key []: /home/tigergraph/nginx-selfsigned.key
/home/tigergraph/nginx-selfsigned.key
...
Test servers with supplied settings? [Y/n] Y
...
Success. All settings are valid
Save settings? [y/N] y
```

After saving the settings, apply the configuration settings.

```
$ gadmin config-apply

[FAB ][2017-12-12 18:48:16] check_config
[FAB ][2017-12-12 18:48:16] update_config_all
Local config modification Found, will restart dict server and update confi
[FAB ][2017-12-12 18:48:21] launch_zookeepers
[FAB ][2017-12-12 18:48:31] gsql_mon_alert_on
[FAB ][2017-12-12 18:48:31] launch_zookeepers
[FAB ][2017-12-12 18:48:42] launch_gsql_subsystems:DICT
[FAB ][2017-12-12 18:48:42] gsql_mon_alert_on
Local config modification sync to dictionary successfully!
```

Then restart the external-facing services: gsql, nginx, and vis.

```
$ gadmin restart gsql nginx vis -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:14

<!doctype html><html lang="en"><head><meta charset="utf-8"><title>GraphStu
```

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 -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 exa
export encrypted_file_path='<path-to-encrypted-file>'

# The size of encrypted file to be created (used by dd command), for examp
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_root="<tigergraph-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_root/gstore /mnt/secretfs/gstore
ln -s /mnt/secretfs/gstore $tigergraph_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. In this case, simply move $tigergraph_root instead of $tigergraph_root/gstore.

```
mv $tigergraph_root /mnt/secretfs/tigergraph
ln -s /mnt/secretfs/tigergraph $tigergraph_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

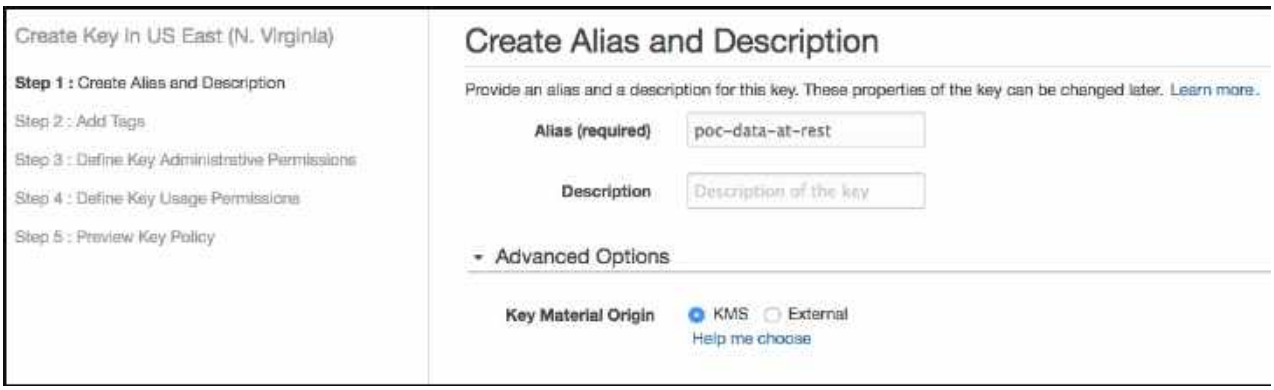
from Amazon Data-at-Rest blog

```
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>/LuksInternalStora
        }
    ]
}
The preceding policy grants read access to the bucket where the encrypted
(The following instructions have been updated since the original blog post

2."Select type of trusted entity: Choose AWS service .
3."Select the service that will use this role": Choose EC2 then choose Nex
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 c
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

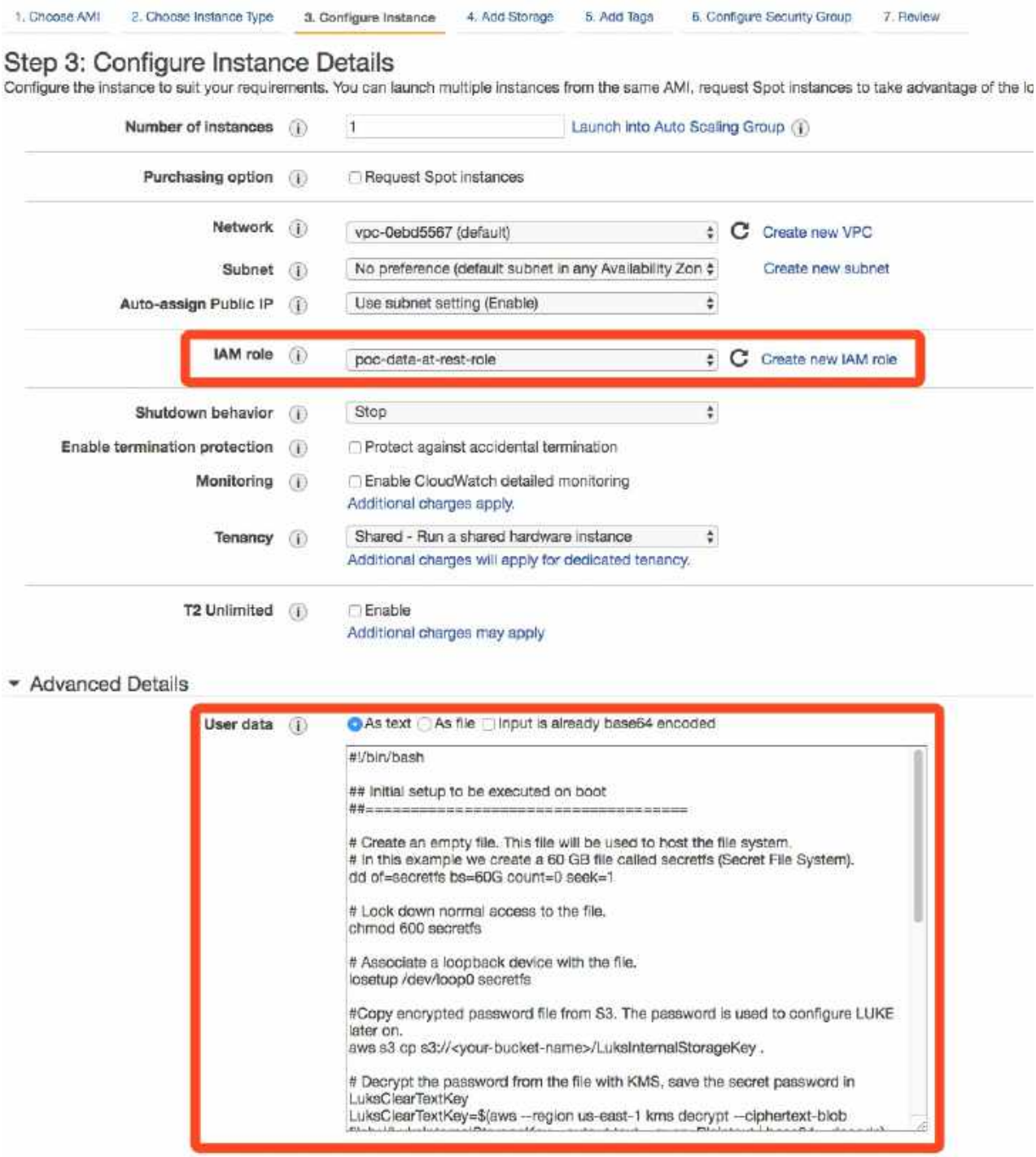
from Amazon Data-at-Rest blog

```
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>/LuksInternalStora
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 <path
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 l
aws s3 cp s3://<your-bucket-name>/LuksInternalStorageKey .

# Decrypt the password from the file with KMS, save the secret password in
LuksClearTextKey=$(aws --region <your-region> kms decrypt --ciphertext-blo

# 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_gstor
echo "$LuksClearTextKey" | cryptsetup luksOpen /dev/loop0 tigergraph_gstor

# 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 sys
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 under
```

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

Admin Portal Guide

TigerGraph Admin Portal UI Guide

Overview

The TigerGraph Admin Portal is a browser-based dashboard which 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.

As of June 2018, the Admin Portal is certified on following browsers:

Browser	Chrome	Safari	Firefox	Opera
Supported version	54.0+	11.1+	59.0+	52.0+

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 one of the servers for your TigerGraph system, then you can use `localhost` for your <tigergraph_server_ip_address>. The Admin Portal is on the admin page:

```
http://<tigergraph_server_ip_address>:14240/admin/
```

If user authentication has been enabled, then users need to log in to access the

Admin Portal.

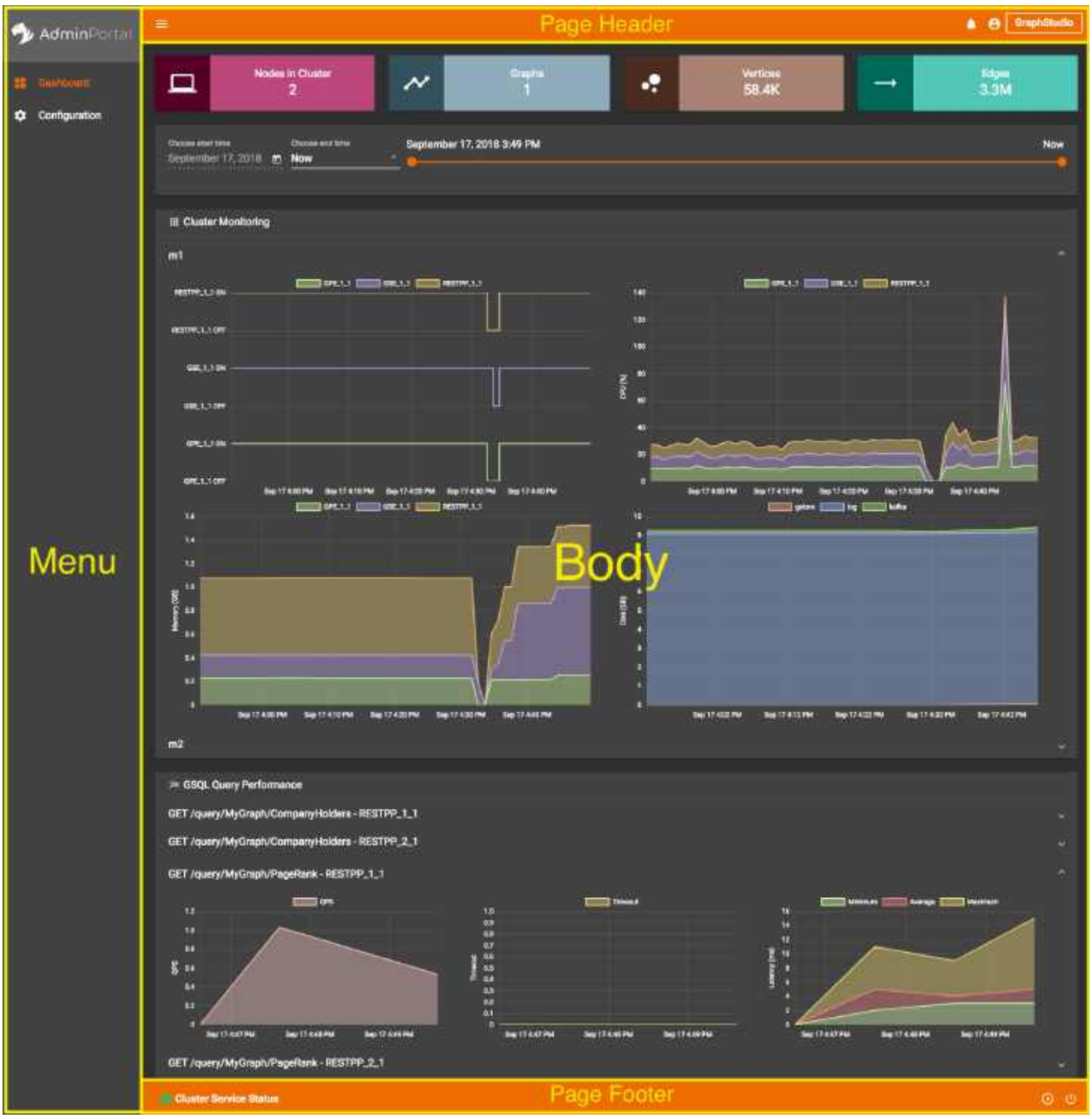
If you are already at GraphStudio, simply click the Admin button at the right end of the top menu bar.

Page Layout

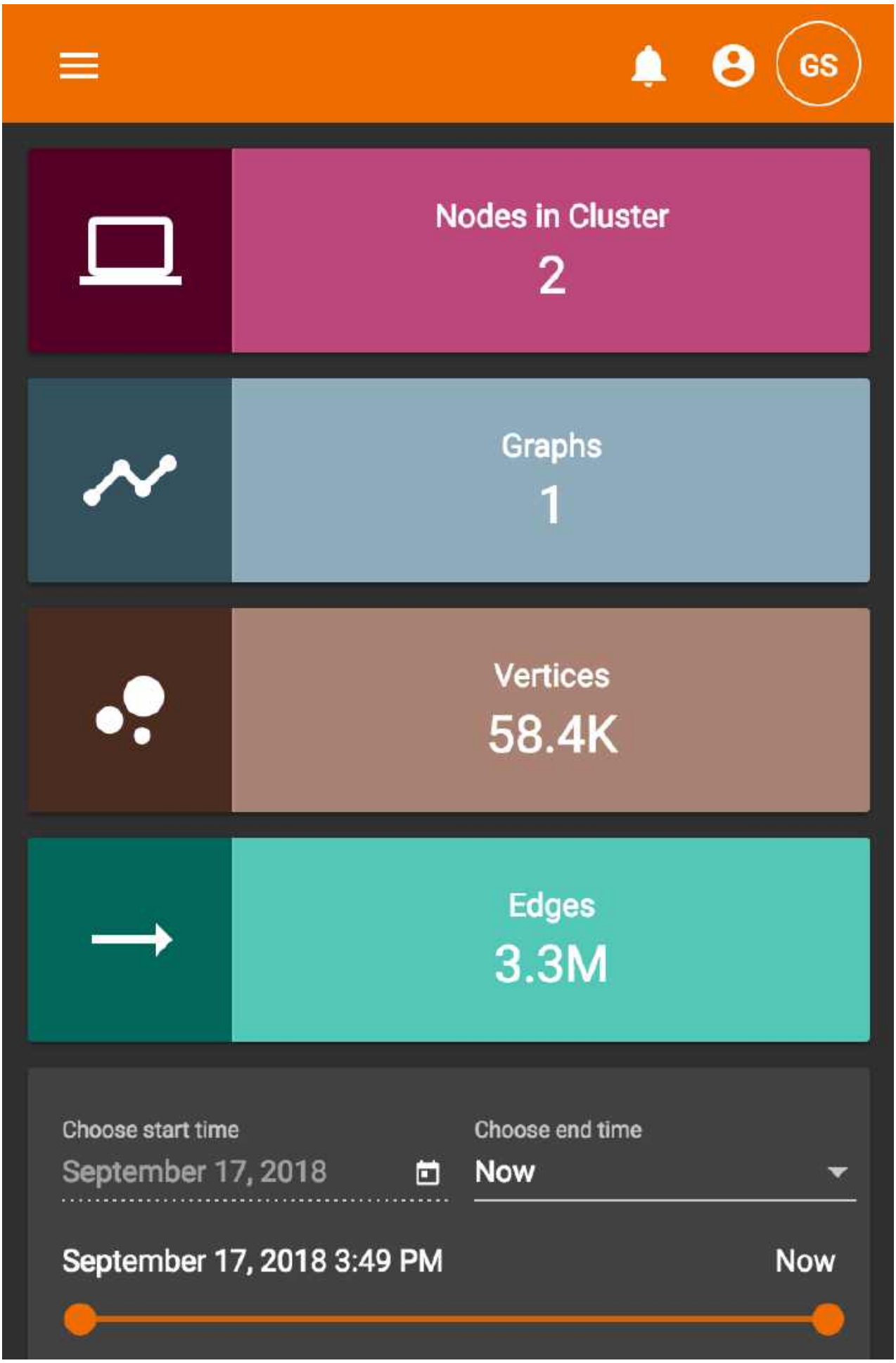
The Admin Portal has two pages: Dashboard and Configuration . Both pages have the same Header, Footer, and Navigation Menu.

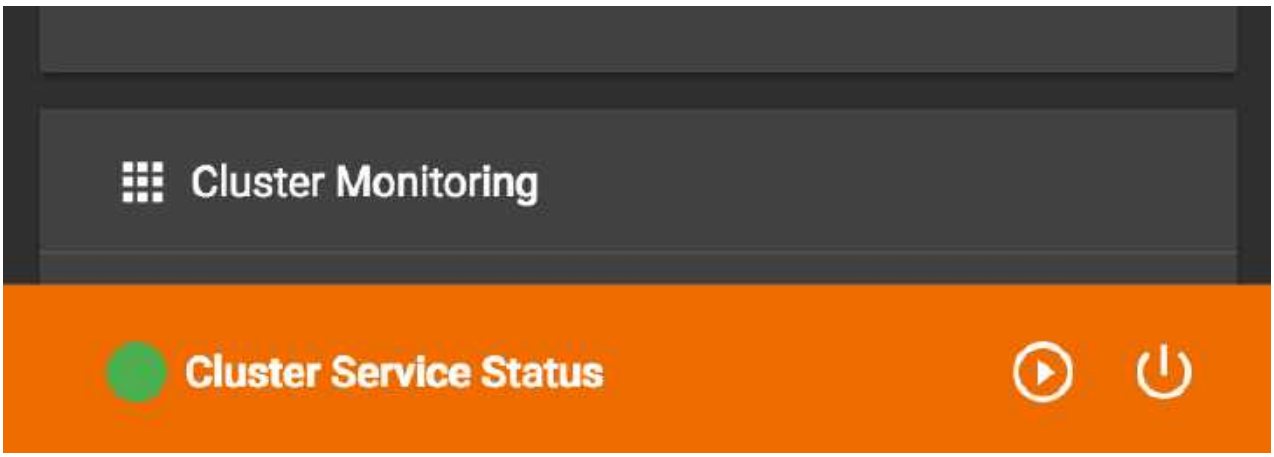
> The layout of the Admin Portal is responsive to screen size. The layout will automatically adjust for devices with small screens like phones and tablets.

The full screen version of the Admin Portal is shown below, with the Dashboard page selected.



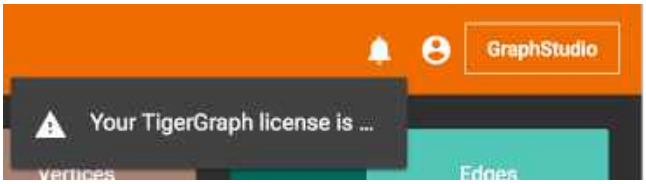
The mobile version is shown below:


GS
Nodes in Cluster
2
Graphs
1
Vertices
58.4K
Edges
3.3M
Choose start time
September 17, 2018
Choose end time
Now
September 17, 2018 3:49 PM
Now

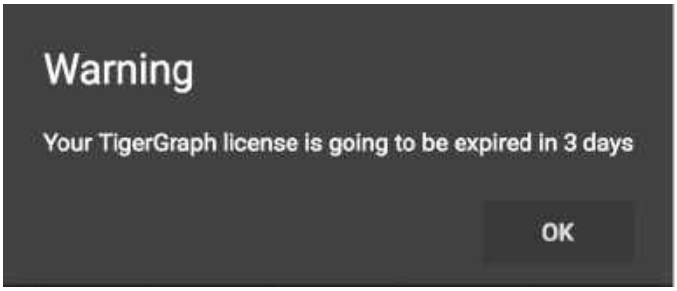


Page Header

Clicking on the **Notification** icon will open up a list of notifications. If a notification is too long, some of its content will be omitted:

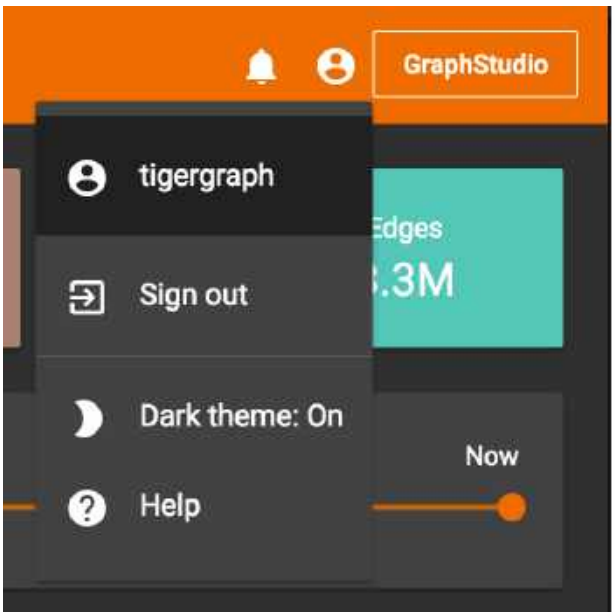


To view the full text, you can click on a notification to open a popup window containing the full message and its severity:

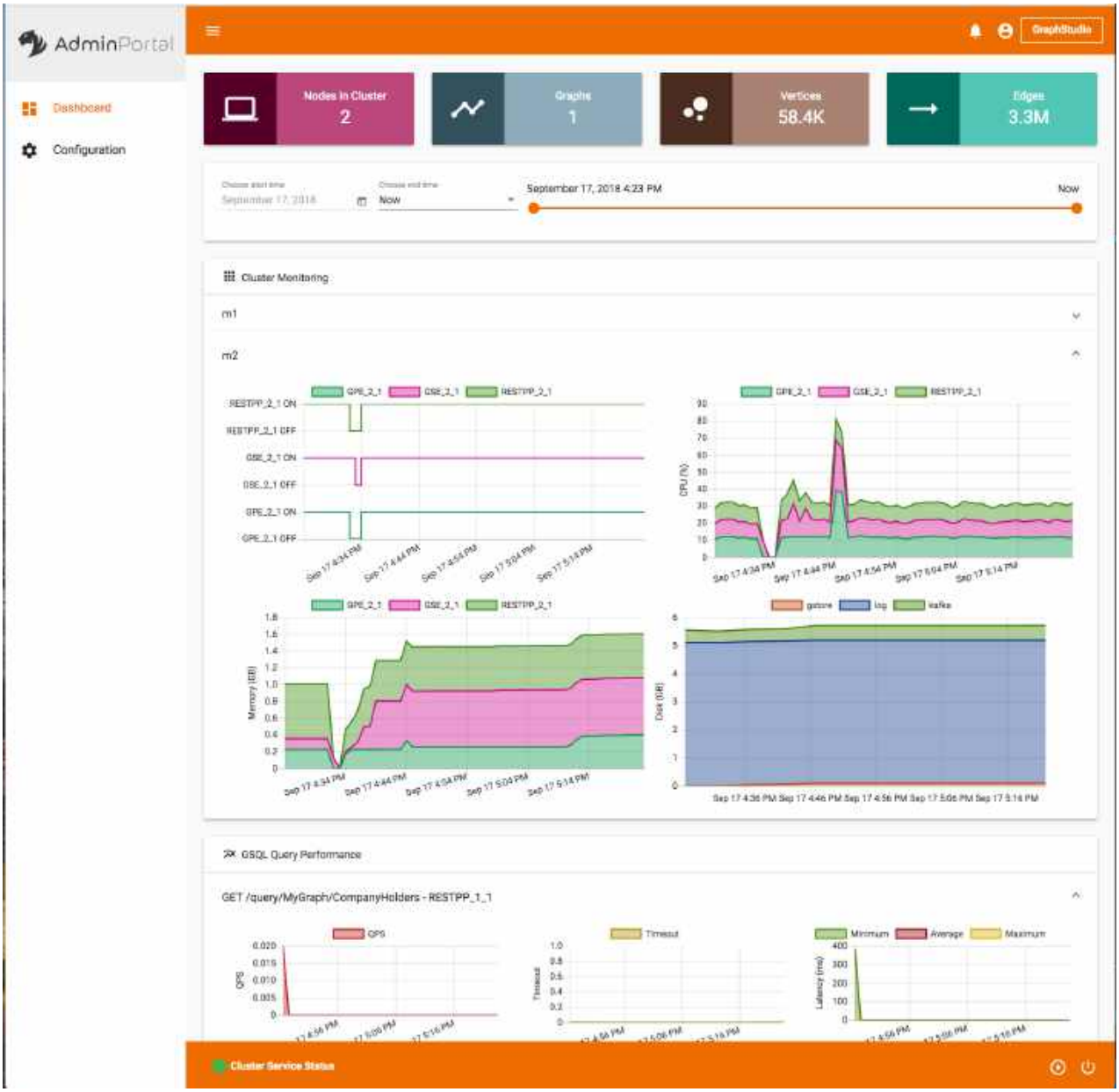


There are three severity levels: info, warning and error.

The **Account** icon will open the user menu:



You can switch between a dark theme and 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 GraphStudio.

Page Footer

The overall system status 

is always shown in the footer. This single indicator shows:

- Green indicates all services are online.
- Gray means one or more service statuses is unknown.
- Red means on of the component services is offline.

Clicking on the button will show you the list of statuses for the services in our system:

You can start or stop services from the Admin Portal by using the right most buttons  (NOTE: ONLY a superuser can see these buttons).

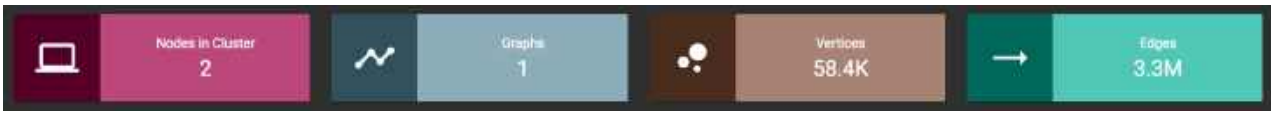
Clicking on the **Stop** icon  will stop all of the services in the TigerGraph system.

Clicking on the **Start** icon  will start all of the services in the TigerGraph system (NOTE: because there is an interval between data collection period, the real status of the system will not be reflected in the status section right away).

Dashboard Page

The Dashboard page has three main parts: Overall Statistics, the Time Range Picker, and several Charts.

Overall Cluster Statistics



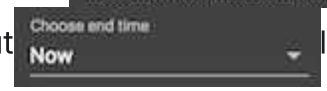
Just below the page header, there are four cards showing statistics of our system, including number of nodes, number of graphs, number of vertices and number of edges. These statistics are refreshed live. (The default refresh interval is 1 minute).

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 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.

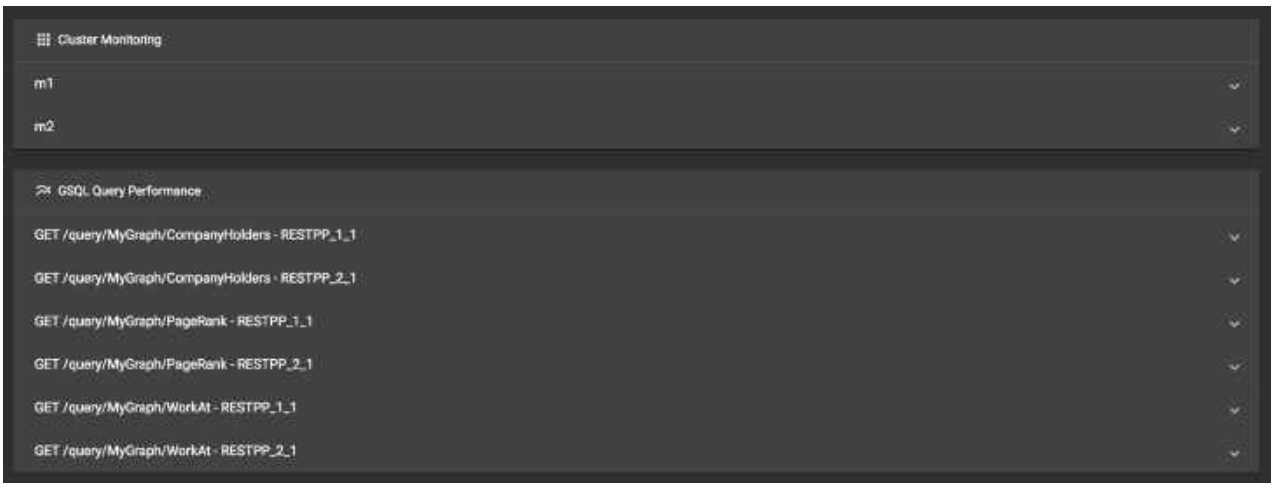
The sliding bar on the right lets you fine tune the range. Click and drag an endpoint to adjust the start or end time.

Changing any of these selections will trigger a request for statistics data and the chart will be re-rendered accordingly.

Charts

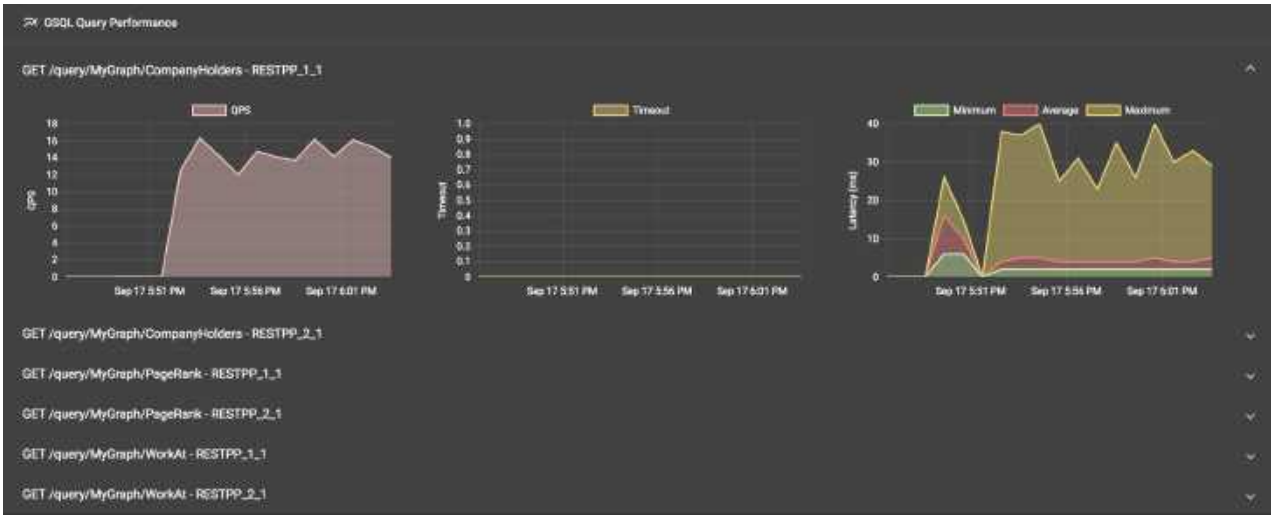
Each charts displays some statistic or state information on the vertical axis and time on the horizontal axis.

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.



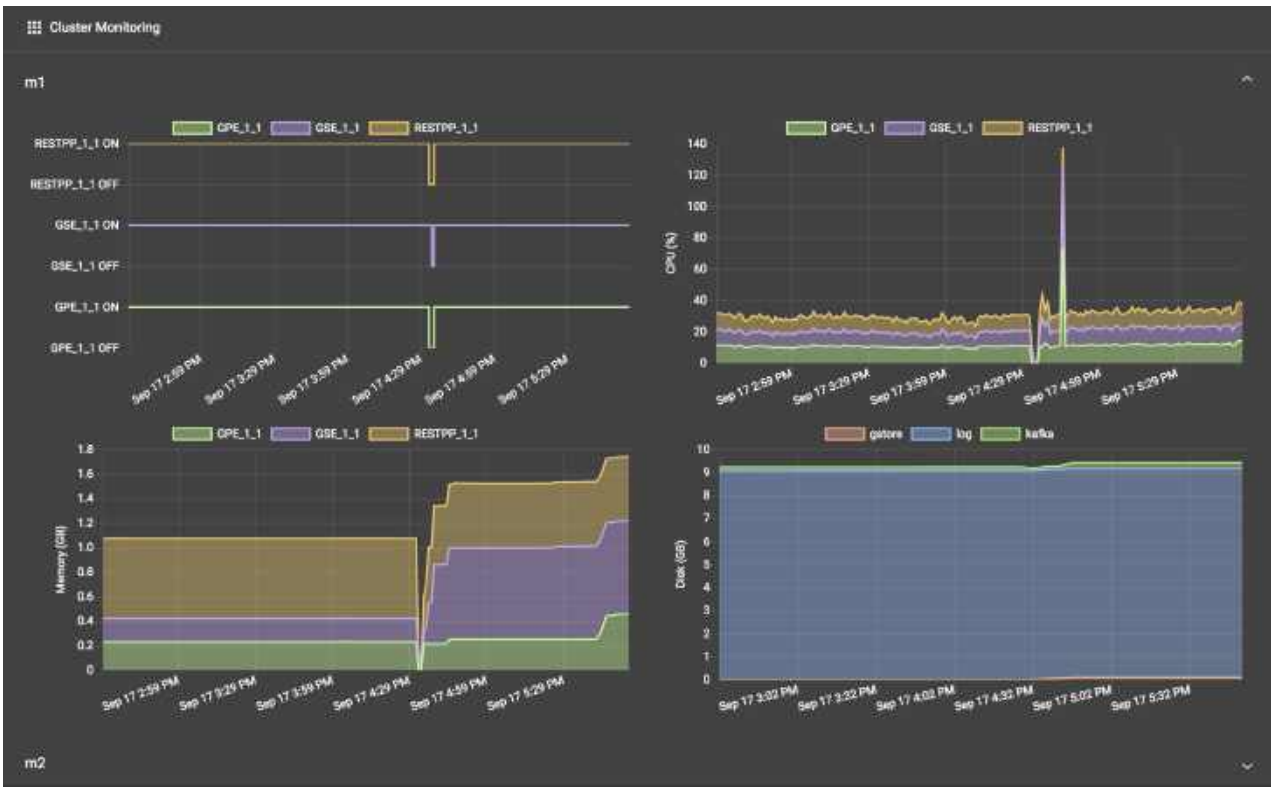
A Query Monitoring Panel includes three charts:

- 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).

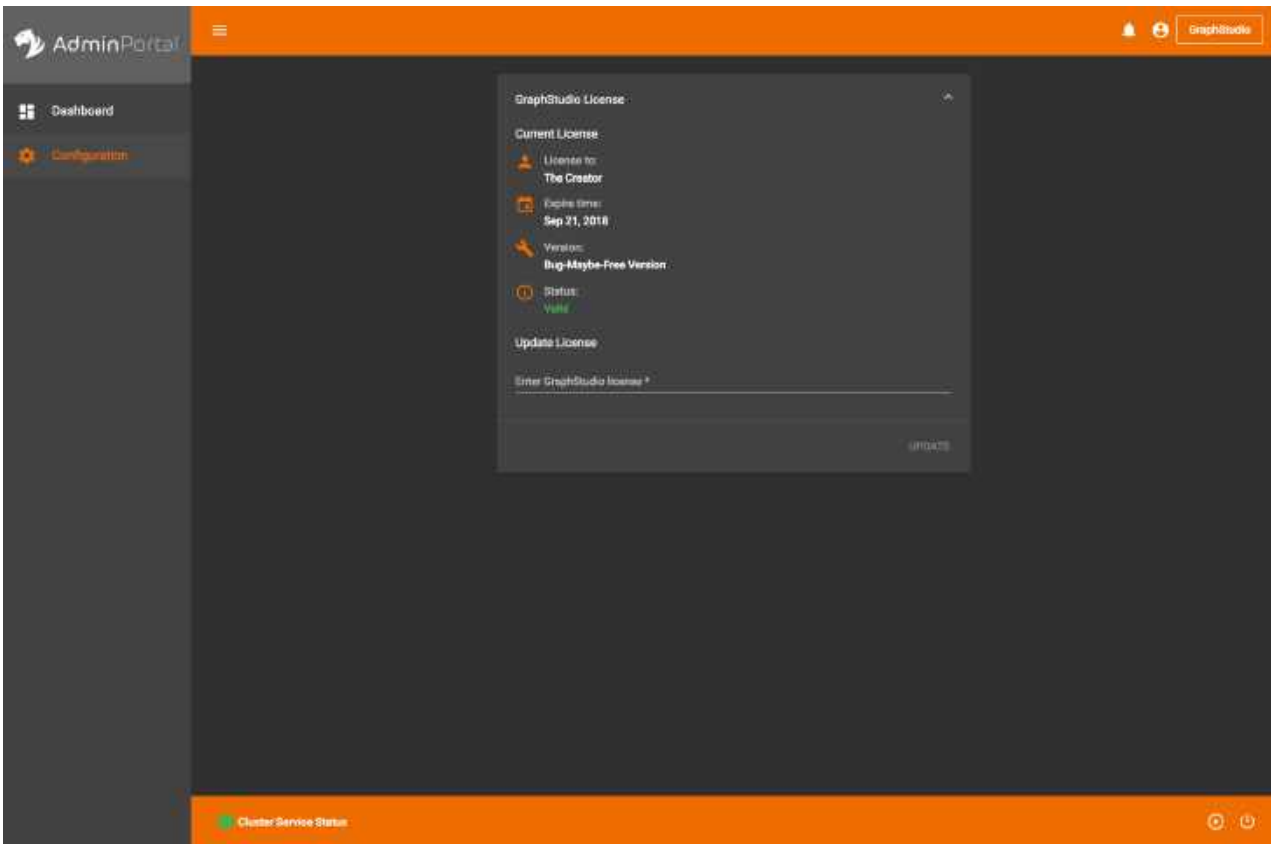
- Service status: ON or OFF status for the given component
- CPU Usage: percentage of available CPU time used by the given component
- Memory Usage: GB used by the given component
- Disk Usage: GB used by the given component



Configuration Page

Currently (as of v2.2), the Configuration page supports one configuration operation: updating the GraphStudio license key.

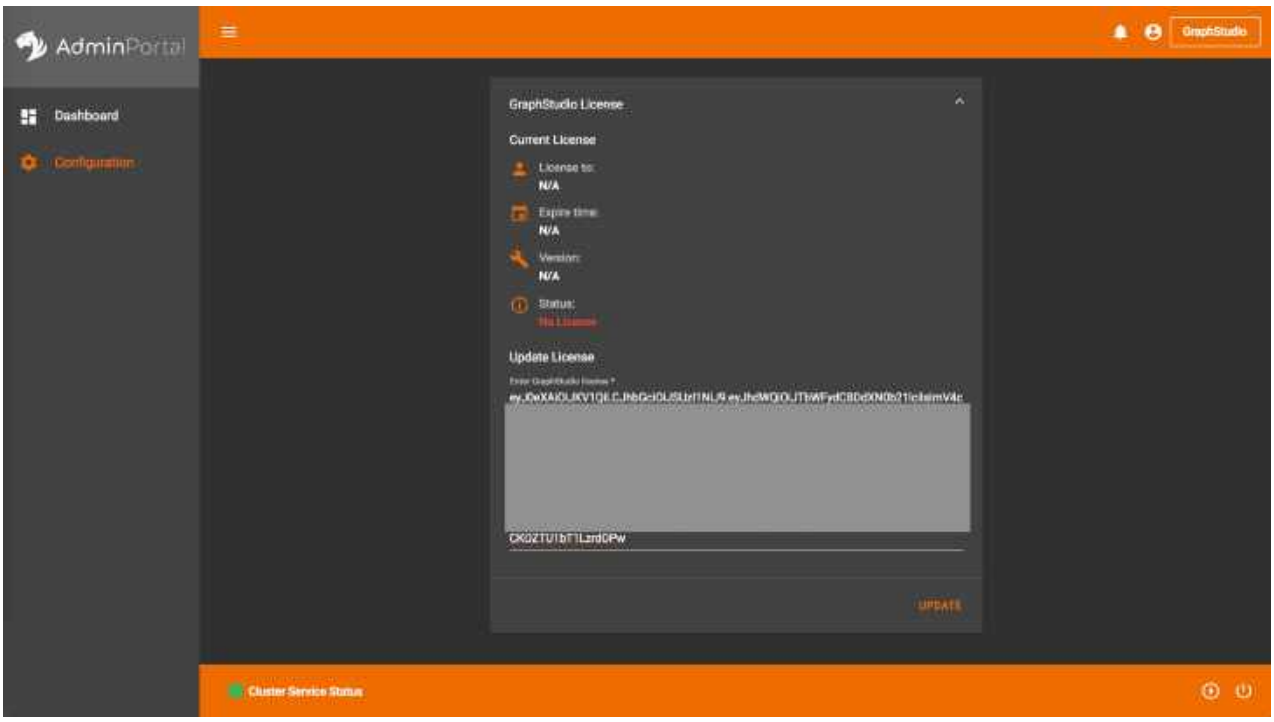
Additional configuration operations, which are currently only available from a Linux console, will be added in future releases.



Update GraphStudio License

An example of the GraphStudio License Update panel is shown below. The panel displays the full information about your license, including the expiration date.

To apply a new license key, paste the key into the text box below "Enter GraphStudio license" and click **Update**.


AdminPortal
Dashboard
Configuration
GraphStudio
GraphStudio License
Current License
License to:
N/A
Expire time:
N/A
Version:
N/A
Status:
Update License
CKQZTU1bT1Lzrd0Pw
UPDATE
Cluster Service Status

Managing with gadmin

Managing TigerGraph Servers 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
$ man gadmin
$ info gadmin
```

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 config-apply

Command Listing

Below is the man page for gadmin. Most of the commands are self-explanatory.

```
GADMIN(1)                          User Commands

NAME
       gadmin - manual page for TigerGraph Administrator.

SYNOPSIS
       gadmin [options] COMMAND [parameters]

DESCRIPTION
       Version 1.0, Sept, 19, 2017

       gadmin is a tool for managing TigerGraph servers

OPTIONS
  -h, --help
    show this help message and exit

  --configure
    invoke interactive (re)configuration tool. Options:
      single_dir:/xxx/yyy(deploy directory will be /xxx/yyy),
      or a keyword(e.g., 'gadmin --configure port',
      will configure any entry whose name has string 'port')

  --set  set one configuration

  --dump-config
  dump current configuration after parsing config files and command line o

  --dry-run
    show what operation will be performed but don't actually do it

  -p SSH_PASSWORD, --password=SSH_PASSWORD
    the password to ssh to other nodes

  -y, --yes
    silently answer Yes to all prompts

  -v, --verbose
    enable verbose output

  --version
    show gadmin version and exit

  -f, --force
    execute without performing checks

  --wait wait for the last command to finish (e.g., snapshot)
```

```
 Commands:
Server status
  gadmin status [gpe gse restpp dict,...]

IUM status
  gadmin ium_status

Disk space of devices
  gadmin ds [path]

Mount info of a path
  gadmin mount {path}

Memory usage of TigerGraph components
  gadmin mem [gse gpe restpp dict,...]

CPU usage of TigerGraph components
  gadmin cpu [gse gpe restpp dict,...]

Check TigerGraph system prerequisites and resources
  gadmin check

Show log of gpe, gse, restpp and issued fab commands
  gadmin log [gse gpe restpp dict fab,...]

Get various information about gpe, gse and restpp
  gadmin info [gse gpe restpp dict,...]

Software version(s) of TigerGraph components
  gadmin version [gse gpe restpp dict,...]

Stop specified or all services
  gadmin stop [gse gpe restpp dict,...]

Restart specified or all services
  gadmin restart [gse gpe restpp dict,...]

Start specified or all services
  gadmin start [gse gpe restpp dict,...]

Start the RESTPP loaders
  gadmin start_restpp_loaders

Start the KAFKA loaders
  gadmin start_kafka_loaders

Stop the RESTPP loaders
  gadmin stop_restpp_loaders
```

```
Stop the KAFKA loaders
  gadmin stop_kafka_loaders

Dump partial or full graph to a directory
  gadmin dump_graph {gse, gpe [*, segment], all}, dir, separator

Snapshot gpe and gse
  gadmin snapshot

Reset the kafka queues
  gadmin reset

Show the available packages
  gadmin pkg-info
```

Examples

```
Install new package to TigerGraph system
  gadmin pkg-install

Update gpe, gse, restpp, dict, etc. without configuration change
  gadmin pkg-update

Remove available packages or binaries from package pool
  gadmin pkg-rm [files]

Apply new configure. Note some modules may need to restart
  gadmin config-apply [gse gpe restpp dict kafka zk]

Set a new license key
  gadmin set-license-key license key string

Update the new graph schema
  gadmin update_graph_config

Update components under a directory
  gadmin update

Setup sync of all gstore data in mutiple machines
  gadmin setup_gstore_sync

Setup rate control of RESTPP loader
  gadmin setup_restpploader_rate_ctl

 Restart sync of all gstore data in mutiple machines
  gadmin gstore_sync_restart

Stop sync of all gstore data in mutiple machines
  gadmin gstore_sync_stop
```

```
$ gadmin status

=== zk ===
[SUMMARY][ZK] process is up
[SUMMARY][ZK] /home/tigergraph/tigergraph/zk is ready
=== kafka ===
[SUMMARY][KAFKA] process is up
[SUMMARY][KAFKA] queue is ready
=== gse ===
[SUMMARY][GSE] process is down
[SUMMARY][GSE] id service has NOT been initialized
=== dict ===
[SUMMARY][DICT] process is up
[SUMMARY][DICT] dict server is ready
=== graph ===
[SUMMARY][GRAPH] graph has NOT been initialized
=== restpp ===
[SUMMARY][RESTPP] process is down
[SUMMARY][RESTPP] restpp has NOT been initialized
=== gpe ===
[SUMMARY][GPE] process is down
[SUMMARY][GPE] graph has NOT been initialized
=== glive ===
[SUMMARY][GLIVE] process is up
[SUMMARY][GLIVE] glive is ready
=== Visualization ===
[SUMMARY][VIS] process is up (WebServer:2254; DataBase:2255)
[SUMMARY][VIS] Web server is working
```

Stopping a particular server, such as the rest server (name is “restpp"):

```
$ gadmin stop restpp
```

Changing the retention size of queue to 10GB:

```
$ gadmin --set -f online.queue.retention_size 10
```

Updating the TigerGraph License Key

A TigerGraph license key is initially set up during the installation process. If you have obtained a new license key, run the command

```
gadmin set-license-key <new_key>
```

to install your new key. You should then follow this with

```
gadmin config-apply
```

Example: Setting the license key

```
$ gadmin set-license-key new_license_key

[RUN ] /home/tigergraph/.gsql/gpe_auto_start_add2cron.sh
[RUN ] /home/tigergraph/.gsql/all_log_cleanup_add2cron.sh
[RUN ] rm -rf /home/tigergraph/tigergraph_coredump
[RUN ] mkdir -p /home/tigergraph/tigergraph/logs/coredump
[RUN ] ln -s /home/tigergraph/tigergraph/logs/coredump /home/tigergraph/ti

$ gadmin config-apply
[FAB ][2017-03-31 15:03:05] check_config
[FAB ][2017-03-31 15:03:06] update_config_all
Local config modification Found, will restart dict server and update confi
[FAB ][2017-03-31 15:03:11] launch_zookeepers
[FAB ][2017-03-31 15:03:21] launch_gsql_subsystems:DICT
[FAB ][2017-03-31 15:03:22] gsql_mon_alert_on
Local config modification sync to dictionary successfully!

$
```

Backup and Restore

GBAR - Graph Backup and Restore

GBAR (Graph Backup And Restore), is an integrated tool for backing up and restoring the data and data dictionary (schema, loading jobs, and queries) of a single TigerGraph node. In Backup mode, it packs TigerGraph data and configuration information in a single file onto disk or a remote AWS S3 bucket. Multiple backup files can be archived. Later, you can use the Restore mode to rollback the system to any backup point. This tool can also be integrated easily with Linux cron to perform periodic backup jobs.

Introduction and 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 config
       gbar list

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
  -t BACKUP_TAG  Tag for backup file, required on backup
```

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.

Changes between v2.0 and v2.1

Config

- For S3 configuration, the AWS access key and secret are not provided, then GBAR will use the attached IAM role.
- You can specify the number of parallel processes for backup and restore.
- If GSQL authentication is enabled, you must provide a username and password.

Backup

- A backup archive is stored as several files in a folder, rather than as a single file.
- Distributed backup performance is improved.

Restore

- To select a backup archive to restore, the full backup name must be specified.
- Restore asks fewer interactive questions than before:
  - The user must provide a full archive name; there is no option to select the latest from a set of archives.
  - GBAR restore does not estimate the the uncompressed data size and check whether there is sufficient disk space.

Config

```
gbar config
```

GBAR Config must be run before using GBAR backup/restore functionality. GBAR Config will open the following configuration template interactively in a text editor. Using the comments as a guide, edit the configuration file to set the configuration parameters according to your own needs.

Synopsis

```
# Configure file for GBAR
# you can specify storage method as either local or s3.

# Assign True if you want to store backup files on local disk.
# Assign False otherwise, in this case no need to set path.
store_local: False
path: PATH_TO_BACKUP_REPOSITORY

# Assign True if you want to store backup files on AWS S3.
# Assign False otherwise, in this case no need to set AWS key and bucket.
# AWS access key and secret is optional. If not specified, it will use
# attached IAM role of the instance.
store_s3: False
aws_access_key_id:
aws_secret_access_key:
bucket: YOUR_BUCKET_NAME

# The maximum timeout value to wait for core modules(GPE/GSE) on backup.
# As a roughly estimated number,
# GPE & GSE backup throughoutput is about 2GB in one minute on HDD.
# You can set this value according to your gstore size.
# Interval string could be with format 1h2m3s, means 1 hour 2 minutes 3 s
# or 200m means 200 minutes.
# You can set to 0 for endless waiting.
backup_core_timeout: 5h

# The number of processes to be created during compressing backup archive
# Compressing in parallel can gain improved performance.
# The same number of processes will be spawned for decompression on resto
compress_process_number: 8

# Need to put gsql user/passwd here if gsql authentication is on
gsql_user:
gsql_passwd:
```

If you do not wish to store the username and password in the config file, you can prepend the user login credentials, as environment variables, to the gbar command you wish to run.

Leaving the config file's username and password fields blank will require you to manually prepend the login information to the gbar command, as seen below.

```
$ GSQL_USERNAME=tigergraph GSQL_PASSWORD=tigergraph gbar backup -t daily
```

Backup

```
gbar backup -t <backup_tag>
```

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 `store_local` is true, the folder is a local folder on every node in a cluster, to avoid massive data moving across nodes in a cluster. If `store_s3` is 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. If IAM policy is used for authentication, every node in the cluster needs to be attached with the IAM policy.

GBAR Backup performs a live backup, meaning that normal operations may continue while backup is in progress. When GBAR backup starts, it sends a request to gadmin , 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 records TigerGraph system version. 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. It's highly recommended when issuing the backup command, no active data update is in progress. A no-write time period of about 5 seconds is sufficient.

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, file's size in human readable format, and its creation time.

Restore

```
gbar restore <archive_name>
```

Restore is an offline operation, requiring the data services to be temporarily shut down. The user must specific 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 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. When these actions are all done, GBAR will restart the TigerGraph servers.

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. Specific requirements are listed below.

Restore Requirements and Limitations

Restore is supported if the TigerGraph system has had only minor version updates since the backup.

- TigerGraph version numbers have the format X.Y[.Z], where X is the major version number and Y is the minor version number.

- Restore is supported if the backup archive and the current system have the same major version number AND the current system has a minor version number that is greater than or equal to the backup archive minor version number.
- Backup archives from a 0.8.x system cannot be Restored to a 1.x system.
- Examples:

Backup archive's system version	current system version	Restore is allowed?
0.8	1.0	NO - Major versions differ
1.1	1.1	YES - Major and minor versions are the same
1.1	1.2	YES - Major versions are the same; current minor version > archived minor version
1.1	1.0	NO - Major versions are the same; current minor version < archived minor version

⚠ Restore needs enough free space to accommodate both the old gstore and the gstore to be restored.

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, and 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 suffix
```

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

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 set-license-key <your_license_key>
```

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"
```

What are the components of the TigerGraph platform?

```
GPE
GSE
RESTPP
ZK
KAFKA
NGINX
DICT
GSQL
GLIVE
VISUALIZATION (GraphStudio)
```

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?

The default RESTful API port is 9000. It can be changed by configuration. To find out the current RESTful API port, use following command:

```
gadmin --dump-config | grep nginx.port
```

The default port for the GraphStudio UI is 14240. (Prior to TigerGraph 1.2, it was 44240.) Use the following to check its configuration:

```
gadmin --dump-config | grep nginx.services.port
```

If you are using a remote GSQL client, it communicates with the GSQL server via port 14240.

```
gadmin --dump-config | grep gsql.server.port
```

To see a list of all ports:

```
gadmin --dump-config | grep port
```

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?

The command

```
gadmin status graph -v
```

will tell you the size of graph data on disk, number of vertices and edges.

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
gadmin stop

# stop selected services
gadmin stop <component_name(s)>
```

Note: running "gadmin stop" still does not stop every single TigerGraph service. Ts3 will still run because it is monitoring other services, and the Admin server will still run because it manages the other services. If you need to perform a full shutdown, for example, before a software upgrade or before a hardware change, perform the following sequence of commands:

```
gadmin stop ts3 -y
gadmin stop -f -y
gadmin stop admin -y
```

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>
```

If you want to look at only the last N lines of the log:

```
gadmin log -v [component] [-n number_of_lines]
```

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 --configure timeout_seconds
```

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>" '<re
```

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`

Web UI

GraphStudio UI Guide

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 May 2019, the GraphStudio UI is certified on following browsers:

Browser	Chrome	Safari	Firefox	Opera
Supported version	54.0+	11.1+	59.0+	52.0+

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 the GraphStudio UI does not show, the visualization service might be off. To verify, in a linux shell of the server, type

```
$ gadmin status vis
```

If it is off, turn it on:

```
$ gadmin start vis
```

If you still cannot access GraphStudio, check your firewall rules and open 14240 port to public. For example, if your Linux OS uses firewalld:

```
$ firewall-cmd –zone=public –add-port=14240/tcp –permanent
```

Home Page

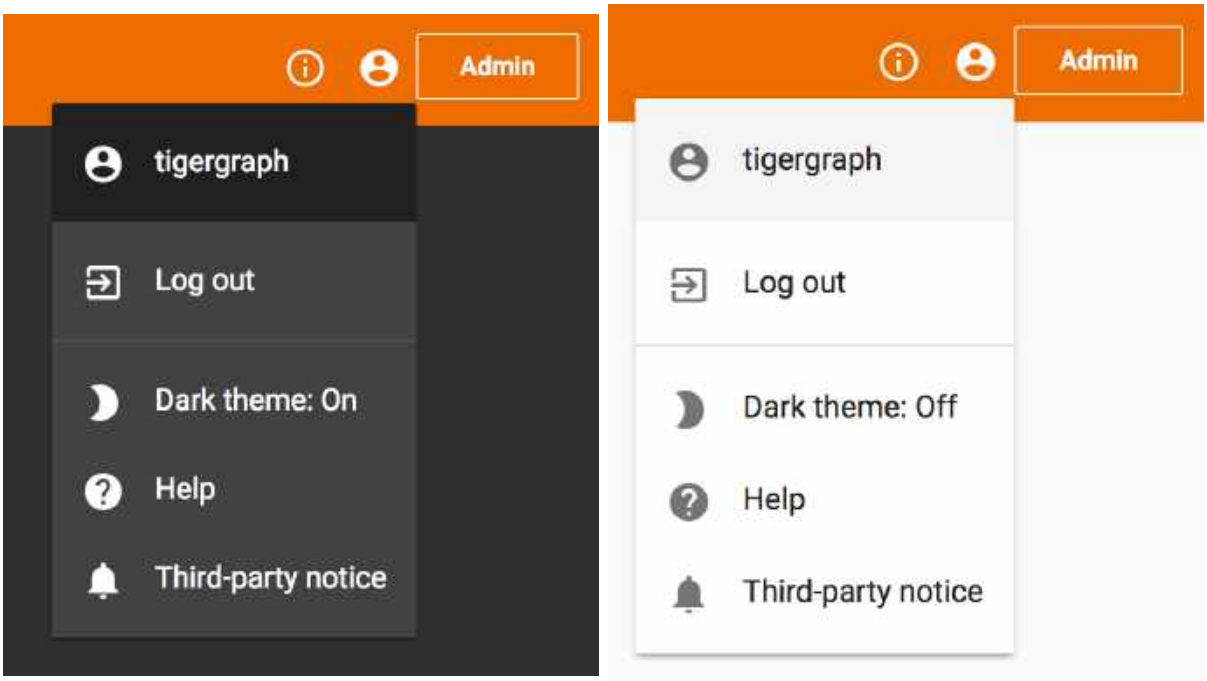
The home page of GraphStudio contains links to each of the five steps of solving a business problem: Design Schema, Map Data To Graph, Load Data, Explore Graph, and Write Queries. Users can also navigate to each step from the buttons in the left menu bar. Each of these major steps has its own page. To hide/show the left menu bar, click the top-left menu button:  . Clicking the logo on the banner 

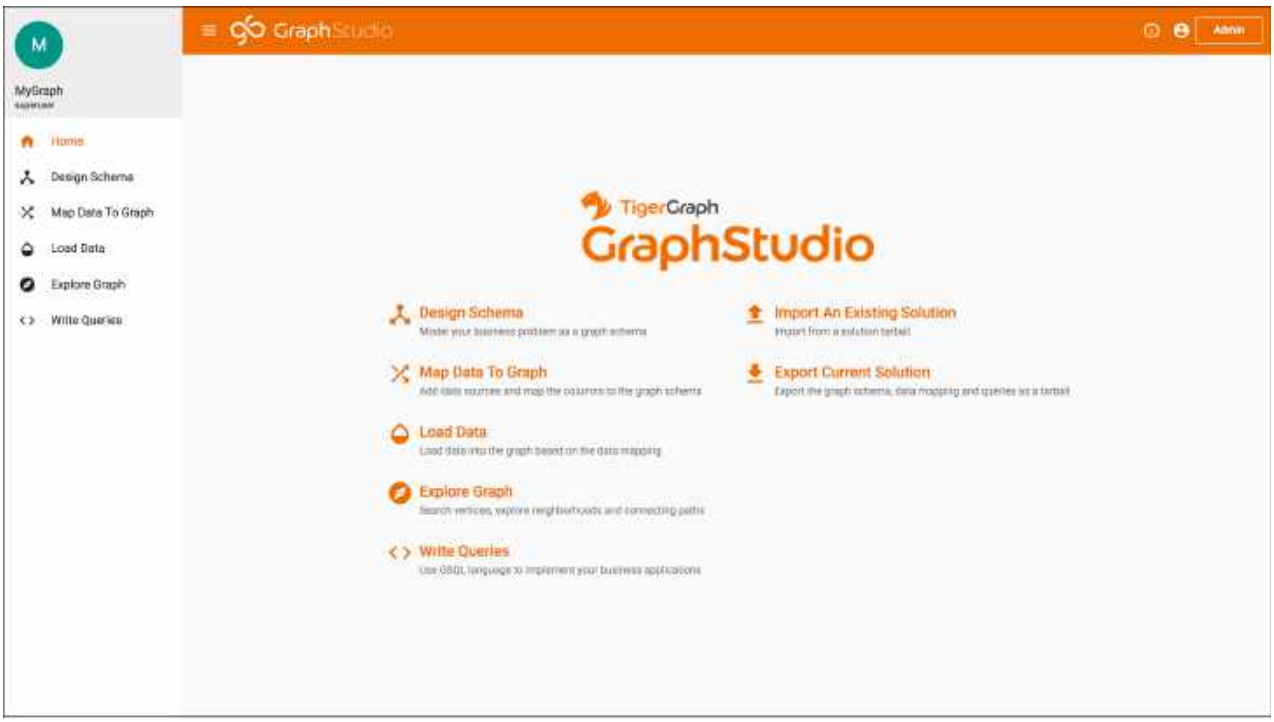
will take you back to the home page. You can click 

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:





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.
- Edges from a set of vertex types to a set of vertex types are not supported.

- There are limitations for MultiGraph. See User Access Management

Map Data to Graph

- Cannot load JSON data.
- Cannot create a data mapping for a MAP type and UDT type attribute.

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

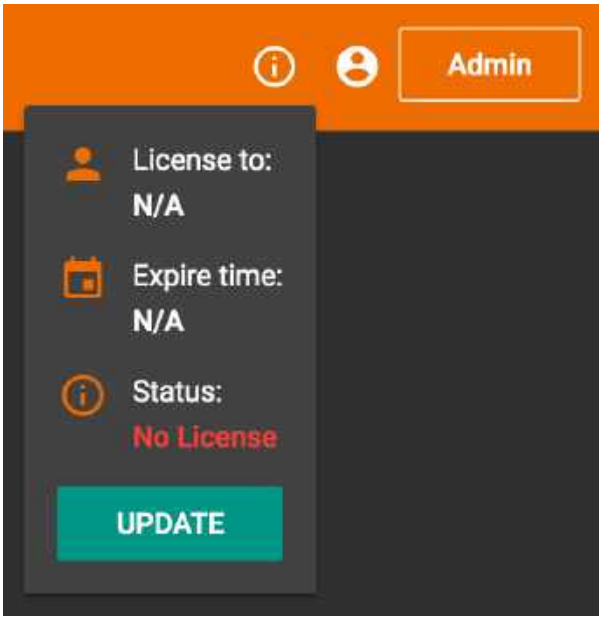
- You cannot define a user-defined function (you can use the user-defined functions created from TigerGraph server).

GraphStudio License

GraphStudio License

GraphStudio operation requires a valid license. The GraphStudio license is independent from the TigerGraph database license; some TigerGraph product editions come with a GraphStudio license pre-installed. The GraphStudio license expiration date might be different from that of the TigerGraph license.

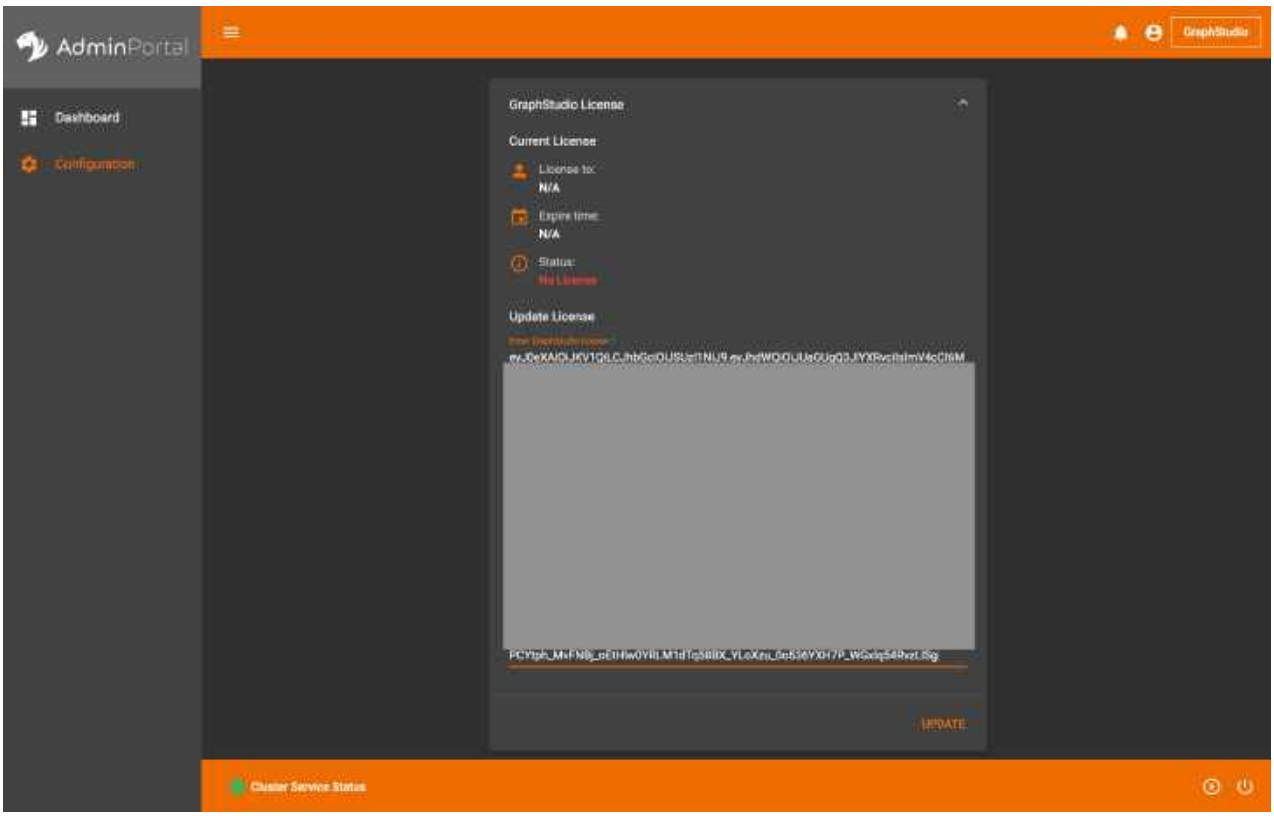
Clicking the GraphStudio Information icon will show the current GraphStudio license status. If a GraphStudio license key has not been installed, the license status will look like the following:



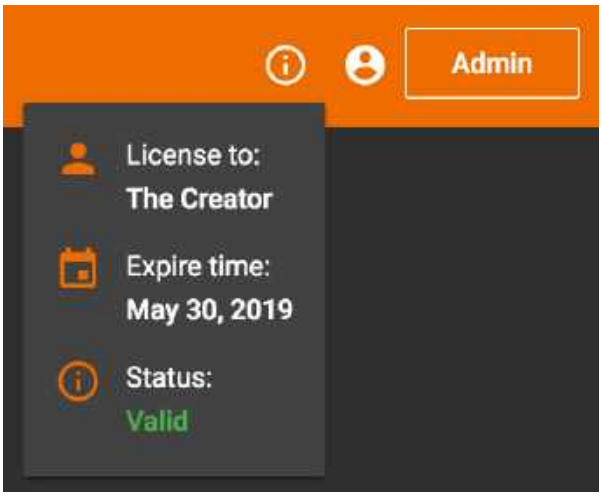
Without a license, it is not possible to navigate to the Design Schema, Map Data To Graph, Load Data, Explore Graph or Write Queries pages.

Applying a GraphStudio License Key

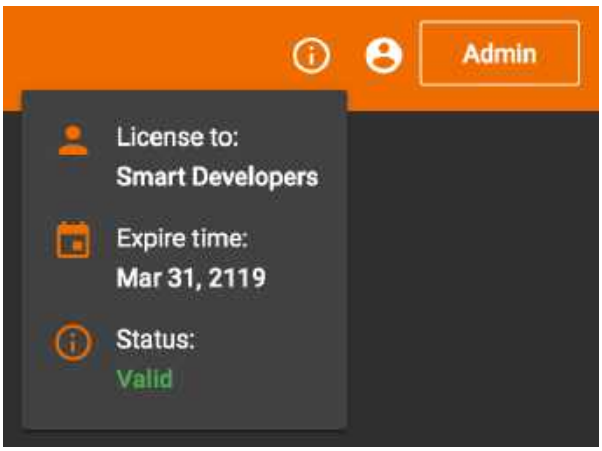
Click the UPDATE link on the bottom of the license status to be redirected to Admin Portal configuration page to apply a GraphStudio license key:



Enter the license key in the Update License text box, and click **update**. Click GraphStudio at the top-right corner 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.


Admin
License to:
Smart Developers
Expire time:
Mar 31, 2119
Status:
Valid

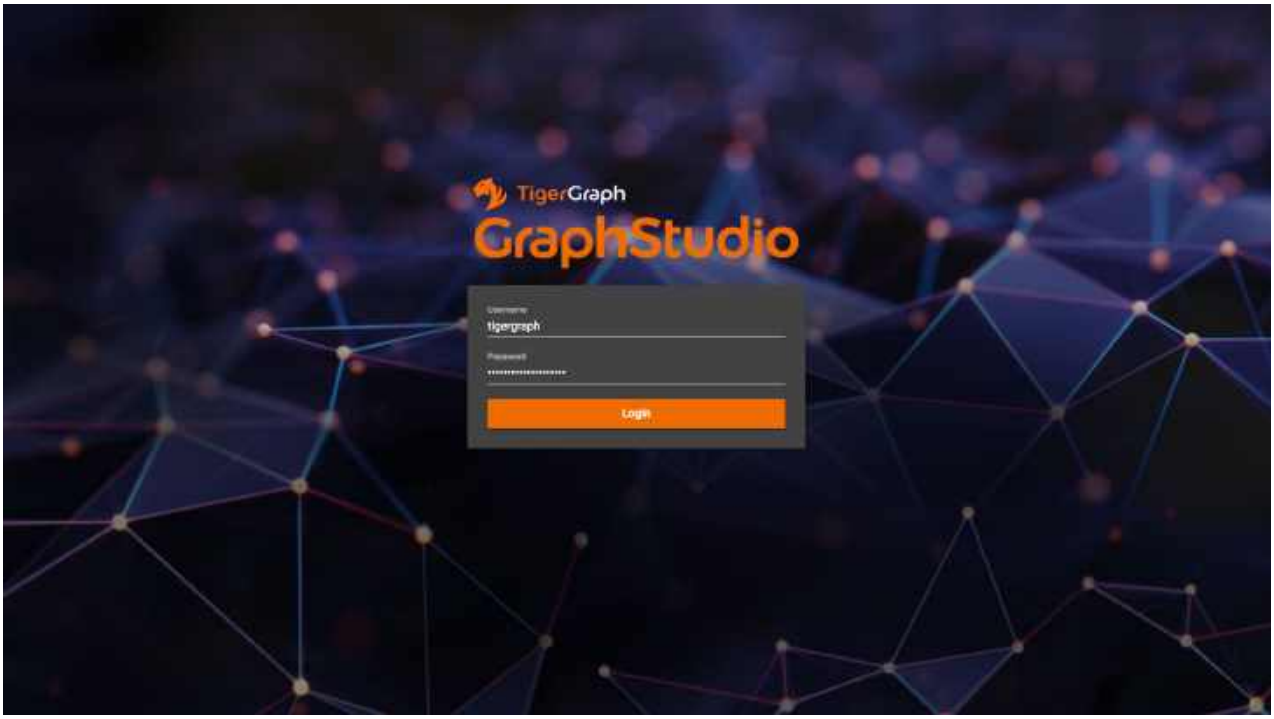
User Access Management

User Access Management

GraphStudio follows TigerGraph user authentication and role-based access control model. Read more in the document Managing User Privileges and Authentication.

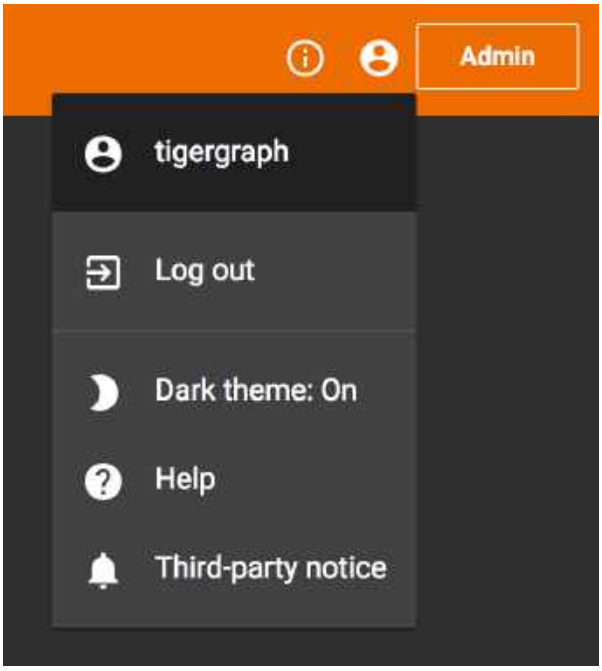
Log On

If user authentication is not enabled, i.e., GSQL *tigergraph* superuser password hasn't been changed, then no user login is needed for GraphStudio. If user authentication has been enabled, then users must provide credentials (e.g., username and password) to enter 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.

To logout, click the User icon and then the Sign Out icon.

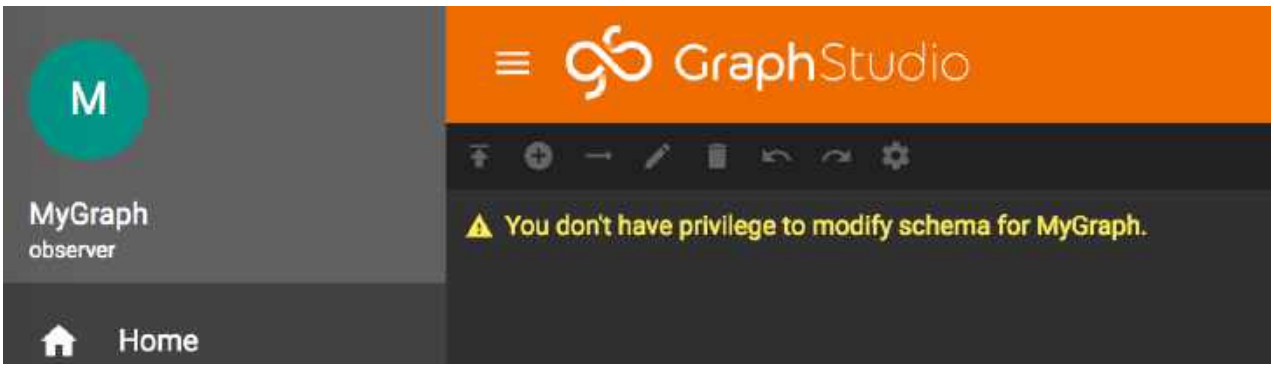


Role- and Graph- Based Access Control

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.

> **Current Limitation**
>
> Currently, role assignments can only be made in the GSQL shell. In the future Admin Portal will support user management functionality.

