



Altair® Graph Lakehouse™ 2025.0 Documentation

Last Updated: 3/16/2025

Online documentation is available at docs.cambridgesemantics.com

Table of Contents

About This Doc	11
Best Practice	12
Graph Lakehouse Features and Benefits	13
Graph Lakehouse Architecture	16
Planning and Deployment Guidelines	19
Server and Cluster Requirements	20
Sizing Guidelines for In-Memory Storage	29
Securing a Graph Lakehouse Environment	38
Deploy	41
Container Image Deployments	42
Container Engine Requirements	43
Deploy the Graph Lakehouse Container Image	46
Kubernetes Deployments	56
Install the Kubernetes Command Line Client	57
Configure Access to a Kubernetes Cluster	58
Install Helm	60
Deploy Graph Lakehouse with Helm	61
Enterprise Linux 9 Deployments	65
Pre-Installation Requirements	66

Install Graph Lakehouse	69
Post-Installation Configuration	78
Uninstalling and Updating Graph Lakehouse	88
IBM Cloud Pak Deployments	91
Get Started	96
Quickstart with the Query Console	97
Quickstart with the CLI	103
Licensing Methods	105
Install or Upgrade a License	107
Learn SPARQL	117
SPARQL Query Basics	118
SELECT	119
CONSTRUCT	121
ASK	123
DESCRIBE	123
PREFIX Clause	125
FROM Clause	126
WHERE Clause	129
SPARQL Best Practices	134
SPARQL Tips and Tricks	137
Managing Your Data	137

Exploring Your Data	140
Understanding Your Data as a Graph	144
Sample Data and Tutorials	148
Working with SPARQL and the Tickit Data	149
Working with Cypher and the Movie Data	166
Load & Manage Data	172
Load RDF Data from Files	173
RDF Load File Requirements	174
Data Type Handling	177
Load RDF Files with the IO Load Service	179
Load Local RDF Files with SPARQL LOAD	183
Load or Virtualize Non-RDF Sources with SPARQL Queries	187
Introduction to the Graph Data Interface	188
GDI Concepts and Basic Usage	191
Getting Started with GDI Queries	191
Generating a Knowledge Graph	207
Reading Data Source Metadata	222
Pagination Options	239
Binding and Hierarchy Concepts	243
Incremental Load Concepts	251
Options for Data Types, Data Connections, and Models	256

Data Type Formatting Options	256
Model Normalization Options	259
Data Linking Options	268
Advanced Usage by Data Source Type	272
Query a Database Source	272
Query an HTTP Source	286
Query an Elasticsearch Source	307
Query a File Source	331
GDI Property Reference	385
Universal Properties	385
DbSource Properties	390
FileSource Properties	393
ElasticSource Properties	398
Use a Query Context	401
Create a Labeled Property Graph (RDF-star)	405
Defining Properties in Turtle Load Files	406
Defining Properties in INSERT Queries	408
Querying Property Graphs	412
Return Edges and Vertexes as JSON Objects	414
Infer New Data (RDFS+ Inferencing)	424
RDFS-Plus Rules	426

Validate Data with SHACL (Preview)	436
Introduction to SHACL	437
Constraint Component Reference	438
Create a Shapes Graph	453
Validate a Data Graph	458
Copy Graphs to Files	465
Schedule Automated Data Updates	468
Access & Analyze Data	478
Use the Query & Admin Console	479
Use the Graph Lakehouse CLI	493
Use Third-Party Visualization Tools	498
Access the SPARQL and RDF Endpoints	509
Access Data with OData Protocol	518
Create a Data on Demand Endpoint	519
Access a Data on Demand Endpoint	522
OData Reference	529
Create and Save Views	537
Create and Save a View for Reuse	538
Create a View Inline for One-Time Use	539
WITH Syntax	539
Examples	541

Save Queries for Reuse	544
Create and Save a Query for Reuse	545
Create a Query Inline for One-Time Use	547
WITH Syntax	547
Examples	549
SPARQL Query Language Reference	552
Built-in Functions	553
Aggregate Functions	553
Casting Functions	568
Date and Time Functions	579
Graph Algorithms	597
Hash Functions	614
Informational or Testing Functions	619
Logical Functions	625
Math Functions	636
Property Paths	657
String Functions	658
Update Functions	681
Window Aggregate and Ranking Functions	686
Advanced Grouping Sets	702
Extension Libraries	704

Apache Arrow Library	704
Data Science Library	715
Geospatial Library	742
Matrix Utilities Library	890
Sketch Library	939
Utilities Library	956
Cypher Query Language Reference	964
Cypher Language Overview	965
Cypher Patterns	971
Cypher Types, Lists, and Maps	976
Comparability, Equality, Orderability, and Equivalence	983
Cypher Expressions, Variables, and Parameters	990
Cypher Operators	993
Cypher Clauses	997
Cypher Functions	1013
Admin	1020
Start and Stop Graph Lakehouse	1021
Deploy the Frontend Container	1023
Authentication and Access Control	1031
Access Control Basics and Terminology	1032
Configure Graph Lakehouse for LDAP Authentication	1046

Create and Manage Roles from the Console	1050
Monitor Access Control Activity	1061
Manage the Server Configuration	1064
System Settings Reference	1065
Change System Settings	1079
Manage File Access Policies	1080
Ignore Missing Graphs and Unbound Variables in Queries	1084
Change the Default FROM Clause Behavior	1085
Relocate Graph Lakehouse Directories	1086
Manage Automatic Database Restart Options	1087
Develop	1092
UDX Terminology and Concepts	1093
Developing User-Defined Extensions	1102
UDX Development Process Overview	1103
Reviewing UDX Interface Files	1105
Creating New UDX Library Source Files	1116
Registering a UDX in an Extension Library	1121
Compiling UDX Source Files	1126
Loading a UDX to the Database	1129
Using Extensions in SPARQL Queries	1130
UDX Examples	1131

User-Defined Function (UDF) Examples	1132
User-Defined Aggregate (UDA) Examples	1138
FAQ & Troubleshooting	1150
FAQ	1151
Error Message Reference	1160
Retrieving Diagnostic Files	1162
Getting Support	1168

About This Doc

This document contains deployment instructions, best practices, usage, administration, and troubleshooting documentation for Graph Lakehouse for Developers 2025.0. It is a PDF version of the content that is available at <https://docs.cambridgesemantics.com/anzograph/v3.1/userdoc>.

This document is intended for application developers who have a Graph Lakehouse license for standalone use without Graph Studio™. If you use Graph Studio, refer to the [Graph Studio Documentation](#) for all Graph Studio and Graph Lakehouse / Anzo & AnzoGraph usage information.

Best Practice

Graph Lakehouse is a high performance graph OLAP database that lets users perform BI-style analytics with unparalleled speed and scalability. Graph Lakehouse supports parallel loading of data so users can begin performing analytics on data quickly. Graph Lakehouse uses standards from the W3C regarding RDF data formats and the SPARQL query language.

Graph Lakehouse can be deployed in cloud environments such as Amazon AWS, Google Cloud, Microsoft Azure, and IBM Cloud Pak, or on-premises on Linux bare metal and virtual machines. Graph Lakehouse also supports Docker and Kubernetes deployments with data staged locally or in shared NFS, HDFS, or object storage. This section provides an overview of Graph Lakehouse features and architecture.

In this section:

- [Graph Lakehouse Features and Benefits](#) 13
- [Graph Lakehouse Architecture](#) 16
- [Planning and Deployment Guidelines](#) 19
- [Securing a Graph Lakehouse Environment](#) 38

Graph Lakehouse Features and Benefits

Graph Lakehouse is a native, massively parallel processing (MPP) graph OLAP database, built to deliver hyperfast advanced analytics at big data scale. This topic provides details about the key Graph Lakehouse features and the benefits that they provide.

- [Native Graph Database](#)
- [Massively Parallel Processing](#)
- [Performance at Scale](#)
- [Graph OLAP Technology and Multi-Graph Support](#)
- [Standards-Based Query Languages and Protocols](#)
- [Advanced Analytics](#)
- [Flexible and Schema-less Data Loading](#)

Native Graph Database

Graph Lakehouse is built to handle graph workloads throughout the computing stack, from the query language to the database and memory management engine, and the file system. Data is stored in native graph format whether it is on disk or in memory. Graph Lakehouse's use of the organic graph model avoids the overhead that non-native graph databases employ for simulating graph traversal and reformatting data on disk. Graph Lakehouse processes queries faster, scales better, and runs efficiently on hardware, virtual, or cloud platforms.

Massively Parallel Processing

Graph Lakehouse is a massively parallel processing (MPP) graph database. Its compressed in-memory and on disk data storage and MPP design provides extremely fast data loading, real-time updates, and interactive analytics on huge amounts of data. For more information, see [Graph Lakehouse Architecture](#).

Performance at Scale

Graph Lakehouse scales with your needs by distributing graph data across cluster nodes and processing queries in parallel on all nodes. Because of Graph Lakehouse's MPP and fast intra-cluster network implementation, load and query performance increases as the data and cluster size grow.

Graph OLAP Technology and Multi-Graph Support

Unlike transaction-oriented graph databases, Graph Lakehouse is a modern enterprise Graph Online Analytics Processing (GOLAP) database that enables users to interactively view, analyze, and update graph data. Graph Lakehouse provides unmatched analytic processing of complex queries that require many joins, filters, and aggregation. Graph Lakehouse enables data scientists, data architects, and application developers to deliver supercharged analytic insights at massive scale to support vital real-time solutions for detecting fraud, ensuring compliance, optimizing supply chains, building enterprise knowledge bases, and more. In accordance with the RDF/SPARQL standard, Graph Lakehouse has robust multi-graph support.

Standards-Based Query Languages and Protocols

Graph Lakehouse adheres to the W3C RDF and SPARQL 1.1 standards and offers the standard SPARQL 1.1 and RDF Graph Store Protocol on HTTP/S for sending and receiving SPARQL queries between client applications and the database. Graph Lakehouse also supports the industry standard CSV and RDF load file formats. Developers and analysts do not need to learn a proprietary query language to work with Graph Lakehouse and can incorporate Graph Lakehouse into their existing infrastructure of products that support standard graph APIs, such as data preparation, graph transaction processing, visualization, business intelligence, and machine learning tools.

In addition to SPARQL, Graph Lakehouse provides Cypher query language support. Graph Lakehouse supports the Bolt protocol to provide a Cypher-based CLI from which users can directly execute Cypher statements. Other Cypher applications that use the Bolt protocol can also execute either Cypher or SPARQL queries against Graph Lakehouse data. For more information on Graph Lakehouse Cypher language support, see [Cypher Query Language Reference](#).

Advanced Analytics

Graph Lakehouse extends the SPARQL 1.1 specification to add support for advanced analytics such as window aggregates and advanced grouping capabilities. Graph Lakehouse also supports conditional expressions, named queries and views, inferencing (RDFS+), labeled property graphs (using the W3C RDF-star proposed standard), and graph algorithms. In addition, Graph Lakehouse provides pre-built extension libraries that you can also use in the same way as other native, built-in analytic functions. For more information about using built-in analytics and extensions, see [SPARQL Query Language Reference](#).

In addition to supporting all standard SPARQL functions, Graph Lakehouse includes a rich library of SQL and Microsoft Excel-like built-in functions as well as both C++ and Java APIs for creating user-defined or custom extension functions, aggregates, and services. For more information about the extensions, see [Develop](#).

Flexible and Schema-less Data Loading

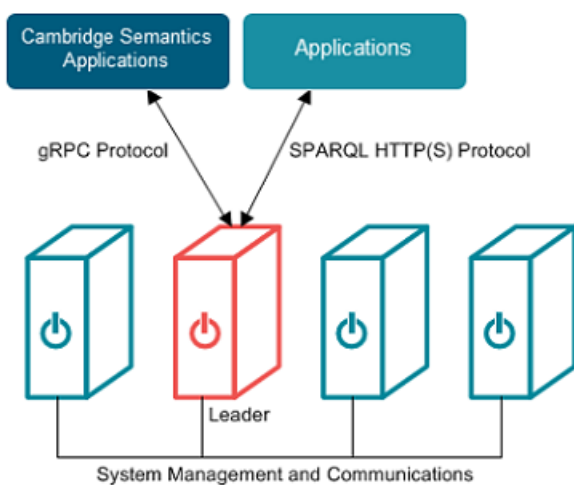
Loading data to Graph Lakehouse does not require maintenance of error-prone and time-consuming ETL pipelines, rigid schemas, or relational database models. And Graph Lakehouse's virtually unlimited capacity and real-time performance enables users to load structured, unstructured, internal, or external data on-demand, bringing immediate access and analysis to everyone. For more information, see [Load & Manage Data](#).

Graph Lakehouse provides a number of different data samples, tutorials, and notebooks to help you get started quickly using Graph Lakehouse and also familiarize you with the various operations you can perform. For more information, see [Get Started](#).

Graph Lakehouse Architecture

Graph Lakehouse uses massively parallel processing (MPP) to perform analytic operations on graph data conforming to RDF and RDF* standards. You can scale Graph Lakehouse to run in environments ranging from a single server to multiple servers in a cluster, in either on-premises or cloud environments.

Though all servers in an Graph Lakehouse cluster store the system metadata and have the ability to perform leader operations, one server acts as the leader for the cluster. All client applications should connect to this server.



In-Memory Data Storage Architecture

To provide the highest performance possible, Graph Lakehouse stores all graph data and performs all analytic operations entirely in memory. At startup, Graph Lakehouse sets the number of shards (called "slices" in Graph Lakehouse) per node to the number of cores on a single server. To utilize massively parallel processing of queries, Graph Lakehouse distributes (as evenly as possible) the data into memory across all of the slices. When data is loaded, Graph Lakehouse hashes on subjects to determine how the data is distributed. Distributing on subject allows the database to avoid distributing data over the network under certain conditions. Every slice contains several blocks that store the triples.

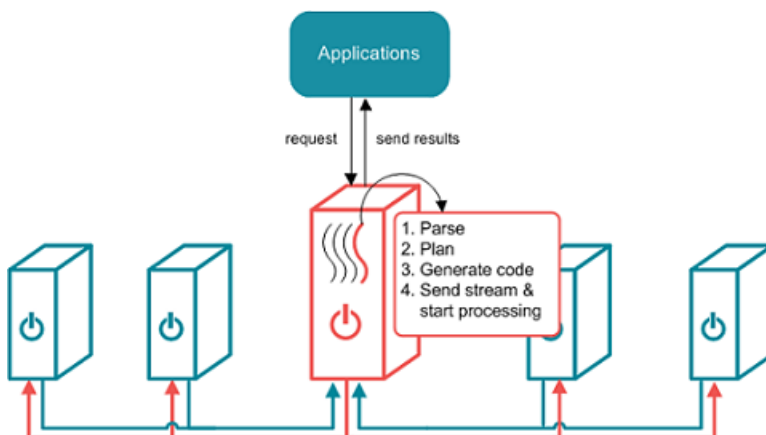


Note

When installed in a cluster, Graph Lakehouse requires that all servers provide the same equivalent hardware and quality of service.

Leader and Query Processing

When an application sends a request, the leader node dedicates a thread to process the request. All other threads remain ready for subsequent requests. The leader routes the query through parsing and planning. The planner determines the steps that the query requires, for example, whether a hash join, merge join, or an aggregation step is needed. The planner passes the final query plan to the code generator, which assembles the groups of steps into segments. The code generator then packages all of the segments for the query into a stream. The leader sends the stream to all of the nodes in the cluster and to its own slices. The nodes process the stream in parallel; each node dedicates a thread to process each segment. The nodes then return the results to the leader to send to the application.



For information about server requirements and recommendations, see [Planning and Deployment Guidelines](#).

Planning and Deployment Guidelines

This section provides information on choosing an appropriate system design and configuration for deploying Graph Lakehouse. It also helps you determine the optimum amount of memory needed to handle the requirements of analytic workloads of various sizes and details other best practices in planning, designing, and implementing solutions using Graph Lakehouse.

The main selection criteria for choosing a system design, as well as the associated server sizing and scaling, starts with the analytic applications you intend to run and their importance to your business. You also need to consider the sources and volume of data you plan to analyze as well as your performance requirements.

In this section:

Server and Cluster Requirements	20
Sizing Guidelines for In-Memory Storage	29

Server and Cluster Requirements

Before installing and configuring Graph Lakehouse, it is important to determine the appropriate size and scale of the environment, whether you are installing on "bare metal" servers, virtual machines, or using a cloud service. This topic details the minimum requirements and recommendations to follow for setting up Graph Lakehouse host servers and cluster environments.

- [Hardware Requirements](#)
- [Software Requirements](#)
- [Firewall Requirements](#)
- [Virtual Environments and Cluster Configuration](#)

Hardware Requirements

Altair lists above average production system hardware requirements as a guideline. Large production data sets running interactive queries may require significantly more powerful hardware and RAM configurations. Provision production server hardware accordingly to avoid performance issues.

Component	Minimum	Recommended	Guidelines
Available RAM	16 GB (for small-scale testing only)	200+ GB	Graph Lakehouse needs enough RAM to store data, intermediate query results, and run the server processes. Altair recommends that you allocate 3 to 4 times as much RAM as the planned data size. Do not overcommit RAM on a VM or on the hypervisor/container host. Avoid memory paging to disk (swapping) to achieve the highest possible level of performance. For more information about determining the server and cluster size that is ideal for hosting Graph Lakehouse, see Sizing Guidelines for In-

Component	Minimum	Recommended	Guidelines
Memory Storage.			
Disk space & Type	40 GB HDD	200+ GB SSD	Graph Lakehouse requires 30 GB for internal requirements. The amount of additional disk space required for load file staging and graph data persistence depends on the size of the data to be loaded. For persistence, Altair recommends that you have twice as much disk space on the local Graph Lakehouse file system as RAM on the server.
CPU	2	32 or 64	Once you provision sufficient RAM and a high-performing I/O subsystem, performance depends on raw CPU capabilities. Always use multi-core CPUs. A greater number of cores can make a dramatic difference in the performance of interactive queries. Note Intel x86-64 processors are recommended, but Graph Lakehouse is supported on Epyc and later generation AMD processors. Graph Lakehouse does not run on Opteron AMD processors or Mac ARM-based processors.

Component	Minimum	Recommended	Guidelines
Networking	10gbE	20+gbE	Not applicable for single server installations. Since Graph Lakehouse is high performance, massively parallel processing (MPP) graph OLAP engine, inter-cluster communications bandwidth dramatically affects performance. Graph Lakehouse clusters require optimal network bandwidth. Important All servers in a cluster must be in the same network. Make sure that all instances are in the same VLAN, security group, or placement group. In a switched network, make sure that all NICs link to the same Top Of Rack or Full-Crossbar Modular switch. If possible, enable SR-IOV and other HW acceleration methods and dedicated layer 2 networking that guarantees bandwidth.

Graph Lakehouse requires that all elements of the infrastructure provide the same quality of service (QoS). Do not run Graph Lakehouse on the same server as any other software except when in single-server mode and with an expectation of lowered performance. Providing the same QoS is especially important when using Graph Lakehouse in a clustered configuration. If any of the servers in the cluster perform additional processing, the cluster becomes unbalanced and may perform

poorly. A single poor performing server degrades the other servers to the same performance level. **All nodes require the same hardware specification and configuration.** Also use static IP addresses or make sure that DHCP leases are persistent.

To ensure the maximum and most reliable QoS for CPU, memory, and network bandwidth, do not co-locate other virtual machines or containers (such as Docker containers) on the same hypervisor or container host. For hypervisor-managed VMs, configure the hypervisor to reserve the available memory for the Graph Lakehouse server. For clusters, make sure there is enough physical RAM to support all of the Graph Lakehouse servers, and reserve the memory via the hypervisor.

In addition, running memory compacting services such as Kernel Same-page Merging (KSM) impacts CPU QoS significantly and does not benefit Graph Lakehouse. Live migrations also impact the performance of VMs while they get migrated. While live migration can provide value for planned host maintenance, Graph Lakehouse performance may be impacted if live migrations occur frequently. For more information about Kernel Same-page Merging, see https://en.wikipedia.org/wiki/Kernel_same-page_merging.

Note

Advanced configurations may benefit from CPU pinning on the hypervisor host and disabling CPU hyper-threading. For more information about CPU pinning, see https://en.wikipedia.org/wiki/Processor_affinity. For information about hyper-threading, see <https://en.wikipedia.org/wiki/Hyper-threading>.

Altair can provide benchmarks to establish relative cluster performance metrics and validate the environment.

Software Requirements

The table below lists the software requirements for Graph Lakehouse host servers.

Note

For container deployments, the required software and tuning is included in the Graph Lakehouse images. For RHEL/Rocky deployments, [Pre-Installation Requirements](#) provides details about configuring the required software on single server and cluster deployments.

Component	Requirement	Description
Operating System	RHEL/Rocky Linux 9.3+	Graph Lakehouse is not supported on Enterprise Linux 7 or 8.
Glibc-devel Library	Installed on all host servers	For compiling queries, Graph Lakehouse requires the latest version of the <code>glibc-devel</code> library for your operating system.
GNU binutils	Installed on all host servers	To compile and link programs, Graph Lakehouse requires the latest version of the <code>binutils</code> package for your operating system.
OpenJDK or GraalVM	Version 21 Installed on all host servers	Graph Lakehouse uses a Java client interface to access data sources. A Java 21 environment is required for using the Java client. Graph Lakehouse supports OpenJDK 21 and GraalVM 21.

Optional Software

Program	Description
vim	Editor for creating or changing files.
sudo	Enables users to run programs with alternate security privileges.
net-tools	Networking utilities.

Program	Description
psutils	Python system and process utilities for retrieving information on running processes and system usage.
tuned	Linux system service to apply tunables.
wget	Utility for downloading files over a network.

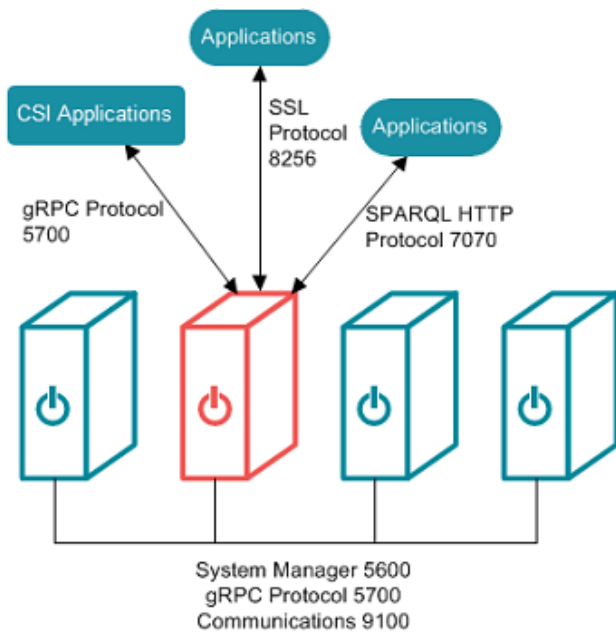
Firewall Requirements

Graph Lakehouse servers communicate via TCP/IP sockets. Client applications connect to Graph Lakehouse using the standard SPARQL HTTP(S) protocol. Altair applications, such as the Graph Lakehouse front end, communicate with the database via the secure, encrypted, gRPC-based protocol.

Important

For Graph Lakehouse clusters, all servers in the cluster must be in the same network. Make sure that all instances are in the same VLAN, security group, or placement group.

Open the TCP ports listed in the table below. This image shows a visual representation of the communication ports:



Port	Description	Required Access
5700	gRPC protocol port for secure communication between Graph Lakehouse servers.	The following list describes the access that is needed for port 5700: - Between all Graph Lakehouse servers in the cluster. - Available for Graph Lakehouse on single node installations. - Between the Graph Lakehouse leader server and any applications that connect to Graph Lakehouse using gRPC protocol, such as Apache Zeppelin and the Graph Lakehouse console. Note Make sure that the Linux environment variables http_proxy and https_proxy are not set on the servers. The Graph Lakehouse gRPC protocol cannot make connections to the database when proxies are enabled.

Port	Description	Required Access
8256	SPARQL HTTPS port for SSL communication between applications and Graph Lakehouse.	The following list describes the access that is needed for port 8256: • Between applications and the Graph Lakehouse leader server. • Between all Graph Lakehouse servers in a cluster. • Available for Graph Lakehouse on single node installations.
7070	Optional SPARQL HTTP port for communication between applications and Graph Lakehouse.	The following list describes the access that is needed for port 7070 if you have applications that will access Graph Lakehouse over HTTP: • Between applications and the Graph Lakehouse leader server. • Between all Graph Lakehouse servers in a cluster. • Available for Graph Lakehouse on single node installations.
9100	The internal fabric communications port.	The following list describes the access that is needed for port 9100: • Between all Graph Lakehouse servers in a cluster. • Available for Graph Lakehouse on single node installations.
5600	The SSL system management port.	The following list describes the access that is needed for port 5600: • Between all Graph Lakehouse servers in a cluster. • Available for Graph Lakehouse on single node

Port	Description	Required Access
installations.		

Virtual Environments and Cluster Configuration

When your data loading and performance requirements warrant using a server cluster, the minimal size cluster you create should have no fewer than four nodes. When using a single node, data gets redistributed in memory without using a network. If you add one or two more nodes to create a two- or three-node cluster, data then gets distributed over the network to utilize the additional CPUs in the cluster. However, the CPU gain you would get from the additional one or two nodes added to the cluster does not outweigh the performance degradation that the network introduces.

Once you have created and are using a cluster, you can always provision additional servers to add CPU and memory capacity to boost performance and increase the total amount of memory available to load graph data. Graph Lakehouse requires that all elements the infrastructure provide the same quality of service. Do not run Graph Lakehouse on the same server as any other application software.

Sizing Guidelines for In-Memory Storage

This topic provides guidance on determining the server and cluster size that is ideal for hosting Graph Lakehouse, depending on the characteristics of your data and Graph Lakehouse use case.

- [Memory and Cluster Size Guidelines](#)
- [Analyzing Data Characteristics in Load Files](#)
- [Estimating Memory Requirements Based on Data Characteristics](#)

Memory and Cluster Size Guidelines

Since Graph Lakehouse is a high-performance, in-memory database, it is important to consider the amount of memory needed to store the data that you plan to load. Estimating the amount of memory your workload requires can help you decide what size server to use and whether to use multiple servers. The sections below describe the key points to consider about memory usage and Graph Lakehouse.

- [Data at rest should use less than 50% of the total memory](#)
- [Graph Lakehouse reserves 20% of the memory for the OS](#)
- [Memory usage depends on data characteristics](#)
- [Memory usage can be high during loads](#)
- [Memory Usage Examples](#)

Data at rest should use less than 50% of the total memory

The data loaded into memory should not consume more than 50% of the total available memory on the instance or across a cluster. Preserve at least 50% of the memory for server processes, query processing, and storing intermediate results.

Note

Altair recommends that you allocate 3 to 4 times as much RAM as the planned data size, especially if the planned workload includes running complex analytic queries. There is no hard-wired limit on the number of queries you can run concurrently, however, you can set a limit, configured by the [user_queues](#) setting, that determines how many queries may be started before additional queries are placed in a queue.

Graph Lakehouse reserves 20% of the memory for the OS

To avoid unexpected shutdowns by the Linux operating system, the default Graph Lakehouse configuration leaves 20% of memory available for the OS; Graph Lakehouse will not use more than 80% of the total available memory. Account for this memory buffer in sizing calculations.

Memory usage depends on data characteristics

Memory usage varies significantly depending on the makeup of the data, such as the data types and sizes of literal values, and the complexity of the queries that you run. Data is loaded into Graph Lakehouse as triples, and the storage required for each triple ranges anywhere from 12 bytes per triple to 1 megabyte, for a triple that stores pages of text from an unstructured document.

- Triples with integer objects like the following example require about 16 bytes to store in memory.

```
<http://anzograph.com/resource/person1> <http://anzograph.com/resource/age> 50
```

- Triples made up of URIs like the following example require about 18 bytes to store in memory.

```
<http://anzograph.com/resource/person1> <http://anzograph.com/resource/friend>  
<http://anzograph.com/resource/person100>
```

- Triples with user-defined data types (UDTs) like the following example also require about 18 bytes to store in memory.

```
<http://anzograph.com/resource/person1> <http://anzograph.com/resource/height>  
"5'8"^^height
```

- Triples with dateTime values like the following example require about 20 bytes to store in memory.

```
<http://www.wikidata.org/entity/Q65949130>
<http://www.wikidata.org/prop/direct/P585>
"1995-01-01T00:00:00Z"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
```

- Triples with long strings like the following example require about 700 bytes to store in memory.

```
<http://dbpedia.org/resource/Keanu_Reeves>
<http://dbpedia.org/ontology/abstract> "Keanu Charles Reeves
(/ker'ɑ:nu:/ kay-AH-noo; born September 2, 1964) is a Canadian actor,
producer, director and musician.
Reeves is best known for his acting career, beginning in 1985 and spanning
more than three decades.
He gained fame for his starring role performances in several blockbuster films
including comedies
from the Bill and Ted franchise (1989-1991), action thrillers Point Break
(1991) and Speed (1994),
and the science fiction-action trilogy The Matrix (1999-2003). He has also
appeared in dramatic
films such as Dangerous Liaisons (1988), My Own Private Idaho (1991), and
Little Buddha (1993),
as well as the romantic horror Bram Stoker's Dracula (1992)."
```

The following table provides estimates for the number of triples that you can load and query with specific amounts of available RAM. The table also lists the number of triples that could be stored in given amounts of memory, using the triples described in the previous examples.

Note

The estimates listed in the table represent the number of triples at rest and take into consideration that the data should not consume more than 50% of all available RAM.

Available RAM	General Estimate	Examples
16 GB	Up to about 100 million triples	Considering that the data at rest should use less than 8 GB RAM, a server with 16 GB total RAM could store:

Available RAM	General Estimate	Examples
		- About 12 million 700-byte triples like the Keanu Reeves example above. - About 475 million 18-byte URI triples like the example above.
32 GB	Up to about 200 million triples	Considering that the data at rest should use less than 16 GB RAM, a server with 32 GB total RAM could store: - About 24 million 700-byte triples like the Keanu Reeves example above. - About 850 million 20-byte triples like the dateTime example above.
64 GB	Up to about 400 million triples	Considering that the data at rest should use less than 32 GB RAM, a server with 64 GB total RAM could store: - About 48 million 700-byte triples like the Keanu Reeves example above. - About 1.7 billion 20-byte URI triples.
128 GB	Up to about 800 million triples	Considering that the data at rest should use less than 64 GB RAM, a server with 128 GB total RAM could store: - About 96 million 700-byte triples like the Keanu Reeves example above. - About 3.4 billion 20-byte URI triples.
256 GB	Up to about 1.5 billion triples	Considering that the data at rest should use less than 128 GB RAM, a server with 256 GB total RAM could store:

Available RAM	General Estimate	Examples
		- About 192 million 700-byte triples like the Keanu Reeves example above. - About 6.8 billion 20-byte URI triples.
512 GB	Up to about 3 billion triples	Considering that the data at rest should use less than 256 GB RAM, a server with 512 GB total RAM could store: - About 390 million 700-byte triples like the Keanu Reeves example above. - About 13 billion 20-byte URI triples.

Memory usage can be high during loads

During the load process, before the data can be moved to its final storage block, memory usage temporarily increases, particularly if the data includes many string values.

Memory Usage Examples

The following table provides memory requirement estimates for public or commercial data sets that users may already use or be familiar with:

Data Set	Memory Requirements	Description
Graph-500 (.csv)	2 GB	Graph search data set created by graph500.org to facilitate benchmark testing of large data and CPU intensive computing.
WikiData (.nt)	340 GB at rest; 900+ GB to load and run queries	Large downloadable data set reflecting contents of various Wikimedia projects.

Analyzing Data Characteristics in Load Files

Graph Lakehouse allows you to perform pre-load analysis on load files without actually loading the data into memory. You can use this method to run statistical queries, such as counting the number of triples, getting to know the data, or returning a list of the nodes or subjects and predicates.

Performing a "dry run" of a data load, beforehand, enables you to analyze data set characteristics to help with tasks such as memory sizing and overall capacity planning. You can use this method to capture statistics about a large data set on a smaller system than what would actually be required to load the data in memory.

Important Considerations for Analyzing Load Files

- Since Graph Lakehouse scans the files on-disk, queries run much slower than they do when run against data in memory. Consider performance when deciding how many files to query at once and how complex to make queries.
- Though the pre-load feature does not use memory for storing data, queries that you run against files do consume some memory. The server must have sufficient memory available to use for these intermediate query results.
- Unlike loads into the database, pre-load analysis does not prune duplicate triples. Statistics returned for load file queries may differ somewhat from the statistics returned after the data is loaded.

Analysis Query Syntax

The syntax that you use to query load files depends on the file type. For example, for files in triple or quad format, like Turtle (.ttl), N-Triple (.n3 and .nt), N-Quad (.nq and .quads), and TriG (.trig) files, you can use the following syntax:

```
SELECT <expression>
FROM EXTERNAL <dir:/path/dir_or_file_name>
[ FROM EXTERNAL <dir:/path/dir_or_file_name> ]
WHERE { <triple_patterns> }
```

Option	Description
SELECT <expression>	In the SELECT clause, specifies an expression that returns statistical results such as a count of the total number of triples or the number of distinct predicates. Queries that return values for a specific property may return an error.
FROM EXTERNAL <dir:/path/dir_ or_file_name>	The URI in the FROM clause specifies the location of the load file or directory of files. For example, this URI specifies a single file on the local file system: <pre><file:/home/user/data/tickit.ttl></pre> This example specifies a directory of files: <pre><dir:/data/load-files/tickit.ttl.gz></pre>

For example, the following query analyzes the tickit.ttl.gz directory to count the total number of triples in the files:

```
SELECT (count (*) as ?triples)
FROM EXTERNAL <dir:/opt/anzograph/etc/tickit.ttl.gz>
WHERE { ?s ?p ?o . }
```

```
triples
-----
5368800
1 rows
```

The example below analyzes the tickit.ttl.gz directory to count the total number of triples and the number of distinct subjects and predicates:

```
SELECT
  (count (*) as ?triples)
  (count(distinct ?s) as ?subjects)
  (count(distinct ?p) as ?preds)
FROM EXTERNAL <dir:/opt/anzograph/etc/tickit.ttl.gz>
WHERE { ?s ?p ?o . }
```

```
triples | subjects | preds
-----+-----+-----
5368800 | 424319      | 45
1 rows
```

Estimating Memory Requirements Based on Data Characteristics

Although the memory required to load and perform queries on specific data sets will vary based on the size and type of data contained in a data set, you can still obtain a reasonable estimate or starting point for the amount of memory you will need to load any specific data set. Using the method of pre-load analysis of load files described earlier (see [Analyzing Data Characteristics in Load Files](#)), you can query the data set to calculate a rough estimate of the memory required to load the data set in memory.

1. Calculate the number of triples the data set will generate when stored in Graph Lakehouse.
2. Multiply the number of triples by an average triple size.
3. Add the number of characters stored in all of the character strings contained in the data set.

Using the example of the Tickit data set provided with Graph Lakehouse, you can perform a query like the following to calculate the number of triples the Tickit data set will contain when loaded into memory:

```
SELECT (COUNT(*) as ?triple_count)
FROM EXTERNAL <dir:/opt/anzograph/etc/tickit.ttl.gz>
WHERE {?s ?p ?o}

triple_count
-----
7696012
1 rows
```

Note

Queries run against files on disk will run significantly slower than they do when run against data in memory. Also, note that pre-load analysis of data sets does not prune duplicate triples, unlike data sets loaded in memory, so the calculation of the number of triples may differ

somewhat from the number reported after the data set is loaded in memory.

Once you know the total number of triples, multiply the value by the average triple storage size. The [Memory usage depends on data characteristics](#) section above shows some example triples and their estimated size. If you are familiar with the data in the files, you may be able to determine the average size based on the examples. Otherwise, Altair recommends using 30 bytes as the average triple size. For example, using the triple count above and an average triple size of 30 bytes:

```
7696012 x 30 = 230,880,360 bytes
```

To calculate the additional memory required for in-memory storage of character string data, you can run a query like the following:

```
SELECT
  (SUM(IF(DATATYPE(?o)=<http://www.w3.org/2001/XMLSchema#string>, (STRLEN(?o)),0)) AS
  ?char_count)
FROM EXTERNAL <dir:/opt/anzograph/etc/tickit.ttl.gz>
WHERE {?s ?p ?o.}

char_count
-----
4893660
1 rows
```

For ASCII characters, Graph Lakehouse requires a single byte of memory for each character, so adding the total number of characters to the previous memory calculation for storing triples, the result is the following:

```
230,880,360 + 4,893,660 = 235,774,020 total bytes
```

Note that the calculation of **235,774,020** total bytes (0.24 GB) provides an estimate for data set storage "at rest" and takes into account only one data set stored in memory. When coming up with a final recommendation for total memory requirements of an Graph Lakehouse deployment, account for any other data sets you may want to load in memory at the same time. You also need to keep in mind other memory sizing guidelines, for example, that all loaded data should not consume more than 50% of all available RAM.

Securing a Graph Lakehouse Environment

This topic lists the recommended procedures to follow to strengthen the security of Graph Lakehouse environments.

- [Set Up Firewall Rules](#)
- [Replace the Default Self-Signed Certificates with Trusted Certificates](#)
- [Configure File Access Policies](#)

Set Up Firewall Rules

In order to protect the environment from malicious systems and prevent man-in-the-middle attacks or leaking of data source credentials, firewall rules should be configured for the Graph Lakehouse cluster network. Rules should allow outbound connections only to trusted data sources and services. For information about the ports that need to be opened for inbound and outbound connections to support normal operations, see [Firewall Requirements](#).

Replace the Default Self-Signed Certificates with Trusted Certificates

Graph Lakehouse installations include self-signed certificates, `serv.crt` and `ca.crt`, and private and public keys, `serv.key` `serv.pub.key`, in the `<install_path>/config` directory. The certificates and keys are required for encrypted communication over gRPC protocol. You can follow the steps below to replace the default certificates and keys with your own trusted files.

Important

Your certificates must meet the following requirements:

- All servers in the cluster must use the same certificates and keys.
- The DNS in the certificates must be `localhost`.
- Your certificates and keys must use the same file names as the default files that you are replacing.
- The public key should be generated from the new private key.

Note

The private and public keys are used to encrypt and decrypt the system manager password. If you replace the keys and have enabled (or plan to enable) system manager authentication (as described in [Securing a Graph Lakehouse Environment](#) below), you must also generate a new azgmgrd password and re-authenticate azgmgrd as described in [Securing a Graph Lakehouse Environment](#).

1. On the leader server, run the following commands to stop the database and the system manager, azgmgrd:

```
sudo systemctl stop anzograph
```

```
sudo systemctl stop azgmgrd
```

2. On the leader server, open the `<install_path>/config/settings.conf` file for editing.
3. Uncomment the `use_custom_ssl_files=false` line and change the value to **true**.
4. Save and close settings.conf.
5. On each server in the cluster, replace the `serv.crt`, `ca.crt`, `serv.key`, and `serv.pub.key` files in the `<install_path>/config` directory with your files. Make sure that the new files have the same file names as the default files.
6. Restart Graph Lakehouse with the following commands. Run the first command on all servers in the cluster. Then run the second command on the leader server:

```
sudo systemctl start azgmgrd
```

```
sudo systemctl start anzograph
```

Configure File Access Policies

Graph Lakehouse offers configuration options for ensuring that only certain files or directories on the server are accessible during the execution of a query. These configuration settings specify patterns that are used to determine whether a directory or file is accessible. When Graph Lakehouse receives a request that includes a path to a file or directory, it checks that path against

the allowed and denied access patterns. If the specified file or directory matches one of the allowed access patterns and it is not matched to a deny pattern, the query is executed. If the specified path is matched to a denied pattern or is not matched to any of the allowed patterns, the query is aborted and Graph Lakehouse returns an access denied error message. For details and configuration instructions, see [Manage File Access Policies](#).

Deploy

Graph Lakehouse can be deployed on-premises or on any of the common cloud platforms. Graph Lakehouse also integrates with popular container management applications. The main selection criteria for choosing deployment options, as well as the associated server sizing and scaling, starts with the analytic applications you intend to use with Graph Lakehouse. Consider the data sources and size of the data you plan to load as well as your query performance requirements. For more information, see [Sizing Guidelines for In-Memory Storage](#).

Important

By default, all deployments include an embedded anonymous free edition license. The license does not expire, but it is valid for single server deployments only and it enforces a limit of 8 GB RAM usage. Though Graph Lakehouse will not use more than 8 GB RAM in the free edition version, Altair recommends that you deploy Graph Lakehouse on a host server that has at least 16 GB of RAM available. To increase the memory limit or remove all limitations, you can upgrade the license by contacting [Altair Customer Support](#). For more information, see [Install or Upgrade a License](#).

In this section:

Container Image Deployments	42
Kubernetes Deployments	56
Enterprise Linux 9 Deployments	65
IBM Cloud Pak Deployments	91

Container Image Deployments

The topics in this section help you get started quickly by deploying Graph Lakehouse as a single server container image using a container engine such as Docker, Podman, or Rancher. The following topics list server and software requirements, give guidance on configuring the container engine environment for use with Graph Lakehouse, and provide instructions for deploying an Graph Lakehouse container image.

In this section:

Container Engine Requirements	43
Deploy the Graph Lakehouse Container Image	46

Container Engine Requirements

This topic lists the container engine requirements for running Graph Lakehouse container images.

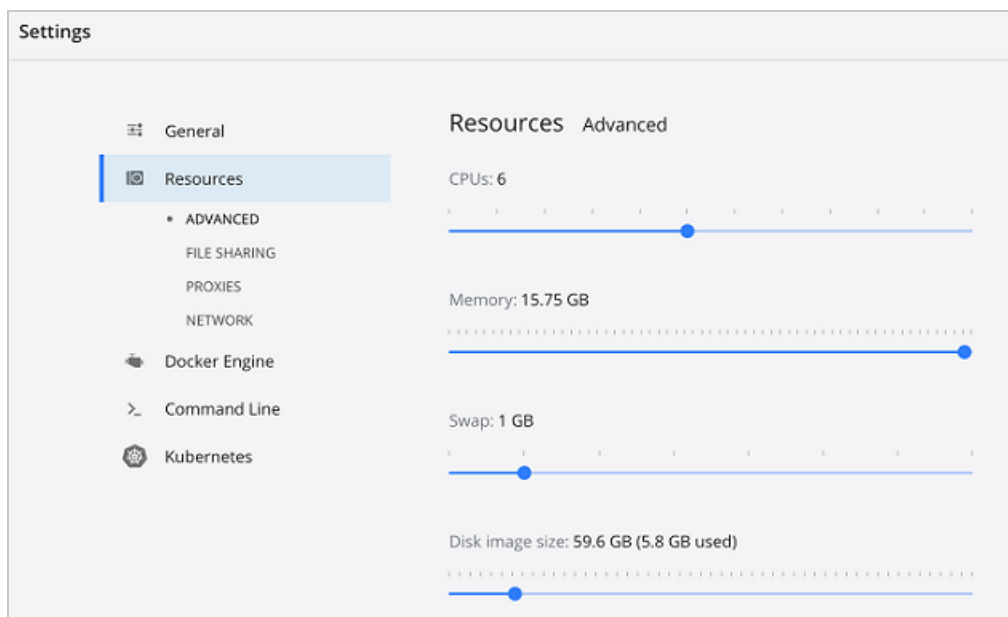
Component	Guidelines
Operating Systems	MacOS, Linux, Windows 10 Professional or Enterprise edition. Important • Mac ARM: Mac ARM-based processors are not supported. The Graph Lakehouse image will not run in Docker on an ARM-based Mac, even if the <code>--platform linux/amd64</code> flag is used. • Ubuntu: Graph Lakehouse is not supported on Ubuntu 16.04 LTS. To deploy an Graph Lakehouse container image on Ubuntu, use Ubuntu 18.04 LTS or later. • Windows: Container management applications use a hypervisor with a VM, and the host server must support virtualization. Since older Windows versions and Windows 10 Home edition do not support Hyper-V, Windows 10 Professional or Enterprise is required.
Available RAM	Minimum: 8 GB; Recommended: 16 GB. Graph Lakehouse needs enough RAM to store data, intermediate query results, and run the server processes. Altair recommends that you allocate 3 to 4 times as much RAM as the planned data size. For guidance on sizing Graph Lakehouse servers, see Sizing Guidelines for In-Memory Storage.
Available Disk Space	Graph Lakehouse requires 30 GB for internal requirements. The amount of additional disk space required for load file staging, persistence, or backups depends on the size of the data to be loaded. For persistence, Altair recommends that you have twice as much disk space available as RAM on the

Component	Guidelines
	server.
CPU Count	Minimum: 2; Recommended 8+. Intel x86-64 processors are recommended, but Graph Lakehouse is supported on Epyc and later generation AMD processors. Graph Lakehouse does not run on Opteron AMD processors or Mac ARM-based processors.

Adjusting Container Resources

If you use a desktop container application, you may need to adjust the resources that are available to the Graph Lakehouse image. Graph Lakehouse requires at least 2 CPU, 10 GB of available disk space, and 8 GB of available RAM to start the database. Altair recommends that you make at least 16 GB memory available to the image. For instructions on tuning resources, see the documentation for your container engine.

For example, in Docker Desktop for Mac, click the Docker icon in the menu bar and select **Preferences**. On the Settings screen, select **Resources**. For example:



Increase the CPUs, Memory, and Disk image size as needed to meet the Graph Lakehouse requirements. Then click **Apply & Restart** to apply the changes and restart Docker.

For instructions on deploying the Graph Lakehouse image, see [Deploy the Graph Lakehouse Container Image](#).

Deploy the Graph Lakehouse Container Image

This topic provides instructions for downloading and deploying a Graph Lakehouse container image using the command line. The steps include Docker commands that may need to be customized depending on your application.

- [Deploy Graph Lakehouse in a Mac Desktop Application](#)
- [Deploy Graph Lakehouse in a Linux Container Engine](#)
- [Deploy Graph Lakehouse in a Windows Desktop Application](#)

Deploy Graph Lakehouse in a Mac Desktop Application

Follow the instructions below to deploy Graph Lakehouse on a Mac desktop container application.

1. If necessary, start the desktop application, and then open the Mac Terminal app.
2. Applications typically cache images on the host. If you have deployed an Graph Lakehouse container previously, that image will be used to redeploy Graph Lakehouse. If you want to deploy the latest release, first pull the latest image. To do so, run the following command:

```
docker pull cambridgesemantics/anzograph:latest
```

Tip

You can deploy alternate Graph Lakehouse versions by replacing the "latest" tag with any of the tags that are available on the [AnzoGraph Docker Hub](#) site.

3. If you are deploying the Graph Lakehouse container for the first time, Altair recommends that you create a directory on the local file system where load files, query files, and other files can be staged and shared with the container file system. When you deploy Graph Lakehouse, you map the directory on the local file system to a directory in the container. This way the files are shared, and if you remove the Graph Lakehouse container, the local file system retains a copy of the shared files. If you redeploy a new Graph Lakehouse image, the new container can be mapped to the same local directory and access the existing files. To create the directory, navigate to a location on the host and run the following command to create a

directory in the current directory:

```
mkdir <directory_name>
```

For example:

```
mkdir shared-files
```

Note

On Mac and Linux, Docker is configured by default to allow local directories to be shared with containers. On Mac, the /Users, /Volumes, /private, and /tmp directories are shared. If necessary, you can configure additional locations in Docker **Preferences > Resources > File Sharing**.

4. In Terminal, run the following command to deploy the Graph Lakehouse image. The command runs the Graph Lakehouse image and configures HTTP and HTTPS access by mapping the container ports to the HTTP and HTTPS ports on the host:

```
docker run -d -p <host_http_port>:8080 -p <host_https_port>:8443 \
-v /<path>/<shared_directory>:/opt/<shared_directory> \
--name=<container_name> cambridgesemantics/anzograph:<tag>
```

The list below describes each of the parameters:

- **host_http_port** is the port on the local host to use for HTTP access to the Graph Lakehouse user interface. In the container, the user interface binds to port 8080 for HTTP access. Altair recommends that you specify **80** to map the container's HTTP port to port 80 on the local host. If port 80 is in use, specify an alternate port for host_http_port.
- **host_https_port** is the port on the local host to use for HTTPS access to the Graph Lakehouse user interface. In the container, the user interface binds to port 8443 for HTTPS access. Altair recommends that you specify **443** to map the container's HTTPS port to port 443 on the local host. If port 443 is in use, specify an alternate port for host_https_port.

- **/path/shared_directory**: The path and directory name for the shared directory on the local file system.
- **shared_directory**: The directory on the container file system to map to the shared directory on the local file system. The directory is created when the container is deployed.

Note

The command above lists `/opt/` as a convenient path on the container file system because the Graph Lakehouse path is `/opt/anzograph`. You can specify a different path.

- **container_name** is the short name to use to identify the Graph Lakehouse container. For example, **anzograph**.
- **tag** is the tag from the [AnzoGraph Docker Hub](#) site that identifies the version of Graph Lakehouse to deploy. If you pulled an image in the first step, this tag should match the tag from the pull command. Usually the **latest** tag is specified so the most recent release is deployed.

For example:

```
docker run -d -p 80:8080 -p 443:8443 -v /Volumes/shared-files:/opt/shared-files --name=anzograph cambridgesemantics/anzograph:latest
```

When the prompt returns the container ID, the container is running. For example:

```
c16b912a4a8944592297cf052f90447a5657c3362540334aba2195ae8941f1af
```

Graph Lakehouse is now installed and ready to use. You can access the user interface by opening a browser and going to **http://127.0.0.1**. If you specified a port other than 80 for the host HTTP port, specify that port in the URL. For example, **http://127.0.0.1:8888**.

Use the following credentials to log in to the user interface:

- Username: **admin**
- Password: **Passw0rd1**

For next steps, see [Get Started](#), a brief tutorial designed to introduce you to the Graph Lakehouse user interface and command line interface and get you started with loading data and running SPARQL queries.

Deploy Graph Lakehouse in a Linux Container Engine

Follow the instructions below to deploy Graph Lakehouse on Linux.

1. If necessary, start the container application.
2. Applications typically cache images on the host. If you have deployed an Graph Lakehouse container previously, that image will be used to redeploy Graph Lakehouse. If you want to deploy the latest release, first pull the latest image. To do so, run the following command:

```
docker pull cambridgesemantics/anzograph:latest
```

Tip

You can deploy alternate Graph Lakehouse versions by replacing the "latest" tag with any of the tags that are available on the [AnzoGraph Docker Hub](#) site.

3. If you are deploying the Graph Lakehouse container for the first time, Altair recommends that you create a directory on the local file system where load files, query files, and other files can be staged and shared with the container file system. When you deploy Graph Lakehouse, you map the directory on the local file system to a directory in the container. This way the files are shared, and if you remove the Graph Lakehouse container, the local file system retains a copy of the shared files. If you redeploy a new Graph Lakehouse image, the new container can be mapped to the same local directory and access the existing files. To create the directory, navigate to a location on the host and run the following command to create a directory in the current directory:

```
mkdir <directory_name>
```

For example:

```
mkdir shared-files
```

Note

On Mac and Linux, Docker is configured by default to allow local directories to be shared with containers. On Mac, the /Users, /Volumes, /private, and /tmp directories are shared. If necessary, you can configure additional locations in Docker **Preferences > Resources > File Sharing**.

4. Run the following command to deploy the Graph Lakehouse image. The command instructs Docker to start Graph Lakehouse and configure HTTP and HTTPS access to the application by mapping the container ports to the HTTP and HTTPS ports on the local host:

```
docker run -d -p <host_http_port>:8080 -p <host_https_port>:8443 \
-v /<path>/<shared_directory>:/opt/<shared_directory> \
--name=<container_name> cambridgesemantics/anzograph:<tag>
```

The list below describes each of the parameters:

- **host_http_port** is the port on the local host to use for HTTP access to the Graph Lakehouse user interface. In the container, the user interface binds to port 8080 for HTTP access. Altair recommends that you specify **80** to map the container's HTTP port to port 80 on the local host. If port 80 is in use, specify an alternate port for host_http_port.
- **host_https_port** is the port on the local host to use for HTTPS access to the Graph Lakehouse user interface. In the container, the user interface binds to port 8443 for HTTPS access. Altair recommends that you specify **443** to map the container's HTTPS port to port 443 on the local host. If port 443 is in use, specify an alternate port for host_https_port.
- **/path/shared_directory**: The path and directory name for the shared directory on the local file system.
- **shared_directory**: The directory on the container file system to map to the shared directory on the local file system. The directory is created when the container is deployed.

Note

The command above lists `/opt/` as a convenient path on the container file system because the Graph Lakehouse path is `/opt/anzograph`. You can specify a different path.

- **container_name** is the short name to use to identify the Graph Lakehouse container. For example, **anzograph**.
- **tag** is the tag from the [AnzoGraph Docker Hub](#) site that identifies the version of Graph Lakehouse to deploy. If you pulled an image in the first step, this tag should match the tag from the pull command. Usually the **latest** tag is specified so the most recent release is deployed.

For example:

```
docker run -d -p 80:8080 -p 443:8443 -v /opt/shared-files:/opt/shared-files --name=anzograph cambridgesemantics/anzograph:latest
```

When the prompt returns the container ID, the container is running. For example:

```
c16b912a4a8944592297cf052f90447a5657c3362540334aba2195ae8941f1af
```

Tip

If you want to attach to the container and explore the Graph Lakehouse file system, you can run the following command.

```
docker exec -it anzograph /bin/bash
```

Graph Lakehouse is now installed and ready to use. To open the user interface, open a browser and go to the following URL:

```
https://<host_IP_address>
```

Where `<host_IP_address>` is the IP address of the host server where the container image is installed. If you mapped the container's HTTPS port to port 443 on the host, you do not need to specify a port. If you specified a port other than 443, include the port in the URL. For example, `https://10.100.0.1:8888`.