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:



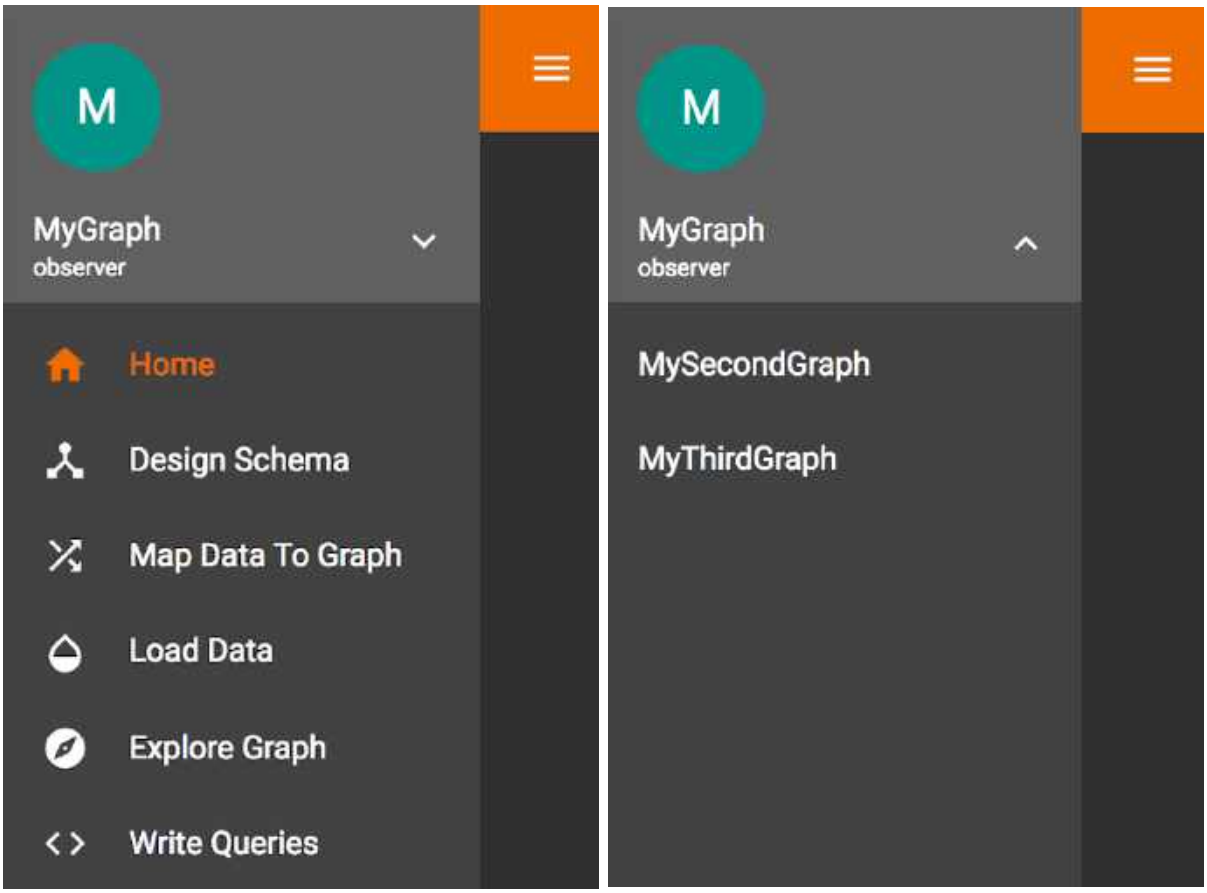
The table below summarizes the built-in roles and of their key privileges on GraphStudio:

	superuser	admin	designer	querywriter	queryreader	obs
Create a new graph schema	YES					
Modify a graph schema	YES	GSQL - yes; GraphStudio - not yet supported	GSQL - yes; GraphStudio - not yet supported			
View a graph schema	YES	YES	YES	YES	YES	YES
Create a data mapping	YES	YES	YES			
View a data mapping	YES	YES	YES	YES	YES	YES
Load data	YES	YES	YES	YES	YES	
Explore a graph	YES	YES	YES	YES	YES	
Write a query	YES	YES	YES	YES		

Run a

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. If you have access to more than one graph, at the top of the Menu Bar an arrow will appear. Click the arrow to expand the graph list and select a graph.



Current Limitations

Currently, not all of the TigerGraph capabilities for creating and using multiple graphs are available through GraphStudio; some operations can only be performed from the GSQL shell. Below is the list of current MultiGraph-related limitations.

Creating a New Graph Schema:

1. A superuser can create a graph schema only if no graphs currently exist.
2. Admin and designer users cannot create a graph schema in GraphStudio.

Modifying a Graph Schema:

1. A superuser can modify a graph schema if and only if exactly one graph exists.
2. Admin and designer users cannot modify a graph schema in GraphStudio.
3. Only superusers can modify visual styling of schemas – color , vertex icons, and layout. Visual styling is supported even when there are multiple graphs.

A graph admin user or superuser grants each user access to particular graphs. Currently, granting and revoking privileges must be done as GSQL commands; user roles cannot be managed in GraphStudio yet.

Design Schema

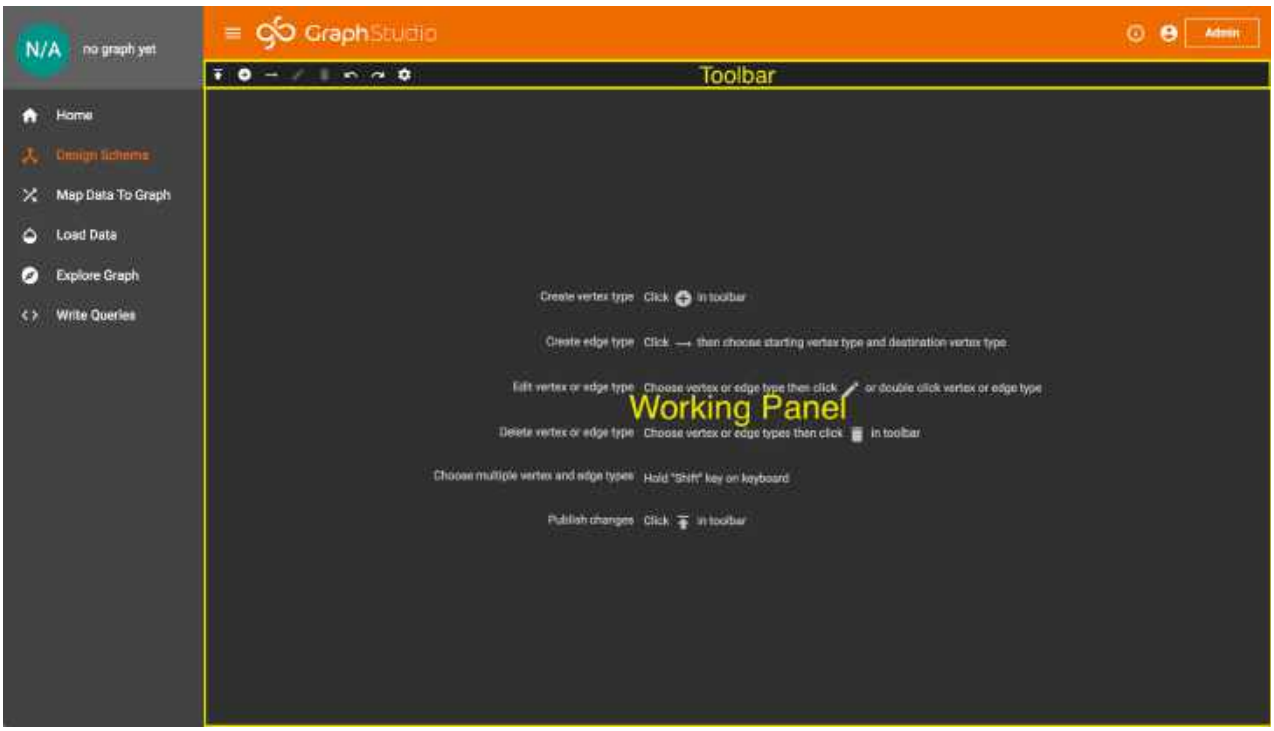
Design Schema

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 already in Design Schema mode, click "Design Schema" on the left side menu bar.

Current Limitations

1. Only users with superuser privilege can use Design Schema to modify a graph schema.
2. If there is already more than one graph, then the superuser can only modify the visual style of the graphs.

When there is no graph schema in the system, this page will show some hints:



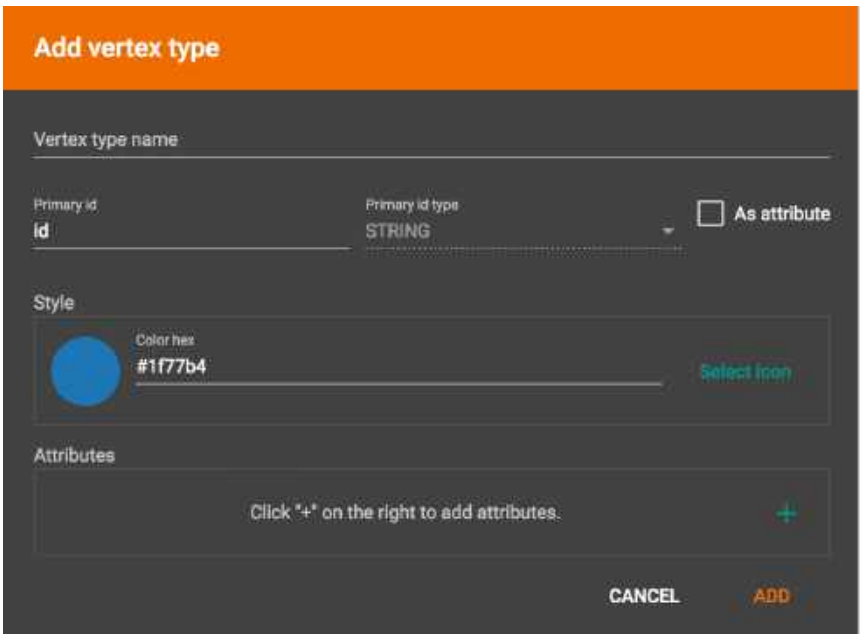
Otherwise this page will visualize the schema:

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 out; two-finger moving down zooms in. Similarly, if your mouse has a scroll wheel, spinning forward zooms out; spinning backward zooms in.

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.

Add A Vertex Type

Click the add vertex type button to add a vertex type. The add vertex type window will pop up:

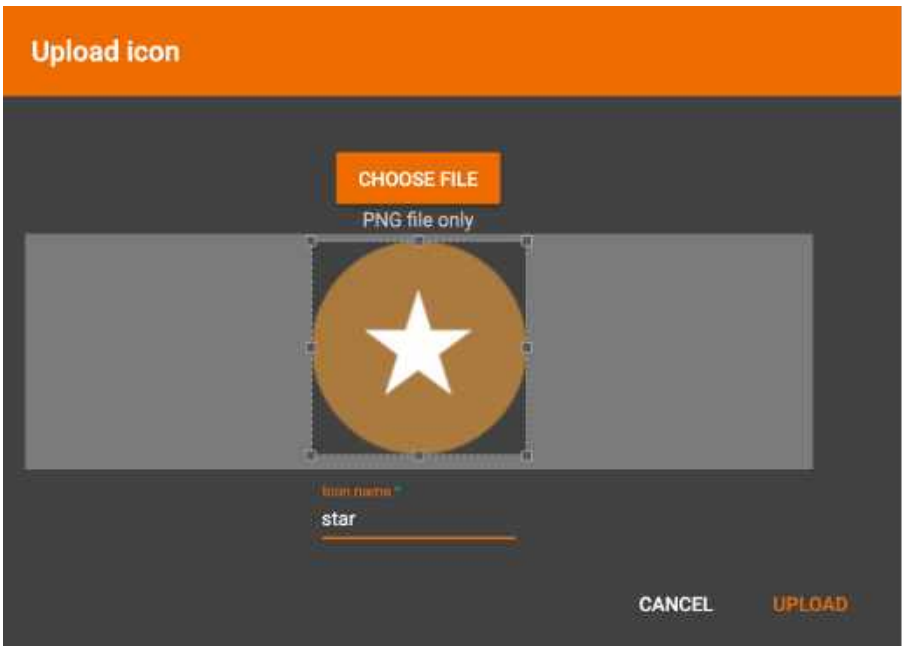


In this window you specify a vertex type name, primary id name. GraphStudio will automatically select a color for your vertex type icon. You can change the vertex type color by clicking the value under the "Color hex" label. A color palette window will pop up allowing you to choose a new color:

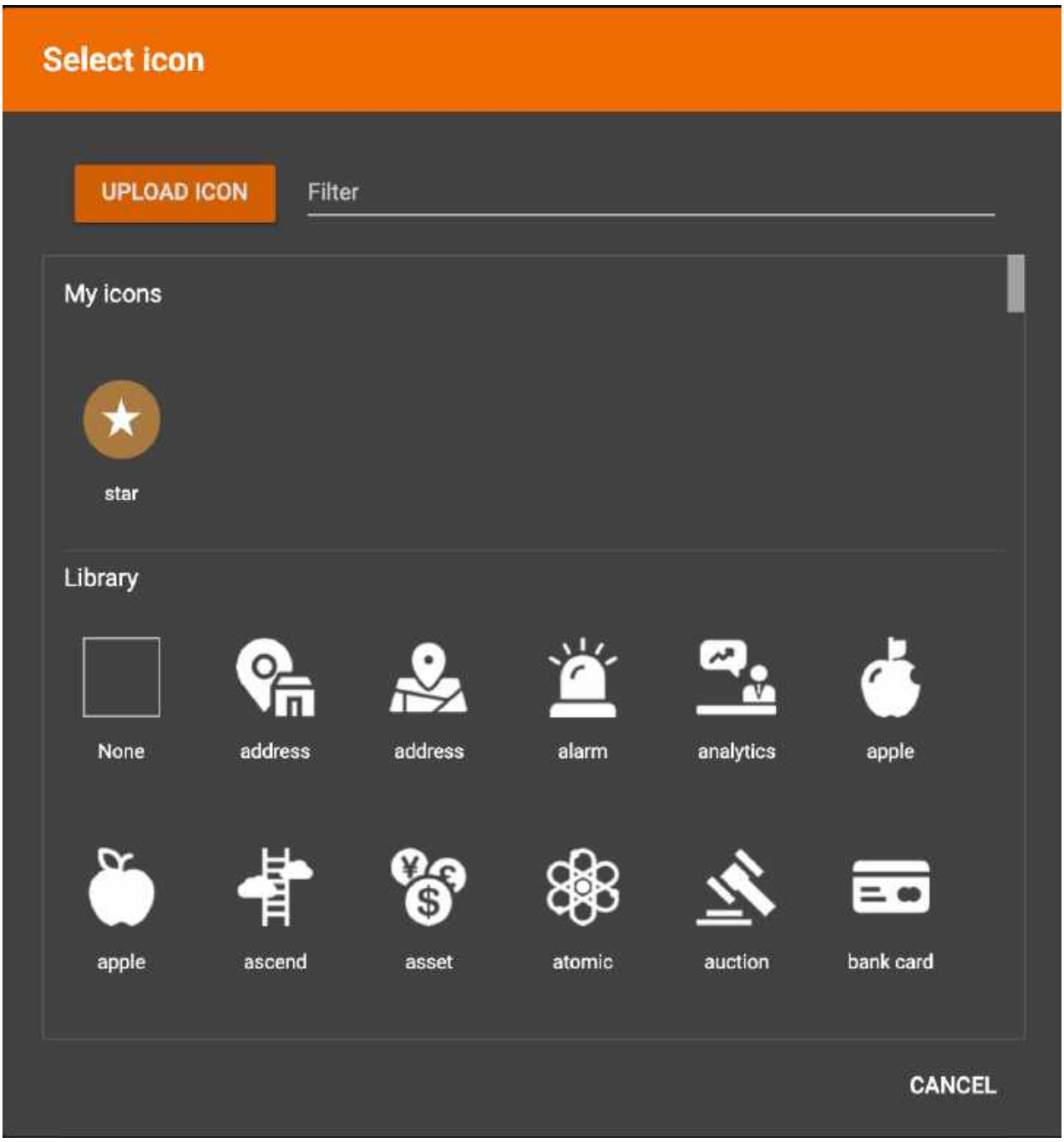
Once you are satisfied with the color, click anywhere outside of the color palette window to set the color.

You can also choose an icon for the vertex type by clicking the Select Icon button Select Icon. 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:

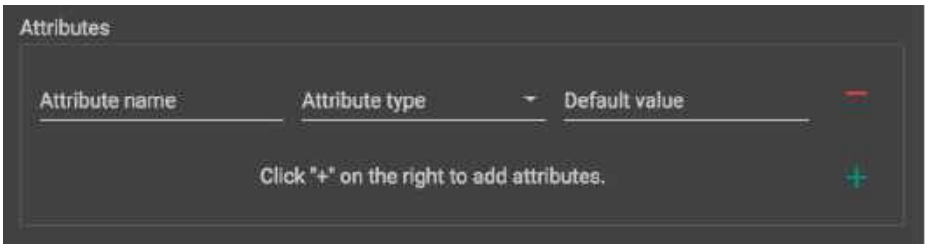


Then you can use your uploaded icons:



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.)

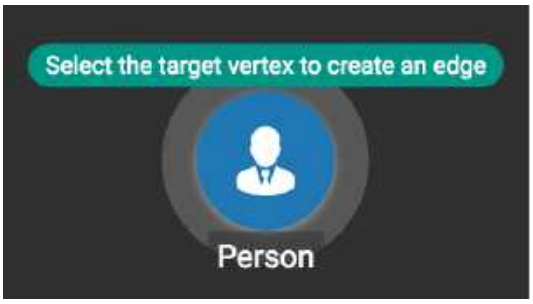
To delete an attribute, click the red minus sign to the right of the attribute to delete an existing attribute.

Once you are satisfied with the vertex type settings, click the Add button ADD to add the vertex type. A new circle will appear in the working panel. You can drag the circle to any desired position.

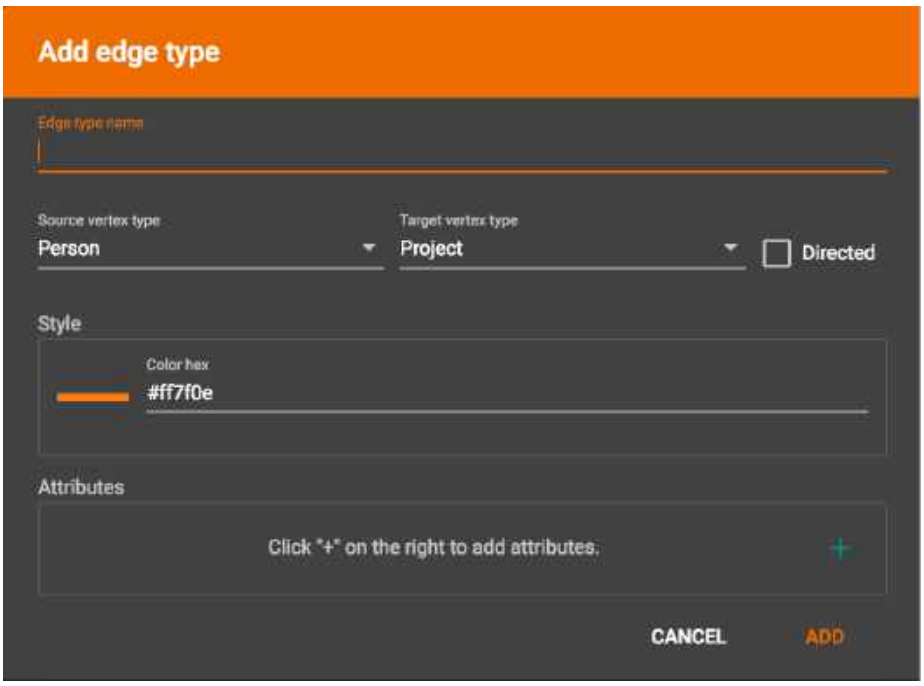
Add An Edge Type

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 a source vertex type and a target vertex type. First, click the source vertex type. A hint will appear on the vertex type circle:

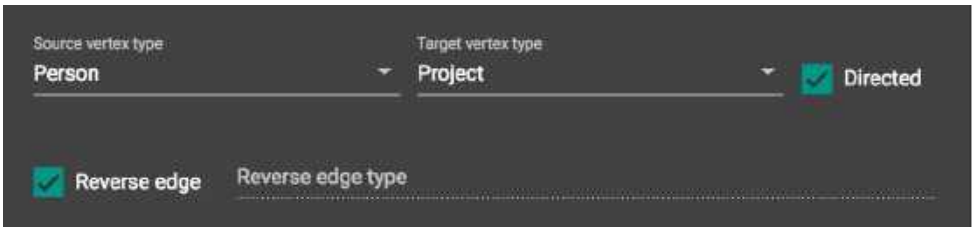


Then click the target vertex type. The add edge type window will pop up:



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.

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.

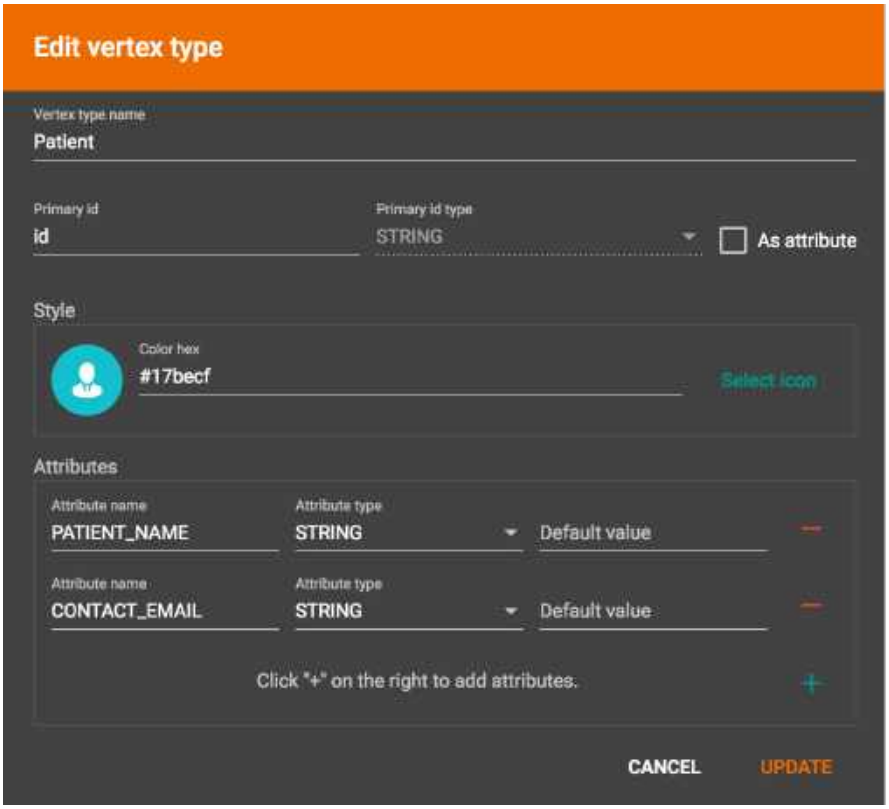
Once you are satisfied with the edge type settings, click the Add button ADD to add the edge type. A new link between the selected source vertex type circle and

target vertex type circle 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), to make the Edit Attributes window pop up:



Once you are satisfied with the change, click the Update button UPDATE.

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 click, to select multiple items.

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.

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.

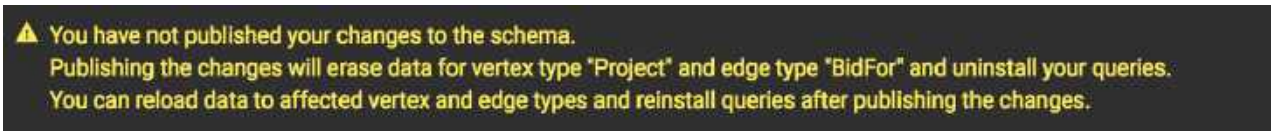
Developer Edition:
SCHEMA_CHANGE is not supported. Publish Schema will always "DROP ALL" (erase all data) before creating your new schema.

Enterprise Edition:
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, GraphStudio will employ a GSQL SCHEMA_CHANGE job, in order to retain the graph data you already loaded.

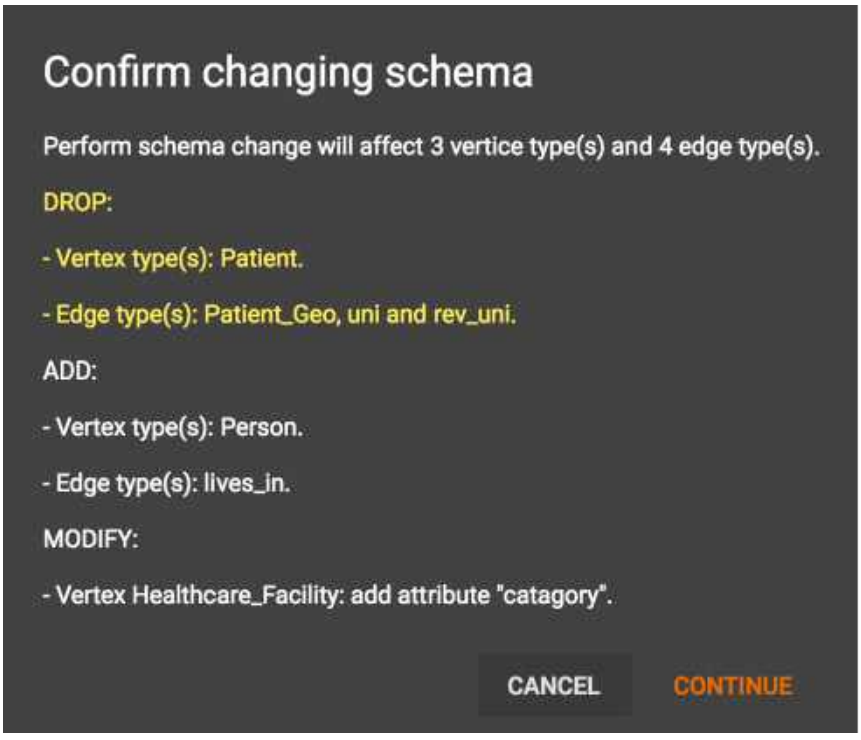
All other types of changes, including **renaming** the vertex or edge type, changing **attribute name or data type**, changing **edge direction**, adding or removing **reverse edge** will result in removing the old vertex or edge type and then adding the new one with your desired configurations. In that case, the loaded data to that vertex or edge type will be erased. Please think twice before you do that kind 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-right 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:

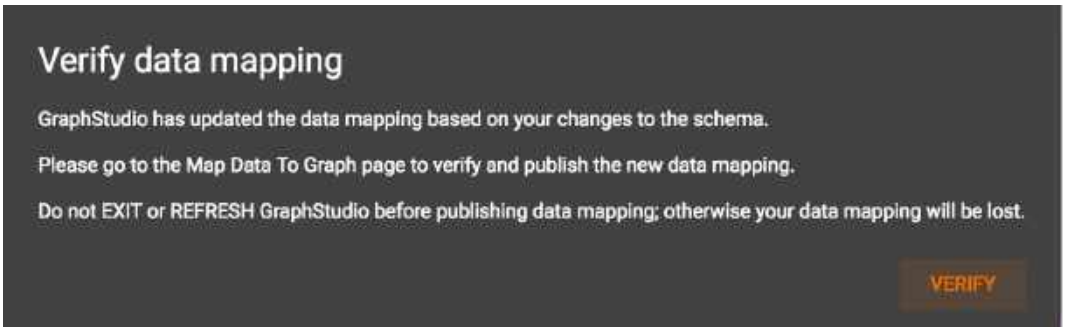


Click continue button, and GraphStudio will start changing your schema:

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.

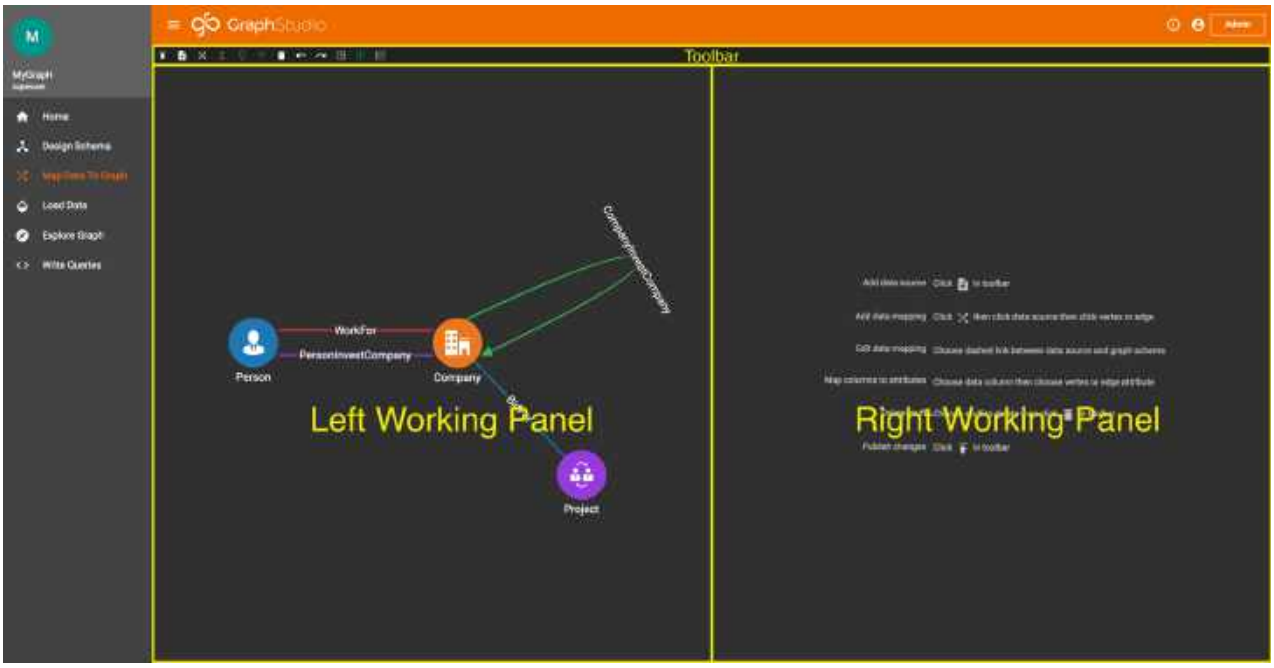
If you have published some data mapping through GraphStudio, then after schema is changed successfully, a pop up window will guide you to go to the Map Data To Graph page to confirm and publish the migrated data mapping:



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 a variety of different data sources. Originally, data could only be loaded from local files. TigerGraph 2.4 adds support for using Amazon S3 data files directly through the GUI. 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.

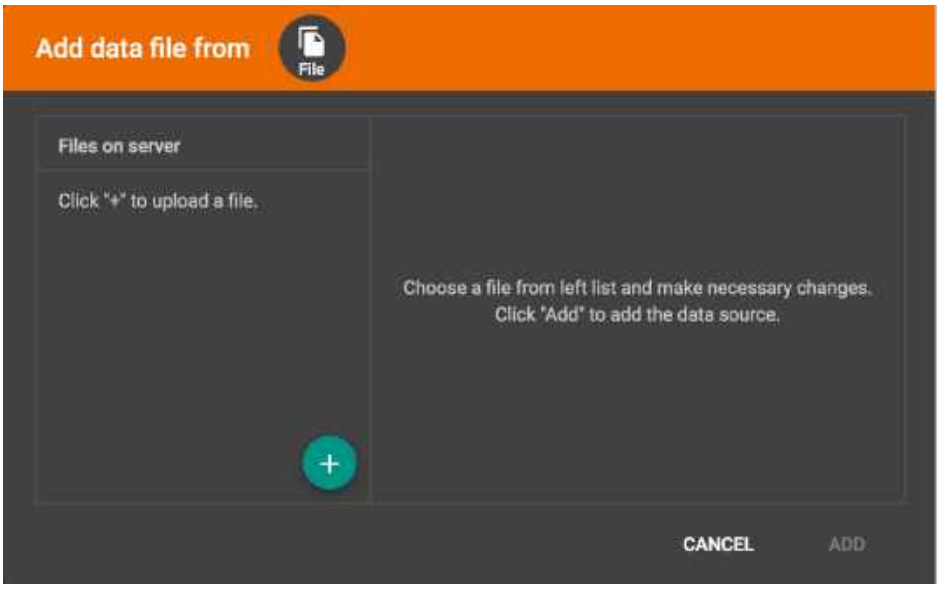
2 Add Data Files

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

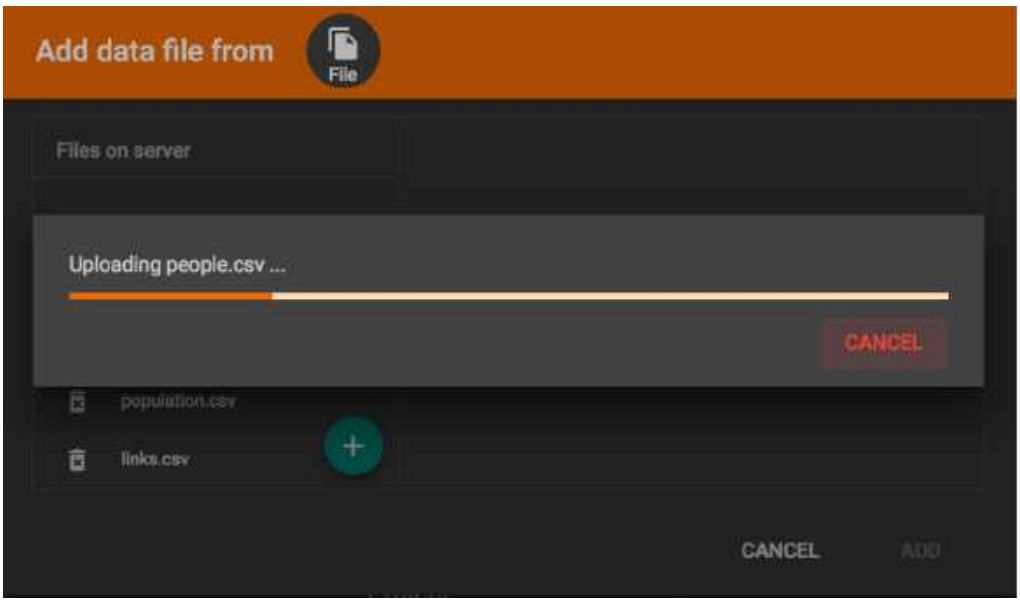
In this step, you inform GraphStudio about your data files. A data file is a file containing structured data to be loaded into the graph, creating vertex and/or edge instances. The first step for data mapping is to specify your data files. Click the Add Data File button to add data files. The Add Data 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 file you want to use. The file will be uploaded to the server data folder:



There is a limit of 500MB on file size. If you are using on-premises deployment, you can bypass this limit by directly putting the data files or their softlinks in the server data folder, located at <TigerGraph_root_dir>/loadingData.

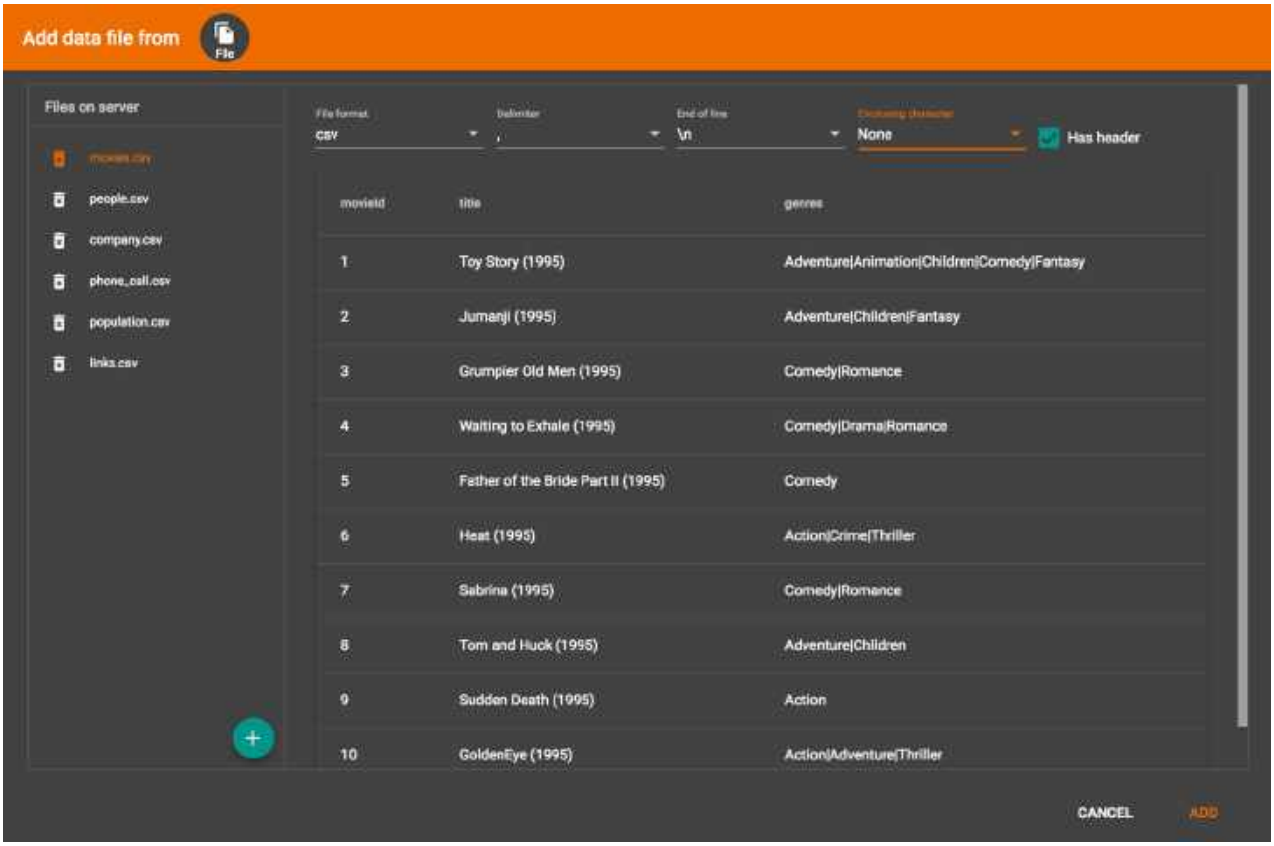
Once the file is uploaded to the server, it will appear in the "Files on server" list on the left side of the Add Data Files window.

Data Files must be .csv files

The Add Data File box will only upload files which end in ".csv". If you manually place files in the <TigerGraph_root_dir>/loadingData folder, please don't put any files into subfolders because they will be ignored.

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.

Once you are satisfied with the file parsing configuration, click the add button ADD to add the data file into left working panel. The data file will be shown as a file icon on the working panel:

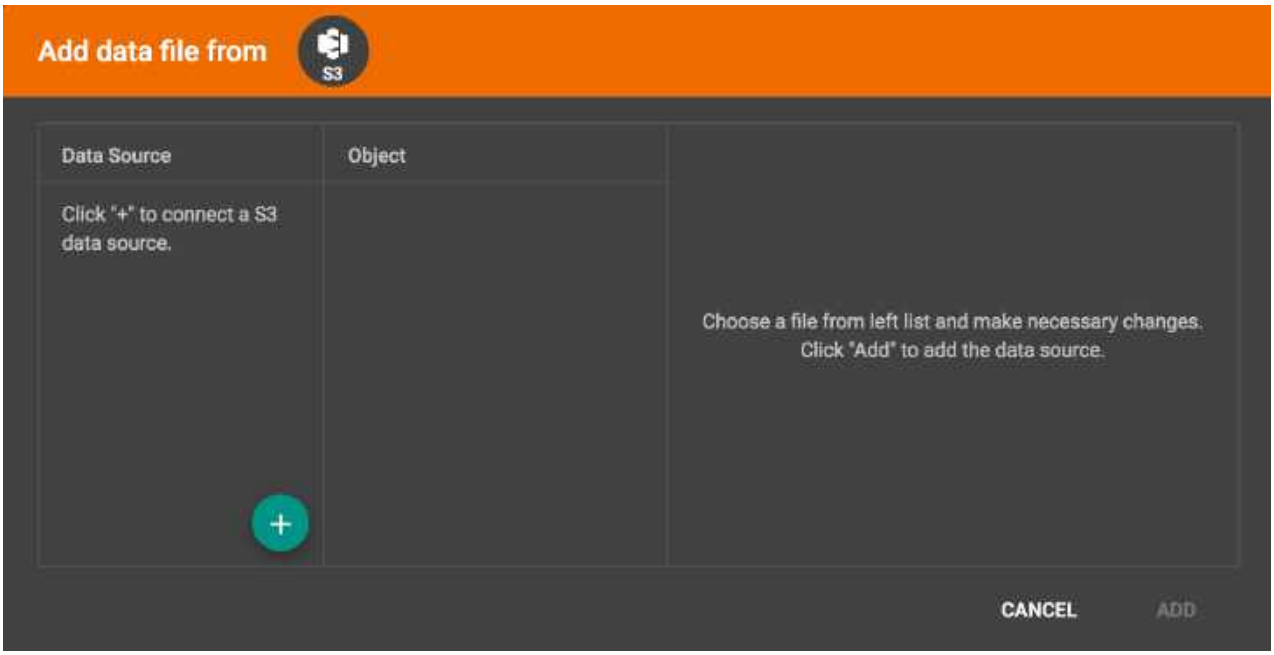
Once you think a file is no longer needed, you can remove it from server by clicking the delete button to the left of each file. Please note that you also need to

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

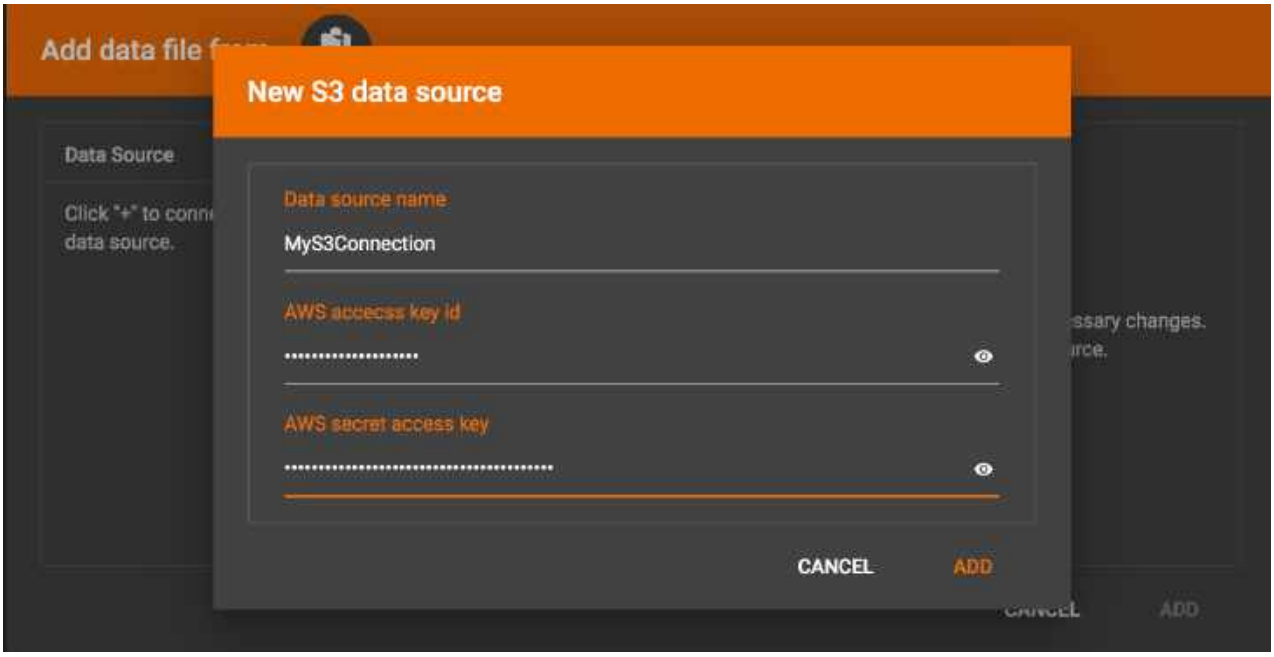
After you click the S3 data source icon, you should see the following window:



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.

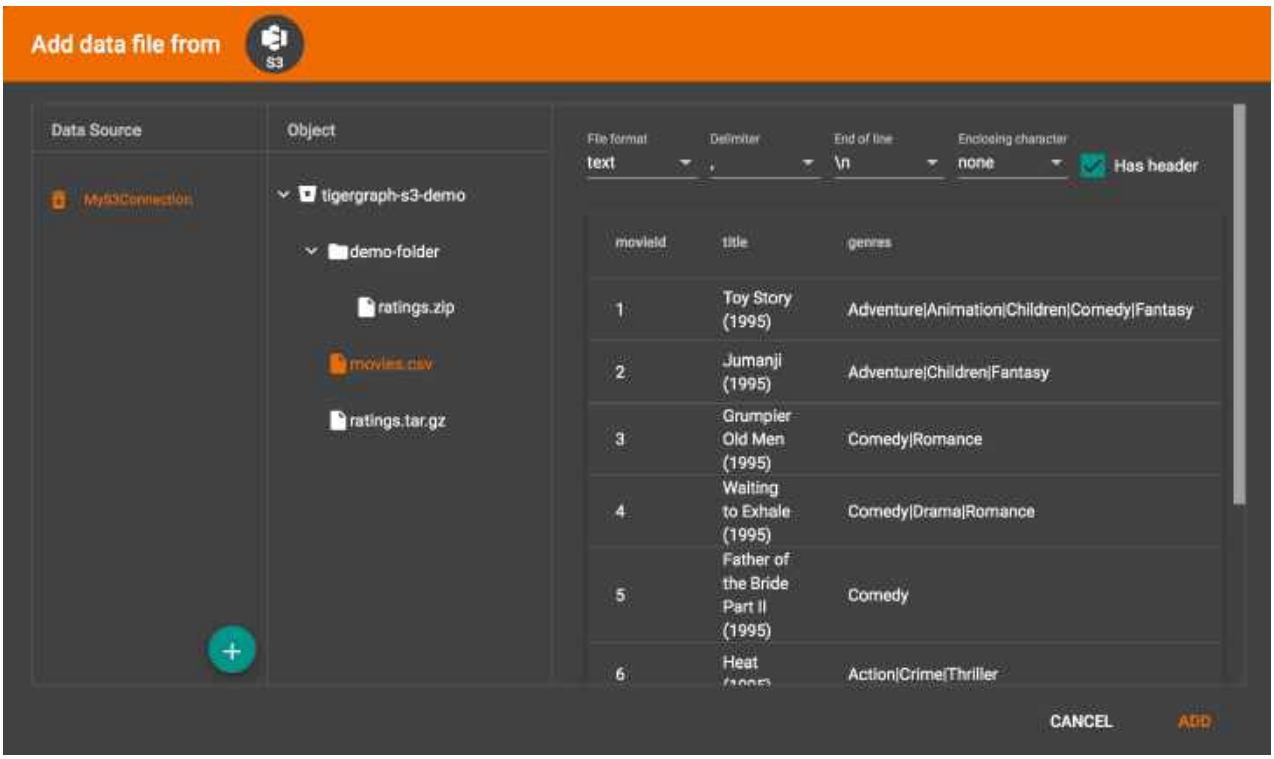
Click the Add new data source button  , then the new S3 data source window will pop up. Give a name to the data source, and provide the access key id and secret access key to connect to S3. Then click the ADD button:



The data source will be created and shown in the Data Source list:

⚠ For security reasons, user-created data sources won't be exported when you export solutions. If you import a solution with S3 data sources, you will need to manually create the data sources again (either though GraphStudio Map Data To Graph page or through the GSQL shell).

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:

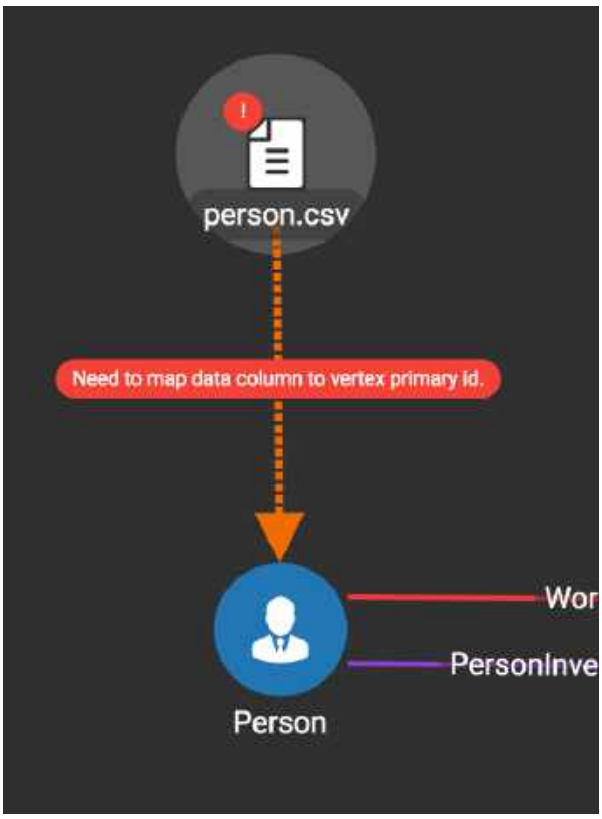
After adding all your s3 data files, continue with Step 3 Map Data Files to Vertex/Edge Types

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 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 are finished mapping 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.

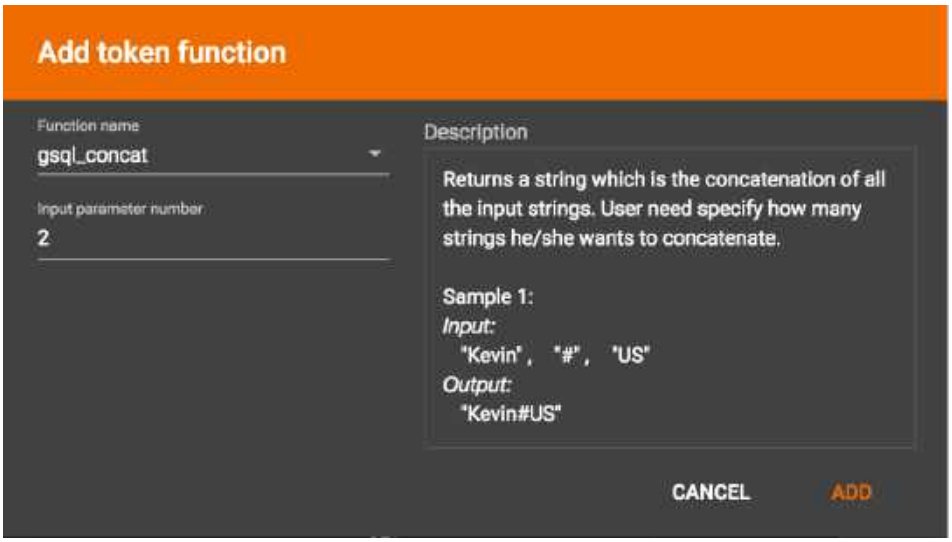
In order to a column in the data file to a vertex or edge field, first click the row representing the data column in the left side data file table:

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.

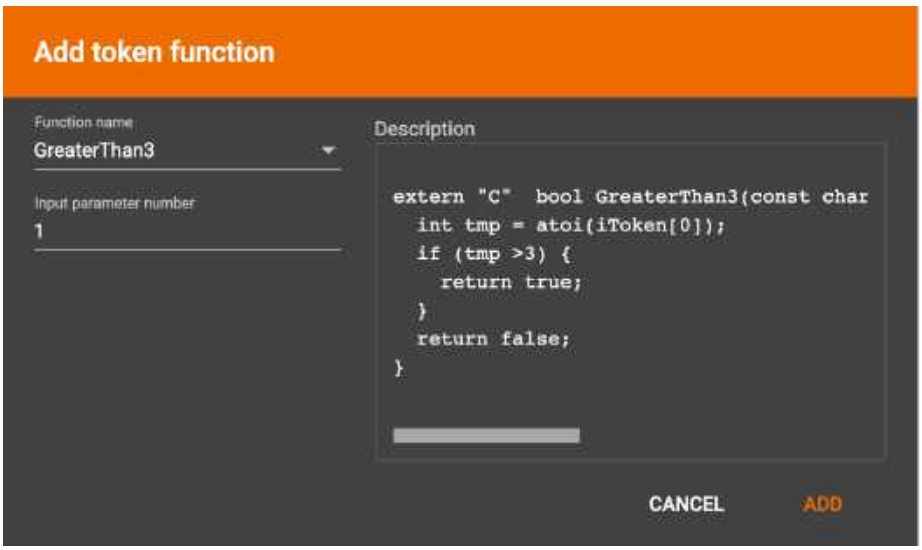
Using a Token Function

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.

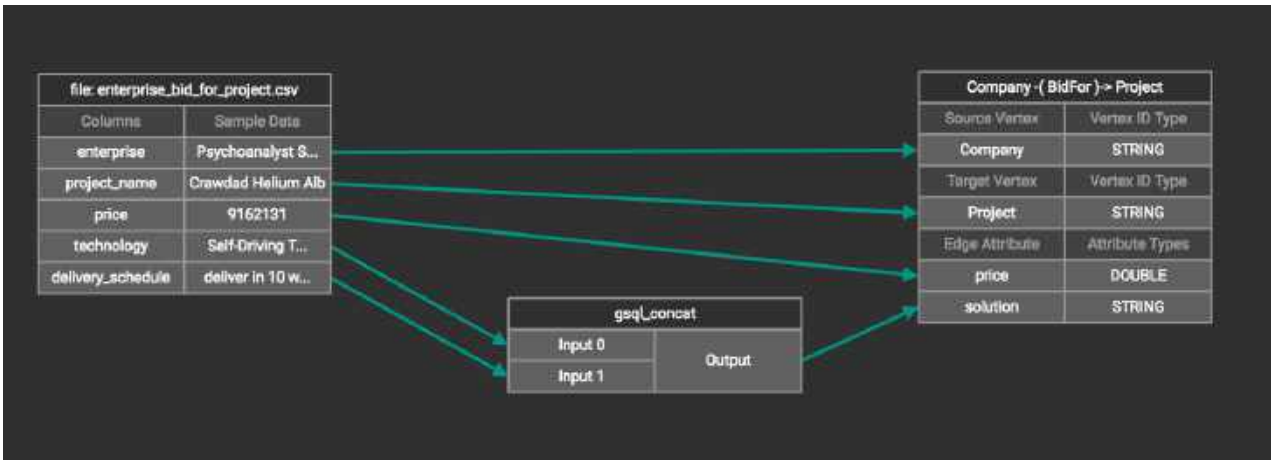
First click the add token function button Σ. The Add Token Function window will pop up. Click the down arrow to see the list of available token functions and select one. For some functions, you may also specify the number of input parameters. (Most token functions have a fixed number of input parameters; 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 right hand side for your reference:



A token function table will be added to the attribute mapping panel. You can drag the tables to rearrange them. Token functions act as an intermediate step 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

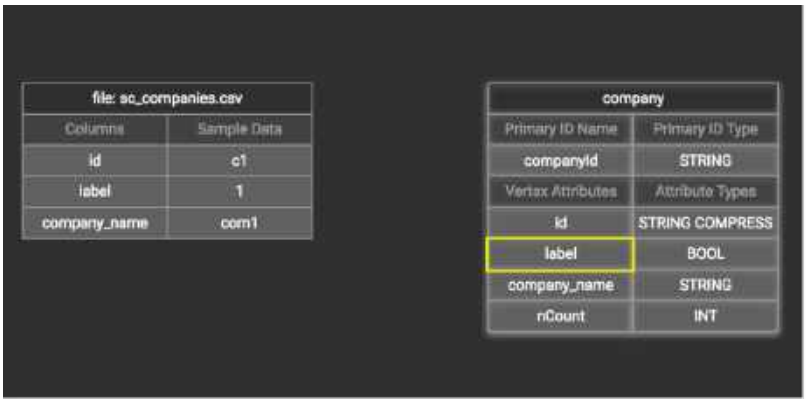
If the data file columns and the vertex/edge attributes have very similar names (only capitalization and hyphen differences), you can click the auto mapping button . All similar columns will be mapped automatically.

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.

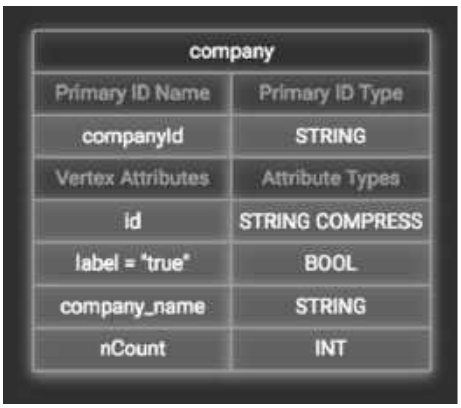
Loading A Constant to An Attribute

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:



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)"** .



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.

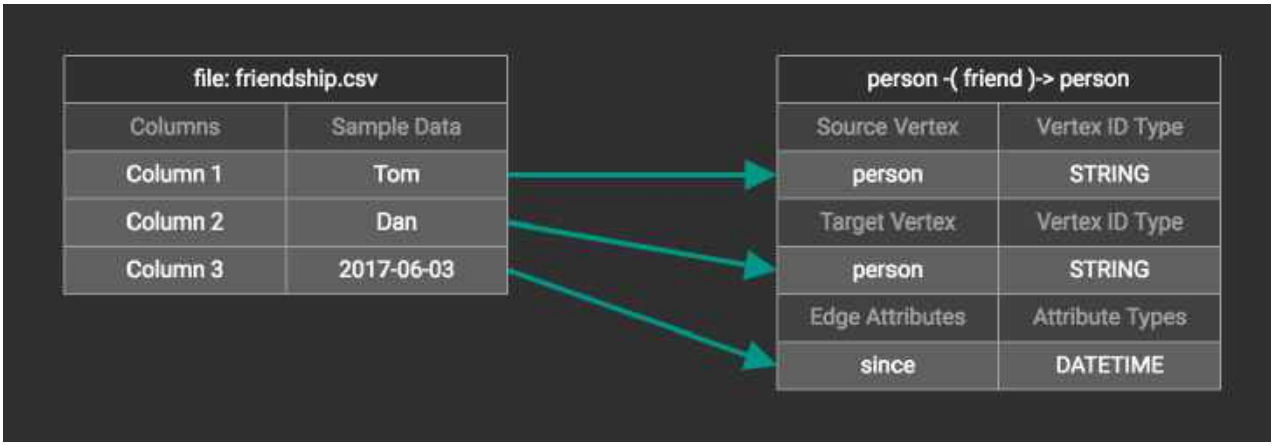
This will cause the Load Constant window to pop up. Type in the constant value and click the Add button to apply the mapping. After adding the constant value, the input's label will change to **Input = "(your input value)"** .

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

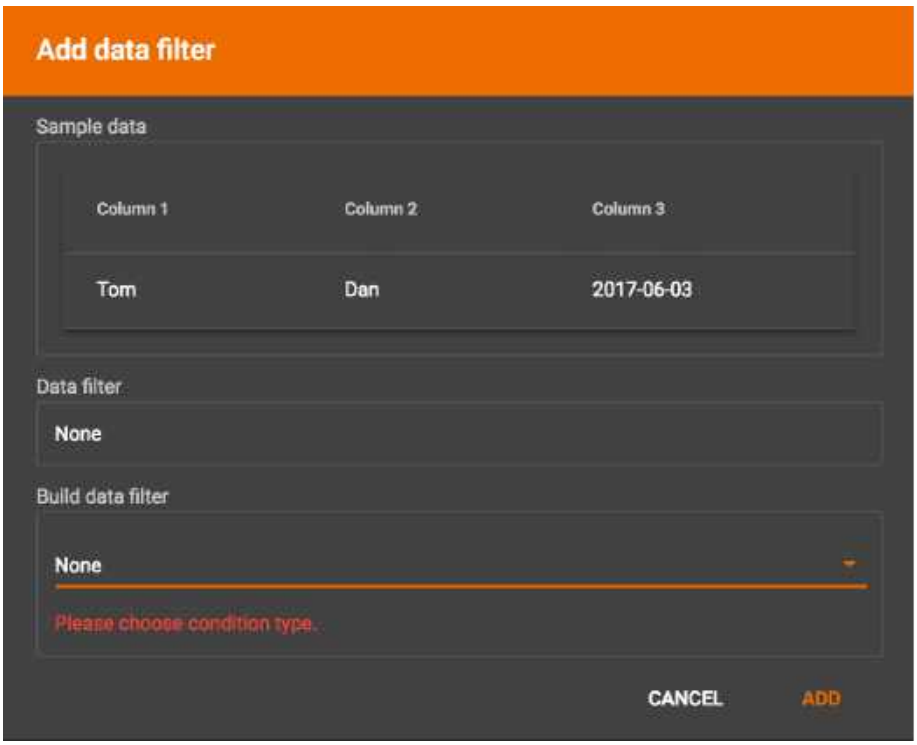
You can add a data filter to a data mapping so that only data records which meet conditions that you specify will be loaded into the graph. This is equivalent to the WHERE clause in a GSQL load statement.

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:



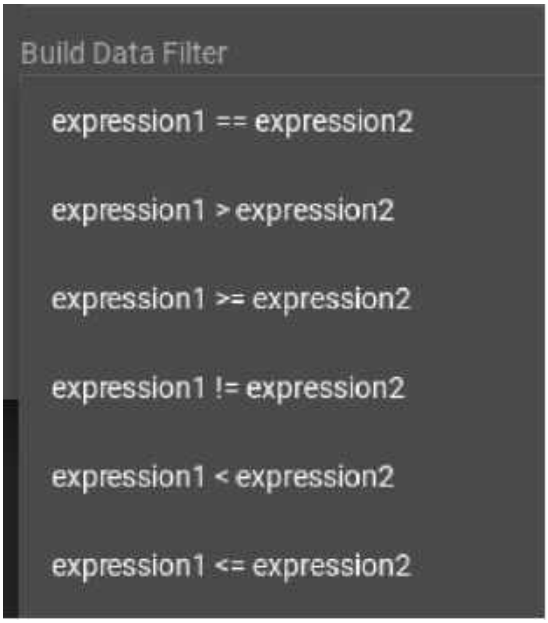
By default, there is no data filter. Click the Data Filter button to start creating a data filter. The Add Data Filter window will appear. The window contains three parts:

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.

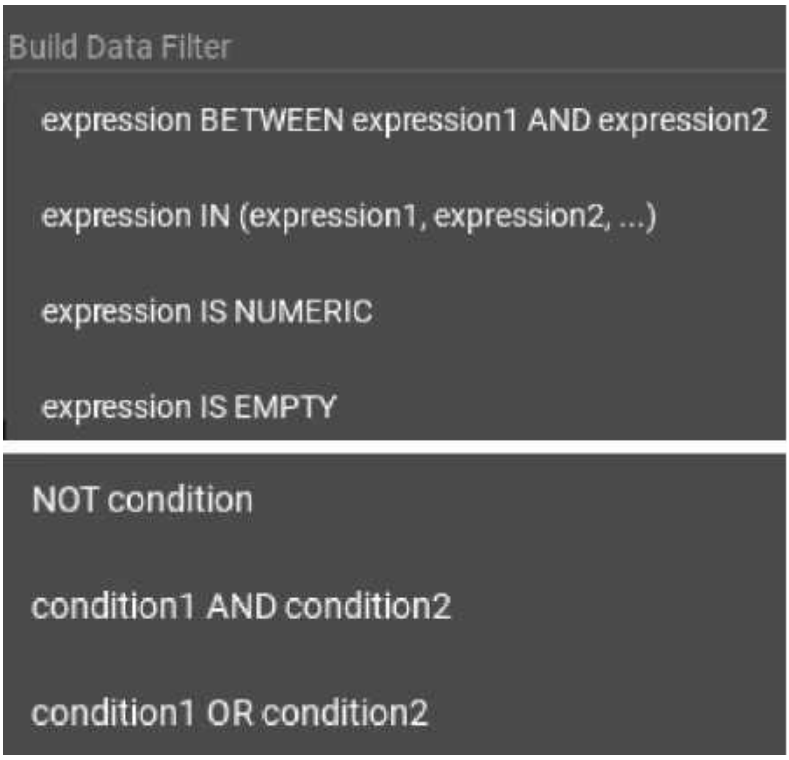


A data filter condition is a Boolean expression, which can be a nested set of conditions. GraphStudio evaluates the condition for each line in your input file. If the condition evaluates to true, then the line is loaded.

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:



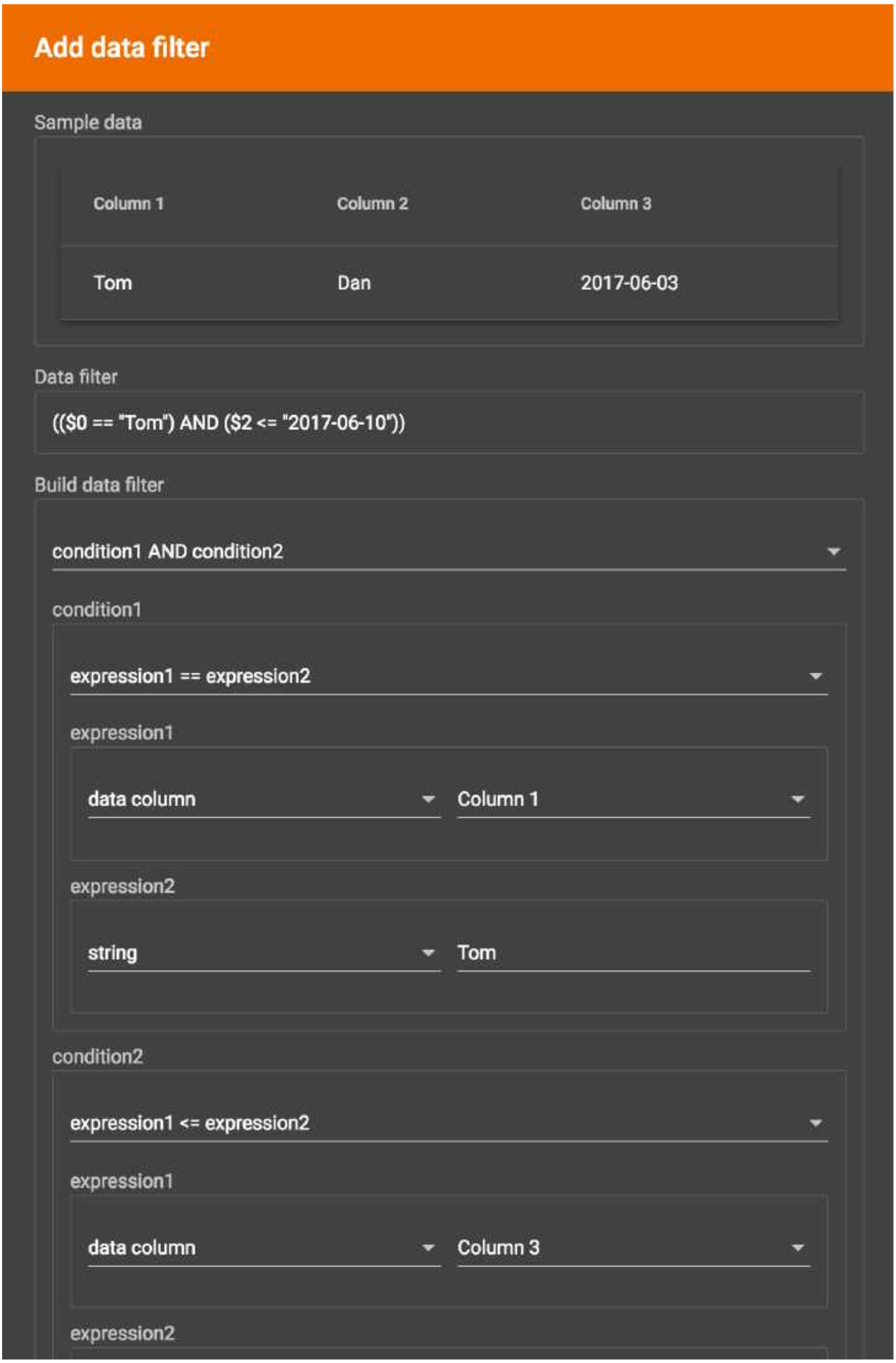
After this are several more options, using operators such as AND, OR, NOT, IN, BETWEEN...AND, IS NUMERIC, and IS EMPTY.

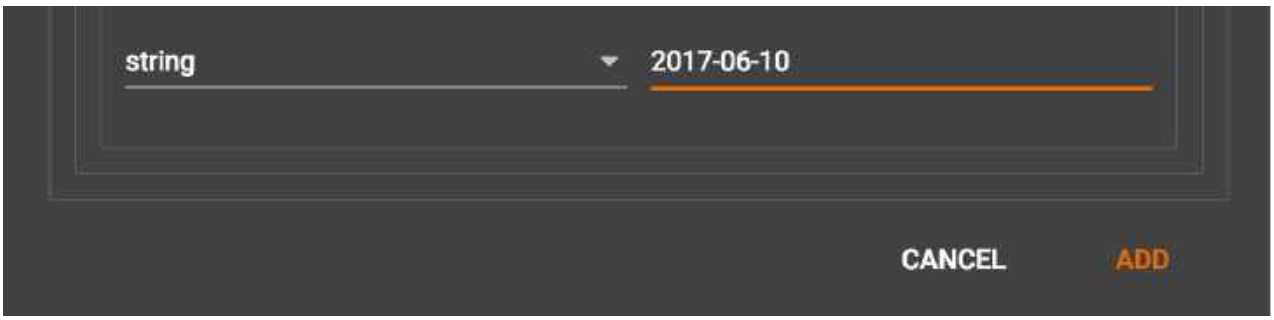


Note that each of these expressions calls for 1, 2, 3, or a list of operands, and the operands themselves can be expressions. When you select an expression, additional choosers will appear below, for you to specify the operand expressions. The operand choices are context-sensitive, but typically they include

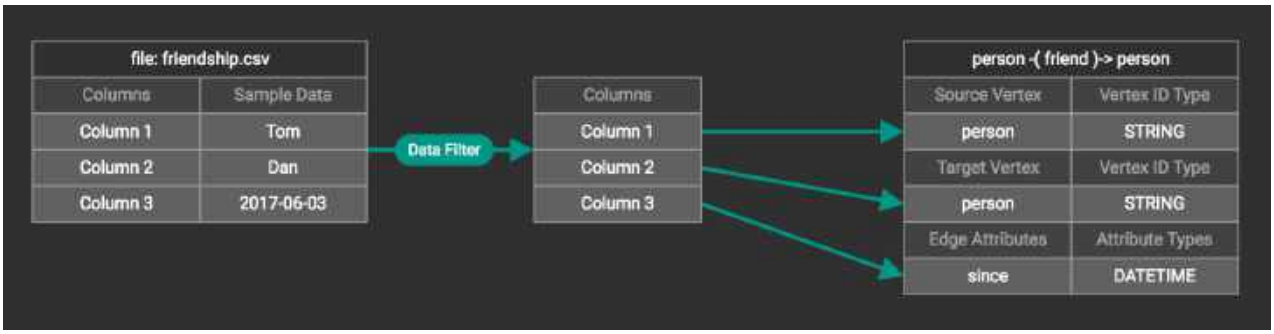
- a Data Column from the input file
- A constant value
- If the operator is AND, OR, or NOT, then the operand can be another condition. Thus is how conditions can be nested.

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.

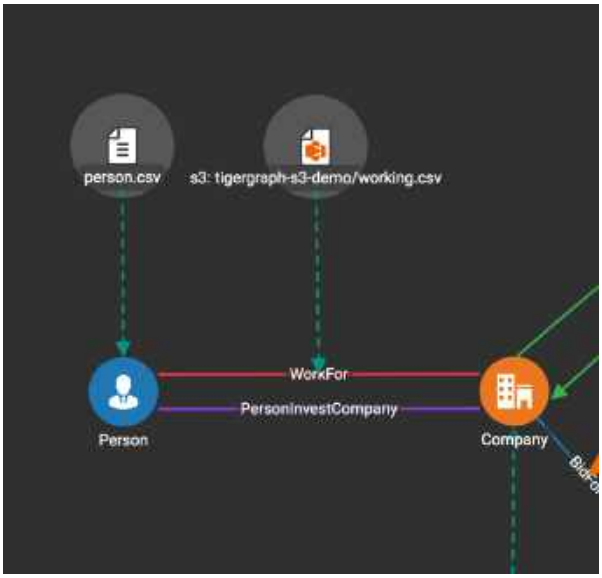
Delete Options

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.

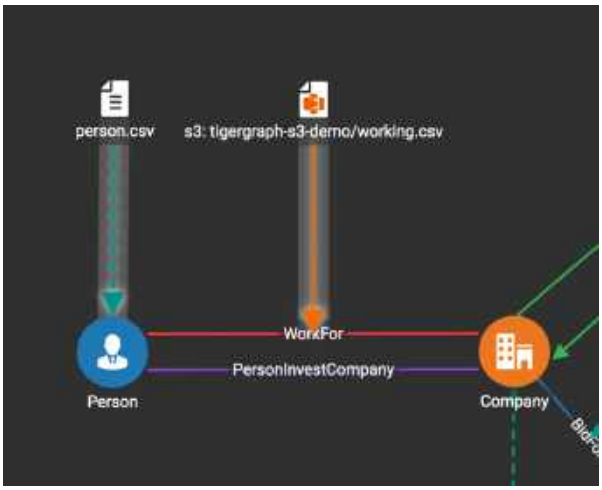
Delete Data File

Select the data file icon(s), then click the delete button.



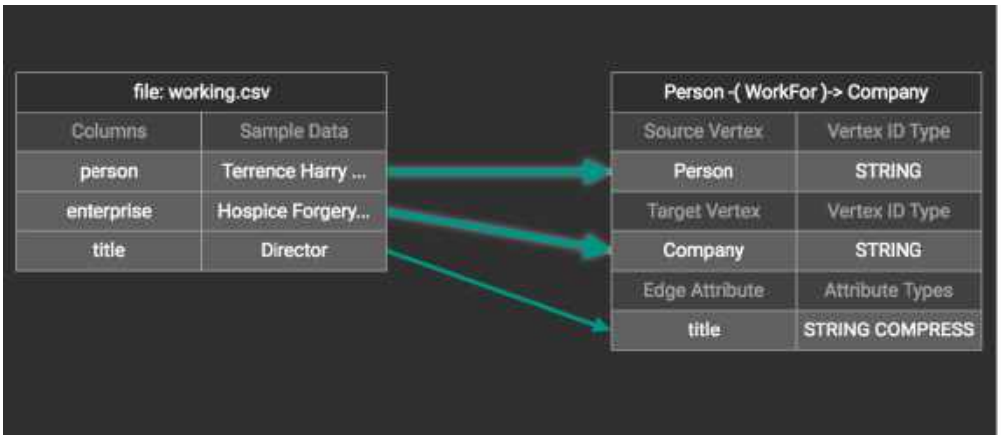
Delete Data File To Vertex Or Edge Mapping

Select the dashed green link(s) between data file and mapped vertex/edge type, then click the delete button.