Tip

If you are using Docker for Linux locally and need to know the IP address of the Graph Lakehouse container, you can run the following command:

```
docker inspect <container_name> | grep '"IPAddress"' | head -n 1
```

For example:

```
docker inspect anzograph | grep '"IPAddress"' | head -n 1
```

```
"IPAddress": "172.17.0.2"
```

Use the following credentials to log in to the user interface:

- Username: **admin**
- Password: **Passw0rd1**

For next steps, see [Get Started](#), a brief tutorial designed to introduce you to the Graph Lakehouse user interface and command line interface and get you started with loading data and running SPARQL queries.

Deploy Graph Lakehouse in a Windows Desktop Application

Follow the instructions below to deploy Graph Lakehouse on a Windows desktop container application.

1. If necessary, start the desktop application, and then open the Windows PowerShell application.
2. Applications typically cache images on the host. If you have deployed an Graph Lakehouse container previously, that image will be used to redeploy Graph Lakehouse. If you want to deploy the latest release, first pull the latest image. To do so, run the following command:

```
docker pull cambridgesemantics/anzograph:latest
```

Tip

You can deploy alternate Graph Lakehouse versions by replacing the "latest" tag with any of the tags that are available on the [AnzoGraph Docker Hub](#) site.

3. If you are deploying the Graph Lakehouse container for the first time, Altair recommends that you create a directory on the local file system where load files, query files, and other files can be staged and shared with the container file system. When you deploy Graph Lakehouse, you map the directory on the local file system to a directory in the container. This way the files are shared, and if you remove the Graph Lakehouse container, the local file system retains a copy of the shared files. If you redeploy a new Graph Lakehouse image, the new container can be mapped to the same local directory and access the existing files. To create the directory, navigate to a location on the host and run the following command to create a directory in the current directory:

```
mkdir <directory_name>
```

For example:

```
mkdir shared-files
```

4. In PowerShell, run the following command to deploy the Graph Lakehouse image. The command runs Graph Lakehouse and configures HTTP and HTTPS access by mapping the container ports to the HTTP and HTTPS ports on the local host:

```
docker run -d -p <host_http_port>:8080 -p <host_https_port>:8443 \  
-v \<path>\<shared_directory>:/opt/<shared_directory> \  
--name=<container_name> cambridgesemantics/anzograph:<tag>
```

The list below describes each of the parameters:

- **host_http_port** is the port on the local host to use for HTTP access to the Graph Lakehouse user interface. In the container, the user interface binds to port 8080 for HTTP access. Altair recommends that you specify **80** to map the container's HTTP port to port 80 on the local host. If port 80 is in use, specify an alternate port for host_http_port.

- **host_https_port** is the port on the local host to use for HTTPS access to the Graph Lakehouse user interface. In the container, the user interface binds to port 8443 for HTTPS access. Altair recommends that you specify **443** to map the container's HTTPS port to port 443 on the local host. If port 443 is in use, specify an alternate port for **host_https_port**.
- **\path\shared_directory**: The path and directory name for the shared directory on the local file system.
- **shared_directory**: The directory on the container file system to map to the shared directory on the local file system. The directory is created when the container is deployed.

Tip

The command above lists `/opt/` as a convenient path on the container file system because the Graph Lakehouse path is `/opt/anzograph`. You can specify a different path.

- **container_name** is the short name to use to identify the Graph Lakehouse container. For example, **anzograph**.
- **tag** is the tag from the [AnzoGraph Docker Hub](#) site that identifies the version of Graph Lakehouse to deploy. If you pulled an image in the first step, this tag should match the tag from the pull command. Usually the **latest** tag is specified so the most recent release is deployed.

For example:

```
docker run -d -p 80:8080 -p 443:8443 -v C:\shared-files:/opt/shared-files --name=anzograph cambridgesemantics/anzograph:latest
```

When the prompt returns the container ID, the container is running. For example:

```
c16b912a4a8944592297cf052f90447a5657c3362540334aba2195ae8941f1af
```

Graph Lakehouse is now installed and ready to use. You can access the user interface by opening a browser and going to **http://127.0.0.1**. If you specified a port other than 80 for the host HTTP port, specify that port in the URL. For example, **http://127.0.0.1:8888**.

Use the following credentials to log in to the user interface:

- Username: **admin**
- Password: **Passw0rd1**

For next steps, see [Get Started](#), a brief tutorial designed to introduce you to the Graph Lakehouse user interface and command line interface and get you started with loading data and running SPARQL queries.

Kubernetes Deployments

Since Kubernetes has become the de facto package manager for Docker hosts, and Helm is emerging as the de facto package manager for Kubernetes application architectures, Altair offers a Graph Lakehouse for Kubernetes deployment that is managed by Helm. For instructions on quickly deploying Graph Lakehouse in a test environment with another container engine, see [Container Image Deployments](#).

The topics in this section provide information about setting up a local machine so that you can deploy Graph Lakehouse on a remote Kubernetes cluster; they include reference information and examples for installing the Kubernetes command line client, configuring access to a Kubernetes cluster, and installing Helm. In addition, this section provides detailed instructions for using Helm to deploy and customize Graph Lakehouse on Kubernetes.

Note

To deploy Graph Lakehouse with Kubernetes Minikube, you must use **Minikube Version 1.27 or later**. The Linux kernel that ships with earlier Minikube versions is not sufficient for running Graph Lakehouse.

In this section:

Install the Kubernetes Command Line Client	57
Configure Access to a Kubernetes Cluster	58
Install Helm	60
Deploy Graph Lakehouse with Helm	61

Install the Kubernetes Command Line Client

The Kubernetes command line client, **kubectl**, enables users to deploy applications on Kubernetes. Though Helm includes the Kubernetes API, the local host needs the kubectl client to access Kubernetes clusters and display status and resource details.

The Kubernetes client that you install depends on your operating system or the vendor that hosts your Kubernetes cluster. For example, if the Google Cloud Platform hosts your environment, you can download kubectl as part of the Google Cloud SDK. For instructions, see [Download as part of the Google Cloud SDK](#) in the Kubernetes documentation. For more information and instructions for other operating systems, see [Install and Set Up kubectl](#) in the Kubernetes documentation.

This topic provides an example installation that downloads and configures the kubectl binary on a Linux operating system.

1. Run the following cURL command to download the kubectl binary:

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"
```

2. Run the following command to make the binary executable:

```
chmod +x ./kubectl
```

3. Run the following command to move the binary to your PATH:

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

4. To confirm that the binary is installed and that you can run kubectl commands, run the following command to display the client version:

```
kubectl version --client
```

Now that the Kubernetes CLI is installed, you can use it to set up access to the Kubernetes cluster to use for deploying Graph Lakehouse. See [Configure Access to a Kubernetes Cluster](#) for next steps.

Configure Access to a Kubernetes Cluster

After installing the Kubernetes command line client, `kubectl`, you can set up the default Kubernetes configuration context that Helm uses to access the Kubernetes cluster and deploy Graph Lakehouse. The method that you use to set up the configuration context depends on the vendor that hosts the Kubernetes cluster. For information, see the configuration instructions for your vendor. See [Configure Access to Multiple Clusters](#) in the Kubernetes documentation for more general information about configuring access to clusters.

This topic provides example instructions that configure a local Linux environment to access a Kubernetes cluster hosted on the Google Cloud Platform.

1. Run the following command to set the project to the Kubernetes cloud project:

```
gcloud config set project <k8s_project_name>
```

For example:

```
gcloud config set project cloud-kube-1111
```

```
Updated property [core/project]
```

2. Run the following command to set the compute zone for the Kubernetes cluster:

```
gcloud config set compute/zone <zone_name>
```

For example:

```
gcloud config set compute/zone us-central1
```

```
Updated property [compute/zone]
```

3. To confirm that you can access the cluster, you can run the following command to view cluster details:

```
gcloud container clusters list
```

The command returns the information such as the name, location, number of nodes, and status of the cluster. For example:

NAME	LOCATION	MASTER_VERSION	MASTER_IP	MACHINE_TYPE	NODE_
VERSION	NUM_NODES	STATUS			
cloud-k8s	us-central1	1.23.17-gke.3600	10.100.10.10	n1-standard-1	
1.23.17-gke.3600	27	RUNNING			

4. Run the following command to fetch the credentials for the cluster:

```
gcloud container clusters get-credentials <cluster_name>
```

For example:

```
gcloud container clusters get-credentials cloud-k8s
```

```
Fetching cluster endpoint and auth data.
kubeconfig entry generated for cloud-k8s.
```

Once access to the Kubernetes cluster is configured, see [Install Helm](#) for next steps.

Install Helm

For instructions on downloading and installing Helm for your operating system, see [Installing Helm](#) in the Helm documentation.

This topic provides example instructions that install Helm on Linux.

1. Run the following cURL command to download the installer script, **get-helm.sh**, from GitHub:

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
```

The script includes documentation so that you can review what it does.

2. Run the following command to set permissions for `get_helm.sh`:

```
chmod 700 get_helm.sh
```

3. Run the following command to install Helm:

```
./get_helm.sh
```

The command downloads the latest Helm tarball for your operating system and installs it. For example:

```
Downloading https://get.helm.sh/helm-v3.11.3-linux-amd64.tar.gz
Verifying checksum... Done.
Preparing to install helm into /usr/local/bin
helm installed into /usr/local/bin/helm
```

Helm can now be used to deploy Graph Lakehouse. See [Deploy Graph Lakehouse with Helm](#) for instructions.

Deploy Graph Lakehouse with Helm

Follow the instructions below to deploy Graph Lakehouse using Helm.

1. Run the following command to add the Cambridge Semantics repository to Helm:

```
helm repo add csi-helm https://storage.googleapis.com/csi-helm/
```

2. Run the following command to update the metadata for the Helm repository.

```
helm repo update
```

3. Run the following command to find the available Graph Lakehouse Helm charts.

```
helm search repo anzograph
```

The command returns details about the two charts that are available, one for Graph Lakehouse and one for the Cambridge Semantics Apache Zeppelin image (see [Use Third-Party Visualization Tools](#) for information about the Zeppelin image).

NAME	CHART VERSION	APP VERSION	DESCRIPTION
csi-helm/anzograph	2.0.20230427	3.1.5	CSI Anzograph deployment on K8S
csi-helm/zeppelin	0.2.20191219	0.8.2	CSI Zeppelin deployment on K8s that enables...

4. Run the following command to fetch and view the readme for the Graph Lakehouse Helm chart:

```
helm inspect readme csi-helm/anzograph | tee Readme.md
```

5. Run the following command to fetch and view the Graph Lakehouse Helm chart values (values.yaml):

```
helm inspect values csi-helm/anzograph | tee values.yaml
```

6. By default the Helm chart is configured to deploy a single Graph Lakehouse node with 2 CPU and 8 GiB of RAM. If you want to customize the deployment, such as to specify a larger instance or create a cluster, customize values.yaml before you deploy Graph Lakehouse. In addition, if you obtained a license key from Altair, add that key to values.yaml. The steps

below provide guidance for customizing node or cluster sizes and adding a license key to the deployment. For more detailed information about all of the Graph Lakehouse Helm chart options, view the readme, **Readme.md**.

- a. Open **values.yaml** in a text editor. The file is in the `$HELM_HOME` directory that was defined when you initialized Helm, usually your home directory. You can run `helm home` to view the `HELM_HOME` location.
- b. The option that controls the number of instances for the cluster is in the `Values for statefulset` section of the file:

```
# Values for statefulset
replicas: 1
```

To create a cluster, change the **replicas** value from 1 to the number of nodes that you want to deploy. To achieve the best performance, specify a multiple of 4, i.e., 4, 8, 12, etc. For guidance on sizing Graph Lakehouse servers and clusters, see [Sizing Guidelines for In-Memory Storage](#).

- c. To increase the number of CPU or amount of memory on the instances that will be deployed, change the values for the **cpu** and **memory** settings under **database.resources.requests**. Depending on the values that you specify for requests, you might need to increase the values under **limits**.

For example, the following values create a cluster with instances that have 16 CPU and 120 GiB of RAM each. The upper limit are instances with 32 CPU and 160 GiB.

```
database:
  image:
    repository: "docker.io"
    name: "cambridgesemantics/anzograph-db"
    tag: "3.1.0"
    pullPolicy: "IfNotPresent"
  resources:
    requests:
      cpu: "16000m"
      memory: "120000Mi"
    limits:
      cpu: "32000m"
```

```
memory: "160000Mi"
tolerations: []
```

d. When you have finished customizing the file, save and close values.yaml

7. Run the following command to deploy Graph Lakehouse:

```
helm install -f ~/values.yaml <deployment_name> csi-helm/anzograph
```

Where <deployment_name> is the unique name that you want to assign to this Graph Lakehouse deployment. For example:

```
helm install -f ~/values.yaml anzograph-1 csi-helm/anzograph
```

Helm deploys Graph Lakehouse and displays the initial status. For example:

```
NAME:      anzograph-1
LAST DEPLOYED: Fri May 12 22:32:12 2023
NAMESPACE: default
STATUS:    DEPLOYED

RESOURCES:
==> v1/Pod(related)

NAME                                READY   STATUS    RESTARTS   AGE
anzograph-anzograph-1-0           0/1     Pending   0           0s

==> v1/Secret

NAME                                AGE
anzograph-1-ui-secrets             1s
anzograph-1-license                 1s

==> v1/ConfigMap
anzograph-1-configmap              1s

==> v1/Service
anzograph-1-ui                     1s
anzograph-1-statefulset            1s

==> v1beta1/StatefulSet
anzograph-anzograph-1             1s
```

8. Run the following command to refresh the status and monitor the deployment:

```
helm status <deployment_name>
```

For example:

```
helm status anzograph-1
```

When the status says "Running," the deployment is complete. In the status output under **v1/Service**, note the first service name (with **-ui** appended to the release name). In the example above, the service name is `anzograph-1-ui`.

9. Using the service name for your deployment, run the following command to view the cluster and endpoint information for Graph Lakehouse:

```
kubectl get service <service_name>
```

For example:

```
kubectl get service anzograph-1-ui
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
	AGE			
anzograph-1-ui	LoadBalancer	10.47.254.111	35.225.23.113	443:30281/TCP,80:30704/TCP
		1h		

For next steps, see [Get Started](#) for brief tutorials that are designed to introduce you to the Graph Lakehouse user interface and CLI and get you started with loading data and running SPARQL queries.

Enterprise Linux 9 Deployments

This section provides instructions for using an installer to deploy Graph Lakehouse on a RHEL or Rocky 9.3+ operating system.

In this section:

Pre-Installation Requirements	66
Install Graph Lakehouse	69
Post-Installation Configuration	78
Uninstalling and Updating Graph Lakehouse	88

Pre-Installation Requirements

This page describes the installation requirements and other important information to know before you install Graph Lakehouse. The list below summarizes the requirements and recommendations:

1. Make sure that the host server operating system is RHEL or Rocky Linux 9.3+ and that the server has at least 16 GB RAM and 40 GB disk space available for Graph Lakehouse. For more information about the hardware, software, and firewall requirements, see [Server and Cluster Requirements](#).
2. Certain software packages are required to be installed before the Graph Lakehouse installation. The installer will not run until these prerequisites are installed. See [Prerequisite Software](#) for details and instructions.
3. Additional dependencies are required to be installed to support Graph Lakehouse extensions like the remote read (load) and write service, the Data Science functions, and Apache Arrow integration. However, Altair recommends that you deploy these dependencies after Graph Lakehouse is installed because the installation includes a `.repo` file that can aid you in the installing the packages. See [Post-Installation C++ Dependencies](#) for details.
4. When the installer is run with elevated privileges (sudo mode), the installer can complete the Graph Lakehouse installation as well as the important post-installation configuration so that Graph Lakehouse is running and ready to use when the installation is finished. See [Installation Modes and User Accounts](#) for details about the installation modes and Graph Lakehouse users.

Prerequisite Software

The following software must be installed on the host servers before Graph Lakehouse is installed.

- [Install a Java 21 Virtual Environment](#)
- [Install the GNU C Devel Library](#)
- [Install the GNU Binutils Library](#)

Install a Java 21 Virtual Environment

All Graph Lakehouse servers are required to include a Java 21 virtual environment. OpenJDK 21 and GraalVM 21 are supported. For example, you can run the following command to install OpenJDK 21. **Install the JVM on all servers in the cluster:**

```
sudo dnf install java-21-openjdk
```

Note

You do not need to set the `$JAVA_HOME` variable to use the Java installation. Graph Lakehouse's system management daemon (`azgmgrd`) requires `JAVA_HOME`, and it is set when services are configured as part of the installation (see [Configuring the Graph Lakehouse Services and Starting the Database](#)).

Install the GNU C Devel Library

All Graph Lakehouse servers are required to include the latest version of the GNU C `glibc-devel` library for your operating system. **On all servers in the cluster**, run the following command to install `glibc-devel`:

```
sudo dnf install glibc-devel
```

Install the GNU Binutils Library

All Graph Lakehouse servers are required to include the latest version of the GNU `binutils` library for your operating system. **On all servers in the cluster**, run the following command to install `binutils`:

```
sudo dnf install binutils
```

Post-Installation C++ Dependencies

Additional libraries are required to be installed on all servers in the cluster to support the C++ extensions that Graph Lakehouse offers, including the remote read (load) and write service, the Data Science functions, and the integration with Apache Arrow. Though you can install the C++ dependencies before you install Graph Lakehouse, if you wait until after the installation you can use

the included `csi-obs-cambridgesemantics-udxcontrib.repo` file to enable the Cambridge Semantics repository and install the C++ dependencies with or without internet access. For more information, see [Installing the C++ Dependencies](#) in the post-installation instructions.

Installation Modes and User Accounts

There are two modes in which you can run the installer, **root (sudo)** or **non-root (current user)**. This section describes both modes and the user account and file ownership implications for each mode.

Mode	Description
Sudo Mode	Running the installer in sudo mode is the preferred method of installation. In sudo mode, the installer prompts you to enter the Graph Lakehouse service user name. Systemd units for the system management daemon (<code>azgmgrd</code>) and database (<code>anzograph</code>) processes are created in <code>/etc/systemd/system</code> . The units start Graph Lakehouse as the specified user, and file system permissions for the <code>anzograph</code> directory and any files that Graph Lakehouse writes are based on the same user. The services also configure the appropriate resource limits (ulimits) for Graph Lakehouse and set <code>\$JAVA_HOME</code> for your Java or GraalVM installation.
Non-Root Mode	When running the installer as a non-root user, the installer does not create users and file system permissions are based on the user account that performs the installation. Example systemd units, in the <code><install_path>/examples</code> directory, are provided as a template for you to configure and enable manually. For more information, see Configuring the Graph Lakehouse Services and Starting the Database in the post-installation instructions.

Once the prerequisites are in place, proceed to [Install Graph Lakehouse](#) for instructions on installing the software.

Install Graph Lakehouse

Before installing Graph Lakehouse, make sure that each host server meets the requirements in [Server and Cluster Requirements](#) and that you have installed any prerequisite software (see [Pre-Installation Requirements](#)).

Note

When deploying Graph Lakehouse in a cluster, run the installer on each server in the cluster. Choose one server to be the leader node and designate all other servers as compute/worker nodes.

The installer does support "silent" (unattended) operation in which no user interaction is required and prompt answers are provided by a response file. After the Graph Lakehouse installation, the response file (`response.varfile`) is generated in the `<install_path>/install4j/` directory. After installing Graph Lakehouse on one compute/worker node, for example, you could copy the response file to the other compute/worker servers and then run the following command to silently run the installer with the same responses as the first installation:

```
./<script_name>.sh -q -varfile /<path_to_file>/response.varfile
```

For more information about silent (unattended) installation and using response files, see [Generated Installers](#) in the install4j Help.

To proceed with the installation, follow the steps below:

1. Go to the [AnzoGraph Download](#) page, which lists the Graph Lakehouse releases that are available to download.
2. Download the Installer to each Graph Lakehouse host server. The installer is an interactive shell script that prompts you to specify configuration options for your deployment.
3. Change directories to the location where you copied the script and run the following command to make the script executable:

```
chmod +x <file_name>.sh
```

4. Start the installer using root (sudo) permissions. For example:

```
sudo ./anzograph_linux_3_1_0_r202401232307.sh
```

The installer first verifies that the prerequisites listed in [Prerequisite Software](#) are installed and presents a message if any are missing. For example, on a server where OpenJDK and binutils are installed but the glibc-devel package is missing, the installer displays the following message and the installation is canceled. In this case, the user would follow the instructions in [Install the GNU C Devel Library](#) and then restart the installation.

```
Starting Installer ...
Prerequisite software packages

The following packages are missing on your system and are required to
be installed before proceeding with the Graph Lakehouse installation:
- glibc-devel

Canceling the installation, install the missing software, and then run
the installer again.
```

If the prerequisite software is installed, but the C++ dependencies are not, the installer presents an informational message to let you know the dependencies are required but it is recommended that you install them after the Graph Lakehouse installation. For example:

```
Starting Installer ...
Prerequisite software packages

INFO: The following C++ dependencies are also required and missing from your
system. However, Altair recommends that you install them after
the Graph Lakehouse installation is complete. Follow the Post-Installation
instructions in the online documentation.
libarchive
libarmadillo12
libboost_filesystem1_80_0
libboost_iostreams1_80_0
libboost_system1_80_0
libflatbuffers2
libhdfs3
libnfs13
libserd-0-0
```

```
libsmb2
shadow-utils
```

If all dependencies are installed or you proceed with the installation after getting the informational message, the license agreement is displayed. Press **Enter** to scroll through the text. At the end you are prompted to accept the agreement.

```
Starting Installer ...
Please read the following License Agreement. You must accept the terms of
this agreement before continuing with the installation.

ANZOGRAPH(R) DB
END USER LICENSE AGREEMENT
...
I accept the agreement
Yes [1], No [2]
```

5. Enter **1** to accept the license agreement and continue the installation. The installer prompts you to specify the directory where Graph Lakehouse should be installed.

```
Where should Graph Lakehouse be installed?
[/opt/altair]
```

Note

Two subdirectories and an `uninstall` script are created inside the directory that you specify in this prompt. One subdirectory is named `anzograph` and includes the installation files. Several other directories are created and include files such as systemd service files, a `tuned` profile, and a `.repo` file that can be used to install the C++ extension dependencies. Because an `anzograph` directory will be created, you may not want to specify `/opt/anzograph` as the install location because that will result in an `/opt/anzograph/anzograph` directory.

6. Press **Enter** to select the default installation directory or specify an alternate location and then press **Enter**. **When deploying a cluster, make sure you install Graph Lakehouse in the same location on each server in the cluster.**

Next the installer asks if you are installing Graph Lakehouse on a single server or a cluster:

```
Type of server being installed.  
Server Installation Type  
Standalone [1, Enter], Cluster Leader [2], Cluster Compute/Worker [3]
```

7. If you are deploying Graph Lakehouse on a single server, press **Enter** or type **1** (the default) and then press **Enter**. When deploying a cluster, specify whether the server you are currently installing is the leader node or a compute/worker node.

Type **2** if the role of the server is the leader node, or type **3** if the role of the server is a compute or worker node. Then press **Enter** to continue.

The installer now prompts you to set up the user credentials for the administrator:

```
Set up the Graph Lakehouse Admin user.  
Graph Lakehouse Admin user  
[admin]
```

8. Press **Enter** to accept the default user name, **admin**, or type another user name (case-sensitive) and press **Enter**. The installer asks you to create a password for the Admin user.

```
Graph Lakehouse Admin password
```

9. Enter the case-sensitive password for the Admin user and then press **Enter**. It is recommended that you create a non-trivial secure password that includes some combination of upper and lower case letters and digits.

Important

There is no limitation placed on the length of the password you can specify. However, some special characters, such as \$ and *, have a specific meaning to the host server operating system and should be avoided. For security purposes, the installer does not echo characters entered as you type them.

Next, the installer asks if Graph Lakehouse will be used with Anzo:

```
Is this Graph Lakehouse installation intended for use with Anzo?  
Yes [y, Enter], No [n]
```


10. Type **n** (No) to install Graph Lakehouse as a standalone deployment without Anzo and then press **Enter**. Type **y** (Yes) or press **Enter** if the Graph Lakehouse installation will be used with Anzo.

Note

Prompts differ based on whether the Graph Lakehouse installation will be used with or without Anzo. The following instructions describe options for deployments that are not integrated with Anzo. See [Deploying a Static Graph Lakehouse Cluster](#) in the Graph Studio Documentation for installation instructions of Graph Lakehouse for use with Graph Studio.

After specifying that Graph Lakehouse is not intended for use with Graph Studio, the installer prompts you to specify the Graph Lakehouse service user name. The default name is `anzograph`.

```
Setup Graph Lakehouse service user name
Graph Lakehouse service user
[anzograph]
```

11. Press **Enter** to accept the default name, `anzograph`, or type an alternate name and then press **Enter** if a different name is used for the account.

Next, the installer asks which components to install. The first option, Graph Lakehouse, installs only the backend graph database without the Jetty web-based frontend application. The second option, Graph Lakehouse Frontend, only installs the frontend application.

```
Which components should be installed?
1: Graph Lakehouse
2: Graph Lakehouse Frontend
Please enter a comma-separated list of the selected values or press [Enter]
for
the default selection:
[1]
```

Note

The instructions below describe the prompts that are displayed when both the backend and frontend are installed.

12. Press **Enter** to accept the default, **1**, if you do not want to install the user interface. Specify **1,2** and press **Enter** to install both Graph Lakehouse and the frontend application. Or specify **2** and press **Enter** to install the frontend only.

The installer now prompts you to set up the user credentials for the query user, a user without administrative access:

```
Set up the Graph Lakehouse Query User
Graph Lakehouse Query User
[query]
```

13. Press **Enter** to accept the default user name, **query**, or type another user name (case-sensitive) and press **Enter**. The installer asks you to create a password for the query user.

```
Graph Lakehouse Query password
```

14. Enter the case-sensitive password for the query user and then press **Enter**. It is recommended that you create a non-trivial secure password that includes some combination of upper and lower case letters and digits.

Next, the installer asks whether you want to enable access control (ACL) functionality:

```
ACL Configurations
Do you want to enable ACL?
Yes [y], No [n, Enter]
```

15. If you do not want to enable ACL, press **Enter** or type **n** and then press **Enter**. To enable the functionality, type **y** (Yes) and press **Enter**. See [Authentication and Access Control](#) for information.
16. If you chose to enable ACL, the installer prompts you to specify a password for the **superadmin** user. This user has the permission to manage roles and grant or revoke permissions to members in those roles.

```
Please enter ACL superadmin password
```

Enter the password (case-sensitive) you want to use for the **superadmin** user and then press **Enter**.

17. Next, if you chose to install the frontend, the installer prompts you to specify the HTTP/S ports to use for the application:

```
Choose the network HTTP/HTTPS ports and other configurations to use for the
Graph Lakehouse Frontend.
Graph Lakehouse Frontend HTTP Port
[8080]
```

18. Press **Enter** to select the default port, **8080**, as the port to use for HTTP connections, or specify another port if 8080 is in use. The installer prompts you to set the HTTPS port:

```
Graph Lakehouse Frontend HTTPS Port
[8443]
```

19. Press **Enter** to select the default, **8443**, as the port to use for HTTPS connections, or specify another port if 8443 is in use. Next the installer asks if you want to install the OData drivers. The OData drivers are used for creating Data on Demand endpoints. See [Access Data with OData Protocol](#) for information.