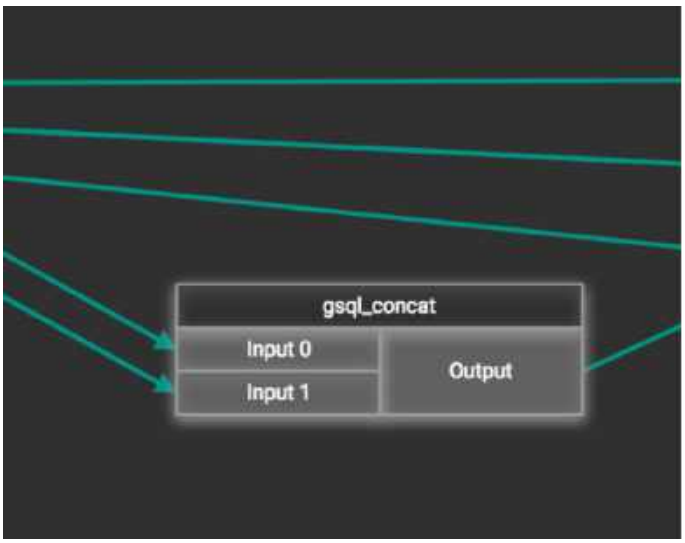
Delete Data Column To Vertex Or Edge Attribute Mapping

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

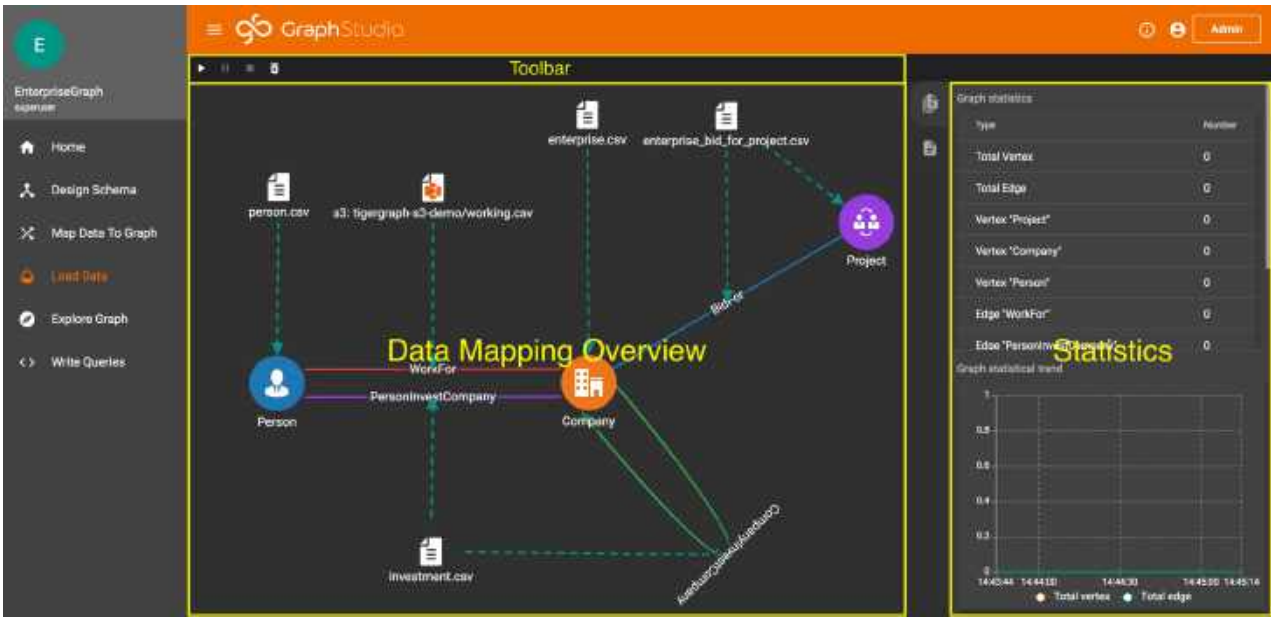
The following three buttons allow you to select the relative sizing of the left and right working panels:  .

By default, the two windows have equal widths. Click the left button to widen the left working panel, or click the right button to widen 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 on progress.

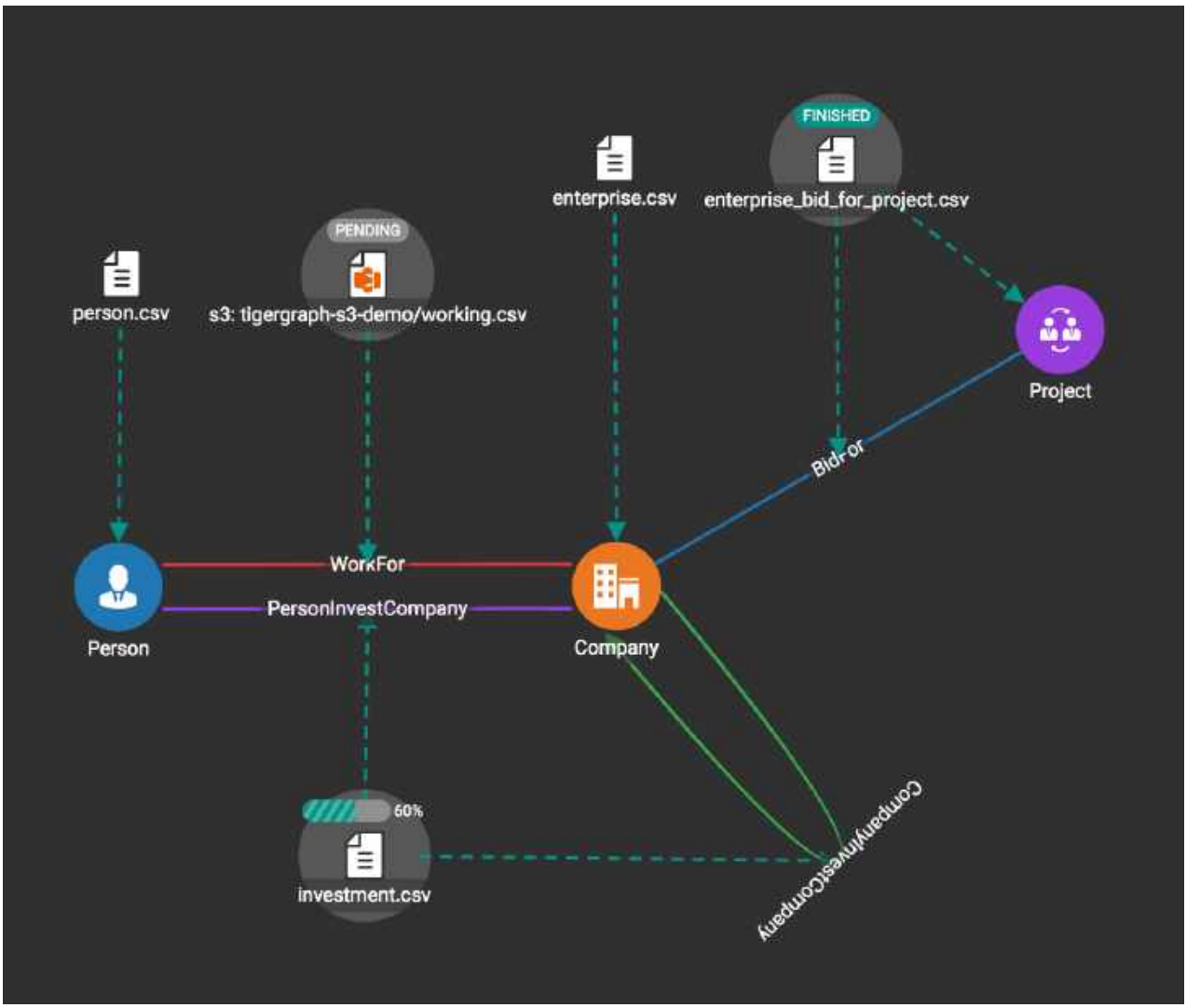
Start Loading

GraphStudio provides two types of loading:

- Partial Loading: load a subset of the data files which the user selects.
- Complete Loading: load all of the data files.

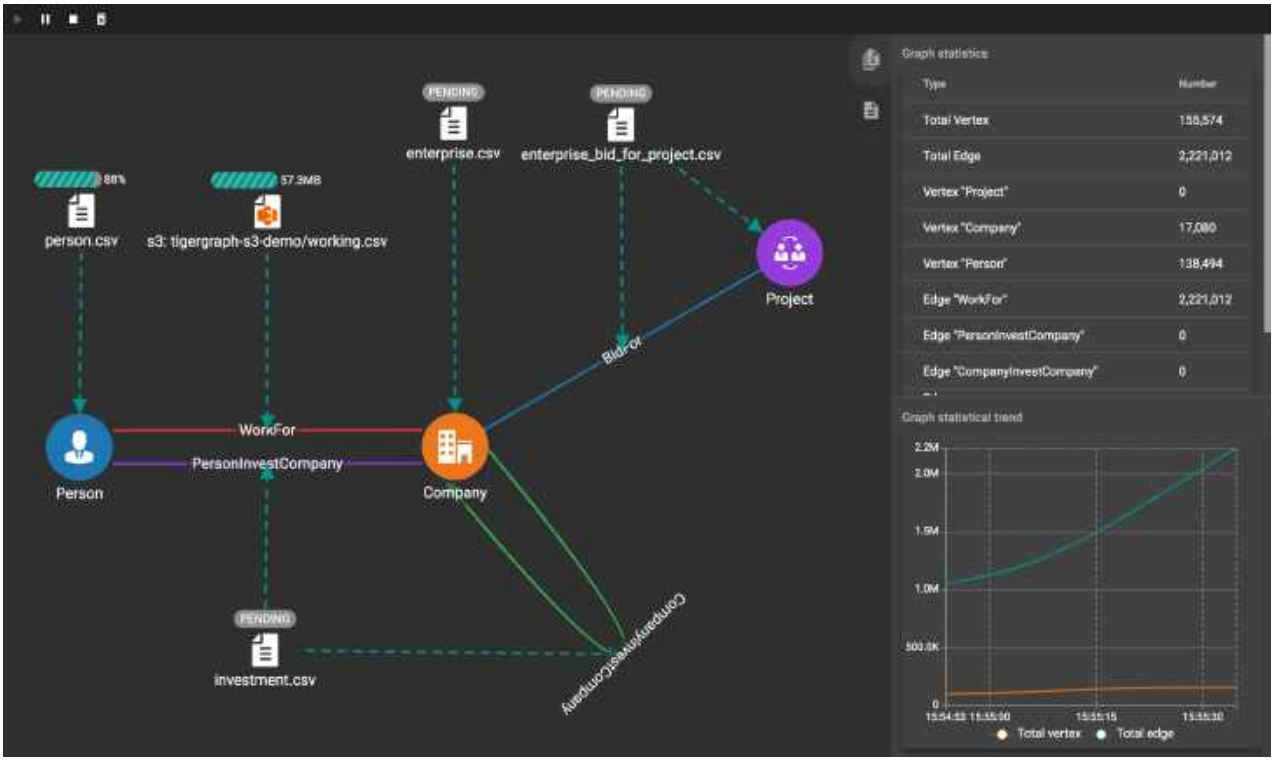
Load Some Data Files

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

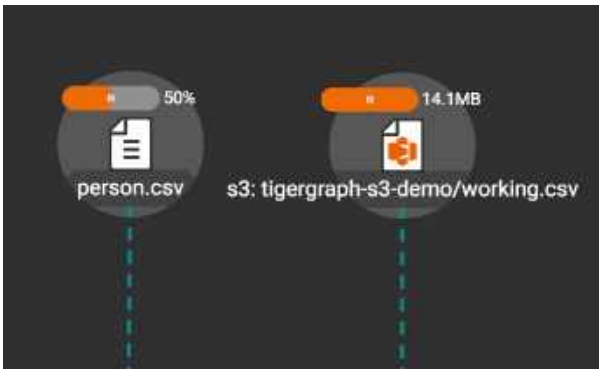
Click on a blank space in the data mapping overview panel to unselect the data sources, and click on the "start/resume loading" button ▶ on the toolbar. While loading is in progress a green hatched bar will appear over each data file to show its real time progress.



Pause Loading

Similar to Start Loading, you can pause loading some of the data files, or all loading data files.

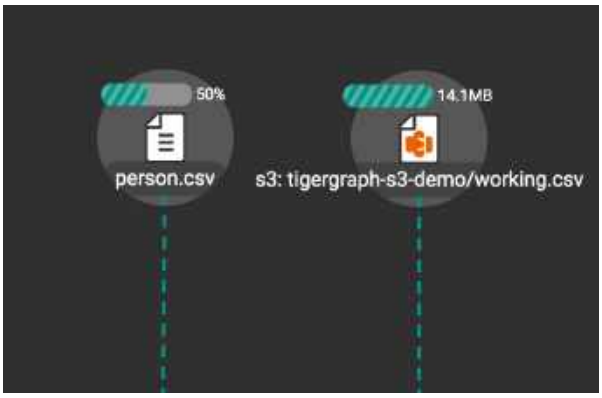
Select one or more data files (holding down the "shift" key to select multiple data files), and click on the "pause loading" button on the toolbar. In the Paused state, the progress bar will change to a solid orange color.



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

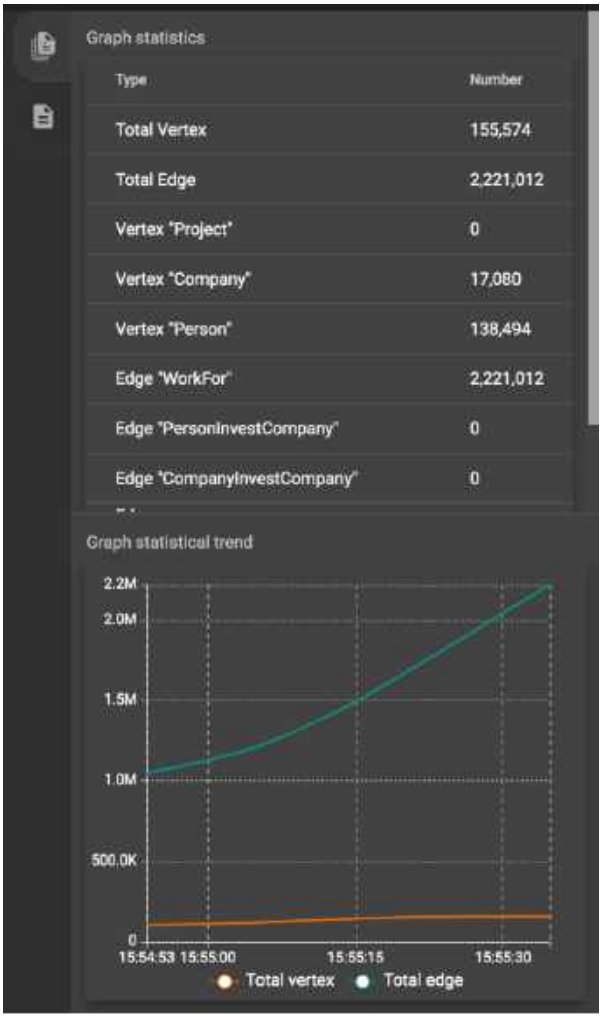
After loading has been started or paused, you can stop loading from these data files by clicking the "stop load" button ■. Similar to Start Loading, you can stop loading some or all loading data files. After stopping, the loading status of the data files will become "Stopped":

Statistics Panel

The Statistics panel contains two tabs: Graph Statistics (1st tab) and Data Loading Statistics (2nd tab).

Graph Statistics

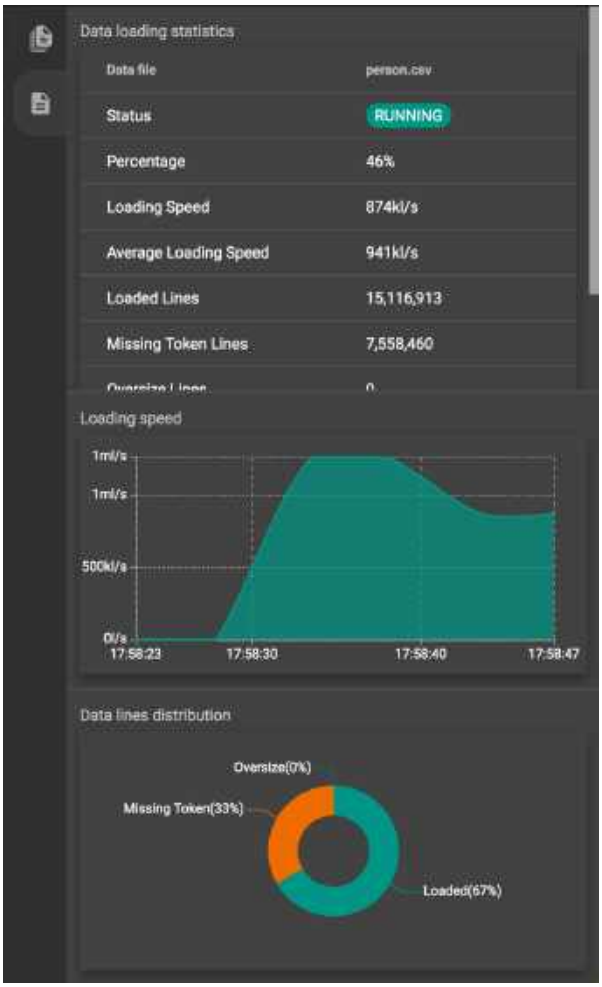
By default if no data file is selected, the Statistics panel will show Graph Statistics.



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

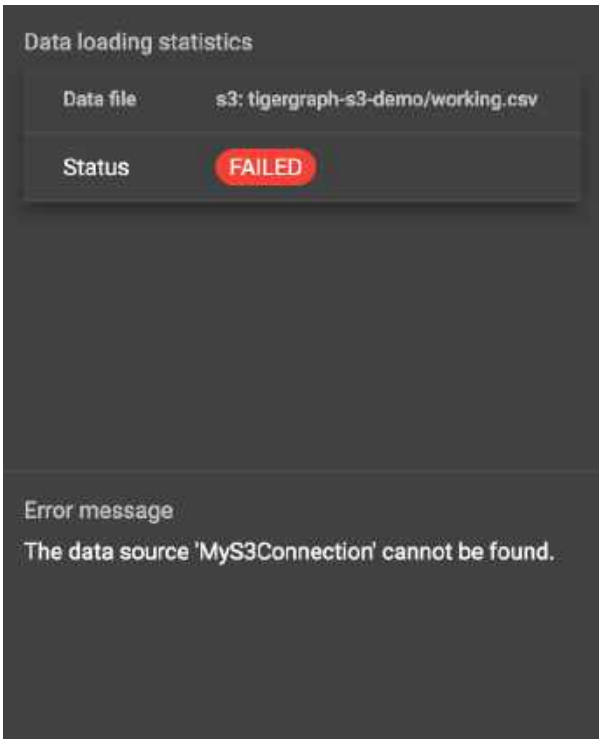
The area chart in the middle shows the real-time loading speed (lines per second) for this data file.

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.

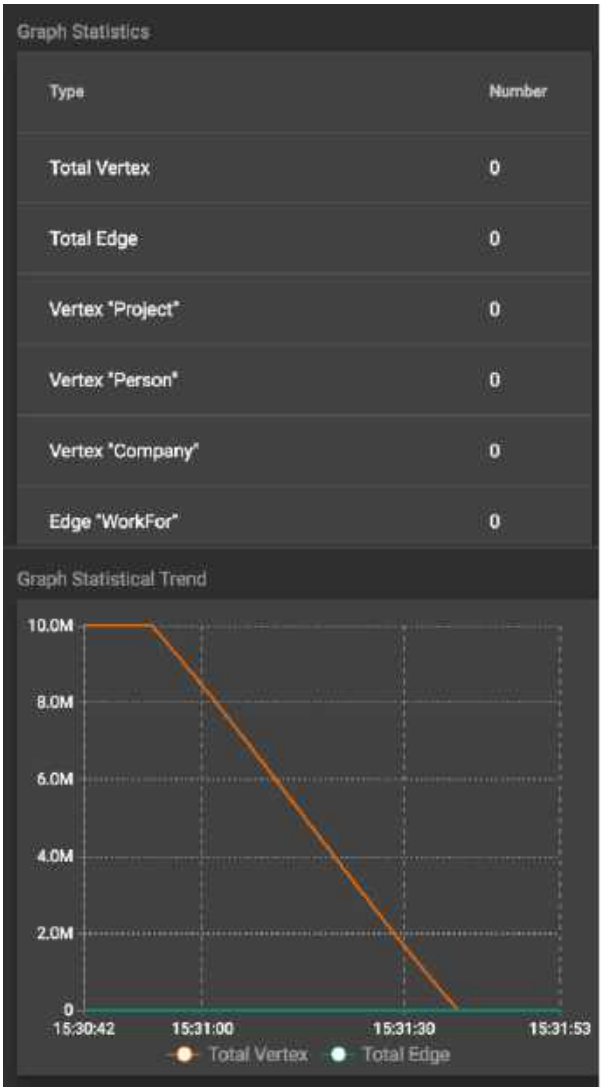
If data file loading encounters any issues and gets an error message, the error message will be shown at the bottom:



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.

After the clear operation, the graph vertex and edge number statistics will both drop to 0.

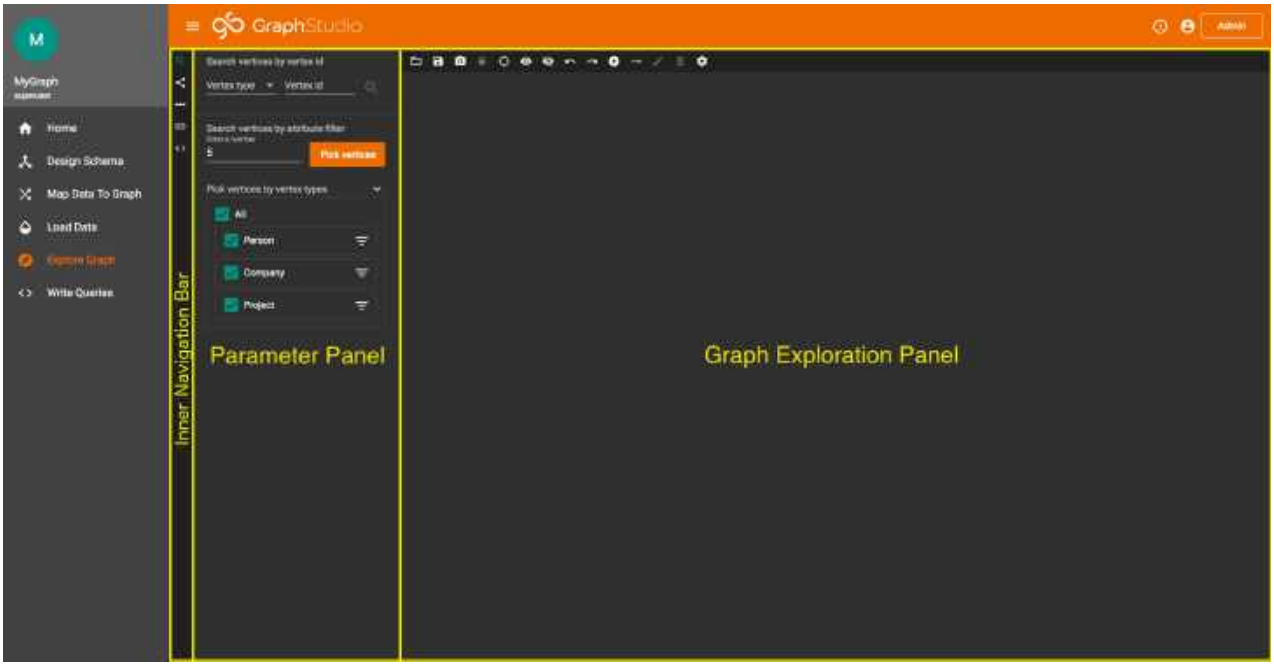


After data has been loaded, you can go to the Explore Graph or Write Queries pages.

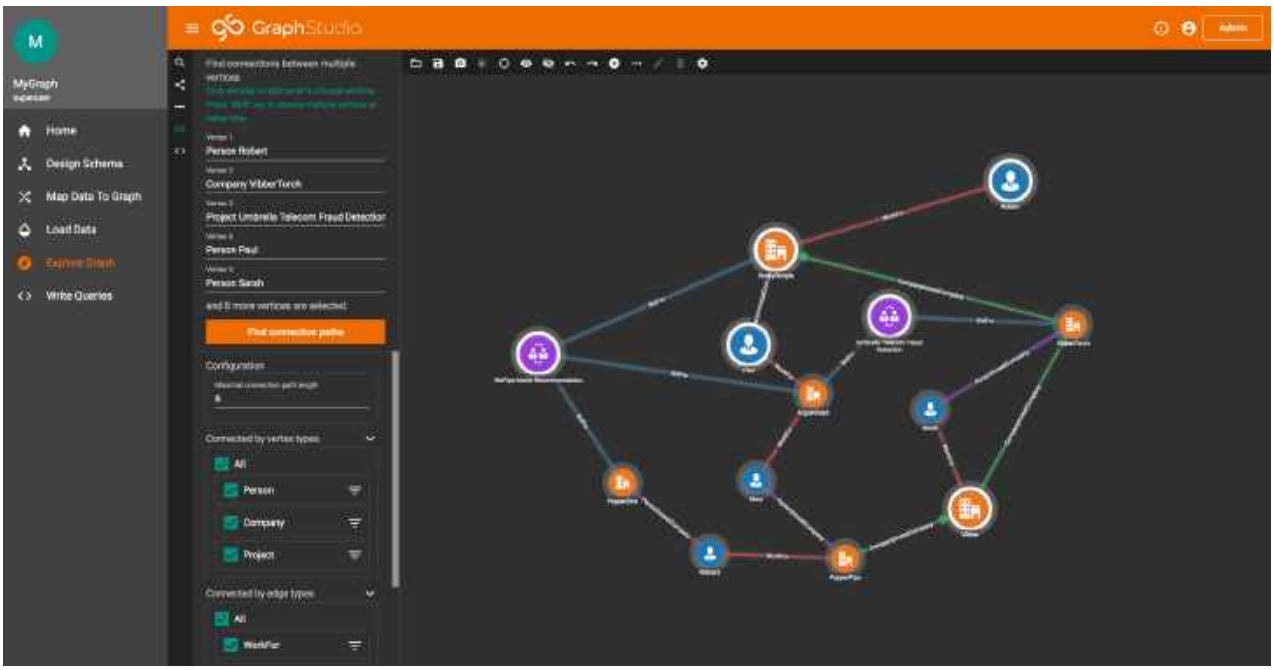
Explore Graph

Explore Graph

After data has been loaded, the Explore Graph page allows you to search for vertices in a graph, to discover nearby vertices which satisfy conditions of your choice, and to find the paths between vertices.



Below is an example of an exploration result:



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.

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 any more.

- **Save screenshot** : Save the current visualization result as a png file.
- **Change layout** : Arrange the vertices according to one of the built-in layout patterns, such as sphere, tree, circle, or force.
- **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.

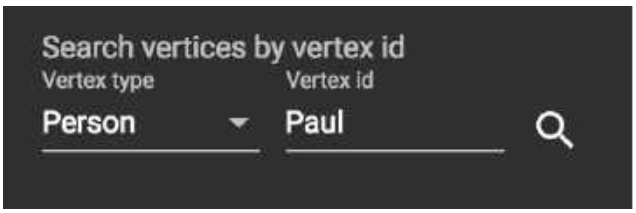
The Parameter Panel can be hidden by clicking its corresponding button in the Explore Graph Menu.

Search Vertices In Graph

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.

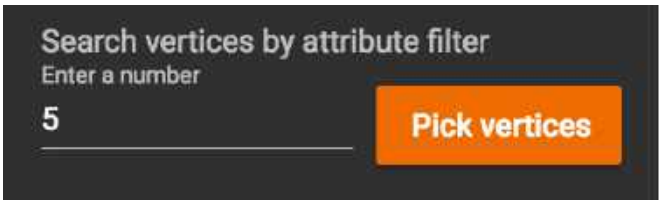
Search Vertices by ID

Choose vertex type from the Vertex type dropdown list, and enter the vertex id in the Vertex id input box, then click Search button. If there is one vertex that matches the vertex type and id, it will be shown in Graph Exploration panel.

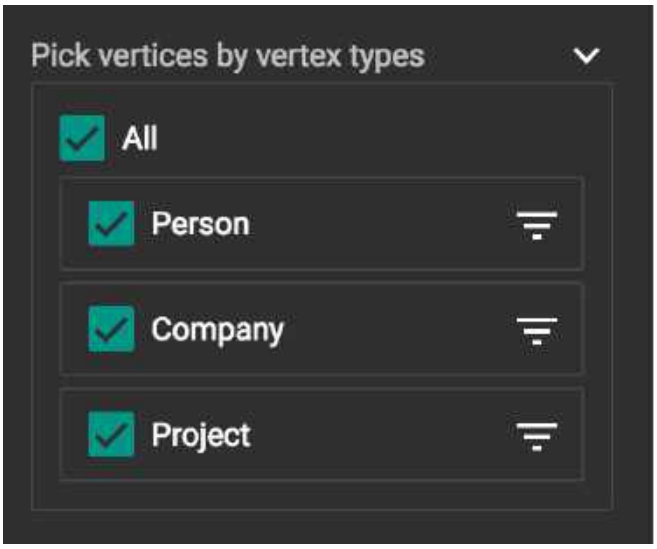


Let GraphStudio pick vertices

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 Pick vertices button Pick vertices. The explorer will pick this number of vertices for each vertex type included in your search.

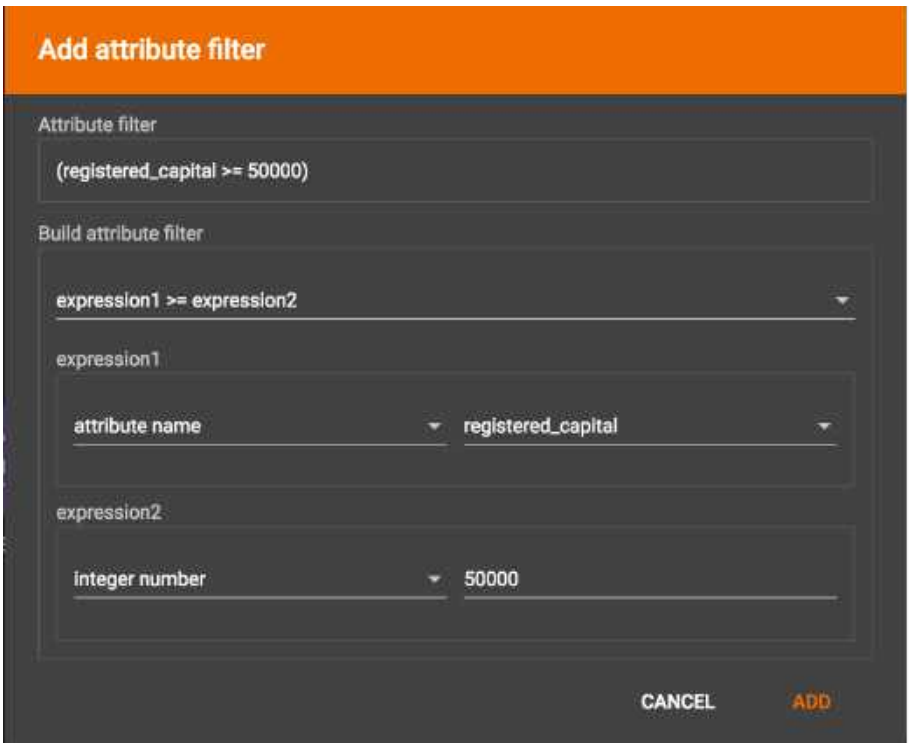


The Configuration section in the Parameter Panel specifies which types of vertices you want to include in your selection. By default, all vertex types are selected. Uncheck some boxes if you want to narrow your selection.

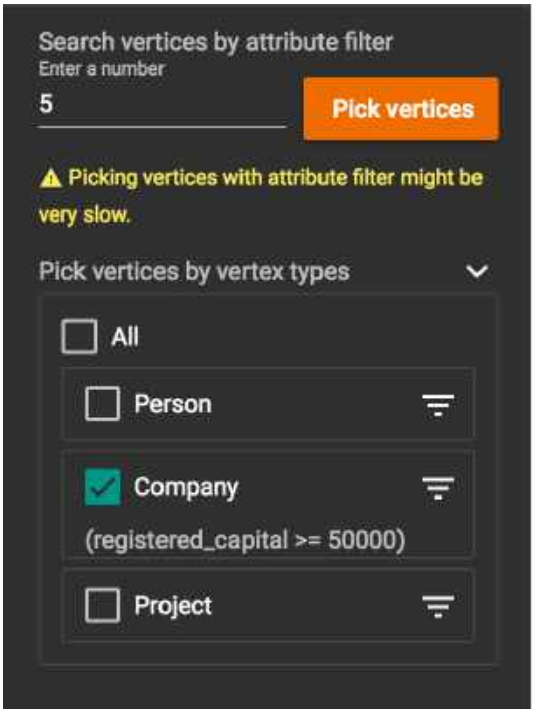


Search vertices with attribute filters

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 same as creating data filters when you do data mapping. Here is an example attribute filter for searching Company vertices with registered_capital >= 50,000:



Click ADD, then the filter condition is shown below Company vertex type:

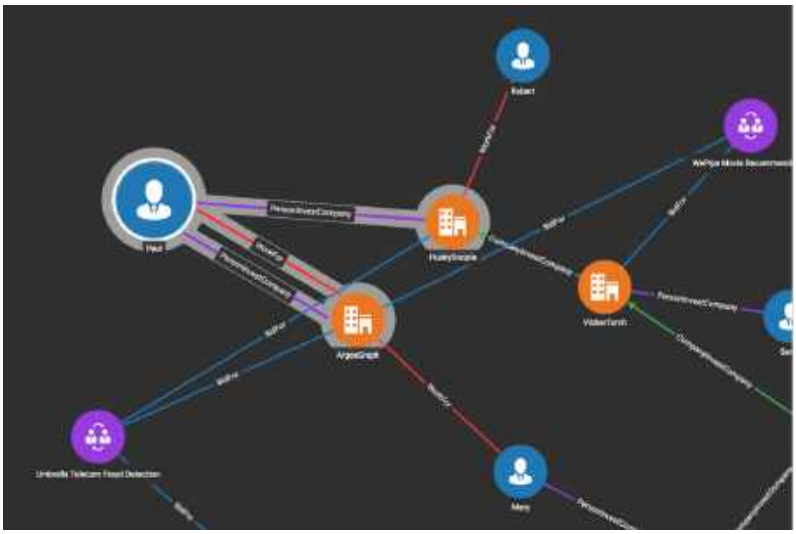


Click Pick vertices button Pick vertices again, TigerGraph will search for up to 5 Company vertices with a registered_capital >= 50,000.

If your graph contains a large number of vertices, searching vertices with attribute filters can be extremely slow. Attributes indexing support in TigerGraph is in our

roadmap.

NOTE: If you keep exploring the graph in 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 action 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.

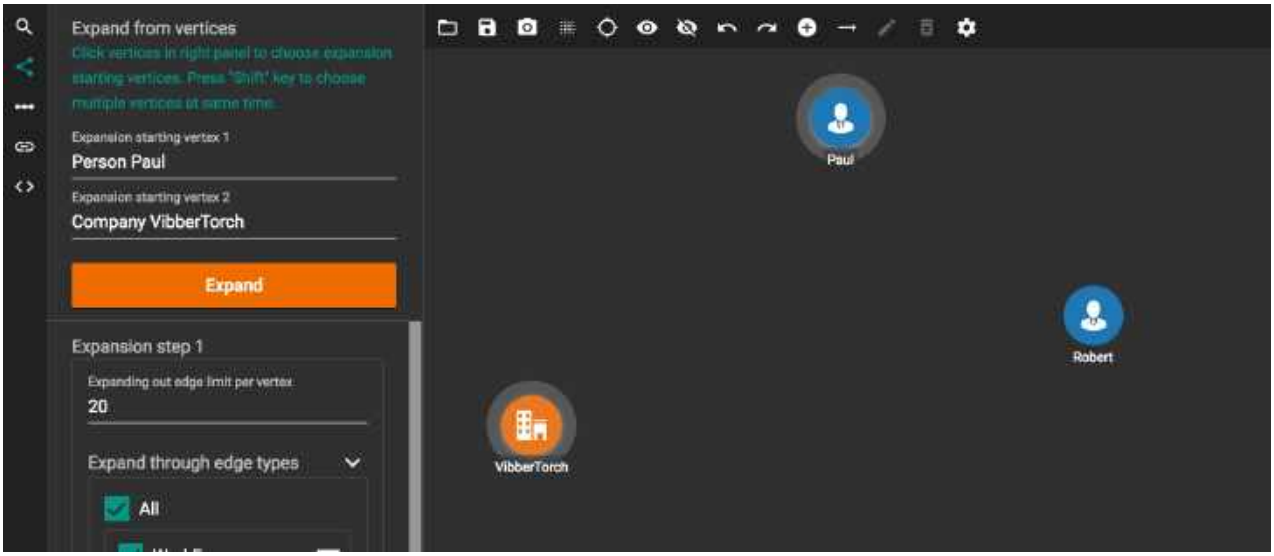
Shortcut: double-clicking on a vertex will expand to up to 200 neighbors of that vertex.

Choose Vertices To Expand

There may already be some selected vertices from the previous action. A vertex that is selected has a thick gray border around it. The standard click and shift-click behaviors for selecting one or multiple objects applies:

- 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.

To unselect all vertices, click on a blank area of the panel.

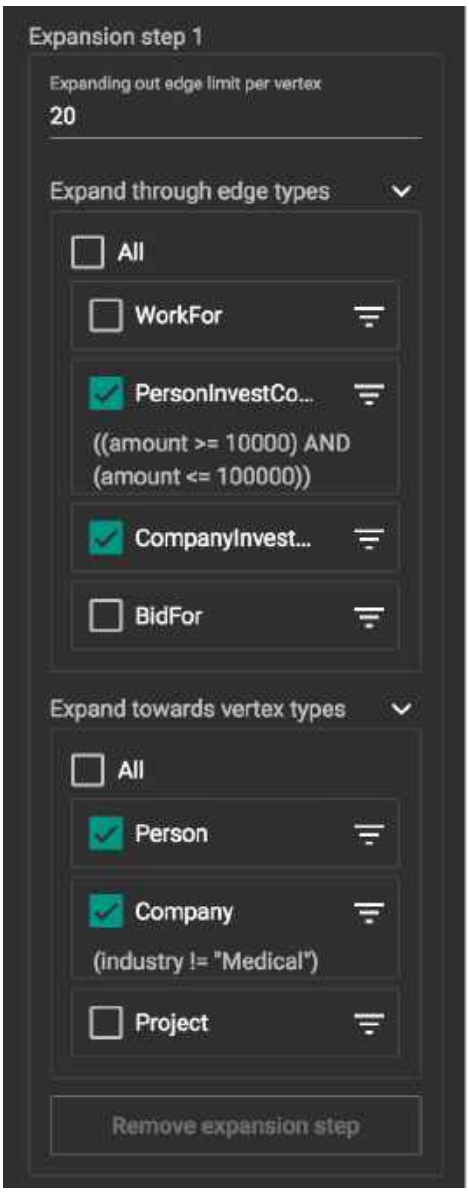


Set Expansion Conditions

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 which have 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.



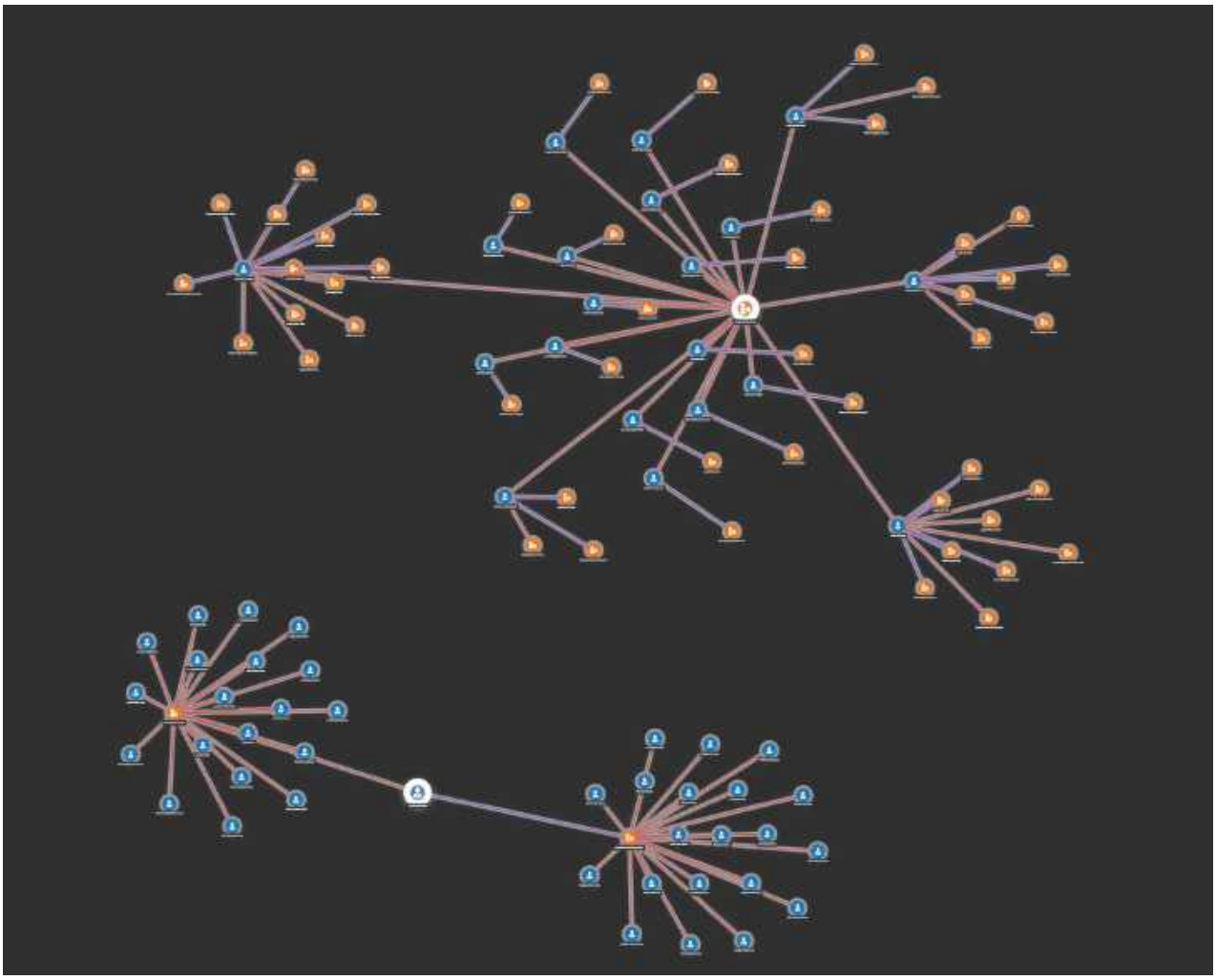
Initially, the expansion conditions panel for only one expansion step is shown. Click "Add Expansion Step" to add more expansion steps.

Similarly, you can remove expansion steps by clicking the "Remove Expansion Step" button.

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 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

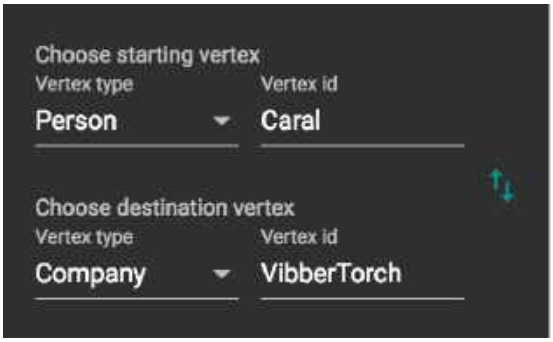
The top section of the Parameter Panel asks for your desired starting vertex and destination vertex.

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.



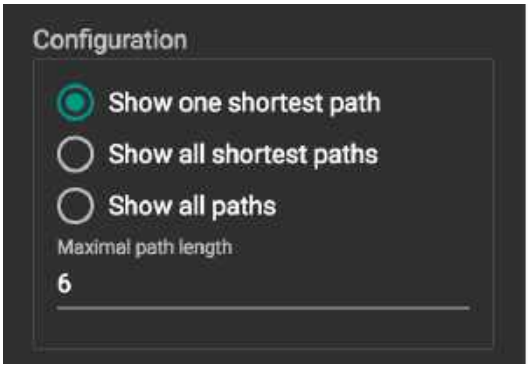
You can click the swap icon (two green arrows) at right to switch the starting vertex and the destination vertex.

Set Conditions For Paths

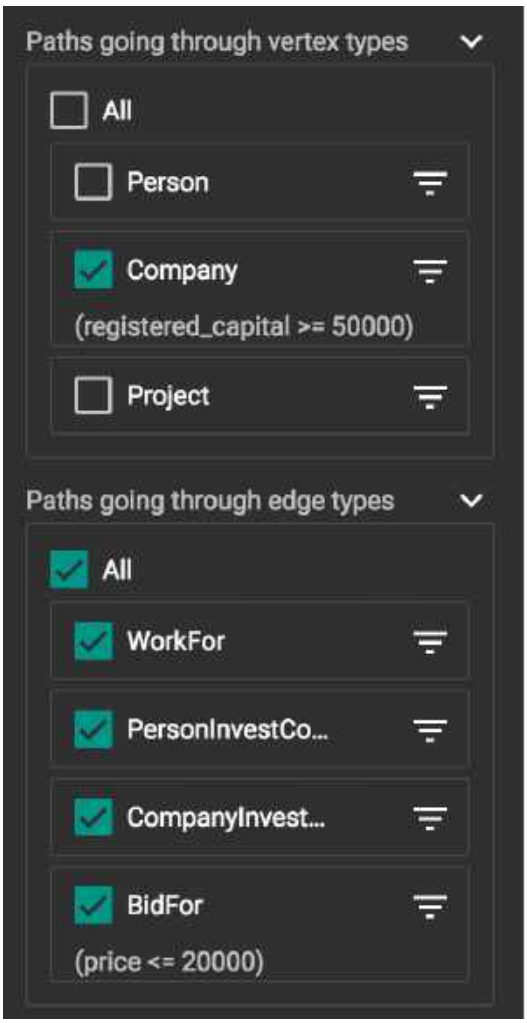
GraphStudio provide three types of path searches:

1. One shortest path: search for and highlight a 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

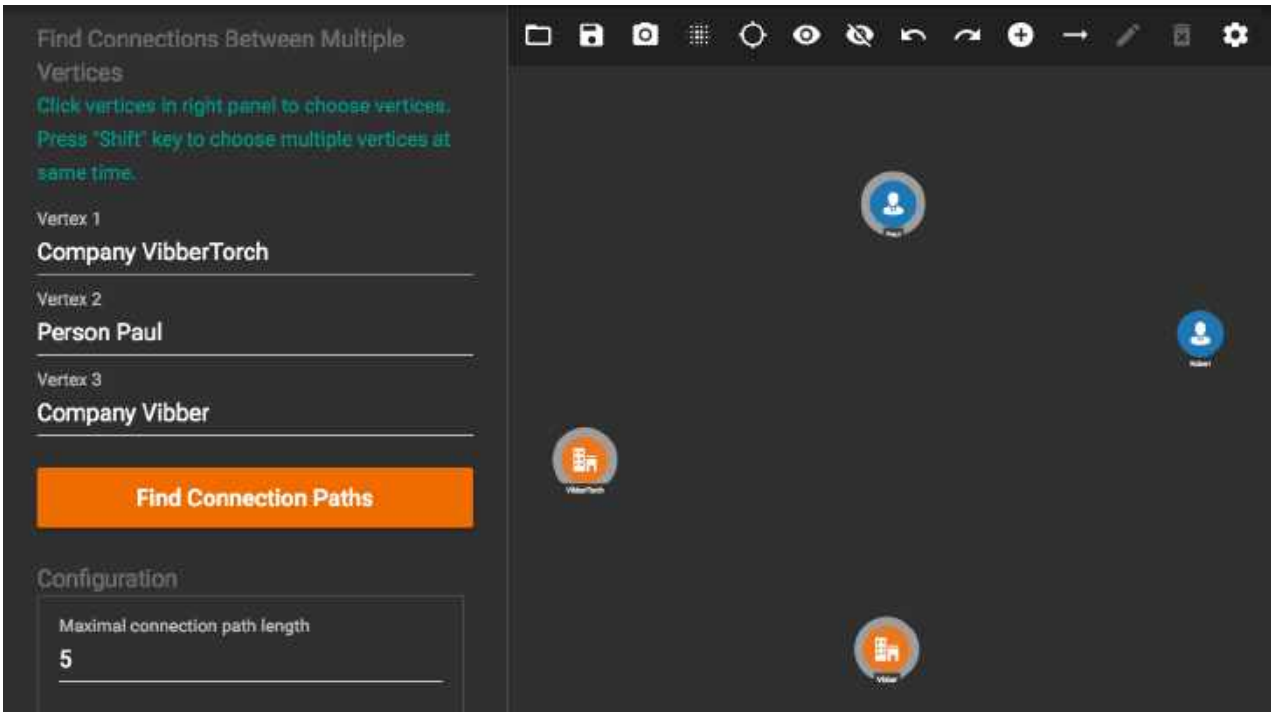
The fourth button in the Explore Graph Menu is the "Find connections" option. Given a set of starting vertices, this feature finds a "connection community" which is defined as follows:

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

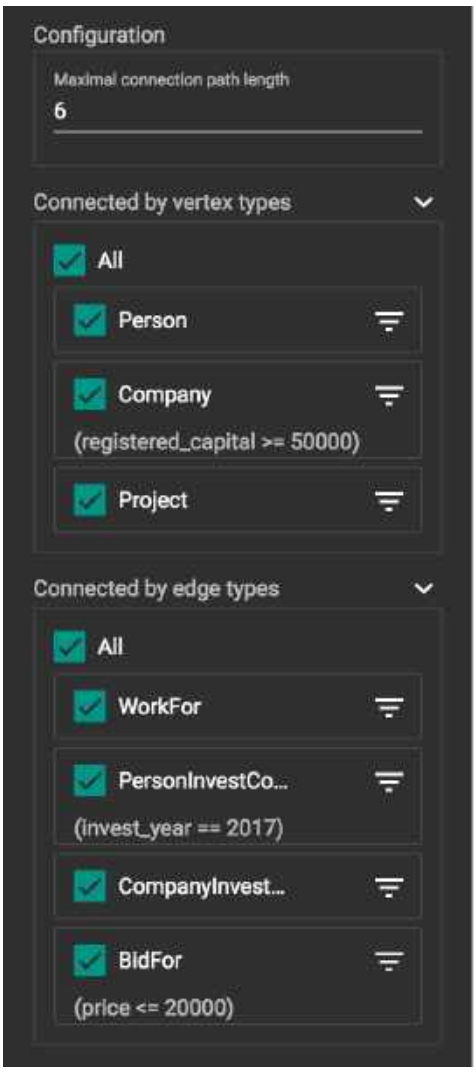
Click on a vertex to select it. Use shift-click to select more than one object. Each time you select another vertex, it will be added to the list in the Parameter Panel.



Set Conditions For Connection Finding

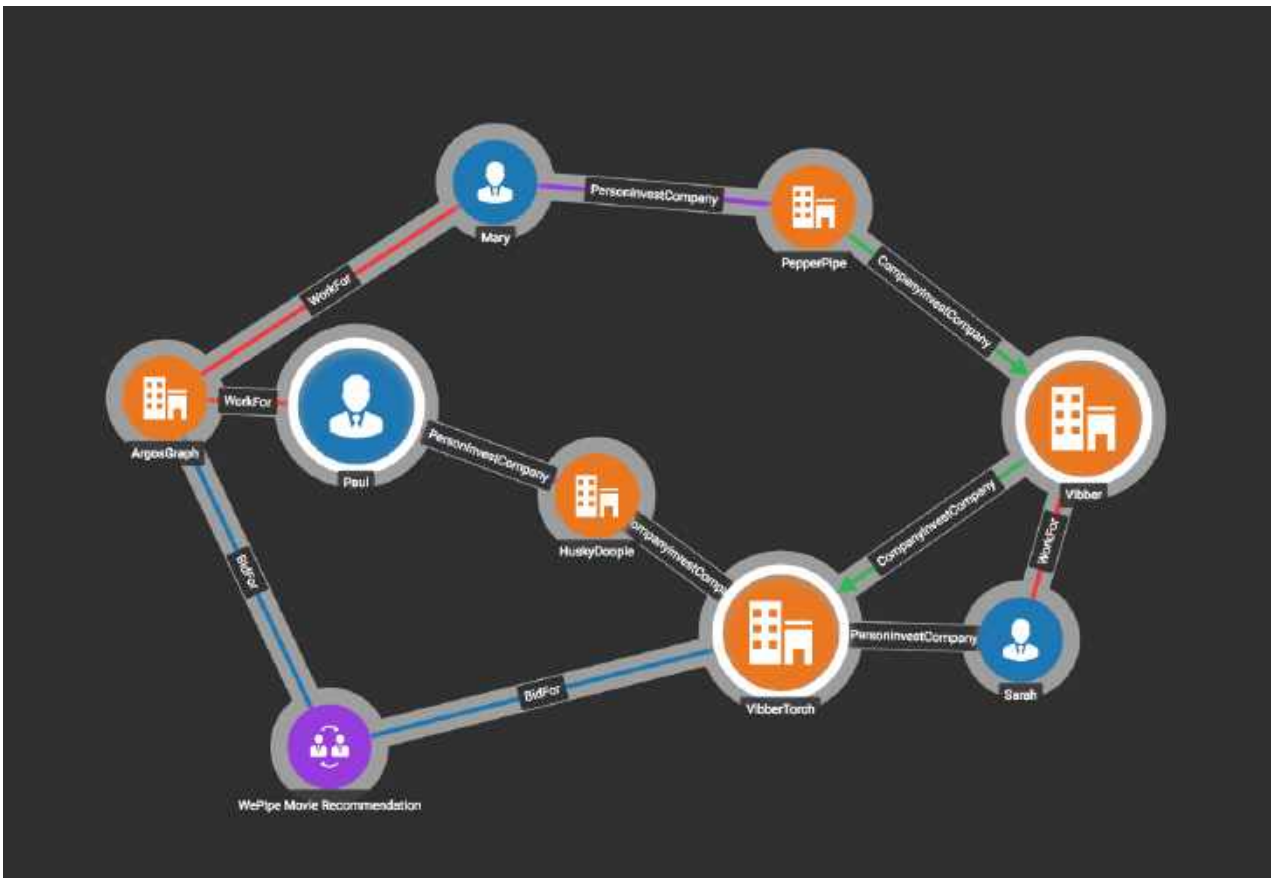
Since this query may have high computational cost if the graph is very large, a parameter is available to limit the path length.

You can also specify the valid vertex types and edge types which may be included in the connections.



Find Connections

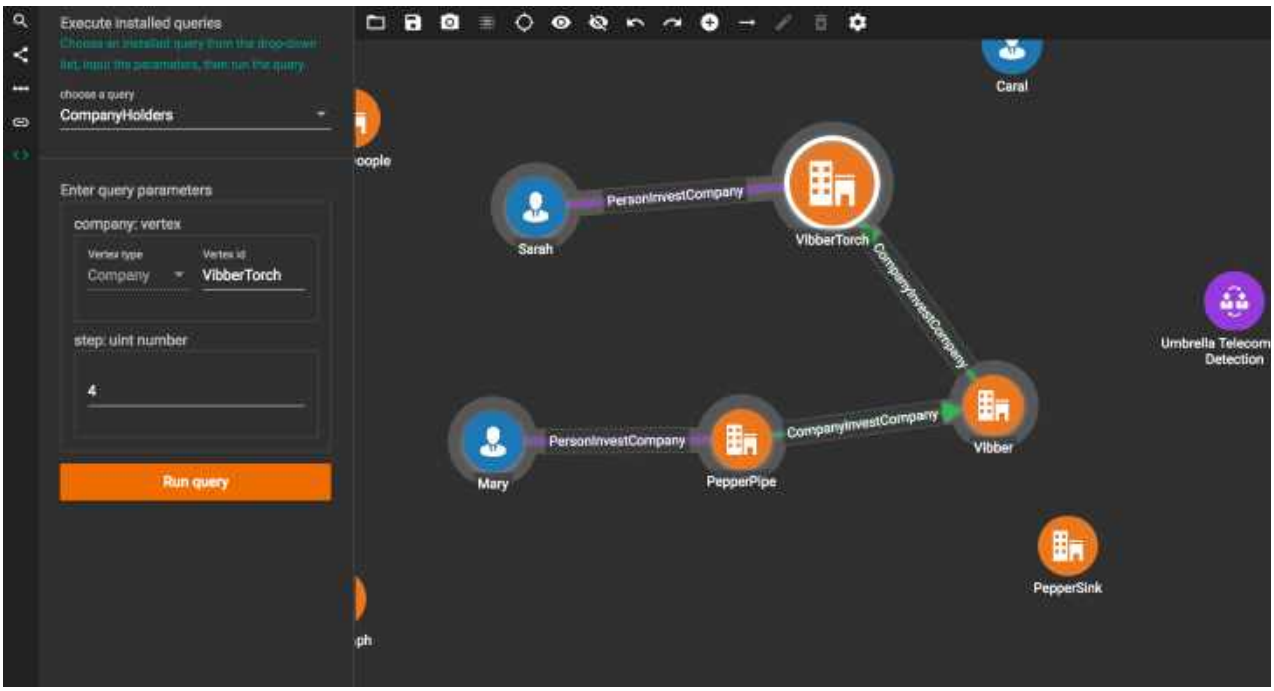
After selecting the vertices and setting the search conditions, click on the "Find Connection Paths" button  to start the search.



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 Run query. The query execution result subgraph will be merged with previous graph exploration result and highlighted:



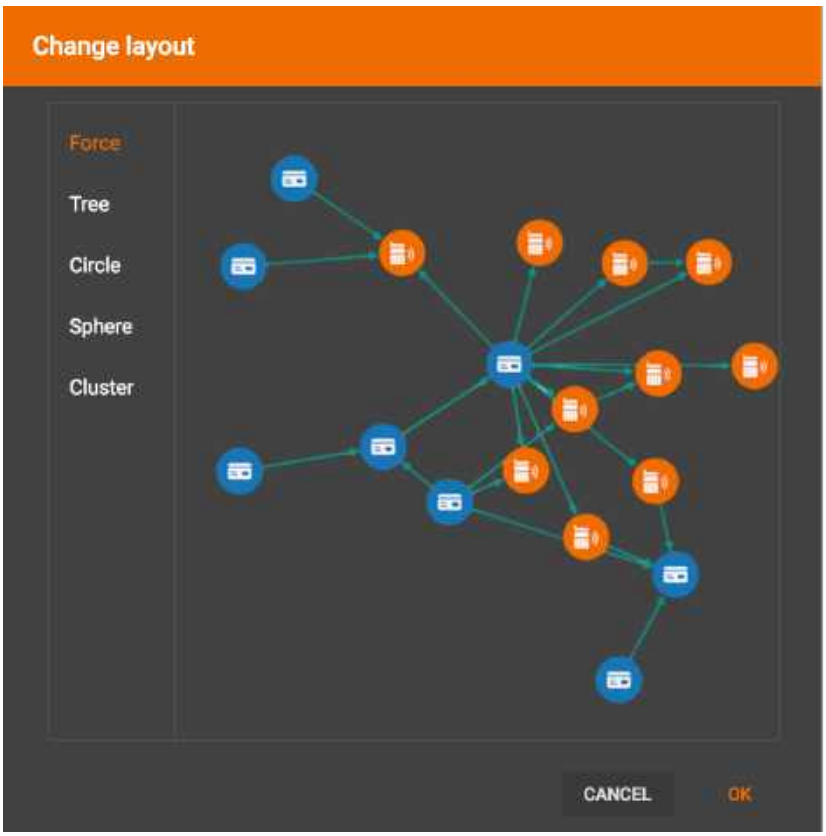
Allowing running GSQL queries mixed with other graph exploration functionalities enables better data analysis possibilities since you can refer to your previous exploration result, and keep gaining insights from your data.

Graph Exploration Panel Options

After you have a subgraph displayed in the Graph Exploration Panel, you can use the buttons in the Explorer View Menu to customize the display. You can even make modifications to the graph database itself.

Change Layout

Click the Change Layout button to select one of the built-in layout styles for systematic arrangement of the vertices. The Change Layout popup menu shows a sample of each layout style, for a dummy graph.



Locate Vertex In Result

The Locate Vertex In Result feature searches for and then zooms in on vertices which 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).

Add New Vertex

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

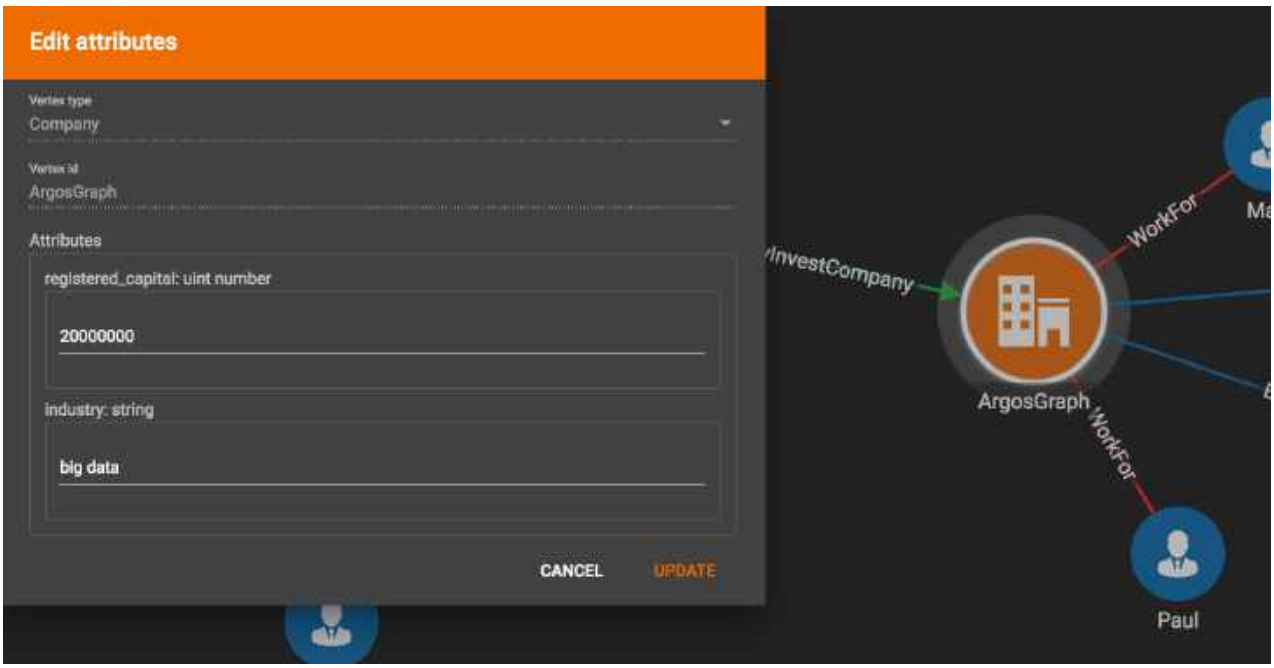
Click the Add New Edge button to add a new edge to the graph database. Next, click the source vertex of the edge in the Graph Exploration Panel, and then click the target vertex of the edge. Then the Add New Edge panel will pop up. Choose the edge type from the dropdown menu. Only types that match the 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. The edit attributes panel will pop up.



When you finish editing, click the Update button to apply the change.

Delete Vertices And Edges

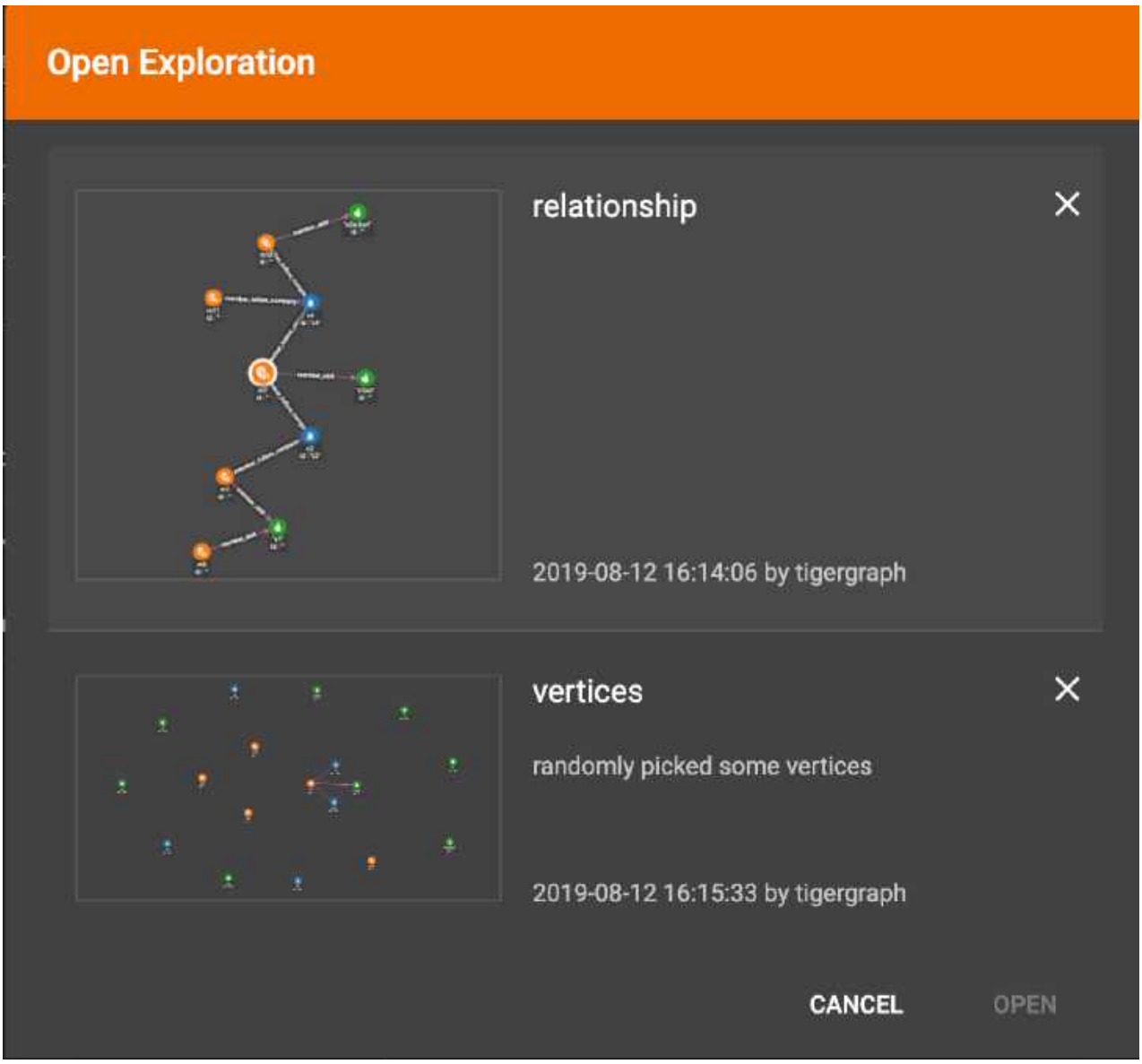
To delete vertices or edges, select the objects you want to delete, and click the Delete Selected Elements button .

> ⚠ "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.

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:

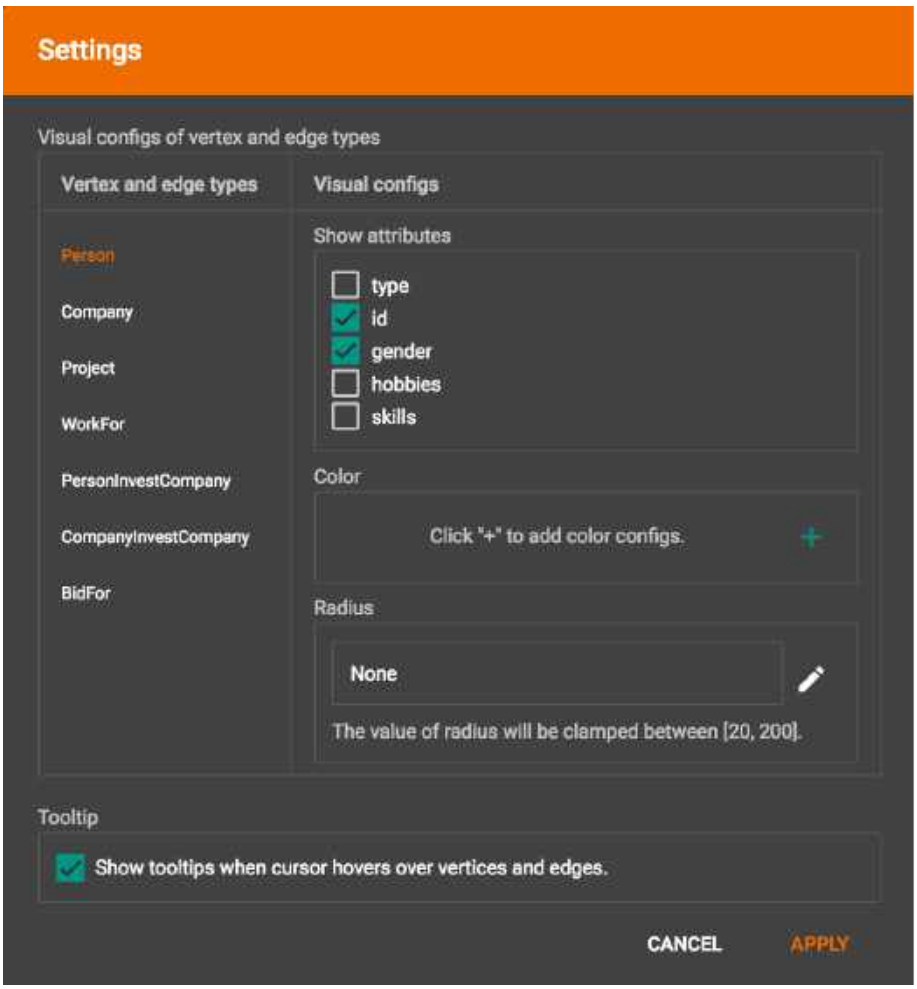


Save Screenshot

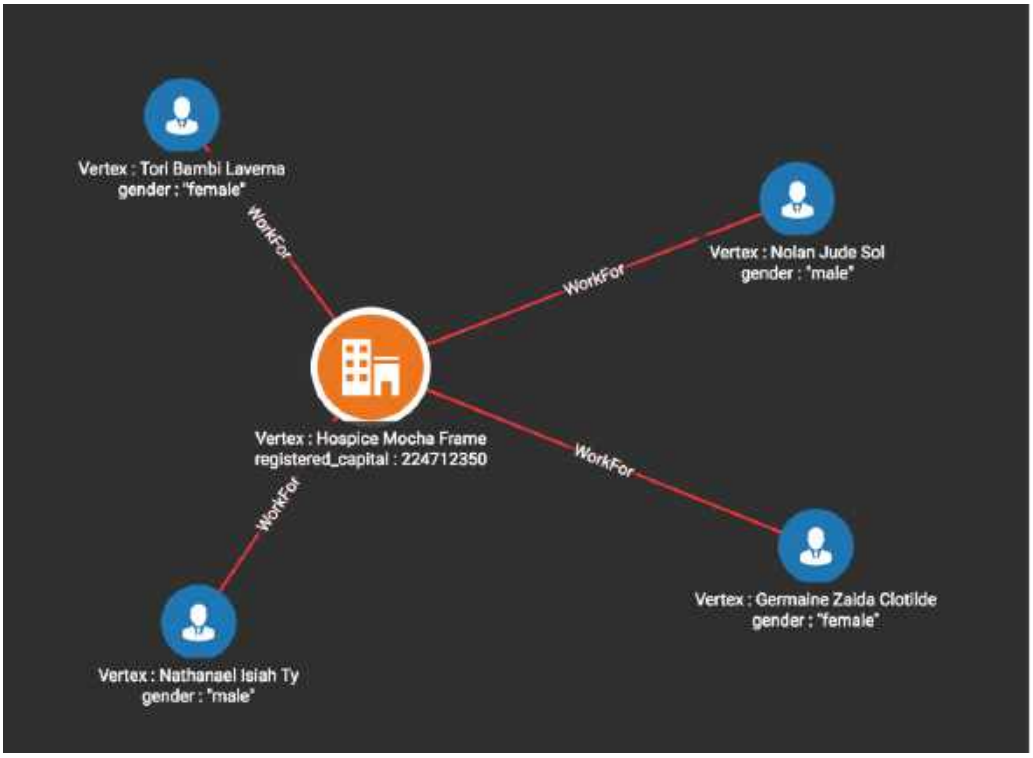
When you find something interesting during exploration and want to save the result as a picture, you can click the Save Screenshot button. The exploration result will be saved as a PNG picture to your local file system.

Change Settings

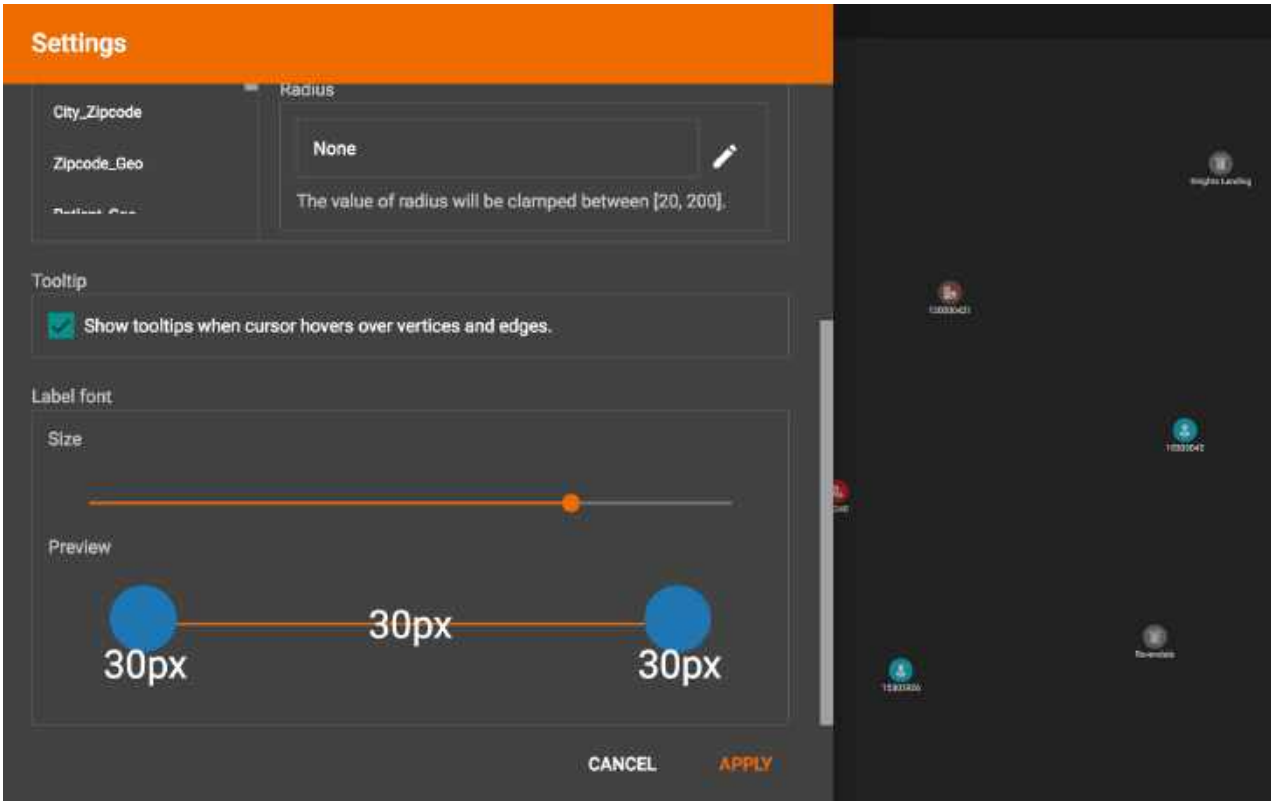
To change graph exploration settings by clicking Settings button. Currently you can select what attributes to show for each vertex type and edge type, and set whether to show an object's detailed information in a popup tooltip when the cursor hovers over it. Click Apply and the new settings will take effect.



In the example below, the ID and gender for Person vertices are shown. The ID and the registered_capital attribute for Company vertices are shown.



You can also config the label size of vertices and edges.

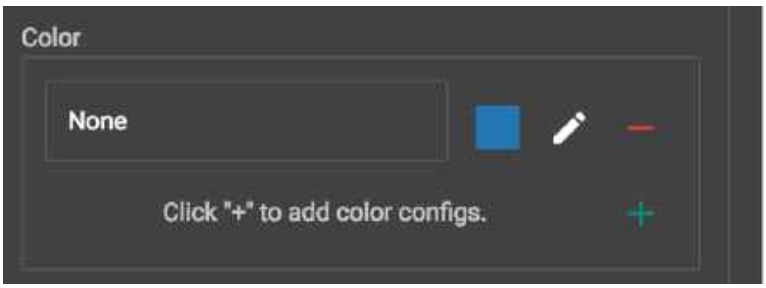


Other than the above, you can also config vertex and edge size and color to augment the visualization in settings. It is so important that we will use next independent section to introduce.

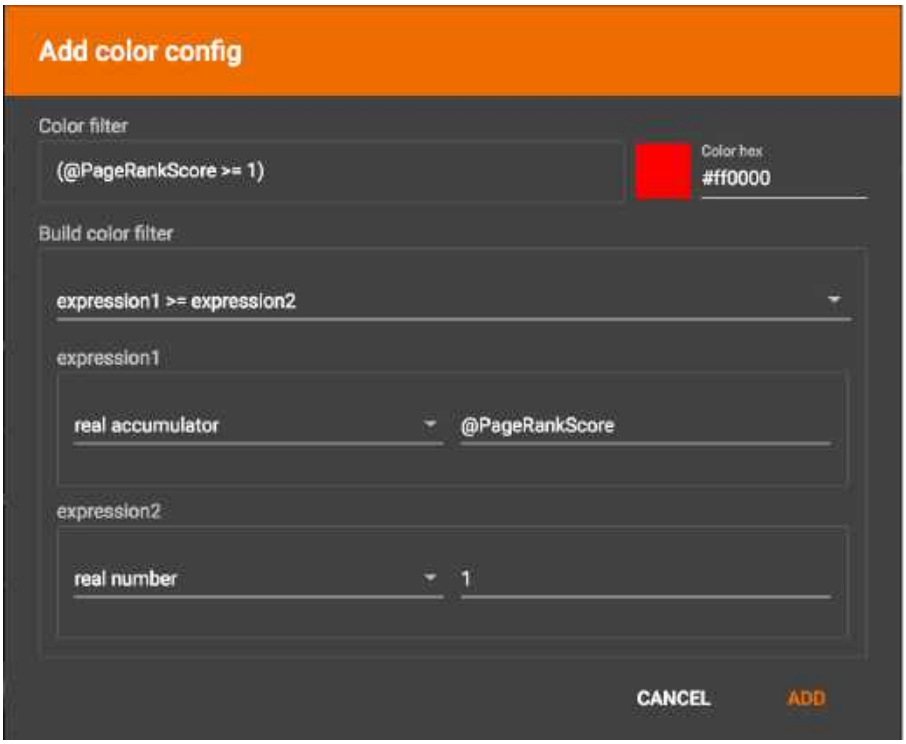
Augment Graph Visualization Result

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 config 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 config entry appears:



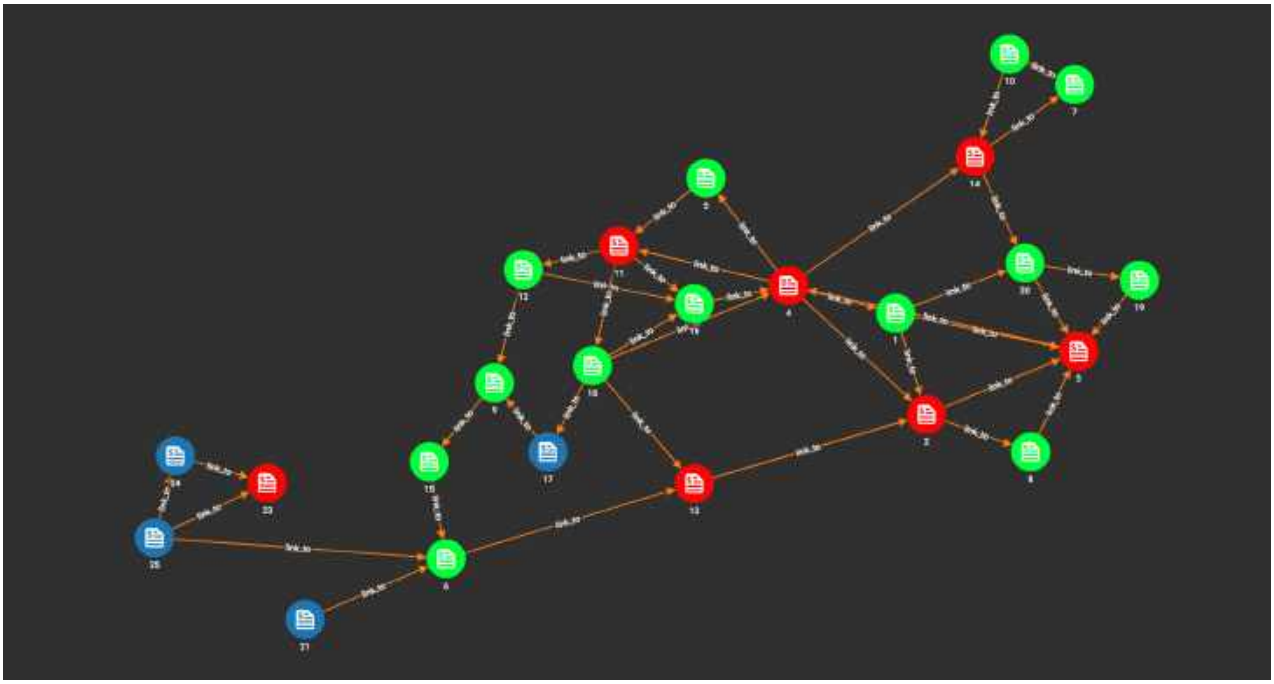
Click the Edit color config button, in the pop up window choose red color, and build a condition specifying @PageRankScore >= 1.0:



Click ADD, and the condition and updated color is shown in the Color settings section:

Similarly, you can add another color config 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 color of edges.

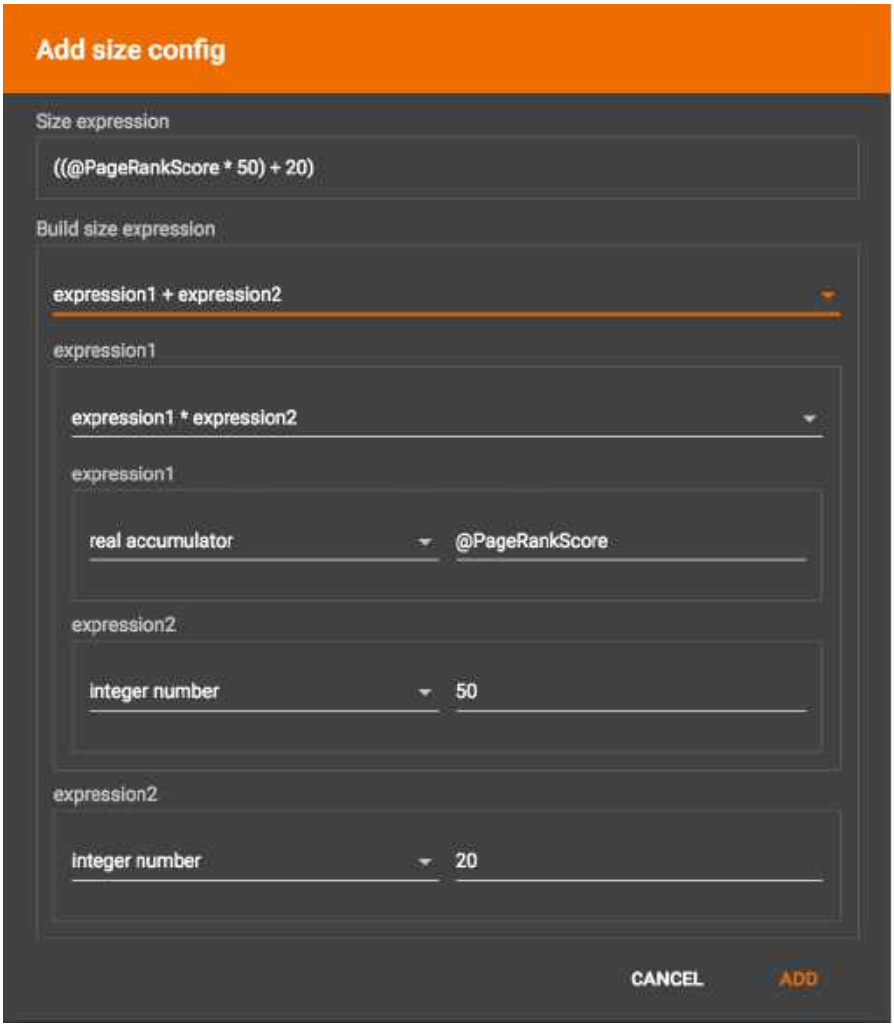
If you want to cancel one color configuration, just click the remove button — to the right side of that configuration.

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 config vertex radius and edge thickness according to their attributes or numeric

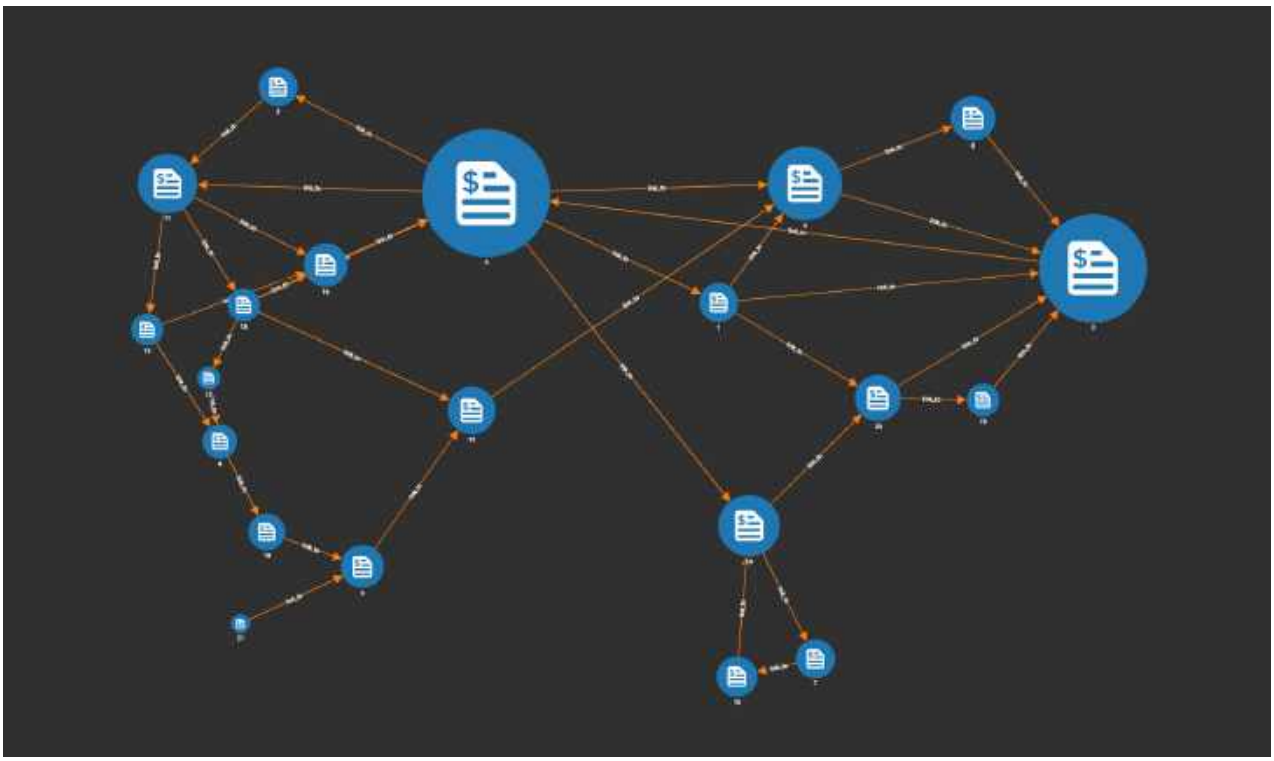
accumulator values of GSQL query result. A classical example is page rank. You can set vertices radius proportional to their page rank values, then the importance of each vertex is visually apparent according to its size.

First choose the vertex type you want to config its radius, then click the Edit button in Radius section. In the popup window you can create the radius expression:



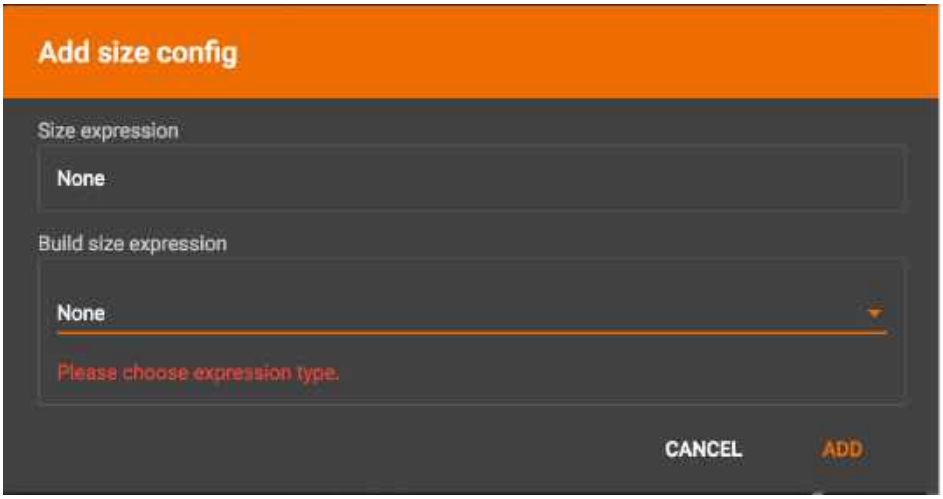
After click ADD button, the radius expression will be shown in Radius section:

After click APPLY button, the vertices will be rendered in different size according to the expression value:



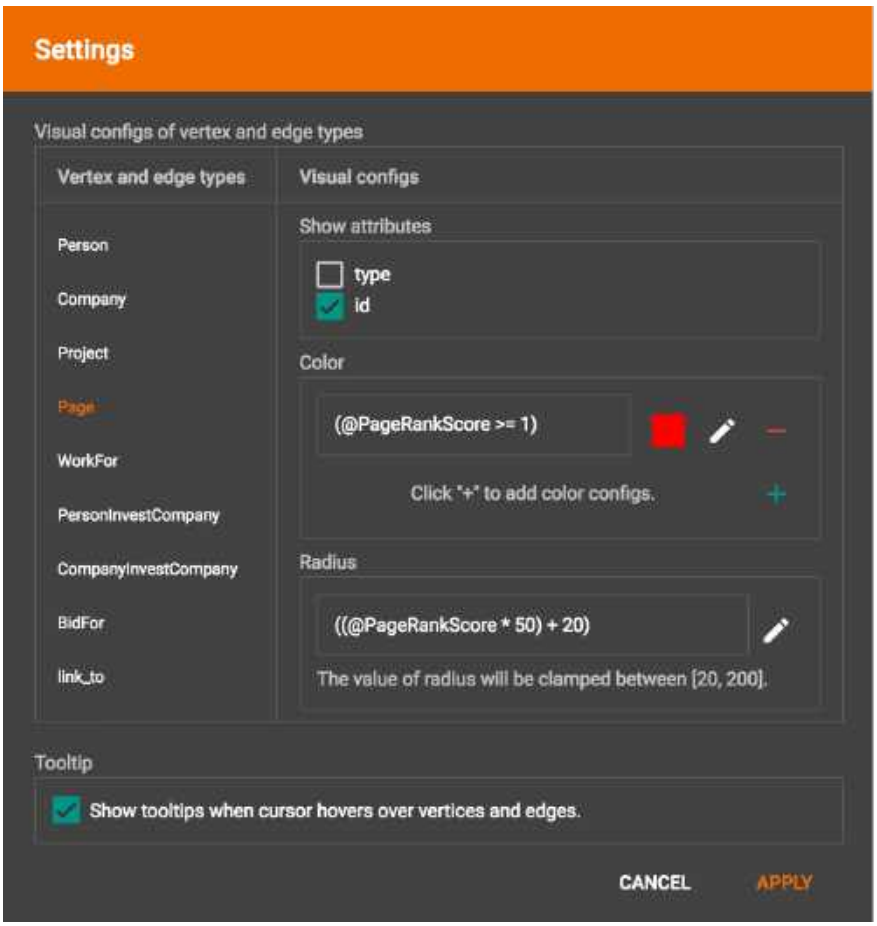
Similarly, you can config different thickness for the edges.

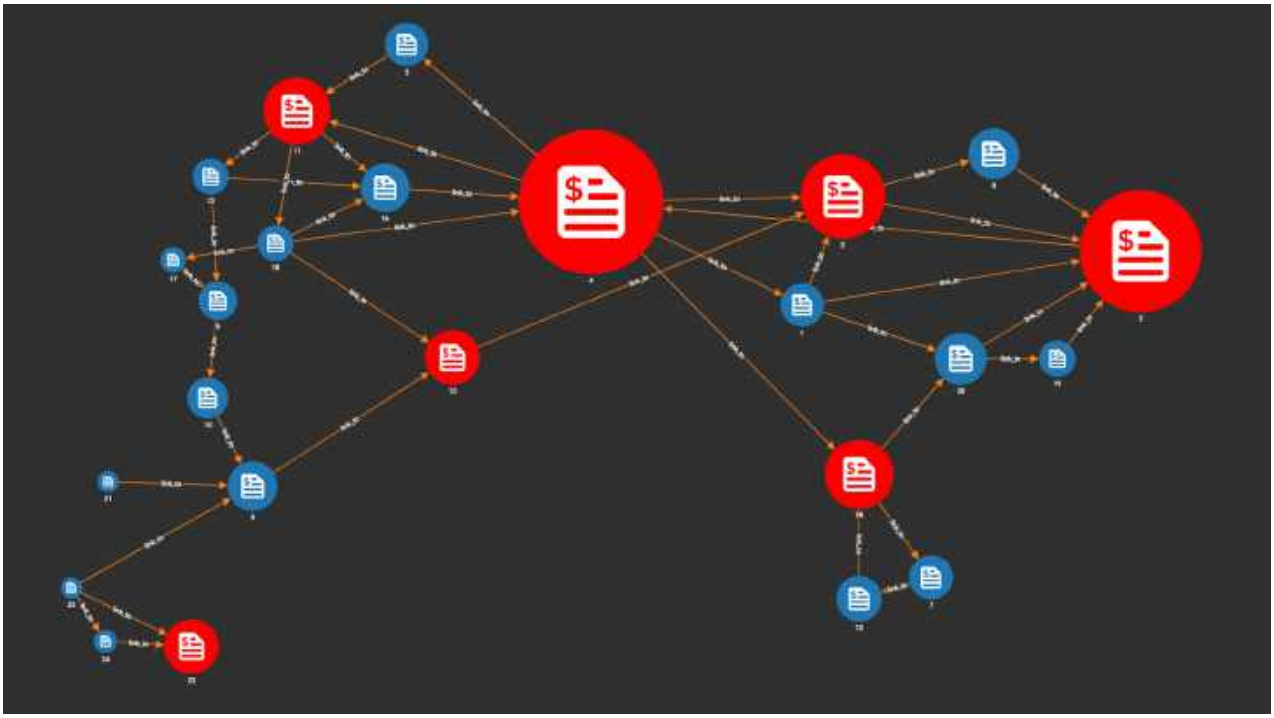
If you want to cancel the vertex radius or edge thickness configuration, click Edit button in Radius or Thickness section, in the pop up window choose None in the top level expression dropdown list:



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:

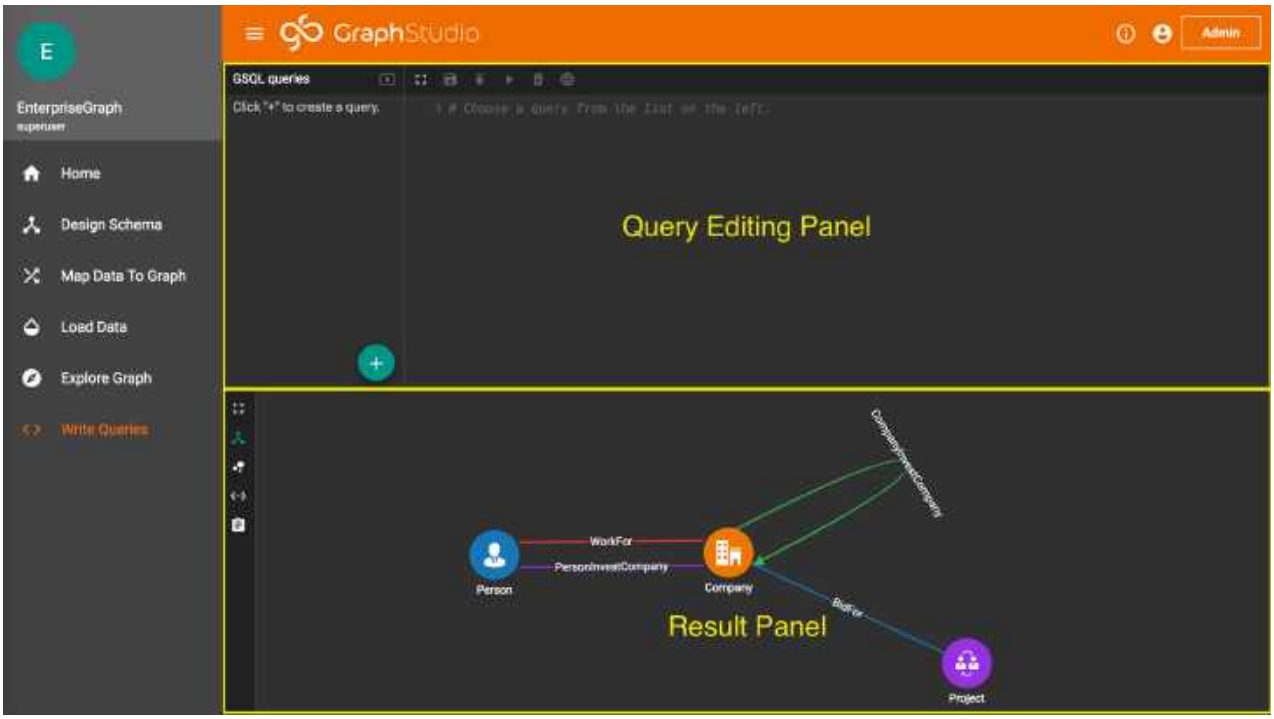

Settings
Visual configs of vertex and edge types
Vertex and edge types
Visual configs
Person
Company
Project
Page
WorkFor
PersonInvestCompany
CompanyInvestCompany
BidFor
link_to
Show attributes
type
id
Color
(@PageRankScore >= 1)
Click "+" to add color configs.
Radius
((@PageRankScore * 50) + 20)
The value of radius will be clamped between [20, 200].
Tooltip
Show tooltips when cursor hovers over vertices and edges.
CANCEL
APPLY



Write Queries

Write Queries

On the Write Queries page, you can design and run custom queries with TigerGraph's powerful graph query language – GSQL.



The Write Query page is horizontally divided into two parts:

1. Query Editing Panel
2. Result, Log and Visualization Panel

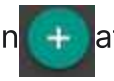
Query Editing Panel

The Query Editing panel is divided into two subpanels: the left subpanel is used to select a query to edit, and the right, larger subpanel displays the selected query for editing. Here you can edit, save, delete, 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:

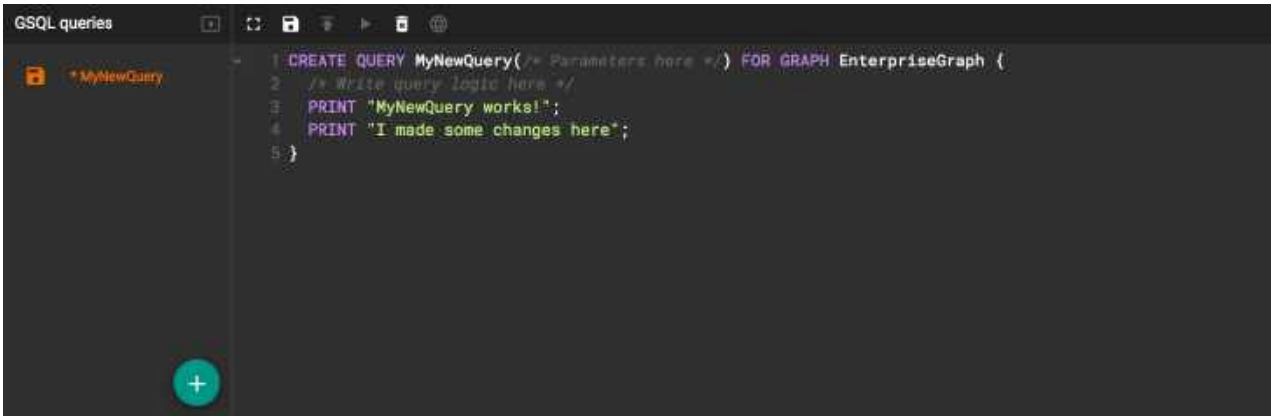
- **Expand/Collapse** : Expand or collapse the Query Editing panel to or from full page mode. The icon changes depending on whether the panel is currently expanded or collapsed.
- **Save** : Save the current query draft.
- **Install** : Install the query into the database.
- **Run** : Run the installed query.
- **Delete** : Delete the selected query.
- **Show query endpoint:** Show the RESTFul endpoint to execute the query. Only installed queries can see their RESTFul endpoints.

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 subpanel, 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.

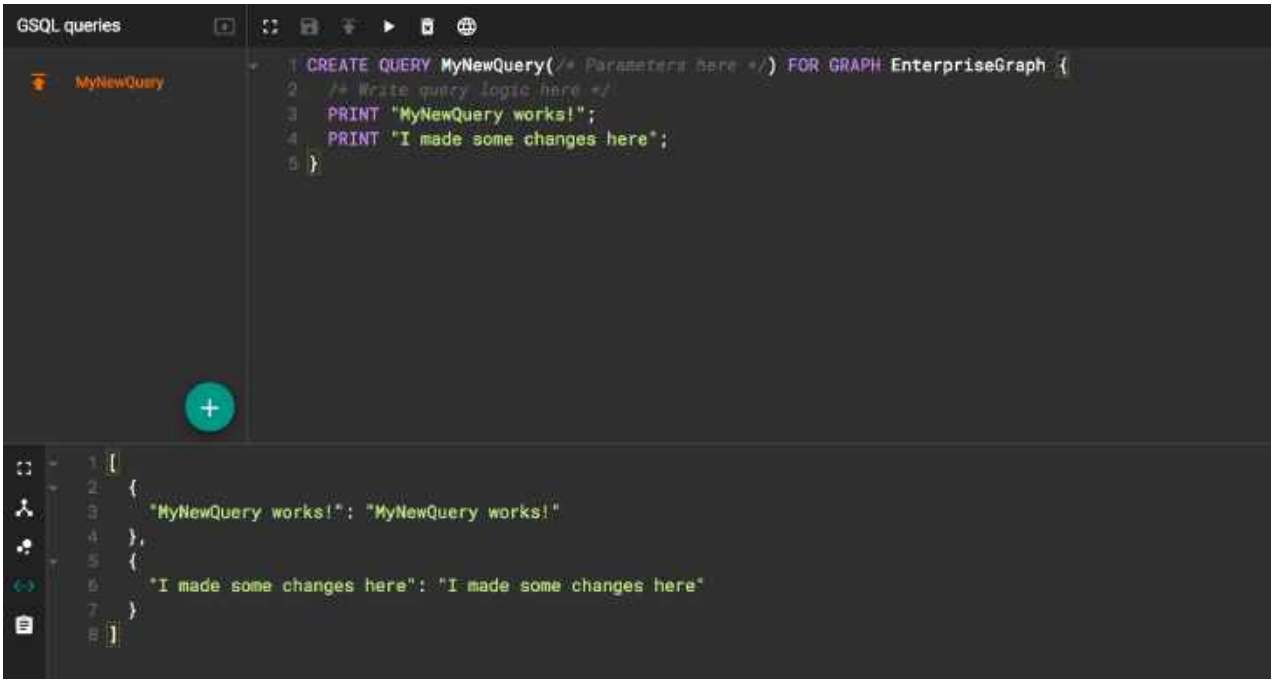
Install Query

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:

Run Query

A query has to be installed before you can run it.

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 Run query 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.

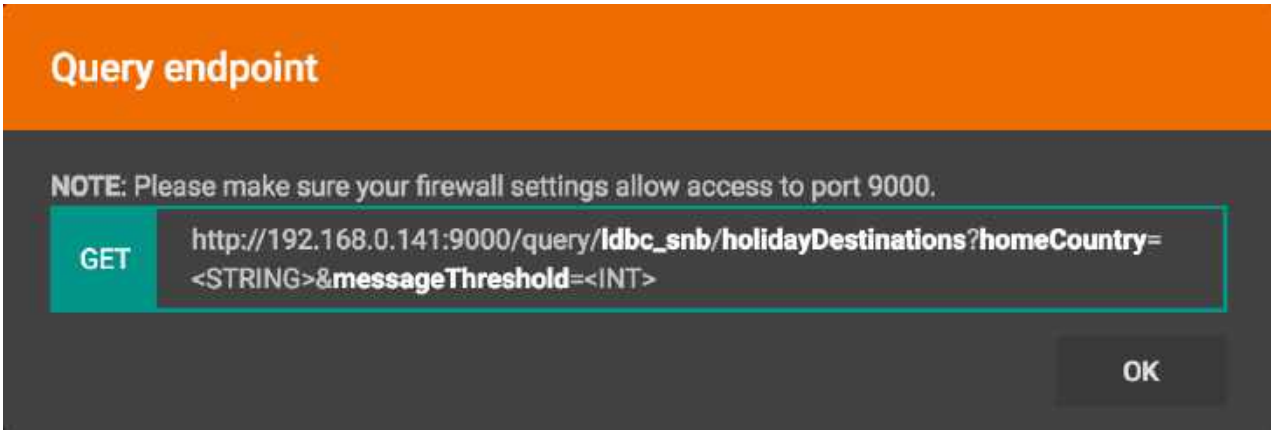
The query will be executed, and the results will be shown in the Result Panel.

Delete Query

Choose the query you want to delete and click on the "delete" button. The query will be deleted permanently.

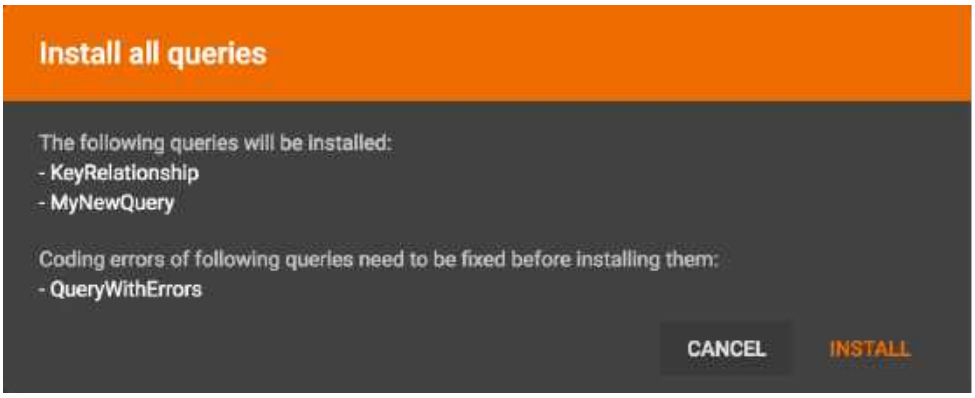
Show Query Endpoint

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.

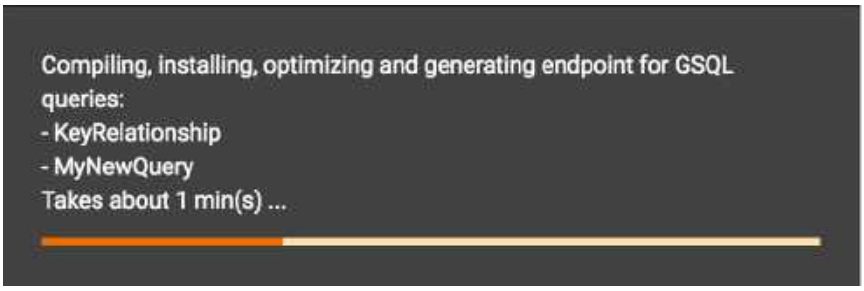


Install All Queries

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.
	Visual Result: Show the visual result of the last run query.

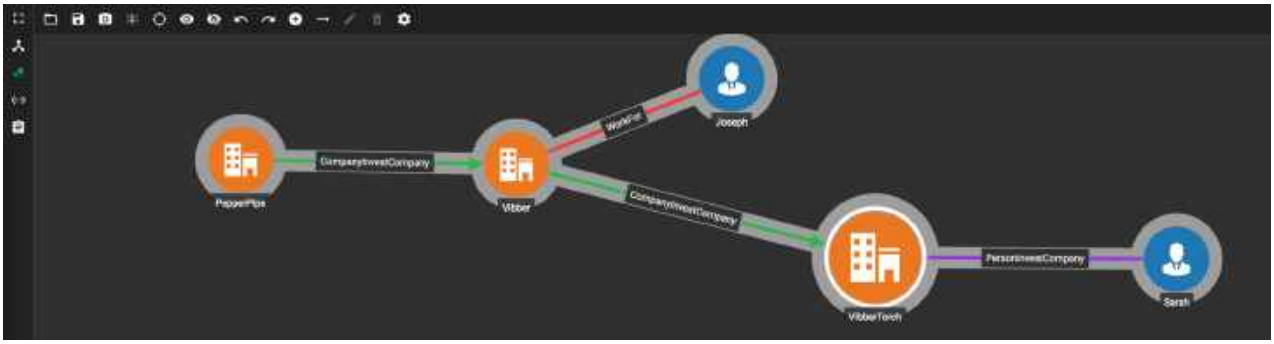
	JSON Result: Show the raw text result in JSON format of the last run query.
	Query Log: Show the log for the last run

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.

Visual 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.

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.

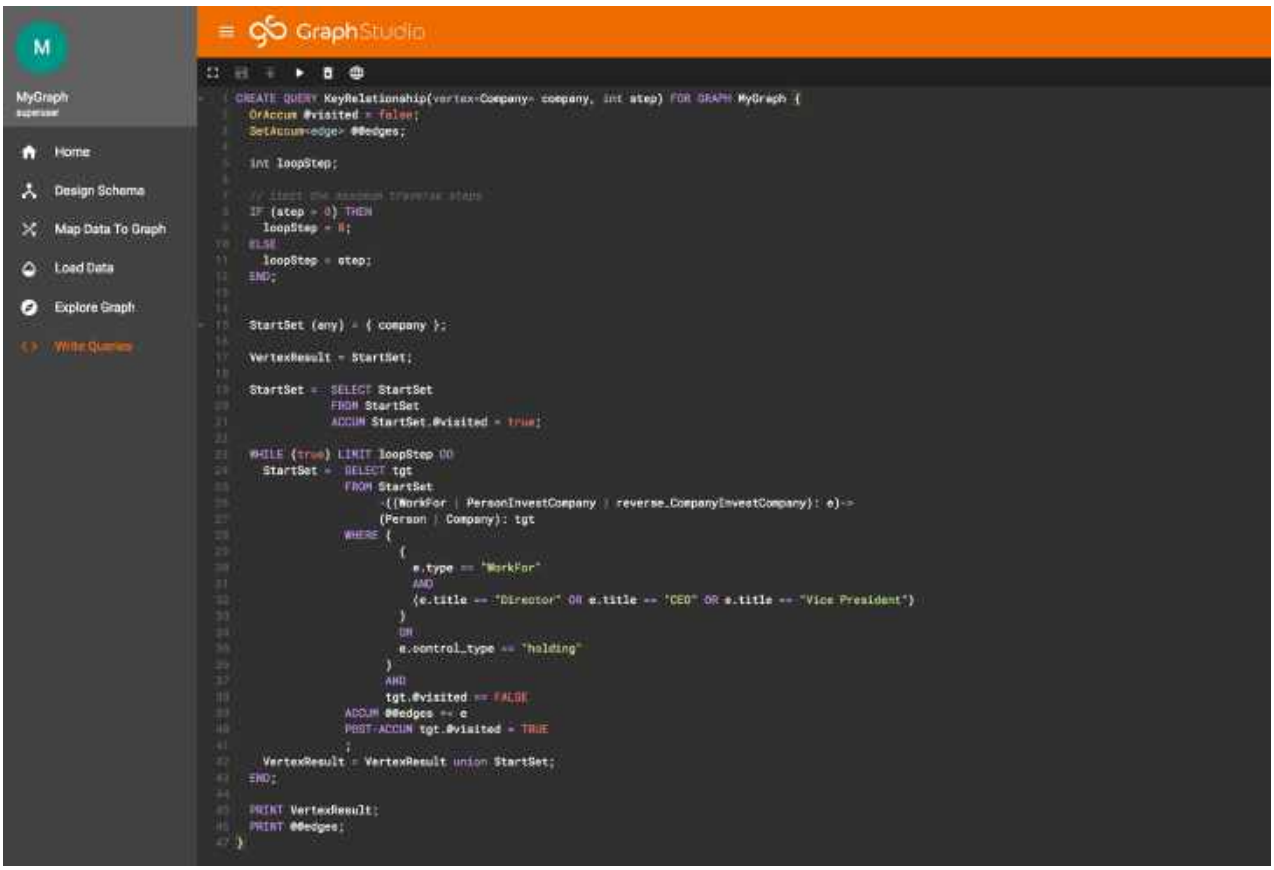
Query Log

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:

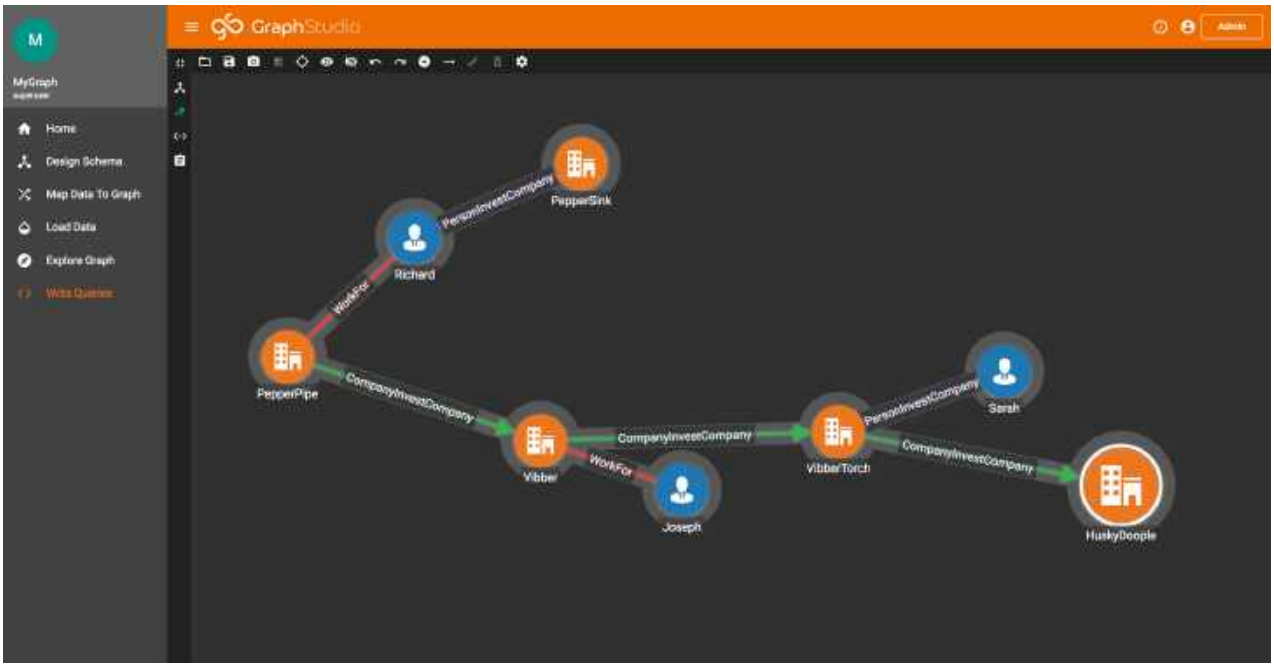
Expand Panels

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.

Export And Import Solution

Export And Import Solution

These two features can be found in the GraphStudio Home page. You can return to the Home page by clicking "Home" the Menu Bar on the left or clicking the "GraphStudio" logo at the top.

Export

Click on the "Export Current Solution" link to export the whole solution and download it as a tarball, including the schema, the loading jobs and the queries.

ATTENTION:

1. The graph data and data files will not be exported.
2. If a query has been modified since it was last installed, GraphStudio will export the modified draft instead of the version that have been installed in the TigerGraph engine.

Import

Click on the "Import an Existing Solution" will upload a previously exported tarball of a solution.

In order to optimize the time required for Import, the imported queries will not be

installed but saved as drafts. You need to install them manually.

⚠ For security reasons, user-created data sources won't be exported. If you import a solution with S3 data sources, you will need to manually create the data sources again (either though GraphStudio Map Data To Graph page or through GSQL shell). In GraphStudio, you can delete the previously created data sources and create new data sources to avoid duplicate data sources and ensure proper data loading.

Known Issues

GraphStudio is not perfect, like any other software. The following issues are known and will be fixed in the future.

After upgrading to v2.4, data mappings created in earlier versions of GraphStudio will disappear.

GraphStudio v2.4 changes internal loading job generation. Older version data mappings are deprecated. Please contact TigerGraph support if you need to migrate them from an earlier version.

Integers larger than 2^53 - 1 may lose precision.

Read more at https://stackoverflow.com/questions/307179/what-is-javascripts-highest-integer-value-that-a-number-can-go-to-without-losin ↗.

In the future, GraphStudio will use BigInt to solve this problem.

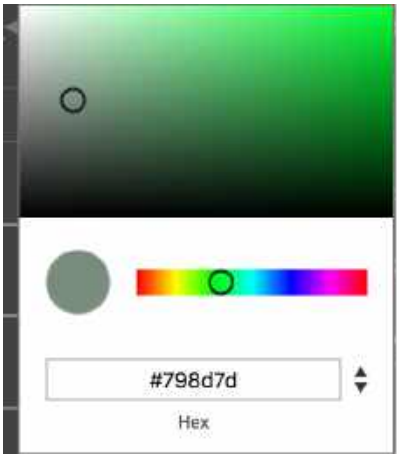
Change Color From Gray To Another Color

Since we upgraded the color picker library to new version, we find that if you set color to grey (left or bottom border of the color picker), then it will be locked to red as its pivot color. In this case when you try to change the pivot color, like this:

When you release mouse, it will be back to red:

You need first drag the indicator in the above panel to leave the border:

Then change the pivot color in the color stripe. Like this:



Graph Exploration Result Disappears

Sometimes when you double-click a vertex, the graph exploration result disappears. This is only a front-end rendering issue. The data is still there.
Workaround : click the change layout button  , and choose any layout. Everything will be rendered.

Edge Response Area Is Too Big

When there are edges very close to one another, their click response areas may overlap, making it hard to select the edge you want. This happens after zoom-in / zoom-out or connecting to another screen sometimes.
Workaround: click a blank place in the working panel then zoom-in and zoom-out. The response area will back to normal.

You Cannot Use Fixed Binary Type Attributes

Currently GraphStudio doesn't support fixed binary type attributes in schema. If you create your graph schema from GSQL with such attributes, GraphStudio will refuse to work. We will support this feature in future releases.

Loading Jobs Created From GSQL Won't Be Shown

GraphStudio can only recognize data mapping procedures created through GUI. If you create loading jobs from GSQL, they won't be shown in GraphStudio.

You Cannot Map Data To Map and UDT Type Attributes

Currently you cannot map data to map and UDT type attributes in GraphStudio. We will support this feature in future releases.

You Cannot Modify Schema In Multiple Graphs Scenario

Currently you cannot modify your schema in GraphStudio if there are more than one graph in TigerGraph system.

Workaround: you can still change the schema in GSQL and GraphStudio will work smoothly with your modified schema.

Report Bugs To Us

If you find any bugs, please report them to support@tigergraph.com. ↗We really appreciate it!

GraphStudio Patent and Third Party Notice

v2.2, January 2019

U.S. Pat. No. 9953106, 9977837, 10120956. Patents pending.

This TigerGraph software program uses some third-party software components that are licensed under their own terms.

This list of software components uses abbreviations to refer to common licenses, e.g., "MIT". A dictionary for these abbreviations is provided at the end of this document.

Third Party Component	License
Zoomcharts 	Copyright (c) 2018 Data Visualization Software Lab https://zoomcharts.com/en/legal/ ↗ Licensed under OEM license
angular/animations	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/cdk	Copyright (c) 2019 Google LLC https://github.com/angular/material2 ↗ Licensed under MIT
angular/common	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/compiler	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/core	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗

	Licensed under MIT
angular/flex-layout	Copyright (c) 2019 Google LLC https://github.com/angular/flex-layout ↗ Licensed under MIT
angular/forms	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/http	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/material	Copyright (c) 2019 Google LLC https://github.com/angular/material2 ↗ Licensed under MIT
angular/material-moment-adapter	Copyright (c) 2019 Google LLC https://github.com/angular/material2 ↗ Licensed under MIT
angular/platform-browser	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/platform-browser-dynamic	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/router	Copyright (c) 2014-2018 Google, Inc. https://github.com/angular/angular ↗ Licensed under MIT
angular/zone.js	Copyright (c) 2016-2018 Google, Inc. https://github.com/angular/zone.js ↗ Licensed under MIT
aws-sdk	Copyright (c) 2012-2017 Amazon.com, Inc. or its affiliates

	https://github.com/aws/aws-sdk-js ↗ Licensed under Apache2
cgjs/fs	Copyright (c) 2017 Andrea Giammarchi https://github.com/cgjs/fs ↗ Licensed under ISC
chalk	Copyright (c) 2017 Sindre Sorhus https://github.com/chalk/chalk ↗ Licensed under MIT
chart.js	Copyright (c) 2018 Chart.js Contributors https://github.com/chartjs/Chart.js ↗ Licensed under MIT
codemirror	Copyright (c) 2017 Marijn Haverbeke marijnh@gmail.com ↗ and others https://github.com/codemirror/CodeMirror ↗ Licensed under MIT
crypto	Copyright (c) 2014 Chris Veness https://github.com/chrisveness/crypto ↗ Licensed under MIT
CssColorParser.js	Copyright (c) 2012 Dean McNamee https://github.com/deanm/css-color-parser-js ↗ Licensed under MIT
d3.js	Copyright (c) 2010-2017 Mike Bostock https://github.com/d3/d3 ↗ Licensed under BSD3
echarts 3.4.0	Copyright (c) 2017 Baidu Inc. https://github.com/ecomfe/echarts-gl ↗ Licensed under BSD3
hammerjs	Copyright (c) 2011-2017 Jorik Tangelder https://github.com/hammerjs/hammer.js ↗ Licensed under MIT

jinder/path	Copyright (c) 2015 Joyent, Inc. and other Node contributors. https://github.com/jinder/path ↗ Licensed under MIT
js-yaml	Copyright (c) 2011-2015 Vitaly Puzrin https://github.com/nodeca/js-yaml ↗ Licensed under MIT
jsbn	Copyright (c) 2003-2005 Tom Wu http://www-cs-students.stanford.edu/~tjw/jsbn/ ↗ Licensed under MIT
jshttp/cookie	Copyright (c) 2012-2014 Roman Shtylman, 2015 Douglas Christopher Wilson https://github.com/jshttp/cookie ↗ Licensed under MIT
jsrsasign	Copyright (c) 2010-2018 Kenji Urushima https://github.com/kjur/jsrsasign ↗ Licensed under MIT
koa-body	Copyright (c) 2014 Charlike Mike Reagent and Daryl Lau https://github.com/dlau/koa-body ↗ Licensed under MIT
koa-bodyparser	Copyright (c) 2014 YiYu He heyiyu.deadhorse@gmail.com ↗ https://github.com/koajs/bodyparser ↗ Licensed under MIT
koa-multer	Copyright (c) 2014 Hage Yaapa, 2015 Fangdun Cai https://github.com/koa-modules/multer ↗ Licensed under MIT
koa-router	Copyright (c) 2015 Alex Mingoia https://github.com/alexmingoia/koa-router ↗

	Licensed under MIT
koa-send	Copyright (c) 2013-2019 koa-send contributors https://github.com/koajs/send ↗ Licensed under MIT
koa-static	Copyright (c) 2013-2019 koa-static contributors https://github.com/koajs/static ↗ Licensed under MIT
koajs	Copyright (c) 2018 Koa contributors https://github.com/koajs/koa ↗ Licensed under MIT
Leaflet.js	Copyright (c) 2010-2018 Vladimir Agafonkin, 2010-2011, CloudMade https://github.com/Leaflet/Leaflet/blob/master/LICENSE ↗ Licensed under BSD2
lodash	Copyright (c) 2017 JS Foundation and other contributors https://github.com/lodash/lodash ↗ Licensed under MIT
material-design-icons	Copyright (c) 2016 Material Design Authors https://github.com/google/material-design-icons ↗ Licensed under Apache2
moment	Copyright (c) 2016 JS Foundation and other contributors https://github.com/moment/moment ↗ Licensed under MIT

moment timezone	Copyright (c) 2016 JS Foundation and other contributors https://github.com/moment/moment-timezone/ ↗ Licensed under MIT
mysqljs	Copyright (c) 2012 Felix Geisendorfer https://github.com/mysqljs/mysql ↗ Licensed under MIT
ng2-nouislider	Copyright (c) Tomasz Bak https://github.com/tb/ng2-nouislider ↗ Licensed under MIT
ngx-image-cropper	Copyright (c) 2018 Martijn Willekens https://github.com/Mawi137/ngx-image-cropper ↗ Licensed under MIT
ngx-color-picker	Copyright (c) 2017 ZEF Oy https://github.com/zefoy/ngx-color-picker ↗ Licensed under MIT
node-ip	Copyright (c) 2012 Fedor Indutny https://github.com/indutny/node-ip ↗ Licensed under MIT
node-jsonwebtoken	Copyright (c) 2015 Auth0, Inc. https://github.com/auth0/node-jsonwebtoken ↗ Licensed under MIT
nouislider	Copyright (c) 2018 Léon Gersen https://github.com/leongersen/noUiSlider ↗ Licensed under MIT
randomcolor	Copyright (c) 2015 David Merfield https://github.com/davidmerfield/randomColor ↗ Licensed under CC0

reactivex/rxjs	Copyright (c) 2015-2018 Google, Inc., Netflix, Inc., Microsoft Corp. and contributors https://github.com/reactivex/rxjs ↗ Licensed under Apache2
request	Copyright (c) 2010 Mikeal Rogers https://github.com/request/request ↗ Licensed under Apache2
resumablejs	Copyright (c) 2011 Steffen Tiedemann Christensen https://github.com/23/resumable.js ↗ Licensed under MIT
roboto-fontface	Copyright (c) 2013 Christian Hoffmeister https://github.com/choffmeister/roboto-fontface-bower ↗ Licensed under Apache2
roboto-mono-webfont	Copyright (c) 2016 Christian Robertson https://github.com/Dilatorily/roboto-mono ↗ Licensed under MIT AND Apache2
sqlite3	Copyright (c) 2013 MapBox https://github.com/mapbox/node-sqlite3 ↗ Licensed under BSD3
websockets/ws	Copyright (c) 2011 Einar Otto Stangvik https://github.com/websockets/ws ↗ Licensed under MIT
winston-daily-rotate-file	Copyright (c) 2015 Charlie Robbins https://github.com/winstonjs/winston-daily-rotate-file ↗ Licensed under MIT
	Copyright (c) 2010 Charlie Robbins

License Abbreviations

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. The copy of each license is also available from TigerGraph and is included in the doc/legal folder of the product package.

License Abbreviation	License Detail
AGPL3	GNU Affero General Public License version 3 https://www.gnu.org/licenses/agpl-3.0.en.html ↗
Apache2	Apache License version 2.0 https://www.apache.org/licenses/LICENSE-2.0 ↗
BOOST	Boost Software License 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 ↗
CC0	Creative Commons CC0 1.0 Universal https://creativecommons.org/publicdomain/zero/1.0/ ↗
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 ↗
GPL3	GNU General Public License version 3.0 https://www.gnu.org/licenses/gpl-3.0.en.html ↗
ISC	Internet Systems Consortium https://www.isc.org/downloads/software-support-policy/isc-license/ ↗
JSON	JSON License http://www.json.org/license.html ↗
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 ↗
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 https://www.zlib.net/zlib_license.html ↗

Developer's Guides

GSQL Demo Examples

Using the TigerGraph TM 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 master 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 >

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.

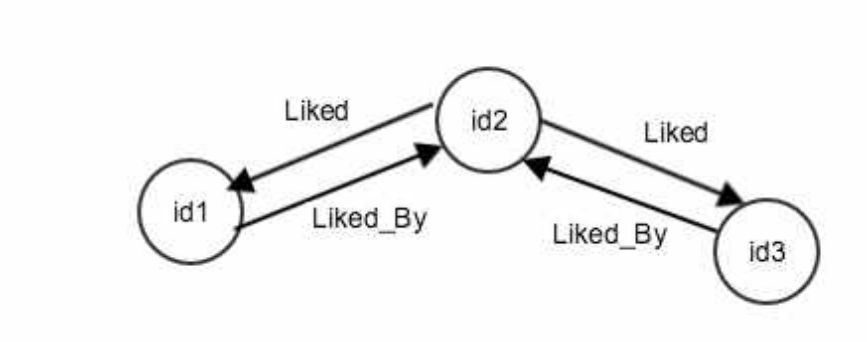


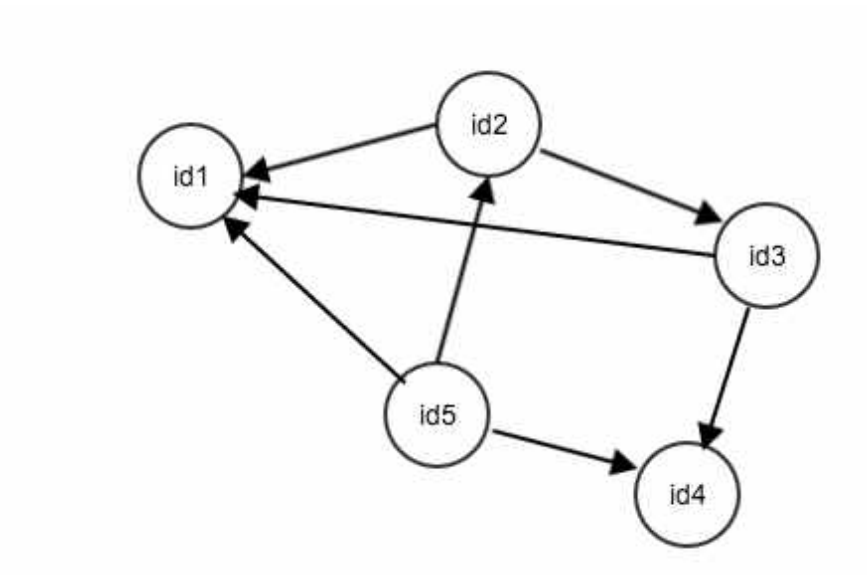
Figure 1 - Example graph for the collaborative filtering model

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.

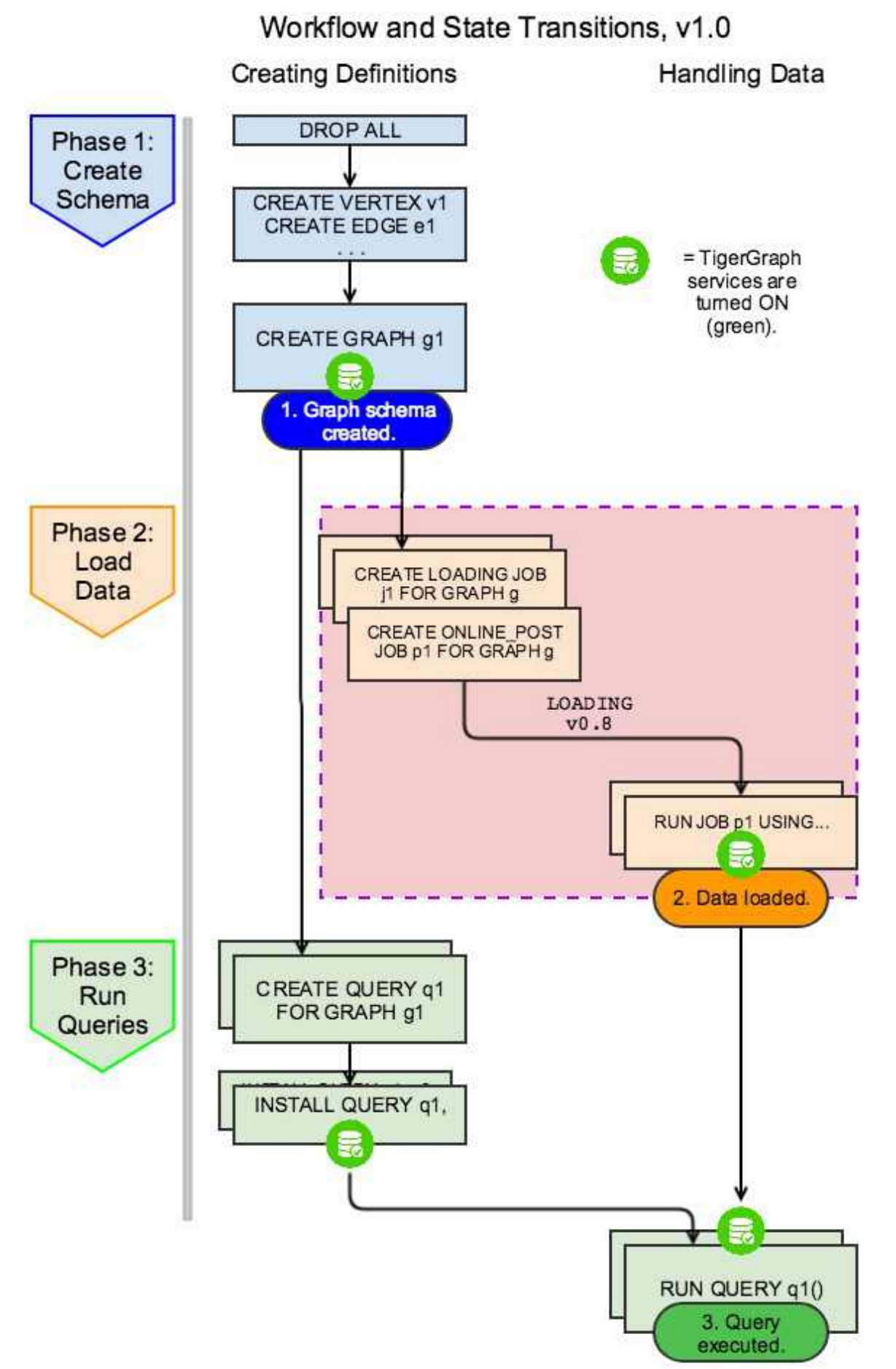
```
> gsql -g gsql_demo cf_query.gsql
> gsql -g gsql_demo 'INSTALL QUERY topCoLiked'
> gsql -g gsql_demo 'RUN QUERY topCoLiked("id1", 20)'
```

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 = "Liked
```

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 = "Liked
#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.

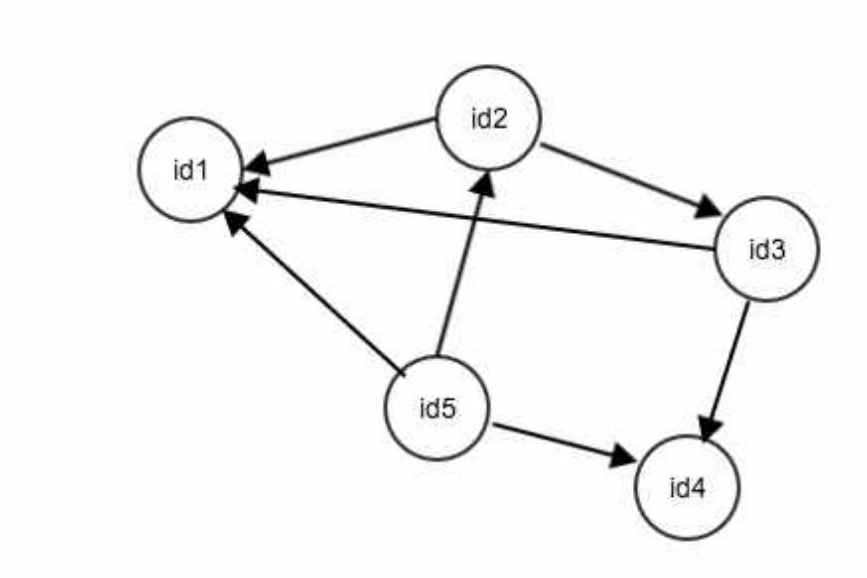


Figure 2 - Graph for Collaborative Filtering Calculation

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`	⚠ **DEPRECATED**. Display the estimated count of User vertices. Use count(*) or approx_count(*) instead.
`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 gsc
{
  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=ic
```

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 dampingFac
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 iterati
  SumAccum<float> @received_score = 0; # sum of scores each vertex receive
  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.@rece
                    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 output
  PRINT AllV[AllV.page_id, AllV.@score];  # print the results, JSON output
} # 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

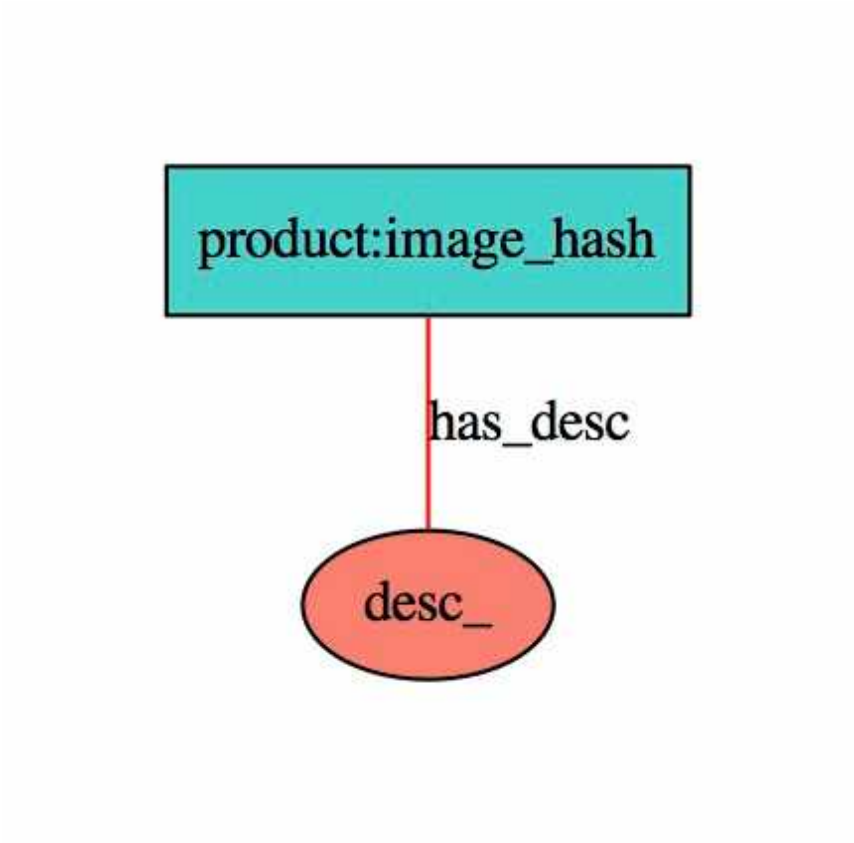
```
CREATE VERTEX Product (PRIMARY_ID pid string, image_hash uint)
CREATE VERTEX DescWord (PRIMARY_ID id string)
CREATE UNDIRECTED EDGE Has_desc (FROM Product, TO DescWord)
```

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.

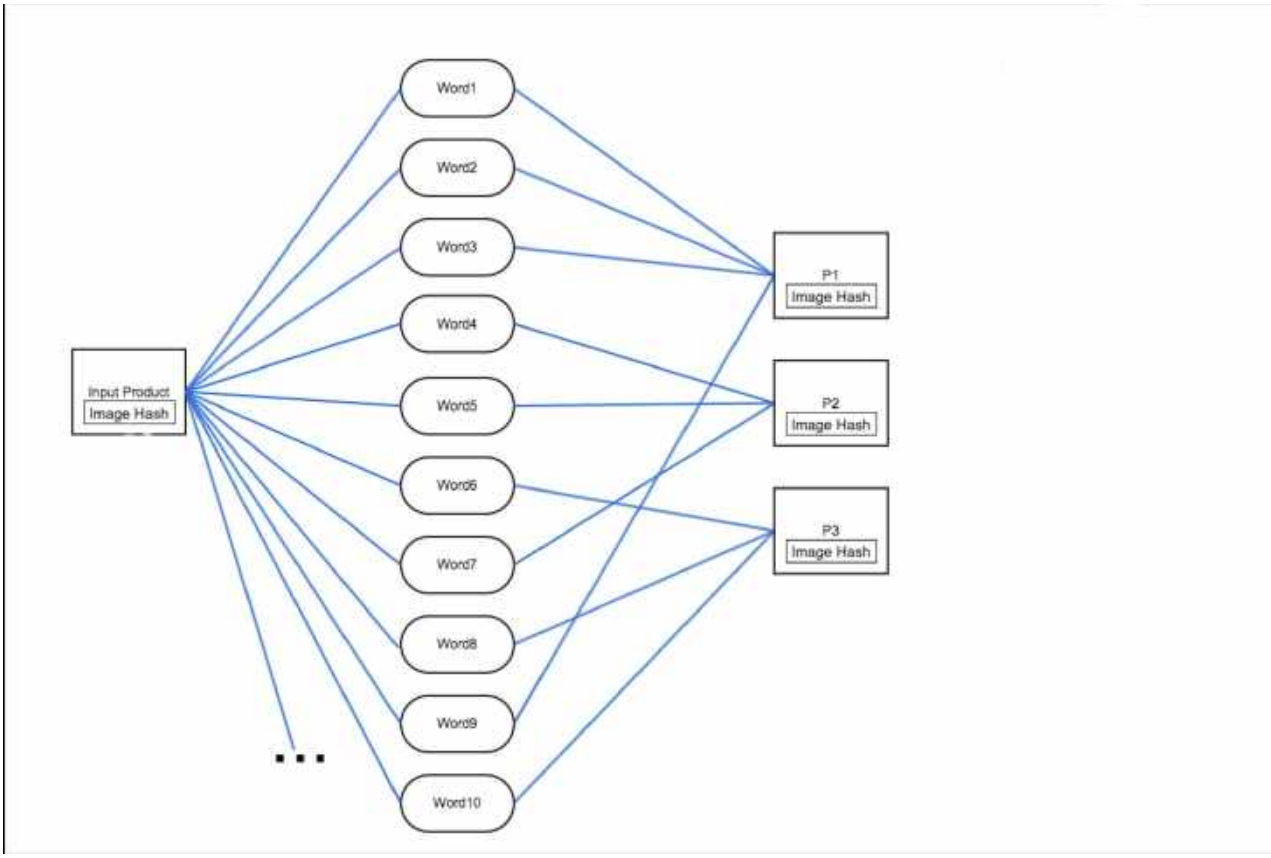
1. The HEADER="true" option tells the loader that the data file's first line contains column headings instead of data. It will read the column headings and permit these heading names to be used instead of index numbers $1, $2, etc.
2. DEFINE HEADER and USER_DEFINED_HEADER allow the loading job to define 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, i
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 t
    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 in
     *  of this query. From each word in ProductWords, activate all other
     *  which contain the word in their description, and accumulate a coun
     *  each activated product to record how many words it has in common w
     *  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 JSO
}
```

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.

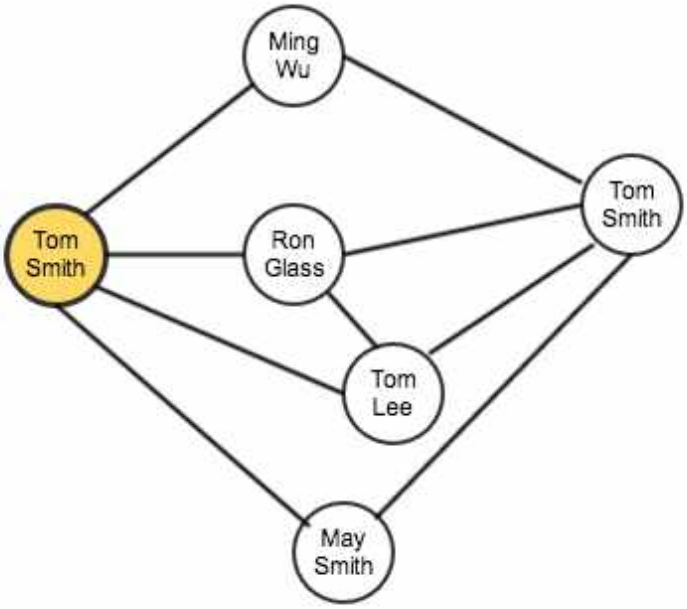
Input Data: A social network, where each person has a first and last name and may also display a picture of themselves.

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=6
```

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 s
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, string
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 == lastNam
                    THEN  u2.@match = 3
                WHEN u2.firstname != firstName AND u2.lastname == lastName
                    THEN  u2.@match = 2
```

```
                WHEN u2.firstname == firstName AND u2.lastname != lastName
                    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_time
CREATE VERTEX AttributeTag (PRIMARY_ID tag_id string, content string, att_
CREATE UNDIRECTED EDGE Video_AttributeTag (FROM Video, TO AttributeTag, we
CREATE UNDIRECTED EDGE User_Video (FROM VidUser, TO Video, rating float DE
```

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_v
RUN LOADING JOB load_videoE USING f2="../video/data/video_recommendation_e
```

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": "1",
      "attributes": {
        "date_time": 0,
        "@score": 5.2,
        "@viewedBySeed": false,
        "content": "v1"
      },
      "v_type": "Video"
    },
    {
      "v_id": "4",
      "attributes": {
        "date_time": 0,
        "@score": 17.2,
        "@viewedBySeed": false,
        "content": "v4"
      },
      "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)
```

$$score(A,B) = \sum \frac{connectionStrength(A \rightarrow x) \cdot connectionStrength(x \rightarrow B)}{degree(A) \cdot degree(B)}$$

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 g
{
  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 s
             @@startPdegree += src.outdegree(); #  save seed outdegre

  # 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.outdegr
       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": "3",
      "attributes": {"L2.@score": 0.075},
      "v_type": "Person"
    },
    {
      "v_id": "6",
      "attributes": {"L2.@score": 0.1},
      "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 boo
CREATE DIRECTED EDGE SocialConn (FROM SocialUser, TO SocialUser) WITH REVE
```

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 re
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<Socia
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_contacts
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 o
          #  B.neighbors('edgeType1') returns only the neighbors connected
        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("Social
    PRINT Result; # the result includes B's static attributes and B.@fromN
}
```

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"
    }
  ]}]
}
```

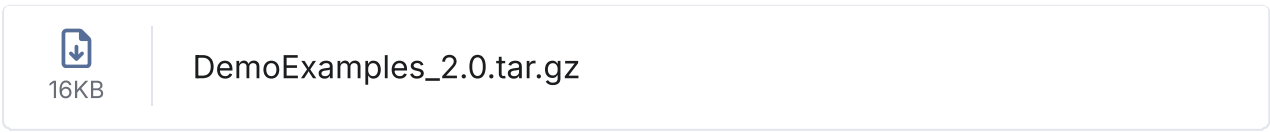
Classic Graph Algorithms

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:



DemoExamples_2.0.tar.gz

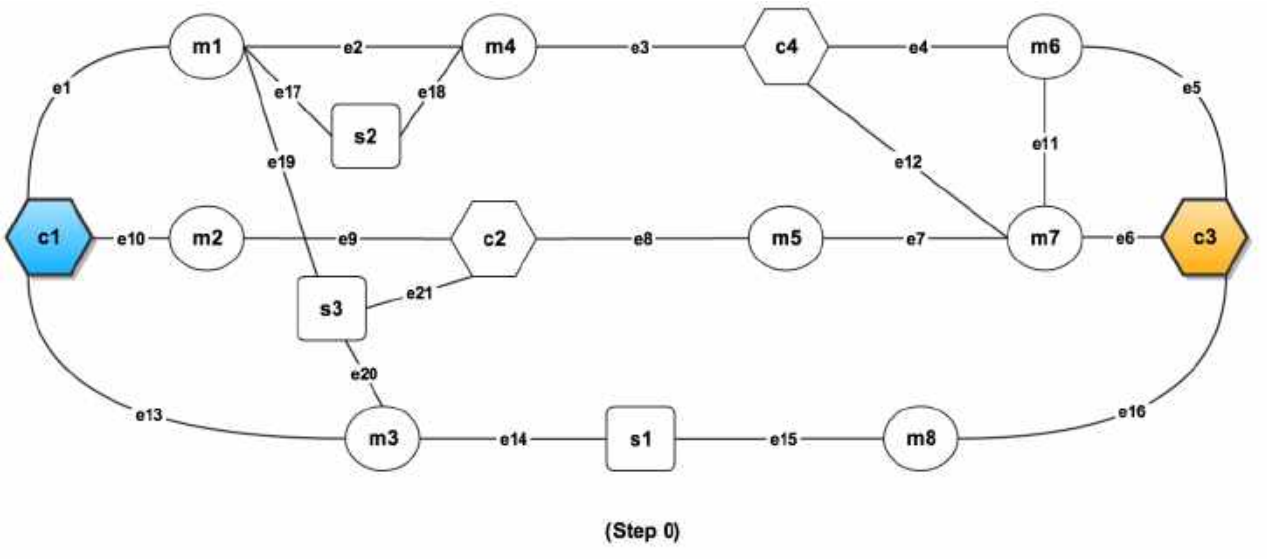


Figure 1: Example Graph used to demonstrate the Shortest Path algorithms.

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_nar
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, posi
CREATE UNDIRECTED EDGE person_person (FROM persons, TO persons, connect_ti
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_nam
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, posi
CREATE UNDIRECTED EDGE person_person (FROM persons, TO persons, connect_ti
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, respectiv
# 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 FOREACH

CREATE QUERY shortest_path_1D (VERTEX<company> S, VERTEX<company> T, INT m

  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 require
            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.