```
Install OData drivers?
Yes [y], No [n, Enter]
```

20. Press **Enter** or type **n** (No) to accept the default. Otherwise, type **y** (Yes) to install the OData drivers. If you chose to install the frontend console, the installer asks if you want to be able to connect to multiple Graph Lakehouse instances from the same user interface:

```
Enable ability to add and use multiple DB Contexts in the Frontend
(To be used without LDAP configuration)?
Yes [y], No [n, Enter]
```

21. Press **y** (Yes) if you want the frontend console to be allowed to access multiple Graph Lakehouse instances. Otherwise, press **Enter** or type **n** (No) to disable the functionality.

If you specified that this is a cluster, the installer prompts you to specify the IP addresses of all host servers in the cluster:

```
IP Address of nodes in cluster.  
Comma separated list of Cluster Node IP Addresses. Leader node address is  
always first. Order must be the same on all nodes in cluster.  
[XXX.n.n.XXX]
```

22. Enter a comma-separated list of all server IP addresses in the cluster. The first IP address in the list is the server designated as the leader node. Then press **Enter**.

Important

Specify the list of IP addresses in the same order on each server in the cluster. The configuration of host servers is saved to the `<install_path>/config/ip_addrs.conf` file on each server.

For standalone or leader node installations, the installer asks if you want to add any specific configuration settings to use when Graph Lakehouse starts up:

```
Extra configuration settings for server  
...  
WARNING: Typically, additional settings should only be added  
after consultation with Altair to address any  
specific requirements for Graph Lakehouse deployment in your environment  
or use for a specific application.  
[]
```

23. If you have a custom `setting=value` to add, enter it in the prompt, and then press **Enter**. The installer asks if you want to automatically start the Graph Lakehouse services after it completes the installation. First it asks if you want to start the **azgmgrd** service. This service provides system management and communication functions and is run on all host servers:

```
Start azgmgrd service?  
Do you want to start the azgmgrd systemd service?  
Yes [y], No [n, Enter]
```

24. Type **y** (Yes) and press **Enter** to start the system management service. If this is a single server (standalone) or leader node install the installer prompts you to start the **anzograph**

service. This is the database service. In a cluster, only the leader node runs this service, and the leader starts the database on the compute/worker nodes.

```
Start AnzoGraph service?
Do you want to start the AnzoGraph systemd service?
Yes [y], No [n], Enter]
```

25. Type **y** (Yes) and press **Enter** to start the database service. If you installed the frontend, the installer asks if you want to start the **jetty** service. This service provides management functions for the jetty frontend application.

```
Start Frontend/jetty service?
Do you want to start the AnzoGraph frontend systemd service?
Yes [y], No [n], Enter]
```

26. Type **y** (Yes) and press **Enter** to start the jetty service. The installer begins installing Graph Lakehouse based on your responses.

```
Extracting files...
```

When the installer is finished, it displays the following message.

```
Setup has finished installing Graph Lakehouse on your computer.
Finishing installation...
```

The installer creates an `anzograph` subdirectory in the install location that you specified. That directory contains the Graph Lakehouse executables, configuration, logs, and other program files. The frontend, example systemd service and repo files, and the uninstall script are saved at the same level as the `anzograph` directory. Next, complete the required and optional post-configuration steps. See [Post-Installation Configuration](#).

Post-Installation Configuration

This topic provides instructions for completing the required and optional post-installation configuration of Graph Lakehouse.

- [Installing the C++ Dependencies](#)
- [Optimizing the Linux Kernel Configuration for Graph Lakehouse](#)
- [Configuring the Graph Lakehouse Services and Starting the Database](#)

Installing the C++ Dependencies

Dependencies are required to be installed on all servers in the cluster to support the C++ extensions that Graph Lakehouse offers, including the remote read (load) and write service, the Data Science functions, and the integration with Apache Arrow. The installer provides a .repo file to aid you in configuring the Cambridge Semantics repository and installing the required software packages with or without internet access.

Note

The ability to write to the `/etc/yum.repos.d` directory requires root access permissions. See [Adding, Enabling, and Disabling a YUM Repository](#) in the Red Hat documentation for more information on defining and using yum repositories.

The following packages need to be installed on each host server in the cluster:

```
libarchive
libarmadillo12
libboost_filesystem1_80_0
libboost_iostreams1_80_0
libboost_system1_80_0
libflatbuffers2
libhdfs3
libnfs13
libserd-0-0
libsmb2
shadow-utils
```

- [Install Dependencies via Internet Access to the Cambridge Semantics Repository](#)
- [Install Dependencies without Internet Access via the Repository Mirror \(tarball\)](#)

Install Dependencies via Internet Access to the Cambridge Semantics Repository

Follow the steps below if the Graph Lakehouse servers have external internet access and you want to install the dependencies directly from the Cambridge Semantics repository.

1. Copy the **csi-obs-cambridgesemantics-udxcontrib.repo** file from the `<install_path>/pre-req/yum.repos.d` directory to the `/etc/yum.repos.d` directory. For example, the following command copies the file from the default installation path to `/etc/yum.repos.d`:

```
sudo cp /opt/altair/pre-req/yum.repos.d/csi-obs-cambridgesemantics-udxcontrib.repo /etc/yum.repos.d
```

2. Next, run the following command to enable the repository and install the required packages:

```
sudo dnf install --enablerepo=crb $(cat <install_path>/pre-req/rh9-anzograph-requirements.txt)
```

For example, on a server where Graph Lakehouse is installed in the default location:

```
sudo dnf install --enablerepo=crb $(cat /opt/altair/pre-req/rh9-anzograph-requirements.txt)
```

3. Repeat these steps on all servers in the cluster.

Install Dependencies without Internet Access via the Repository Mirror (tarball)

Follow the steps below if the Graph Lakehouse servers do not have external internet access and you want to install the dependencies from the mirrored Cambridge Semantics repository. The steps below give instructions for copying the repository to each Graph Lakehouse host server and configuring the `.repo` file accordingly. You can also chose to set up the mirror on a remote server that each of the Graph Lakehouse servers can access.

1. From a computer that does have internet access, download the dependency tarball, **csi-obs-cambridgesemantics-udxcontrib.rocky9.tar.xz**, from the following Cambridge Semantics Google Cloud Storage location: <https://storage.googleapis.com/csi-anzograph/udx/csi-os->

[contrib/rocky9/2023-03/20230321945/csi-obs-cambridgesemantics-udxcontrib.rocky9.tar.xz](https://storage.googleapis.com/csi-anzograph/udx/csi-os-contrib/rocky9/2023-03/20230321945/csi-obs-cambridgesemantics-udxcontrib.rocky9.tar.xz).

You can run the following cURL command to download the tarball:

```
curl -OL https://storage.googleapis.com/csi-anzograph/udx/csi-os-contrib/rocky9/2023-03/20230321945/csi-obs-cambridgesemantics-udxcontrib.rocky9.tar.xz(.sha512)
```

2. Also from the computer that has internet access, download the **repomd.xml.key** from the following Cambridge Semantics Google Cloud Storage location:

<https://storage.googleapis.com/csi-rpmm-dpd/CambridgeSemantics:/UDXContrib/ubi-9/repodata/repomd.xml.key>.

You can run the following cURL command to download the file:

```
curl -OL https://storage.googleapis.com/csi-rpmm-dpd/CambridgeSemantics:/UDXContrib/ubi-9/repodata/repomd.xml.key
```

3. On each of the Graph Lakehouse servers, create a directory called `/tmp/repo`.
4. Copy **csi-obs-cambridgesemantics-udxcontrib.rocky9.tar.xz** to the `/tmp/repo` directory on each server.
5. Then run the following command to unpack the tarball in the `/tmp/repo` directory:

```
tar -xvf csi-obs-cambridgesemantics-udxcontrib.rocky9.tar.xz
```

The files are unpacked into subdirectories under `/tmp/repo/dl/rocky9/csi-obs-cambridgesemantics-udxcontrib`.

6. Next, copy the **repomd.xml.key** file to the `/tmp/repo/dl/rocky9/csi-obs-cambridgesemantics-udxcontrib` directory on each of the Graph Lakehouse servers.
7. Now, open the **csi-obs-cambridgesemantics-udxcontrib.repo** file in the `<install_path>/examples/yum.repos.d` directory. The contents of the file are shown below:

```
[csi-obs-cambridgesemantics-udxcontrib]
name=Contrib directory for CambridgeSemantics AnzoGraph UDX dependencies
baseurl=https://storage.googleapis.com/csi-rpmm-dpd/CambridgeSemantics:/UDXContrib/ubi-9
```



```
gpgkey=https://storage.googleapis.com/csi-rpmmmd-  
pd/CambridgeSemantics:/UDXContrib/ubi-9/repodata/repomd.xml.key  
gpgcheck=1  
enabled=1
```

8. Edit the **csi-obs-cambridgesemantics-udxcontrib.repo** file contents to replace the **baseurl** and **gpgkey** values so that they point to the repo files that you unpacked in the `/tmp/repo` directory. In addition, change the **gpgcheck** and **enabled** values from 1 to 0. The contents of the updated file are shown below:

```
[csi-obs-cambridgesemantics-udxcontrib]  
name=Contrib directory for CambridgeSemantics AnzoGraph UDX dependencies  
baseurl=file:///tmp/repo/dl/rocky9/csi-obs-cambridgesemantics-udxcontrib  
gpgkey=file:///tmp/repo/dl/rocky9/csi-obs-cambridgesemantics-  
udxcontrib/repomd.xml.key  
gpgcheck=0  
enabled=0
```

9. Save and close the file.
10. Copy **csi-obs-cambridgesemantics-udxcontrib.repo** from `<install_path>/pre-req/yum.repos.d` to the `/etc/yum.repos.d` directory. For example, the following command copies the file from the default installation path to `/etc/yum.repos.d`:

```
sudo cp /opt/altair/pre-req/yum.repos.d/csi-obs-cambridgesemantics-  
udxcontrib.repo /etc/yum.repos.d
```

11. Next, run the following command to enable the repository and install the required packages:

```
sudo dnf install --enablerepo=crb $(cat <install_path>/pre-req/rh9-anzograph-  
requirements.txt)
```

For example, on a server where Graph Lakehouse is installed in the default location:

```
sudo dnf install --enablerepo=crb $(cat /opt/altair/pre-req/rh9-anzograph-  
requirements.txt)
```

Repeat the steps above as needed to install the dependencies on all servers in the cluster.

Optimizing the Linux Kernel Configuration for Graph Lakehouse

To streamline the configuration of the operating system for peak Graph Lakehouse performance, the installer includes a **tuned** profile that you can activate. Tuned is a daemon program that uses the `udev` device monitor to statically and dynamically tune operating system settings based on the specified profile. It is highly recommended that you activate the Graph Lakehouse tuned profile.

Tip

[Tuned](#) and [Performance Tuning with Tuned and Tuned-ADM](#) in the RedHat Performance Tuning Guide provide an overview of the tuned daemon and more information on using the tuned service to improve performance of specific workloads.

Activating the Tuned Profile

The profile, called **azg**, is in the `<install_path>/examples/tuned-profile` directory and consists of two files: `tuned.conf` and `additional-tuneables.sh`. For details about the files, see [Tuned Profile Reference](#) below.

Note

The ability to write to the `/etc/tuned` directory and activate the tuned profile requires root access permissions.

To activate the azg profile, follow the steps below. Complete these steps on all servers in the cluster:

1. **If you ran the installer with root (sudo) privileges, you can skip this step.** The installer copied the tuned profile to the `etc/tuned` directory but it did not automatically activate the profile. If you ran the installer as a non-root user, copy the `azg` directory from `<install_path>/examples/tuned-profile` to the `/etc/tuned` directory. For example, the following command copies `azg` from the default installation path to `/etc/tuned`:

```
sudo cp -r /opt/altair/examples/azg /etc/tuned
```

- Next, tuned is installed by default with RHEL 9. If you are using Rocky9, you may need to install tuned. You can run the following command to install the program:

```
sudo dnf install -y tuned
```

- Run the following command to activate the azg profile:

```
sudo tuned-adm profile azg
```

The host servers are now configured to use the tuned profile that is optimal for Graph Lakehouse.

Tip

To disable tuned profiles, you can run the following command:

```
sudo tuned-adm off
```

After running the command, no tuned profiles will be active.

Tuned Profile Reference

This section describes the tuned Graph Lakehouse profile files and the kernel configuration changes that they apply.

tuned.conf

The table below describes the Linux kernel configuration settings that are modified by tuned.conf.

Setting	Description	AZG Profile Change
vm.dirty_ratio	Specifies the percentage of system memory that can be occupied by "dirty" data before flushing the cache to disk. Dirty data are pages in memory that have been updated and do not match what is stored on disk.	Reduces <code>vm.dirty_ratio</code> to 2% to increase the frequency with which the system cache is flushed.
vm.swappiness	Controls the tendency of the kernel to move processes out of physical memory and onto the	Sets <code>vm.swappiness</code> to

Setting	Description	AZG Profile Change
	swap disk. A value of 0 means the kernel avoids swapping processes out of physical memory for as long as possible. A value of 100 tells the kernel to aggressively swap processes out of physical memory to the swap disk.	30.
vm.max_map_count	Sets the limit on the maximum number of memory map areas a process can use. Since Graph Lakehouse is memory intensive, it may reach the default maximum map count of 65535 and be shut down by the operating system.	Increases <code>vm.max_map_count</code> to 2097152.
net.ipv4.tcp_rmem	Controls the size of the receive buffer for TCP connections. It sets the minimum, default, and maximum sizes of the buffer in bytes.	Sets <code>tcp_rmem</code> to "4096 87380 16777216".
net.ipv4.tcp_wmem	Controls the size of the send buffer for TCP connections. It sets the minimum, default, and maximum sizes of the buffer in bytes.	Sets <code>tcp_wmem</code> to "4096 16384 16777216".
net.ipv4.udp_mem	Controls the amount of memory that can be allocated for the kernel's UDP buffer. It sets the minimum, default, and maximum sizes of the buffer in bytes.	Sets <code>udp_mem</code> to "3145728 4194304 16777216".
transparent_hugepages	Controls whether Transparent Huge Pages (THP) is enabled or disabled system-wide. When THP is enabled system-wide, it can dramatically degrade Graph Lakehouse performance.	Disables THP by setting <code>transparent_hugepages</code> to never.

additional-tunables.sh

The additional-tuneables.sh script is called by tuned.conf and configures the following Linux kernel configuration settings so that they are optimal for Graph Lakehouse.

Setting	Description	AZG Profile Change
overcommit_memory	Controls whether obvious overcommits of the address space are allowed.	Sets <code>overcommit_memory</code> to 0 to ensure that very large overcommits are not allowed but some overcommits can be used to reduce swap usage.
overcommit_ratio	Controls the percentage of memory that is allowed to be used for overcommits.	Sets <code>overcommit_ratio</code> to 50%.
transparent_hugepage/defrag	Though the tuned profile disables Transparent Huge Pages (THP) system-wide, this setting controls whether huge pages can still be enabled on a per process basis (inside <code>MADV_HUGEPAGE</code> <code>madvise</code> regions).	Sets <code>transparent_hugepage/defrag</code> to <code>madvise</code> so that the kernel only assigns huge pages to individual process memory regions that are specified with the <code>madvise()</code> system call.
tcp_timestamps	Controls whether TCP timestamps are enabled or disabled.	Sets <code>tcp_timestamps</code> to 0, which disables TCP timestamps in order to reduce performance spikes related to timestamp generation.

Configuring the Graph Lakehouse Services and Starting the Database

Note

When running the installer as root (**sudo**), the installer automatically creates Graph Lakehouse systemd services in the `/etc/systemd/system` directory for `azgmgrd`, `anzograph`, and `jetty` (if the frontend is installed). In addition, the installer asks if you want to automatically start the services at the end of the installation. If Graph Lakehouse is already running, see [Get Started](#) for next steps.

When running the installer as a non-root user, the installer does not automatically create systemd services in the `/etc/systemd/system` directory, but example service files are available in the `<install_path>/examples/systemd-services` directory for you to customize and enable manually. It is highly recommended that you implement the services because they are configured to tune user resource limits (ulimits) for the Graph Lakehouse process as well as set `$JAVA_HOME` to the JVM path.

Important

The `azgmgrd` service needs to be enabled on all host servers. But the `anzograph` and `jetty` services should only be enabled on single server environments and on the leader node if you have a cluster. The `anzograph` and `jetty` services should not be invoked on the compute/worker nodes in a cluster.

After tailoring the service files to your environment, follow these steps to enable and start the services:

1. Copy the `azgmgrd` service file from the `<install_path>/examples/systemd-services` directory to the `/etc/systemd/system` directory on each server in the cluster.
2. Then run the following commands on each server in the cluster to start the `azgmgrd` service and enable the service to start automatically each time the host server is started.

```
sudo systemctl start azgmgrd.service
```

```
sudo systemctl enable azgmgrd.service
```

3. For single server environments and on the leader node in a cluster, copy the `anzograph` and `jetty` files from the `<install_path>/examples/systemd-services` directory to the `/etc/systemd/system` directory. **Do not copy these services on compute/worker nodes.**
4. Next, run the following commands to start the `anzograph` and `jetty` services and enable the services to start automatically each time the host server is started.

```
sudo systemctl start anzograph.service
```

```
sudo systemctl enable anzograph.service
```

```
sudo systemctl start jetty.service
```

```
sudo systemctl enable jetty.service
```

Important

If you do not employ the `systemd` services, make sure that you manually set `$JAVA_HOME` to the JVM installation path. If you set up a cluster, set `$JAVA_HOME` on each server in the cluster.

For next steps, see [Get Started](#).

Uninstalling and Updating Graph Lakehouse

This topic provides instructions for uninstalling and updating Graph Lakehouse.

- [Upgrading Graph Lakehouse](#)
- [Uninstalling Graph Lakehouse](#)

Upgrading Graph Lakehouse

A key area of growth in Graph Lakehouse is the development and support of custom, user-managed extensions, such as the Graph Data Interface for virtualization. Most Graph Lakehouse releases include revisions to the API and prepackaged extensions. Because of the frequency of Graph Lakehouse updates and because the extensions directory (`<install_path>/lib/udx`) is user-managed rather than Graph Lakehouse- or installer-controlled, you must uninstall the existing version of Graph Lakehouse and then install the new version. **In-place upgrades are not supported.**

Follow the instructions below to back up any custom files and remove the installation directory before installing the new version.

1. First, run the following commands to stop the database and the system management daemon. On a cluster, run these commands on the leader node:

```
sudo systemctl stop anzograph
```

```
sudo systemctl stop azgmgrd
```

2. Next, if you have custom files, such as certificates in `<install_path>/config` or JDBC drivers in the `<install_path>/lib/udx` directory, make a backup copy of those files. Make sure that you choose a backup location that is outside of the Graph Lakehouse installation path. After installing the new version of Graph Lakehouse, you can place the custom files back into the appropriate directories.

Note

If you have modified the settings file, `<install_path>/config/settings.conf`, Altair Engineering Inc. recommends that you make a backup copy of the file on the leader server so that you can refer to it when configuring the new deployment. As a best practice, however, do not overwrite `settings.conf` in the new version of Graph Lakehouse with the backup copy from the previous version. Instead, Altair recommends that you apply all changes to the new file. Since new releases may add or remove settings or change the default value of certain settings, it is important to use the version of the file that was installed with the release.

3. Remove the Graph Lakehouse installation directory from the file system. You can remove the software by deleting the installation directory or by running the `<install_path>/uninstall` script as described below in [Uninstalling Graph Lakehouse](#). On a cluster, uninstall Graph Lakehouse on all nodes.
4. Now that Graph Lakehouse is uninstalled, follow the instructions in [Install Graph Lakehouse](#) to install the new version.

Uninstalling Graph Lakehouse

This section provides instructions for uninstalling Graph Lakehouse. On clusters, complete steps 2 through 4 below on each server in the cluster.

1. First, make sure the database and system management daemon processes are stopped. Run the following commands to stop the services. On a cluster, run these commands on the leader server:

```
sudo systemctl stop anzograph
```

```
sudo systemctl stop azgmgrd
```

2. Next, if you have custom files, such as JDBC drivers or user-defined extensions in the `<install_path>/lib/udx` directory, make a backup copy of those files on the leader node. Make sure that you choose a backup location that is outside of the Graph Lakehouse

installation path.

If you install a new version of Graph Lakehouse, you can place the custom files back into the appropriate directory on the leader node.

3. Run the following command to begin the uninstall process:

```
sudo ./<install_path>/uninstall
```

The script asks if you want to proceed:

```
Do you want to proceed with Graph Lakehouse installation?  
OK [o, Enter], Cancel [c]
```

4. Press **Enter** to confirm that you want to uninstall Graph Lakehouse.

The wizard asks if you want to clear the installation directory and user and configuration files:

```
Are you sure you want to completely remove Graph Lakehouse and all of its  
components?  
Yes [y, Enter], No [n]
```

5. Altair recommends that you remove all installation and configuration files. Press **Enter** to remove the entire installation directory as well as all configuration and user files.

The wizard uninstalls Graph Lakehouse.

IBM Cloud Pak Deployments

This topic provides instructions for deploying Graph Lakehouse with IBM Cloud Pak.

Prerequisites

Before deploying Graph Lakehouse install the following applications on your workstation:

- **Helm Version 2 or 3:** See [Installing Helm](#) in the Helm documentation.
- **Docker Engine:** See the [Docker documentation](#) for instructions.
- **OpenShift CLI:** See the [OpenShift documentation](#) for installation instructions.

Deploying Graph Lakehouse with Cloud Pak

Follow the instructions below to deploy Graph Lakehouse.

1. Log in to the OpenShift client:

```
oc login openshiftURL:port
```

2. Run the following command to retrieve the internal registry information:

```
oc registry info
```

3. Add the internal registry to the insecure registry list in the Docker daemon. For instructions, see [Test an insecure registry](#) in the Docker documentation.

4. Run the following command to log in to Docker:

```
docker login -u admin -p $(oc whoami -t) $(oc registry info)
```

5. To offer versatility for different types of environments and deployment preferences, Altair provides three Graph Lakehouse DB container images:

- **anzograph (all-in-one image):** The all-in-one image includes the front end (user interface) as well the back end (database) in one image.

- **anzograph-frontend (user interface):** The front end image includes the user interface only. One front end client can connect to multiple Graph Lakehouse DB instances, or multiple users can deploy the front end locally and use it to access a central Graph Lakehouse DB cluster.
- **anzograph-db (back end/database):** The back end image includes the database only. If you have existing client applications to use with Graph Lakehouse DB and do not need the front end, you can deploy the database by itself.

See [Red Hat Container Registry Authentication](#) for information about accessing the Red Hat registry. Then run the following commands as needed to pull the desired Graph Lakehouse DB images from `registry.connect.redhat.com`:

```
docker pull registry.connect.redhat.com/cambridgesemantics/anzograph
```

```
docker pull registry.connect.redhat.com/cambridgesemantics/anzograph-frontend
```

```
docker pull registry.connect.redhat.com/cambridgesemantics/anzograph-db
```

6. Run the following Docker commands to tag the images and push them to the internal repository. Run the commands for each of the images that you want to push to the repository:

Note

Run the `docker images` command to return the list of images and view the image IDs, image names, and tags.

```
docker tag image_id internal_repo/cambridgesemantics/image_name:tag
```

```
docker push internal_repo/cambridgesemantics/image_name:tag
```

7. Run the following command to create an OpenShift Service Account:

```
oc -n namespace create serviceaccount service_account_name
```

8. Run the following command to provide access to the service account to pull images from the internal registry:

```
oc policy add-role-to-user \
system:image-puller system:serviceaccount:namespace:service_account_name \
--namespace=cambridgesemantics
```

9. Create a Security Context Constraint (SCC) for the service account to be able to start the Graph Lakehouse container as root. Note that the actual service in the container runs unprivileged.

- a. Create a file called `scc.yml` and add the following contents to the file:

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints

metadata:
  name: csi-anyuid
  namespace: namespace
priority: 10
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
fsGroup:
  type: RunAsAny
users:
- system:serviceaccount:namespace:service_account_name
```

- b. Save the file and then run the following command to give OpenShift the SCC resource specification:

```
oc create -f scc.yml
```

10. Configure Helm for use with your version of Cloud Pak. First, change directories to the Helm directory:

```
cd ~/.helm
```

Then run the appropriate commands below depending on your version of Cloud Pak:

Cloud Pak 2.5

```
tiller_pod=$(oc get po | grep icpd-till | awk '{print $1}');
oc cp ${tiller_pod}:etc/certs/..data/helm.cert.pem cert.pem;
oc cp ${tiller_pod}:etc/certs/..data/helm.key.pem key.pem
```

Cloud Pak 3.0+

```
cd $HELM_HOME && ocget secret helm-secret -n $TILLER_NAMESPACE -o yaml|grep
-A3 '^data:'|tail -3 | awk -F: '{system("echo \"$2\" |base64 --decode >
"$1) }'
```

```
export HELM_TLS_CA_CERT=$HELM_HOME/ca.cert.pem
```

```
export HELM_TLS_CERT=$HELM_HOME/helm.cert.pem
```

```
export HELM_TLS_KEY=$HELM_HOME/helm.key.pem
```

```
export TILLER_NAMESPACE=zen
```

```
helm version --tls
```

11. Deploy Graph Lakehouse DB with Helm. See [Deploy Graph Lakehouse with Helm](#) for instructions.

Important

The Graph Lakehouse Helm chart includes **sample-values** files. Use the **values04-ibm-cloud-pak-data.yaml** sample file for your deployment and customize the values as needed. In the .yaml file, make sure that you update the **serviceAccountName** value with the OpenShift Service Account (**service_account_name** from the previous steps).

12. When you have finished deploying Graph Lakehouse DB, run the following command to create a route to expose the Graph Lakehouse DB service:

```
oc create route passthrough --service=anzograph-helm_release_name-frontend-lb
--port=https anzograph
```

13. (Optional) Create a route to expose the Graph Lakehouse DB Open Data Protocol (OData) service. The OData service enables users to generate OData-based feeds that can be used to access Graph Lakehouse programmatically via a RESTful API or from business

intelligence tools such as TIBCO Spotfire, Tableau, and Microsoft Power BI. Run the following command to create an OData route:

```
oc create route passthrough --service=anzograph-helm_release_name-frontend-lb  
--port=http odata
```

Get Started

After deploying Graph Lakehouse, you can get a quick start to loading data and running SPARQL queries by using the GUI-based Query & Admin Console or the command line interface (CLI). The topics in this section help you get started with Graph Lakehouse by providing five-minute tutorials using the console and the CLI. This section also includes instructions for upgrading an Graph Lakehouse license and offers sample data and tutorials.