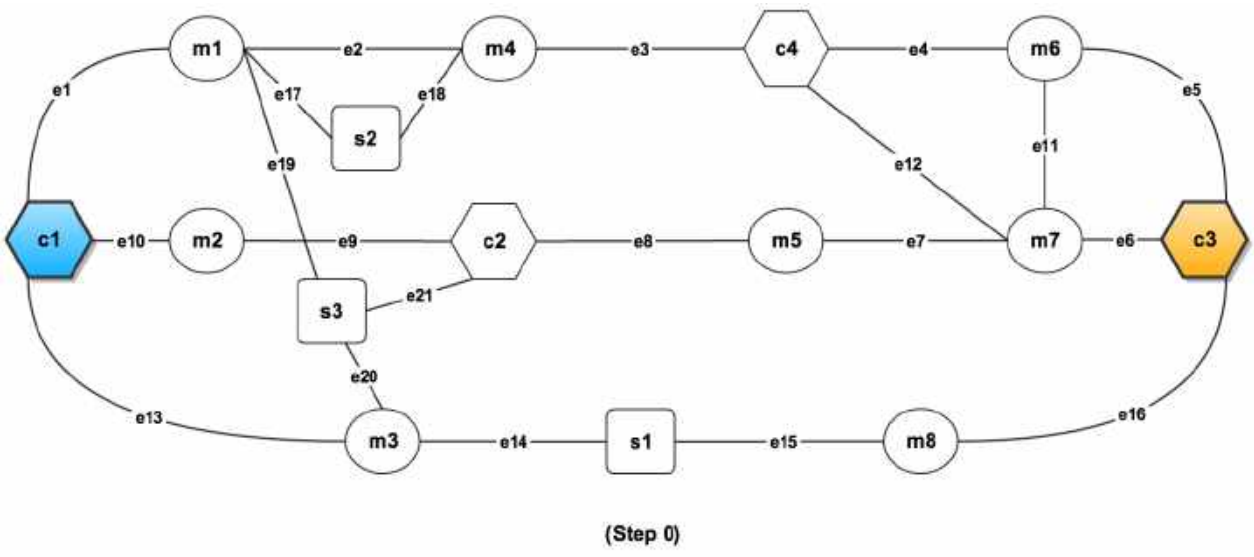


Figure 2: The starting state for our graph. From here, we go on to the first step of the algorithm. We start at c1, and go along each of its edges.

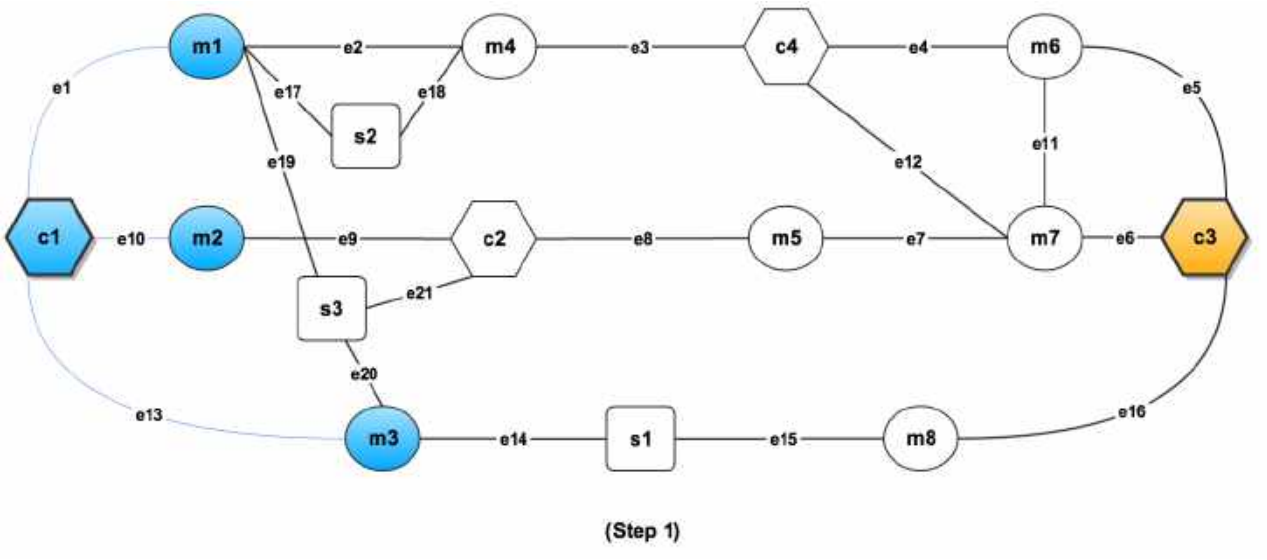


Figure 3: This is the graph after one step. We have traveled from c1 to all of its neighbors, labeling them as visited. For each one that we visit, we update its @pathResult accumulator value in order to keep track of our path as we traverse the graph.

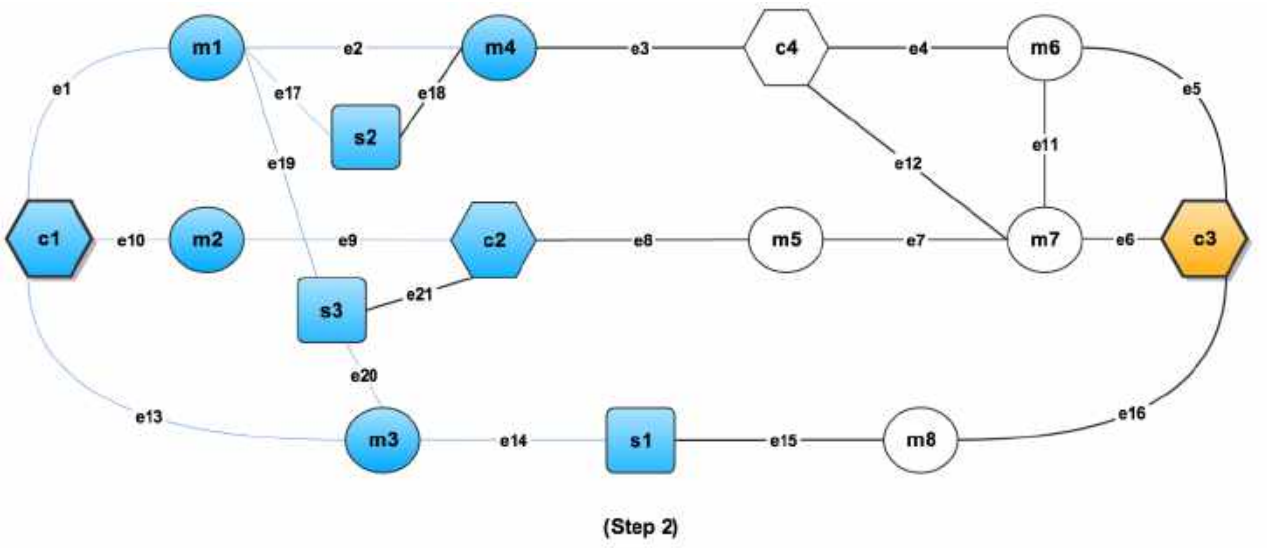


Figure 4: This graph shows where we have traveled after two steps. We traveled to our new vertices s1, s2, s3, c2 and m4 by traveling one edge away from the nodes that we had visited in step 1. Note that the blue edges also tell us how we can get from c1 to a vertex. For example, we notice that e21 is 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.

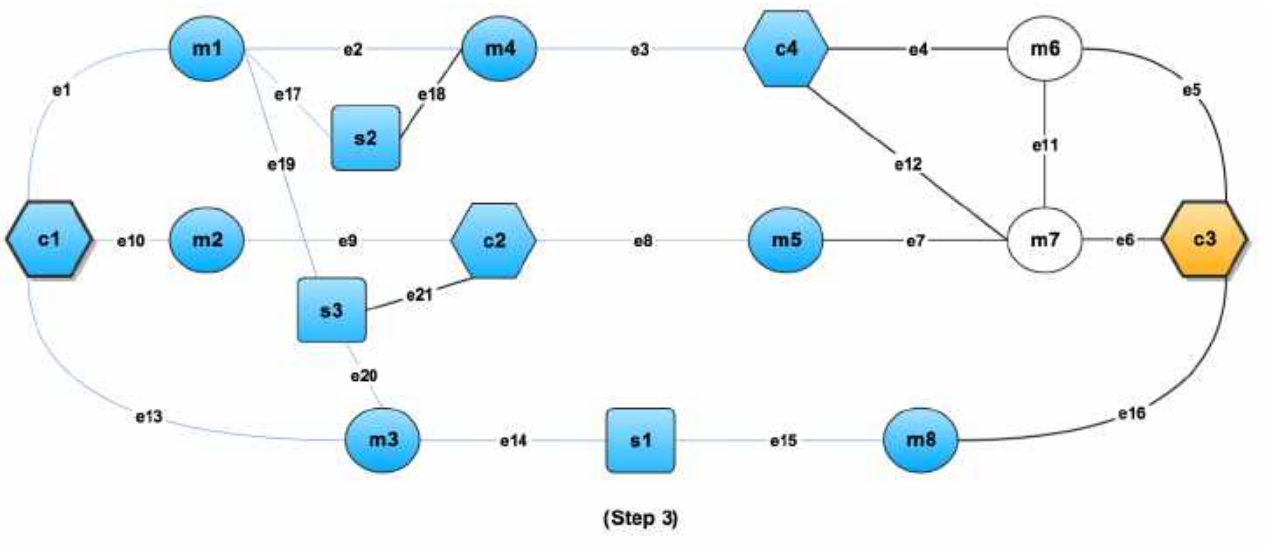


Figure 5: At the third step of our algorithm, we have reached the nodes m8, m5 and c4. We got here by moving one edge away from the vertices that we reached in step 2.

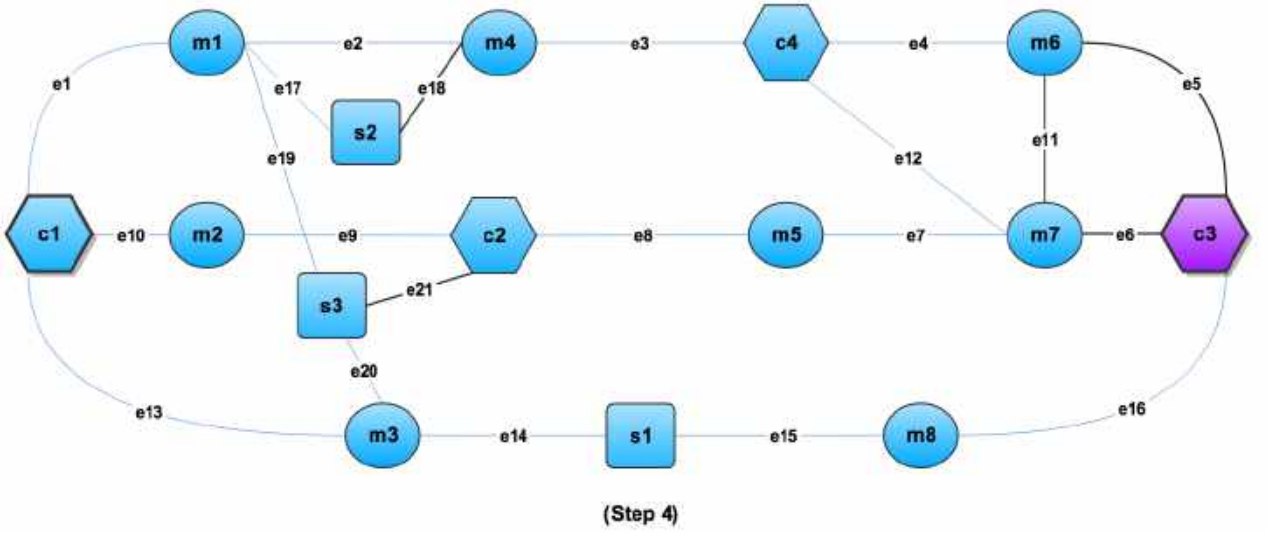


Figure 6: Finally, we have reached the end of our algorithm. Note that when we travel one edge away from m8, we arrive at our target node of c3. Working

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. It is important to note that if w

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": {
    "edition": "developer",
    "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": {
    "edition": "developer",
    "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 consist 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.

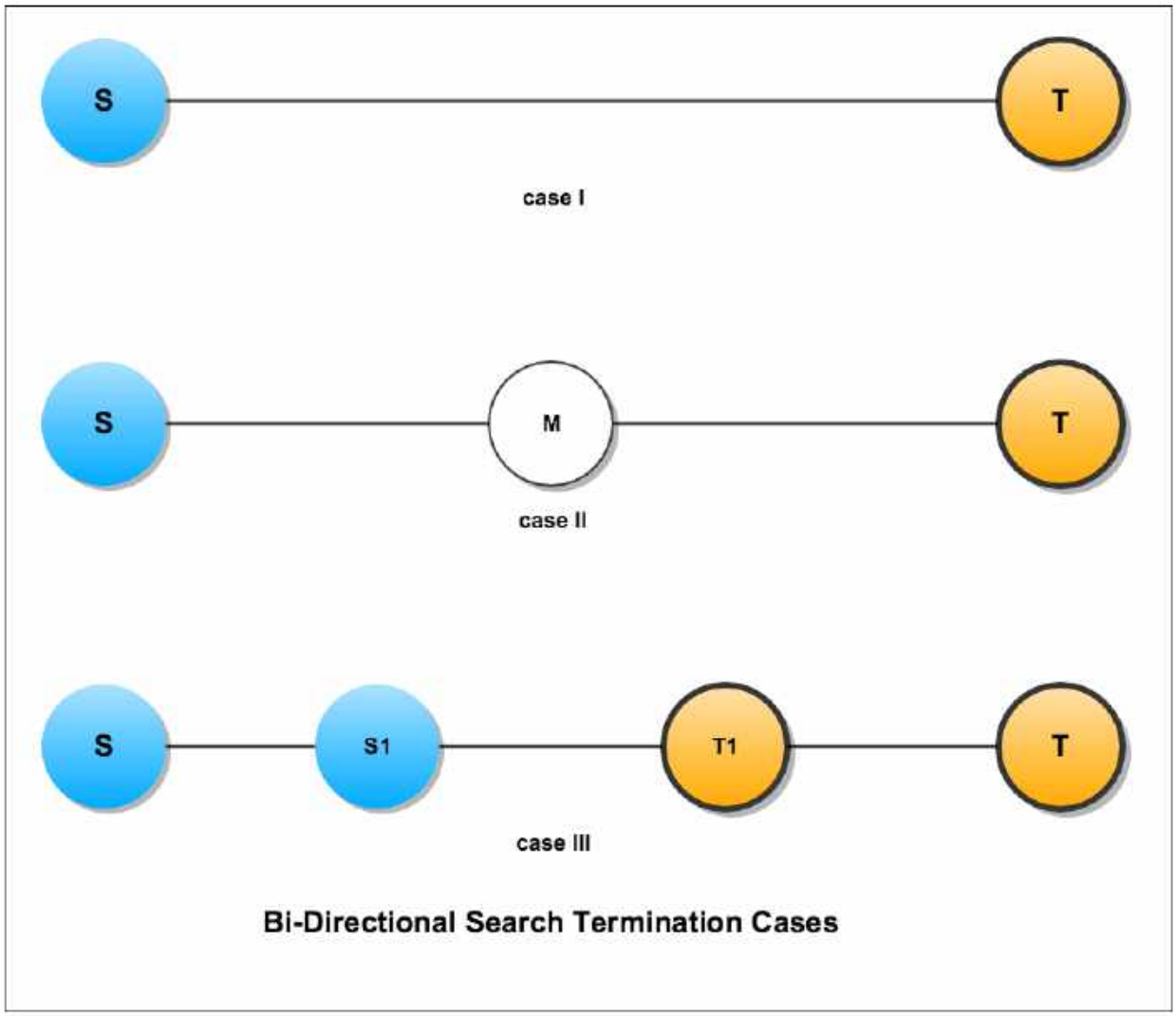


Figure 7 : Three cases for terminating a bi-directional path search.

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 fron
    // then V is part of vertex group T.  The vertices who have the longes
    // 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.v
                    // If the frontiers of S and T are neighbors (Case III
                    if (v.visitedByS) {
                        final_path = v.pathFromS + v.ID + b.ID + b.pathFro
                    }
                    if (v.visitedByT) {
                        final_path = v.pathFromT + v.ID + b.ID + b.pathFro
                    }
                    stop = true;
                    break;
                } else if ((v.visitedByS && not b.visitedByS) || (v.visite
                    // If b (the neighbor of v) is not yet part of v's ver
                    // then add b to the vertex group, and update b's path
                    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 (Case
            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 FOREACH
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 'm
  SumAccum<int>   @sLength = 0;  // vertex runtime attribute: # steps from
  SumAccum<int>   @tLength = 0;  // vertex runtime attribute: # steps from
  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 ve
            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 forw
        FROM StartSet:u-(:e)->:v
        WHERE ((u.@seenS==true AND v.@seenS!=true) OR // from S frontier t
               (u.@seenT==true AND v.@seenT!=true))   // from T frontier t
        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 s
                    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 + pat
                                END
                            END,
```

```
                            v.@meet = true,
                            v.@resultLength =  u.@sLength + v.@tLength + 1
                    END
            // Else, since u->v does not complete a path, move the frontie
            // If u is in S's frontier, then extend S's frontier to v. Ass
            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. Ass
            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 four
            CASE WHEN (v.@seenS == true AND v.@seenT == true AND @@stop ==
                THEN @@stop += true,
                    // Insert v.id between the source paths and the target
                    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

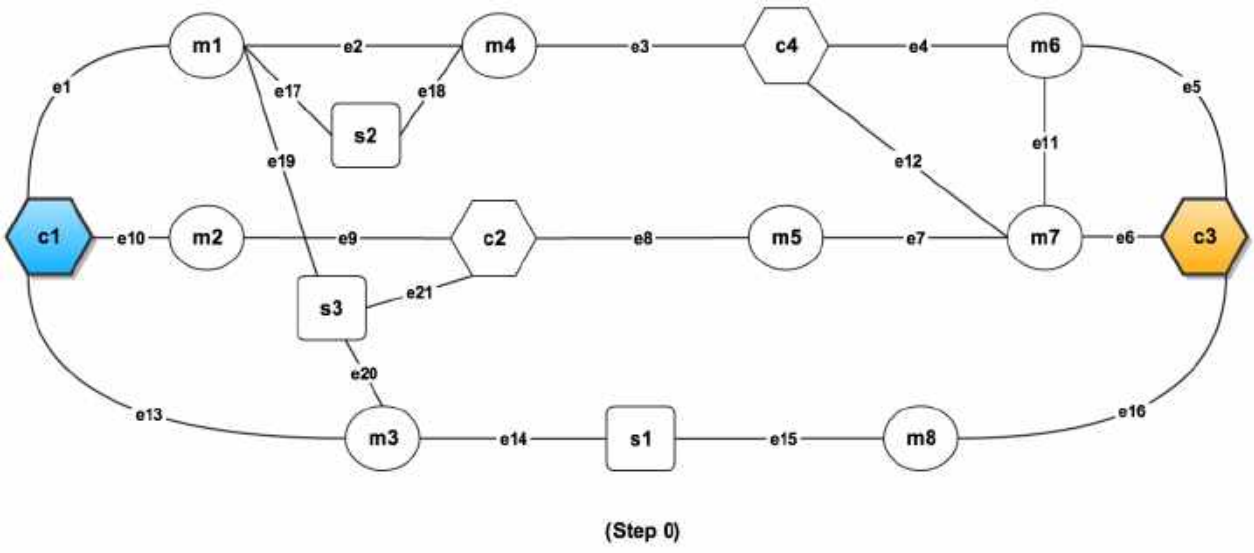


Figure 8: Initialization - prepare to start the search process. The two given vertices (c1 and c3) are activated and colored as Blue and Orange respectively. The rest of the graph remains untouched.

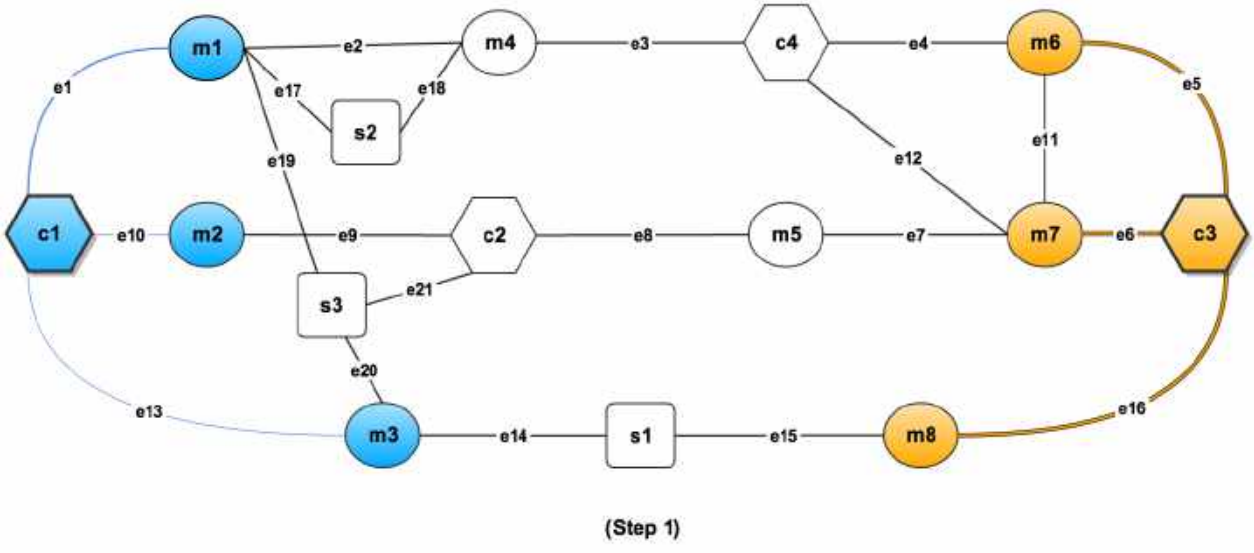


Figure 9: The graph after the first step. The search process starts simultaneously from c1 and c3. If a vertex is seen by the agent starting from c1 (c3), we will say it is seen by c1 (c3).

- 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 m6, m7 and m8 are discovered and become the frontier of c3's vertex group.

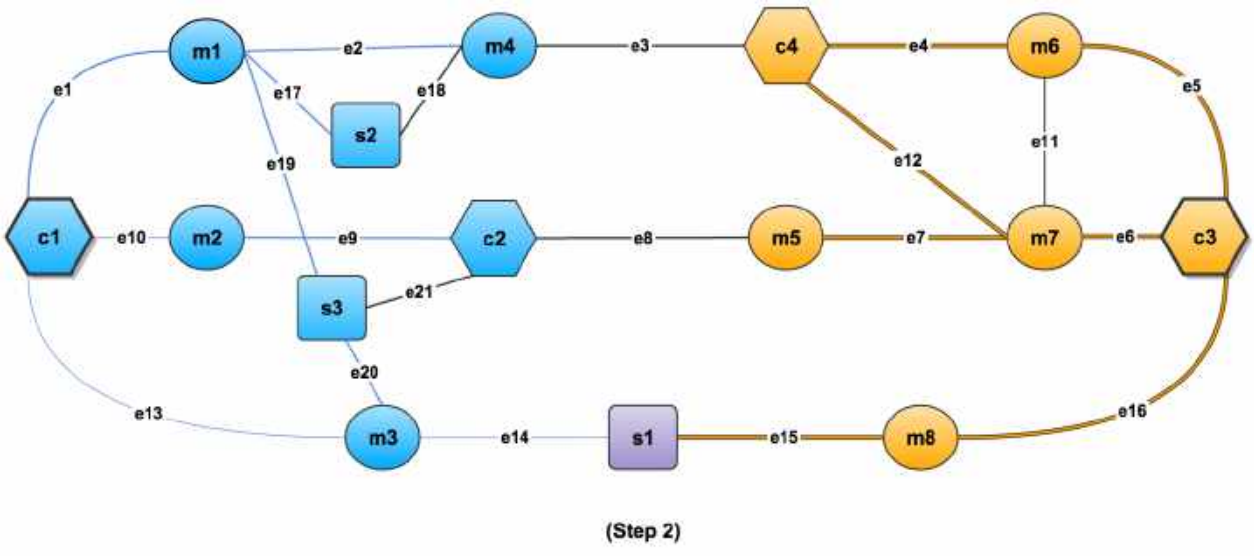


Figure 10: As the two groups have not been met yet, the search process continues.

- 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": {
    "edition": "developer",
    "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.

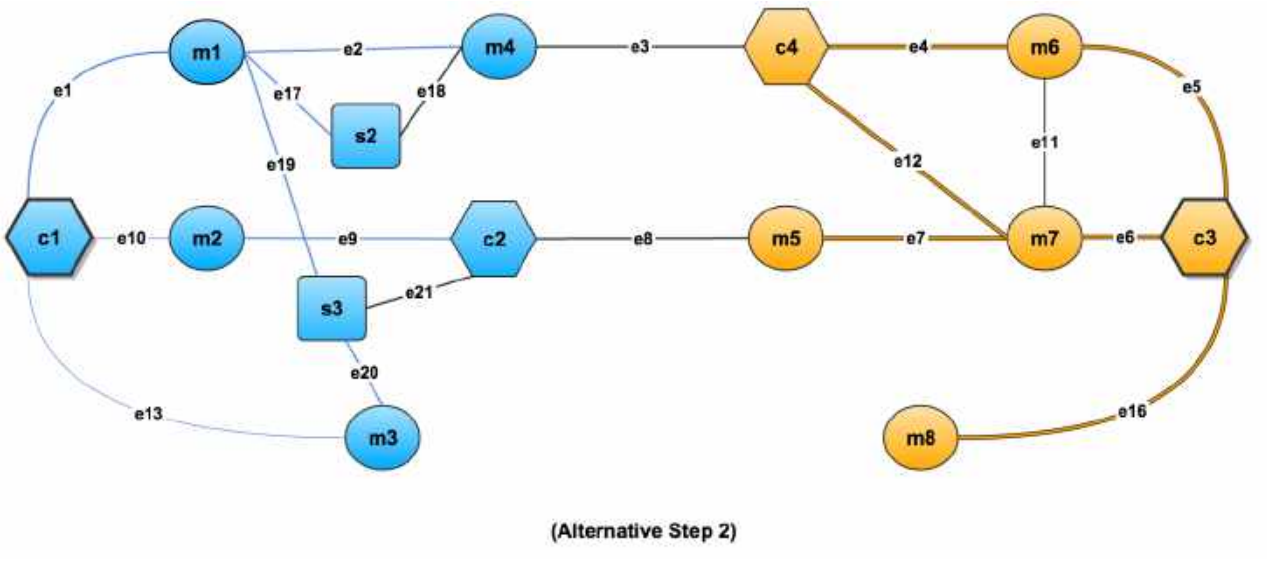


Figure 11: 2nd Iteration in a modified graph in which s1 does not exist. We got here by traveling one edge away form the vertices that were visited in the previous step. However, as we do not yet have a crossing, we must complete one more iteration.

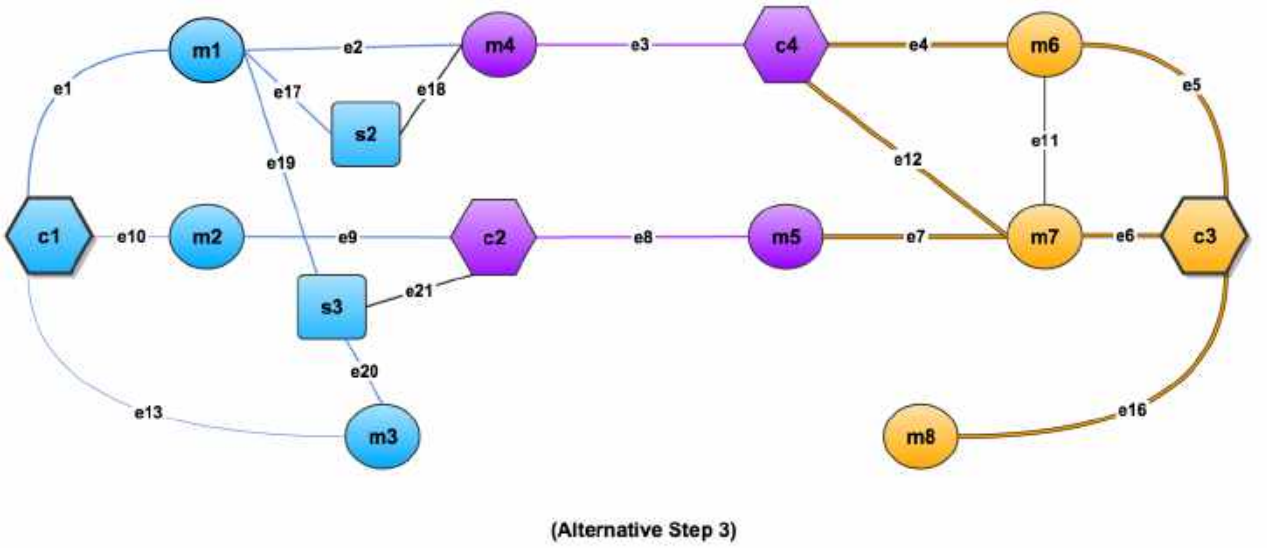


Figure 12: Here, the paths from c1 have finally found a vertex that was previously found by the paths from c3 (and vice versa). That is, the blue paths traveled from c2 to m5 and from m4 to c4. In Figure 11, m5 and c4 were both orange. In Figure 12, we change a vertex's color to purple when one frontier 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": {
    "edition": "developer",
    "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"
    }
  ]}]
}
```

GSQL Language Reference

Part 1 - Data Definition & Loading

Introduction

The GSQL™ software program is the TigerGraph comprehensive environment for designing graph schemas, loading and managing data to build a graph, and querying the graph to perform data analysis. In short, TigerGraph users do most of their work via the GSQL program. This document presents the syntax and features of the GSQL language.

This document is a reference manual, not a tutorial. The user should read ***GSQL Demo Examples*** prior to using this document. There are also User Guides or Tutorials for particular aspects of the GSQL environment. This document is best used when the reader already has some basic familiarity with running GSQL and then wants a more detailed understanding of a particular topic.

This document is Part 1 of the GSQL Language Reference, which describes system basics, defining a graph schema, and loading data. Part 2 describes querying.

A handy GSQL Reference Card lists the syntax for the most commonly used GSQL commands for graph definition and data loading . Look for the reference card on our User Document home page.

GSQL Workflow

The GSQL workflow has four major steps:

1. Define a graph schema or model.
2. Load data into the TigerGraph system.
3. Create and install queries.
4. Run queries.

After initial data and queries have been installed, the user can run queries or go back to load more data and create additional queries. This document provides specifications and details for steps 1 and 2. The Appendix contains flowcharts which provide a visual understanding of the required and allowed sequence of commands to proceed through the workflow.

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 */

Documentation Notation

In the documentation, code examples are either **template code** (formally describing the syntax of part of the language) or **actual code examples** . Actual code examples show code that can be run exactly as shown, e.g., copy-and-paste. Template code, on the other hand, cannot be run exactly as shown because it uses placeholder names and additional symbols to explain the syntax. It should be clear from context whether an example is template code or actual code.

This guide uses conventional notation for software documentation. In particular, note the following:

- **Shell prompts**
  Most of the examples in this document take place within the GSQL shell. When clarity is needed, the GSQL shell prompt is represented by a greater-than arrow: `>`
  When a command is to be issued from the operating system, outside of the GSQL shell, the prompt is the following: `os$`
- **Keywords**
  In the GSQL language, keywords are not case sensitive, but user-defined identifiers are case sensitive. In code examples, keywords are in ALL CAPS to make clear the distinction between keywords and user-defined identifiers.

> ⚠ In a very few cases, some option keywords are case-sensitive. For example, in the command to delete all data from the graph store,
> clear graph store -HARD
>
> the option -HARD must be in all capital letters.

- **Placeholder identifiers and values**
  In template code, any token that is not a keyword, a literal value, or punctuation is a placeholder identifier or a placeholder value.
  Example:

```
CREATE UNDIRECTED EDGE edge_type_name (FROM vertex_type_name1 , TO vertex_
attribute_name type [DEFAULT default_value ],...)
```

The user-defined identifiers are *edge_type_ name* , *vertex_type_name1*, *vertex_type_name2*, *attribute_name* and *default_value* . As explained in the Create Vertex section, **type** is one of the attribute data types.

- **Quotation Marks**
  When quotation marks are shown, they are to be typed as shown (unless stated otherwise). A placeholder for a string value will not have quotation marks in the template code, but if a template is converted to actual code, quotation marks should be used around string values.

- **Choices**
  The vertical bar | is used to separate the choices, when the syntax requires that the user choose one out of a set of values. Example: Either the keyword `VERTEX` or `EDGE` is to be used. Also, note the inclusion of quotation marks.

  Template:

```
LOAD " file_path " TO VERTEX|EDGE object_type_name VALUES (id_expr, attr_e
```

  Possible actual values:

```
LOAD "data/users.csv" TO VERTEX user VALUES ($0, $1, $2)
```

- **Optional content**
  Square brackets are used to enclose a portion that is optional. Options can be nested. Square brackets themselves are rarely used as part of the GSQL language itself.
  Example: In the RUN JOB statement, the -n flag is optional. If used, -n is to be followed by a value.

```
RUN JOB [-n count ] job_name
```

Sometimes, options are nested, which means that an inner option can only be used if the outer option is used:

```
RUN JOB [-n [ first_line_num , ] last_line_num ] job_name
```

means that *first_line_num* may be specified if and only if *last_line_num* is specified first. These options provide three possible forms for this statement:

```
RUN JOB job_name
RUN JOB -n last_line_num job_name
RUN JOB -n first_line_num , last_line_num job_name
```

- **Repeated zero or more times**
  In template code, it is sometimes desirable to show that a term is repeated an arbitrary number of times. For example, a vertex definition contains zero or more user-defined attributes. A loading job contains one or more LOAD statements. In

formal template code, if an asterisk (Kleene star) immediately follows option brackets, then the bracketed term can be repeated zero or more times. For example:

```
TO VERTEX|EDGE object_name VALUES ( id_expr [, attr_expr ]*)
```

means that the VALUES list contains at least one attribute expression. It may be followed by any number of additional attribute expressions. Each additional attribute expression must be preceded by a comma.

- **Long lines**

  For more convenient display, long statements in this guide may sometimes be displayed on multiple lines. This is for display purposes only; the actual code must be entered as a single line (unless the multi-line mode is used). When necessary, the examples may show a shell prompt before the start of a statement, to clearly mark where each statement begins.
  Example: A SELECT query is grammatically a single statement, so GSQL requires that it be entered as a single line.

Long statement displayed as one line

```
SELECT *|attribute_name FROM vertex_type_name [WHERE conditions] [ORDER BY
```

However, the statement is easier to read and to understand when displayed one clause per line:

Long statement displayed on multiple lines but with only one prompt

```
SELECT *|attribute_name
    FROM vertex_type_name
    [WHERE conditions]
    [ORDER BY attribute1,attribute2,...]
    [LIMIT k]
```

System and Language Basics

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 if 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`

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

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.
exit_on_error	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 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 VE

CREATE UNDIRECTED EDGE e2 (FROM u, TO v) #error 2: vertex type u doesn't e
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 l
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]+	from –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]+	from 0 to 2^64 - 1 (18,446,744,073,709,551,615)	8-byte unsigned integer
FLOAT	0.0	[-+] ? [0 - 9] * \. ? [0 - 9] + ([eE] [-+] ? [0 - 9] +) ?	+/- 3.4 E +/-38, ~7 bits of precision	4-byte single-precision floating point 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

⚠ 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	empty string	.*	UTF-8	string with a finite set of categorical values. The GSQL system uses dictionary encoding to assign a unique integer to each new string value, and then to store the

				values as integers.
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 support following complex data types:

Collection Types

- **LIST/SET** : A set is a **unordered** collection of unique elements of the same type; A list is an **ordered** collection of elements of the same type. A list can contain duplicate elements; a set cannot. The default value of either is an empty list/set. The supported element types of a list or a set are INT, UINT, DOUBLE, FLOAT, STRING, STRING COMPRESS, DATETIME, and UDT. To declare a list or set type, use <> brackets to enclose the element type, e.g., SET<INT>, LIST<STRING COMPRESS>.

⚠ 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.

- **MAP** : A map is a collection of key-value pairs. It cannot contain duplicate keys, and each key maps to one value. The default value is an empty map. The supported key types are INT, STRING, STRING COMPRESS, and DATETIME. The supported value types are 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 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. A 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 Choices (in Byte, except STRING)	Range (N is size)
INT	optional	1, 2, 4 (default), 8	0 to 2^(N*8) - 1
UINT	optional	1, 2, 4 (default), 8	-2^(N*8 - 1) to 2^(N*8 - 1) - 1
FLOAT	no		same as FLOAT attribute
DOUBLE	no		same as DOUBLE attribute
DATETIME	no		same as DATETIME attribute
BOOL	no		true, false
STRING	required	Any number of characters	Any string in N characters

To define a UDT, use the TYPEDEF TUPLE statement, before using the UDT as a field in a vertex type or edge type. Below is an example of a TYPEDEF TUPLE statement.

Example: UDT definition

```
TYPEDEF TUPLE <field1 INT (1), field2 UINT, field3 STRING (10), field4
```

In this example, myTuple is the name of this UDT. It contains four fields: a 1-byte INT field named field1, a 4-byte UINT field named field2, a 10-character STRING field named field3, and a (8-byte) DOUBLE field named field4.

Defining a Graph Schema

Before data can be loaded into the graph store, the user must define a ***graph schema.*** A graph schema is a "dictionary" that defines the types of entities, ***vertices*** and ***edges*** , in the graph and how those types of entities are related to one another. Each vertex or edge type has a name and a set of attributes (properties) associated with it. For example, a Book vertex could have title, author, publication year, genre, and language attributes.

In the figure below, circles represent vertex types, and lines represent edge types. The labeling text shows the name of each type. This example has four types of vertices: *User, Occupation, Book,* and *Genre* . Also, the example has 3 types of edges: *user_occupation, user_book_rating,* and *book_genre* . Note that this diagram does not say anything about how many users or books are in the graph database. It also does not indicate the cardinality of the relationship. For example, it does not specify whether a User may connect to multiple occupations.

An edge connects two vertices; in TigerGraph terminology these two vertices are the ***source vertex*** and the ***target vertex*** . An edge type can be either ***directed*** or ***undirected*** . A directed edge has a clear semantic direction, from the source vertex to the target vertex. For example, if there is an edge type that represents a plane flight segment, each segment needs to distinguish which airport is the origin (source vertex) and which airport is the destination (target vertex). In the example schema below, all of the edges are undirected. A useful test to decide whether an edge should be directed or undirected is the following: "An edge type is directed if knowing there is a relationship from A to B does not tell me whether there is a relationship from B to A." Having nonstop service from Chicago to Shanghai does not automatically imply there is nonstop service from Shanghai to Chicago.

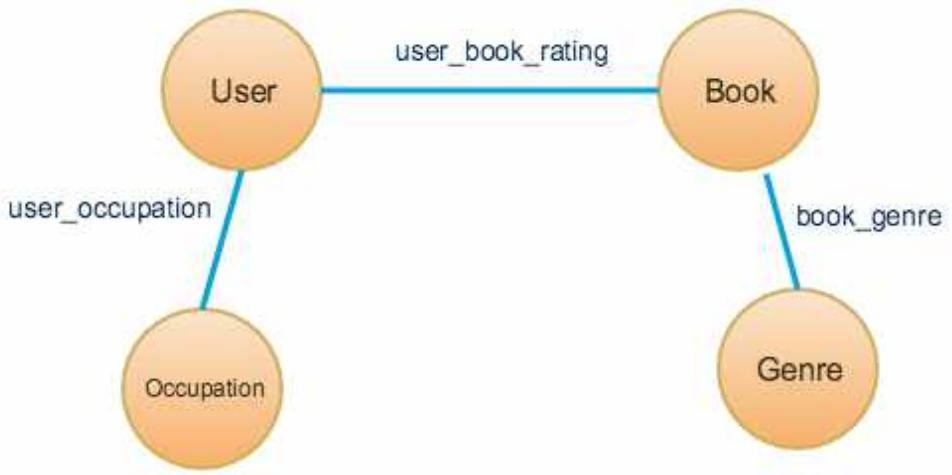


Figure 1 - A schema for a User-Book-Rating graph

An expanded schema is shown below, containing all the original vertex and edge types plus three additional edge types: *friend_of, sequel_of, and user_book_read* . Note that *friend_of* joins a User to a User. The friendship is assumed to be bidirectional, so the edge type is undirected. *Sequel_of* joins a Book to a Book but it is directed, as evidenced by the arrowhead. *The Two Towers* is the sequel of *The Fellowship of the Ring* , but the reverse is not true. User_book_read is added to illustrate that there may be more than one edge type between a pair of vertex types.

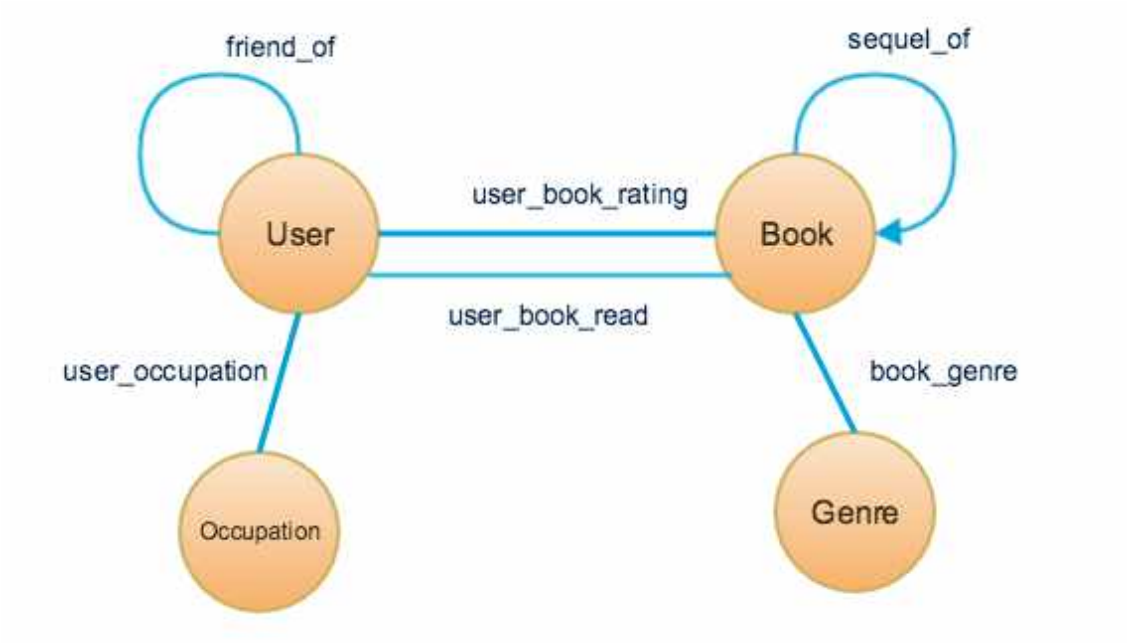


Figure 2 - Expanded-User-Book-Rating schema with additional edges

The TigerGraph system user designs a graph schema to fit the source data and the user's needs and interests. The TigerGraph system user should consider what type

of relationships are of interest and what type of analysis is needed. The TigerGraph system lets the user modify an existing schema, so the user is not locked into the initial design decision.

In the first schema diagram above, there are seven entities: four vertex types and three edge types.You may wonder why it was decided to make Occupation a separate vertex type instead of an attribute of User. Likewise, why is Genre a vertex type instead of an attribute of Book? These are examples of design choices. Occupation and Genre were separated out as vertex types because in graph analysis, if an attribute will be used as a query variable, it is often easier to work with as a vertex type.

Once the graph designer has chosen a graph schema, the schema is ready to be formalized into a series of GSQL statements.

> Graph Creation and Modification Privileges
>
> Only superusers and globaldesigners can define global vertex types. global edge types, and graphs, using CREATE VERTEX / EDGE / GRAPH.
> However, once a graph has been created, its admin and designer users can customize its schema, including adding new local vertex types and local edge types, by using a SCHEMA_CHANGE JOB, described at Modifying a Graph Schema.

CREATE VERTEX

Available to superuser and globaldesigner roles only.

The CREATE VERTEX statement defines a new global vertex type, with a name and an attribute list. At a high level of abstraction, the format is

```
CREATE VERTEX vertex_type_name (id_and_attribute_list) [ vertex_options ]
```

More specifically, the syntax is as follows, assuming that the vertex ID is listed first:

CREATE VERTEX Syntax

```
CREATE VERTEX vertex_type_name (primary_id_name_type
    [, attribute_name type [DEFAULT default_value] ]*)
    [WITH [STATS="none"|"outdegree_by_edgetype"][primary_id_as_attribute="
```

Keys and Attributes

Beginning with v2.3, there are two syntaxes for specifying the primary id/key:

Legacy PRIMARY_ID syntax: The legacy syntax remains valid, but there are additional options and additional flexibility:

PRIMARY KEY syntax. This syntax is modeled after SQL.

PRIMARY_ID and WITH primary_id_as_attribute

The primary_id is a required field whose purpose is to uniquely identify each vertex instance. GSQL creates a hash index on the primary id with O(1) time complexity. Its data type may be STRING, INT, or UINT. The syntax for the primary_id_name_type term is as follows:

```
primary_id_name_type := PRIMARY_ID id_name id_type
```

NOTE: In default mode, the primary_id field is not one of the attribute fields. The purpose of this distinction is to minimize storage space for vertices. The functional consequence of this difference is that a query cannot read the primary_id or use it as part of an expression.

Beginning with v2.3:

1. The Primary_id can be treated as an attribute, if the clause
   WITH primary_id_as_attribute="true"
   is used with the CREATE VERTEX statement.
2. The primary_id designation can be used with any one of the attributes; it is not restricted to the first attribute.

Example:

```
CREATE VERTEX movie (PRIMARY_ID id UINT, name STRING, year UINT)
    WITH primary_id_as_attribute="true"
```

PRIMARY KEY

Instead of the legacy PRIMARY_ID syntax, starting with v2.3, GSQL now offers another option for specifying the primary key. The keyword phrase PRIMARY KEY may be appended to any one of the attributes in the attribute list, though it is conventional for it to be the first attribute. Each vertex instance must have a unique value for the primary key attribute. GSQL creates a hash index on the PRIMARY KEY attribute with O(1) time complexity. It is recommended that the primary key data type be STRING, INT, or UINT.

```
primary_id_name_type := id_name_id_type PRIMARY KEY
```

Note the differences between PRIMARY_ID and PRIMARY KEY:

1. "PRIMARY_ID" precedes the (name, type) pair. "PRIMARY KEY" follows the (name, type) pair.
2. In default mode, a PRIMARY_ID is not an attribute, but the WITH primary_id_as_attribute="true" clause can be used to make it an attribute. Alternately, the PRIMARY KEY is always an attribute; the WITH option is unneeded.

Example:

```
CREATE VERTEX movie (id UINT PRIMARY KEY, name STRING, year UINT)
```

⚠ PRIMARY KEY is not supported in GraphStudio. If you decide to use this feature, you will only be able to use command line interface.

COMPOSITE KEY

Beginning with v2.4, GSQL PRIMARY KEY supports composite keys - grouping multiple attributes to create a primary key for a specific vertex. Composite Key

usage is similar to a single PRIMARY KEY, but rather than appending "PRIMARY KEY" after an attribute, the syntax is a bit different.

```
composite_id_name_type := PRIMARY KEY "(" attribute_name ("," attribute_na
```

Example:

```
CREATE VERTEX movie (id UINT, title STRING, year UINT, PRIMARY KEY (title,
```

⚠ COMPOSITE KEY is not supported in GraphStudio. If you decide to use this feature, you will only be able to use command line interface.

Vertex Attribute List

The attribute list, enclosed in parentheses, is a list of one or more *id definitions* and *attribute descriptions* separated by commas:

```
primary_id_name_type
[, attribute_name type [DEFAULT default_value ] ]*
```

The available attribute types, including user-defined tuples, are listed in the section Attribute Types.

1. Every attribute data type has a built-in default value (e.g., the default value for INT type is 0). The `DEFAULT default_value` option overrides the built-in value.
2. Any number of additional attributes may be listed after the primary_id attribute. Each attribute has a name, type, and optional default **value** (for primitive-type, DATETIME, or STRING COMPRESS attributes only)

Example:

- Create vertex types for the graph schema of Figure 1.

Vertex definitions for User-Book-Rating graph

```
CREATE VERTEX User (PRIMARY_ID user_id UINT, name STRING, age UINT, gender
CREATE VERTEX Occupation (PRIMARY_ID occ_id UINT, occ_name STRING)
    WITH STATS="outdegree_by_edgetype"
CREATE VERTEX Book  (PRIMARY_ID bookcode UINT, title STRING, pub_year UINT
    WITH STATS="none"
CREATE VERTEX Genre (PRIMARY_ID genre_id STRING, genre_name STRING)
```

Unlike the tables in a relational database, vertex types do not need to have a foreign key attribute for one vertex type to have a relationship to another vertex type. Such relationships are handled by edge types.

WITH STATS

By default, when the loader stores a vertex and its attributes in the graph store, it also stores some statistics about the vertex's outdegree – how many connections it has to other vertices. The optional WITH STATS clause lets the user control how much information is recorded. Recording the information in the graph store will speed up queries which need degree information, but it increases the memory usage. There are two* options. If "outdegree_by_edgetype" is chosen, then each vertex records a list of degree count values, one value for each type of edge in the schema. If "none" is chosen, then no degree statistics are recorded with each vertex. If the WITH STATS clause is not used, the loader acts as if "outdegree_by_edgetype" were selected.

The graph below has two types of edges between persons: phone_call and text. For Bobby, the "outdegree_by_edgetype" option records how many phone calls Bobby made (1) and how many text messages Bobby sent (2). This information can be retrieved using the built-in vertex function outdegree(). To get the outdegree of a specific edge type, provide the edgetype name as a string parameter. To get the total outdegree, omit the parameter.

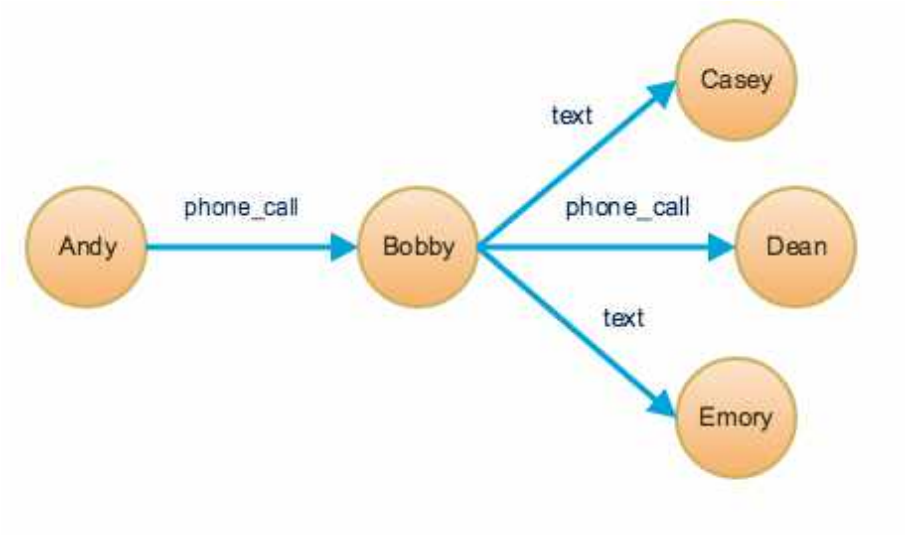


Figure 3 - Outdegree stats illustration

WITH STATS option (case insensitive)	Bobby.outdegree()	Bobby.outdegree("text")	Bobby.outdegree("phone_call")
"none"	not available	not available	not available
"outdegree_by_edgetype" (default)	3	2	1

CREATE EDGE

Available to superuser and globaldesigner roles only.

The CREATE EDGE statement defines a new global edge type. There are two forms of the CREATE EDGE statement, one for directed edges and one for undirected edges. Each edge type must specify that it connects FROM one vertex type TO another vertex type. Additional attributes may be added. Each attribute follows the same requirements as described in the Attribute List subsection for the "CREATE VERTEX" section.

CREATE UNDIRECTED EDGE

```
CREATE UNDIRECTED EDGE edge_type_name (FROM vertex_type_name, TO vertex_ty
  [, attribute_name type [DEFAULT default_value]]* )
```

CREATE DIRECTED EDGE

```
CREATE DIRECTED EDGE edge_type_name (FROM vertex_type_name, TO vertex_type
  [, attribute_name type [DEFAULT default_value]]* )
  [WITH REVERSE_EDGE="rev_name"]
```

Viewed at a higher level of abstraction, the format is

```
CREATE UNDIRECTED|DIRECTED EDGE edge_type_name (FROM vertex_type_name , TO
edge_attribute_list ) [ edge_options ]
```

Note that edges do not have a PRIMARY_ID field. Instead, each edge is uniquely identified by a FROM vertex, a TO vertex, and optionally other attributes. The edge type may also be a distinguishing characteristic. For example, as shown in Figure 2 above, there are two types of edges between User and Book. Therefore, both types would have attribute lists which begin `(FROM User, To Book,...).`

An edge type can be defined which connects FROM any type of vertex and/or TO any type of vertex. Use the wildcard symbol * to indicate "any vertex type". For example, the any_edge type below can connect from any vertex to any other vertex:

Wildcard edge type

```
CREATE DIRECTED EDGE any_edge (FROM *, TO *, label STRING)
```

WITH REVERSE_EDGE

If a CREATE DIRECTED EDGE statement includes the WITH REVERSE_EDGE=" *rev_name* " optional clause, then an additional directed edge type called " `rev_name` " is automatically created, with the FROM and TO vertices swapped. Moreover, whenever a new edge is created, a reverse edge is also created. The reverse edge will have the same attributes, and whenever the principal edge is updated, the corresponding reverse edge is also updated.

In a TigerGraph system, reverse edges provide the most efficient way to perform graph queries and searches that need to look "backwards". For example, referring to the schema of Figure 2, the query "What is the sequel of Book X, if it has one?" is

a forward search, using *sequel_of* edges. However, the query "Is Book X a sequel? If so, what Book came before X?" requires examining reverse edges.

Example:

Create undirected edges for the three edge types in Figure 1.

Edge definitions for User-Book-Rating graph

```
CREATE UNDIRECTED EDGE user_occupation (FROM User, TO Occupation)
CREATE UNDIRECTED EDGE book_genre (FROM Book, TO Genre)
CREATE UNDIRECTED EDGE user_book_rating (FROM User, TO Book, rating UINT,
```

The `user_occupation` and `book_genre` edges have no attributes. A `user_book_rating` edge symbolizes that a user has assigned a rating to a book. Therefore it includes an additional attribute `rating`. In this case the `rating` attribute is defined to be an integer, but it could just as easily have been set to be a float attribute.

Example :

Create the additional edges depicted in Figure 2.

Additional Edge definitions for Expanded-User-Book-Rating graph

```
CREATE UNDIRECTED EDGE friend_of (FROM User, TO User, on_date UINT)
CREATE UNDIRECTED EDGE user_book_read (FROM User, To Book, on_date UINT)
CREATE DIRECTED EDGE sequel_of (FROM Book, TO Book) WITH REVERSE_EDGE="pre
```

Every time the GSQL loader creates a `sequel_of` edge, it will also automatically create a `preceded_by` edge, pointing in the opposite direction.

Special Options

Sharing a Compression Dictionary

The STRING COMPRESS and STRING_SET COMPRESS data types achieve compression by mapping each unique attribute value to a small integer. The mapping table ("this string" = "this integer") is called the dictionary. If two such attributes have the same or similar sets of possible values, then it is desirable to have them share one dictionary because it uses less storage space.

When a STRING COMPRESS attribute is declared in a vertex or edge, the user can optionally provide a name for the dictionary. Any attributes which share the same dictionary name will share the same dictionary. For example, v1.attr1, v1.attr2, and e.attr1 below share the same dictionary named "e1".

Shared STRING COMPRESS dictionaries

```
CREATE VERTEX v1 (PRIMARY_ID main_id STRING, att1 STRING COMPRESS e1, att2
CREATE UNDIRECTED EDGE e (FROM v1, TO v2, att1 STRING COMPRESS e1)
```

CREATE GRAPH

Available to superuser and globaldesigner roles only.

> Multiple Graph support
>
> If the optional MultiGraph service is enabled, CREATE GRAPH can be invoked multiple times to define multiple graphs, and vertex types and edge types may be shared among multiple graphs. There is an option to assign an admin user for the new graph.

After all the required vertex and edge types are created, the CREATE GRAPH command defines a graph schema which contains the given vertex types and edge types, and prepares the graph store to accept data. The vertex types and edge types may be listed in any order.

CREATE GRAPH syntax

```
CREATE GRAPH gname (vertex_or_edge_type, vertex_or_edge_type...) [WITH ADM
```

The optional WITH ADMIN clause sets the named user to be the admin for the new graph.

As a convenience, executing CREATE GRAPH will set the new graph to be the working graph.

Instead of providing a list of specific vertex types and edge types, it is also possible to define a graph type which includes all the available vertex types and edge types. It is also legal to create a graph with an empty domain. A SCHEMA_CHANGE can be used later to add vertex and edge types.

Examples of CREATE GRAPH with all vertex & edge types and with an empty domain.

```
CREATE GRAPH everythingGraph (*)
CREATE GRAPH emptyGraph ()
```

Examples :

Create graph Book_rating for the edge and vertex types defined for Figure 1:

Graph definition for User-Book-Rating graph

```
CREATE GRAPH Book_rating (*)
```

The following code example shows the full set of statements to define the expanded user-book-rating graph:

Full definition for the Expanded User-Book-Rating graph

```
CREATE VERTEX User (PRIMARY_ID user_id UINT, name STRING, age UINT, gende
CREATE VERTEX Occupation (PRIMARY_ID occ_id UINT, occ_name STRING)
    WITH STATS="outdegree_by_edgetype"
CREATE VERTEX Book  (PRIMARY_ID bookcode UINT, title STRING, pub_year UIN
    WITH STATS="none"
CREATE VERTEX Genre (PRIMARY_ID genre_id STRING, genre_name STRING)
CREATE UNDIRECTED EDGE user_occupation (FROM User, TO Occupation)
CREATE UNDIRECTED EDGE book_genre (FROM Book, TO Genre)
CREATE UNDIRECTED EDGE user_book_rating (FROM User, TO Book, rating UINT,
CREATE UNDIRECTED EDGE friend_of (FROM User, TO User, on_date UINT)
CREATE UNDIRECTED EDGE user_book_read (FROM User, To Book, on_date UINT)
CREATE DIRECTED EDGE sequel_of (FROM Book, TO Book) WITH REVERSE_EDGE="pre
CREATE GRAPH Book_rating (*)
```

USE GRAPH

New requirement for MultiGraph support. Applies even if only one graph exists.

Before a user can make use of a graph, first the user must be granted a role on that graph by an admin user of that graph or by a superuser. (Superusers are automatically granted the admin role on every graph). Second, for each GSQL session, the user must set a working graph. The USE GRAPH command sets or changes the user's working graph, for the current session.

For more about roles and privileges, see the document Managing User Privileges and Authentication.

USE GRAPH syntax

```
USE GRAPH gname
```

Instead of the USE GRAPH command, gsql can be invoked with the -g <graph_name> option.

DROP GRAPH

Available to superuser and globaldesign roles only. The effect of this command takes into account shared domains.

DROP GRAPH syntax

```
DROP GRAPH gname
```

The DROP GRAPH deletes the logical definition of the named graph. Furthermore, if any of the vertex types or edge types in its domain are not shared by any other graph, then those non-shared types and their data are deleted. Any shared types

are unaffected. To delete only selected vertex types or edge types, see DROP VERTEX | EDGE in the Section "Modifying a Graph Schema".

SHOW - View Parts of the Catalog

The SHOW command can be used to show certain aspects of the graph, instead of manually filtering through the entire graph schema when using the ls command. You can either type the exact identifier or use regular expression / Linux globbing to search.

```
SHOW <VERTEX> | <EDGE> | <JOB> | <QUERY> | <GRAPH> [ <name> | <glob> | -r
```

This feature supports the **?** and ***** from linux globbing operations, and also regular expression matching. Usage of the feature is limited to the scope of the graph the user is currently in - if you are using a global graph, you will not be able to see vertices that are not included in your current graph.

> Regular expression searching will not work with escaping characters.

To use regular expressions, you will need to use the **-r** flag after the part of the schema you wish to show. If you wish to dive deeper into regular expressions, visit "Java Patterns". The following are a few examples of what is supported by the SHOW command.

```
Linux Globbing examples
SHOW VERTEX us*            //shows all vertices that start with the lette
SHOW VERTEX co?*y          //shows the vertex that starts with co and ends
SHOW VERTEX ?????          //shows all vertices that are 5 letters long

Regular Expression Examples
SHOW VERTEX -r "skil{2}"    //match the pattern "skill"
SHOW EDGE -r "test[1][13579]*"    //match pattern that only contains odd r
SHOW JOB -r "[a-zA-Z]*"     //match all jobs that contain only letters
```

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 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 STA
ADD UNDIRECTED EDGE e_type_name (FROM v_type_name',' TO v_type_name [',' e
ADD DIRECTED EDGE e_type_name (FROM v_type_name',' TO v_type_name [',' edg
    [WITH REVERSE_EDGE '=' "rev_name"];
```

ALTER VERTEX | EDGE

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 ... 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]* );
```

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]*
```

RUN SCHEMA_CHANGE JOB

RUN JOB job_name performs the schema change job. After the schema has been changed, the GSQL system checks all existing GSQL queries (described in "GSQL Language Reference, Part 2: Querying"). 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 STRIN
    ADD UNDIRECTED EDGE wrote_review (FROM User, TO Review);
    ADD UNDIRECTED EDGE review_of_book (FROM Review, TO Book);
}
RUN JOB add_reviews
```

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 which 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
}
```

Those both global and local schema change jobs have ADD and DROP statements, they have different meanings. The table below outlines the differences.

	local SCHEMA_CHANGE	GLOBAL SCHEMA_CHANGE
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 vertex/edge type.	Adds or drops attributes from a global vertex/edge type, which may affect several graphs.

⚠ 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 ... 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]* );
```

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 JOB job_name** performs the global schema change job. After the schema has been changed, the GSQL system checks all existing GSQL queries (described in "GSQL Language Reference, Part 2: Querying"). 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 JOB alter_friendship_make_library
```

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 which 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 the RUN statement. These two statements, and the components with the loading job, are detailed below.

The structure of a loading job will be presented hierarchically, top-down:

CREATE ... JOB, which may contain a set of DEFINE and LOAD statements

- DEFINE statements
- LOAD statements, which can have several clauses

New LOADING JOB Capabilities

Beginning with v2.0, the TigerGraph platform introduces an extended syntax for defining and running loading jobs which 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. Previously, RESTPP could manage only one loading job at a time. In v2.0, 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 RESTPP-LOADER.Replicas.

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.

To provide this added capability for loading, there is an expanded syntax for creating loading jobs and running loading jobs. Below is a summary of changes and additions. Full details are then presented, in the remainder of this document (GSQL Language Reference Part 1).

- A loading job begins with CREATE LOADING JOB. (Note that the keyword "LOADING" is included.)
- A new statement type, DEFINE FILENAME, is added, to define filename variables.
- The file locations can refer either to the local machine, to specific machines, or to all machines.

- When a job starts, it is assigned a job_id. Using the job_id, you can check status, abort a job, or restart a job.

Below is a simple example:

Concurrent Loading Job Example

```
CREATE LOADING JOB job1 FOR GRAPH graph1 {

   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 LOADING JOB job1 USING file1="m1:/data/v1_1.csv", file2="m2:/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 each file variable. Then, the job can be run in portions, according to each file variable.

⚠ pre-v2.0 CREATE JOB syntax is deprecated

If the new CREATE LOADING JOB syntax with DEFINE FILENAME is used, the user can take advantage of concurrent loading.

Pre-v2.0 loading syntax will still be supported for v2.x but is deprecated. Pre-v2.0 loading syntax does not offer concurrent loading.

✓ Example loading jobs and data files for the book_rating schema defined earlier in the document are available in the **/doc/examples/gsql_ref** folder in your TigerGraph platform installation.

CREATE LOADING JOB Block

The v2.0 CREATE LOADING JOB can be distinguished from the pre-v2.0 loading jobs first by its header, and then by whether its contains DEFINE FILENAME statements or not. Once the loading type has been determined, there are subsequent rules for the format of the individual LOAD statements and then the RUN statement.

Loading type	Block Header	Has DEFINE FILENAME statements?	Run
v2.0 loading	CREATE LOADING JOB	Yes	RUN LOADING JOB
Non-concurrent offline loading (DEPRECATED)	CREATE LOADING JOB	No	RUN JOB
Non-concurrent online loading (DEPRECATED)	CREATE ONLINE_POST JOB	Not permitted	RUN JOB USING FILENAME...

ⓘ The CREATE LOADING JOB and DROP LOADING JOB privileges are reserved for the designer, admin, and superuser roles.

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 LOAD for offline loading

```
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
As of version 2.2, a LOADING JOB may contain either LOAD or DELETE statements but not both.
A JOB which includes both will be rejected when the CREATE statement is executed.

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* . I **f the path is not valid 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 *path* . **If the path is not valid on any of the machines, those machines 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 statements

A LOAD statement tells the GSQL loader 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 ***Platform Knowledge Base / FAQs*** and the tutorials, such as ***Get Started with TigerGraph*** , 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.

LOAD statement

```
LOAD [filepath_string|filevar|TEMP_TABLE table_name ] Destination_Clause |
```

The *filevar* must have been previously defined in a DEFINE FILENAME statement.

The *filepath_string* must satisfy the same rules given above in the DEFINE FILENAME section.

> "__GSQL_FILENAME_n__": Position-based File Identifiers
>
> When a CREATE LOADING JOB block is processed, the GSQL system will count the number of unique filepath_strings and assign them position-based index numbers 0, 1, 2, etc. starting from the top. A filepath_string is considered one item, even if it has multiple machine indexes and file locations. These index numbers can then be used as an alternate naming scheme for the filespath_strings:
>
> When running a loading job, the nth filepath_string can be referred as "__GSQL_FILENAME_n__", where n is replaced with the index number. Note that the string has double underscores at both the left and right ends.

The remainder of this section of the document will provide details on the format and use of the file_path, Destination_Clause, its subclauses. USING clause is introduced later in Section "Other Optional LOAD Clauses".

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, target_id_expr [, attr_
    [WHERE conditions] [OPTION (options)]
```

TEMP_TABLE Destination Clause

```
TO TEMP_TABLE table_name (id_name [, attr_name]*) VALUES (id_expr [, a
    [WHERE conditions] [OPTION (options)]
```

For the TO VERTEX and TO EDGE destination clauses, 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.

In contrast, the TO TEMP_TABLE clause is defining 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.

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. 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 is creating a new vertex or edge, then the skipped attribute will be assigned the default value.
2. If the LOAD 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 are a month (1 to 12) and a 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 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 > myTupl
CREATE VERTEX v_udt  (id STRING PRIMARY KEY, att_udt myTuple)
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 v_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 which 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 through 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 which 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 that 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 parameters. See the example below.

Example: loading composite key to vertex and edge

```
#schema setup
CREATE VERTEX compositePerson (id uint, name string, PRIMARY KEY (name
CREATE VERTEX compositeMovie (id uint, title string, country string, y
CREATE DIRECTED EDGE compositeRoles (from compositePerson,to composite
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", se

  LOAD "$sys.data_root/persons.csv" TO VERTEX compositePerson VALUES
       ($"id",$"name") USING header = "true", separator =",";

  LOAD "$sys.data_root/compositeroles.csv" TO EDGE compositeRoles VALU
       (($"personName", $"personId"),($"movieTitle",$"movieYear",$"mov
       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);
  }
```

Built-in Loader Token Functions

The GSQL Loader provides several built-in 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.

Token Functions for Attribute Expressions

The following token functions can be used in an id or attribute expression

Function name and parameters	Output type	Description of function
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.
		Returns a modified version of *in_string* , in which each

gsql_split_by_space(*in_string*)	string	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.
gsql_regex_match(*str*, *regex*)	bool	Returns true if the given string token matches the 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, number_of_sub_fields_in_one_group*)		See the section "TEMP_TABLE and Flatten Functions" below.

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

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`

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, return(load) *arg* .

⚠ 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 operation can be used to express complex operation 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

⚠ The token functions in the WHERE clause and those token functions used for attribute expression are different. They cannot be used exchangeably.

User-Defined Token Functions

Users can write their own token functions in C++ and install them in the GSQL system. The system installation already contains a source code file containing sample functions. Users simply add their customized token functions to this file. The file for user-defined token functions for attribute expressions or WHERE clauses is at <tigergraph.root.dir>/dev/gdk/gsql/src/TokenBank/TokenBank.cpp. There are a few examples in this file, and details are presented below .

Testing your functions is simple. In the same directory with the TokenBank.cpp file is a command script called compile.

1. To test that your function compiles:

```
./compile
```

2. To test that your function works correctly, write your own test and add it to the main() procedure in the TokenBank.cpp. Then, compile the file and run it. Note that files located in ../TokenLib need to be included:

```
g++ -I../TokenLib TokenBank.cpp
./a.out
```

User-defined Token Functions for Attribute Expressions

Attribute type	Function signature
string or string compress	extern "C" void funcName (const char* const iToken[], uint32_t iTokenLen[], uint32_t iTokenNum, char* const oToken, uint32_t& oTokenLen)
bool	extern "C" bool funcName (const char* const iToken[], uint32_t iTokenLen[], uint32_t iTokenNum)
uint	extern "C" uint64_t funcName (const char* const iToken[], uint32_t iTokenLen[], uint32_t iTokenNum)
int	extern "C" int64_t funcName (const char* const iToken[], uint32_t iTokenLen[], uint32_t iTokenNum)
float	extern "C" float funcName (const char* const iToken[], uint32_t iTokenLen[], uint32_t iTokenNum)
double	extern "C" double funcName (const char* const iToken[], uint32_t iTokenLen[], uint32_t iTokenNum)

The parameters are as follows: iToken is the array of string tokens, iTokenLen is the array of the length of the string tokens, and iTokenNum is the number of tokens. Note that the input tokens are always in string (char*) format.

If the attribute type is not string nor string compress, the return type should be the corresponding type: bool for bool; uint64_t for uint; int64_t for int; float for float double for double. If the attribute type is string or string compress, the return type should be void, and use the extra parameters (char *const oToken, uint32_t& oTokenLen) for storing the return string. oToken is the returned string value, and oTokenLen is the length of this string.

The built-in token function gsql_concat is used as an example below. It takes multiple-token parameter and returns a string.

gsql_concat

```
extern "C" void gsql_concat(const char* const iToken[], uint32_t iToke
  int k = 0;
  for (int i=0; i < iTokenNum; i++) {
    for (int j =0; j < iTokenLen[i]; j++) {
           oToken[k++] = iToken[i][j];
    }
  }
  oTokenLen = k;
}
```

User-defined Token Functions for WHERE Clause

User-defined token functions (described above) can also be used to construct the boolean conditional expression in the WHERE clause. However, there are some restrictions in the WHERE clause:

⚠ In the clause "WHERE *conditions* ",

- The only type of user-defined token function allowed are those that return a boolean value.
- If a user-defined token function is used in a WHERE Clause, then it must constitute the entire condition; it cannot be combined with another function or operator to produce a subsequent value. However, the arguments of the UDF can include other functions.

The source code for the built-in token function gsql_token_equal is used as an example for how to write a user-defined token function.

gsql_token_equal

```
extern "C" bool gsql_token_equal(const char* const iToken[], uint32_t
  if (iTokenNum != 2) {
    return false;
  }
  if (iTokenLen[0] != iTokenLen[1]) {
    return false;
  }
  for (int i =0; i < iTokenLen[0]; i++) {
    if (iToken[0][i] != iToken[1][i]) {
      return false;
    }
  }
  return true;
}
```

Other Optional LOAD Clauses

OPTION clause

There are no supported options for the OPTION clause at this time.

USING clause

A USING clause contains one or more optional parameter value pairs:

```
USING parameter=value [,parameter=value]*
```

⚠ In the v2.0 loading syntax, the USING clause only appears at the end of a LOAD statement.

In earlier versions, the location of the USING clause and which parameters were valid depending the whether the job was a v1.x online loading job or v1.x offline loading job.

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 "
USER_DEFINED_HEADER	specifies the name of the header variable, when a header has been defined in the loading job, rather than in the data file	the variable name in the preceding DEFINE HEADER 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 note below)	whether each line is a json object (see Section "JSON Loader" below for more details)	"true", "false" Default is "false"
	whether the data file's first line is a header line.	"true", "false"

HEADER	The header assigns names to the columns. The LOAD statement must refer to an actual file with a valid header.	Default is "false"
VERTEXMUSTEXIST (See note below)	specifies whether to require that the endpoint vertices of an edge must exist in order to load a given edge.	"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":"nam
    TO VERTEX members VALUES($"members",$"members") USING JSON_FILE="t
}
```

267B encoding.json

encoding.json - Download

encoding.json

```
{"encoding": "UTF-7","plug-ins":["c"],"indent" : { "length" : 30, "use
{"encoding":"UTF-1","indent":{"use_space": "dontloadme"}, "plug-ins" :
{"plug-ins":["C","c++"],"indent":{"length" : 3, "use_space": false},"e
```

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

VertexMustExist 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 VERTEXMUSTEXIST="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 VERTEXMUSTEXIST="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 [, a
    [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 into 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"
```

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"
```

The 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.

bookcode	genre
101	fiction
101	fantasy
101	young_adult
102	fiction
102	science_fiction
102	Chinese

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:

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:fantas
102|"The Three-Body Problem"|"FIC:fiction,SF:science fiction,CHN:Chine
```

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, flatt
      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"), $"indent":"length") USING
  LOAD TEMP_TABLE t2
    TO VERTEX encoding VALUES ($"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:

594B encoding3.json

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, inder
    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", $"inde
}
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
"go"	4.0	"new"	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:

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":3
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", $"inder
    USING SEPARATOR="|";
  LOAD TEMP_TABLE t4
    TO VERTEX encoding3 VALUES ($"name", $"score", $"prop_age", $"inde
}
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 cc

  # 2. Delete each edge which has the given edge type, source vertex i
  DELETE EDGE edge_type_name (FROM id_expr, TO id_expr) FROM f [WHERE

  # 3. Delete all edges which have the given edge type and source vert
  DELETE EDGE edge_type_name (FROM id_expr) FROM f [WHERE condition] ;

  # 4. Delete all edges which have the given source vertex id. (Edge t
  DELETE EDGE * (FROM id_expr vertex_type_name) FROM f [WHERE conditic
}
```

An example using book_rating data is shown below:

DELETE example

```
# Delete all user occupation edges if the user is in the new files, th
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_u
```

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.

offline2online Job Conversion (DEPRECATED)

```
offline2online <offline_job_name>
```

The gsql command offline2online converts an installed offline loading job to an equivalent online loading job or set of jobs.

Online Job Names

An offline loading job contains one or more LOAD statements, each one specifying the name of an input data file. The offline2online will convert each LOAD statement into a separate online loading job. The data filename will be appended to the offline job name, to create the new online job name. For example, if the offline job has this format:

```
CREATE LOADING JOB loadEx FOR GRAPH graphEx {
  LOAD "fileA" TO ...
  LOAD "fileB" TO ...
}
```

then running the GSQL command `offline2online loadEx` will create two new online loading jobs, called **loadEx_fileA** and **loadEx_fileB** . The converted loading jobs are installed in the GSQL system; they are not available as text files. However, if there are already jobs with these names, then a version number will be appended: first "_1", then "_2", etc.

For example, if you were to execute `offline2online loadEx` three times, this would generate the following online jobs:

- 1st time: loadEx_fileA, loadEx_fileB
- 2nd time: loadEx_fileA_1, loadEx_fileB_1
- 3rd time: loadEx_fileA_2, loadEx_fileB_2

Conversion and RUN JOB Details

⚠ Some parameters of a loading job which are built in to offline loading jobs instead cannot be included in online jobs:

- input data filename
- SEPARATOR
- HEADER

Instead, they should be provided when running the loading job. However, online jobs do not have full support for HEADER.

When running any online loading job, the input data filename and the separator character must be provided. See sections on the USING clause and Running a Loading Job for more details.

If an online loading job is run with the HEADER="true" option, it will skip the first line in the data file, but it will not read that line to get the column names. Therefore,

offline jobs which read and use column header names must be manually converted to online jobs.

The following example is taken from the Social Network case in the GSQL Tutorial with Real-Life Examples . In version 0.2 of the tutorial, we used offline loading. The job below uses the same syntax as v0.2, but some names have been updated:

Offline loading example, based on social_load.gsql, version 0.2

```
CREATE LOADING JOB load_social FOR GRAPH gsql_demo
{
  LOAD "data/social_users.csv"
    TO VERTEX SocialUser VALUES ($0,$1,$2,$3)
  USING QUOTE="double", SEPARATOR=",", HEADER="true";

  LOAD "data/social_connection.csv"
    TO EDGE SocialConn VALUES ($0, $1)
    USING SEPARATOR=",", HEADER="false";
}
```

To run, this job:

```
RUN LOADING JOB load_social
```

Note that the first LOAD statement has HEADER="true", but is does not make use of column names. It simply uses column indices $0, $1, $2, and $3. Therefore, the HEADER option can still be used with the converted job. Running **offline2online load_social1** , creates two new jobs called **load_social_social_users.csv** and **load_social_social_connection.csv.**

The equivalent run commands for the jobs are the following:

```
RUN LOADING JOB load_social_social_users.csv USING FILENAME="data/social_u
RUN LOADING JOB load_social_social_connection.csv USING FILENAME="data/soc
```

For comparison, here is the online loading job in the current version of the Tutorial and its loading commands:

social_load.gsql, version 0.8.1

```
CREATE LOADING JOB load_social1 FOR GRAPH gsql_demo
{
  LOAD
    TO VERTEX SocialUser VALUES ($0,$1,$2,$3) USING QUOTE="double";
}
CREATE LOADING JOB load_social2 FOR GRAPH gsql_demo {
  LOAD
    TO EDGE SocialConn VALUES ($0, $1);
}
# load the data
RUN JOB load_social1 USING FILENAME="../social/data/social_users.csv",
RUN JOB load_social2 USING FILENAME="../social/data/social_connection.
```

Running a Loading Job

Clearing and Initializing the Graph Store

There are two aspects to clearing the system: flushing the data and clearing the schema definitions in the catalog. Two different commands are available.

CLEAR GRAPH STORE

ⓘ Available only to superusers.

The CLEAR GRAPH STORE command flushes all the data out of the graph store (database). By default, the system will ask the user to confirm that you really want to discard all the graph data. To force the clear operation and bypass the confirmation question, use the -HARD option, e.g.,

```
CLEAR GRAPH STORE -HARD
```

Clearing the graph store does not affect the schema.

⚠
1. Use the -HARD option with extreme caution. There is no undo option. -HARD must be in all capital letters.
2. CLEAR GRAPH STORE stops all the TigerGraph servers (GPE, GSE, RESTPP, Kafka, and Zookeeper).
3. Loading jobs and queries are aborted.

DROP ALL

ⓘ Available only to superusers.

The DROP ALL statement clears the graph store and removes all definitions from the catalog: vertex types, edge types, graph types, jobs, and queries.

Running a Loading Job

Running a loading job executes a previously installed loading job. The job reads lines from an input source, parses each line into data tokens, and applies loading rules and conditions to create new vertex and edge instances to store in the graph data store.

TigerGraph 2.0 introduces enhanced data loading with slightly modified syntax for CREATE and RUN statements. The previous RUN JOB syntaxes for v1.x online loading and offline loading and still supported for backward compatibility. Additionally, loading jobs can also be run by directly submitted a HTTP request to the REST++ server.

RUN LOADING JOB

> pre-v2.0 RUN JOB syntax is deprecated
>
> As of v2.0, RUN LOADING JOB is the preferred syntax for running all loading jobs. The pre-v2.0 syntaxes for running online post jobs and offline loading jobs are still support for now but are deprecated.

RUN LOADING JOB syntax for concurrent loading

```
RUN LOADING JOB [-noprint] [-dryrun] [-n [i],j] jobname [ USING filevar [=
```

Note that the keyword LOADING is included. This makes it more clear to users and to GSQL that the job is a loading job and not some other type of job (such as a SCHEMA_CHANGE JOB).

When a concurrent loading job is submitted, it is assigned a job ID number, which is displayed on the GSQL console. The user can use this job ID to refer to the job, for a status update, to abort the job, or to re-start the job. These operations are described later in this section.

Options

`-noprint`

By default, the command will print several several lines of status information while the loading is running.
If the -noprint option is included, the job will run omit the progress and summary details, but it will still display the job id and the location of the log file.

Example of minimal output when -noprint option is used

```
Kick off the following job:
  JobName: load_videoE, jobid: gsql_demo_m1.1525091090494
  Loading log: '/usr/local/tigergraph/logs/restpp/restpp_loader_logs/gsql_
```

`-dryrun`

If -dryrun is used, the system will read the data files and process the data as instructed by the job, but will NOT load any data into the graph. This option can be a useful diagnostic tool.

`-n [i], j`

The `-n` option limits the loading job to processing only a range of lines of each input data file. 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.

`filevar` **list**

The optional USING clause may contain a list of file variables. Each file variable may optionally be assigned a *filepath_string* , obeying the same format as in the CREATE LOADING JOB. This list of file variables determines which parts of a loading job are run and what data files are used.

1. When a loading job is compiled, it generates one RESTPP endpoint for each *filevar and filepath_string* . As a consequence, a loading job can be run in parts.

When RUN LOADING JOB is executed, only those endpoints whose *filevar* or file identifier (" `__GSQL_FILENAME_n__"`) is mentioned in the USING clause will be used. However, if the USING clause is omitted, then the entire loading job will be run.

2. If a *filepath_string* is given, it overrides the *filepath_string* defined in the loading job. If a particular *filevar* is not assigned a *filepath_string* either in the loading job or in the RUN LOADING JOB statement, then an error is reported and the job exits.

`CONCURRENCY`

The CONCURRENCY parameter sets the maximum number of concurrent requests that the loading job may send to the GPE. The default is 256.

`BATCH_SIZE`

The BATCH_SIZE parameter sets the number of data lines included in each concurrent request sent to the GPE. The default is 1024.

Running Loading Jobs as REST Requests

Another way to run a loading job is to submit an HTTP request to the **POST /ddl/<graph_name>** endpoint of the REST++ server. Since the REST++ server has more direct access to the graph processing engine, this can execute more quickly than a RUN LOADING JOB statement in GSQL.

When a CREATE LOADING JOB block is executed, the GSQL system creates one REST endpoint for each file source. Therefore, one REST request can invoke loading for one file source at a time. Running an entire loading job may take more than one REST request.

The Linux curl command is a handy way to make HTTP requests. If the data size is small, it can be included directly in the command line by using the -d flag with a data string:

Curl/REST++ syntax for loading using the POST /ddl endpoint

```
curl -X POST -d "<data_string>" "http://<server_ip>:9000/ddl/<graph_name>?
```

If the data size is large, it is better to reference the data filename, using the --data-binary flag:

Curl/REST++ syntax for loading using the POST /ddl endpoint

```
curl -X POST --data-binary @<data_filename> "http://<server_ip>:9000/ddl/<
```

<filepath> should be replaced with either a file variable (from a DEFINE FILENAME statement) or a position-based file identifier ("__GSQL_FILENAME_n__") for an explicit filepath_string.

For more information, about sending REST++ requests, see the RESTPP API User Guide .

Example : The code block below shows three equivalent commands for the same loading job. The first uses the gsql command RUN JOB. The second uses the Linux curl command to support a HTTP request, placing the parameter values in the URL's query string. T he third gives the parameter values through the curl command's data payload -d option.

REST++ ddl loading examples

```
# Case 1: Using GSQL
GSQL -g gsql_demo RUN LOADING JOB load_cf USING FILENAME="../cf/data/cf_da

# Case 2: Using REST++ Request with data in a file, where file1 is one of
curl -X POST --data-binary @data/cf_data.csv "http://localhost:9000/ddl/gs

# Case 3: Using REST++ Request with data inline, where file1 is one of the
curl -X POST -d
"id2,id1\nid2,id3\nid3,id1\nid3,id4\nid5,id1\nid5,id2\nid5,id4" "http://lo
```

Inspecting and Managing Loading Jobs

Starting with v2.0, there are now commands to checking loading job status, to abort a loading job and to restart a loading job.

Job ID and Status

When a loading job starts, the GSQL server assigns it a job id and displays it for the user to see. The job id format is typically the name of the loading job, followed by the machine alias, following by a code number, e.g., `gsql_demo_m1.1525091090494`

Example of SHOW LOADING STATUS output

```
Kick off the following job, i.e.
  JobName: load_test1, jobid: demo_graph_m1.1523663024967
  Loading log: '/home/tigergraph/tigergraph/logs/restpp/restpp_loader_logs

Job "demo_graph_m1.1523663024967" loading status

[RUNNING] m1 ( Finished: 3 / Total: 4 )
  [LOADING] /data/output/company.data
  [=============                          ]  20%, 200 kl/s
  [LOADED]
  +-------------------------------------------------------------------+
  |               FILENAME |   LOADED LINES |   AVG SPEED |   DURATION|
  | /data/output/movie.dat |            100 |     100 l/s |     1.00 s|
  |/data/output/person.dat |            100 |     100 l/s |     1.00 s|
  | /data/output/roles.dat |            200 |     200 l/s |     1.00 s|
  +-------------------------------------------------------------------+
[RUNNING] m2 ( Finished: 1 / Total: 2 )
  [LOADING] /data/output/company.data
  [=========================              ]  60%, 200 kl/s
  [LOADED]
  +-------------------------------------------------------------------+
  |               FILENAME |   LOADED LINES |   AVG SPEED |   DURATION|
  | /data/output/movie.dat |            100 |     100 l/s |     1.00 s|
  +-------------------------------------------------------------------+
```

By default, an active loading job will display periodic updates of its progress. There are two ways to inhibit these automatic output displays:

1. Run the loading job with the -noprint option.
2. After the loading job has started, enter CTRL+C. This will abort the output display process, but the loading job will continue.

SHOW LOADING STATUS

The command SHOW LOADING JOB shows the current status of either a specified loading job or all current jobs:

SHOW LOADING JOB syntax

```
SHOW LOADING STATUS job_id|ALL
```

The display format is the same as that displayed during the periodic progress updates of the RUN LOADING JOB command. If you do not know the job id, but you know the job name and possibly the machine, then the ALL option is a handy way to see a list of active job ids.

ABORT LOADING JOB

The command ABORT LOADING JOB aborts either a specified load job or all active loading jobs:

ABORT LOADING JOB syntax

```
ABORT LOADING JOB job_id|ALL
```

The output will show a summary of aborted loading jobs.

ABORT LOADING JOB example

```
gsql -g demo_graph "abort loading job all"

Job "demo_graph_m1.1519111662589" loading status
[ABORT_SUCCESS] m1
[SUMMARY] Finished: 0 / Total: 2
  +-------------------------------------------------------------------------
  |                  FILENAME |   LOADED LINES |   AVG SPEED  |   DURATION
  | /home/tigergraph/data.csv |       23901701 |     174 kl/s |   136.83 s
  |/home/tigergraph/data1.csv |              0 |        0 l/s |     0.00 s
  +-------------------------------------------------------------------------

Job "demo_graph_m2.1519111662615" loading status
[ABORT_SUCCESS] m2
[SUMMARY] Finished: 0 / Total: 2
  +-------------------------------------------------------------------------
  |                  FILENAME |   LOADED LINES |   AVG  SPEED |   DURATION
  | /home/tigergraph/data.csv |       23860559 |     175 kl/s |   136.23 s
  |/home/tigergraph/data1.csv |              0 |        0 l/s |     0.00 s
  +-------------------------------------------------------------------------
```

RESUME LOADING JOB

The command RESUME LOADING JOB will restart a previously-run job which ended for some reason before completion.

RESUME LOADING JOB syntax

```
RESUME LOADING JOB job_id
```

If the job is finished, this command will do nothing. The RESUME command should pick up where the previous run ended; that is, it should not load the same data twice.

RESUME LOADING JOB example

```
gsql -g demo_graph "RESUME LOADING JOB demo_graph_m1.1519111662589"
[RESUME_SUCCESS] m1
[MESSAGE] The current job got resummed
```

Verifying and Debugging a Loading Job

Every loading job creates a log file. When the job starts, it will display the location of the log file. Typically, the file is located at

<TigerGraph.root.dir>/logs/restpp/restpp_loader_logs/<graph_name>/<job_id>.log

This file contains the following information which most users will find useful:

- A list of all the parameter and option settings for the loading job
- A copy of the status information that is printed
- Statistics report on the number of lines successfully read and parsed

The statistics report include how many objects of each type is created, and how many lines are invalid due to different reasons. This report also shows which lines cause the errors. Here is the list of statistics shown in the report. There are two types of statistics. One is file level (the number of lines), and the other is data object level (the number of objects). If an file level error occurs, e.g., a line does not have enough columns, this line of data is skipped for all LOAD statements in this loading job. If an object level error or failed condition occurs, only the corresponding object is not created, i.e., all other objects in the same loading job are still created if no object level error or failed condition for each corresponding object.

File level statistics	Explanation
Valid lines	The number of valid lines in the source file
Reject lines	The number of lines which are rejected by reject_line_rules
Invalid Json format	The number of lines with invalid JSON format
Not enough token	The number of lines with missing column(s)
Oversize token	The number of lines with oversize token(s). Please increase "OutputTokenBufferSize" in the `tigergraph/dev/gdk/gsql/config` file.

Object level statistics	Explanation

Valid Object	The number of objects which have been loaded successfully
No ID found	The number of objects in which PRIMARY_ID is empty
Invalid Attributes	The number of invalid objects caused by wrong data format for the attribute type
Invalid primary id	The number of invalid objects caused by wrong data format for the PRIMARY_ID type
	The number of invalid objects caused by the

Note that failing a WHERE clause is not necessarily a bad result. If the user's intent for the WHERE clause is to select only certain lines, then it is natural for some lines to pass and some lines to fail.

Below is an example.

```
CREATE VERTEX movie (PRIMARY_ID id UINT, title STRING, country STRING COMP
CREATE DIRECTED EDGE sequel_of (FROM movie, TO movie)
CREATE GRAPH movie_graph(*)
CREATE LOADING JOB load_movie FOR GRAPH movie_graph{
  DEFINE FILENAME f
  LOAD f TO VERTEX movie VALUES ($0, $1, $2, $3) WHERE to_int($3) < 2000;
}
RUN LOADING JOB load_movie USING f="movie.dat"
```

movie.dat

```
0,abc,USA,-1990
1,abc,CHN,1990
2,abc,CHN,1990
3,abc,FRA,2015
4,abc,FRA,2005
5,abc,USA,1990
6,abc,1990
```

The above loading job and data generate the following report

load_output.log (tail)

```
--------------------Statistics------------------------------
Valid lines:             6
Reject lines:            0
Invalid Json format:     0
Not enough token:        1 [ERROR] (e.g. 7)
Oversize token:          0

Vertex:                  movie
Valid Object:            3
No ID found:             0
Invalid Attributes:      1 [ERROR] (e.g. 1:year)
Invalid primary id:      0
Incorrect fixed
binary length:           0
Passed condition lines:  4
Failed condition lines:  2 (e.g. 4,5)
```

There are a total of 7 data lines. The report shows that

- Six of the lines are valid data lines
- One line (Line 7) does not have enough tokens.

Of the 6 valid lines,

- Three of the 6 valid lines generate valid movie vertices.
- One line has an invalid attribute (Line 1: year)
- Two lines (Lines 4 and 5) do not pass the WHERE clause.

Appendix

Keywords & 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 a user-defined identifier, for the name of a vertex type, edge type, graph, or attribute.
A separate list of reserved keywords exists for the Query language here.
The compiler will reject the use of a Reserved Word as a user-defined identifier.

```
_SUBSTRING          ABORT               ACCESS              ADD
ADMIN               AFTER               ALL                 ALLOCATE
ALTER               ANALYZE             AND                 ANY
APPROX_COUNT        ARCHIVE             ARE                 ARRANGE
ARRAY               AS                  ASC                 ASENSITIVE
ASYMMETRIC          AT                  ATOMIC              ATTRIBUTE
AUTHORIZATION       AV                  AVG                 BAG
BEFORE              BEGIN               BETWEEN             BIGINT
BINARY              BINSTORAGE          BLOB                BOOL
BOOLEAN             BOTH                BUCKET              BUCKETS
BY                  BYTEARRAY           CACHE               CALL
CALLED              CASCADE             CASCADED            CASE
CAST                CAT                 CD                  CHANGE
CHAR                CHARACTER           CHARARRAY           CHECK
CLEAR               CLOB                CLOSE               CLUSTER
CLUSTERED           CLUSTERSTATUS       COGROUP             COLLATE
COLLECTION          COLUMN              COLUMNS             COMMIT
COMPACT             COMPACTIONS         COMPRESS            COMPUTE
CONCAT              CONCATENATE                             CONF
CONNECT             CONST               CONSTRAINT          CONTINUE
COPYFROMLOCAL       COPYTOLOCAL         CORRESPONDING       COUNT
CP                  CREATE              CROSS               CUBE
CURRENT             CURRENT_DATE        CURRENT_PATH        CURRENT_ROLE
CURRENT_TIME        CURRENT_TIMESTAMP   CURRENT_USER        CURSOR
CYCLE               DATA                DATA_SOURCE         DATABASE
DATABASES           DATE                DATETIME            DAY
DBPROPERTIES        DD                  DEALLOCATE          DEC
DECIMAL             DECLARE             DECRYPT             DEFAULT
DEFERRED            DEFINE              DEFINED             DELETE
DELIMITED           DEPENDENCY          DEREF               DESC
DESCRIBE            DETERMINISTIC       DIFF                DIRECTED
DIRECTORIES         DIRECTORY           DISABLE             DISCONNECT
DISTINCT            DISTRIBUTE          DISTRIBUTED         DM
DO                  DOUBLE              DROP                DRYRUN
DU                  DUMP                DYNAMIC             EACH
EDGE                ELEMENT             ELEM_TYPE           ELSE
ELSEIF              EMPTY               ENABLE              END
EOL                 ESCAPE              ESCAPED             EVAL
EXCEPT              EXCHANGE            EXCLUSIVE           EXEC
EXECUTE             EXISTS              EXIT                EXPLAIN
EXPORT              EXTENDED            EXTERN              EXTERNAL
FALSE               FETCH               FIELDS              FILE
FILEFORMAT          FILENAME            FILTER              FIRST
FIXED_BINARY        FLATTEN             FLATTEN_JSON_ARRAY  FLOAT
FOLLOWING           FOR                 FOREACH             FOREIGN
FORMAT              FORMATTED           FREE                FROM
FULL                FUNCTION            FUNCTIONS           GENERATE
GET                 GLOBAL              GPATH               GPATH_QUERY
```

GQL	GQUERY	GRANT	GRAPH
GROUP	GROUPING	GSHELL	HANDLER
HARD	HASH_PARTITION	HAVING	HEADER
HELP	HOLD	HOLD_DDLTIME	HOST_GRAPH
HOUR	ICON	IDENTIFIED	IDENTITY
IDXPROPERTIES	IF	IGNORE	ILLUSTRATE
IMMEDIATE	IMPORT	IN	INCREMENTAL
INDEX	INDEXES	INDICATOR	INIT
INNER	INOUT	INPATH	INPUT
INPUTDRIVER	INPUTFORMAT	INPUT_LINE_FILTER	INSENSITIVE
INSERT	INSTALL	INT	INT16
INT32	INT32_T	INT64_T	INT8
INTEGER	INTERPRET	INTERSECT	INTERVAL
INTO	INT_LIST	INT_SET	IS
ITEMS	ITERATE	JAR	JOB
JOIN	JSON	KAFKA	KEY
KEYS	KEY_TYPE	KILL	LANGUAGE
LARGE	LATERAL	LEADING	LEAVE
LEFT	LESS	LIKE	LIMIT
LINES	LISTLOAD	LOADING	LOCAL
LOCALTIME	LOCALTIMESTAMP		LOCK
LOCKS	LOGICAL	LONG	LOOP
LS	MACRO	MAP	MAPJOIN
MATCH	MATCHES	MATERIALIZED	MAX
MEMBER	MERGE	METHOD	MIN
MINUS	MINUTE	MKDIR	MODIFIES
MODULE	MONTH	MSCK	MULTISET
MV	NATIONAL	NATURAL	NCHAR
NCLOB	NEW	NO	NOPRINT
NONE	NOSCAN	NOT	NO_DROP
NULL	NUMERIC	OF	OFFLINE
OFFLINE2ONLINE	OLD	ON	ONLINE_POST
ONLY	ONSCHEMA	OPEN	OPTIMIZE
OPTION	OR	ORDER	OUT
OUTER	OUTPUT	OUTPUTDRIVER	OUTPUTFORMAT
OVER	OVERLAPS	OVERWRITE	OWNER
PARALLEL	PARAMETER	PARTIALSCAN	PARTITION
PARTITIONED	PARTITIONS	PASSWORD	PERCENT
PIG	PIGDUMP	PIGSTORAGE	PLUS
PRECEDING	PRECISION	PREPARE	PRESERVE
PRETTY	PRIMARY	PRIMARY_ID	PRINCIPALS
PROCEDURE	PROTECTION	PROXY	PURGE
PWD	QUERY	QUIT	QUOTE
RANGE	RANGE_PARTITION	READ	READONLY
READS	REAL	REBUILD	RECOMPILE
RECORDREADER	RECORDWRITER	RECURSIVE	REDUCE
REF	REFERENCES	REFERENCING	REFRESH
REGEXP	REGISTER	RELEASE	RENAME
REPAIR	REPEAT	REPLACE	RESIGNAL

```
REPLACE                 REPLAY                  REPLICA                 REQUIRE
RESTRICT                RESULT                  RESUME                  RETURN
RETURNS                 REVERSE_EDGE            REVOKE                  REWRITE
RIGHT                   RLIKE                   RM                      RMF
ROLE                    ROLES                   ROLLBACK                ROLLUP
ROW                     ROWS                    RUN                     S3
SAMPLE                  SAVEPOINT               SCHEMA                  SCHEMAS
SCHEMA_CHANGE           SCOPE                   SCROLL                  SEARCH
SECOND                  SECONDARY_ID            SECRET                  SELECT
SEMI                    SENSITIVE               SEPARATOR               SERDE
SERDEPROPERTIES         SERVER                  SESSION_USER            SET
SETS                    SHARED                  SHIP                    SHOW
SHOW_DATABASE           SIGNAL                  SIMILAR                 SIZE
SKEWED                  SMALLINT                SOME                    SORT
SORTED                  SPECIFIC                SPECIFICTYPE            SPLIT
SQL                     SQLEXCEPTION            SQLSTATE                SQLWARNING
SSL                     START                   START_ID                STATIC
STATISTICS              STATUS                  STATS                   STDERR
STDIN                   STDOUT                  STORE                   STORED
STREAM                  STREAMTABLE             STRING                  STRING_LIST
STRING_SET              STRUCT                  SUBMULTISET             SUM
SYMMETRIC               SYSTEM                  SYSTEM_USER             SYS.FILE_NAME
SYS.INTERNAL_ID         TABLE                   TABLES                  TABLESAMPLE
TBLPROPERTIES           TEMPORARY               TEMP_TABLE              TERMINATED
TEXTLOADER              THEN                    THROUGH                 TIME
TIMESTAMP               TIMEZONE_HOUR           TIMEZONE_MINUTE         TINYINT
TO                      TOKEN                   TOKENIZE                TOKEN_LEN
TOUCH                   TO_FLOAT                TO_INT                  TRAILING
TRANSACTION             TRANSACTIONS            TRANSFORM               TRANSLATION
TREAT                   TRIGGER                 TRUE                    TRUNCATE
TUPLE                   TYPE                    TYPEDEF                 UDF_PARTITION
UINT                    UINT16                  UINT32                  UINT32_T
UINT64_T                UINT8                   UINT8_T                 UINT_SET
UNARCHIVE               UNBOUNDED               UNDIRECTED              UNDO
UNION                   UNIONTYPE               UNIQUE                  UNIQUEJOIN
UNKNOWN                 UNLOCK                  UNNEST                  UNSET
UNSIGNED                UNTIL                   UPDATE                  UPSERT
URI                     USE                     USER                    USING
UTC                     UTCTIMESTAMP            VAL                     VALUE
VALUES                  VALUE_TYPE              VARCHAR                 VARYING
VECTOR                  VERSION                 VERTEX                  VIEW
VOID                    WHEN                    WHENEVER                WHERE
WHILE                   WINDOW                  WITH                    WITHIN
WITHOUT                 YEAR                    CURRENT_DEFAULT_TRANSFORM_GROUP
CURRENT_TRANSFORM_GROUP_FOR_TYPE                INT32_INT32_KV_LIST
UINT32_UDT_KV_LIST                              UINT32_UINT32_KV_LIST
```

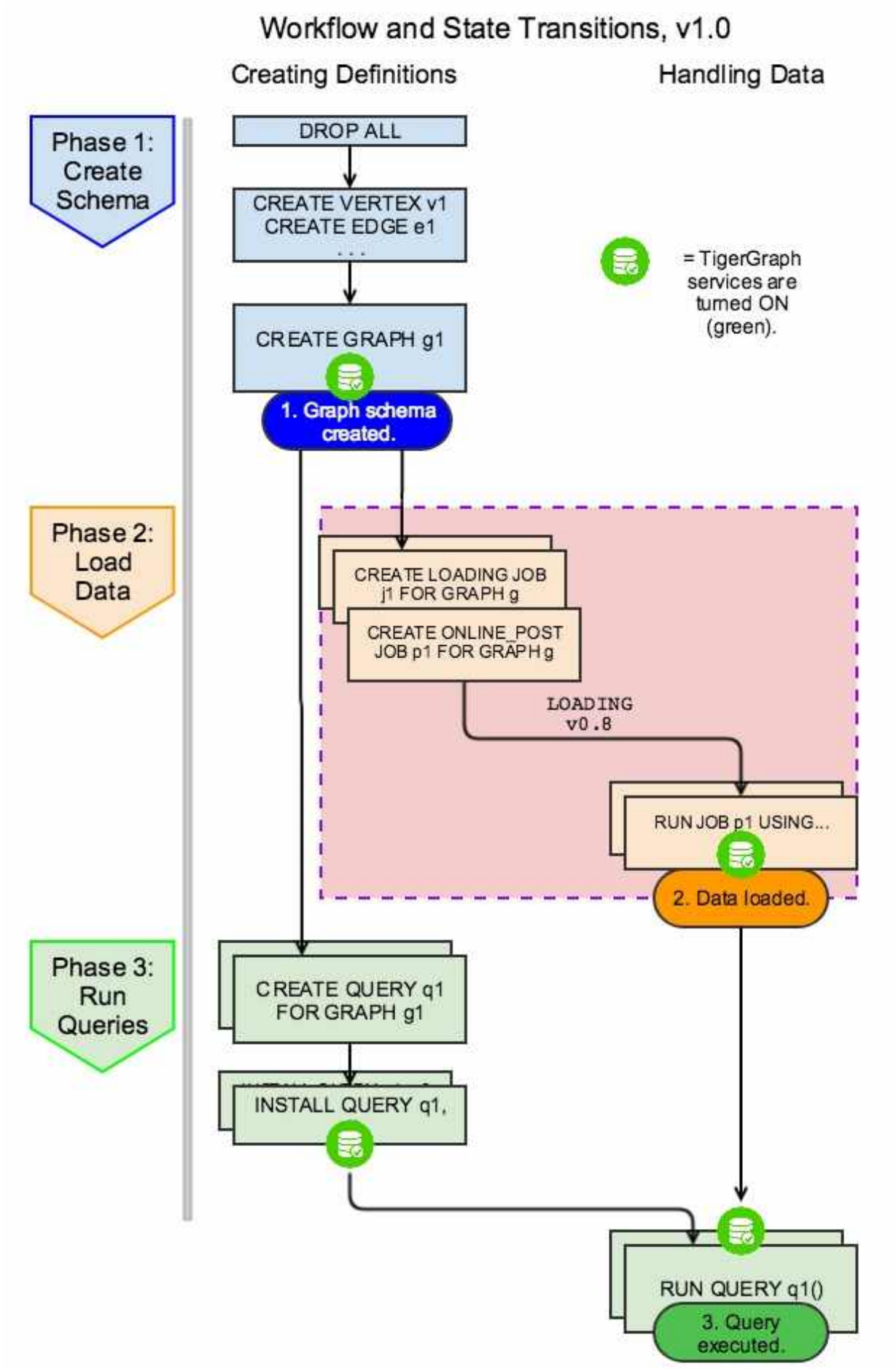

Workflow and State Transitions, v1.0
Creating Definitions
Handling Data
Phase 1: Create Schema
DROP ALL
CREATE VERTEX v1
CREATE EDGE e1
...
CREATE GRAPH g1
1. Graph schema created.
= TigerGraph services are turned ON (green).
Phase 2: Load Data
CREATE LOADING JOB j1 FOR GRAPH g
CREATE ONLINE_POST JOB 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.

Figure B1: Complete GSQL Workflow

Part 2 - Querying

Introduction

The GSQL ® Query Language is a language for the exploration and analysis of large scale graphs. The high-level language makes it easy to perform powerful graph traversal queries in the TigerGraph system. By combining features familiar to database users and programmers with highly expressive new capabilities, the GSQL query language offers both easy authoring and powerful execution. A GSQL query contains one or more SELECT statements, where each SELECT statement describes a traversal over a set of vertices and edges in the graph or describes a selection of a subset of vertices. By combining multiple SELECT statements, the user can map out query patterns to answer a virtually unlimited set of real-life data questions.

This document focuses on the formal specification for the GSQL Query Language. It includes example queries which demonstrate the language, each of which works on one of the following six graphs:**workNet, socialNet, friendNet, computerNet, minimalNet,** and **investmentNet** . Their schemas are shown below. Appendix D lists the full command and data files to create and load these graphs with small sets of data (~10 to 20 vertices). The data sets are small so that you can understand the result of each query example. The tarball file **gsql_ref_examples_2.0.tar.gz** contains all of the graph schemas, data files, and queries. Schemas for Example Graphs

66KB gsql_ref_examples_2.0.tar.gz

gsql_ref_examples_2.0.tar.gz

Graph Schema: socialNet

```
CREATE VERTEX person(PRIMARY_ID personId UINT, id STRING, gender STRING) W
CREATE UNDIRECTED EDGE friend(FROM person, TO person)
CREATE VERTEX post(PRIMARY_ID postId UINT, subject STRING, postTime DATETI
CREATE DIRECTED EDGE posted(FROM person, TO post)
CREATE DIRECTED EDGE liked(FROM person, TO post, actionTime DATETIME)
```

Graph Schema: workNet

```
CREATE VERTEX person(PRIMARY_ID personId STRING, id STRING, locationId STR
CREATE VERTEX company(PRIMARY_ID clientId STRING, id STRING, country STRIN
CREATE UNDIRECTED EDGE worksFor(FROM person, TO company, startYear INT, st
```

Graph Schema: friendNet

```
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)
```

Graph Schema: computerNet

```
CREATE VERTEX computer(PRIMARY_ID compID STRING, id STRING)
CREATE DIRECTED EDGE connected(FROM computer, TO computer, connectionSpeed
```

Graph Schema: minimalNet

```
CREATE VERTEX testV(PRIMARY_ID id STRING)
CREATE UNDIRECTED EDGE testE(FROM testV, TO testV)
```

Graph Schema: investmentNet

```
TYPEDEF TUPLE < age UINT (4), mothersName STRING(20) > SECRET_INFO
CREATE VERTEX person(PRIMARY_ID personId STRING, portfolio MAP<STRING, DOU
CREATE VERTEX stockOrder(PRIMARY_ID orderId STRING, ticker STRING, orderSi
CREATE UNDIRECTED EDGE makeOrder(FROM person, TO stockOrder, orderTime DAT
```

CREATE/INTERPRET/INSTALL/RUN QUERY

Query Action Privileges

Users with querywriter role or greater (architect, admin, and superuser) can create, install and drop queries.

Any user with 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] QUERY name "(" [parameterList] ")"
               FOR GRAPH name
               [RETURNS "("  baseType | accumType ")"]
               [API "(" stringLiteral ")"]
               [SYNTAX syntaxName]
               "{" queryBody "}"

interpretQuery := INTERPRET QUERY "(" ")"
               FOR GRAPH name
               [SYNTAX syntaxName]
               "{" queryBody "}"

parameterValueList := parameterValue ["," parameterValue]*
parameterValue := parameterConstant
                | "[" parameterValue ["," parameterValue]* "]"  // BAG or
                | "(" stringLiteral, stringLiteral ")"        // a generic
parameterConstant := numeric | stringLiteral | TRUE | FALSE
parameterList := parameterType name ["=" constant]
                 ["," parameterType name ["=" 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 |name ["," name
runQuery := RUN QUERY [runOptions] name "(" parameterValueList ")"

showQuery := SHOW QUERY name
dropQuery := DROP QUERY ( "*" | ALL | name ["," name]* )
```

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 Statement

```
createQuery := CREATE [OR REPLACE] QUERY name "(" [parameterList] ")"
               FOR GRAPH name
               [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.

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 which traverse a large portion of the cluster. Not all GSQL query language features are supported in DISTRIBUTED mode. For details, see the separate document: Distributed Query Mode.

> ⚠ OR REPLACE is deprecated
>
> 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. 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.

Typedefs allow the programmer to define custom types for use within the body. The declarations support 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)
  API ("v2") SYNTAX v1
 {
  # 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.

Parameter Types	• any baseType (except EDGE or JSONOBJECT): INT, UINT, FLOAT, DOUBLE, STRING, BOOL, STRING, VERTEX, JSONARRAY • SET<baseType>, BAG<baseType> • Exception: EDGE and JSONOBJECT type are not supported, either as a primitive parameter or as part of a complex type.
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)	"v1" (deprecated) or "v2" (default)
SYNTAX	v1 (default) or v2 (pattern matching)

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 assignStmt 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
               | 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
            | 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 queryBo
        ControlFlow queryBodyStmt2.2 # ControlFlow inside ControlFlow insi
            other queryBodyStmt2.2.1;
            other queryBodyStmt2.2.2;
        END;
        DML queryBodyStmt2.3     # DML inside ControlFlow inside top-level
            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 DMLSubStmt
        ControlFlow DMLSubStmt3.2 # ControlFlow inside DML inside top leve
            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 Chapter 6: "Declaration and Assignment Statements"
vSetVarDeclStmt	Vertex Set Variable Declaration Statement	See Chapter 6: "Declaration and Assignment

		Statements"
gAccumAssignStmt	Global Accumulator Assignment Statement	See Chapter 6: "Declaration and Assignment Statements"
gAccumAccumStmt	Global Accumulator Accumulation Statement	See Chapter 6: "Declaration and Assignment Statements"
funcCallStmt	Functional Call or Query Call Statement	See Chapter 6: "Declaration and Assignment Statements"
selectStmt	SELECT Statement	See Chapter 7: "SELECT Statement"
queryBodyCaseStmt	query-body CASE statement	See Chapter 8: "Control Flow Statements"
queryBodyIfStmt	query-body IF statement	See Chapter 8: "Control Flow Statements"
queryBodyWhileStmt	query-body WHILE statement	See Chapter 8: "Control Flow Statements"
queryBodyForEachStmt	query-body FOREACH statement	See Chapter 8: "Control Flow Statements"
updateStmt	UPDATE Statement	See Chapter 9: "Data Modification Statements"
insertStmt	INSERT INTO statement	See Chapter 9: "Data Modification Statements"
queryBodyDeleteStmt	Query-body DELETE Statement	See Chapter 9: "Data Modification Statements"
printStmt	PRINT Statement	See Chapter 10: "Output Statements"
logStmt	LOG Statement	See Chapter 10: "Output Statements"
returnStmt	RETURN Statement	See Chapter 10: "Output Statements"
raiseStmt	PRINT Statement	See Chapter 11: "Exception Statements"

tryStmt	TRY Statement	See Chapter 11: "Exception Statements"

Here is a descriptive list of DML-sub-statements:

EBNF term	Common Name	Description
assignStmt	Assignment Statement	See Chapter 6: "Declaration and Assignment Statements"
funcCallStmt	Functional Call Statement	See Chapter 6: "Declaration and Assignment Statements"
gAccumAccumStmt	Global Accumulator Accumulation Statement	See Chapter 6: "Declaration and Assignment Statements"
vAccumFuncCall	Vertex-attached Accumulator Function Call Statement	See Chapter 6: "Declaration and Assignment Statements"
localVarDeclStmt	Local Variable Declaration Statement	See Chapter 7: "SELECT Statement"
insertStmt	INSERT INTO Statement	See Chapter 8: "Control Flow Statements"
DMLSubDeleteStmt	DML-sub DELETE Statement	See Chapter 9: "Data Modification Statements"
DMLSubcaseStmt	DML-sub CASE statement	See Chapter 9: "Data Modification Statements"
DMLSubIfStmt	DML-sub IF statement	See Chapter 9: "Data Modification Statements"
DMLSubForEachStmt	DML-sub FOREACH statement	See Chapter 9: "Data Modification Statements"
DMLSubWhileStmt	DML-sub WHILE statement	See Chapter 9: "Data Modification Statements"
logStmt	LOG Statement	See Chapter 10: "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 Predefined-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/inte
  INTERPRET QUERY (int a) FOR GRAPH gsql_demo {
    PRINT a;
  }
'
```

Immediate Mode: Define and Interpret

interpret-immediate-query syntax

```
interpretQuery := INTERPRET QUERY "(" ")"
                  FOR GRAPH name
                  [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:

1. The keyword CREATE is replaced with INTERPRET.
2. The query is executed immediately by the INTERPRET statement. The INSTALL and RUN statements are not used.
3. Parameters are not accepted.

Compare the example below to the example in the Create Query section:

- No query name, no parameters, no RETURN
- 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 Predefined Query

interpret-named-query syntax

```
runQuery := (RUN | INTERPRET) QUERY [runOptions] name "(" parameterValueLi
```

This syntax is like RUN query, except

1. The keyword RUN is replaced with INTERPRET.
2. Some options may not be available, such as DISTRIBUTED mode.

Example of Interpret-Only Mode for INTERPRET QUERY

```
INTERPRET QUERY createQueryEx ("Jane.Doe")
```

INSTALL QUERY

```
installQuery := INSTALL QUERY [installOptions] ( "*" | ALL | name [, name]
```

A query must be installed before it can be executed. The INSTALL QUERY command will install the queries listed:

INSTALL QUERY queryName1, queryName2, ...

It can also install all uninstalled queries, using either of the following commands:
INSTALL QUERY *
INSTALL QUERY ALL

> Note: 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.

The following options are available:

-force Option

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.

-OPTIMIZE Option

During standard installation, the user-defined queries are dynamically linked to the GSQL language code. Anytime after INSTALL QUERY has been performed, another statement, INSTALL QUERY -OPTIMIZE can be executed. 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.

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
```

-DISTRIBUTED Option

If you have a distributed database deployment, installing the query in DISTRIBUTED 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 page: Distributed Query Mode ↗ .

```
INSTALL QUERY -DISTRIBUTED query_name
```

Running a Query

Installing a query creates a REST++ endpoint. Once a query is installed, there are two ways of executing a query. One way is through the GSQL shell:
RUN QUERY query_name(*parameterValues*) .

CREATE, INSTALL, RUN example

```
CREATE QUERY RunQueryEx(INT p1, STRING p2, DOUBLE p3) FOR GRAPH testGraph{
INSTALL QUERY RunQueryEx
RUN QUERY RunQueryEx(1, "test", 3.14)
```

Query output size limitation

There is a maximum size limit of 2GB for the result set of a SELECT block. A SELECT block is the main component of a query which searches for and returns data from the graph. If the result of the SELECT block is larger than 2GB, the system will return no data. NO error message is produced.

The query response time can be reduced by directly submitting an HTTP request to the REST++ server: send a GET request to "`http://server_ip:9000/query/graphname/queryname`". If the REST++ server is local, then server_ip is `localhost`. The query parameter values are either included directly in the query string of the HTTP request's URL or supplied using a data payload.

Starting with TigerGraph v1.2, the graph name is now part of the GET /query URL.

The current version does not support concurrent installation and running of queries. Other concurrent graph operations will be delayed until the installation finishes.

The following two curl commands are each equivalent to the RUN QUERY command above. The first gives the parameter values in the query string in a URL. This example illustrates the simple format for primitive data types such as INT, DOUBLE,

and STRING. The second gives the parameter values through the curl command's data payload -d option.

Running a query via HTTP request

```
curl -X GET "http://localhost:9000/query/testGraph/RunQueryEx?p1=1&p2=test
curl -d @RunQueryExPara.dat -X GET "http://localhost:9000/query/testGraph/
```

where RunQueryExPara.dat has the exact string as the query string in the first URL.

RunQueryExPara.dat

```
p1=1&p2=test&p3=3.14
```

To see a list of the parameter names and types for the user-installed GSQL queries, run the following REST++ request:

`curl -X GET "http://localhost:9000/endpoints?dynamic=true"`

By using the data payload option, the user can avoid using a long and complex URL. In fact, to call the same query but with different parameters, only the data payload file contents need to be changed; the HTTP request can be the same. The file loader loads the entire file, appends multiple lines into one, and uses the resulting string as the URL query string. If both a query string and a data payload are given (which we strongly discourage), both are included, where the URL query string's parameter values overwrite the values given in the data payload.

Complex Type Parameter Passing

This subsection describes how to format the complex type parameter values when executing a query by RUN QUERY or curl command. More details about all parameter types are described in Section "Query Parameter Types"

Parameter type	RUN QUERY	Query string for GET /query HTTP Request
SET or BAG of primitives	Square brackets enclose the collection of values.	Assign multiple values to the same parameter name.

	Example: a set p1 of integers: **[1,5,10]**	Example: a set p1 of integers: **p1=1&p1=5&p1=10**
VERTEX<type>	If the vertex type is specified in the query definition, then the vertex argument is simply **vertex_id** Example: vertex type is person and desired id is person2. **"person2"**	**parameterName=vertex_id** Example: vertex type is person and desired id is person2. **vp=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)** Example: a vertex va w ith id="person1" and type="person: **("person1","person")**	**parameterName=vertex_id &parameterName.type=vertex_type** Example: parameter vertex va when type="person" and id="person1": **va=person1&va.type=person**
SET or BAG of VERTEX<type>	Same as a SET or BAG of primitives, where the primitive type is vertex_id. Example: **["person3", "person4"]**	Same as a SET or BAG of primitives, where the primitive type is vertex_id. Example: **vp=person3&vp=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. Example: **[("person1","person") , ("11","post")]**	The SET or BAG must be treated like an array, specifying the first, second, etc. elements with indices [0], [1], etc. The example below provides the same input arguments as the RUN QUERY example to the left. **vp[0]=person1&vp[0].type=person&vp[1]=11&vp[1].type=post**

⚠ When square brackets are used in a curl URL, the -g option or escape characters must be adopted. If the parameters are given by data payload (either by file or data payload

string), the -g option is not needed and escape characters should not be used.

Below are examples.

Running a query via HTTP request - complex parameter type

```
# 1. SET or BAG
CREATE QUERY RunQueryEx2(SET<INT> p1) FOR GRAPH testGraph{ .... }
# To run this query (either RUN QUERY or curl):
GSQL > RUN QUERY RunQueryEx2([1,5,10])
curl -X GET "http://localhost:9000/query/testGraph/RunQueryEx2?p1=1&p1=5&p


# 2. VERTEX.
# First parameter is any vertex; second parameter must be a person type.
CREATE QUERY printOneVertex(VERTEX va, VERTEX<person> vp) FOR GRAPH social
  PRINT va, vp;
}
# To run this query:
GSQL > RUN QUERY printOneVertex(("person1","person"),"person2")   # 1st pa
curl -X GET 'http://localhost:9000/query/socialNet/printOneVertex?va=perso


# 3. BAG or SET of VERTEX, any type
CREATE QUERY printOneBagVertices(BAG<VERTEX> va) FOR GRAPH socialNet {
  PRINT va;
}
# To run this query:
GSQL > RUN QUERY printOneBagVertices([("person1","person"), ("11","post")]
curl -X GET 'http://localhost:9000/query/socialNet/printOneBagVertices?va\
curl -g -X GET 'http://localhost:9000/query/socialNet/printOneBagVertices?


# 4. BAG or SET of VERTEX, pre-specified type
CREATE QUERY printOneSetVertices(SET<VERTEX<person>> vp) FOR GRAPH socialN
  PRINT vp;
}
# To run this query:
GSQL > RUN QUERY printOneSetVertices(["person3", "person4"])  # [vertex_1
curl -X GET 'http://localhost:9000/query/socialNet/printOneSetVertices?vp=
```

Payload Size Limit

This data payload option can accept a file up to 128MB by default. To increase this limit to xxx MB, use the following command:

```
gadmin --set nginx.client_max_body_size xxx -f
```

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.

For more detailed information about REST++ endpoints and requests, see the ***RESTPP API User Guide*** .

The following options are available when running a query:

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 underscores
curl -X GET 'http://localhost:9000/query/graphname/queryname?__GQUERY__USI
```

Diagnose -d Option

The diagnose option can be turned on in order to produce a diagnostic monitoring log, which contains the processing time of each SELECT block . To turn on the monitoring log, use the -d option in shell mode or `__GQUERY__monitor=true` in the query string of an HTTP request.

```
GSQL > RUN QUERY -d test()

## In a curl URL call.  Note the use of both single and double underscores
curl -X GET 'http://localhost:9000/query/graphname/queryname?__GQUERY__mon
```

The path of the generated log file will be shown as a part of output message. An example log is shown below:

```
Query Block Start (#6) start at 11:52:06.415284
Query Block Start (#6) end   at 11:52:06.415745 (takes 0.000442 s)

Query test takes totally 0.001 s (restpp's pre/post process time not inclu
---------------- Summary (sort by total_time desc) ----------------

Query Block Start on Line 6
-----------------------------------------------------------
  total iterations count      : 1
  avg   iterations stats      : 0.000442s
  max   iterations stats      : 0.000442s
  min   iterations stats      : 0.000442s
  total activated vertex count : 2
  max   activated vertex count : 2
  min   activated vertex count : 2
```

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:

"version" field	value
api	A string specifying the output API version. Values are specified as follows: • "v1": Output API used in TigerGraph platform v0.8 through v1.0. If the output does not have a "version" field, the JSON format is presumed to be v1. • "v2": Output API introduced in TigerGraph platform v1.1. This is the latest API. (Note: for backward compatibility, TigerGraph platforms which support the v2 output api can be configured to produce either v1 or v2 output.)
edition	A string indicating which edition of the product. Current possible values are "developer" and "enterprise".
schema	An 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 (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 following REST response misspells the name of the endpoint

For backward compatibility, TigerGraph platforms whose principal output API is v2 can also produce output with API v1.

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 the Chapter "Output Statements".

Changing the Default Output API

GET echo/ Request and Response

```
curl -X GET "http://localhost:9000/eco"
```

and generates the following output:

```
{"
  version": {"api":"v2","schema":0},
  "error": true,
  "message": "Endpoint is not found from url = /eco, please use GET /endpo
  "code": "REST-1000"
}
```

The following GSQL statement can be used to set the JSON output API configuration.

```
SET json_api = <version_string>
```

Currently, the legal values for <version_string> are "v1" and "v2". 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.

SHOW QUERY

To show the GSQL text of a query, run "SHOW QUERY *query_name* ". Additionally, the "ls" GSQL command lists all created queries and identifies which queries have been installed.

As of v2.3, the query_name argument can now use * or ? wildcards from Linux globbing, or it can be a regular expression, when preceded by -r. See SHOW: View Parts of the Catalog

DROP QUERY

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 Q if another query R 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.

Distributed Query Mode

Distributed Query Mode

In a distributed graph (where the data are spread across multiple machines), the default execution plan is as follows:

- One machine will be selected as the execution hub, regardless of the number or distribution of starting point vertices.
- All the computation work for the query will take place at the execution hub. The vertex and edge data from other machines will be copied to the hub machine for processing.

TigerGraph Enterprise Edition offers a Distributed Query mode which provides a more optimized execution plan for queries which are likely to start at several machines and continue their traversal across several machines.

- A set of machines representing one full copy of the entire graph will participate in the query. If the cluster has a replication factor of 2 (so there are two copies of each piece of data), then half the machines will participate.
- The query executes in parallel across all the machines which have source vertex data for a given hop in the query. That is, each SELECT statement defines a 1-hop traversal from a set of source vertices to a set of target vertices. Unlike the default mode where all the needed data are brought to one machine, in Distributed Query mode, the computation moves across the cluster, following the traversal pattern of the query.
- The output results will be gathered at one machine.

To invoke Distributed Query mode, simply insert the keyword "DISTRIBUTED" before "QUERY" in a query definition:

```
createQuery := CREATE [OR REPLACE] [DISTRIBUTED] QUERY name "(" [paramete
               [RETURNS "("  baseType | accumType ")"]
               [API "(" stringLiteral ")"]
               "{" [typedefs] [declStmts] [declExceptStmts] queryBodyStmts
```

Guidelines for Selecting Distributed Query Mode

The basic trade-off between distributed query mode and default mode is greater parallelism for the given query vs. using more system resources, which reduces the potential for concurrency with other operations. Each machine has a certain number of workers available for concurrent execution of queries. A query in default mode uses only one worker out of the whole system. (This one worker will have multiple threads for processing edge traversals in parallel.) However, a query in distributed mode uses one query worker *per machine.* This means this query can run faster, but it leaves fewer workers for other queries running concurrently.

We suggest the following guidelines for deciding whether to use default mode or distributed mode.

1. Queries with one or a few starting point vertices and which take only a few hops → default mode is better.
2. Queries which start at a very large set of starting point vertices and which traverse many hops → distributed mode is better.
   For example, algorithms which either compute a value for every vertex or one value for the entire graph should use Distributed Mode. This includes PageRank, Centrality, and Connected Component algorithms.
3. For applications where the same query (same logic but with different input parameters) will be run many times in production, the application designer can simply try both modes during development and chose the one which works better for their use case and data.

Supported and Unsupported Features

Currently, Distributed Query mode cannot be used for all queries. Please note the limitations carefully. In most cases, the GSQL parser and compiler will report an error if you try to write a Distributed Query using an unsupported feature.

> Support was added for many features in TigerGraph 2.2. The table below has changed significantly between v2.1 and v2.2.

The following GSQL features are not supported in Distributed Query mode:

Feature	Not Supported	Supported as of v2.2 (1)
General	User-defined exceptions	Data update to the graph Access to target vertex's values in ACCUM Query calling a distributed query
Statement Types	LOADACCUM	FOREACH, WHILE, UPDATE, INSERT, DELETE
SELECT clauses	SAMPLE clause exact count for LIMIT clause (2)	
Data types	LIST, SET, BAG JSONOBJECT, JSONARRAY ArrayAccum	SET<> parameter, GroupByAccum
Operations and Operators		Any data update to the graph, including assignment statements to vertex attributes
vertex and edge functions	.neighbors(), .neighborAttribute(), .edgeAttribute() isDirected()	.outdegree()
accumulator and collection functions	reallocate()	size(), get(), top(), pop(), update(), remove(), removeAll(), clear(), contains), containskey(), resize()
Other functions	selectVertex(), to_vertex(), to_vertex_set(), COALESCE(), EVALUATE() sum(), count(), min(), max(), avg()	

(1) Items in the Supported column are listed only for clarity, so you can compare to the Unsupported column. If a feature which is supported in non-distributed queries is not mentioned in either column, then it is supported in Distributed Query mode .

(2) If the query contains "LIMIT N", and if the number of GPEs working on this query is G, then the output size will be N +/- (G-1). In a conventional cluster configuration, there is one GPE per machine. For example, if N=10 and the graph is distributed across 4 machines, then the output size will be between 7 and 13, inclusive.

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: (1) the query's input parameters, (2) the vertices, edges, and their attributes which are encountered when traversing the graph, or (3) variables defined within the query that are used 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 "_" or

type := baseType | name | accumType | STRING COMPRESS

baseType := INT
          | UINT
          | FLOAT
          | DOUBLE
          | STRING
          | BOOL
          | VERTEX ["<" name ">"]
          | EDGE
          | JSONOBJECT
          | JSONARRAY
          | DATETIME

filePath := name | stringLiteral

typedef := TYPEDEF TUPLE "<" tupleType ">" name

tupleType := (baseType name) | (name baseType) ["," (baseType name) | (nam

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 a `name`. It can be a sequence of letters, digits, or underscores ("_"). Other punctuation characters are not supported. The initial character can only be 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 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 baseType	• 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	STRING COMPRESS	• more compact storage of STRING, when there is a limited number of different values
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.

BNF

```
baseType := INT
          | UINT
          | FLOAT
          | DOUBLE
          | STRING
          | BOOL
          | VERTEX ["<" name ">"]
          | 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 the ***GSQL Language Reference, Part 1***.

type	default value
INT, UINT, FLOAT, DOUBLE (see note below)	0
BOOL	false
STRING	""
DATETIME	1970-01-01 00:00:00
VERTEX	"Unknown"
EDGE	No edge: {}
JSONOBJECT	An empty object: {}
JSONARRAY	An empty 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 and EDGE

VERTEX and EDGE are the two types of objects which form a graph. A query parameter or variable can be declared as either of these two types. In additional, the schema for the graph defines specific vertex and edge types (e.g., CREATE VERTEX *person*). The parameter or variable type can be restricted by giving the vertex/edge type in angle brackets < > after the keyword VERTEX/EDGE. A VERTEX

or EDGE variable declared without a specifier is called a *generic* type. Below are examples of generic and typed vertex and edge variable declarations:

Examples of generic and typed VERTEX and EDGE declarations

```
VERTEX anyVertex;
VERTEX<person> owner;
EDGE anyEdge;
EDGE<friendship> friendEdge;
```

Vertex and Edge Attribute Types

The following table map s vertex or edge attribute types in the Data Definition Language (DDL) to GSQL query language types. Accumulators are introduced in Section "Accumulators".

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* >
DATETIME	DATETIME

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 at www.json.org ↗. A JSONOBJECT instance's external representation (as input and output) is a string, starting and ending with curly braces "{" and "}", which enclose an unordered list of ***string:value*** pairs. A JSONARRAY is represented as a string, starting and ending with square brackets "[" and "]", 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 entitled "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 baseType 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 "<" tupleType ">" name

tupleType := (baseType name) | (name baseType) "," (baseType name) | (nar
```

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 which has been defined in the graph schema does not need to be re-defined in a query.

The graph schema investmentNet contains two complex attributes:

- user-defined tuple SECRET_INFO, which is used for the secret_info attribute in the person vertex.
- portfolio MAP<STRING, DOUBLE > attribute, also in the person vertex.

investmentNet schema

```
TYPEDEF TUPLE <age UINT (4), mothersName STRING(20) > SECRET_INFO
CREATE VERTEX person(PRIMARY_ID personId STRING, portfolio MAP<STRING, DOU
CREATE VERTEX stockOrder(PRIMARY_ID orderId STRING, ticker STRING, orderSi
CREATE UNDIRECTED EDGE makeOrder(FROM person, TO stockOrder, orderTime DAT
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 key:value type as the original portfolio map. (The "Accumulators" chapter has more information about accumulators.) 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;       # alrea
  TYPEDEF TUPLE <STRING ticker, FLOAT price, DATETIME orderTime> ORDER_REC

  SetAccum<SECRET_INFO> @@info;
  ListAccum<ORDER_RECORD> @@orderRecords;
  MapAccum<STRING, DOUBLE> @@portf;       # corresponds to MAP<STRING, DOU

  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.

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 (introduced in Section "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 PRINT statement, which is introduced in), 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 fron
            @@strCompressList += s.interestList; # use the dictionary fron

  @@strCompressList2 += @@strCompressList;  # @@strCompressList2 gets the
  @@strCompressList2 += "xyz";   # "xyz" is not in the dictionary, so stor

  @@strCompressSet2 += @@strCompressSet;
  @@strCompressSet2 += @@strSet;

  @@strList2 += @@strCompressList;  # string compress integer values are c
  @@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 or JSONOBJECT). A parameter can also be a SET or BAG which uses base type (except EDGE 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 (See Section "Accumulator" for more details).

> 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.

BNF

```
parameterType := baseType
               | [ 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 "@"name ["=" constant][, "@"name ["=" constant]
               | "@"name ["=" constant][, "@"name ["=" constant]]* accumTy
               | [STATIC] accumType "@@"name ["=" constant][, "@@"name ["=
               | [STATIC] "@@"name ["=" constant][, "@@"name ["=" constant


accumType := "SumAccum" "<" ( INT | FLOAT | DOUBLE | STRING ) ">"
           | "MaxAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
           | "MinAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
           | "AvgAccum"
           | "OrAccum"
           | "AndAccum"
           | "BitwiseOrAccum"
           | "BitwiseAndAccum"
           | "ListAccum" "<" type ">"
           | "SetAccum"  "<" elementType ">"
           | "BagAccum"  "<" elementType ">"
           | "MapAccum"  "<" elementType "," ( baseType | accumType | name
           | "HeapAccum" "<" name ">" "(" (integer | name) "," name [ASC |
           | "GroupByAccum" "<" elementType name ["," elementType name]* )
           | "ArrayAccum" "<" name ">"

elementType := baseType | name | STRING COMPRESS

gAccumAccumStmt := "@@"name "+=" 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

All accumulator variables must be declared at the beginning of a query, immediately after any typedefs, and before any other type of statement. The scope of the accumulator variables is the entire query.

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 "@"name ["=" constant][, "@"name ["=" constant]
               | "@"name ["=" constant][, "@"name ["=" constant]]* accumTy
               | [STATIC] accumType "@@"name ["=" constant][, "@@"name ["=
               | [STATIC] "@@"name ["=" constant][, "@@"name ["=" constant
```

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 it no longer needed, it should be cleared to minimize the 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 ) ">"
           | "MaxAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
           | "MinAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
           | "AvgAccum"
           | "OrAccum"
           | "AndAccum"
           | "BitwiseOrAccum"
           | "BitwiseAndAccum"
           | "ListAccum" "<" type ">"
           | "SetAccum"  "<" elementType ">"
           | "BagAccum"  "<" elementType ">"
           | "MapAccum"  "<" elementType "," (baseType | accumType | name
           | "HeapAccum" "<" name ">" "(" (integer | name) "," name [ASC |
           | "GroupByAccum" "<" elementType name ["," elementType name]* ,
           | "ArrayAccum" "<" name ">"

elementType := baseType | name | STRING COMPRESS

gAccumAccumStmt := "@@"name "+=" 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< *typ e* > (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 [,*	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> provide 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 alphabetically largest id.

Tuple data types are treated as hierarchical structures, where the first field used for ordering is the leftmost one.

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"
    }
  ]}]
}
```

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 a 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 which 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 (IN
  BitwiseOrAccum  @@bwOrAccumVar;  # default value = 64-bits of 0 = 0 (INT

  # 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 which 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 List
  # 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 combinations
  # 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"], ["Strin
```

```
  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);  // las
  @@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 workNe

  # Each person has an LIST<INT> of skills and a LIST<STRING COMPRESS> of
  # This function copies their lists into ListAccums, and then udpates the
  # 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 which 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 which 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 which 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> tupleName;
...
HeapAccum<tupleName>(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 which 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 n
  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": [
```


```
GroupByAccum<INT a, STRING b, MaxAccum<INT> maxa, ListAccum<ListAccum<INT>
```


```
    "lastName": "Stark",
    "score": 100
```

```
      },
      {
        "firstName": "Bruce",
        "lastName": "Banner",
        "score": 95
      },
      {
        "firstName": "Jean",
        "lastName": "Summers",
        "score": 95
      }
    ]},
    {"@@topTestResults.size()": 0}
  ]
}
```

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 *key_value2*... **)**	BOOL	Accessor	Returns true/false if the accumulator 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 {

  ## declaration, first two primitive type are group by keys; the rest acc
  GroupByAccum<INT a, STRING b, MaxAccum<INT> maxa, ListAccum<ListAccum<IN
  GroupByAccum<STRING gender, MapAccum<VERTEX<person>, DATETIME> m> @@grou
  # nested GroupByAccum
  GroupByAccum<INT a, MaxAccum<INT> maxa, GroupByAccum<INT a, MaxAccum<INT

  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.

  ## 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 inte
```

```
    },
    {
      "@@group2.get(Male).m": {
        "person3": 1263618953,
        "person1": 1263209520,
        "person8": 1263180365,
        "person7": 1263295325,
        "person6": 1263468185
      },
      "@@group2": [
        {
          "gender": "Male",
          "m": {
```

```
ListAccum<ListAccum<INT>>
ListAccum<ListAccum<ListAccum<INT>>>
ListAccum<SetAccum<INT>> # illegal
```

```
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
```

```
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, Functions, and Expressions

An *expression* is a combination of fixed values, variables, operators, function calls, and groupings which 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_

mathOperator := "*" | "/" | "%" | "+" | "-" | "<<" | ">>" | "&" | "|"

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 :=   ["@@"]name
        | name "." "type"
        | name "." ["@"]name
        | name "." "@"name ["\'"]
        | name "." name "." name "(" [argList] ")"
        | name "." name "(" [argList] ")" [ "."FILTER "(" condition ")" ]
        | name ["<" type ["," type"]* ">"] "(" [argList] ")"
        | name "." "@"name ("." name "(" [argList] ")")+ ["." name]
        | "@@"name ("." name "(" [argList] ")")+ ["." name]
        | COALESCE "(" [argList] ")"
        | ( COUNT | ISEMPTY | MAX | MIN | AVG | SUM ) "(" setBagExpr ")"
        | expr mathOperator expr
        | "-" expr
        | "(" expr ")"
        | "(" argList "->" argList ")"   // key value pair for MapAccum
        | "[" argList "]"                // a list
        | constant
        | setBagExpr
        | name "(" argList ")"

setBagExpr := ["@@"]name
         | name "." ["@"]name
         | name "." "@"name ("." name "(" [argList] ")")+
         | name "." name "(" [argList] ")" [ "."FILTER "(" condition ")"
         | "@@"name ("." name "(" [argList] ")")+
         | setBagExpr (UNION | INTERSECT | MINUS) setBagExpr
         | "(" argList ")"
         | "(" setBagExpr ")"


argList := expr ["," expr]*
```

Constants

```
constant := numeric | stringLiteral | TRUE | FALSE | GSQL_UINT_MAX | GSQL_
```

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 which 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 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 x
    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 which 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 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)?
```

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

The LIKE operator is used for string pattern matching. 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. LIKE may be used only in WHERE clauses. Additionally, *string_pattern*supports the following wildcard and other symbols, in order to express a pattern:

character or syntax		meaning
		matches zero or more characters.

%		*Example* : `%abc%` matches any string which contains the sequence "abc".
_ (underscore)		matches any single character. *Example* : `_abc_e` matches any 6-character string where the 2nd to 4th characters are "abc" and the last character is "e".
[charlist]		match any character in charlist. charlist is a concatenated character set, with no separators. *Example* : `[Tiger]` matches either T, i, g, e, or r.
[^charlist]		matches any character NOT in charlist. *Example* : `[^qxz]` matches any character other than q, x, or z.
[!charlist]		matches any character NOT in charlist.
special syntax within charlist	α-β	matches a character in the range from α to β. A charlist can have multiple ranges. *Example* : `[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 charlist	\\	matches the character \

Mathematical Functions

There are a number of built-in functions which act on either an accumulator, a base type, or vertex variable. The accumulator function calls are discussed in detail in the "Accumulators" section.

Below is a list of built-in functions which act on either INT, FLOAT, or DOUBLE value(s).

function name and parameters(NUM means INT, FLOAT, or DOUBLE)	description	return type
abs (NUM *num*)	Returns the absolute value of *num*	Same as parameter type
sqrt (NUM *num*)	Returns the square root of *num*	FLOAT
pow (NUM *base* , NUM *exp*)	Returns *base* *exp*	If base and exp are both INT → INT; Otherwise → FLOAT
acos (NUM *num*)	arc cosine	FLOAT
asin (NUM *num*)	arc sine	FLOAT
atan (NUM *num*)	arc tangent	FLOAT
atan2 (NUM *y* , NUM *x*)	arc tangent of *y* / *x*	FLOAT
ceil (NUM *num*)	rounds upward	INT
cos (NUM *num*)	cosine	FLOAT
cosh (NUM *num*)	hyperbolic cosine	FLOAT
exp (NUM *num*)	*base-e* exponential	FLOAT
floor (NUM *num*)	rounds downward	INT

fmod (NUM *numer* , NUM *denom*)	floating-point remainder of *numer* / *denom*	FLOAT
ldexp (NUM *x* , NUM *exp*)	*x* * 2 *exp*	FLOAT
log (NUM *num*)	natural logarithm	FLOAT
log10 (NUM *num*)	common (base-10) logarithm	FLOAT
sin (NUM *num*)	sine	FLOAT
sinh (NUM *num*)	hyperbolic sine	FLOAT
tan (NUM *num*)	tangent	FLOAT
tanh (NUM *num*)	hyperbolic tangent	FLOAT
to_string (NUM *num*)	Converts *num* to a STRING value	STRING
float_to_int (FLOAT *num*)	Converts *num* to a INT value by truncating the floating part	INT
str_to_int (STRING *str*)	Converts *str* to a INT value. If str is a floating number, the floating part is	INT

Type Conversion Functions

function name and parameters	description	return type
to_string (NUM *num*)	Converts *num* to a STRING	STRING
float_to_int (FLOAT *num*)	Converts *num* to a INT value by truncating the floating point	INT
str_to_int (STRING *str*)	Concerts *str* to a INT value. If str is a floating number, the floating part is truncated. If str is not a numerical value, returns 0.	INT

String Functions

The following built-in functions are provided for text processing. Note that these functions do not modify the input parameter. They each return a new string.

function name and parameters	description	return type
lower(STRING *str*)	Converts *str* to all lowercase letters	STRING
upper(STRING *str*)	Converts *str* to all uppercase letters	STRING
trim([[**LEADING** | **TRAILING** | **BOTH**] [STRING *removal_char*] **FROM**] STRING *str*)	Trims* characters from the leading and/or trailing ends of *str*	STRING

- In the syntax for trim(), the words in **bold** (**LEADING, TRAILING, BOTH,** and **FROM**) are keywords which should appear exactly as shown.
- STRING is just an indicator of the datatype; it is not an explicit keyword.
- The trim() function have the following options:
  - 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 removed from both ends.
  - *removal_char* is a single character. The function will remove consecutive instances of *removal_char* , until it encounters a different character. If *removal_char* is not specified, then trim() removes whitespace (spaces, tabs, and newlines).

```
CREATE QUERY stringFuncEx() FOR GRAPH minimalNet {
    #Example strings
    string a = "  Abc   ";
    string b = "aa ABC aaa";
    string c = " a A   ";

    PRINT lower(a);                         # prints "  abc   "
    PRINT upper(b);                         # prints "AA ABCC AAA"
    PRINT trim(a);                          # prints "Abc"
    PRINT trim(BOTH a);                     # prints "Abc"
    PRINT trim(LEADING c);                  # prints "a A   "
    PRINT trim(TRAILING "a" FROM b);        # prints "aa ABC "

    #You can combine functions for more convenient calling:
    PRINT trim(BOTH trim(BOTH " " FROM c) FROM b);
    # prints "BC"
}
```

Notes about the trim() function:

Datetime Functions

The following functions convert from/to DATETIME to/from other types.

function name and parameters	description	return type
to_datetime (STRING *str*)	Converts *str* to a DATETIME value	DATETIME
epoch_to_datetime (INT *int_value*)	Converts *int_value* to a DATETIME value by epoch time conversion	DATETIME
datetime_to_epoch (DATETIME *date*)	Converts *date* to epoch time.	INT

The following function converts a DATETIME value into a string format specified by the user:

function name and parameters	description	return type
datetime_format(DATETIME *date*[, STRING *str*])	Prints *date* as the *str* indicates. The following specifiers may be used as the format of *str* . The "%" character is required before the format specifier characters. If *str* is not given, "%Y-%m-%d %H:%M:%S" is used. Specifier: • %Y: Year, numeric, four digits • %S: Seconds (0..59) • %m: Month, numeric (1..12) • %M: Minutes, numeric (0..59) • %H: Hour, numeric (0..23) • %d: Day of the month, numeric (1..31)	STRING

datetime_format example

```
# Show all posts's post time
CREATE QUERY allPostTime() FOR GRAPH socialNet api("v2") {
  start = {post.*};
  #PRINT datetime_format(start.postTime, "a message was posted at %H:%M:%S
  PRINT start[datetime_format(start.postTime, "a message was posted at %H:
}
```

The followings are other functions related to DATETIME :

function name and parameters	description	return type
now()	Returns the current time in DATETIME type.	DATETIME

year(DATETIME *date*)	Extracts the year of *date* .	INT
month(DATETIME *date*)	Extracts the month of *date.*	INT
day(DATETIME *date*)	Extracts the day of month of *date* .	INT
hour(DATETIME *date*)	Extracts the hour of *date* .	INT
minute(DATETIME *date*)	Extracts the minute of *date* .	INT
second(DATETIME *date*)	Extracts the second of *date* .	INT
datetime_add(DATETIME *date*, INTERVAL *int_value time_unit*)	INTERVAL is a keyword; *time_unit* is one of the keywords YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND. The function returns the DATETIME value which is *int_value* units later than *date* . For example, datetime_add(now() , INTERVAL 1 MONTH) returns a DATETIME value which is 1 month from now.	DATETIME
datetime_sub(DATETIME *date*, INTERVAL *int_value time_unit*)	Same as datetime_add, except that the returned value is *int_value* units **earlier** than *date* .	DATETIME
datetime_diff(DATETIME*date1* , DATETIME *date2)*	Returns the difference in seconds of these two DATETIME values: (*date1* -	INT

JSONOBJECT and JSONARRAY Functions

JSONOBJECT and JSONARRAY are base types, meaning they can be used as a parameter type, an element type for most accumulators, or a return type. This enables the input and output of complex, customized data structures. For input and output, a string representation of the JSON is used. Hence, the GSQL query

language offers several functions to convert a formatted string into JSON and then to search and access the components of a JSON structure.

Data Conversion Functions

The following parsing functions convert a string into a JSONOBJECT or a JSONARRAY:

function name	description	return type
parse_json_object(STRING *str*)	Converts *str* into a JSON object	JSONOBJECT
parse_json_array(STRING *str*)	Converts *str* into a JSON array	JSONARRAY

Both functions generate a run-time error if the input string cannot be converted into a JSON object or a JSON array. 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 (in double quotes "), a number, a boolean (**true** or **false**), or a JSONOBJECT or JSONARRAY. Each key of a key:value pair must be a string in double quotes.

See examples below.

parse_json_object and parse_json_array example

```
CREATE QUERY jsonEx (STRING strA, STRING strB) FOR GRAPH minimalNet {
  JSONARRAY jsonA;
  JSONOBJECT jsonO;

  jsonA = parse_json_array( strA );
  jsonO = parse_json_object( strB );

  PRINT jsonA, jsonO;
}
```

jsonEx.json Result

```
GSQL > RUN QUERY jsonEx("[123]","{\"abc\":123}")
or curl -X GET 'http://localhost:9000/query/jsonEx?strA=\[123\]&strB=\{"ab
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{
    "jsonA": [123],
    "jsonO": {"abc": 123}
  }]
}
GSQL > RUN QUERY jsonEx("{123}","{\"123\":\"123\"}")
Runtime Error: {123} cannot be parsed as a json array.
```

Data Access Methods

JSONOBJECT and JSONARRAY are object classes, each class supporting a set of data access methods, using dot notation:

jsonVariable.functionName(parameter_list)

The following methods (class functions) can act on a JSONOBJECT variable:

method name	description	return type
containsKey(STRING *keyStr*)	Returns a boolean value indicating whether the JSON object contains the key *keyStr*.	BOOL
getInt(STRING *keyStr*)	Returns the numeric value associated with key *keyStr* as an INT.	INT
getDouble (STRING *keyS tr*)	Returns the numeric value associated with key *keyStr* as a DOUBLE.	DOUBLE
getString (STRING *keyS tr*)	Returns the string value associated with key *keyStr*.	STRING

getBool (STRING *keyS tr*)	Returns the bool value associated with key *keyStr* .	BOOL
getJsonObject (STRING *keyS tr*)	Returns the JSONOBJECT associated with key *keyStr* .	JSONOBJECT
getJsonArray (STRING *keySt r*)	Returns the JSONARRAY associated with key *keyStr* .	JSONARRAY

The above getType(STRING *keyStr*) function generates a run-time error if

1. The key *keyStr* doesn't exist, or
2. The function's return type is different than the stored value type. See the next note about numeric data.
3. Pure JSON stores "numbers" without distinguishing between INT and DOUBLE, but for TigerGraph, if the input value is all digits, it will be stored as INT. Other numeric values are stored as DOUBLE. The getDouble function can read an INT and return its equivalent DOUBLE value, but it is an error to call getINT for a DOUBLE value.

The following methods can act on a JSONARRAY variable:

method name	description	return type
size()	Returns the size of this array.	INT
getInt(INT *idx*)	Returns the numeric value at position *idx* as an INT.	INT
getDouble(INT *idx*)	Returns the numeric value at position *idx* as a DOUBLE.	DOUBLE
getString(INT *idx*)	Returns the string value at position *idx* .	STRING
getBool(INT *idx*)	Returns the bool value at position *idx* .	BOOL
getJsonObject(INT *idx*)	Returns the JSONOBJECT value at position *idx* .	JSONOBJECT

Similar to the methods of JSONOBJECT, the above getType(INT *idx*) function generates a run-time error if

1. *idx* is out of bounds, or
2. The function's return type is different than the stored value type. See the next note about numeric data.
3. Pure JSON stores "numbers" without distinguishing between INT and DOUBLE, but for TigerGraph, if the input value is all digits, it will be stored as INT. Other numeric values are stored as DOUBLE. The getDouble function can read an INT and return its equivalent DOUBLE value, but it is an error to call getINT for a DOUBLE value.

Below is an example of using these functions and methods :

JSONOBJECT and JSONARRAY function example

```
CREATE QUERY jsonEx2 () FOR GRAPH minimalNet {

  JSONOBJECT json0, json02;
  JSONARRAY jsonA, jsonA2;
  STRING str, str2;

  str = "{\"int\":1, \"double\":3.0, \"string\":\"xyz\", \"bool\":true, \'
  str2 = "[\"xyz\", 123, false, 5.0]";
  json0 = parse_json_object( str ) ;
  jsonA = parse_json_array( str2 ) ;


  json02 = json0.getJsonObject("obj");
  jsonA2 = json0.getJsonArray("arr");

  PRINT json0;
  PRINT json0.getBool("bool"), json0.getJsonObject("obj"), json0.getJsonAr
}
```

jsonEx2.json Result

```
GSQL > RUN QUERY jsonEx2()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [ {"jsonO": { "arr": [ "xyz", 123, true ],
      "bool": true,
      "string": "xyz",
      "double": 3,
      "obj": {"obj": {"bool": false}},
      "int": 1
    }},
    {
      "jsonO.getBool(bool)": true,
      "jsonA.getDouble(3)": 5,
      "jsonA.getDouble(1)": 123,
      "jsonO.getJsonObject(obj)": {"obj": {"bool": false}},
      "jsonO2.getJsonObject(obj)": {"bool": false},
      "jsonO.getJsonArray(arr)": [ "xyz", 123, true ],
      "jsonA2.getString(0)": "xyz"
    }
  ]
}
```

Vertex, Edge, and Accumulator Functions and 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.

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 == "post
    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"
    }]}
  ]
}
```

Vertex Functions

Below is a list of built-in functions that can be accessed by vertex aliases, using the dot operator:

Syntax for vertex functions

```
vertex_alias.function_name(parameter)[.FILTER(condition)]
```

Currently, these functions are only available for vertex aliases (defined in the FROM clause); vertex variables do not have these functions.