In this section:

- [Quickstart with the Query Console](#) 97
- [Quickstart with the CLI](#) 103
- [Licensing Methods](#) 105
- [Install or Upgrade a License](#) 107
- [Learn SPARQL](#) 117
- [Sample Data and Tutorials](#) 148

Quickstart with the Query Console

After deploying Graph Lakehouse, you can get a quick start to loading data and running SPARQL queries by using the GUI-based Query & Admin Console. This brief tutorial introduces you to the application and gets you started with loading data and running SPARQL queries.

- 1. [Log in to the Console](#)
- 2. [Load and Query Sample Data](#)

Log in to the Console

The Graph Lakehouse console supports the latest Safari, Google Chrome, Mozilla Firefox, and Microsoft Edge browsers.

- 1. Depending on whether you deployed Graph Lakehouse using the RHEL/Rocky installer, Docker, or Kubernetes with Helm, follow the appropriate instructions below to access the user interface:

Deployment	Instructions
Desktop Container Engine	You can use the desktop application to open the Graph Lakehouse container in a browser, or open a browser and go to the following URL: <code>http://127.0.0.1</code>. If you specified a port other than 80 for the host HTTP port when you deployed Graph Lakehouse, include that port in the URL. For example, <code>http://127.0.0.1:8888</code>.
Linux Container Engine	If you are accessing a container image on a remote Linux host, note the IP address of the host, and then open a browser and go to the following URL: <code>https://<host_IP_address></code>. If you mapped the container's HTTPS (8443) port to port 443 on the host when you deployed Graph Lakehouse, you do not need to specify

Deployment	Instructions
	a port. If you specified a port other than 443, include the port in the URL. For example, <code>https://10.100.0.1:8888</code>. Tip If you are using Docker locally on a Linux machine and need to know the IP address of the Graph Lakehouse container, you can run the following command: <pre>sudo docker inspect <container_name> grep '"IPAddress"' head -n 1</pre> For example: <pre>sudo docker inspect anzograph grep '"IPAddress"' head -n 1 "IPAddress": "172.17.0.2"</pre>
Kubernetes with Helm	Using the Graph Lakehouse cluster or external IP obtained from the <code>kubectl get service</code> command, open a browser and go to the following URL: <code>https://<IP_address></code> .
EL9 Installer	Use the following URL to access the console: <code>https://<host_IP_address>:<https_port></code> .

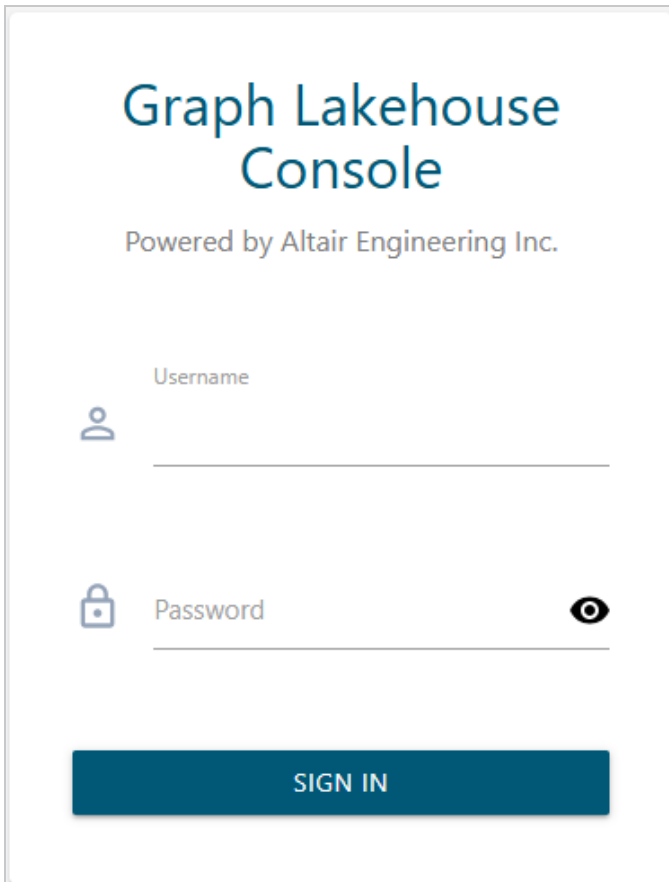
Note

If you use the HTTPS endpoint, your browser may warn you that the connection is not private. The warning is normal behavior. Graph Lakehouse servers use self-signed certificates, and browsers automatically trust only the certificates from well-known certificate authorities. For more information about certificate warnings, see [Security](#)

[Certificate Errors](#) on the DigiCert website. Depending on your browser, follow the appropriate instructions below to either bypass the warning and continue to the console or configure the browser to trust the certificate:

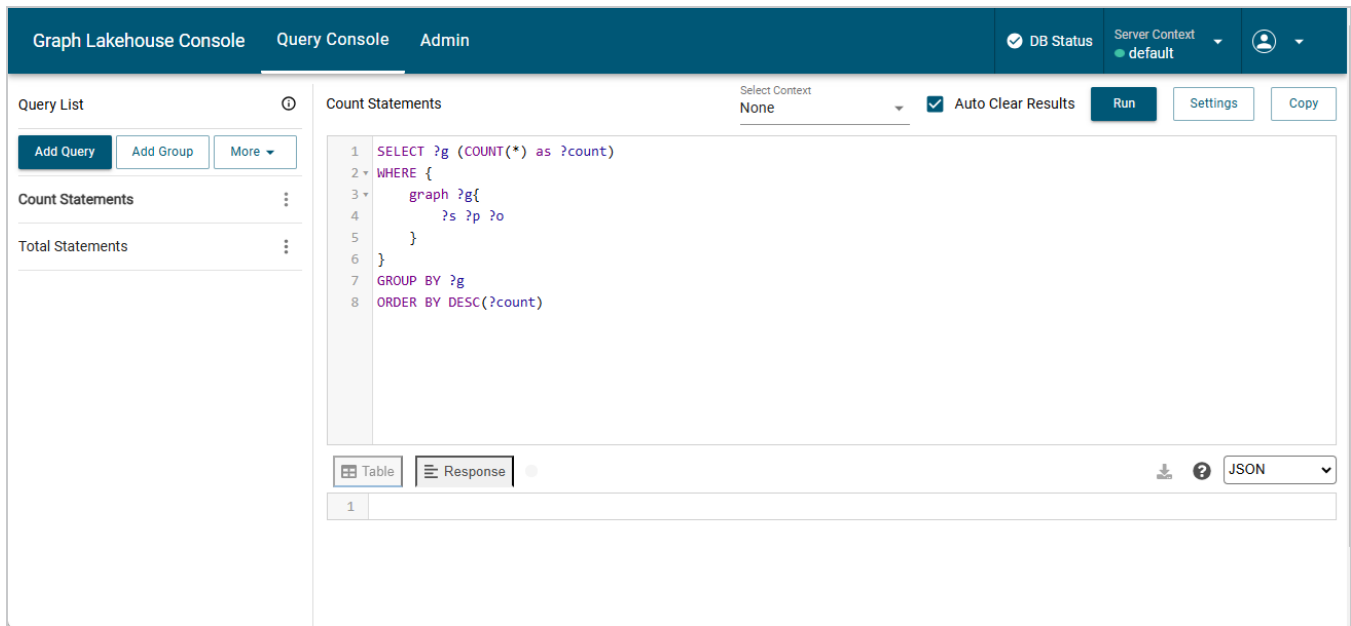
- On Chrome, click the **Advanced** link at the bottom of the page and then click the **Proceed to *ip* (unsafe)** link.
- On Safari, click the **Show Details** button and then click **Visit Website** to import the certificate.
- On Firefox, click **Advanced** and then click **Add Exception**. On the next screen, click **Add Security Exception** to confirm the exception for the endpoint.

The browser displays the login screen.



The image shows a login screen for the Graph Lakehouse Console. At the top, the text "Graph Lakehouse Console" is displayed in a large, blue, sans-serif font. Below this, in a smaller, grey font, it says "Powered by Altair Engineering Inc.". The login form consists of two input fields. The first field is labeled "Username" and has a person icon to its left. The second field is labeled "Password" and has a lock icon to its left and a toggle icon (an eye) to its right. Below the password field is a dark blue button with the text "SIGN IN" in white, uppercase letters.

2. On the login screen, type the username and password for the Admin user that was set up during deployment. If you deployed Graph Lakehouse with Docker, use **admin** as the user name and **Passw0rd1** as the password.
3. Then click **Sign In**. The browser displays the Graph Lakehouse End User License Agreement (EULA).
4. Review the agreement, and then click **Accept** to agree to the EULA terms and proceed to the Query & Admin Console.



The left-side navigation pane of the Query Console shows two default queries, Count Statements and Total Statements. The Count Statements query returns a list of each named graph and the number of statements in the graph. The Total Statements query returns the total number of statements in all named graphs. Click a query in the navigation pane to open the query in the main window. You can edit or overwrite the default queries, or you can click **Add Query** to create a new query from scratch. To run a query, click the **Run** button (▶) at the top of the screen. The results are displayed at the bottom of the screen.

Load and Query Sample Data

1. In the Query Console, click **Add Query** to add a new, blank query (or you can overwrite one of the default queries). Copy the following query and then paste it into the query text box. This query loads the sample Tickit data from the **tickit.ttl.gz** directory on the Graph Lakehouse file system. This simple data set captures sales activity for a fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts.

```
LOAD <dir:/<install_path>/etc/tickit.ttl.gz>
INTO GRAPH <http://anzograph.com/tickit>
```

2. Once you have pasted the query, replace the placeholder `<install_path>` with the path to the Graph Lakehouse installation directory. With Kubernetes or Docker container deployments, the installation path is `/opt/anzograph`:

```
LOAD <dir:/opt/anzograph/etc/tickit.ttl.gz>
INTO GRAPH <http://anzograph.com/tickit>
```

For RHEL/Rocky deployments, the installation path is specified during the installation. The default path is `/opt/altair/anzograph`:

```
LOAD <dir:/opt/altair/anzograph/etc/tickit.ttl.gz>
INTO GRAPH <http://anzograph.com/tickit>
```

Modify the path as needed if an alternate path was chosen.

3. Click the **Run** button to run the query and load the data. When the load completes, the console displays an **Update Successful** message.
4. In the console, replace the load statement with the following simple query that counts the number of triples in the Tickit data set. Then click **Run** to run the query:

```
SELECT (count(*) as ?number_of_triples)
FROM <http://anzograph.com/tickit>
WHERE { ?s ?p ?o }
```

For example:

Graph Lakehouse Console

Query Console

Admin

DB Status

Server Context

default

Query List

Add Query

Add Group

More

Count Statements

Total Statements

Query 3

Query 3

Select Context

None

Auto Clear Results

Run

Settings

Copy

1 SELECT (count(*) as ?number_of_triples)

2 FROM <http://anzograph.com/tickit>

3 WHERE { ?s ?p ?o }

Table

Response

1 result in 2.848 seconds

Query ID: 113

Simple view

Ellipse

Filter query results

Page size: 50

JSON

number_of_triples

1

5509095

Showing 1 to 1 of 1 entries

<

1

>

If you want to continue to work with the Tickit data and run more complex queries or view explanations of the query syntax, see [Working with SPARQL and the Tickit Data](#). For more information about loading data and to review load file requirements and recommendations, see [Load RDF Data from Files](#).

Quickstart with the CLI

After deploying Graph Lakehouse, you can get a quick start to loading data and running SPARQL queries by using the Graph Lakehouse command line interface (CLI). This brief tutorial introduces you to the CLI and gets you started with loading data and running SPARQL queries.

1. [Introduction to the CLI](#)
2. [Load and Query a Sample Data Set](#)

Introduction to the CLI

The command line client, `azgi`, uses SSL protocol to interact with the database. The client exists in the `<install_path>/bin` directory. In a container deployment, the installation path is `/opt/anzograph`. On RHEL/Rocky deployments, the installation path is customizable. The default path is `/opt/altair/anzograph`. In a cluster, use `azgi` on the leader node only.

Running the following command displays the options that `azgi` supports:

```
./<install_path>/bin/azgi -help
```

To get started, the following list describes the most frequently used `azgi` options:

- **`azgi -c "command"`**: Runs the command enclosed within the quotation marks. For example, the following command runs a SELECT query that returns the `?s ?p ?o` triple patterns for a graph named `graph`:

```
./<install_path>/bin/azgi -c "select * from <graph> where {?s ?p ?o}"
```

- **`azgi -f file.rq`**: Runs the query or queries saved in a file. For example, if a file called `query.rq` existed in the `/tmp` directory, the following command would run the query or queries saved in `query.rq`:

```
./<install_path>/bin/azgi -f /tmp/query.rq
```

The following examples guide you through using the `azgi -c` option to load RDF data. For more information about the command line interface, see [Use the Graph Lakehouse CLI](#).

Load and Query a Sample Data Set

The instructions below guide you through using the CLI to load the sample Tickit data set from the **tickit.ttl.gz** directory on the file system. This simple data set captures sales activity for a fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts.

1. Run the following command to load the Tickit Turtle files:

```
azgi -c "load <dir:/<install_path>/etc/tickit.ttl.gz> into graph
<http://anzograph.com/tickit>"
```

Where **<install_path>** is the path to the anzograph installation directory. For example:

```
azgi -c "load <dir:/opt/altair/anzograph/etc/tickit.ttl.gz> into graph
<http://anzograph.com/tickit>"
```

Graph Lakehouse loads the data from the files into memory, and the prompt returns when the load completes.

2. Run the following command to count the number of triples in the tickit graph:

```
azgi -c "select (count(*) as ?number) from <http://anzograph.com/tickit> where
{?s ?p ?o}"
```

```
number
-----
5509095
1 rows
```

If you want to continue to work with the Tickit data and run more complex queries or view explanations of the query syntax, see [Working with SPARQL and the Tickit Data](#). For more detailed information about loading RDF data and to review load file requirements and recommendations, see [Load RDF Data from Files](#).

Licensing Methods

Two licensing methods are supported for Graph Lakehouse:

- [Altair Units Licensing](#)
- [License Keys](#)

Altair Units Licensing

[Altair Units](#) is a value-based license management system enabling metered usage of an entire suite of products. All Altair Data Analytics products have the capability to use a single pool of recyclable Altair Units. The license units consumption of Graph Lakehouse depends on the CPU resources (the number of CPU cores on the host system). As of this version, Graph Lakehouse consumes 6 units per core.

Managed Altair Licensing

For Managed Altair Licensing, the License Server is hosted by Altair, and the application checks out the required license units by contacting the license servers in the [Altair One](#) cloud via HTTPS. For information on managing your licenses at the Altair One™ portal, see **Administrator Dashboard** and **Account Administration** in the [Altair One Documentation](#).

HTTPS connectivity must exist from the Graph Lakehouse host to four servers in the Altair cloud:

1. Managed Licensing Client: **client.hhwu.altair.com**
2. Managed Licensing Auth Sever1: **auth.hhwu.altair.com**
3. Altair License Activation System: **alas.admin.altairone.com**
4. Altair One Admin Portal: **auth.admin.altairone.com**

If you are using a proxy server for HTTPS connections, there is an option to specify a proxy in the application settings if necessary. If you are using a Proxy Auto-Configuration script, you can add exceptions for these servers in the proxy script to allow both inbound and outbound https traffic.

Access to the Altair One portal <https://altairone.com> from the web browser is required for at least one user to manage the account (authorize machine(s), add users, view the license usage, download usage history logs, etc.) and access software downloads, documentation, resources (videos, white papers, etc.), Customer Support portal and user community.

On-premises Altair License Server

If you opted for a self-hosted Altair License Server, the Graph Lakehouse application checks out the required license units by connecting via TCP to a license server with Altair License Manager running in your corporate network or in a cloud managed by your organization.

Altair License Manager is an LM-X license server built and distributed by Altair, which includes features such as License Distribution Service, automatic heartbeats, license borrowing, and High Availability Licensing. It uses a proprietary TCP/IP based protocol and uses port 6200 by default. The license file, which defines the licensed features and the available number of Altair units in the license pool, is tied to the MAC address of the license server host.

The Altair License Manager software installation & administration guide and release notes are available on the [Altair Documentation](#) website (search for "Licensing" in the product filter and select "Altair License Management System").

License Keys

Graph Lakehouse activation with a Cambridge Semantics license key is a legacy method supported in parallel with the new Altair Units Licensing. To obtain and apply a license key, please refer to the topic [Install or Upgrade a License](#).

Install or Upgrade a License

For the information on the license types available for Graph Lakehouse, see [Licensing Methods](#)

If you intend to use Graph Lakehouse with a legacy license key, please contact [Altair Engineering Customer Support](#) to request a new license key. This topic provides instructions for installing or upgrading a legacy license using the Graph Lakehouse user interface and the command line interface. It also provides instructions for activating an Altair Units License.

- [Activate Graph Lakehouse with Altair Units Licensing](#)
- [Install or Upgrade a Legacy License from the User Interface](#)
- [Install or Upgrade a Legacy License from the Command Line](#)

Activate Graph Lakehouse with Altair Units Licensing

Altair Units Licensing offers two methods - **self-hosted Altair License Server** (hosted on premises or in a customer-managed cloud) and **Managed Altair Licensing** (with license servers hosted by Altair and licenses managed through the [Altair One](#) portal). See [Licensing Methods](#) for details.

To enable Altair Units licensing in the Graph Lakehouse configuration, edit the `<install_path>/config/settings.conf` file. See the topic [Change System Settings](#) for detailed instructions on configuring Graph Lakehouse using this file.

1. In the `settings.conf` file, set the `enable_altair_licensing` value to `true`.

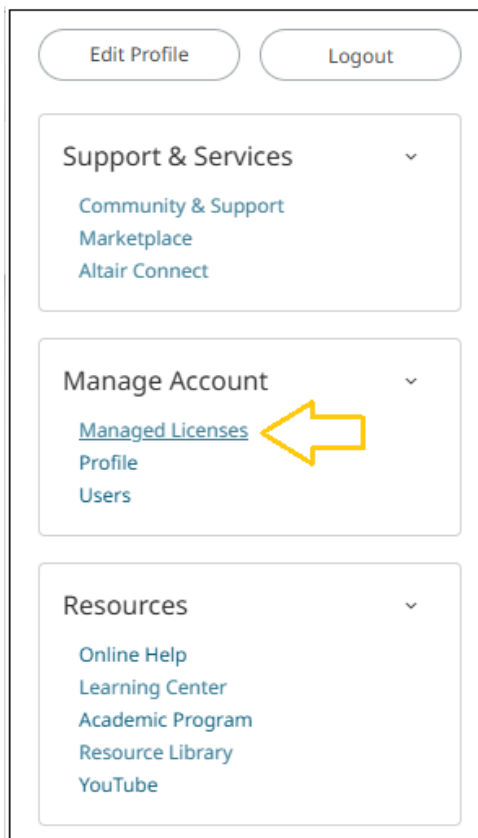
- If you are using a self-hosted Altair License Server, set the `license_server` configuration setting in `settings.conf` to `6200@<license_server_name_or_IP>`. The default license server port is 6200. Change it if the license server is configured to use a different port. Alternatively, set the `ALTAIR_LICENSE_PATH` environment variable to the same `port@server` value.
- If you are using managed Altair Licensing, authorize the Graph Lakehouse host using the **Altair License Utility** on the leader node of the cluster. The Altair License Utility (`almutil`) is in the `/usr/bin` directory of the Graph Lakehouse container. It can also be found in the

bin subdirectory of the Altair License Server program files

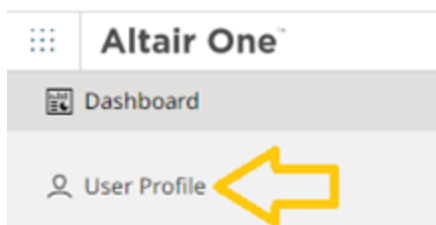
(/usr/local/altair/licensing2025.0/bin) if it is installed on the same host as Graph Lakehouse.

First, generate an authorization code from your [Altair One](#) account. If you are not registered on AltairOne, ask any registered user at your company to generate and send you an authorization code.

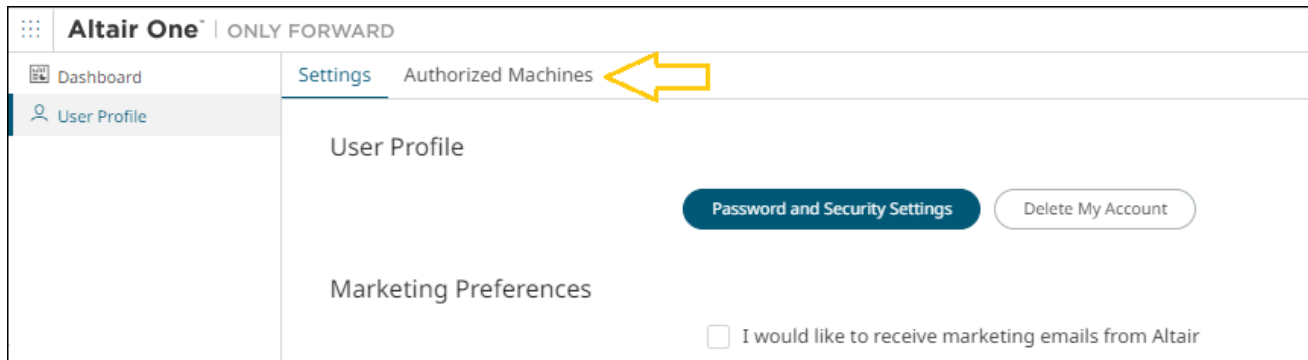
From a web browser on any machine, sign in to [Altair One](#) and click on the menu next to your name at the top-right corner. In the **Manage Account** section, click **Managed Licenses**.



In the left panel of the next screen, click **User Profile**.



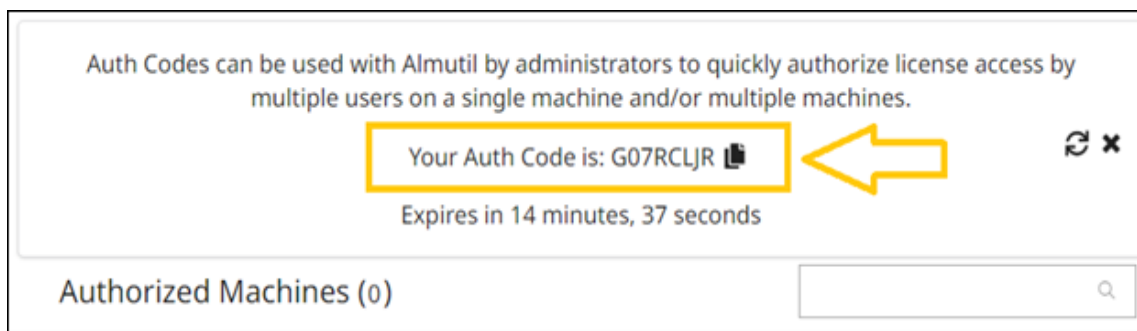
Click the **Authorized Machines** tab



In the Authorized Machines page, click the **Generate Auth Code** button.



The generated authorization code will be displayed. Copy the code to the clipboard. *Note:* The code is valid for 15 minutes, so you must perform the next step within 15 minutes of generating the code.



Change to the directory that contains `almutil`, for example `cd /usr/bin`.

Run the following command on the leader node of the cluster within 15 minutes of generating the code:

```
./almutil -alauth -system -code <auth_code>
```

This command authorizes the node for the use of your managed Altair license by generating an encrypted token written to a configuration file `altair_hostedhwu.cfg` whose default location is `/usr/local/altair`.

If you are using an http(s) proxy, set the proxy configuration for almutil **before** running almutil -alauth:

```
./almutil -system -host <proxy_host> -port <proxy_port>
```

The proxy info is saved in altair_hostedhwu.cfg. If your proxy configuration requires a proxy user ID and password, use the command:

```
./almutil -system -host <proxy_host> -port <proxy_port> -user <proxy_user> -  
passwd <password>
```

The encrypted credentials of the proxy user will be saved in the file altair_hostedhwu_ex.cfg in the same directory as altair_hostedhwu.cfg.

To test the connectivity with the Altair Cloud license servers, use the command:

```
./almutil -conntest
```

To print out the configuration information and test the validity of authorization token if found, use the command

```
./almutil -alauth -test
```

To display the current license statistics including current usage:

```
./almutil -licstat
```

Run the almutil command with no arguments to view the detailed usage help and advanced options.

2. Reinitialize Graph Lakehouse for the new license configuration to take effect. Reinitializing the database requires running the following system manager (azgctl) command on the file system. [How do I access the Graph Lakehouse file system with Docker?](#) If you are using a cluster, run the command on the leader node:

```
./<install_path>/bin/azgctl -restart -init
```

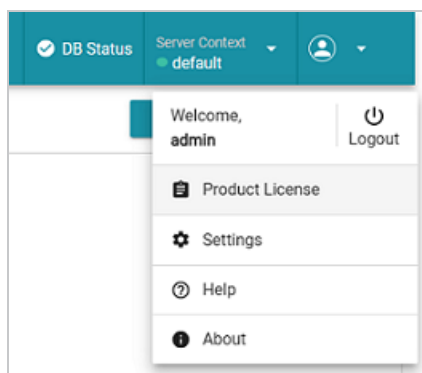
Install or Upgrade a Legacy License from the User Interface

First, open the user interface by following the appropriate instructions below according to your method of deployment:

Deployment	Instructions
Desktop Container Engine	You can use the desktop application to open the Graph Lakehouse container in a browser, or open a browser and go to the following URL: <code>http://127.0.0.1.</code> If you specified a port other than 80 for the host HTTP port when you deployed Graph Lakehouse, include that port in the URL. For example, <code>http://127.0.0.1:8888.</code>
Linux Container Engine	If you are accessing a container image on a remote Linux host, note the IP address of the host, and then open a browser and go to the following URL: <code>https://<host_IP_address>.</code> If you mapped the container's HTTPS (8443) port to port 443 on the host when you deployed Graph Lakehouse, you do not need to specify a port. If you specified a port other than 443, include the port in the URL. For example, <code>https://10.100.0.1:8888.</code> Tip If you are using Docker locally on a Linux machine and need to know the IP address of the Graph Lakehouse container, you can run the following command: <pre>sudo docker inspect <container_name> grep '"IPAddress"' head -n 1</pre> For example: <pre>sudo docker inspect anzograph grep '"IPAddress"' head -n 1</pre>

Deployment	Instructions
	<pre>"IPAddress": "172.17.0.2"</pre>
Kubernetes with Helm	Using the Graph Lakehouse cluster or external IP obtained from the <code>kubectl get service</code> command, open a browser and go to the following URL: <code>https://<IP_address></code> .
EL9 Installer	Use the following URL to access the console: <code>https://<host_IP_address>:<https_port></code> .

1. On the login screen, type the username and password for the Admin user. If you deployed Graph Lakehouse with a container engine like Docker, Podman, or Rancher, use **admin** as the user name and **Passw0rd1** as the password. Then click **Sign In**.
2. On the top right of the screen, click the User drop-down menu and select **Product License**.



The Product License screen is displayed. For example, if you are using a legacy license key license, the screen is as follows:

Product License

Altair Graph Lakehouse

Created: 03 Feb 2025

License Requirements

Enter Key

License Key

Apply

OR

Upload Key

Upload

License Information

License Name	Graph Lakehouse For Docker 32 GB License
Product Family	Graph Lakehouse
Feature SKU	AZG-10012
Start Date	03 Feb 2025
Max Nodes	1
Altair Graph Lakehouse Version	[2.0.0,4.0.0)
Registered	true
Max RAM	8192
Is Evaluation License	false

[Learn more about licensing](#)

Altair Engineering Inc.

Altair Graph Lakehouse Support:

<https://web.altair.com/support-contact>

[Request license](#)

Close

- On the Product License screen, click **License Requirements** and copy them to the clipboard. Click the **Request License** link to go to the [Contact Altair Support](#) web page.
- Contact Altair Customer Support to request the license for your system, providing the License Requirements copied at the previous step.

5. Once you received the license key or file from Altair License Admin: Paste the license key in the **License Key** field and click **Apply**. Alternatively, click the **Upload** button upload the file with the license key information. Graph Lakehouse displays a "License was updated successfully" message and the screen is refreshed to display the details for the new license.
6. Click **Close** to close the Product License dialog box and return to the General tab.

Important

Because Graph Lakehouse shards data across slices, and the number of slices is determined by the number of available CPU allowed by the license, if your new license allows for an increased number of CPU and/or nodes in the cluster, Graph Lakehouse must be re-initialized to clear the existing persisted data and take advantage of the increased resources.

7. Reinitialize Graph Lakehouse to configure the database with the updated license. Reinitializing the database requires running the following system manager (`azgctl`) command on the file system. [How do I access the Graph Lakehouse file system with Docker?](#) If you are using a cluster, run the command on the leader node:

```
./<install_path>/bin/azgctl -restart -init
```

Graph Lakehouse is now configured according to the specifications of the new license key.

Install or Upgrade a Legacy License from the Command Line

1. Run the following command to display your current deployment's license information:

```
<install_path>/bin/azgctl -getlicenseid
```

Note

If Graph Lakehouse is stopped, you can run the following command to return the server ID:

```
<install_path>/bin/azg_get_server_id
```

The command returns a number of attributes associated with the current license, including a **property_license.id** value. The **property_license.id** value will be used as the Server ID when upgrading your license. For example:

```
property_license.id: 2191-680E-178F-3D28-1535-D0F1
```

2. Copy the **property_license.id** value. This is your server ID. Contact [Altair Support](#) to request a new license and provide the required license ID and the system properties (Server ID, CPU Cores, Max RAM, and Max Nodes).
3. After receiving the new license key from Altair License Admin, apply the key to your deployment using one of the following options:
 - If Graph Lakehouse is stopped: Open in a text editor the **license.pem** file located in the `<install_path>/config` directory on the leader node. In the **license.pem** file, replace the existing contents with the new license key that you copied in the previous step. Then save and close the file.
 - If Graph Lakehouse is running, you can use the system manager to import the license key. On the leader node, run the following command to import the key and update the **license.pem** file:

```
<install_path>/azg/bin/azgctl -license <license_key_text>
```

For example:

```
/opt/anzograph/bin/azgctl -license  
H4sIAAAAAAAAAAG2RT0vDQBDf7/kUC54ELfsnu00LAaut...
```

Important

Because Graph Lakehouse shards data across slices, and the number of slices is determined by the number of available CPU allowed by the license, if your upgraded license allows for an increased number of CPU and/or nodes in the cluster, Graph Lakehouse must be re-initialized to clear the existing persisted data and take advantage of the increased resources.

4. Reinitialize Graph Lakehouse to configure the database with the updated license.

Reinitializing the database requires running the following system manager (`azgctl`) command on the file system. If you are using a cluster, run the command on the leader node:

```
./<install_path>/bin/azgctl -restart -init
```

Graph Lakehouse is now configured to validate against the new license key. For information about loading data, see [Load & Manage Data](#).

Learn SPARQL

The topics in this section introduce you to SPARQL and provide some best practices and tips. For additional basic information about SPARQL, the semantic web, or RDF, see the Altair [Semantic University](#).

In this section:

SPARQL Query Basics	118
SPARQL Best Practices	134
SPARQL Tips and Tricks	137

SPARQL Query Basics

SPARQL statements are constructed using the following basic query types, clauses, and other solution modifiers.

Standard Query Types

- **SELECT**: Run SELECT queries when you want to find and return all of the data that matches certain patterns.
- **CONSTRUCT**: Run CONSTRUCT queries when you want to create or transform data based on the existing data.
- **ASK**: Run ASK queries when you want to know whether a certain pattern exists in the data. ASK queries return only "true" or "false" to indicate whether a solution exists.
- **DESCRIBE**: Run DESCRIBE queries when you want to view the RDF graph that describes a particular resource.

Query Clauses

All queries may also include one or more of the following clauses:

- **PREFIX Clause**: The optional PREFIX clause declares the abbreviations that you want to use to reference URIs in the query.
- **FROM Clause**: The optional FROM clause defines the data sets or graphs to query. By default, if you do not specify the FROM clause, the scope of a query is limited to the default graph.
- **WHERE Clause**: The WHERE clause specifies the query pattern for data to match in the graphs or data sets specified in the query.

Solution Modifiers

The following optional query modifiers enable you to restrict, group, sort, or further refine query results:

- **ORDER BY:** This modifier sorts the result set in a particular order. It sorts query solution results based on the value of one or more variables.
- **OFFSET:** Using this modifier in conjunction with LIMIT and ORDER BY returns a slice of a sorted solution set, for example, for paging.
- **LIMIT:** This modifier puts an upper bound on the number of results returned by a query.
- **GROUP BY:** This modifier is used with aggregate functions and specifies the key variables to use to partition the solutions into groups. For information about the Graph Lakehouse GROUP BY clause extensions, see [Advanced Grouping Sets](#).
- **HAVING:** This modifier is used with aggregate functions and further filters the results after applying the aggregates.

SELECT

Like SQL, SPARQL provides a SELECT query form for selecting or finding data. This section describes the SELECT form.

Syntax

```
[ PREFIX Clause ]
SELECT [ DISTINCT | REDUCED ] result_expressions_and_variables
[ FROM Clause ]
WHERE Clause
[ Solution Modifiers ]
```

The optional DISTINCT solution sequence modifier eliminates duplicate results. The REDUCED modifier permits duplicate solutions to be removed but does not guarantee that they are eliminated from the results.

Examples

The following simple SELECT statement queries the sample Tickit data set to return all of the predicates and objects for event100:

```
SELECT ?predicate ?object
FROM <http://anzograph.com/tickit>
WHERE {
```

```

    <http://anzograph.com/ticket/event100> ?predicate ?object
}
ORDER BY ?predicate

```

predicate	object
-----+-----	
-	
http://anzograph.com/ticket/catid	http://anzograph.com/ticket/category8
http://anzograph.com/ticket/dateid	http://anzograph.com/ticket/date2129
http://anzograph.com/ticket/eventname	Siegfried
http://anzograph.com/ticket/starttime	2008-10-30T15:00:00Z
http://anzograph.com/ticket/venueid	http://anzograph.com/ticket/venue300
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://anzograph.com/ticket/event
6 rows	

This simple **SELECT** statement uses the **DISTINCT** modifier to list each distinct event name in the Tickit data set:

```

SELECT DISTINCT ?name
FROM <http://anzograph.com/ticket>
WHERE {
    ?event <http://anzograph.com/ticket/eventname> ?name.
}

```

```

name
-----
Adriana Lecouvreur
Eugene Onegin
La Cenerentola (Cinderella)
A Chorus Line
Hairspray
Le Reve
Hedda Gabler
Endgame
Othello
Miss Julie
...
576 rows

```

This example queries the sample Tickit data set to select the top 10 listings where the price per ticket is greater than \$500.00:


```

SELECT ?listing ?priceperticket
FROM <http://anzograph.com/tickit>
WHERE {
    ?listing <http://anzograph.com/tickit/listtime> ?listtime .
    ?listing <http://anzograph.com/tickit/priceperticket> ?priceperticket .
    FILTER(?priceperticket > 500.00)
}
ORDER BY desc(?priceperticket)
LIMIT 10

```

listing	priceperticket
-----	-----
http://anzograph.com/tickit/listing227412	2500
http://anzograph.com/tickit/listing222840	2500
http://anzograph.com/tickit/listing225362	2500
http://anzograph.com/tickit/listing214995	2500
http://anzograph.com/tickit/listing209658	2500
http://anzograph.com/tickit/listing211035	2500
http://anzograph.com/tickit/listing213613	2500
http://anzograph.com/tickit/listing215156	2500
http://anzograph.com/tickit/listing210898	2500
http://anzograph.com/tickit/listing224389	2500
10 rows	

CONSTRUCT

Use the CONSTRUCT query form to create new data from your existing data.

Syntax

```

[ PREFIX Clause ]
CONSTRUCT { graph_or_triple_template }
WHERE Clause
[ Solution Modifiers ]

```

CONSTRUCT queries take each solution and substitute it for the variables in the graph or triple template. Graph Lakehouse combines the triples into a single graph in N-Triple format. If you specify a pattern that produces a triple that contains an unbound variable or an illegal RDF construct such as a literal value in the subject or predicate position, these problematic triples are excluded from the output graph.

Note

CONSTRUCT query results are always returned in RDF format. They cannot be returned in any other format and any options normally used to specify a result format, such as the `Accept`, `Content-Type`, or `format` parameters, are ignored.

Example

The following example query specifies a triple template that constructs a new age predicate and approximate age value for the person triples in the sample Tickit data set. The resulting age values are approximations because the calculation excludes days and months.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX tickit: <http://anzograph.com/tickit/>
CONSTRUCT { ?person tickit:age ?age . }
WHERE { GRAPH <http://anzograph.com/tickit> {
    SELECT ?person ((YEAR(?date))-(YEAR(xsd:dateTime(?birthdate)))) AS ?age
    WHERE {
        ?person tickit:birthday ?birthdate .
        BIND(xsd:dateTime(NOW()) AS ?date)
    }
}
}
ORDER BY ?person
LIMIT 10
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
<http://anzograph.com/tickit/person1> <age>
"55"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/tickit/person10> <age>
"75"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/tickit/person100> <age>
"32"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/tickit/person1000> <age>
"38"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/tickit/person10000> <age>
"77"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/tickit/person10001> <age>
"27"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/tickit/person10002> <age>
"75"^^<http://www.w3.org/2001/XMLSchema#int> .
```

```
<http://anzograph.com/ticket/person10003> <age>
"69"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/ticket/person10004> <age>
"50"^^<http://www.w3.org/2001/XMLSchema#int> .
<http://anzograph.com/ticket/person10005> <age>
"72"^^<http://www.w3.org/2001/XMLSchema#int> .
```

ASK

Use the ASK query form to determine whether a particular triple pattern exists in the specified data set. ASK returns true or false, depending on whether the solution or match exists.

Syntax

```
[ PREFIX Clause ]
ASK
[ FROM Clause ]
{ triple_template }
```

Examples

The following example ASK statement queries the sample Ticket data set to ask whether data exists for the event named Wicked:

```
ASK FROM <http://anzograph.com/ticket> { ?s <http://anzograph.com/ticket/eventname>
"Wicked" . }
```

```
true
```

DESCRIBE

Use the DESCRIBE query form to return all triples that are associated with a specified resource, not just the triples that are bound to any variables that you specify. Running a DESCRIBE query can be helpful when learning about the data that exists without having to know the structure of the data.

Like CONSTRUCT, DESCRIBE returns results in RDF format.

Syntax

```
[ PREFIX Clause ]
DESCRIBE <resource>
```

```
[ FROM Clause ]  
[ WHERE Clause ]
```

Examples

The following simple DESCRIBE example queries the sample Tickit dataset to describe all of the resources that are associated with person2:

```
DESCRIBE <http://anzograph.com/tickit/person2>  
FROM <http://anzograph.com/tickit>  
  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.  
<http://anzograph.com/tickit/listing160217> <http://anzograph.com/tickit/sellerid>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/listing30988> <http://anzograph.com/tickit/sellerid>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/listing43813> <http://anzograph.com/tickit/sellerid>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/listing52091> <http://anzograph.com/tickit/sellerid>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/listing54017> <http://anzograph.com/tickit/sellerid>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/listing86046> <http://anzograph.com/tickit/sellerid>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person1099> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person1268> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person12847> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person12996> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person13112> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person15212> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person15323> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person15929> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person16636> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2> .  
<http://anzograph.com/tickit/person17109> <http://anzograph.com/tickit/friend>
```

```

<http://anzograph.com/ticket/person2> .
<http://anzograph.com/ticket/person2> <http://anzograph.com/ticket/birthday> "1979-09-
21"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://anzograph.com/ticket/person2> <http://anzograph.com/ticket/card>
"6614771532725111"^^<http://www.w3.org/2001/XMLSchema#long> .
<http://anzograph.com/ticket/person2> <http://anzograph.com/ticket/city> "Murfreesboro"
.
<http://anzograph.com/ticket/person2> <http://anzograph.com/ticket/dislike> "broadway"
.
<http://anzograph.com/ticket/person2> <http://anzograph.com/ticket/email>
"Suspendisse.tristique@nonnisiAenean.edu" .
<http://anzograph.com/ticket/person2> <http://anzograph.com/ticket/firstname>
"Vladimir" .
...

```

PREFIX Clause

The PREFIX clause declares any abbreviations for URIs that you want to reference in a query. You can declare prefixes to simplify query text if your data includes long URI names. If you do not declare prefixes, you must include the full URI names in the query.

Syntax

```

PREFIX uri_abbreviation: <uri_to_substitute_for_abbreviation>
[ PREFIX uri_abbreviation: <uri_to_substitute_for_abbreviation> ]
[ ... ]

```

URIs can be well-formed and absolute, such as `<http://www.w3.org/1999/02/22-rdf-syntax-ns>` or relative, such as `<http://cambridgesemantics.com/property#>` or `<ticket/sales/>`.

Note

Graph Lakehouse places no special restrictions on URIs, only that you place the URI text between `<` `>` characters, for example, `<sales>`.

Examples

The following example query declares a prefix and then uses the declared URI abbreviation in place of full the URI names in the query.

```
PREFIX t: <http://anzograph.com/tickit/>
SELECT ?eventname ?location
FROM <http://anzograph.com/tickit>
WHERE {
    ?eventid t:venueid ?venueid .
    ?venueid t:venueid ?location .
    ?eventid t:dateid ?dateid .
    ?eventid t:eventname ?eventname .
}
ORDER BY ?location
```

FROM Clause

The optional FROM clause defines the data sets or graphs to query. You can include any number of FROM or FROM NAMED statements.

Important

By default, if a query omits FROM clauses, the scope of the query is limited to the default graph (DEFAULTSET). Triples in named graphs will not be included in the scope of the query. The default behavior is controlled by the [sparql_spec_default_graph](#) configuration setting. To configure Graph Lakehouse to conform to the SPARQL specification and include the default graph and all named graphs in the scope of a query that omits the FROM clause, change the value of `sparql_spec_default_graph` to **true**. For more information, see [Change the Default FROM Clause Behavior](#).

Syntax

```
FROM [ NAMED | EXTERNAL ] <graph_uri>
```

Option	Description
FROM	Use <code>FROM <graph_uri></code> when you want to query for the same triple pattern match against one or more graphs. For example, the statement <code>FROM <graphA></code> means that Graph Lakehouse takes all of the triples that belong to graphA and adds them to the default graph (called DEFAULTSET). For the following FROM

Option	Description
	clause: <pre data-bbox="492 279 703 342">FROM <graphA> FROM <graphB></pre> Graph Lakehouse takes all of the triples associated with graphA and graphB and adds them to the default graph. The triple patterns in the query's WHERE clause are matched against all of the graphA and graphB triples in the default graph. For example, the following query looks for the ?s ?p ?o triple pattern match in graphA and graphB: <pre data-bbox="492 695 703 915">SELECT * FROM <graphA> FROM <graphB> WHERE { ?s ?p ?o . }</pre>
FROM NAMED	Use FROM NAMED <graph_uri> when you want to query multiple graphs and specify which patterns to match against which graph by naming the graphs in the WHERE clause. For example, the statement FROM NAMED <graphA> means that the graphA triples are not added to the default graph. Patterns in the WHERE clause are only matched against graphA when a graph pattern is specified (such as GRAPH <graphA> { triple_patterns } or GRAPH ?g { triple_patterns}). Triple patterns without a GRAPH clause are still matched against the default graph. For example, the following query only finds matches in graphA because graphB is not named in the WHERE clause: <pre data-bbox="492 1654 800 1801">SELECT * FROM <graphA> FROM NAMED <graphB> WHERE {</pre>

Option	Description
	<pre>?s ?p ?o . }</pre> To match the <code>?s ?p ?o</code> triple pattern against graphA and graphB, the query needs to name graphB using a GRAPH clause. For example: <pre>SELECT * FROM <graphA> FROM NAMED <graphB> WHERE { ?s ?p ?o . { GRAPH <graphB> { ?s ?p ?o . } } }</pre> When using FROM NAMED, triple patterns that are outside of a GRAPH clause are matched against the default graph. Triple patterns that are included in a GRAPH clause are matched against that named graph. You can also use the GRAPH <i>?variable</i> construct (such as GRAPH ?g) to allow a pattern to match against one of the named graphs in the query. The URI of the matching graph is bound to the variable.
FROM EXTERNAL	Use FROM EXTERNAL to run queries against files on disk. See Analyzing Data Characteristics in Load Files for more information.

Example

The FROM clause in the following example query includes two FROM NAMED statements. The WHERE clause includes two graph patterns to find the events in the Tickit graph and the movies in the Movies graph.

```
SELECT (COUNT(?eventid) as ?tickit_events) (COUNT(?movieid) as ?num_movies)
FROM NAMED <tickit>
FROM NAMED <movies>
WHERE {
  { GRAPH <tickit> { ?event <eventid> ?events . } }
}
```



```
{ GRAPH <movies> { ?film <movieid> ?movieid . } }
```

WHERE Clause

The WHERE clause defines the query patterns to look for in the specified graphs or data sets.

WHERE clauses can include graph and triple templates, subqueries, and the following clauses:

- **BIND**: Assigns the results of an expression to a new variable.
- **FILTER**: Applies boolean conditions or tests to constrain results and filter out values that do not meet the specified conditions.
- **MINUS, OPTIONAL, UNION**: MINUS subtracts matches from the query result based on the evaluation of the pattern that you specify. OPTIONAL tries to match a graph pattern but does not fail to return results if the optional match fails. UNION Includes results from either of two graph patterns with solutions to both sides of the union are included in the results.
- **SERVICE**: Queries remote data at a SPARQL endpoint.
- **VALUES**: Enables users to include data in a graph pattern to filter results on more specific requirements. The data is joined with the results of the query evaluation.

BIND

Use the following syntax when incorporating BIND in the WHERE clause:

```
BIND(expression AS ?variable)
```

Where *expression* evaluates to the values that you want to bind to the *variable*.

FILTER

Use the following syntax when incorporating filters in WHERE clauses:

```
FILTER(expression [ logical_operator expression ] [...])
```

Where *expression* is the condition to test for. You can also use the logical operators **&&** (AND), **||** (OR), or **!** (NOT) to combine filter expressions. For information about using logical operators and functions in filters, see [Logical Functions](#).

MINUS, OPTIONAL, UNION

Use the following syntax when incorporating MINUS, OPTIONAL, or UNION statements in WHERE clauses:

```
KEYWORD { triple_patterns }
```

SERVICE

Use the following syntax when incorporating SERVICE statements in WHERE clauses. SERVICE statements have the same structure as named graph statements:

```
SERVICE{ <endpoint_URL> { triple_patterns } }
```

Where *endpoint_URL* is the URL for accessing the remote SPARQL endpoint, and *triple_patterns* define the patterns to look for in the remote data.

For example, the following query uses a SERVICE call to retrieve data from the DBpedia SPARQL endpoint.

```
SELECT *
WHERE {
  SERVICE <http://dbpedia.org/sparql> {
    <http://dbpedia.org/resource/Keanu_Reeves> <http://dbpedia.org/ontology/abstract>
    ?o.
    FILTER LANGMATCHES (LANG(?o), "en")
  }
}
```

The query returns an abstract about Keanu Reeves:

```
o
-----
Keanu Charles Reeves (/keɪˈɑːnuː/ kay-AH-noo; born September 2, 1964) is a Canadian actor...
```

VALUES

Use the following syntax when incorporating VALUES statements in WHERE clauses:

```
VALUES ?variable { value_for_variable [ another_possible_value ] [...] }
```

Where *?variable* is the node that you want to find values for, and *value_for_variable* is the value to look for.

To further constrain the results by specifying multiple variables and possible values, use the following syntax:

```
VALUES ( ?variable1 ?variable2 [...] ) {  
  ( [UNDEF] | value_for_variable1 [UNDEF] | value_for_variable2 [...])  
  ( [UNDEF] | another_value_for_variable1 [UNDEF] | another_value_for_variable2 [...])  
  ( [ ... ] )  
}
```

The UNDEF keyword acts as a wildcard instead of a specific value. For example, the following VALUES clause states: "Include matches when ?b =n regardless of the value for ?a, and include matches when ?a=x regardless of the value for ?b."

```
VALUES ( ?a ?b ) {  
  ( UNDEF "n" )  
  ( "x" UNDEF )  
}
```

WHERE Clause Examples

The WHERE clause in the example query below uses triple patterns and a filter to query the sample Tickit data set and return a list of all of the musicals that occur on a holiday.

```
PREFIX tickit: <http://anzograph.com/tickit/>  
SELECT DISTINCT ?event ?category  
FROM <http://anzograph.com/tickit>  
WHERE {  
  ?s tickit:eventid ?eventid .  
  ?eventid tickit:dateid ?dateid .  
  ?dateid tickit:caldate ?date .  
  ?eventid tickit:eventname ?event .  
  ?eventid tickit:catid ?cat .  
  ?cat tickit:catname ?category .  
  ?dateid tickit:holiday ?holiday .  
  FILTER(?holiday=true && ?category="Musicals")  
}  
ORDER BY ?event
```

event	category
A Catered Affair	Musicals
Chicago	Musicals
Curtains	Musicals
Dirty Dancing	Musicals
Folies Bergere	Musicals
Grease	Musicals
High Society	Musicals
Legally Blonde	Musicals
Mamma Mia!	Musicals
Oliver!	Musicals
Phantom of the Opera	Musicals
Spamalot	Musicals
The King and I	Musicals
The Phantom of the Opera	Musicals
West Side Story	Musicals

15 rows

The WHERE clause in the query below includes a subquery that joins sales and event data to return ticket, commission, and price paid information for each event. The top-level result clause uses the subquery results to subtract the commission paid from the total price paid to calculate the profit for each event.

```

PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event ?tickets ((?total_paid - ?commission_paid) as ?profit)
FROM <http://anzograph.com/tickit>
WHERE {
  SELECT ?event (sum(?qty) as ?tickets) (sum(?comm) as ?commission_paid) (sum(?price)
as ?total_paid)
  WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?sales tickit:commission ?comm .
    ?sales tickit:pricepaid ?price .
  }
  GROUP BY ?event
}
ORDER BY desc(?profit)
LIMIT 10

```

event	tickets	profit
Mamma Mia!	3658	965136
Spring Awakening	3025	826927
The Country Girl	2871	773979
Macbeth	2733	733193
Jersey Boys	2781	690095
Legally Blonde	2272	683896
Chicago	2535	672344
Spamalot	2199	607161
Hedda Gabler	1891	561865
Thurgood	1894	543895
10 rows		

SPARQL Best Practices

When compared with SQL, SPARQL's syntax and grammar is less enforceable. In a graph database, since the data defines the schema, the data cannot be evaluated against the schema. In addition, since RDF graphs typically contain semi-structured data, the database can include data that is incomplete or unknown. This topic provides tips to help you avoid getting unexpected results when running SPARQL queries.

- [Look for Typos](#)
- [Make Some Triple Patterns Optional](#)
- [Avoid Unexpected Results When Constructing Data](#)

Look for Typos

Mistyping a predicate, for example, does not produce an error such as "predicate does not exist." Instead the query might not produce any results.

Example

The following query counts the distinct number of likes in the sample Tickit data. As shown in the WHERE clause, the predicate in the tickit graph is "<like>". The results show that there are 10 distinct likes, or 10 distinct objects for the <like> predicate:

```
SELECT (count(?o) as ?numberOfLikes)
FROM <http://anzograph.com/tickit>
WHERE { {
    SELECT DISTINCT ?o
    WHERE { ?s <http://anzograph.com/tickit/like> ?o }
}
```

```
numberOfLikes
-----
10
1 rows
```

Misspelling "like" as "likes" does not produce an error, but the query returns no results:

```

SELECT (count(?o) as ?numberOfLikes)
FROM <http://anzograph.com/ticket>
WHERE { {
    SELECT DISTINCT ?o
    WHERE { ?s <http://anzograph.com/ticket/likes> ?o }
  }
}

```

```

numberOfLikes
-----
0
1 rows

```

Make Some Triple Patterns Optional

Some queries might need to account for missing or incomplete data. To ensure that triples are not excluded from the results because they follow some of the query's triple patterns but not all, you can use the `OPTIONAL` keyword to make certain triple patterns optional.

For example, the sample Tickit dataset includes person graphs. These graphs contain triples with a person subject and predicates such as first name, last name, birthday, credit card number, like, and dislike. Some person graphs are missing like or dislike predicates, so querying for person data using like or dislike in the pattern may produce unexpected results.

Example

The following example queries the Tickit dataset to find the first and last name and likes and dislikes for all of the people who have bought tickets:

```

PREFIX tickit: <http://anzograph.com/ticket/>
SELECT ?fname ?lname ?like ?dislike
FROM <http://anzograph.com/ticket>
WHERE {
    ?sale tickit:buyerid ?person .
    ?person tickit:firstname ?fname .
    ?person tickit:lastname ?lname .
    ?person tickit:like ?like .
    ?person tickit:dislike ?dislike .
}
GROUP BY ?fname ?lname ?like ?dislike

```

The patterns in the WHERE clause ask for person data where the triples include firstname, lastname, like, and dislike. Any person triples that are missing any of the patterns are excluded from the results. This query returns **188536** rows.

Using OPTIONAL clauses in the query changes the criteria so that all of the first and last names are returned and like or dislike data is returned if it exists. This query makes like and dislike optional:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?fname ?lname ?like ?dislike
FROM <http://anzograph.com/tickit>
WHERE {
    ?sale tickit:buyerid ?person .
    ?person tickit:firstname ?fname .
    ?person tickit:lastname ?lname .
    OPTIONAL { ?person tickit:like ?like }
    OPTIONAL { ?person tickit:dislike ?dislike }
}
GROUP BY ?fname ?lname ?like ?dislike
```

This query returns **202862** rows because it includes person triples with first and last names and does not exclude triples that are missing like or dislike predicates.

Avoid Unexpected Results When Constructing Data

CONSTRUCT queries return a single RDF graph specified by the template that you supply. The result takes each query solution and substitutes for the variables in the template and then combines the triples into a graph. If you specify a pattern that produces a triple that contains an unbound variable or an illegal RDF construct such as a literal value in the subject or predicate position, then you may get unexpected results because the problematic triples are excluded from the output graph.