Note that in order to calculate outdegree by edge type, the graph schema must be defined such that vertices keep track of their edge types using WITH STATS="OUTDEGREE_BY_EDGETYPE" (however, "OUTDEGREE_BY_EDGETYPE" is now the default STATS option).

function name	description	return type
outdegree ([STRING *edgeType*])	Returns the recent number* [see footnote] of outgoing or undirected edges connected to the vertex. If the optional STRING argument *edgeType*is given, then count only edges of the given edgeType.	INT
neighbors ([STRING *edgeType*])	Returns the set of ids for the vertices which are out-neighbors or undirected neighbors of the vertex. If the optional STRING argument *edgeType* is given, then include only those neighbors reachable by edges of the given *edgeType* .	BagAccum<VERTEX>
neighborAttribute (STRING *edgeType,STRING targetVertexType, STRING attribute*)	From the given vertex, traverses the given *edgeType* to the given *targetVertexType* , and return the set of values for the given *attribute* . *edgeType* can only be string literal.	BagAccum<attributeType>
edgeAttribute (STRING *edgeType, STRINGattribute*)	From the given vertex, traverses the given *edgeType* , and return the set of values for the given edge *attribute* . *edgeType*can only be string literal.	BagAccum<attributeType>

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.

Vertex function examples

```
CREATE QUERY vertexFunctionExample(vertex<person> m1) FOR GRAPH socialNet

  SetAccum<Vertex> @neighborSet;
  SetAccum<Vertex> @neighborSet2;
  SetAccum<DATETIME> @attr1;
  BagAccum<DATETIME> @attr2;

  int deg1, deg2, deg3, deg4;

  S = {m1};
  S2 = SELECT S
       FROM S - (posted:e) -> post:t
       ACCUM deg1 = S.outdegree(),
             deg2 = S.outdegree("posted"),
             deg3 = S.outdegree(e.type),  # same as deg2
             STRING str = "posted",
             deg4 = S.outdegree(str);     # same as deg2
  PRINT deg1, deg2, deg3, deg4;

  S3 = SELECT S
       FROM S:s
       POST-ACCUM s.@neighborSet += s.neighbors(),
                  s.@neighborSet2 += s.neighbors("posted"),
                  s.@attr1 += s.neighborAttribute("posted", "post", "post
                  s.@attr2 += s.edgeAttribute("liked", "actionTime");
  PRINT S3;
}
```

vertexFunctionExample Result

```
GSQL > RUN QUERY vertexFunctionExample("person5")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "deg4": 2,
      "deg2": 2,
      "deg3": 2,
      "deg1": 5
    },
    {"S3": [{
      "v_id": "person5",
      "attributes": {
        "@attr2": [1263330725],
        "@attr1": [
          1297054971,
          1296694941
        ],
        "gender": "Female",
        "@neighborSet": [
          "6",
          "11",
          "4",
          "person7",
          "person4"
        ],
        "id": "person5",
        "@neighborSet2": [
          "4",
          "11"
        ]
      },
      "v_type": "person"
    }]}
  ]
}
```

FILTER

The optional .FILTER(condition) clause offers an additional filter for selecting which elements are added to the output set of the neighbor, neighborAttribute and edgeAttribute functions. The condition is evaluated for each element . If the condition is true, the element is added to the output set; if false, it is not. An example is shown below:

Example: vertex functions with optional filter

```
CREATE QUERY filterEx (SET<STRING> pIds, INT yr) FOR GRAPH workNet api("v2

  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.startYe
         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.@allCour
}
```

Results in filterEx.json

```
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
  }]
}
```

Edge Functions

Below are the built-in functions that can be accessed by edge aliases, using the dot operator. Edge functions follow the same general rules as vertex functions (see above).

function name	description	return type

isDirected ()	Returns a boolean value indicating whether this edge is directed or undirected	BOOL

Accumulator Functions

This section describes functions which all to all or most accumulators. Other accumulator functions for each accumulator type are illustrated at 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.

BNF

```
setBagExpr := ["@@"] name
            | name "." ["@"] name
            | name "." "@" name ("." name "(" [argList] ")")+
            | name "." name "(" [argList] ")" [ ".".FILTER "(" condition "
            | "@@" name ("." 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 operant is a bag, the result expression is a bag. If one operant 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, 4
  @@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 u
  @@AunionBbag = @@setA UNION @@setB;  PRINT @@AunionBbag;  // (1, 2, 3, 4
                                                          // because set u
}
```

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 blacklist.

Set Membership example

```
CREATE QUERY friendsNotInblacklist (VERTEX<person> seed, SET<VERTEX<persor
  Start = {seed};
  Result = SELECT v
      FROM Start:s-(friend:e)-person:v
      WHERE v NOT IN blackList;
  PRINT Result;
}
```

Results for Query friendsNotInblacklist

```
GSQL > RUN QUERY friendsNotInblacklist("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"
  }]}]
}
```

Aggregation Functions - COUNT, SUM, MIN, MAX, AVG

The aggregation functions take a set/bag expression as its input parameter and return one value or element.

- **count()** : Returns the size (INT) of the set.

- **sum()** : Returns the sum of all elements. This is only applicable to a set/bag expression with numeric type.
- **min()** : Returns the member with minimum value. This is only applicable to a set/bag expression with numeric type.
- **max()** : Returns the member with maximum value. This is only applicable to a set/bag expression with numeric type.
- **avg()** : Returns the average of all elements. This is only applicable to a set/bag expression with numeric type. The average is INT if the element type of the set/bag expression is INT.

Aggregation function example

```
CREATE QUERY aggregateFuncEx(BAG<INT> x) FOR GRAPH minimalNet   {
  BagAccum<INT> @@t;
  @@t += -5; @@t += 2; @@t+= -1;
  PRINT max(@@t), min(@@t), avg(@@t), count(@@t), sum(@@t);
  PRINT max(x), min(x), avg(x), count(x), sum(x);
}
```

aggregateFuncEx.json Result

```
GSQL > RUN QUERY aggregateFuncEx([1,2,5])
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "sum(@@t)": -4,
      "count(@@t)": 3,
      "max(@@t)": 2,
      "avg(@@t)": -1,
      "min(@@t)": -5
    },
    {
      "avg(x)": 2,
      "count(x)": 3,
      "max(x)": 5,
      "min(x)": 1,
      "sum(x)": 8
    }
  ]
}
```

Miscellaneous Functions

SelectVertex()

SelectVertex() reads a data file which 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. 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. SelectVertex() has five parameters explained in the below table: filePath, vertexIdColumn, vertexTypeColumn, separator, and header. The rules for column separators and column headings are the same as for the GSQL Loader.

parameter name	type	description
filePath	string	The absolute file path of the input file to be read. A relative path is not supported.
vertexIdColumn	$ *num* , or $ *"column_name"* if header is true.	The vertex id column position.
vertexTypeColumn	$ *num* , *$ "column_name"* if header is true, or a vertex type	The vertex type column position or a specific vertex type.
separator	single-character string	The column separator character.
header	bool	Whether this file has a header.

One vertex set variable declaration statement can have multiple SelectVertex() function calls. However, if a declaration statement has multiple SelectVertex() calls referring to the same file, they must use the same separator and header parameters. If any row of the file contains an invalid vertex type, a run time error occurs; if any row of the file contains an nonexistent vertex id, a warning message is shown with the count of nonexistent ids.

Below is a query example using SelectVertex calls, reading from the data file selectVertexInput.csv.

selectVertexInput.csv

```
c1,c2,c3
person1,person,3
person5,person,4
person6,person,5
```

selectVertex example

```
CREATE QUERY selectVertexEx(STRING filename) FOR GRAPH socialNet {
  S = {SelectVertex(filename, $"c1", $1, ",", true),
       SelectVertex(filename, $2, post, ",", true)
  };
  PRINT S;
}
```

Result

```
GSQL > RUN QUERY selectVertexEx("/file_directory/selectVertexInput.csv")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"S": [
    {
      "v_id": "4",
      "attributes": {
        "postTime": "2011-02-07 05:02:51",
        "subject": "coffee"
      },
      "v_type": "post"
    },
    {
      "v_id": "person1",
      "attributes": {
        "gender": "Male",
        "id": "person1"
      },
      "v_type": "person"
    },
    {
      "v_id": "person5",
      "attributes": {
        "gender": "Female",
        "id": "person5"
      },
      "v_type": "person"
    },
    {
      "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"
    },
    {
      "v_id": "person6",
      "attributes": {
        "gender": "Male",
        "id": "person6"
      },
      "v_type": "person"
    }
  ]}]
}
```

to_vertex() and to_vertex_set()

to_vertex() and to_vertex_set() convert a string or a string set into a vertex or a vertex set, respectively, of a given vertex type. These two functions are useful when the vertex id(s) are obtained and only known at run-time.

⚠ Running these functions requires real-time conversion of an external id to a GSQL internal id, which is a relatively slow process. Therefore,

1. 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().
2. Calling to_vertex_set() one time is much faster than c alling to_vertex() multiple times . Use to_vertex_set() instead of to_vertex() as much as possible.

The first parameter of to_vertex() is the vertex id string. The first parameter of to_vertex_set() is a string set representing vertex ids. The second parameter of both functions is the vertex type string.

Function signatures for to_vertex() and to_vertex_set()

```
 VERTEX to_vertex(STRING id, STRING vertex_type)
SET<VERTEX> to_vertex_set(SET<STRING>, STRING vertex_type)
SET<VERTEX> to_vertex_set(BAG<STRING>, STRING vertex_type)
```

to_vertex_set can accept a bag of vertices as input, but the function will reduce the bag to a set by eliminating duplicate items.

If the vertex id or the vertex type doesn't exist, to_vertex() will have a run-time error, as shown below. However, to_vertex_set() will have a run-time error only if the vertex type doesn't exist. If one or more vertex ids are nonexistent, to_vertex_set() will display a warning message but will still run, converting all valid ids and skipping nonexistent vertex ids. If the user wants an error instead of a warning if a nonexistent id is given when converting a string set to a vertex set, the user can use to_vertex() inside a FOREACH loop, instead of to_vertex_set(). See the example below .

to_vertex() and to_vertex_set() example

```
CREATE QUERY to_vertex_setTest (SET<STRING> uids, STRING uid, STRING vtype
  SetAccum<VERTEX> @@v2, @@v3;
  SetAccum<STRING> @@strSet;
  VERTEX v;

  v = to_vertex (uid, vtype);           # to_vertex assigned to a vertex v
  PRINT v;                              # vertex variable -> only vertex i

  @@v2 += to_vertex (uid, vtype);   # to_vertex accumulated to a SetAccum<
  PRINT @@v2;                           # SetAccum of vertex -> only verte

  S2 = to_vertex_set (uids, vtype); # to_vertex_set assigned to a vertex s
  PRINT S2;                             # vertex set variable-> full detai

  @@strSet = uids;                      # Show SET<STRING> & SetAccumm<STR
  S3 = to_vertex_set(@@strSet, vtype); # Input to to_vertex_set is SetAccu
  SDIFF = S2 MINUS S3;                  # Now S2 = S3, so SDIFF2 is empty
  PRINT  SDIFF.size();

  #FOREACH vid in uids DO    # In this case non-existing ids in uids cause
  #  @@v3 += to_vertex( vid, vtype );
  #END;
  #L3 = @@v3;
  #PRINT L3;
}
```

to_vertex_set.json Results

```
GSQL > RUN QUERY to_vertex_setTest(["person1","personx","person2"], "perso
{
  "error": false,
  "message": "Runtime Warning: 1 ids are invalid person vertex ids.",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"v": "person3"},
    {"@@v2": ["person3"]},
    {"S2": [
      {
        "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": "person2",
        "attributes": {
          "interestList": ["engineering"],
          "skillSet": [ 6, 5, 3, 2 ],
          "skillList": [ 2, 3, 5, 6],
          "locationId": "chn",
          "interestSet": ["engineering"],
          "id": "person2"
        },
        "v_type": "person"
      }
    ]},
    {"SDIFF.size()": 0}
  ]
}


GSQL > RUN QUERY to_vertex_setTest(["person1","personx"], "person1", "abc"
Runtime Error: abc is not valid vertex type.
```

getvid()

The getvid(v) function returns the *internal* ID number of the given vertex v. 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.

getvid_ex.gsql

```
CREATE QUERY getvid_ex () FOR GRAPH socialNet {
  MinAccum<int> @cc_id = 0;       //each vertex's tentative component id

  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;
}
```

getvid_ex.json

```
GSQL > RUN QUERY getvid_ex()
{
  "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"
      }
    ]}]
}
```

COALESCE()

The COALESCE function evaluates each argument value in order, and returns the first value which is not NULL. This evaluation is the same as that used for IS NULL and IS NOT NULL. The COALESCE function requires all its arguments have the same data type (BOOL, INT, FLOAT, DOUBLE, STRING, or VERTEX). The only exception is that different numeric types can be used together. In this case, all values are converted into the first argument type.

coalesce function example

```
CREATE QUERY coalesceFuncEx (INT p1, DOUBLE p2) FOR GRAPH minimalNet {
  PRINT COALESCE(p1, p2, 999.5);  # p2 and the last value will be converte
}
```

coalesceFuncEx.json Results

```
GSQL > RUN QUERY coalesceFuncEx(_,_)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"coalesce(p1,p2,999.5)": 999}]
}
GSQL > RUN QUERY coalesceFuncEx(1,2)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"coalesce(p1,p2,999.5)": 1}]
}
GSQL > RUN QUERY coalesceFuncEx(_,2.5)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"coalesce(p1,p2,999.5)": 2}]
}
```

The COALESCE function is useful when multiple optional parameters are allowed, and one of them must be chosen if available. For example,

coalesce function example

```
CREATE QUERY coalesceFuncEx2 (STRING homePhone, STRING cellPhone, STRING c
  PRINT "contact number: " + COALESCE(homePhone, cellPhone, companyPhone);
  PRINT "contact number: " + COALESCE(homePhone, cellPhone, companyPhone,
}
```

⚠ The COALESCE function's parameter list should have a default value as the last argument. Otherwise, i f all values are NULL, the default value of the data type is returned.

coalesceFuncEx2.json Results

```
GSQL > RUN QUERY coalesceFuncEx2(_,_,_)
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"contact number: +coalesce(homePhone,cellPhone,companyPhone)": "conta
    {"contact number:+coalesce(homePhone,cellPhone,companyPhone,N/A)": "co
  ]
}
```

Dynamic Expressions with EVALUATE()

The function evaluate() takes a string argument and interprets it as an expression which is evaluated during run-time. This enables users to create a general purpose query instead of separate queries for each specific computation.

```
evaluate(expressionStr, typeStr)
```

The evaluate() function has two parameters: expressionStr is the expression string, and typeStr is a string literal indicating the type of expression. This function returns a value whose type is typeStr and whose value is the evaluation of expressionStr. The following rules apply:

1. evaluate() can only be used inside a SELECT block, and only inside a WHERE clause, ACCUM clause, POST-ACCUM clause, HAVING clause, or ORDER BY clause. It cannot be used in a LIMIT clause or outside a SELECT block.
2. The result type must be specified at query installation time: typeStr must be a string literal for a primitive data type, e.g., one of "int", "float", "double", "bool",

"string" (case insensitive). The default value is "bool".

3. In expressionStr, identifiers can refer only to a vertex or edge aliases, vertex-attached accumulators, global accumulators, parameters, or scalar function calls involving the above variables. The expression may not refer to local variables, global variables, or to FROM clause vertices or edges by type.
4. Any accumulators in the expression must be scalar accumulators (e.g., MaxAccum) for primitive-type data. Container accumulators (e.g., SetAccum) or scalar accumulators with non-primitive type (e.g. VERTEX, EDGE, DATETIME) are not supported. Container type attributes are not supported.
5. evaluate() cannot be nested.

The following situations generate a run-time error:

1. The expression string expressionStr cannot be compiled (unless the error is due to a non-existent vertex or edge attribute).
2. The result type of the expression does not match the parameter typeStr.

> Silent failure conditions
>
> If any of the following conditions occur, the query may continue running, but the entire clause or statement in which the evaluate() function resides will fail, without producing a run-time error message. For conditional clauses (WHERE, HAVING), a failing evaluate() clause is treated as if the condition is false. An assignment statement with a failing evaluate() will not execute, and an ORDER BY clause with a failing evaluate() will not sort.
>
> 1. The expression references a non-existent attribute of a vertex or edge alias.
> 2. The expression uses an operator for non-compatible operation. For example, 123 == "xyz".

The following example employs dynamic expressions in both the WHERE condition and the accumulator value in the POST-ACCUM clause.

Evaluate example

```
CREATE QUERY evaluateEx (STRING whereCond = "TRUE", STRING postAccumIntExp
  SetAccum<INT> @@timeSet;
  MaxAccum<INT> @latestLikeTime, @latestLikePostTime;

  S = {person.*};
  S2 = SELECT s
       FROM S:s - (liked:e) -> post:t
       WHERE evaluate(whereCond)
       ACCUM s.@latestLikeTime += datetime_to_epoch( e.actionTime ),
             s.@latestLikePostTime += datetime_to_epoch( t.postTime )
       POST-ACCUM @@timeSet += evaluate(postAccumIntExpr, "int")
       ;
  PRINT @@timeSet;
}
```

Results for Query evaluateEx

```
GSQL > RUN QUERY evaluateEx(_,_)
{
  "error": false,
  "message": "",
  "results": [{"@@timeSet": [1]}]
}

GSQL > RUN QUERY evaluateEx("s.gender==\"Male\"", "s.@latestLikePostTime")
{
  "error": false,
  "message": "",
  "results": [
    {
      "@@timeSet": [1263295325,1296752752,1297054971,1296788551]
    }
  ]
}

GSQL > RUN QUERY evaluateEx("s.gender==\"Female\"", "s.@latestLikeTime + 1
{
  "error": false,
  "message": "",
  "results": [
    {
      "@@timeSet": [1263293536,1263352566,1263330726]
    }
  ]
}


GSQL > RUN QUERY evaluateEx("xx", _)
Runtime Error: xx is undefined parameter.


GSQL > RUN QUERY evaluateEx("e.xyz", _)'   # The attribute doesn't exist,
{
  "error": false,
  "message": "",
  "results": [{"@@timeSet": []}]
}


GSQL > RUN QUERY evaluateEx("e.actionTime", _)
Runtime Error: actionTime is not a primitive type attribute.

GSQL > RUN QUERY evaluateEx("s.id", _)
Runtime Error: Expression 's.id' value type is not bool.
```

```
GSQL > RUN QUERY evaluateEx("s.gender==\"Female\"", "s.xx")   # The attrib
{
  "error": false,
  "message": "",
  "results": [{"@@timeSet": []}]
}
```

Queries as Functions

A query that has been defined (with a CREATE QUERY ... RETURNS statement) can be treated as a callable function. A query can call itself recursively.

The following limitations apply to queries calling queries:

1. Each parameter of the called query may be one of the following types:
   a. Primitives: INT, UINT, FLOAT, DOUBLE, STRING, BOOL
   b. VERTEX
   c. A Set or Bag of primitive or VERTEX elements
2. The return value may be one of the following types. See also the "Return Statement" section.
   a. Primitives: INT, UINT, FLOAT, DOUBLE, STRING, BOOL
   b. VERTEX
   c. a vertex set (e.g., the result of a SELECT statement)
   d. An accumulator of primitive types. GroupByAccum and accumulators containing tuples are not supported.
3. A query which returns a SetAccum or BagAccum may be called with a Set or Bag argument, respectively.
4. The order of definition matters. A query cannot call a query which has not yet been defined.

Subquery Example 1

```
CREATE QUERY subquery1 (VERTEX<person> m1) FOR GRAPH socialNet RETURNS(Bag
{
  Start = {m1};
  L = SELECT t
      FROM Start:s - (liked:e) - post:t;
  RETURN L;
}
CREATE QUERY mainquery1 () FOR GRAPH socialNet
{
  BagAccum<VERTEX<post>> @@testBag;
  Start = {person.*};
  Start = SELECT s FROM Start:s
          ACCUM @@testBag += subquery1(s);
  PRINT @@testBag;
}
```

User-Defined Functions

Users can define their own expression functions in C++ in <tigergraph.root.dir>/dev/gdk/gsql/src/QueryUdf/ExprFunctions.hpp. 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. Once defined, the new functions will be added into GSQL automatically next time GSQL is executed.

⚠ If a user-defined struct or a helper function needs to be defined, define it in <tigergraph.root.dir>/dev/gdk/gsql/src/QueryUdf/ExprUtil.hpp.

Here is an example:

new code in ExprFunction.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;
}
```

user defined expression function

```
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");
}
```

udfExample.json Results

```
 GSQL > RUN QUERY udfExample()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"greater_than_three(x)": true},
    {"y": false},
    {"reverse(abc)": "cba"}
  ]
}
```

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 user-defined function. Therefore, please test each new user-defined expression function after adding it. One way of testing the function is creating a new cpp file test.cpp and compiling it by
> g++ test.cpp
> ./a.out
You might need to remove the include header #include <gle/engine/cpplib/headers.hpp> in ExprFunction.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;
}
```

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 == "company
```

```
  company2 = SELECT t FROM seed:s -(worksFor)-> :t WHERE (s.id == "company

  #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 "." "@" name ("." name "(" [argList] ")")+ ["." name]
     s.@companyCount += s.@companyNames.size(),

    #expr := name "." "@" name ["\'"]
    @@totalRelationshipCount += s.@companyCount,

  FOREACH comp IN s.@companyNames DO
     #expr := "(" argList "->" argList ")"
     @@companyEmployeeRelationships += (s.id -> comp)
  END;

  PRINT employees;
  PRINT @@totalRelationshipCount;
  PRINT @@companyEmployeeRelationships;

  #expr := "@@" name ("." 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"
```

```
        }
      ]},
      {"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" }],
        "id": "person8",
        "@companyNames": ["company1"]
```


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 := "@@" name ("+=" | "=") expr
  @@a = 10;

  PRINT x, @@a;

  start = {person.*};

  employees = SELECT s FROM start:s -(worksFor)-> :t
              ACCUM #exprStmnt := name "." "@" name ("+="| "=") expr
                    s.@employers += t.id,
                    #exprStmnt := name ["<" type ["," type"]* ">"] "(" [ar
                    s.@employerInfo += companyInfo(t.country, t.id),
                    #gAccumAccumStmt := "@@" name "+=" expr
                    @@countrySet += t.country
                    #exprStmnt := name "." "@" name ["." name "(" [argLis
              POST-ACCUM s.@employerCount += s.@employers.size();

  #exprStmnt := "@@" name ["." 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" ]
```

```
        "@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": "u
        "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": "u
      "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": "u
      "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"
  },
  {
```

EBNF

```
## Declarations ##
accumDeclStmt := accumType "@"name ["=" constant][, "@"name ["=" constant]
               | "@"name ["=" constant][, "@"name ["=" constant]]* accumTy
               | [STATIC] accumType "@@"name ["=" constant][, "@@"name ["=
               | [STATIC] "@@"name ["=" constant][, "@@"name ["=" constant

baseDeclStmt    := baseType name ["=" constant][, name ["=" constant]]*

fileDeclStmt := FILE fileVar "(" filePath ")"
fileVar := name

localVarDeclStmt := baseType name "=" expr

vSetVarDeclStmt := name ["(" vertexEdgeType ")"] "=" (seedSet | simpleSet

simpleSet := name | "(" simpleSet ")" | simpleSet (UNION | INTERSECT | MIN

seedSet := "{" [seed ["," seed ]*] "}"
seed := '_'
      | ANY
      | ["@@"]name
      | name ".*"
      | "SelectVertex" selectVertParams

selectVertParams := "(" filePath "," columnId "," (columnId | name) ","
                 stringLiteral "," (TRUE | FALSE) ")" [".".FILTER "(" cond

columnId := "$" (integer | stringLiteral)

## Assignment Statements ##
assignStmt := name "=" expr
            | name "." name "=" expr
            | name "." "@"name ("+="| "=") expr

gAccumAssignStmt :=  "@@"name ("+=" | "=") expr

loadAccumStmt := "@@"name "=" "{" "LOADACCUM" loadAccumParams ["," "LOADAC

loadAccumParams := "(" filePath "," columnId "," [columnId ","]*
                stringLiteral "," (TRUE | FALSE) ")" [".".FILTER "(" condi


## Function Call Statement ##
funcCallStmt := name ["<" type ["," type"]* ">"] "(" [argList] ")"
              | "@@"name ("." name "(" [argList] ")")+

argList := expr ["," expr]*
```

Declaration Statements

There are six types of variable declarations in a GSQL query:

- Accumulator
- Global baseType variable
- Local baseType 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

```
vertexEdgeType := "_" | ANY | name | ( "(" name ["|" name]* ")" )
```

Accumulator declaration is discussed in Section 4: "Accumulators".

Global Variables

After accumulator declarations, base type variables can be declared as global variables. The scope of a global variable is from the point of declaration until the end of the query.

EBNF for global variable declaration

```
baseDeclStmt    := baseType name ["=" constant][, name ["=" constant]]*
```

A global variable can be accessed (read) anywhere in the query; however, there are restrictions on wh ere it can be updated. See the subsection below on "Assignment Statements".

Global Variable Example

```
# Assign global variable at various places
CREATE QUERY globalVariable(VERTEX<person> m1) FOR GRAPH socialNet {

  SetAccum<VERTEX<person>> @@personSet;
  SetAccum<Edge> @@edgeSet;

  # Declare global variables
  STRING gender;
  DATETIME dt;
  VERTEX v;
  VERTEX<person> vx;
  EDGE ee;

  allUser = {person.*};
  allUser = SELECT src
            FROM allUser:src - (liked:e) -> post
            ACCUM dt = e.actionTime,
                  ee = e,           # assignment does NOT take effect yet
                  @@edgeSet += ee   # so ee is null
            POST-ACCUM @@personSet += src;
  PRINT @@edgeSet;  # EMPTY because ee was frozen in the SELECT statement.
  PRINT dt;         # actionTime of the last edge e processed.

  v = m1;              # assign a vertex value to a global variable.
  gender = m1.gender;  # assign a vertex's attribute value to a global va
  PRINT v, gender;

  FOREACH m IN @@personSet DO
     vx = m;             # global variable assignment inside FOREACH takes
     gender = m.gender;  # global variable assignment inside FOREACH takes
     PRINT vx, gender;   # display the values for each iteration of the lo
  END;
}
```

globalVariable Query Result

```
GSQL > RUN QUERY globalVariable("person1")
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {"@@edgeSet": [{}]},
    {"dt": "2010-01-12 21:12:05"},
    {
      "gender": "Male",
      "v": "person1"
    },
    {
      "vx": "person3",
      "gender": "Male"
    },
    {
      "vx": "person7",
      "gender": "Male"
    },
    {
      "vx": "person1",
      "gender": "Male"
    },
    {
      "vx": "person5",
      "gender": "Female"
    },
    {
      "vx": "person6",
      "gender": "Male"
    },
    {
      "vx": "person2",
      "gender": "Female"
    },
    {
      "vx": "person8",
      "gender": "Male"
    },
    {
      "vx": "person4",
      "gender": "Female"
    }
```

```
  ]
}
```

Multiple global variables of the same type can be declared and initialized at the same line, as in the example below:

Multiple variable declaration example

```
CREATE QUERY variableDeclaration() FOR GRAPH minimalNet {
   INT a=5,b=1;
   INT c,d=10;

   MaxAccum<INT> @@max1 = 3, @@max2 = 5, @@max3;
   MaxAccum<INT> @@max4, @@max5 = 2;

   PRINT a,b,c,d;
   PRINT @@max1, @@max2, @@max3, @@max4, @@max5;
}
```

variableDeclaration.json Result

```
GSQL > RUN QUERY variableDeclaration()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [
    {
      "a": 5,
      "b": 1,
      "c": 0,
      "d": 10
    },
    {
      "@@max3": -9223372036854775808,
      "@@max2": 5,
      "@@max1": 3,
      "@@max5": 2,
      "@@max4": -9223372036854775808
    }
  ]
}
```

Local Variables

A local variable can be declared only in an ACCUM, POST-ACCUM, or UPDATE SET clause, and its scope is limited to that clause. Local variables can only be of base types (e.g. INT, FLOAT, DOUBLE, BOOL, STRING, VERTEX). A local variable must be declared and initialized together at the same statement.

EBNF for local variable declaration and initialization

```
localVarDeclStmt := baseType name "=" expr
```

Within a local variable's scope, another local variable with the same name cannot be declared at the same level. However, a new local 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 variable may only be used in source vertex statements or target vertex statements, not both.

Local Variable Example

```
# An example showing a local variable succeeded where a global variable fa
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 dt
            dtGlob = e.actionTime,           # dtGlob doesn't update 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

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* .

EBNF for Vertex Set Variable Declaration

```
vSetVarDeclStmt := name ["(" vertexEdgeType ")"] "=" (seedSet | simpleSet
## Seed Sets ##
seedSet := "{" [seed ["," seed ]*] "}"
seed := '_'
      | ANY
      | ["@@"]name
      | name ".*"
      | "SelectVertex" selectVertParams

selectVertParams := "(" filePath "," columnId "," (columnId | name) ","
                 stringLiteral "," (TRUE | FALSE) ")" [".".FILTER "(" conc

columnId := "$" (integer | stringLiteral)

simpleSet := name | "(" simpleSet ")" | simpleSet (UNION | INTERSECT | MIN
```

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 (S1) or typed (S2)
- a vertex set parameter, untyped (S3) or typed (S4)
- a global SetAccum<VERTEX> accumulator, untyped (S5) or typed (S6)
- all vertices of any type (S7, S9) or of one type (S8)
- a list of vertex ids in an external file (S10)
- copy of another vertex set (S11)
- a combination of individual vertices, vertex set parameters, or global variables (S12)
- union of vertex set variables (S13)

Seed Set Example

```
CREATE QUERY seedSetExample(VERTEX v1, VERTEX<person> v2, SET<VERTEX> v3,
  SetAccum<VERTEX> @@testSet;
  SetAccum<VERTEX<person>> @@testSet2;
  S1 = { v1 };
  S2 = { v2 };
  S3 = v3;
  S4 = v4;
  S5 = @@testSet;
  S6 = @@testSet2;
  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;
  S12 = {@@testSet, v2, v3};      # S1 is not allowed to be in {}
  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 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 comes before the base variable name: **type varName**

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 soc
  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 a 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.

> When referring to a FILE object, we always capitalize the word FILE, to distinguish it from ordinary files.

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 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.

Note that the declaration statement is invoking the FILE object constructor. The syntax for the constructor includes parentheses surrounding the filepath parameter.

Currently, the filePath must be a absolute path.

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/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 baseType 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 (expr) to define the new value of the variable.

EBNF for Assignment Statements

```
## Assignment Statement ##
assignStmt := name "=" expr                        # baseType variable, ver
            | name "." name "=" expr               # attribute of a vertex
            | name "." "@"name ("+="| "=") expr    # vertex.attached accumu

gAccumAssignStmt := "@@"name "=" expr              # global accumulator
                  | loadAccumStmt
loadAccumStmt := "@@"name "=" "{" "LOADACCUM" loadAccumParams ["," "LOADAC
```

Restrictions on Assignment Statements

In general, assignment statements can take place anywhere after the variable has been declared. However, t here 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
- Global 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 := "@@" name "=" "{" "LOADACCUM" loadAccumParams ("," "LOADA


loadAccumParams := "(" filePath "," columnId "," [columnId ","]*
                stringLiteral "," (TRUE | FALSE) ")" [".".FILTER "(" condi
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: (filePath, fieldColumn_1, ..., fieldColumn_n, separator, header), where n is the number of fields in the accumulator. 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.

⚠ Any accumulator using generic VERTEX as an element type cannot be initialized by LOADACCUM().

parameter name	type	description
filePath	string	The absolute file path of the input file to be read. A relative path is not supported.
accumField1,...., accumFieldN	$ *num* , or $ *"column_name"* if header is true.	The column position(s) or column name(s) of the data file which supply data values to each field of the accumulator.
separator	single-character string	The separator of columns.
header	bool	Whether this file has a header.

Below is an example with an external file

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> @@tes

  @@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] ")"
              | "@@"name ("." name "(" [argList] ")")+

argList := expr ["," expr]*
```

Typically, a function call returns a value and so is part of an expression (see Section 5 - Operators, Functions and Expressions). 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);
```

SELECT Statement

NOTE: There are now two versions of the grammar for the SELECT Statement. This section covers both what is common in both version (which is the majority of the grammar) plus the details for the original V1 syntax. The details of the V2 syntax are described in Pattern Matching. A query writer indicates which grammar they wish to use in the SYNTAX clause of the CREATE QUERY header.

The key differences between the two grammars:

1. 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`
2. 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 which 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`
   - There are rules about which vertex or edge aliases may be used in the WHERE or ACCUM clauses.

This section discusses the SELECT statement in depth and covers the following EBNF syntax:

EBNF for Select Statement

```
selectStmt  := name "=" selectBlock

selectBlock := SELECT name FROM ( edgeSet | vertexSet )
                    [sampleClause]
                    [whereClause]
                    [accumClause]
                    [postAccumClause]
                    [havingClause]
                    [orderClause]
                    [limitClause]

vertexSet := name [":" name]

edgeSet   := name [":" name]
             "-" "(" [vertexEdgeType] [":" name] ")" "->"
             [vertexEdgeType] [":" name]

vertexEdgeType := "_" | ANY | name | ( "(" name ["|" name]* ")" )

sampleClause := SAMPLE ( expr | expr "%" ) EDGE WHEN condition
              | SAMPLE expr TARGET WHEN condition
              | SAMPLE expr "%" TARGET PINNED WHEN condition

whereClause := WHERE condition

accumClause := ACCUM DMLSubStmtList

postAccumClause := POST-ACCUM DMLSubStmtList

DMLSubStmtList := DMLSubStmt ["," DMLSubStmt]*

DMLSubStmt := assignStmt           // Assignment
            | funcCallStmt         // Function Call
            | gAccumAccumStmt      // 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

vAccumFuncCall := name "." "@"name ("." name "(" [argList] ")")+
```

```
havingClause := HAVING condition

orderClause := ORDER BY expr [ASC | DESC] ["," expr [ASC | DESC]]*

limitClause := LIMIT ( expr | expr "," expr | expr OFFSET expr )
```

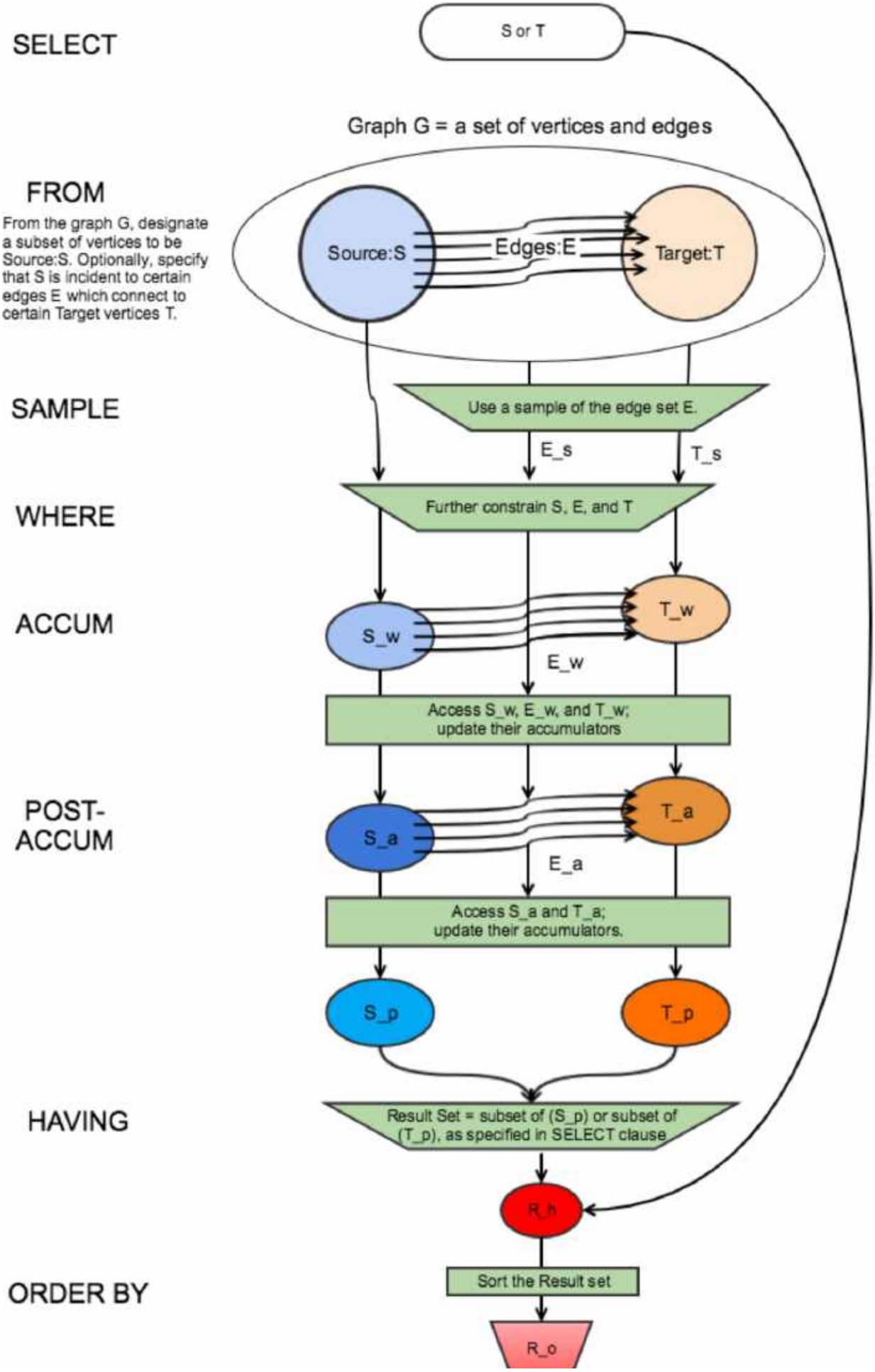
The SELECT block selects a set of vertices FROM a *vertex set* or *edge set* . There are a number of optional clauses that define and/or refine the selection by constraining the vertex or edge set or the result set. There are two types of SELECT, *vertex-induced* and *edge-induced* . Both result in a vertex set, known as the *result set* .

> ⚠ Size limitation
>
> There is a maximum size limit of 2GB for the result set of a SELECT block . If the result of the SELECT block is larger than 2GB, the system will return no data. NO error message is produced.

SELECT Statement Data Flow

The SELECT statement is an assignment statement with a SELECT block on the right hand side. The SELECT block has many possible 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.

FROM Clause: Vertex and Edge Sets


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

There are two options for the FROM clause: vertexSet or edgeSet. If vertexSet is used, then the query will be a vertex-induced selection. If edge is used, then the query is an edge-induced selection.

FROM clause

```
### selectBlock := SELECT name FROM ( edgeSet | vertexSet ) ...
```

Vertex-Induced Selection

EBNF for vertexSet, signaling a vertex-induced selection

```
vertexSet := name [":" name]
```

A vertex-induced selection takes an input set of vertices and produces a result set, which is a subset of the input set. The FROM argument has the form **Source:s** , where **Source** is a vertex set. **Source**is optionally followed by **:s** , where s is a vertex alias which represents any vertex in the set **Source.**

```
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-induced selection.

Vertex-Induced SELECT example

```
# displays all 'post'-type vertices
CREATE QUERY printAllPosts() FOR GRAPH socialNet
{
  start =   {post.*};              # initialized with all vertices of type
  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": "2011-03-06 01:02:02",
        "subject": "tigergraph"
      },
      "v_type": "post"
    },
    {
      "v_id": "7",
      "attributes": {
        "postTime": "2011-02-03 01:02:41",
```


Edge-Induced Selection

EBNF for edgeSet, signaling edge-induced selection

```
edgeSet   := name [":" name]
             "-" "(" [vertexEdgeType] [":" name] ")" "->"
             [vertexEdgeType] [":" name]

vertexEdgeType := "_" | ANY | name | ("(" name ["|" name]* ")")
```


```
      "attributes": {
        "postTime": "2011-03-03 23:02:00",
        "subject": "tigergraph"
      },
      "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"
```

```
    },
    {
      "v_id": "6",
      "attributes": {
        "postTime": "2011-02-05 02:02:05",
        "subject": "tigergraph"
      },
      "v_type": "post"
    }
  ]}]
}
```

target vertices. The edge alias e represents any edge that fits the complete pattern. Likewise, s and t are aliases that represent any source vertices and target vertices, respectively, that fit the complete pattern.

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.

Edge Set and Target Vertex Set Options

The FROM clause chooses edges and target vertices by type. The EBNF symbol vertexEdgeType describes the options:

accepted vertex/edge types	accepted vertex/edge types
_	any type
ANY	any type
name	the given vertex/edge type
name | name ...	any of the vertex/edge types listed

Note that eType and tType are optional. If **eType/tType** is omitted (or if ANY or _ is used), then the SELECT will seek out any edge or target vertex that is valid (i.e., there exists a valid path between two vertices over an edge). For the example below, if **V1** and **V2** are the only possible reachable vertex types via **eType** , we can omit the target vertex type, making all of the following SELECT statements equivalent. The system will infer the target vertex type at run time.

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;
```

Type inference is used whenever possible for the edge set and target vertex set to prune ineligible edges and thereby optimize performance. The vertex type in Source is checked against the graph schema to find all incident edge types. The knowledge of the graph schema is combined with the selection's explicit type conditions given by eType and tType, as well as explicit and implicit type conditions in the WHERE clause to determine a final set of eligible edge sets which match the pattern Source → eType → tType. With type inference, the user has the freedom to express only as much as necessary to select edges.

Similarly, the GSQL engine will infer the edge type at run time. For example, if **E1, E2** , and **E3** are the only possible edge types that can be traversed to reach vertices of

type **tType** , we can omit specifying the edge type, making the following SELECT statements equivalent.

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 c
# 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:targe

    # select all vertices of any type which can be reached from 'start' in
    #   using an edge of type 'liked'
    allPostsLiked = SELECT targetVertex FROM start -(liked:e)-> :targetVer
    # ----


    # --- statements produce equivalent results
    # start with the vertex set from above, and traverse all edges of type
    #   (locally those edges are just given a name 'e' in case they need a
    #   and return all vertices of type 'post' which can be reached within
    allPostsMade = SELECT targetVertex FROM start -(posted:e)-> post:targe

    # start with the vertex set from above, and traverse all edges of type
    #   (locally those edges are just given a name 'e' in case they need a
    #   and return all vertices of any type which can be reached within on
    allPostsMade = SELECT targetVertex FROM start -(posted:e)-> :targetVer
    # ----


    # --- 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 edg
    #   this will include any vertex connected by 'liked' or 'posted' edge
    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:t

    # select all vertices of any type which can be reached from 'start' in
    #   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, and
#   of the user
CREATE QUERY printAllRelatedItems(vertex<person> seed) FOR GRAPH socialNet
{
    sourceVertex = {seed};

    # -- statements produce equivalent output
    # returns all vertices of type either 'person' or 'post' that can be r
    #   from the sourceVertex set using one edge of any type
    everythingRelated = SELECT v FROM sourceVertex -(:e)-> (person|post):v

    # returns all vertices of any type that can be reached from the source
    #   using one edge of any type
    # this statement is equivalent to the above one because the graph sche
    #   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


```
        "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"
```

Vertex and Edge Aliases

Vertex variables

```
results = SELECT v FROM allVertices:v;
results = SELECT v FROM allVertices:s -()-> :t;
```

```
      "attributes": {
        "postTime": "2011-02-06 01:02:02",
        "subject": "tigergraph"
      },
      "v_type": "post"
    },
```

Edge variables

```
results = SELECT v
          FROM allVertices:s -(:e)-> :t
          ACCUM VERTEX v = t, EDGE eg = e;
```

```
      },
      "v_type": "person"
    },
    {
      "v_id": "8",
      "attributes": {
```

```
        "postTime": "2011-02-03 17:05:52",
        "subject": "cats"
      },
      "v_type": "post"
    }
  ]
}
```

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 selected
    SumAccum<INT> @@totalAbs;
    MapAccum<STRING,ListAccum<STRING>> @@pctEdges; // record each selected
    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 ve
             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 edg
             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 accumulato
    # 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 b
    # reached when for each source vertex, only one edge is used for trave
    # This is demonstrated by the values of 'timesTraversedWithSample' ver
    # is increased for each edge incident to the vertex which is used in t
    # 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": {
          "@timesTraversedNoSample": 3,
          "@timesTraversedWithSample": 3,
          "id": "company3"
        },
        "v_type": "company"
      },
      {
        "v_id": "company2",
        "attributes": {
          "country": "chn",
```


WHERE Clause

EBNF for Where Clause

```
whereClause := WHERE condition
```

```
            "@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 verti
               WHERE v.subject == "cats";  # which have a subset of 'cats
    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 ver
        WHERE v.subject IN ("Graph", "tigergraph"); # which have a subject
    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 statemer
# 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 a
    #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 a
    #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 e

    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",
    "attributes": {
```

```
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;
}
```

```
      "id": "person12"
    },
    "v_type": "person"
  },
  {
    "v_id": "person3",
    "attributes": {
      "interestList": ["teaching"],
      "skillSet": [ 6, 3, 4, 1 ],
      "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

that meaningful.

⚠ 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.

ⓘ As of v1.1, the keyword POST-ACCUM may also be spelled with an underscore: POST_ACCUM.

⚠ Each statement within the POST-ACCUM clause can refer to either source vertices or target vertices but not both.

In edge-induced selection, 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 := ACCUM DMLSubStmtList

postAccumClause := POST-ACCUM DMLSubStmtList

DMLSubStmtList := DMLSubStmt ["," DMLSubStmt]*

DMLSubStmt := assignStmt           // Assignment (including vertex-attache
            | funcCallStmt         // Function Call
            | gAccumAccumStmt      // Assignment (global accumulate)
            | 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 unkno
  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 orde
  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).

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 socialNe
  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 ed
                      @@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 also
                         # meets the constraint condition
                         ACCUM @@vertexOnlyWhereAccum += 1

                         #Happens once for each vertex in the result set "v
                         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 discovered
                          ACCUM @@sourceWithEdgeAccum += 1

                         #Happens once for each vertex in result set "s" (s
                         POST-ACCUM @@sourceWithEdgePostAccum += 1;

  #Select all target "t" vertices found from exploring all "worksFor" edge
  selectTargetWithEdge = SELECT t from start:s -(worksFor)-> :t
                          #Happens once for each "worksFor" edge discovered
                          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 t
              # 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 live
              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
```

A HAVING clause can only be used if there is an ACCUM or POST-ACCUM clause . The condition 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 po
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 pos
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 the
# is male before (WHERE) the ACCUM clause or after (HAVING) the ACCUM clau
# 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 3
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.

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 frier
# 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,
        "id": "person7"
```

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 = Offset
result = SELECT v FROM S -(:e)-> :v LIMIT k OFFSET j; # case 3: k = Count,
    {
      "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"
    }
  ]}]
}
```

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 a 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  conditi
DMLSubIfStmt := IF  condition  THEN  DMLSubStmtList   [ELSE IF  condition
```

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 coworke
CREATE QUERY countFriendsOf2(vertex<person> seed, BOOL includeCoworkers) F
{
       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., posts
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 socia
{
    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 r
# 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 p
            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
                   | CASE expr (WHEN constant  THEN queryBodyStmts)+ [ELSE
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 su
CREATE QUERY userNetworkPosts (vertex<person> seedUser, STRING subjectName
    SumAccum<INT> @@topicSum = 0;
    OrAccum @visited;
    reachableVertices = {};            # empty vertex set
    visitedVertices (ANY) = {seedUser};  # set that can contain ANY type o

    WHILE visitedVertices.size() !=0 DO        # loop terminates when all
        visitedVertices = SELECT s              # s is all neighbors of vi
            FROM visitedVertices-(:e)->:s
            WHERE s.@visited == false
            ACCUM s.@visited = true,
                CASE
                    WHEN s.type == "post" and s.subject == subjectName THE
                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 socialNet
       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.@socialImpa
                    CASE WHEN v.outdegree("friend") > 2 THEN v.@socialImpa
                    CASE WHEN v.outdegree("posted") > 1 THEN v.@socialImpa
                    CASE WHEN v.outdegree("posted") > 3 THEN v.@socialImpa
       #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 ab
          WHEN v.subject == "Graphs" THEN v.@rating += 2   # +2 if post ab
          WHEN v.subject == "tigergraph" THEN v.@rating += 10 # +10 if pos
          END
      WHEN "liked" THEN v.@rating += 3                 # +3 each time post
      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 (name | integer)] DO queryB
DMLSubWhileStmt := WHILE  condition  [LIMIT (name | integer)] DO DMLSubStm
```

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 ve

    WHILE visitedVertices.size() !=0 DO        # loop terminates when all
        visitedVertices = SELECT s              # s is all neighbors of vi
                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",
        "nagement"],
```


Example 2. WHILE loop using a LIMIT

```
# find all vertices which are reachable within two hops from a starting se
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 ve

    WHILE visitedVertices.size() !=0 LIMIT 2 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;
}
```

reachableWithinTwo Results


```
        "country": "chn",
        "@visited": true,
        "ny2"
      },
      "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": [ 2, 3, 5, 6 ],
        "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 := (name | "(" name [, name]+ ")") IN setBagExpr
                | name IN RANGE "[" expr , expr"]" [".STEP(" expr ")"]
```


```
        "id": "company2"
      },
      "v_type": "company"
    },
    {
      "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"
    }
  ]}]
}
```

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 $|k*c| \leq |b-a|$.

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 ver

  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 fro

  # Iterate the ListAccum and compare each element to the countryName para
  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" ],
```

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",
          "interestSet": [ "financial", "management" ],
```

```
# 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;
}
```

```
          "locationId": "us",
          "et": [ "financial", "management" ],
```

employeeByCountry Results

```
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, friend
# 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 it
        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"
          },
          {
```

Example 3. While loop using break statement

```
            "attributes": {
              "@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,
              "id": "person12"
```

```
# find at least the top-k companies closest to a given seed vertex, if the
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 vis

        # 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 co

  #Build a map using company country as a key and a list of strings to hol
  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 h
  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.

EBNF

```
QueryBodyDeleteStmt := DELETE name FROM ( edgeSet | vertexSet ) [whereClau
```

The vertexSet and edgeSet 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 vertexSet or edgeSet.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 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 "(" name ")"
```

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 occurs
    # 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 ta
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": [{"selectedPosts": [{
    "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": [{"selectedPosts": []}]
}
```

INSERT INTO Statement

The INSERT INTO statement adds edges or vertices to the graph. However, if 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 prior to 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-substatement.

EBNF

```
insertStmt := INSERT INTO name ["(" ( PRIMARY_ID | FROM "," TO ) ("," name
                    VALUES "(" ( "_" | expr ) [name] ["," ( "_" | expr )
```

The formal syntax is complex because it encompasses several options, and even so, it requires additional explanation. The first *name* symbol is the vertex type or edge type. The user then has two options:

1) Provide a value for the ID(s) and then each attribute, in the canonical order for the vertex or edge type. This format is similar to that of a LOAD statement. In this case, it is not necessary to explicitly name the attributes, since it is assumed that every one is being referenced, in order.

INSERT with implicit attribute names

```
INSERT INTO name VALUES (full_list_of_parameter_values)
```

2) 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 name (IDs, specified_attributes) VALUES (values_for_specified_
```

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 examples into the workNet graph.

INSERT statement

```
CREATE QUERY insertEx(STRING name, STRING name2, STRING name3, STRING comp
  # Vertex insertion
    # Adds 2 'company' vertices. One is located in the USA, and a sister c
    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', fil
    # values for startYear (0), startMonth (0), and fullTime (false).
    INSERT INTO worksFor VALUES (name person, comp company, _, _, _);

    # Adds a 'worksFor' edge from person 'name2' to the company 'comp', fi
    # values for startMonth (0), but specifying values for startYear and f
    INSERT INTO worksFor (FROM, TO, startYear, fullTime) VALUES (name2 per

    # Adds a 'worksFor' edge from person 'name3' to the company 'comp', fi
    # values for startMonth (0), and fullTime (false) but specifying a val
    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 show a DML-sub level INSERT. Because the statement applies to allCompanies, several vertices will be inserted.

DML-sub INSERT statement

```
# Add a child company of a given company name. The new child company is ir
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 insertior
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 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 attribute of each vertex or edge in a vertex set or edge set, respectively, with new attribute values.

EBNF

```
updateStmt := UPDATE name FROM ( edgeSet | vertexSet ) SET DMLSubStmtList
```

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 block. In the SET clause, the DMLSubStmtList may contain 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 vertexSet or edgeSet.

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 still
  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. Below is an example.

⚠ Updating a vertex's attribute value in a ACCUM clause is not allowed, because the update can occur multiple times in parallel, and possibly result in an 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 the SELECT statement instead of the UPDATE statement, but is functional similar to the query above. Query updateEx2 reverses the locationId change made by updateEx (changing the location back to "us" from "USA").

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 (e.startYear = e.startYear + 1):

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,
      e.fullTime = false
  WHERE s.locationId == "us";

  PRINT S;
}
```

Other Update Methods

In addition to the above UPDATE statement and SELECT statement, a simple assignment statement at the query-body level can be used to update the attribute value of a single vertex/edge, if the vertex/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.

EBNF

```
printStmt := PRINT printExpr {,printExpr} [WHERE condition] [TO_CSV (fileP
printExpr := (expr | vExprSet) [ AS name]
vExprSet  := expr "[" vSetProj {, vSetProj} "]"
vSetProj  := expr [ AS 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 key for the expression, overriding the default key (explained below).

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:

1. A literal value
2. A global or local variable (including VERTEX and EDGE variables)
3. An attribute of a vertex variable, e.g., Person.name
4. A global accumulator
5. An expression whose terms are among the types above. The following operators may be used:

Numeric	Arithmetic: `+ - * / . %` Bit: `<< >> & |`
String	concatenation: `+`
Set	`UNION INTERSECT MINUS`

Parentheses can be used for controlling order of precedence.

6. A vertex set variable
7. 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.

⚠ In output API v2, the print expression list can be a mixed list of any of the expression types.
In output API v1, vertex set variables cannot be on the same PRINT statement with other types of expressions.

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 a 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 STR

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.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, whic
# happens to be a set of edges (SetAccum<EDGE>), so edge format is shown a
  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"
      }
      {
        "v_id": "person2",
        "attributes": {
          "gender": "Female",
          "id": "person2",
          "@postedSet": [{
            "from_type": "person",
            "to_type": "post",
            "directed": true,
            "from_id": "person2",
            "to_id": "1",
            "attributes": {},
            "e_type": "posted"
          }]
        },
        "v_type": "person"
      }
    ]},
    {"VSetExpr": [
      {
        "v_id": "person4",
```


Printing CSV to a FILE Object

PRINT to CSV FILE syntax example

```
PRINT @@setOfVertices TO_CSV file1;
```

```
        "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"
      }
    ]}
  ]
}
```

1. The query prints only one set of data, and the order of the set is not important.
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 cont
  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\r
}
```

Printing to a CSV File as a Filepath (DEPRECATED)

Instead of printing CSV output to a FILE object, data can be written to a regular file.

PRINT to CSV FILE syntax example

```
  PRINT @@setOfVertices TO_CSV "/home/tigergraph/vset.csv";
```

This feature is deprecated because printing to a FILE object covers the same functionality.

The table below shows the differences between printing TO_CSV <FILE object> vs. TO_CSV <fllepath>.

	FILE Object	filepath
When filepath is specified	Either run-time or compile-time, depending on how users	compile-time

	chooses to write the query	
Vertex IDs	displayed correctly	displayed as TigerGraph internal ID codes
Append or overwrite	Appends, but FILE object declaration will reset the FILE.	Always appends.
filepath can be absolute or relative	Currently only absolute	Absolute or relative

FILE println statement

One of the two ways to write data to a FILE object is with the FILE println statement. (The other way is with the PRINT statement's TO_CSV option.)

EBNF for FILE println statement

```
printlnStmt := fileVar".println" "(" expr {, expr} ")"
```

println is a method (function) of a FILE object variable. The println statement can be used either at the query-body level or a 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 most of the expressions handled by PRINT. Note, however, that this does not include vertex expression sets (vExprSet). If the println statement has a list of expressions to print, then this 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 same format produced by TO_CSV.

⚠ 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. Moreover, the output from println statements in an ACCUM block can be interspersed with the query-body statements.

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
```

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 the the FILE object. In the query definition, the footer is the last FILE statement, but the println statements from the SELECT block happen to be delayed and are printed AFTER the footer line.

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
```

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 socialN
    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

```
createQuery := CREATE QUERY name "(" [parameterList] ")" FOR GRAPH name [F

returnStmt := RETURN expr
```

The RETURN statement specifies data that a *sub-query* passes back to an outer query that called the sub-query. In order for a query to be used as a subquery, its initial CREATE QUERY statement must include the optional RETURNS clause, and its body must end with a RETURN statement. Exactly one type is allowed in the RETURNS clause, and thus RETURN statement can only return one expression.The returned expression must have the same type as the RETURNS clause indicates. A sub-query must be created before its corresponding super-query. A sub-query must be install either before or in the same INSTALL QUERY command with its super-query.

The return type can be any base type or any accumulator type. For the purposes of return type, SetAccum is equivalent to SET, and BagAccum is equivalent to BAG. A vertex set variable can be returned if SET<VERTEX<type>> or SetAccum<VERTEX<type>> (<type> is optional) is used in the RETURNS clause.

For subqueries to return a HeapAccum or GroupByAccum, the accumulators must be defined at the global level (outside of any query), as part of the schema. HeapAccum also requires a corresponding tuple type to be defined. See example below.

See also Section 5.11 - Queries and Functions.

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;
    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);
    RETURB @@heap;
}

CREATE QUERY query1() FOR GRAPH socialNet {
    PRINT subquery1();
}
```

Subquery Example 1

```
CREATE QUERY subquery1 (VERTEX<person> m1) FOR GRAPH socialNet RETURNS(Ba
{
  Start = {m1};
  L = SELECT t
      FROM Start:s - (liked:e) - post:t;
  RETURN L;
}
CREATE QUERY mainquery1 () FOR GRAPH socialNet
{
  BagAccum<VERTEX<post>> @@testBag;
  Start = {person.*};
  Start = SELECT s FROM Start:s
          ACCUM @@testBag += subquery1(s);
  PRINT @@testBag;
}
```

Result

```
GSQL > RUN QUERY mainquery1()
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@testBag": [
    "6",
    "3",
    "8",
    "4",
    "4",
    "0",
    "0",
    "0",
    "10"
  ]}]
}
```

Subquery Example 2

```
CREATE QUERY subquery2 (VERTEX<person> m1) FOR GRAPH socialNet RETURNS(INT
{
  int x;
  Start = {m1};
  Start = SELECT t FROM Start:t
          ACCUM CASE WHEN t.gender == "Male" THEN x = 5
                     WHEN t.gender == "Female" THEN x = 10
                     ELSE x = -1
                END;
  RETURN x;
}
CREATE QUERY mainquery2 (SET<VERTEX<person>> m1) FOR GRAPH socialNet
{
  SumAccum<INT> @@sum1;
  Start = {m1};
  Start = SELECT t FROM Start:t
          ACCUM @@sum1 += subquery2(t);
  PRINT @@sum1;
}
```

Result

```
GSQL > RUN QUERY mainquery2(["person1","person2"])
{
  "error": false,
  "message": "",
  "version": {
    "edition": "developer",
    "schema": 0,
    "api": "v2"
  },
  "results": [{"@@sum1": 15}]
}
```

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) FOR
  # Count how many edges there are from each member of the input person se
  # 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<perso
  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?vSe
{
  "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?vse
{
  "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?vSe
{
  "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 [elseExcep
caseExceptBlock := WHEN exceptVarName THEN queryBodyStmts+
elseExceptBlock := ELSE queryBodyStmts
```

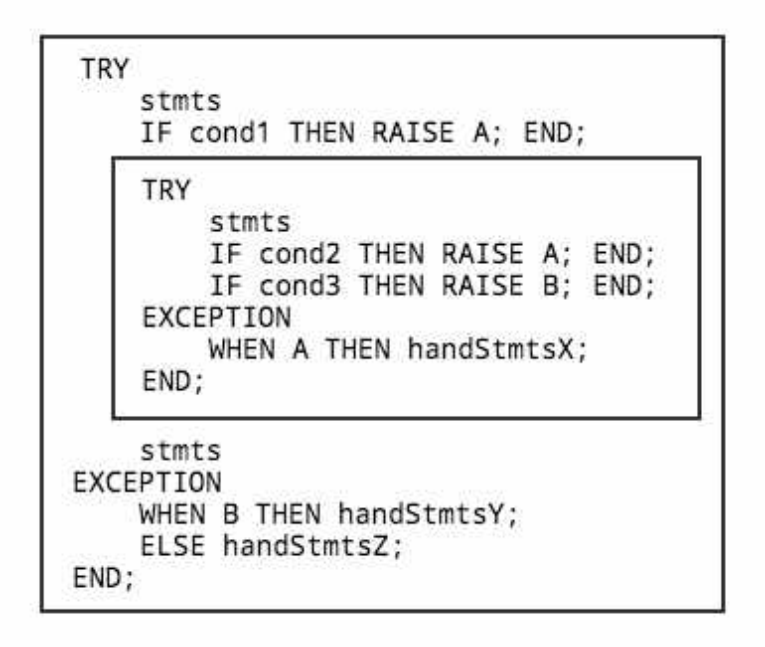
The TRY...EXCEPTION Statement is used to define and apply user-defined exception handling to a block of query-body statements. A TRY...EXCEPTION statement can be nested within a TRY block or EXCEPTION block.

> ⚠ 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.



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, BO
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.

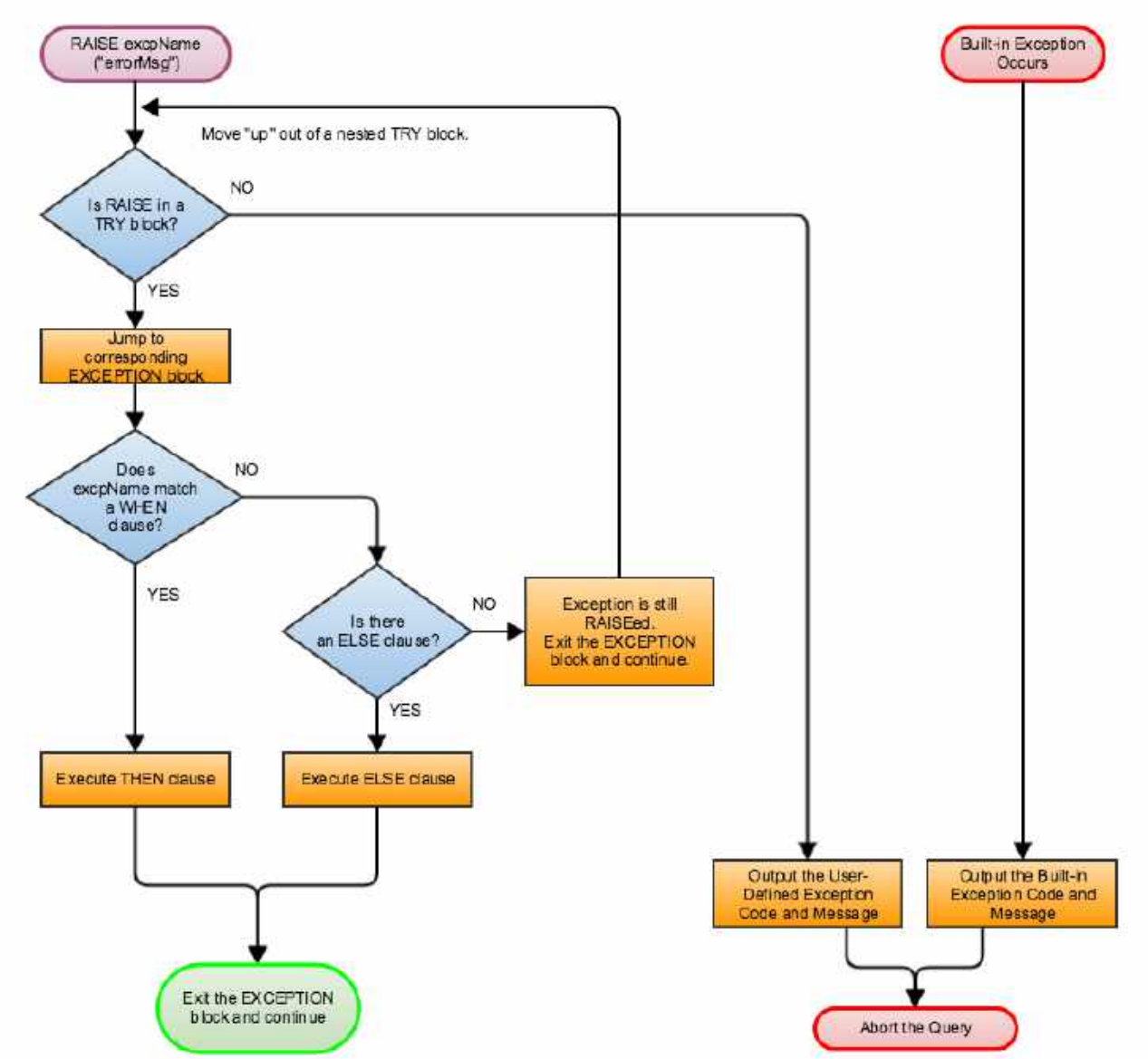
compPathValid.json

```
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 ta
}
```

```
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
YES
Exception is still RAISEed. Exit the EXCEPTION block and continue
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

Comments

A comment is a section of text that is ignored by the language parser; its purpose is to provide information to human readers. The comment markers follow the conventions used in C++ and SQL:

- Single-line or partial-line comments begin with either # or // and end at the end of the line (with the newline character).
- Multi-line comment blocks begin with /* and end with */

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 involves
equality test between float/double numeric values. We suggest to do such
comparison with an error margin, e.g. 'abs((v) - (v2)) < epsilon', where e
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 ".

Complete Formal Syntax for Query Language

This is the definition for the GSQL Query Language syntax. It is defined as a set of rules expressed in EBNF notation.

Notation Used to Define Syntax

This defines the EBNF notation used to describe the syntax. Rules contains terminal and non-terminal symbols. A terminal symbol is a base-level symbol which expresses literal output. All symbols in single or double quotes (e.g., '+', "=", ")", "10") are terminal symbols. A non-terminal symbol is defined as some combination of terminal and non-terminal symbols. The left-hand side of a rule is always a non-terminal; this rule defines the non-terminal. The example rule below defines assignmentStmt (that is, an Assignment Statement) to be a name followed by an equal sign followed by an expression, operator, and expression with a terminating semi-colon. AssignmentStmt, name, and expr are all non-terminals. Additionally, all KEYWORDS are in all-capitals and are terminal symbols. The ":=" is part of EBNF and states the left hand side can be expanded to the right hand side.

EBNF Syntax example: A rule

```
assignmentStmt := name "=" expr op expr ";"
```

A **vertical bar |** in EBNF indicates choice. Choose either the symbol on the left or on the right. A sequence of vertical bars means choose any one of the symbols in the sequence.

EBNF Syntax: vertical bar

```
op := "+" | "-" | "*" | "/"
```

Square brackets [ ] indicate an optional part or group of symbols. **Parentheses ( )** group symbols together. The rule below defines a constant to be one, two, or three digits preceded by an optional plus or minus sign.

EBNF Syntax: Square brackets and parentheses

```
constant := ["+" | "-"] (digit | (digit digit) | (digit digit digit))
```

Star * and **plus +** are symbols in EBNF for closure. Star means zero or more occurrences, and plus means one or more occurrences. The following defines intConstant to be an optional plus or minus followed by one or more digits. It also defines floatConstant to be an optional plus or minus followed by zero or more digits followed by a decimal followed by one or more digits. The star and plus also can be applied to groups of symbols as in the definition of list. The non-terminal list is defined as a parenthesized list of comma-separated expressions (expr). The list has at least one expression which can be followed by zero or more comma-expression pairs.

EBNF Syntax: square brackets and parentheses

```
intConstant := ["+" | "-"] digit+
floatConstant := ["+" | "-"] digit* "." digit+
list := "(" expr ["," expr]* ")"
```

Curly braces **{ }** enclose an optional group of symbols which are repeated zero or more times. Therefore, curly braces are equivalent to square brackets or parentheses followed by a star + to indicate zero or more repetitions. All of the following expressions are equivalent:

```
list1 := expr ["," expr]*
list2 += expr ("," expr)*
list3 := expr {"," expr}
```

For brevity, the literal comma is sometimes shown without quotation marks:

```
list4 := expr {, expr}
```

GSQL Query Language EBNF

```
##################################################################
## EBNF for GSQL Query Language

createQuery := CREATE [OR REPLACE][DISTRIBUTED] QUERY name "(" [parameterL
               FOR GRAPH name
               [RETURNS "("  baseType | accumType ")"]
               [API "(" stringLiteral ")"]
               [SYNTAX syntaxName]
               "{" queryBody "}"

interpretQuery := INTERPRET QUERY "(" ")"
               FOR GRAPH name
               [SYNTAX syntaxName]
               "{" queryBody "}"

parameterValueList := parameterValue [, parameterValue]*
parameterValue := parameterConstant
                | "[" parameterValue [, parameterValue]* "]"  // BAG or SE
                | "(" stringLiteral, stringLiteral ")"        // a generic
parameterConstant := numeric | stringLiteral | TRUE | FALSE
parameterList := parameterType name ["=" constant]
                 ["," parameterType name  ["=" constant]]*

syntaxName := name

queryBody := [typedefs] [declStmts] [declExceptStmts] queryBodyStmts
typedefs := (typedef ";")+
declStmts := (declStmt ";")+
declStmt := baseDeclStmt | accumDeclStmt | fileDeclStmt
declExceptStmts := (declExceptStmt ";")+
queryBodyStmts := (queryBodyStmt ";")+
queryBodyStmt := assignStmt           // Assignment
               | vSetVarDeclStmt      // Declaration
               | gAccumAssignStmt     // Assignment
               | gAccumAccumStmt      // 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

installQuery := INSTALL QUERY [installOptions] ( "*" | ALL |name [, name]*
runQuery := (RUN | INTERPRET) QUERY [runOptions] name "(" parameterValueLi

showQuery := SHOW QUERY name
dropQuery := DROP QUERY ( "*" | ALL | name [, name]* )

#############################################################################
## Types and names

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 "_" or

type := baseType | name | accumType | STRING COMPRESS

baseType := INT
          | UINT
          | FLOAT
          | DOUBLE
          | STRING
          | BOOL
          | VERTEX ["<" name ">"]
          | EDGE
          | JSONOBJECT
          | JSONARRAY
          | DATETIME

filePath := name | stringLiteral

typedef := TYPEDEF TUPLE "<" tupleType ">" name

tupleType := (baseType name) | (name baseType) ["," (baseType name) | (nam

parameterType := baseType
               | [ SET | BAG ] "<" baseType ">"
               | FILE
```

```
                | ...---

##############################################################################
## Accumulators

accumDeclStmt := accumType "@"name ["=" constant][, "@"name ["=" constant]
               | "@"name ["=" constant][, "@"name ["=" constant]]* accumTy
               | [STATIC] accumType "@@"name ["=" constant][, "@@"name ["=
               | [STATIC] "@@"name ["=" constant][, "@@"name ["=" constant


accumType := "SumAccum" "<" ( INT | FLOAT | DOUBLE | STRING | STRING COMPR
           | "MaxAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
           | "MinAccum" "<" ( INT | FLOAT | DOUBLE ) ">"
           | "AvgAccum"
           | "OrAccum"
           | "AndAccum"
           | "BitwiseOrAccum"
           | "BitwiseAndAccum"
           | "ListAccum" "<" type ">"
           | "SetAccum"  "<" elementType ">"
           | "BagAccum"  "<" elementType ">"
           | "MapAccum"  "<" elementType "," baseType | accumType | name "
           | "HeapAccum" "<" name ">" "(" (integer | name) "," name [ASC |
           | "GroupByAccum" "<" elementType name ["," elementType name]* ,
           | "ArrayAccum" "<" name ">"

elementType := baseType | name | STRING COMPRESS

gAccumAccumStmt := "@@"name "+=" expr

##############################################################################
## Operators, 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)
```

```
comparisonOperator := "<" | "<=" | ">" | ">=" | "==" | "!="

expr :=   ["@@"]name
        | name "." "type"
        | name "." ["@"]name
        | name "." "@"name ["\'"]
        | name "." name "." name "(" [argList] ")"
        | name "." name "(" [argList] ")" [ ".".FILTER "(" condition ")" ]
        | name ["<" type ["," type"]* ">"] "(" [argList] ")"
        | name "." "@"name ("." name "(" [argList] ")")+ ["." name]
        | "@@"name ("." name "(" [argList] ")")+ ["." name]
        | COALESCE "(" [argList] ")"
        | ( COUNT | ISEMPTY | MAX | MIN | AVG | SUM ) "(" 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 object

setBagExpr := ["@@"]name
            | name "." ["@"]name
            | name "." "@"name ("." name "(" [argList] ")")+
            | name "." name "(" [argList] ")" [ ".".FILTER "(" condition "
            | "@@"name ("." name "(" [argList] ")")+
            | setBagExpr (UNION | INTERSECT | MINUS) setBagExpr
            | "(" argList ")"
            | "(" setBagExpr ")"

###############################################################################
## Declarations and Assignments ##

## Declarations ##
baseDeclStmt    := baseType name ["=" constant][, name ["=" constant]]*
fileDeclStmt := FILE fileVar "(" filePath ")"
fileVar := name

localVarDeclStmt := baseType name "=" expr

vSetVarDeclStmt := name ["(" vertexEdgeType ")"] "=" (seedSet | simpleSet

simpleSet := name | "(" simpleSet ")" | simpleSet (UNION | INTERSECT | MIN

seedSet := "{" [seed ["," seed ]*] "}"
seed := '_'
      | ANY
      | ["@@"]name
```

```
      | name ".*"
      | "SelectVertex" selectVertParams

selectVertParams := "(" filePath "," columnId "," (columnId | name) ","
                 stringLiteral "," (TRUE | FALSE) ")" [".".FILTER "(" cond

columnId := "$" (integer | stringLiteral)

## Assignment Statements ##
assignStmt := name "=" expr
            | name "." name "=" expr
            | name "." "@"name ("+="| "=") expr

gAccumAssignStmt :=  "@@"name ("+=" | "=") expr

loadAccumStmt := "@@"name "=" "{" "LOADACCUM" loadAccumParams ["," "LOADA

loadAccumParams := "(" filePath "," columnId "," [columnId ","]*
                stringLiteral "," (TRUE | FALSE) ")" [".".FILTER "(" cond

## Function Call Statement ##
funcCallStmt := name ["<" type ["," type"]* ">"] "(" [argList] ")"
              | "@@"name ("." name "(" [argList] ")")+

argList := expr ["," expr]*


#########################################################################
## Select Statement

selectStmt  := name "=" selectBlock

selectBlock := SELECT name FROM ( edgeSet | vertexSet )
                    [sampleClause]
                    [whereClause]
                    [accumClause]
                    [postAccumClause]
                    [havingClause]
                    [orderClause]
                    [limitClause]

vertexSet := name [":" name]

edgeSet   := name [":" name]
             "-" "(" [vertexEdgeType] [":" name] ")" "->"
             [vertexEdgeType] [":" name]

vertexEdgeType := "_" | ANY | name | ( "(" name ["|" name]* ")" )
```

```
sampleClause := SAMPLE ( expr | expr "%" ) EDGE WHEN condition
              | SAMPLE expr TARGET WHEN condition
              | SAMPLE expr "%" TARGET PINNED WHEN condition

whereClause := WHERE condition

accumClause := ACCUM DMLSubStmtList

postAccumClause := POST-ACCUM DMLSubStmtList

DMLSubStmtList := DMLSubStmt ["," DMLSubStmt]*

DMLSubStmt := assignStmt           // Assignment
            | funcCallStmt         // Function Call
            | gAccumAccumStmt      // 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

vAccumFuncCall := name "." "@"name ("." name "(" [argList] ")")+

havingClause := HAVING condition

orderClause := ORDER BY expr [ASC | DESC] ["," expr [ASC | DESC]]*

limitClause := LIMIT ( expr | expr "," expr | expr OFFSET expr )

#####################################################################
## Control Flow Statements ##

queryBodyIfStmt := IF condition THEN queryBodyStmts [ELSE IF condition THE
DMLSubIfStmt :=    IF condition THEN DMLSubStmtList [ELSE IF condition THE

queryBodyCaseStmt := CASE      (WHEN condition THEN queryBodyStmts)+ [ELSE
                   | CASE expr (WHEN constant  THEN queryBodyStmts)+ [ELSE
DMLSubCaseStmt := CASE      (WHEN condition THEN DMLSubStmtList)+ [ELSE DM
                | CASE expr (WHEN constant  THEN DMLSubStmtList)+ [ELSE DM

queryBodyWhileStmt := WHILE  condition  [LIMIT (name | integer)] DO queryB
```

```
DMLSubWhileStmt :=    WHILE  condition  [LIMIT (name | integer)] DO DMLSub

queryBodyForEachStmt := FOREACH  forEachControl  DO  queryBodyStmts  END
DMLSubForEachStmt :=    FOREACH  forEachControl  DO  DMLSubStmtList  END

forEachControl := ( name | "(" name (, name)+ ")") (IN | ":") setBagExpr
                | name IN RANGE "[" expr , expr"]" [".STEP(" expr ")"]

###################################################################
## Other Data Modifications Statements ##

queryBodyDeleteStmt := DELETE name FROM ( edgeSet | vertexSet ) [whereClau
                       TE "(" name ")"
```

Query Language Reserved Words

ACCUM	AND	ANY	API
AS	ASC	AVG	BAG
BATCH	BETWEEN	BOOL	BOTH
BREAK	BY	CASE	CATCH
COALESCE	COMPRESS	CONTINUE	COUNT
CREATE	DATETIME	DATETIME_ADD	DATETIME_SUB
DELETE	DESC	DISTRIBUTED	DO
DONE	DOUBLE	EDGE	ELSE
END	ESCAPE	EXCEPTION	FALSE
FILE	FILTER	FLOAT	FOR
FOREACH	FROM	GRAPH	GSQL_INT_MAX
GSQL_INT_MIN	GSQL_UINT_MAX	HAVING	IF
IN	INSERT	INT	INTERPRET
INTERSECT	INTERVAL	INTO	IS
ISEMPTY	JSONARRAY	JSONOBJECT	LEADING
LIKE	LIMIT	LIST	LOAD_ACCUM
LOG	MAP	MAX	MIN
MINUS	NOT	NOW	NULL
OFFSET	OR	ORDER	PINNED
POST_ACCUM	POST-ACCUM	PRIMARY_ID	PRINT
QUERY	RAISE	RANGE	REPLACE
RETURN	RETURNS	RUN	SAMPLE
SELECT	SELECT_VERTEX	SET	STATIC
STRING	SUM	SYNTAX	TARGET
THEN	TO	TO_CSV	TO_DATETIME
TRAILING	TRIM	TRUE	TRY
TUPLE	TYPEDEF	UINT	UNION
UPDATE	VALUES	VERTEX	WHEN
WHERE	WHILE		

```
tryStmt         := TRY queryBodyStmts EXCEPTION caseExceptBlock+ [elseExce
caseExceptBlock := WHEN exceptVarName THEN queryBodyStmts
elseExceptBlock := ELSE queryBodyStmts
```

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 STR
CREATE VERTEX company(PRIMARY_ID clientId STRING, id STRING, country STRIN
CREATE UNDIRECTED EDGE worksFor(FROM person, TO company, startYear INT, st
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) W
CREATE UNDIRECTED EDGE friend(FROM person, TO person)
CREATE VERTEX post(PRIMARY_ID postId UINT, subject STRING, postTime DATETI
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, connectionSpeed
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, DOU
CREATE VERTEX stockOrder(PRIMARY_ID orderId STRING, ticker STRING, orderSi
CREATE UNDIRECTED EDGE makeOrder(FROM person, TO stockOrder, orderTime DAT
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

GSQL features not yet supported in Interpreted Mode

Version 2.4

Currently, interpreted mode cannot be combined with distributed query execution model, i.e., 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 table below lists additional limitations. These limitations are expected to be temporary. We are continuing to expand the capabilities of Interpreted Mode.

The "Not Supported" column is intended to be comprehensive, but the "Supported" column is not. Rather it gives examples to show the contrast with what is not supported.

Category	Supported (highlights, not a full list)	Not Supported
Modes	Queries in regular (non-distributed) mode	Distributed mode
Statement types	• TYPEDEF tuple • SELECT ... FROM <edge_set> • SELECT ... FROM <vertex_set> • CASE ... WHEN ... THEN • IF ... ELSE ... THEN • WHILE • BREAK, CONTINUE	• UPDATE • INSERT • DELETE • LOG • Exceptions: ◦ RAISE

	• FOREACH, FOREACH... RANGE+ • Assignment for accumulators or local variables	◦ TRY
SELECT block clauses	• PRINT • ACCUM • POST-ACCUM • WHERE • HAVING • ORDER BY (but output will not be sorted) • LIMIT	• SAMPLE clause in SELECT
Attributes and Accumulators	• Global and local accumulators • Global and local variables • Most accumulator types	• Attributes cannot be accessed outside of the ACCUM or POST-ACCUM clauses • ArrayAccum • Previous value of accumulator with ' operator, e.g., @@acc'
Functions and Operators	• Math operators • Comparison operators • Boolean operators • to_vertex(), to_vertex_set() • Math functions • Most string functions: to_string(), float_to_int(), str_to_int(), lower(), upper() • IN, NOT IN • LIKE	• BETWEEN ... AND • IS NULL, IS NOT NULL (checking whether parameters are absent/present) • built-in functions • Set functions: COUNT(), MAX(), .FILTER(), etc. • isDirected() • trim() • neighbor(), neighborAttribute() • COALESCE() • evaluate()

		• selectVertex() • LOADACCUM() • User-Defined Functions
Data types	Explicit lists, e.g., [1, 3, 2]	• JSONOBJECT, JSONARRAY • STRING COMPRESS as accumulator type • Built-in Constants GSQL_INT_MAX, GSQL_INT_MIN, GSQL_UINT_MAX • Explicit sets, e.g., (1, 3, 2) • BAG type parameters
Output options		• JSON format V1 • PRINT ... WHERE • PRINT TO_CSV • FILE objects

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>/dev/gdk/gsql/lib/` . Installation consists of copying the file gsql_client.jar to the client machine and storing in anywhere the user finds to be 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.

⚠ 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_root_dir>/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/ex

# 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)'
```

GSQL Cheatsheets

GSQL Language Quick Reference Manuals

PDF
154KB

gsql_ddl_loading_v2.2.pdf
pdf

PDF
194KB

gsql_query_v2.2.pdf
pdf

RESTful API User Guide

Introduction

The TigerGraph TM system uses the well-known Representational State Transfer (REST) architecture to manage communication with the TigerGraph core components, the Graph Processing Engine (GPE) and Graph Storage Engine (GSE). REST++ (or RESTPP) is the TigerGraph customized REST server. (See Figure 1 below) When an upper layer component, such as the Platform Web UI or GSQL, wishes to access the graph engine, it sends a request to the REST++ server. Users can also communicate directly with the REST++ server, either by using one of the standard REST APIs included with the system, or by authoring and then employing a custom endpoint API. This document describes the APIs for the built-in endpoints, which provides methods for basic querying and manipulation of the graph data.