SPARQL Tips and Tricks

The topics in this section describe SPARQL patterns that are frequently used for managing, understanding, and analyzing data. For example, this topic provides details about how to delete all data associated with an object and how to perform a cascaded delete. It also provides tips for understanding your data as a graph by analyzing social networks to find the most connected people and the size of their network out to one or two degrees.

In this section:

Managing Your Data	137
Exploring Your Data	140
Understanding Your Data as a Graph	144

Managing Your Data

This topic provides information about common questions to ask when managing your data.

- [How do I list all of the graphs in the database?](#)
- [How do I find all of the triples that reference a resource?](#)
- [How do I perform a cascaded delete to remove all triples associated with a resource?](#)
- [How do I delete a predicate and all of its values?](#)

How do I list all of the graphs in the database?

The following query returns a list of all of the named graphs in Graph Lakehouse:

```
SELECT DISTINCT ?graph
WHERE {
  GRAPH ?graph { ?s ?p ?o . }
}
```

```
graph
-----
http://anzograph.com/tickit
1 rows
```

How do I find all of the triples that reference a resource?

A common task is to find all of the triples in a graph that refer to a particular resource. That resource might be a subject in one triple and an object in another. For example, the `person2` resource in the sample tickit graph is referenced as the subject in some triples and the object in other triples. For example:

```
<http://anzograph.com/tickit/person2> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person12595>  
<http://anzograph.com/tickit/person12595> <http://anzograph.com/tickit/friend>  
<http://anzograph.com/tickit/person2>
```

To find all of the triples that reference `person2`, the following example query returns all of the triples where `person2` is the subject or object:

```
SELECT ?s ?p ?o  
FROM <http://anzograph.com/tickit>  
WHERE {  
  { BIND (<http://anzograph.com/tickit/person2> AS ?s) ?s ?p ?o . }  
  UNION  
  { BIND (<http://anzograph.com/tickit/person2> AS ?o) ?s ?p ?o . }  
}
```

s	p	o
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/friend	http://anzograph.com/tickit/person49923
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/friend	http://anzograph.com/tickit/person6671
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/friend	http://anzograph.com/tickit/person48422
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/friend	http://anzograph.com/tickit/person32005
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/friend	http://anzograph.com/tickit/person48156
...		
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/dislike	broadway
http://anzograph.com/tickit/person2	http://anzograph.com/tickit/state	WI

```

http://anzograph.com/ticket/person2      | http://anzograph.com/ticket/city
|                                          Murfreesboro
http://anzograph.com/ticket/person2      | http://anzograph.com/ticket/like
|                                          musicals
...
http://anzograph.com/ticket/person32064   | http://anzograph.com/ticket/friend
|      http://anzograph.com/ticket/person2
http://anzograph.com/ticket/person41654   | http://anzograph.com/ticket/friend
|      http://anzograph.com/ticket/person2
http://anzograph.com/ticket/person48892   | http://anzograph.com/ticket/friend
|      http://anzograph.com/ticket/person2
http://anzograph.com/ticket/person6048    | http://anzograph.com/ticket/friend
|      http://anzograph.com/ticket/person2
http://anzograph.com/ticket/person15323   | http://anzograph.com/ticket/friend
|      http://anzograph.com/ticket/person2
...
100 rows

```

How do I perform a cascaded delete to remove all triples associated with a resource?

The example above demonstrates how to find all of the triples that reference a resource. This example shows how to delete the resource and all of the triples that refer to it. The query below deletes person2 and all associated triples from the ticket graph:

```

DELETE { GRAPH <http://anzograph.com/ticket> { ?s ?p ?o . } }
WHERE {
  GRAPH <http://anzograph.com/ticket> { ?s ?p ?o .
    filter(?s=<http://anzograph.com/ticket/person2> ||
    ?o=<http://anzograph.com/ticket/person2>)
  }
}

```

To confirm that person2 and the related triples no longer exist, you can run the query from the first example:

```

SELECT ?s ?p ?o
FROM <http://anzograph.com/ticket>
WHERE {
  { BIND (<http://anzograph.com/ticket/person2> AS ?s) ?s ?p ?o . }
  UNION
  { BIND (<http://anzograph.com/ticket/person2> AS ?o) ?s ?p ?o . }
}

```

```
s | p | o
--+---+---
0 rows
```

How do I delete a predicate and all of its values?

You might need to remove a predicate and all of the associated objects from a graph. For example, the sample Tickit dataset includes person subjects with an ssn predicate whose objects are social security numbers for each person. The following query deletes the ssn predicate and the social security numbers from the tickit graph:

```
DELETE { GRAPH <http://anzograph.com/tickit> { ?person
<http://anzograph.com/tickit/ssn> ?ssn . } }
WHERE { GRAPH <http://anzograph.com/tickit> { ?person <http://anzograph.com/tickit/ssn>
?ssn . } }
```

Exploring Your Data

This topic provides information about common questions to ask when getting to know your data.

- [How do I find out which predicates \(keys\) a data set uses?](#)
- [How do I determine the frequency of a predicate's use?](#)
- [How do I find symmetric predicates?](#)

How do I find out which predicates (keys) a data set uses?

When you receive a new dataset, one of the first things to understand about the new dataset is what predicates are used. The following query lists the predicates used in the sample Tickit dataset.

```
SELECT DISTINCT ?predicates
FROM <http://anzograph.com/tickit>
WHERE {
    ?subject ?predicates ?object .
}
ORDER BY ?predicates
```

```
predicates
-----
http://anzograph.com/tickit/birthday
http://anzograph.com/tickit/buyerid
```

<http://anzograph.com/tickit/caldate>
<http://anzograph.com/tickit/card>
<http://anzograph.com/tickit/catdesc>
<http://anzograph.com/tickit/catgroup>
<http://anzograph.com/tickit/catid>
<http://anzograph.com/tickit/catname>
<http://anzograph.com/tickit/city>
<http://anzograph.com/tickit/commission>
<http://anzograph.com/tickit/dateid>
<http://anzograph.com/tickit/day>
<http://anzograph.com/tickit/dislike>
<http://anzograph.com/tickit/email>
<http://anzograph.com/tickit/eventid>
<http://anzograph.com/tickit/eventname>
<http://anzograph.com/tickit/firstname>
<http://anzograph.com/tickit/friend>
<http://anzograph.com/tickit/holiday>
<http://anzograph.com/tickit/lastname>
<http://anzograph.com/tickit/like>
<http://anzograph.com/tickit/listid>
<http://anzograph.com/tickit/listtime>
<http://anzograph.com/tickit/month>
<http://anzograph.com/tickit/numtickets>
<http://anzograph.com/tickit/phone>
<http://anzograph.com/tickit/pricepaid>
<http://anzograph.com/tickit/priceperticket>
<http://anzograph.com/tickit/qtr>
<http://anzograph.com/tickit/qtysold>
<http://anzograph.com/tickit/saletime>
<http://anzograph.com/tickit/sellerid>
<http://anzograph.com/tickit/ssn>
<http://anzograph.com/tickit/starttime>
<http://anzograph.com/tickit/state>
<http://anzograph.com/tickit/totalprice>
<http://anzograph.com/tickit/venuecity>
<http://anzograph.com/tickit/venuecitypop>
<http://anzograph.com/tickit/venueid>
<http://anzograph.com/tickit/venueName>
<http://anzograph.com/tickit/venueSeats>
<http://anzograph.com/tickit/venueSeatsPct>
<http://anzograph.com/tickit/venueState>
<http://anzograph.com/tickit/week>
<http://anzograph.com/tickit/year>

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

```
46 rows
```

How do I determine the frequency of a predicate's use?

Once you determine what predicates a new data set uses, you might want to see how frequently the predicates are used. Frequency counts can give you an indication of which predicates the data set uses together, which helps to identify objects in the graph. The following query lists the predicates in the sample Tickit data set ordered by the frequency in which they appear.

```
SELECT ?predicate (COUNT (?predicate) AS ?count)
FROM <http://anzograph.com/tickit>
WHERE {
    ?s ?predicate ?o .
}
GROUP BY ?predicate
ORDER BY DESC(?count)
```

predicate	count
http://anzograph.com/tickit/friend	1445832
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	424319
http://anzograph.com/tickit/dateid	373751
http://anzograph.com/tickit/sellerid	364953
http://anzograph.com/tickit/eventid	364953
http://anzograph.com/tickit/numtickets	192497
http://anzograph.com/tickit/listtime	192497
http://anzograph.com/tickit/priceperticket	192497
http://anzograph.com/tickit/totalprice	192497
http://anzograph.com/tickit/qtysold	172456
http://anzograph.com/tickit/buyerid	172456
http://anzograph.com/tickit/listid	172456
http://anzograph.com/tickit/commission	172456
http://anzograph.com/tickit/saletime	172456
http://anzograph.com/tickit/pricepaid	172456
http://anzograph.com/tickit/dislike	121038
http://anzograph.com/tickit/like	120911
http://anzograph.com/tickit/city	49990
http://anzograph.com/tickit/phone	49990
http://anzograph.com/tickit/state	49990
http://anzograph.com/tickit/email	49990
http://anzograph.com/tickit/lastname	49990

http://anzograph.com/ticket/ssn		49990
http://anzograph.com/ticket/firstname		49990
http://anzograph.com/ticket/card		49990
http://anzograph.com/ticket/birthday		49990
http://anzograph.com/ticket/starttime		8798
http://anzograph.com/ticket/venueid		8798
http://anzograph.com/ticket/catid		8798
http://anzograph.com/ticket/eventname		8798
http://anzograph.com/ticket/year		365
http://anzograph.com/ticket/week		365
http://anzograph.com/ticket/holiday		365
http://anzograph.com/ticket/caldate		365
http://anzograph.com/ticket/day		365
http://anzograph.com/ticket/qtr		365
http://anzograph.com/ticket/month		365
http://anzograph.com/ticket/venuecitypop		202
http://anzograph.com/ticket/venuestate		202
http://anzograph.com/ticket/venuecity		202
http://anzograph.com/ticket/venueename		202
http://anzograph.com/ticket/venueseatpct		58
http://anzograph.com/ticket/venueseat		58
http://anzograph.com/ticket/catdesc		11
http://anzograph.com/ticket/catgroup		11
http://anzograph.com/ticket/catname		11
46 rows		

How do I find symmetric predicates?

Another part of analyzing a new data set is understanding how the predicates are used. Predicates can be used in a symmetric way, for example, Ted knows Bob and Bob knows Ted. The following query finds the predicates in the sample Ticket data set that have symmetry and returns a count of the number of times that predicate is used symmetrically:

```
SELECT ?symmetricPredicate (COUNT (?symmetricPredicate) AS ?count)
FROM <http://anzograph.com/ticket>
WHERE {
    ?s ?symmetricPredicate ?o .
    ?o ?symmetricPredicate ?s .
}
GROUP BY ?symmetricPredicate
```

```
symmetricPredicate          | count
-----+-----
http://anzograph.com/tickit/friend | 1293290
1 rows
```

Understanding Your Data as a Graph

This topic provides information about common questions to ask when getting to know your data as a graph.

- [How do I find the most connected people?](#)
- [What is the size of a person's network?](#)
- [What is the density of the social network?](#)
- [Who has the most friends who know each other?](#)

How do I find the most connected people?

Graphs are often used to represent social behavior and social relationships. For example, the following triple represents that person12595 is a friend of person2:

```
<http://anzograph.com/tickit/person2> <http://anzograph.com/tickit/friend>
<http://anzograph.com/tickit/person12595>
```

A common task in social network analytics is to find out how connected are people in the social graph and who is the most connected. These questions are answered by computing the social network degree. People who have high network degree are the hubs of the social network. The following query demonstrates this by counting the number of friend relationships each person has in the sample Tickit data set. This query lists the top ten most connected people in the tickit graph:

```
SELECT ?person (COUNT(?friend) AS ?friendDegree)
FROM <http://anzograph.com/tickit>
WHERE {
    ?person <http://anzograph.com/tickit/friend> ?friend
}
GROUP BY ?person
ORDER BY DESC(?friendDegree)
LIMIT 10
```


person	friendDegree
http://anzograph.com/tickit/person34862	59
http://anzograph.com/tickit/person12763	57
http://anzograph.com/tickit/person7815	56
http://anzograph.com/tickit/person30165	56
http://anzograph.com/tickit/person16101	55
http://anzograph.com/tickit/person33352	55
http://anzograph.com/tickit/person29660	54
http://anzograph.com/tickit/person32511	54
http://anzograph.com/tickit/person17806	54
http://anzograph.com/tickit/person23501	53

10 rows

What is the size of a person's network?

The size of a person's network is usually computed out to two generations: the people a person knows and the people who those people know. The following query computes the size of person2's network in the sample Tickit data set. The COUNT expression subtracts 1 to remove person2 from the count:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT (COUNT(?friend)-1 AS ?networkSize)
FROM <http://anzograph.com/tickit>
WHERE {
  { SELECT DISTINCT ?friend
    WHERE {
      { tickit:person2 tickit:friend ?friend . }
    UNION
      { tickit:person2 tickit:friend ?friend1 .
        ?friend1 tickit:friend ?friend . }
    }
  }
}
```

```
networkSize
-----
969
1 rows
```

What is the density of the social network?

Network density measures whether a network is well-connected. When the network density equals 1, it indicates a clique: everyone is connected to everyone else. Compute network density by finding the ratio of the number of edges to the number of possible edges in a graph. You can use density for comparing different social networks or different regions within a social network.

The following example uses the friend relationships in the sample Tickit data set to determine the social graph density of the tickit graph. The number of edges is the number of triples that contain friend as a predicate. The number of possible friend relationships is $n(n-1)$ where n is the number of people in Tickit.

```
SELECT (?nrEdges/ (?nrNodes * (?nrNodes - 1.0)) AS ?graphDensity)
FROM <http://anzograph.com/tickit>
WHERE {
  { SELECT (COUNT (*) AS ?nrEdges) (COUNT (DISTINCT ?person) AS ?nrNodes)
    WHERE { ?person <http://anzograph.com/tickit/friend> ?anotherPerson . }
  }
}
```

```
graphDensity
-----
0.000578576
1 rows
```

Who has the most friends who know each other?

To find the most important people in a network, you can analyze how well-connected each person is. People are well-connected when their friends know each other. This is called a clique-to-triad. The following example identifies and counts triads to find the people in the sample Tickit data set who have the most friends who know each other. This query uses the friend relationship to rank the top ten people by the number of triads:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?person (COUNT (*) AS ?triads)
FROM <http://anzograph.com/tickit>
WHERE {
  { SELECT DISTINCT ?person ?anotherPerson ?yetAnotherPerson
    WHERE { ?person tickit:friend ?anotherPerson .
            ?person tickit:friend ?yetAnotherPerson .
          }
  }
}
```

```

        ?anotherPerson tickit:friend ?yetAnotherPerson .
    }
}
GROUP BY ?person
ORDER BY desc(?triads)
LIMIT 10

```

person	triads
http://anzograph.com/tickit/person44119	10
http://anzograph.com/tickit/person11134	9
http://anzograph.com/tickit/person43985	9
http://anzograph.com/tickit/person38821	8
http://anzograph.com/tickit/person24700	8
http://anzograph.com/tickit/person20029	8
http://anzograph.com/tickit/person13256	8
http://anzograph.com/tickit/person20305	8
http://anzograph.com/tickit/person29822	8
http://anzograph.com/tickit/person29184	8

10 rows

Sample Data and Tutorials

The topics in this section describe the sample data sets and provide tutorials to help familiarize you with Graph Lakehouse and the SPARQL and Cypher query languages.

In this section:

Working with SPARQL and the Tickit Data	149
Working with Cypher and the Movie Data	166

Working with SPARQL and the Tickit Data

This topic provides information about loading the Tickit demo data and running the example queries. You can also load the data and run queries using the interactive Graph Lakehouse Tutorial notebook as described in [Zeppelin Notebook Integration](#) or you can copy the queries in this topic and run them in the Graph Lakehouse Query Console, command line interface, or another interface.

- [Loading the Tickit Data](#)
- [Getting to Know the Tickit Data](#)
- [Running the Tickit Queries](#)

Loading the Tickit Data

The Tickit data set captures sales activity for a fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts. The example queries for this data set include detailed explanations of the query syntax. The goal is to help guide you through learning about the SPARQL language and concepts as well as demonstrate different analytic use cases.

The following query loads the files in the tickit.ttl.gz directory that is included with your deployment on the Graph Lakehouse file system.

```
LOAD <dir:/<install_path>/etc/tickit.ttl.gz>  
INTO GRAPH <http://anzograph.com/tickit>
```

Where <install_path> is the path to the Graph Lakehouse installation directory. For example, in a container deployment, the install path is /opt/anzograph:

```
LOAD <dir:/opt/anzograph/etc/tickit.ttl.gz>  
INTO GRAPH <http://anzograph.com/tickit>
```

The default installation path for a RHEL/Rocky deployment with the installer is /opt/altair/anzograph. If you specified a different install location, change the following query as needed.

```
LOAD <dir:/opt/altair/anzograph/etc/tickit.ttl.gz>  
INTO GRAPH <http://anzograph.com/tickit>
```

When the load completes, you can run this query to return the total number of triples in the data set:

```
SELECT (count(*) as ?number_of_triples)
FROM <http://anzograph.com/tickit>
WHERE { ?s ?p ?o }
```

For more information about loading data from RDF files, see [Load RDF Data from Files](#).

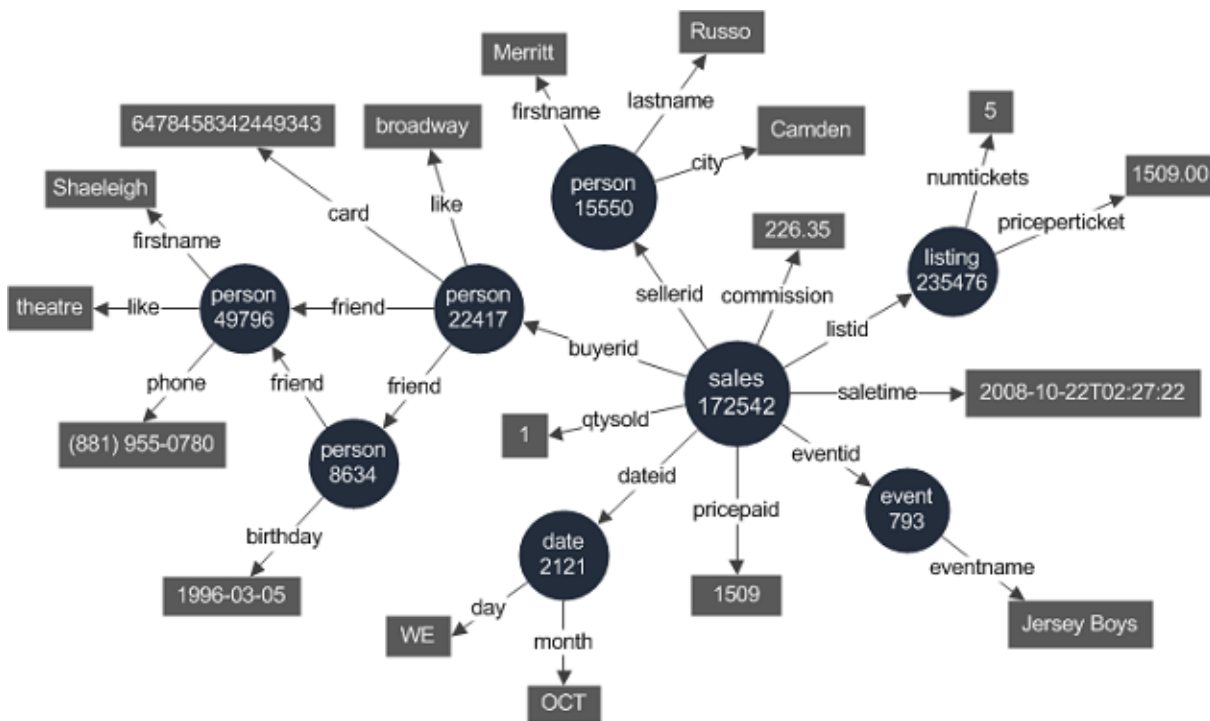
Getting to Know the Tickit Data

The Tickit data set captures sales activity for the fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts. The data consists of person, venue, category, date, event, listing, and sales information. By identifying ticket movement over time, success rates for sellers, the best-selling events and venues, and the most profitable times of the year, analysts can use this data to determine what incentives to offer, how to attract new people, and how to drive advertising and promotions.

To help familiarize you with the data set, the following diagram shows the model or ontology for the tickit graph. Circles represent subjects or classes of data and rectangles represent properties.



To help familiarize you with the triples in the tickit graph, the diagram below shows an instance of a subset of the triples in the graph.



Running the Tickit Queries

This section describes each of the Tickit queries and includes detailed explanations of the query syntax. Queries are grouped into categories such as "The Basics," which covers introductory SPARQL concepts, and "Marketing," "Social Graph," and "Fraud," which demonstrate analytic use cases.

- [The Basics](#)
- [Marketing](#)
- [Social Graph](#)
- [Fraud](#)
- [Finance](#)

The Basics

The basic queries provide guidance for users who are new to the SPARQL query language. These queries demonstrate introductory SPARQL concepts such as fetching and filtering, traversing graphs by joining data, using aggregate functions, and writing subqueries.

- [Fetching and Filtering](#)
- [Graph Traversal](#)
- [Aggregation](#)
- [Subqueries](#)

Fetching and Filtering

The query below fetches the predicates and objects for a specific person.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?predicate ?object
FROM <http://anzograph.com/tickit>
WHERE {
    tickit:person49158 ?predicate ?object .
}
ORDER BY ?predicate
```

- The SELECT list asks for all of the predicates and objects: `SELECT ?predicate ?object`.
- The WHERE clause narrows the results to return just the predicates and objects that are related to `person49158`: `WHERE { tickit:person49158 ?predicate ?object }`.
- The ORDER BY clause orders the results by predicate name. By default ORDER BY lists results in ascending order. Since the results for `?predicate` are string values, the results are in alphabetical order. To reverse the results to descending order, you can change `ORDER BY ?predicate` to `ORDER BY DESC(?predicate)`.

Graph Traversal

The query below reports where and when events take place by traversing the tickit graph and creating joins between different classes in the tickit model.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?eventname ?location ?date
FROM <http://anzograph.com/tickit>
WHERE {
    ?eventid tickit:venueid ?venueid .
    ?venueid tickit:venueName ?location .
}
```



```

?eventid tickit:dateid ?dateid .
?dateid tickit:caldate ?date .
?eventid tickit:eventname ?eventname .
}
ORDER BY ?date ?eventname ?location
LIMIT 100

```

Since the location information for events exists in the venue class and the date data for events exists in the date class, the event, venue, and date data are joined to report on the location and date for the events.

For example, the following two triples join event ID to location using the venue ID for each event:

```

?eventid tickit:venueid ?venueid .
?venueid tickit:venueid ?location .

```

And these triples join the event ID to the calendar date using the date ID:

```

?eventid tickit:dateid ?dateid .
?dateid tickit:caldate ?date .

```

Aggregation

The query below uses the COUNT SPARQL aggregate function to count the number of times each event occurs.

```

PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?event_name (count(*) as ?count)
FROM <http://anzograph.com/tickit>
WHERE {
    ?event tickit:eventname ?event_name
}
GROUP BY ?event_name
ORDER BY desc(?count) ?event_name
LIMIT 10

```

- The aggregate function in the SELECT list `((count(*) as ?count))` counts the event names `(?event_name)` produced by the WHERE clause.

- Since the query includes an aggregate function, a GROUP BY statement (GROUP BY ?event_name) is required to specify any variables in the SELECT list that are not aggregated.
- By using a LIMIT clause (LIMIT 10), the query reports only the 10 events that occurred most frequently.

Subqueries

The query below uses a subquery to find the total number of tickets sold, price paid, and commission paid for each event and then determine profit for each event.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event ?tickets ((?total_paid - ?commission_paid) as ?profit)
FROM <http://anzograph.com/tickit>
WHERE {
  SELECT ?event (sum(?qty) as ?tickets) (sum(?comm) as ?commission_paid) (sum(?price)
as ?total_paid)
  WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?sales tickit:commission ?comm .
    ?sales tickit:pricepaid ?price .
  }
  GROUP BY ?event
}
ORDER BY desc(?profit)
LIMIT 10
```

- The subquery is processed first. The WHERE clause in the subquery joins sales and event data to return ticket, commission, and price paid information for each event.
- The SELECT list for the subquery then calculates the sums of the total tickets, commission, and price paid for each event.
- Because the subquery uses aggregate functions, it requires a GROUP BY statement to group on the non-aggregate variable (?event) in the SELECT list.
- The top-level SELECT list for the query uses the subquery results to subtract the commission paid from the total price paid to calculate the profit for each event.

Marketing

The marketing queries provide analytics that answer questions a user might ask when making event marketing decisions.

- [Most Popular States](#)
- [Least Popular Events](#)
- [Inventory Aging](#)

Most Popular States

The query below reports on the most popular state to host events based on the number of venues per state.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?state (count(?venue) as ?total)
FROM <http://anzograph.com/tickit>
WHERE {
    ?venue tickit:venuestate ?state .
}
GROUP BY ?state
ORDER BY desc(?total) ?state
```

- The aggregate function in the SELECT list (`(count(?venue) as ?count)`) counts the venues (`?venue`) produced by the WHERE clause.
- Since the query includes an aggregate function, a GROUP BY statement (`GROUP BY ?state`) is required to group the non-aggregate variable in the SELECT list (`?state`).
- By ordering the results by total venues in decending order (`ORDER BY desc(?total) ?state`), the most popular state becomes the first state in the results.

Least Popular Events

The query below determines the most unpopular events by returning the 10 events with the least number of ticket sales. It also returns the event category.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?event ?category (sum(?qty) as ?total_tickets)
FROM <http://anzograph.com/tickit>
```

```

WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?eventid tickit:catid ?catid .
    ?catid tickit:catname ?category .
}
GROUP BY ?event ?category
ORDER BY ?total_tickets
LIMIT 10

```

Like the "Most Popular States" query, this examples uses an aggregate function (`(sum(?qty) as ?total_tickets)`) to calculate the total tickets for each event.

In the WHERE clause, the following triples join the sales data and event name using the event ID:

```

?sales tickit:eventid ?eventid .
?eventid tickit:eventname ?event .

```

And these triples join the event with the category name on category ID:

```

?eventid tickit:catid ?catid .
?catid tickit:catname ?category .

```

Since the query uses the SUM aggregate function, the query includes a GROUP BY clause to group on the non-aggregate variables, ?event and ?category.

Inventory Aging

The query below reports on the 20 events for which tickets took the longest to sell.