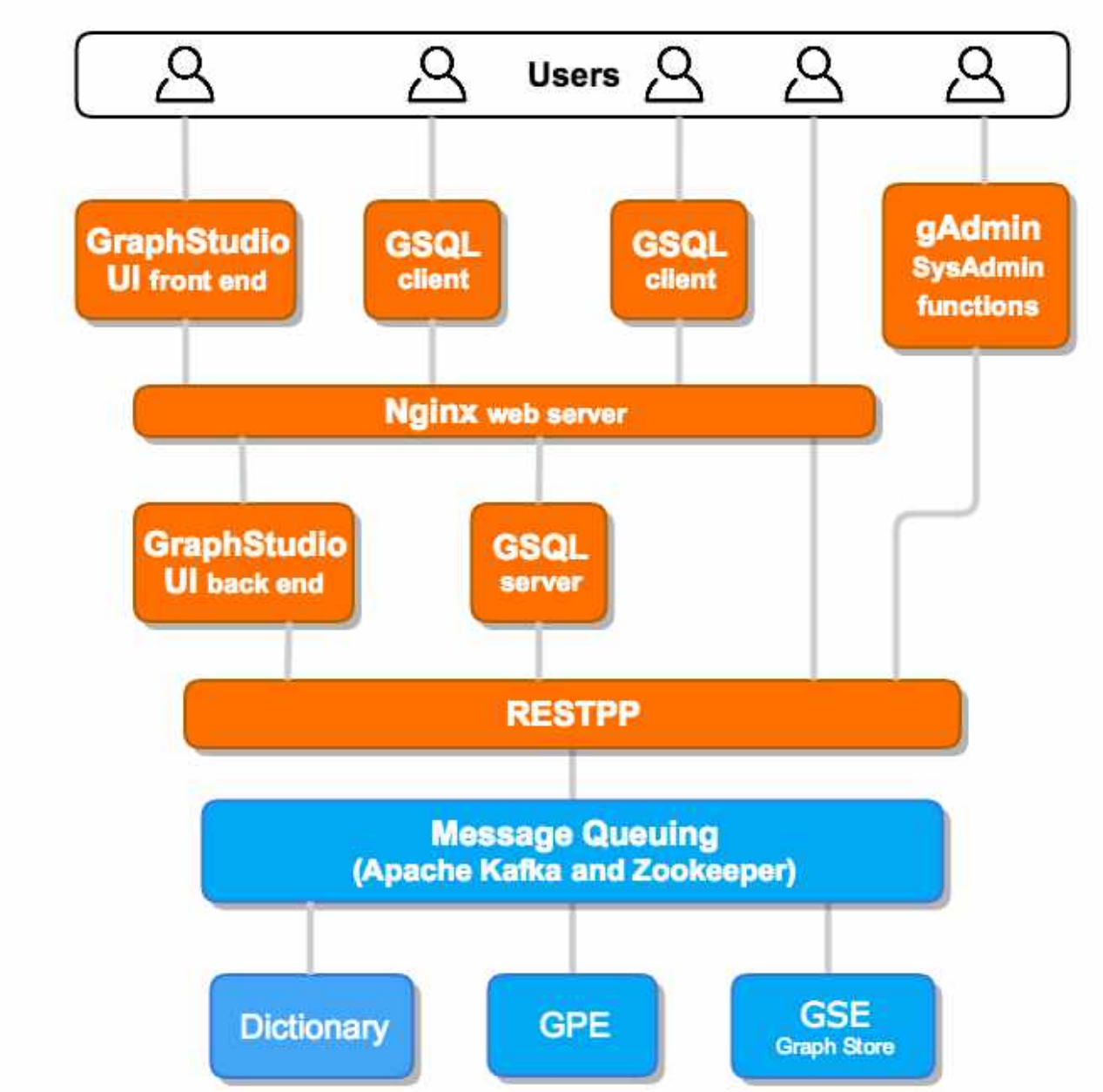


Figure 1: TigerGraph System Block Diagram

Like most RESTful systems, REST++ employs the HTTP protocol (specifically HTTP/1.1 without request pipelining). Accordingly, REST APIs feature request methods and URLs, response status codes, and data responses. This guide describes the request methods and URLs used to query, update, and delete from the graph data. It also describes the format of the data responses.

The TigerGraph REST APIs employ three HTTP request methods:

- **GET** is used to request data.
- **POST** is used to send data.

- **DELETE** is used to delete data.

If the user submits an unsupported HTTP method, the API will return an error message: "endpoint not found".

RESTPP Requests

To submit a request, an HTTP request is sent to the REST++ server. By default, the REST++ server listens for requests at port 9000. A request needs to specify three things:

1. the request method (GET, POST, or DELETE),
2. the endpoint address, and
3. any required or optionally request parameters.

The endpoint address is the the form of a HTTP URL.

Request parameters are appended to the end using standard HTTP query string format.

> In a test or development environment, the requester may be on the same server as REST++. In this case, the server_ip is **localhost** .

The Linux curl command is the most convenient way to submit the HTTP request to the REST++ server.

REST Request Format

```
curl -X method "http://server_ip:9000/path_to_endpoint?request_parameters"
```

Example:

Assume the REST++ server is on the local machine (typical configuration) and there is a graph called socialNet. To get all the User vertices from socialNet:

Example: Get all User vertices

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/User"
```

To list only the first three vertices, we can set limit = 3:

Example: Get up to 3 User vertices

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/User?limit=3"
```

⚠ The HTTP request methods GET, POST, and DELETE are case sensitive. Also, curl option flags are case sensitive.

Input Data for POST

Input data for POST requests should be in JSON format. 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. Use the curl - **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?requ
```

The following example uses the POST /graph endpoint to insert one User type vertex whose id value is "id6" into the graph called "socialNet".

Example using inline input data

```
curl -X POST -d '{"vertices":{"User":{"id6":{"id":{"value":"id6"}}}}}' "ht
```

Data File

Often it will be more convenient for the input data to be in a separate file, especially if it is large.

Use the curl option **--data-binary @path_to_file** as in the example below:

Syntax for a POST request with Payload Data File

```
curl -X POST --data-binary @json_file "http://server_ip:9000/path_to_endpo
```

If we now store the data string in a file (say, my_input.json), then the example above becomes the following:

Example using inline input data

```
curl -X POST --data-binary @my_input.json "http://localhost:9000/graph/soc
```

REST++ Output

All TigerGraph REST responses are in JSON format. The format details for each built-in endpoint are described below in the Built-in Endpoints section. By default, the output is designed for machine reading, with no extra spaces or linefeeds. The output JSON object can have three fields: error, message, and result.

⚠ This document has been updated to show JSON output API v2. Earlier versions of the TigerGraph platform produced JSON output in a slightly different format (v1). Newer platforms can be configured to produce output in either v2 or v1 formats.

✓ To make the output more human readable, use the jq command ↗ or Python json library built into most Linux installations. Specifically,

```
curl -X method "http://server_ip:9000/path_to_endpoint?request_para
curl -X method "http://server_ip:9000/path_to_endpoint?request_para
```

Example:

In the Collaborative Filter example in the ***GSQL Demo Examples*** document, the request

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/User?limit=3"
```

without postprocess formatting returns the following:

```
{"version":{"api":"v2","schema":0},"error":false,"message":"","results":[
```

On the other hand,

```
curl -X GET http://localhost:9000/graph/socialNet/vertices/User?limit=3 |
```

returns this much more readable output:

```
{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "v_id": "id2",
      "v_type": "User",
      "attributes": {}
    },
    {
      "v_id": "id5",
      "v_type": "User",
      "attributes": {}
    },
    {
      "v_id": "id3",
      "v_type": "User",
      "attributes": {}
    }
  ]
}
```

REST++ Authentication

The TigerGraph system administration can choose to enable user authentication for the REST++ endpoints. If authentication is not enabled, then 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. To see how to enable/disable REST++ authentication, see the document Managing User Privileges and Authentication.

The REST++ server implements OAuth 2.0-style authorization as follows: 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. Anyone who has this secret can invoke a special REST endpoint to generate authorization **tokens** (other pseudorandom strings). 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.

Requesting A Token with GET /requesttoken

A user must have a secret before they create a token. Secrets are generated in GSQL (see Managing User Privileges and Authentication). The special endpoint GET /requesttoken is used to create a token. The endpoint has two parameters:

- secret (required): the user's secret
- lifetime (optional): the lifetime for the token, in seconds. The default is one month, approximately 2.6 million seconds.

Example: REST++ Request to Generate a Token

```
curl -X GET 'localhost:9000/requesttoken?secret=jiokmfqqfu2f95qs6ug85o89rp
```

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>'
```

For example, if the token = 01234567abcdefgh01234567abcdefgh, then the collaborative filtering example shown above would be

```
curl -X GET -H "Authorization: Bearer 01234567abcdefgh01234567abcdefgh" "h
```

Refreshing Tokens

If the /requesttoken endpoint is used with the PUT command and the request token already exists, the given token will be refreshed.

```
curl -X PUT 'localhost:9000/requesttoken?secret=jiokmfqqfu2f95qs6ug85o89rp
```

Deleting Tokens

If the DELETE command is used, the given request token will be deleted.

```
curl -X DELETE 'localhost:9000/requesttoken?secret=jiokmfqqfu2f95qs6ug85o8
```

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.

The maximum size for a request body, including the payload file, is set by the system parameter nginx.client_max_body_size. The default value is 128 (in MB). To increase this limit to xxx MB, use the following gadmin command:

```
gadmin --set nginx.client_max_body_size xxx -f
```

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.

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>
```

You can specify the response size limit of a HTTP request with the following header:

```
curl -X <GET/POST> -H "RESPONSE-LIMIT: <size limit in byte>" '<request_URL
```

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"
}
```

Built-in Endpoints

System Utilities

GET /echo and POST /echo

These endpoints are simple diagnostic utilities which respond with the following message.

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.

GET /endpoints

This endpoint returns a list of the installed endpoints and their parameters. There are three types of endpoints, described in the table below.

Type	Description
builtin	preinstalled in the TigerGraph system
dynamic	generated when compiling GSQL queries
static	user-installed endpoints

To include one more more of the endpoint types in the output, include ***TypeName*=true** in the parameter query string for each type. For example,

"builtin=true&static=true" will include builtin and static endpoints. If no type parameters are provided, all endpoints are returned.

Example: Report on all three types of endpoints

```
curl -X GET "http://localhost:9000/endpoints?builtin=true&dynamic=true
```

Example: Report on all built-in endpoints

```
curl -X GET "http://localhost:9000/endpoints?builtin=true" | jq .
```

There are over a dozen built-in endpoints, and some have several parameters, so the formatted JSON output from the **builtin=true** option is over 300 lines long. It is listed in full in Appendix A. To illustrate the format, we show a small excerpt: the output for the **GET /echo** and **GET /endpoints** endpoint.

Subset of GET /endpoints output

```
    "GET /echo": null,
    "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"
            }
        }
    }
```


GET /statistics

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. The REST++ server maintains a truncated log of requests from the current time and backward for a system-configured **log_interval** . Only those endpoints which have completed or timed out during the requested number of **seconds** and are within the **log_interval will** be included in the statistics report. For example:

```
# The following example shows the case when there are two endpoints (/grap
curl -X GET "http://localhost:9000/statistics?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
    ]
  }
}
```

The statistics data are returned in JSON format. For each endpoint which has statistics data, we return the following items:

- 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. By default, segments 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].

Note: If there is no query sent in the past given seconds, a empty json will be returned.

GET /version

This endpoint returns the git versions of all components of the system. This can be useful information when requesting help from TigerGraph's support team.

```
curl -X GET "http://server_ip:9000/version"
{"error":"false", "message":"TigerGraph RESTPP:
 --- Version ---
product              poc4.4_base          7dd9c25dac6be25107aeb1d8c4041138
olgp                 4.4                  8eaa1b27724df53af8bb1536a132e53c
topology             4.4                  334a1a354b87c80dabd0b05b4b7b4647
gpe                  4.4                  fd45a03bde22aa08a0c6ff9bb8cab33a
gse                  4.4                  771e133b719eb037b7b990566f451939
third_party          4.4                  4bc14a5f4f89bbddcd54f242a29175a2
utility              4.4                  0ba01f7d2668bf5cdff8c0fd6a7faef3
realtime             4.4                  e86c9ee9df81b201777915ba5028e342
er                   4.4                  1c80659bee6f6bf1fb1b1559c03ce7ea
tut                  4.4                  2ebb709b95c7bc1404085a8b3e504779
glelib               4.4                  823d806167e84f0494bdcf117763df8b
bigtest              prod_master          9700dbfb596b3602ed772e2c9755ec0c
pta                  prod_master          82ceadbb246c2e5b36783cd029577ede
glive                prod_master          70e69c4339d981d802065ff2b6695545
gui                  4.4                  2d886ca4ac5d9bad6a83bb75de92839f
gvis                 prod_master          7f947364db0dac10decad21df70187fe
blue_features        4.4                  8a4f587ec3b58f0974bb067e4c8d2aac
blue_commons         4.4                  7158570b7fc76da4b50c05a7469f14b2
"}
```

GSQL Server Endpoints

It is also possible to send certain requests to the GSQL Server. There are two key differences:

- Requests are sent to port 14240 (HTTP hub server) instead of 9000 (dedicated RESTPP server).
- Username and password are required.

Get the Graph Schema: GET /gsql/schema

Syntax for GET /get/schema

```
curl --user username -X GET "http://server_ip:14240/gsqlserver/gsql/sc
```

In addition, there are required and optional parameters:

- **graph**: (REQUIRED) the name of the graph
- **type**: (OPTIONAL) the name of either a vertex type or an edge type

The output contains three top-level elements, GraphName, VertexTypes, and EdgeTypes. The following examples illustrate how to use /gsql/schema:

Get the Entire Graph Schema

```
curl --user tigergraph:tigergraph -X GET "localhost:14240/gsqlserver/gsql/
```

This endpoint returns all vertex and edge types within a graph, if no vertices or edges are specified. Here is an example of the results :

```
{
    "error": false,
    "message": "",
    "results": {
        "GraphName": "poc_graph",
        "VertexTypes": [{
            "Config": {
                "STATS": "OUTDEGREE_BY_EDGETYPE",
                "PRIMARY_ID_AS_ATTRIBUTE": false
            },
            "Attributes": [{
                "AttributeType": {
                    "Name": "STRING COMPRESS"
                },
                "IsPartOfCompositeKey": false,
                "PrimaryIdAsAttribute": false,
                "AttributeName": "id",
                "IsPrimaryKey": false
            }],
            "PrimaryId": {
                "AttributeType": {
                    "Name": "STRING"
                },
                "IsPartOfCompositeKey": false,
                "PrimaryIdAsAttribute": false,
                "AttributeName": "companyId",
                "IsPrimaryKey": false
            },
            "Name": "company"
        }, {
            "Config": {
                "STATS": "OUTDEGREE_BY_EDGETYPE",
                "PRIMARY_ID_AS_ATTRIBUTE": false
            },
            "Attributes": [{
                "AttributeType": {
                    "Name": "STRING COMPRESS"
                },
                "IsPartOfCompositeKey": false,
                "PrimaryIdAsAttribute": false,
                "AttributeName": "id",
                "IsPrimaryKey": false
            }],
            "PrimaryId": {
                "AttributeType": {
                    "Name": "STRING"
                },
                "IsPartOfCompositeKey": false,
```

```
                "PrimaryIdAsAttribute": false,
                "AttributeName": "companyId",
                "IsPrimaryKey": false
            },
            "Name": "member"
        }],
        "EdgeTypes": [{
            "IsDirected": false,
            "ToVertexTypeName": "member",
            "Config": {},
            "Attributes": [],
            "FromVertexTypeName": "company",
            "Name": "member_company"
        }]
    }
}
```


Get One Vertex Type

If a vertex type is specified, the only the description for that one vertex type is returned.

```
curl --user tigergraph:tigergraph "localhost:14240/gsql/schema?graph=poc_g
```

```
{
    "error": false,
    "message": "",
    "results": {
        "Config": {
            "STATS": "OUTDEGREE_BY_EDGETYPE",
            "PRIMARY_ID_AS_ATTRIBUTE": false
        },
        "Attributes": [{
            "AttributeType": {
                "Name": "STRING COMPRESS"
            },
            "IsPartOfCompositeKey": false,
            "PrimaryIdAsAttribute": false,
            "AttributeName": "id",
            "IsPrimaryKey": false
        }],
        "PrimaryId": {
            "AttributeType": {
                "Name": "STRING"
            },
            "IsPartOfCompositeKey": false,
            "PrimaryIdAsAttribute": false,
            "AttributeName": "companyId",
            "IsPrimaryKey": false
        },
        "Name": "company"
    }
}
```


Get One Edge Type

If an edge type is specified, the only the description for that one edge type is returned.

```
curl --user tigergraph:tigergraph "server_ip:14240/gsql/schema?graph=poc_g
```

```
{
    "error": false,
    "message": "",
    "results": {
        "IsDirected": false,
        "ToVertexTypeName": "member",
        "Config": {},
        "Attributes": [],
        "FromVertexTypeName": "company",
        "Name": "member_company"
    }
}
```


Accessing and Modifying the Graph Data

To support multiple graphs within one system, the graph data REST endpoint URLs include an optional graph name.

GET /graph/{graph_name}/vertices

Syntax for GET /graph/vertices

```
curl -X GET "http://server_ip:9000/graph/{graph_name}/vertices/{vertex
```

This endpoint returns all vertices having the type *vertex_type* in the graph called *graph_name* . *graph_name* is optional if the database has only one graph but required for a database with multiple graphs. Optionally, the user can instead chose a particular vertex by including its primary_id at the *vertex_id* field . For example:

```
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": {}
    }
  ]
}
```

/graph/{graph_name}/vertices has an optional parameter "count_only". The default value is false. If it is true, the results field contains only the number of vertices selected.

GET /graph/{graph_name}/edges

Syntax for GET /graph/edges

```
curl -X GET "http://localhost:9000/graph/{graph_name}/edges/{source_ve
```

This endpoint returns all edges which connect to a given vertex ID in the graph called *graph_name* . *graph_name* is optional if the database has only one graph but required for a database with multiple graphs.A source vertex ID must be given. The user may optionally specify the edge type, the target vertex type, and the target vertex ID. The URL format is as follows:

- *edge_type* - type name of the edges. Use "_" to permit any edge type. Omitting the *edge_type* field from the URL also permits any edge type. However, skipping *edge_type* also means that *target_vertex_type* and *target_vertex_id* must be skipped.
- *target_vertex_type* - type name of the target vertices.

- *target_vertex_id* - ID of the target vertex.

There are several optional parameters which may be used with either the GET or DELETE requests for /graph/ endpoints. See the Advanced Parameters section below.

The GET /graph/ endpoints can return at most 10240 items.

Here is a simple example:

```
curl -X GET "http://localhost:9000/graph/socialNet/edges/VidUser/0/User_Vi

{
  "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
      }
    },
    {
      "e_type": "User_Video",
      "directed": false,
      "from_id": "0",
      "from_type": "VidUser",
      "to_id": "3",
      "to_type": "Video",
      "attributes": {
        "rating": 10,
        "date_time": 0
      }
    }
  ]
}
```

/graph/{graph_name}/edges has two optional parameters "count_only" and "not_wildcard":

- **count_only** : If it is true, the results contains only the number of edges selected. The default value is false.
- **not_wildcard** : This determines how the edge type name "_" is interpreted. If false (which is the default), "_" means all edge types are included. If not_wildcard is true, "_" is interpreted literally to select only edges with edge type name equal to underscore.

DELETE /graph/{graph_name}/vertices

```
curl -X DELETE "http://server_ip:9000/graph/{graph_name}/vertices/{vertex_
```

This endpoint deletes the given vertex(vertices) in the graph called *graph_name* . *graph_name* is optional if the database has only one graph but required for a database with multiple graphs. The URL structure and semantics are analogous to those in GET /graph/{graph_name}/vertices. This endpoint has an additional parameter "permanent", whose default value is false. If "permanent" is true, the deleted vertex ids can never be inserted back, unless the graph is dropped or the graph store is cleared.

DELETE /graph/{/graph_name}/edges

```
curl -X DELETE "http://localhost:9000/graph/{graph_name}/edges/{source_ver
```

This endpoint deletes the given edge(s). *graph_name* is optional if the database has only one graph but required for a database with multiple graphs. The URL structure and semantics are analogous to those in GET /graph/{graph_name}/edges.

Advanced Parameters for /graph/{graph_name}/vertices and

/graph/{graph_name}/edges

The above four endpoints, GET /graph/{graph_name}/vertices, GET /graph/{graph_name}/edges, DELETE /graph/{graph_name}/vertices, and DELETE /graph/{graph_name}/edges, have optional URL parameters for further operations:

1. **Select** : Specify which attributes to be returned (GET only).
2. **Filter** : Apply a filter on the vertices or edges, based on their attribute values.
3. **Limit** : Limit the total number of vertices or edges.
4. **Sort** : Sort the data. (For DELETE, sort should be used with limit together.)
5. **Timeout** : Timeout in seconds. If set to 0, use system wide endpoint timeout setting.

⚠ The parameter 'Limit' can reduce the search space and leads to quick response of queries. However if Limit and Sort are both provided, the query still needs to traverse all potential vertices/edges and it might lead to slow query response on large graph.

Select

By default the GET /graph/{graph_name}/vertices and /graph/{graph_name}/edges endpoints return all the attributes of the selected vertices or edges. The select parameter can be used to specify either the desired or the undesired attributes. The format is select=attribute_list, where attribute_list is a list of comma-separated attributes. Listing an attribute name means that this attribute should be included, while an attribute name preceded by a minus sign means that this attribute should be excluded. An underscore means all attributes.

- `http://server_ip:9000/graph/{graph_name}/vertices?select=attr1,attr2` returns only attributes attr1 and attr2
- `http://server_ip:9000/graph/{graph_name}/vertices?select=-attr1,-attr2` returns all attributes except attributes attr1 andattr2
- `http://server_ip:9000/graph/{graph_name}/vertices?select=-_` returns no attribute at all

⚠ It is illegal to specify both desired and undesired attributes in the same request.

Example Query: Return the *date_time* attribute of all *product* vertices on socialNet.

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/product?select
```

Filter

The filter parameter is a set of conditions analogous to the WHERE clause in industry-standard SQL language. The format is filter=filter_list, where filter_list is a list of comma-separated filters, and each filter is the concatenation of an attribute, an operator, and a value (with no white spaces separating the parts). The following six comparison operators are supported:

1. `=` equal to
2. `!=` not equal to
3. `>` greater than
4. `>=` greater than or equal to
5. `<` less than
6. `<=` less than or equal to

Here is an example request: It returns all User vertices with age greater than or equal to 30.

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

⚠ Literal strings should be enclosed in double quotation marks. For example, `filter=name="GSQL"`. However, if the URL is itself enclosed in quotes, as is the case when a REST request is submitted using the `curl` command, then the quotation marks around a string should be URL-encoded by replacing each mark with substring `%22`.

Limit

The Limit parameter is used to set a limit on the number of vertices or edges returned from a query request. Note that there is also a system limit of 10240 on the number of vertices or edges returned. The user-defined limit cannot exceed this system limit.

The following example returns up to 3 User vertices on graph "socialNet".

```
curl -X GET "http://localhost:9000/graph/socialNet/vertices/User?limit=3"
```

Sort

The Sort parameter returns results sorted by given attributes. The format is sort=list_of_index_attributes. The results are sorted by the first attribute first, and so on. Groups of the sorted results which have identical values on the first attribute are then sorted by the second attribute, and so on. Below are some examples:

- `http://server_ip:9000/graph/{graph_name}/vertices?sort=attr1` sort by attribute attr1 in ascending order
- `http://server_ip:9000/graph/{graph_name}/vertices?sort=-attr1` sort by attribute attr1 in descending order
- `http://server_ip:9000/graph/{graph_name}/vertices?sort=attr1,-attr2` first sort by attr1 in ascending order, then sort by attr2 in descending order

DELETE /graph/{graph_name}/delete_by_type/vertices

Syntax for GET /graph/delete_by_type/vertices

```
curl -X DELETE "http://server_ip:9000/graph/{graph_name}/delete_by_typ
```

This endpoint deletes all vertices of the given vertex type in the graph called *graph_name* . *graph_name* is optional if the database has only one graph but required for a database with multiple graphs. This endpoint has two additional parameters "permanent" and "ack". The "permanent" parameter is the same as the "permanent" parameter for endpoint DELETE /graph/{graph_name}/vertices. "ack"

specifies whether RESTPP needs to get acknowledgement from GPEs. If "ack" is set to "none", it doesn't need to get acknowledgement from any GPE. If "ack" is set to "all" (default), it needs to get acknowledgement from all GPEs.

POST /builtins/{graph_name}

This endpoint provides statistics for the graph called *graph_name* . *graph_name* is optional if the database has only one graph but required for a database with multiple graphs. A JSON object must be given as a data payload in order to specify the function and parameters. In the JSON object, the keyword "function" is used to specify the function. Below are the descriptions of each function:

stat_vertex_attr

This function 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 bool attribute. There is one parameter:

- type: The vertex type name, or "*", which indicates all vertex types.

Below is an example request on socialNet and its output. The vertex type "Person" has a uint attribute "age".

```
curl -X POST 'http://localhost:9000/builtins/socialNet' -d  '{"function":'

{
  "version": {
      "api": "v2",
      "schema": 0
   },
  "error": false,
  "message": "",
  "results": [
    {
      "vertexName": "Person",
      "attributeStat": [
        {
          "vattrName": "age",
          "MAX": 64,
          "MIN": 15,
          "AVG": 36.5
        }
      ]
    }
  ]
}
```

stat_edge_attr

Similar to stat_vertex_attr, this function returns the statistics of the minimum, maximum, and average of the given edge type's int, uint, float and double attributes, and the count of true and false of a bool attribute. Note each undirected edge is counted twice. There are three parameters:

- type: The edge type name, or "*", which indicates all edge types.
- from_type: Given a vertex type, the function only includes edges whose source vertex type is the given type. "*" indicates all types. Default is all types. If a specific edge type is given, giving a correct from_type can speed up the process.
- to_type: Given a vertex type, the function only includes edges whose destination vertex type is the given type. "*" indicates all types. Default is all types.

Below is an example request and its output. The edge type "Liked" has a float attribute "strength".

```
curl -X POST 'http://localhost:9000/builtins/socialNet' -d  '{"function":'

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "Liked",
      "attributes": {
        "weight": {
          "MAX": 2.5,
          "MIN": 1,
          "AVG": 1.375
        }
      }
    }
  ]
}
```

stat_vertex_number

This function returns the number of vertices of the given vertex type. There is one parameter.

- type: The vertex type name, or "*", which indicates all vertex types.

Below is an example request and its output.

```
curl -X POST 'http://localhost:9000/builtins/socialNet' -d  '{"function":'

{
  "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
    }
  ]
}
```

stat_edge_number

This function returns the number of edges of the given type. There are three parameters.

- type: The edge type name, or "*", which indicates all edge types.
- from_type: Given a vertex type, the function only those edges whose source vertex type is the given type. "*" indicates all types. Default is all types. If a specific edge type is given, giving a correct from_type can speed up the process.
- to_type: Given a vertex type, the function counts only those edges whose destination vertex type is the given type. "*" indicates all types. Default is all types.

```
curl -X POST 'http://localhost:9000/builtins/socialNet' -d  '{"function":'

{
  "version": {
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "e_type": "Liked",
      "count": 4
    },
    {
      "e_type": "Liked_By",
      "count": 5
    },
    {
      "e_type": "Linkto",
      "count": 6
    },
    {
      "e_type": "Has_desc",
      "count": 40
    },
    {
      "e_type": "NameConn",
      "count": 38
    },
    {
      "e_type": "Video_AttributeTag",
      "count": 18
    },
    {
      "e_type": "User_Video",
      "count": 14
    }
  ]
}
```

POST /ddl/{graph_name} - Run a Loading Job

This endpoint is for loading data the the graph called *graph_name* . *graph_name* is optional if the database has only one graph but required for a database with multiple

graphs. For more details, please see GSQL Language Reference Part 1 - Defining Graphs and Loading Data

This endpoint 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. This endpoint accepts six parameters:

name	data type	default	description
tag	string	N.A.	loading job name defined in your DDL loading job (REQUIRED)
filename	string	N.A.	file variable name or filepath for the file containing the data (REQUIRED)
sep	one character string	,	separator of CSV data. If your data is JSON, you do not need to specify this parameter.
eol	one or two character string	\n	end-of-line character. Only one character is allowed, except for the special case "\r\n"
ack	string, can only be "all" or "none"	"all"	"all": request will return after all GPE instances have acknowledged the POST "none": request will return immediately after RESTPP processed the POST.
			Timeout in seconds. If set to 0, use

timeout	UINT32	0	system-wide endpoint timeout setting.

Note that if you have special characters in your parameter values, the special characters should use URL encoding. For example, if your eol is '\n', it should be encoded as %0A. Reference guides for URL encoding of special characters can found on the web, such ashttps://www.w3schools.com/tags/ref_urlencode.asp ↗. To avoid confusion about whether you should you one or two backslashes, we do not support backslash escape for this eol or sep parameter.

POST /graph/{graph_name} - Upsert the given data

This endpoint upserts vertices and/or edges into the graph called *graph_name*. *graph_name* is optional if the database has only one graph but required for a database with multiple graphs. "Upsert" means that if the target vertex or edge does not yet exist, it is insert; if it does exist, it is updated. The endpoint offers some parameters which can fine tune this behavior.

This endpoint accepts three parameters:

name	data type	default	description
ack	string, can only be "all" or "none"	"all"	"**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	bool	false	"**false**": Upsert vertices "**true**": Treat vertices as insert-only. If the input data refers to a vertex which

			already exists, do not update it.
vertex_must_exist	bool	false	"**false**": Always insert new edges. If a new edge refers to an endpoint vertex which does not exist, create the necessary vertices, using given id and default values for other parameters. "**true**": Only insert or update an edge If both endpoint vertices already exist.

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. The identifiers in **bold** are keywords. The italicized terms should be replaced with user-specified values. Moreover, multiple instances may be included at the italicized levels. See the example below for clarification.

Payload (Input) Data

The payload data should be in JSON according the schema shown below:

POST /graph JSON format

```
"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 the exact text which must be used. 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 attr1. Op codes are explained below.

Upsert Example Data 1: Two User vertices

```
{
 "vertices": {
    "User": {
      "id6": {
        "attr1": {
           "value": 1,
           "op": max
         }
      },
      "id1": {
        "attr1": {
           "value": 2
         }
      }
    }
  }
}
```

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 edges 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
              }
            }
          }
        }
      }
    }
  }
}
```

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 "|"	Update to the logical OR of the payload value and the existing value.
5	"max" or ">"	Update to maximum of the payload value and the existing value.
6	[illegible]	Update to minimum of the [illegible]

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 "" 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:

1. Invalid vertex type.
2. Invalid attribute data type.

- For edge upsert:

1. Invalid source vertex type.
2. Invalid edge type.
3. Invalid target vertex type.
4. 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

The following example file *add_id6.json* 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}
```

POST /gsqlserver/interpreted_query - Run an 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 request goes directly to the GSQL server (port 14240) instead of the 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:

```
curl --fail -u myUsername:myPassword -X POST "http://localhost:14240/gsqls
  INTERPRET QUERY (INT a) FOR GRAPH gsql_demo {
    PRINT a;
 }'
```

If you do not want the password to be included in the request, there are ways to have the username and password stored in a file and then to reference the file. The following method by Pierre D ↗ is published in https://stackoverflow.com/questions/2594880/using-curl-with-a-username-and-password ↗:

```
curl --netrc-file my-password-file http://example.com
```

The format of the password file is (as per `man curl`):

```
machine <example.com> login <username> password <password>
```

Note:

1. Machine name must *not* include `https://` or similar! Just the hostname.
2. The words '`machine`', '`login`', and '`password`' are just keywords; the actual information is the stuff after those keywords.

Managing Running Queries

Once a query has been started, it is assigned a request ID. There are built-in endpoints to report on all running queries and to abort queries.

Get Running Queries /showprocesslist/{graph_name}

This endpoint reports statistics of running queries: the query's request ID, start time, expiration time, and the REST endpoint's url.

List All Running Queries

```
curl -X GET "http://localhost:9000/showprocesslist" | 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"
}
```

Abort a Query /abortquery/{graph_name}

The /abortquery endpoint safely aborts the selected query or queries. Either the GET or POST method may be used.

Syntax for GET /abortquery

```
curl "http://server_ip:9000/abortquery/{graph_name}?{requestid=xxx}[&{
```

The following examples show how the options are used to specify which queries to abort.

Abort Query with Given RequestID

To abort one query request, use the requestid parameter to specify its request (obtained from /showprocesslist), e.g., `?requestid=16842763.RESTPP_1_1.1561401340785.N`

To abort multiple query requests, simply use the requestid parameter multiple times, e.g.,
`?requestid=16842763.RESTPP_1_1.1561401340785.N&requestid=16973833.RESTPP_1_1.1561401288421.N`

```
curl -X GET "localhost:9000/abortquery/poc_graph?requestid=16842763.RESTPP

{
  "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"
}
```

Abort All Queries

To abort all queries, instead of using specific requestids, use **requestid=all**.

```
curl -X GET "localhost:9000/abortquery/poc_graph?requestid=all"

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "aborted_queries": [
        {
          "requestid": "16777232.RESTPP_1_1.1561401584866.N",
          "url": "/sleepgpe?milliseconds=110000"
        }
      ]
    }
  ],
  "code": "REST-0000"
}
```

Abort Queries by Endpoint

To abort all queries of a given endpoint or endpoints, there is an option to input a string for the query's REST endpoint url. 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 `url=/sleepgpe.*`

```
curl -X GET "localhost:9000/abortquery/poc_graph?url=/sleepgpe.*"

{
  "version": {
    "edition": "enterprise",
    "api": "v2",
    "schema": 0
  },
  "error": false,
  "message": "",
  "results": [
    {
      "aborted_queries": [
        {
          "requestid": "16973836.RESTPP_1_1.1561401442432.N",
          "url": "/sleepgpe?milliseconds=110000"
        }
      ]
    }
  ],
  "code": "REST-0000"
}
```

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.

Name	Path Type
`/shortestpath`	shortest path between the source and target vertex sets
`/allpaths`	all possible paths up to a given maximum path length between the source and target vertex sets

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\"
```

key	type	value
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 theses vertices. Mutually exclusive with **target**.
vertexFilters	filter array	(OPTIONAL) Restrict the paths to those whose vertices satisfy any of the given filters. See details of filters above.
edgeFilters	filter array	(OPTIONAL) Restrict the paths to those whose edges satisfy any of the given filters. See details of filters above.

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.

POST /shortestpath/{graphName} (Shortest Path Search)

The `/shortestpath` endpoint has the following additional parameters.

Key	Type	Description
maxLength	integer	(OPTIONAL) Maximum length of a shortest path.

		Default is 6.
allShortestPaths	boolean	(OPTIONAL) If **true**, the endpoint will return all shortest paths between the source and target. Default is **false**, meaning that the endpoint will return only one path.

The example below requests a single shortest path between the source vertex set {VidUser 2} and the target vertex set {VidUsers 0 and 3}. The path contains edges of type `User_Video` whose attributes fulfill the predicate `rating > 5 and date_time > 1000.` The result is a single path consisting of 3 vertices and 2 edges: VidUser 0 -- (User_Video edge) -- Video 0 -- (User_Video edge) -- VidUser 3

```
curl -s -X POST 'http://localhost:9000/shortestpath' -d
'{
  "sources":[{"type":"VidUser","id":"2"}],
  "targets":[{"type":"VidUser","id":"0"}, {"type":"VidUser","id":"3"}],
  "edgeFilters":[{"type":"User_Video","condition":"rating > 5 and date_tir
  "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", "y
        { "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 }
        }
      ]
    }
  ]
}
```

To get all shortest paths between the above source and target vertex sets, we can simply do so by specifying the `allShortestPaths` parameter is true. Now we get two paths. The paths partially overlap, so we get a total of 4 vertices and 3 edges:
VidUser 0 -- (User_Video edge) -- Video 0 -- (User_Video edge) -- VidUser 3
VidUser 0 -- (User_Video edge) -- Video 0 -- (User_Video edge) -- VidUser 2

```
curl -s -X POST 'http://localhost:9000/shortestpath' -d
'{
  "sources":[{"type":"VidUser","id":"2"}],
  "targets":[{"type":"VidUser","id":"0"}, {"type":"VidUser","id":"3"}],
  "edgeFilters":[{"type":"User_Video","condition":"rating > 5 and date_tir
  "maxLength":4
  "allShortestPaths": true
}'

# 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 'vertex_filters' filters, use empty filter.",
  "results": [
    {
      "vertices": [
        { "v_id": "3","v_type": "VidUser","attributes": { "name": "Dale" }
        { "v_id": "2","v_type": "VidUser","attributes": { "name": "Chris"
        { "v_id": "0","v_type": "Video","attributes": { "name": "Solo", "y
        { "v_id": "0","v_type": "VidUser","attributes": { "name": "Angel"
      ],
      "edges": [
        {
          "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 }
        },
        {
          "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 }
        }
      ]
    }
  ]
}
```

POST /allpaths/{graphName} (All Paths Search)

The `/allpaths` endpoint has the following additional required parameter.

Key	Type	Description
maxLength	integer	(REQUIRED) 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.

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' -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", "y
        { "v_id": "2","v_type": "Video","attributes": { "name": "Thor", "y
        { "v_id": "4","v_type": "Video","attributes": { "name": "Ran", "ye
      ],
      "edges": [
        {
          "e_type": "Video_AttributeTag", "from_id": "0", "from_type": "Vi
          "to_id": "action", "to_type": "AttributeTag", "directed": false,
          "attributes": { "weight": 1, "date_time": 0 }
        },
        {
          "e_type": "Video_AttributeTag", "from_id": "4", "from_type": "Vi
          "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": "Vi
          "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 path finding algorithms are available in the GSQL Graph Algorithm Library.

Dynamically Generated Endpoints

Each time a new TigerGraph query is installed, a dynamic endpoint is generated and stored at installation_directory/config/endpoints_dynamic. This new endpoint enables the user to run the new TigerGraph query by using curl commands and giving the parameters in URL or in a data payload. See the document "GSQL Language Specification, Part 2: Queries" Section "Running a Query" for more details. For example, the following TigerGraph query can generate a corresponding endpoint in <installation_directory>/config/endpoints_dynamic:

parameterIsNULL.gsql

```
CREATE QUERY parameterIsNULL (INT p) FOR GRAPH anyGraph {
  IF p IS NULL THEN
    PRINT "p is null";
  ELSE
    PRINT "p is not null";
  END;

}
```

parameterIsNULL.end

```
{
   "query":{
      "parameterIsNULL":{
         "GET":{
            "function":"queryDispatcher",
            "action":"query",
            "target":"GPE",
            "payload":[
               {
                  "rule":"AS_QUERY_STRING"
               }
            ],
            "parameters":{
               "query":{
                  "type":"STRING",
                  "default":"parameterIsNULL"
               },
               "p":{
                  "type":"INT64",
                  "min_count":0
               }
            },
            "summary":"This is query entrance"
         }
      }
   }
}
```

The "payload" object enables the query being executed by giving a data payload. The "parameter" object includes the query parameters.

To execute this query, with parameter p=0, the following curl command can be used:

```
curl -X GET "http://localhost:9000/query/anyGraph/parameterIsNULL?p=0"
```

Log Files

The REST servers log files are located in <installation_directory>/logs.

JSON Catalog

The request

```
curl -X GET "http://server_ip:9000/endpoints?builtin=true"
```

generates the following output, appropriately 400 lines long when formatted. In addition to listing each endpoint, the JSON output also lists all the required and optional parameters for each endpoint. In turn, each parameter is described by some or all of these attributes:

- default
- max_count
- min_count
- type
- max_length
- is_id
- id_type

While this information alone is not sufficient for a full understanding of each endpoint, the descriptive names of parameters and the attribute values go a long way towards this goal.

```
{
   "DELETE /graph/{graph_name}/delete_by_type/vertices/{vertex_type}" : {
      "parameters" : {
         "ack" : {
            "default" : "all",
            "max_count" : 1,
            "min_count" : 1,
            "options" : [ "all", "none" ],
            "type" : "STRING"
         },
         "permanent" : {
            "default" : "false",
            "max_count" : 1,
            "min_count" : 1,
            "type" : "BOOL"
         },
         "vertex_type" : {
            "type" : "TYPENAME"
         }
      }
   },
   "DELETE /graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_
      "parameters" : {
         "edge_type" : {
            "max_count" : 1,
            "min_count" : 0,
            "type" : "STRING"
         },
         "filter" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "limit" : {
            "max_count" : 1,
            "min_count" : 0,
            "type" : "UINT64"
         },
         "not_wildcard" : {
            "max_count" : 1,
            "min_count" : 0,
            "type" : "BOOL"
         },
         "permanent" : {
            "default" : "false",
            "max_count" : 1,
            "min_count" : 1,
```

```
            "type" : "BOOL"
         },
         "select" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "sort" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "source_vertex_id" : {
            "id_type" : "$source_vertex_type",
            "is_id" : true,
            "max_count" : 1,
            "max_length" : 256,
            "min_count" : 1,
            "type" : "STRING"
         },
         "source_vertex_type" : {
            "max_count" : 1,
            "min_count" : 1,
            "type" : "TYPENAME"
         },
         "target_vertex_id" : {
            "id_type" : "$target_vertex_type",
            "is_id" : true,
            "max_count" : 1,
            "max_length" : 256,
            "min_count" : 0,
            "type" : "STRING"
         },
         "target_vertex_type" : {
            "max_count" : 1,
            "min_count" : 0,
            "type" : "TYPENAME"
         },
         "timeout" : {
            "default" : "0",
            "max_count" : 1,
            "min_count" : 0,
            "type" : "UINT32"
         }
      }
   },
   "DELETE /graph/{graph_name}/vertices/{vertex_type}/{vertex_id}" : {
```

```
      "parameters" : {
         "filter" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "limit" : {
            "max_count" : 1,
            "min_count" : 0,
            "type" : "UINT64"
         },
         "permanent" : {
            "default" : "false",
            "max_count" : 1,
            "min_count" : 1,
            "type" : "BOOL"
         },
         "sort" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "timeout" : {
            "default" : "0",
            "max_count" : 1,
            "min_count" : 0,
            "type" : "UINT32"
         },
         "vertex_id" : {
            "id_type" : "$vertex_type",
            "is_id" : true,
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "vertex_type" : {
            "type" : "TYPENAME"
         }
      }
   },
   "GET /echo" : {
      "parameters" : {
         "sleep" : {
            "default" : "0",
            "type" : "INT32"
```

```
            }
         }
      },
      "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"
            }
         }
      },
     "GET /graph/{graph_name}/edges/{source_vertex_type}/{source_vertex_id}/
         "parameters" : {
            "count_only" : {
               "default" : "false",
               "max_count" : 1,
               "min_count" : 0,
               "type" : "BOOL"
            },
            "edge_type" : {
               "max_count" : 1,
               "min_count" : 0,
               "type" : "STRING"
            },
            "filter" : {
               "max_count" : 1,
               "max_length" : 2560,
               "min_count" : 0,
               "type" : "STRING"
            },
            "limit" : {
               "max_count" : 1,
               "min_count" : 0,
               "type" : "UINT64"
            },
```

```
      "not_wildcard" : {
         "max_count" : 1,
         "min_count" : 0,
         "type" : "BOOL"
      },
      "select" : {
         "max_count" : 1,
         "max_length" : 2560,
         "min_count" : 0,
         "type" : "STRING"
      },
      "sort" : {
         "max_count" : 1,
         "max_length" : 2560,
         "min_count" : 0,
         "type" : "STRING"
      },
      "source_vertex_id" : {
         "id_type" : "$source_vertex_type",
         "is_id" : true,
         "max_count" : 1,
         "max_length" : 256,
         "min_count" : 1,
         "type" : "STRING"
      },
      "source_vertex_type" : {
         "max_count" : 1,
         "min_count" : 1,
         "type" : "TYPENAME"
      },
      "target_vertex_id" : {
         "id_type" : "$target_vertex_type",
         "is_id" : true,
         "max_count" : 1,
         "max_length" : 256,
         "min_count" : 0,
         "type" : "STRING"
      },
      "target_vertex_type" : {
         "max_count" : 1,
         "min_count" : 0,
         "type" : "TYPENAME"
      },
      "timeout" : {
         "default" : "0",
         "max_count" : 1,
         "min_count" : 0,
         "type" : "UINT32"
```

```
        }
      }
   },
   "GET /graph/{graph_name}/vertices/{vertex_type}/{vertex_id}" : {
      "parameters" : {
         "count_only" : {
            "default" : "false",
            "max_count" : 1,
            "min_count" : 0,
            "type" : "BOOL"
         },
         "filter" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "limit" : {
            "max_count" : 1,
            "min_count" : 0,
            "type" : "UINT64"
         },
         "select" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "sort" : {
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "timeout" : {
            "default" : "0",
            "max_count" : 1,
            "min_count" : 0,
            "type" : "UINT32"
         },
         "vertex_id" : {
            "id_type" : "$vertex_type",
            "is_id" : true,
            "max_count" : 1,
            "max_length" : 2560,
            "min_count" : 0,
            "type" : "STRING"
         },
         "vertex_type" : {
```

```
            "type" : "TYPENAME"
         }
      }
   },
   "GET /statistics" : {
      "parameters" : {
         "seconds" : {
            "default" : "10",
            "type" : "UINT32"
         },
         "segments" : {
            "default" : "10",
            "max" : "100",
            "min" : "1",
            "type" : "UINT32"
         }
      }
   },
   "GET /version" : null,
   "POST /builtins" : null,
   "POST /echo" : {
      "parameters" : {
         "sleep" : {
            "default" : "0",
            "type" : "INT32"
         }
      }
   },
   "POST /graph/{graph_name}" : {
      "parameters" : {
         "ack" : {
            "default" : "all",
            "max_count" : 1,
            "min_count" : 1,
            "options" : [ "all", "none" ],
            "type" : "STRING"
         }
      }
   }
}
```

Atomicity

Consistency

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

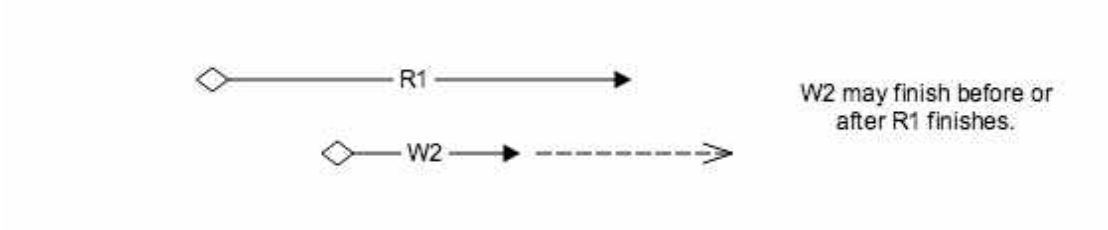
of the graph which has all the changes made by transactions committed before T1 was submitted but has no changes made by any transaction not yet committed when T1 was submitted. The version of the graph T1 is working on will not be changed by any transactions other than T1 , even if they commit before T1 is finished.

Example Scenarios

Let us examine a few transaction processing scenarios.

Scenario 1 Read - Write

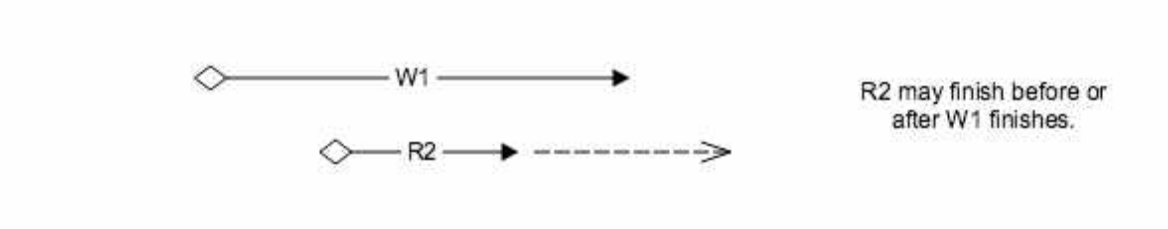
A read-only transaction R1 is running. Before R1 finishes, an update transaction W2 comes in. W2 might finish before R1 is finished. But R1 will not see the changes made by W2 before W2 is committed (no dirty reads). Even if W2 is committed before R1 is finished, if R1 reads the same part of the graph 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

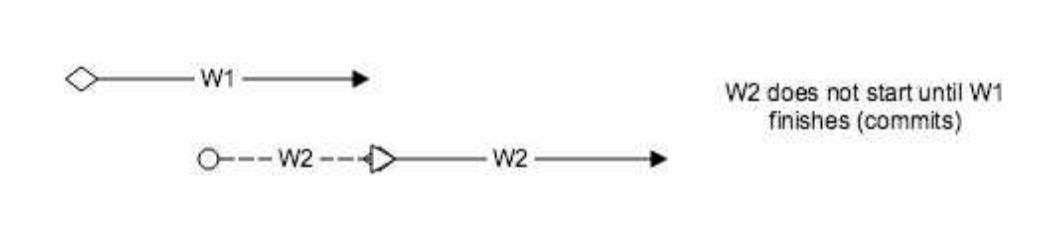
An update transaction W1 is running. Before W1 is committed, a read-only transaction R2 comes in. R2 will not wait for W1 to finish and will be executed as if there is no W1. Later. even if W1 finishes and commits before R2 is finished, R2 will not see any changes made by W1. This is because the graph 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.



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.

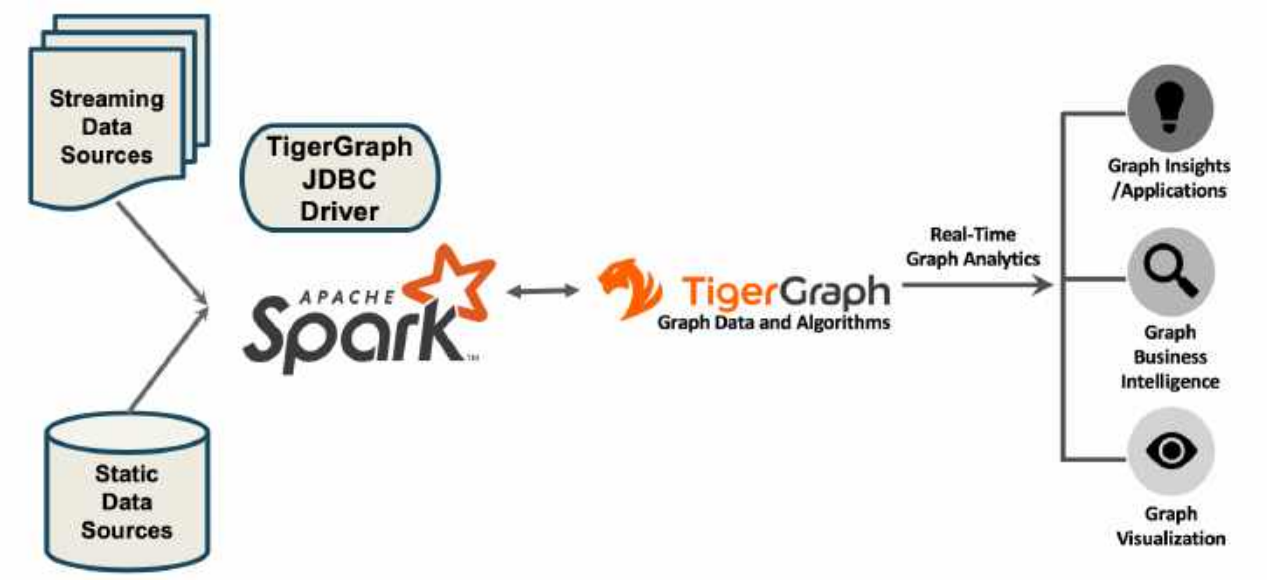
To configure a data source, see the appropriate data loader user guide:

- **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

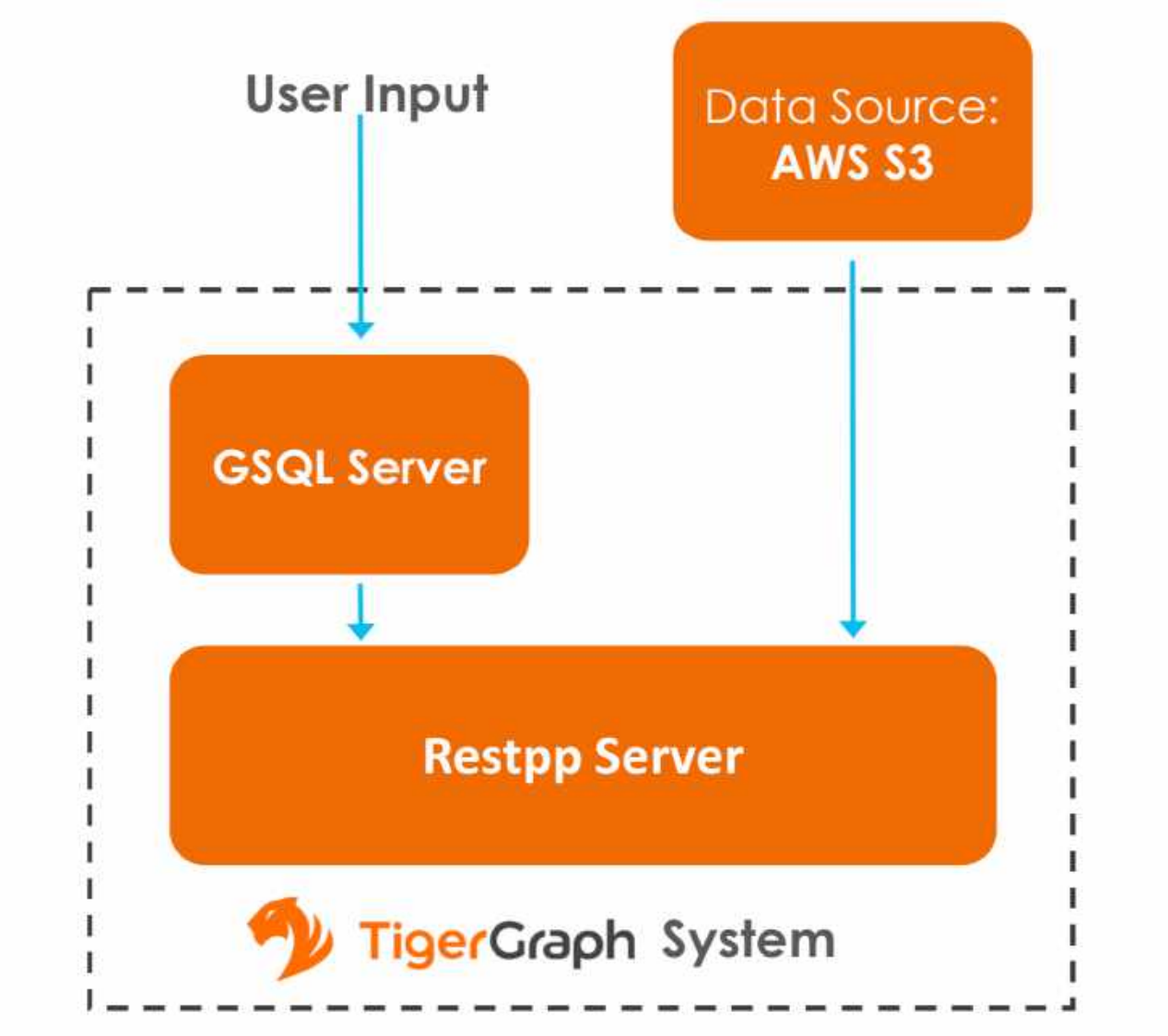
Overview

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*. A sample s3.config 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"
}
```

For simplicity, you can merge the CREATE DATA_SOURCE and SET statements:

```
CREATE DATA_SOURCE S3 data_source_name = "/path/to/s3.config"
```

1. 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.
2. Each time when the config file is updated, you must run "SET data_source_name" to update the data source details in the dictionary.

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 ***s*1**, 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
```

Kafka Loader User Guide

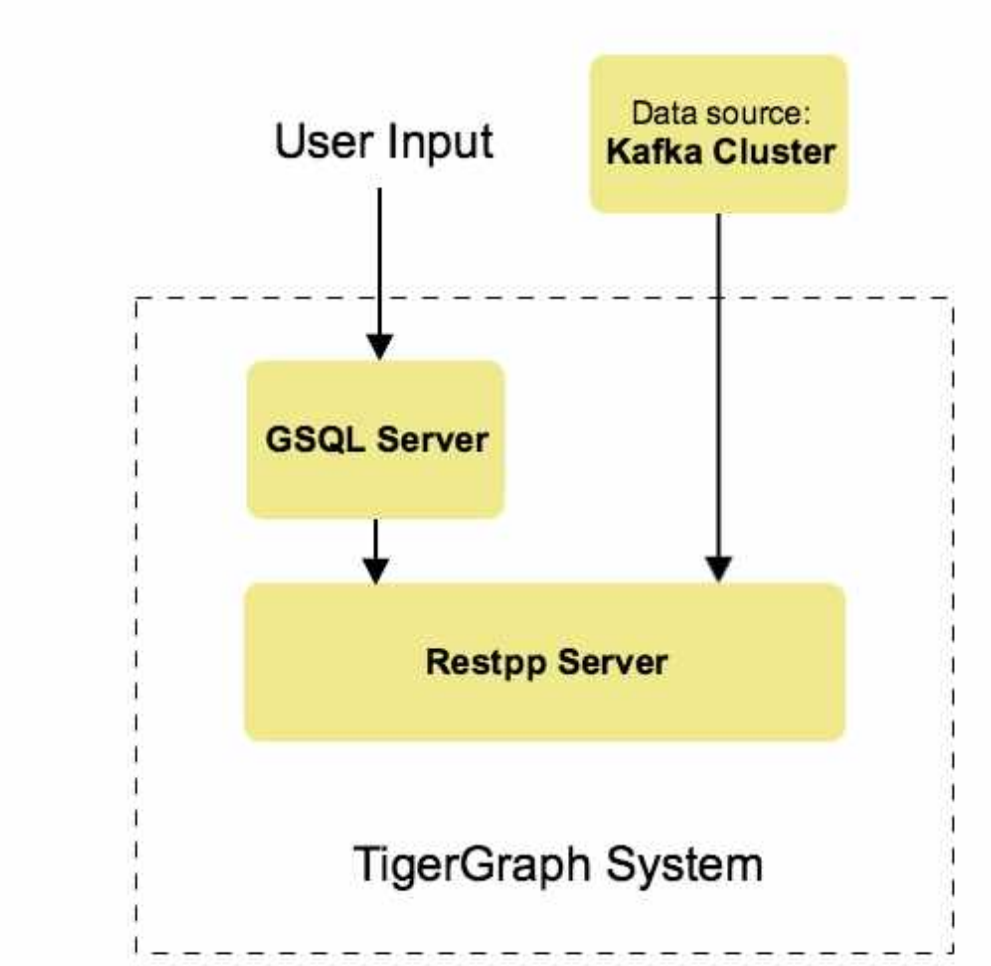
Overview

Kafka is a popular pub-sub system in enterprise IT, offering a distributed and fault-tolerant real-time data pipeline. TigerGraph's Kafka Loader feature lets you easily integrate with a Kafka cluster and speed up your real time data ingestion. It is easily extensible using the many plugins available in the Kafka ecosystem.

The Kafka Loader consumes data in a Kafka cluster and loads into TigerGraph system.

Architecture

From a high level, a user provides instructions to the TigerGraph system through GSQL, and the external Kafka cluster loads data into TigerGraph's RestPP server. The following diagram demonstrates the Kafka Loader data architecture.



Kafka Data Loader Architecture

Prerequisites

You should have a Kafka cluster configured and set up in your environment.

Once you have the external Kafka cluster setup, you need to prepare the following two configuration files and place them in your desired location in TigerGraph system:

1. Kafka data source configuration file: This file includes the external Kafka broker's domain name and port. Through the configuration file, TigerGraph system knows the location and port of the external Kafka broker. Please see an example in **Step 1. Define the Data Source**.
2. Kafka topic and partition configuration file: This file includes the Kafka topic, partition list, and start offset for the loading messages. Please see an example

in **Step 2. Create a Loading Job**.

Configuring and Using the Kafka 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 Kafka Loader is designed to be consistent with the existing GSQL loading syntax.

1. Define the Data Source

CREATE DATA_SOURCE

Before starting a Kafka data loading job, you need to define the Kafka server as a data source. The CREATE DATA_SOURCE statement defines a data_source variable with a subtype of KAFKA:

```
CREATE DATA_SOURCE KAFKA data_source_name
```

Kafka Data Source Configuration File

After the data source is created, then use the SET command to specify the path to a configuration file for that data source.

```
SET data_source_name = "/path/to/kafka.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 Kafka server's global settings, including the data source ip and port. A sample kafka.conf is shown in the following example:

```
{
  "broker": "broker.full.domain.name:9092",
}
```

The "broker" key is required. Additional Kafka configuration parameters may be provided (see Kafka documentation) by using the optional "kafka_config" key. For its value, provide a list of key-value pairs. For example:

```
{
  "broker": "broker.full.domain.name:9092",
  "kafka_config": {"group.id":"tigergraph"}
}
```

For simplicity, you can merge the CREATE DATA_SOURCE and SET statements:

```
CREATE DATA_SOURCE KAFKA data_source_name = "/path/to/kafka.config"
```

1. 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.
2. Each time when the config file is updated, you must run "SET data_source_name" to update the data source details in the dictionary.

ADVANCED: MultiGraph Support

The Kafka 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 it 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 KAFKA k1 = "/path/to/config"
```

2. A **superuser** may grant/revoke a data source to/from one or more graphs:

```
GRANT DATA_SOURCE k1 TO GRAPH graph1, graph2
REVOKE DATA_SOURCE k1 FROM GRAPH graph1, graph2
```

3. An **admin** user may create a local data source for a specified graph which they administer:

```
CREATE DATA_SOURCE KAFKA k1 = "/path/to/config" FOR GRAPH test_graph
```

> In the above statement, the local data_source k1 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 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 of several legal kafka data_source create and drop commands.

```
CREATE DATA_SOURCE KAFKA k1 = "/home/tigergraph/kafka.conf"
CREATE DATA_SOURCE KAFKA k2 = "/home/tigergraph/kafka2.conf"

DROP DATA_SOURCE k1, k2
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:
  - KAFKA k1 ("127.0.0.1:9092")
The global data source will be shown in global scope. The graph scope will
```

2. Create a Loading Job

The Kafka 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 Kafka 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 for use with the Kakfa Loader. In the syntax, $DATA_SOURCE_NAME is the Kafka data source name, and the path points to a configuration file with topic and partition information of the Kafka server. The Kafka 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 Kafka Data Source **k1**, where the path to the topic-partition configuration file is "~/topic_partition1.conf":

```
DEFINE FILENAME f1 = "$k1:~/topic_partition1.conf";
```

Kafka Topic-Partition Configuration File

The topic-partition configuration file tells the TigerGraph system exactly which Kafka records to read. Similar to the data source configuration file described above, the contents are in JSON object format. An example file is shown below:

topic_partition1.conf

```
{
  "topic": "topicName1",
  "partition_list": [
    {
      "start_offset": -1,
      "partition": 0
    },
    {
      "start_offset": -1,
      "partition": 1
    },
    {
      "start_offset": -1,
      "partition": 2
    }
  ]
}
```

The "topic" key is required. Optionally, a "partition_list" array can be included to specify which topic partitions to read and what start offsets to use. If the "partition_list" key is missing or empty, all partitions in this topic will be used for loading. The default offset for loading is "-1", which means you will load data from the most recent message in the topic, i.e., the end of the topic. If you want to load from the beginning of a topic, the "start_offset" value should be "-2".

You can also overwrite the default offset by setting "default_start_offset" in the Kafka topic configuration file. For example,

```
# all partition will be used if no "partition_list" item
{
  "topic": "topicName1"
}

# with empty "partition_list"
{
  "topic": "topicName1",
  "partition_list": []
}

# overwrite the default start offset
{
  "topic": "topicName1",
  "default_start_offset", 0
}
```

Instead of specifying the config file path, you can also directly provide the topic-partition configuration as a string argument, as shown below:

```
DEFINE FILENAME f1 = "$k1:~/topic_partition_config.json";
DEFINE FILENAME f1 = "$k1:{\"topic\":\"zzz\",\"default_start_offset\":2,\'
```

3. Run the Loading Job

The Kafka 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:topic_partition configure", which will override the value defined in the loading job. In the example below, the config files for f3 and f4 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, f3="$k1:~/topic_part3.config", f4="$k1:~/tc
```

A RUN LOADING JOB instance may only use one type of data source. E.g., you may not mix both Kafka 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 Kafka 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 current Kafka message.

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

Kafka 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, using 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 offset for each partition
2. average loading speed
3. loaded size
4. duration

See Inspecting and Managing Loading Jobs ↗ for more details.

Kafka Loader Example

Here is an example code for loading data through Kafka Loader:

```
USE GRAPH test_graph
DROP JOB load_person
DROP DATA_SOURCE k1

#create data_source kafka k1 = "kafka_config.json" for graph test_graph
CREATE DATA_SOURCE KAFKA k1 FOR GRAPH test_graph
SET k1 = "kafka_config.json"

# define the loading jobs
CREATE LOADING JOB load_person FOR GRAPH test_graph {
  DEFINE FILENAME f1 = "$k1:topic_partition_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
```

Legal

Patents and Third Party Software

Patent and Third Party Notice for TigerGraph Platform v2.4 April 2019

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

==========

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	Boost Software License 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 ↗
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 ↗
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 https://www.zlib.net/zlib_license.html ↗

Workshop

Connected Data London 2019

Graphs, Machine Learning, and Explainable AI

2MB

CDL19 - Graphs, ML, and Explainable AI.pdf
pdf

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 ↗