```

PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?location ?kind ?name ?list_date (((?selldate - ?list_date)) as ?sale_age)
FROM <http://anzograph.com/tickit>
WHERE {
    ?sale tickit:saletime ?selldate .
    ?sale tickit:eventid ?event .
    ?listing tickit:eventid ?event .
    ?listing tickit:listtime ?list_date .
    ?event tickit:eventname ?name .
    ?event tickit:venueid ?venue .
    ?event tickit:catid ?cat .
    ?cat tickit:catname ?kind .
}

```

```

    ?venue tickit:venueName ?location .
}
ORDER BY desc(?sellDate) desc(?list_date) ?location ?kind ?name
LIMIT 20

```

- In the WHERE clause, the query traverses the sales, listing, event, category, and venue classes.
- In the SELECT list, the query calculates the duration of the sale time by subtracting the listing date from the sale date: `((?sellDate - ?list_date) as ?sale_age)`.

Social Graph

The social graph queries focus on finding connections between the people (buyers and sellers) in the tickit graph.

- [Event Partners](#)
- [Potential Event Partners](#)

Event Partners

The query below finds 20 pairs of friends who went to the same event most often.

```

PREFIX tickit: <http://anzograph.com/tickit>
SELECT (count(*) as ?gone_together) ?name1 ?name2
FROM <http://anzograph.com/tickit>
WHERE {
    ?sale1 tickit:buyerid ?person1 .
    ?sale1 tickit:eventid ?event .
    ?person1 tickit:friend ?person2 .
    ?person1 tickit:firstname ?first1 .
    ?person1 tickit:lastname ?last1 .
    BIND(CONCAT(?first1, " ", ?last1) AS ?name1)
    ?sale2 tickit:buyerid ?person2 .
    ?sale2 tickit:eventid ?event .
    ?person2 tickit:firstname ?first2 .
    ?person2 tickit:lastname ?last2 .
    BIND(CONCAT(?first2, " ", ?last2) AS ?name2)
    FILTER ( ?name2 > ?name1 )
}
GROUP BY ?name1 ?name2

```

```
ORDER BY desc(?gone_together) ?name1 ?name2
LIMIT 20
```

- In the WHERE clause, the query traverses the sales, event, and person classes. The following triples find the events for which person1 bought tickets:

```
?sale1 tickit:buyerid ?person1 .
?sale1 tickit:eventid ?event .
```

- The ?person1 <friend> ?person2 triple narrows the results to people who are friends of person2.
- To simplify the query's resulting columns, the CONCAT function is used to concatenate the first and last name of person1: `BIND(CONCAT(?first1, " ", ?last1) AS ?name1)`. It also adds a space between the first and last name. And the BIND function binds the concatenated name to the variable name1.
- Similar to the first group of triples for person1, the triples for person2 find the events for which person2 bought tickets.
- The FILTER (`FILTER ( ?name2 > ?name1 )`) is also used to simplify or clean the results to do an ASCII comparison of name1 and name2 to omit duplicate pairs such the following examples:

```
"Joe Smith" <http://anzograph.com/tickit/friend> "Bob Jones"
"Bob Jones" <http://anzograph.com/tickit/friend> "Joe Smith"
```

Since the filter states that name2 is greater than name1, the results will list only one version of the triple, the one where name2 comes later in the alphabet than name1.

- Finally, the SELECT list uses the COUNT aggregate function to count the number of times person1 and person2 bought tickets to the same event. Use of the aggregate function requires the GROUP BY clause to group on name1 and name2.

Potential Event Partners

The query below reports on people who might attend musicals together. This query uses the same concepts as the previous query, "Event Partners." The query also traverses the sales, event, and person classes, but finds pairs of friends who know someone who bought tickets to one or more

musicals.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT (count(*) as ?know_buyer) ?name1 ?name2
FROM <http://anzograph.com/tickit>
WHERE {
    ?sale1 tickit:buyerid ?buyer .
    ?sale1 tickit:eventid ?event .
    ?buyer tickit:friend ?person2 .
    ?buyer tickit:friend ?person1 .
    ?person1 tickit:firstname ?first1 .
    ?person1 tickit:lastname ?last1 .
    BIND(CONCAT(?first1, " ", ?last1) AS ?name1)
    ?person2 tickit:firstname ?first2 .
    ?person2 tickit:lastname ?last2 .
    BIND(CONCAT(?first2, " ", ?last2) AS ?name2)
    ?event tickit:catid ?cat .
    ?cat tickit:catname "Musicals" .
    FILTER ( ?name2 > ?name1 )
}
GROUP BY ?name1 ?name2
ORDER BY desc(?know_buyer) ?name1 ?name2
LIMIT 20
```

- In the WHERE clause, the first 8 triples find the person (?buyer) who bought tickets to events and the names of two people who are friends of the buyer.
- The last 2 triples narrow the results to list only the events in the category "Musicals":

```
?event tickit:catid ?cat .
?cat tickit:catname "Musicals" .
```

- Like the Event Partners query, the FILTER eliminates duplicate but reversed triples from the results.
- In the SELECT list, the COUNT function counts the number of times person1 and person2 knew the person who bought tickets to an event.

Fraud

The fraud queries focus on finding problematic sales or people in the tickit graph, such as identity thieves and ticket scalpers.

- [Possible Ticket Scalpers](#)
- [People with the Same SSN](#)
- [Tickets Sold by Possible Identity Thieves](#)
- [Tickets Sold by Friends of Possible Identity Thieves](#)

Possible Ticket Scalpers

The query below identifies possible ticket scalpers by calculating the average price per ticket for events and then finding cases where tickets are listed for a higher price.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?sellername ?avg_price ?priceperticket ?eventname ?listtime
FROM <http://anzograph.com/tickit>
WHERE {
  { SELECT ?eventname (avg(?priceperticket) as ?avg_price)
    WHERE {
      ?listing tickit:eventid ?eventid .
      ?eventid tickit:eventname ?eventname .
      ?listing tickit:priceperticket ?priceperticket .
    }
    GROUP BY ?eventname
  }
  ?listing tickit:listtime ?listtime .
  ?listing tickit:priceperticket ?priceperticket .
  ?listing tickit:sellerid ?seller .
  ?seller tickit:firstname ?firstname .
  ?seller tickit:lastname ?lastname .
  BIND(CONCAT(?firstname, " ", ?lastname) AS ?sellername)
  FILTER (?priceperticket > ?avg_price)
}
ORDER BY desc(?priceperticket) ?sellername ?eventname
LIMIT 1000
```

- The WHERE clause includes a subquery to calculate the average price of tickets for each event listing. Since subqueries are processed first, that calculation (?avg_price) becomes available to use for comparing with all of the prices listed by sellers.
- Below the subquery, the other triples in the WHERE clause return the price per ticket and seller name for each listing. The FILTER (FILTER (?priceperticket > ?avg_price))

narrows the results to return just the listings where the price per ticket is greater than the average price for that listing.

People with the Same SSN

The query below reveals potential identity thieves by reporting on people who have the same social security number but different names.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?first1 ?last1 ?sameSSN ?first2 ?last2
FROM <http://anzograph.com/tickit>
WHERE {
  ?person1 tickit:ssn ?sameSSN .
  ?person1 tickit:firstname ?first1 .
  ?person1 tickit:lastname ?last1 .
  ?person2 tickit:ssn ?sameSSN .
  ?person2 tickit:firstname ?first2 .
  ?person2 tickit:lastname ?last2 .
  FILTER ( str(?person1) > str(?person2) )
}
ORDER by ?sameSSN
LIMIT 100
```

- The WHERE clause includes triples to compare the first name, last name, and social security number for the people in the person class. Using the same variable, sameSSN, for person1 and person2 limits the results to people who have the same SSN.
- The FILTER, like other queries, eliminates duplicate rows in the results. Since this filter performs the comparison on person1 and person2, which are subjects (URIs) in the tickit graph, the person URIs are converted to strings using the STR function.

Tickets Sold by Possible Identity Thieves

The query below builds on the previous query, "People with the Same SSN," by reporting on events where the seller who sold tickets is one of the people who has the same SSN as someone else.

```
PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?firstname ?lastname ?eventname ?location ?pricepaid ?date
FROM <http://anzograph.com/tickit>
WHERE {
  # get all event info
```

```

?eventid tickit:venueid ?venueid .
?venueid tickit:venueid ?location .
?eventid tickit:dateid ?dateid .
?dateid tickit:caldate ?date .
?eventid tickit:eventname ?eventname .
# note the sellers
?sales tickit:eventid ?eventid .
?sales tickit:pricepaid ?pricepaid .
?sales tickit:sellerid ?thief .
# limit to thieves
?person tickit:ssn ?sameSSN .
?thief tickit:ssn ?sameSSN .
?thief tickit:firstname ?firstname .
?thief tickit:lastname ?lastname .
FILTER ( ?person != ?thief )
}
ORDER BY ?date ?eventname ?location ?firstname ?lastname ?pricepaid
LIMIT 50

```

- In the WHERE clause, the first group of triples traverses the venue, event, and date data to find the location and date for each event.
- The second group of triples joins in the sales data to find the people who sold the tickets to the events.
- The third group of triples compares people's social security numbers and narrows the list of sellers to return only the people who have the same SSN.

Tickets Sold by Friends of Possible Identity Thieves

The query below takes the "Tickets Sold by Possible Identity Thieves" query one step further to report on whether any friends of a possible identity thief sold tickets to events.

```

PREFIX tickit: <http://anzograph.com/tickit>
SELECT ?thief_name ?friend_name ?eventname ?location ?pricepaid ?date
FROM <http://anzograph.com/tickit>
WHERE {
# get all ticket sales info
?eventid tickit:venueid ?venueid .
?venueid tickit:venueid ?location .
?eventid tickit:dateid ?dateid .
?dateid tickit:caldate ?date .

```

```

    ?eventid tickit:eventname ?eventname .
# limit to thieves
    ?person tickit:ssn ?sameSSN .
    ?thief tickit:ssn ?sameSSN .
    ?thief tickit:firstname ?thief_first .
    ?thief tickit:lastname ?thief_last .
    BIND(CONCAT(?thief_first, " ", ?thief_last) AS ?thief_name)
# note thieves friends
    ?friend tickit:friend ?thief .
    ?friend tickit:firstname ?friend_first.
    ?friend tickit:lastname ?friend_last .
    BIND(CONCAT(?friend_first, " ", ?friend_last) AS ?friend_name)
# note the sellers
    ?sales tickit:eventid ?eventid .
    ?sales tickit:pricepaid ?pricepaid .
    ?sales tickit:sellerid ?friend .
    FILTER ( ?person != ?thief )
}
ORDER BY ?date ?eventname ?location ?thief_name ?friend_name ?pricepaid
LIMIT 500

```

- In the WHERE clause, the first group of triples traverses the venue, event, and date data to find the location and date for each event.
- The second group of triples compares people's social security numbers to find the names of people who have the same SSN. To simplify the query's resulting columns, the CONCAT function is used to concatenate the first and last name of "thief": `BIND(CONCAT(?thief_first, " ", ?thief_last) AS ?thief_name)`. It also adds a space between the first and last name. And the BIND function binds the concatenated name to the variable `thief_name`.
- The third group of triples finds a list of the friends of people who have the same SSN.
- And the fourth group of triples finds any sales where the seller is one of the friends found by the third group of triples.

Finance

The finance queries focus on analyzing the financial data in the tickit graph.

- [Big Spenders](#)
- [Top Sales People](#)

Big Spenders

The query below finds the 100 people who spent the most on tickets for events.

```
PREFIX ticketit: <http://anzograph.com/ticketit>
SELECT ?first ?last (sum(?dollars) as ?spent)
FROM <http://anzograph.com/ticketit>
WHERE {
    ?person ticketit:firstname ?first .
    ?person ticketit:lastname ?last .
    ?sale ticketit:buyerid ?person .
    ?sale ticketit:pricepaid ?dollars
}
GROUP BY ?first ?last
ORDER BY desc(?spent) ?first ?last
LIMIT 100
```

- The triples in the WHERE clause traverse the person and sales classes, joining on the buyer ID, to list all of the people who bought tickets and the amount they spent per sale.
- The aggregate function in the SELECT list (`(sum(?dollars) as ?spent)`) calculates the sum of all the dollars spent by each person.
- Since the query uses the SUM aggregate function, the query includes a GROUP BY clause to group on the non-aggregate variables, ?first and ?last.
- To sort the results to list the biggest spenders first, the ORDER BY statement (`ORDER BY desc(?spent) ?first ?last`) lists the amount spent in descending order.

Top Sales People

The query below finds the 100 people who made the most money selling tickets.

```
PREFIX ticketit: <http://anzograph.com/ticketit>
SELECT ?first ?last ?category (sum(?dollars) as ?earned)
FROM <http://anzograph.com/ticketit>
WHERE {
    ?person ticketit:firstname ?first .
    ?person ticketit:lastname ?last .
```

```

    ?sale ticket:sellerid ?person .
    ?sale ticket:pricepaid ?dollars .
    ?sale ticket:eventid ?event .
    ?event ticket:catid ?cat .
    ?cat ticket:catname ?category
}
GROUP BY ?first ?last ?category
ORDER BY desc(?earned) ?first ?last ?category
LIMIT 100

```

- The triples in the WHERE clause traverse the person, event, category, and sales classes to list all of the people who sold tickets, the amount they earned per sale, and the category of the events.
- The aggregate function in the SELECT list (`(sum(?dollars) as ?earned)`) calculates the sum of all the dollars earned by each person.
- Since the query uses the SUM aggregate function, the query includes a GROUP BY clause to group on the non-aggregate variables, ?first, ?last, and ?category.
- To sort the results to list the biggest earners first, the ORDER BY statement (`ORDER BY desc(?earned) ?first ?last ?category`) lists the amount spent in descending order.

Working with Cypher and the Movie Data

This topic provides information about loading the Movies demo data and running example Cypher queries such as those described in the Neo4j sandbox environment (where the Movies database originated). The Movies data set is based on the graph database provided in the Neo4j sandbox environment. For users already familiar with Cypher, using this data set previews Graph Lakehouse support of the Cypher language. You can run many of the same Cypher commands and queries as you would in other environments that support Cypher.

This topic demonstrates how you can run Graph Lakehouse queries using Cypher language syntax if you prefer using Cypher instead of SPARQL.

- [Using the Cypher CLI \(AZGBOLT\)](#)
- [Using Bolt Protocol](#)
- [Loading Data with Cypher CREATE](#)
- [Getting to Know the Movies Data](#)
- [Running Cypher Queries](#)

Note

Cypher language support in Graph Lakehouse follows the open Cypher language specification as described in this Adobe Acrobat PDF document:

<https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf>

Refer to the [Cypher Query Language Reference](#) for a complete description of Graph Lakehouse Cypher language compatibility with the open Cypher language specification.

Cypher® is a registered trademark of Neo4j, Inc.

Using the Cypher CLI (AZGBOLT)

Graph Lakehouse uses the Bolt protocol to provide a client application interface and CLI, `azgbolt`, to run Cypher commands and queries. To view the syntax and command line arguments allowed with the Cypher CLI, simply type `azgbolt` on a new line and press **Enter**.

```
$ ./<install_path>/bin/azgbolt
```

The `azgbolt` CLI returns command line syntax and available arguments, along with a sampling of commonly-used commands and queries:

```
azgbolt (Bolt CLI) [-c "command"] [-f file] [-h hosturl] [-p port] [-nohead ]  
                  [-o outputfile][--help display this message]
```

For example, the following syntax runs a Cypher query:

```
azgbolt -c "any query"
```

Note

When running Cypher commands or queries from the `azgbolt` CLI, you can use the standard Linux shell method of escaping any embedded single or double quote characters . For example, with a character string such as "John Smith", contained within a Cypher query, you would escape each quotation mark character with the backslash (`\`) character, for example:

```
\ "John Smith\ "
```

The following example shows the syntax used to run a Cypher query in a file:

```
azgbolt -f /home/user/match.cql
```

Using Bolt Protocol

In addition to the `azbolt` CLI, users can also connect other applications that support the Bolt protocol to run Cypher queries against Graph Lakehouse data by specifying the Cypher Bolt protocol port (default 7088) following the Graph Lakehouse host server's IP address. That is:

```
<host_IP>:<Cypher_port>
```

Similarly, those same applications can run SPARQL commands and queries by specifying the SPARQL port (default 7098) following the Graph Lakehouse host server's IP address. That is:

```
<host_IP>:<SPARQL_port>
```

Loading Data with Cypher CREATE

Cypher CREATE statements to replicate the original Neo4j Movie dataset in Graph Lakehouse are available in a file you can download from the following location:

After saving the `movies.cql` file to an accessible location on your Graph Lakehouse server, you can run the following command to create the Movies dataset in Graph Lakehouse. The `movies.cql` file contains a single Cypher statement that includes multiple `CREATE IN <dataset>` commands.

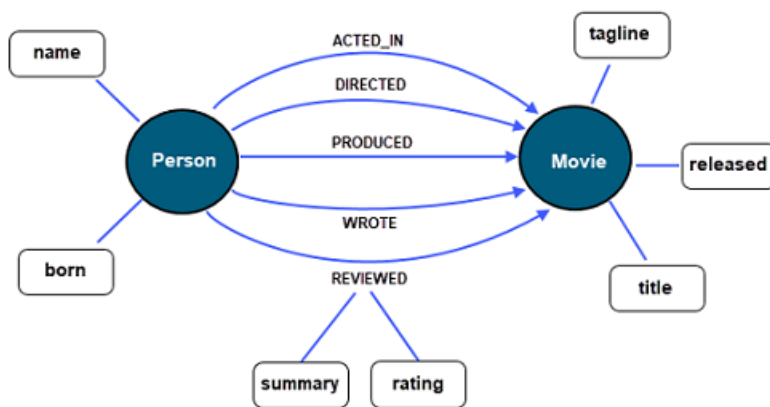
```
azgbolt -f /<filepath>/movies.cql
```

Note

The `IN <dataset>` clause is an Graph Lakehouse extension that was added to the standard Cypher language syntax to allow ingestion of data into a named dataset.

Getting to Know the Movies Data

The Movies dataset captures information about movies and the actors and directors involved with each of these films. To familiarize you with the Movies dataset, the following diagram shows the model or ontology for the Movies dataset.



The Movies database has two primary nodes **Person** and **Movie** with a number of different relationship types such as **ACTED_IN**, **WROTE**, **DIRECTED**, and **REVIEWED**. You can write Cypher queries to traverse the relationships between **Node** and **Movie** instances to retrieve node property values such as an actor's name or a specific movie's title, its director(s), and other information.

Running Cypher Queries

This section provides a brief introduction to the Cypher language. It also demonstrates execution of some basic sample Cypher queries run against Movie data stored in Graph Lakehouse. Like SPARQL, the Cypher language is especially designed for working with graph data and shares some similarities with SQL, with many SQL-like clauses and operations. The primary method of querying data with Cypher uses the MATCH command keyword.

This first query simply returns all nodes with a specified label (people). In this case, it returns the name of all people in the Movies dataset.

```
MATCH (people:Person)RETURN people.name ;
```

Note

Cypher keywords are case-insensitive, however, relationship types and property value are case-sensitive.

A second simple query returns all movie titles in the Movies dataset.

```
MATCH (films:Movie)RETURN films.title ;
```

Of course, Cypher supports more complex query operations that take full advantage of the relationships between entities or nodes that graph databases are able to capture. These capabilities involve fetching and filtering data, traversing graphs by joining data, using aggregate functions, and writing subqueries.

In addition, you may include any of the standard Graph Lakehouse built-in functions in Cypher queries.

Like SPARQL, MATCH statements provides options to specify patterns that Cypher will search for in the database. You can use labels and specify pattern restrictions based on specific relationship types and direction and use a WHERE clause to further filter results that a query returns. For example, using Tom Hanks as an example, you could run the following query to return a list of movies in which Tom acted in.

```
MATCH (actor:Person)-[:ACTED_IN]-(film:Movie) WHERE actor.name='Tom Hanks' RETURN
actor.name, film.title ;
```

The following diagram shows a graphic representation of nodes and relationship types in the Movies dataset using Tom Hanks, both an actor and director, as an example:



To further traverse the relationship between nodes in the Movies dataset, you could run the following query.

```
MATCH (actor:Person)-[:ACTED_IN]-(film:Movie), (director:Person)-[:DIRECTED]-(
film:Movie)
WHERE actor.name='Tom Hanks' RETURN actor.name, film.title, director.name ;
```

In this example, the MATCH pattern identifies and returns all the directors of Movies in which Tom Hanks acted.

Note

For more information on the Cypher language, see the opencypher.org project web site and the Cypher Query Language Reference available at <https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf>. Refer to the [Cypher Query Language Reference](#) for a complete description of Graph Lakehouse Cypher language compatibility with the open Cypher language specification.

Load & Manage Data

Graph Lakehouse supports loading data from RDF and non-RDF files, HTTP/REST endpoints, and relational databases via JDBC connections. The topics in this section describe the ways to load and manage your data.

In this section:

- [Load RDF Data from Files](#) 173
- [Load or Virtualize Non-RDF Sources with SPARQL Queries](#) 187
- [Use a Query Context](#) 401
- [Create a Labeled Property Graph \(RDF-star\)](#) 405
- [Infer New Data \(RDFS+ Inferencing\)](#) 424
- [Validate Data with SHACL \(Preview\)](#) 436
- [Copy Graphs to Files](#) 465
- [Schedule Automated Data Updates](#) 468

Load RDF Data from Files

The topics in this section provide instructions for loading data to Graph Lakehouse from files that are in RDF format: Turtle, N-Triple, N-Quad, TriG, or JSON-LD. RDF files are loaded in parallel using Graph Lakehouse's IO Load service, which enables Graph Lakehouse to read (and write) encrypted or non-encrypted data from remote file storage systems like Amazon, Google, and Azure object stores as well as web servers and network file systems. If you have local triple or quad files, you can also load data using the native SPARQL LOAD query. This section also includes information about file system and directory requirements as well as details about data type handling.

Note

For information about loading data from non-RDF data sources, see [Load or Virtualize Non-RDF Sources with SPARQL Queries](#).

In this section:

RDF Load File Requirements	174
Data Type Handling	177
Load RDF Files with the IO Load Service	179
Load Local RDF Files with SPARQL LOAD	183

RDF Load File Requirements

Graph Lakehouse supports loading RDF data from files on the Graph Lakehouse file system, on a remote web server or object store, or on a mounted file system. You can load data from a single file or multiple files in a directory. This topic provides details about the supported load file types, file storage systems, and load directory requirements.

- [Supported RDF File Types](#)
- [Supported File Systems](#)
- [Directory Name Requirements](#)
- [Note on URI Limitations](#)

Supported RDF File Types

Graph Lakehouse supports the following RDF load file types. See [Introduction to the Graph Data Interface](#) for information about supported non-RDF data sources.

- Turtle (.ttl file type): Terse RDF Triple Language that writes an RDF graph in compact form.
- N-Triple (.n3 and .nt file types): A subset of Turtle known as simple triples.
- N-Quad (.nq and .quads file types): N-Triples with a blank node or graph designation.
- TriG (.trig file type): An extension of Turtle that supports representing a complete RDF data set.
- JSON-LD (.jsonld file type): A method of encoding linked data using JSON. JSON-LD files are supported for loading via the IO services. JSON-LD is not supported by SPARQL LOAD queries.

You can compress any of the supported file types and load the compressed files into the database. The supported compression types are GZIP and ZST when using the IO services or GZIP when using SPARQL LOAD.

The Graph Lakehouse IO Load service supports decryption of load files using the Advanced Encryption Standard (AES). Cipher Block Chaining (CBC) and Galois/Counter Mode (GCM) with standard key sizes 128, 192, and 256 bits are supported.

Supported File Systems

When you have multiple files, Graph Lakehouse loads the files in parallel, using all available cores on all servers in the cluster. While you can load files stored on the leader node's local file system, for optimal performance, it is important to use a shared file system to ensure that all servers in the cluster have access to the files. In a Docker or Kubernetes container environment, the storage system should also be shared with the container file system.

The list below describes the supported file storage systems:

- Network File Systems (NFS) Version 4 or later
- Amazon Simple Cloud Storage Service (S3) object store
- Google Cloud Platform (GCP) object store
- Microsoft Azure Blob Storage
- Microsoft Azure WebDAV
- Web Server

Directory Name Requirements

In order to load a directory of files, the files must be organized in directories by file extension type, and the file type extension must be included in the name of the directory. For example, place TTL files in a **<name>.ttl** directory, place TRIG files in a **<name>.trig** directory, place NQ files in a **<name>.nq** directory, and so on. If the files in the directory are compressed (gzipped), add **.gz** to the directory name. For example, for a directory of gzipped TTL files, **<name>.ttl.gz**.

Note on URI Limitations

Important

Graph Lakehouse supports a maximum URI length of 16K characters. In addition, there is a limit of 64K on the number of **unique** predicate and graph URIs that can be stored in Graph Lakehouse. If the total number of unique predicate and graph URIs exceeds the 64K limit, the load operation that exceeds the limit will fail and Graph Lakehouse returns the message `m_lowest_unused_index <= a_max_value()`.

Data Type Handling

Graph Lakehouse natively supports the following RDF data types. Literal values with types that are not included in the table below are treated as "user-defined" types. User-defined types are stored as strings and can be cast to supported types as needed to perform analytic operations.

Data Type	Description
xsd:boolean	For true or false values. Regardless of whether the input value is "true" or "false" or "0" or "1," Graph Lakehouse stores and displays "t" for true and "f" for false. Note To use 1 and 0 for true and false, you must specify the xsd:boolean type in the load file. Otherwise the system assumes these values are integers.
xsd:byte	For 1-byte integers from -128 to 127.
xsd:date	For date values that follow a format such as YYYY-MM-DD . You can also include timezone indicators in xsd:date values.
xsd:dateTime	8-byte date and time values that follow a format such as YYYY-MM-DDThh:mm:ss . You can also include timezone indicators in xsd:dateTime values.
xsd:double	8-byte double floating point values. Note Decimal values are converted to xsd:double in Graph Lakehouse.
xsd:duration	Duration of time expressed as a number of years, months, days, hours, minutes, and seconds in a format such as PnYnMnDTnHnMnS .

Data Type	Description
xsd:float	4-byte floating point values with potential decimal places.
xsd:int	4-byte integers for values from -2,147,483,648 to 2,147,483,647.
xsd:long	8-byte integers for values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
xsd:short	2-byte integers for values from -32,768 to 32,767.
xsd:string	Character values of varying length, up to 2 MB in size. 2 MB holds approximately 2 million characters. Note To load data that contains strings that are longer than 2 MB, enable the truncate_clob setting. When this setting is enabled, strings that are larger than 2 MB are automatically truncated to 2 MB.
xsd:time	Time values that follow a format such as hh:mm:ss .

Load RDF Files with the IO Load Service

This topic provides instructions for loading locally- or remotely-stored RDF files to Graph Lakehouse using the IO Load service. See [RDF Load File Requirements](#) for details about the supported file types, encryption, storage systems, and the directory naming requirements.

For instructions on loading files with SPARQL LOAD queries, see [Load Local RDF Files with SPARQL LOAD](#).

- [Load Service Query Syntax](#)
- [Load Service Query Examples](#)

Load Service Query Syntax

The following query syntax shows the structure of a load service query. The clauses, patterns, and placeholders that are links are described below.

```
PREFIX io: <http://cambridgesemantics.com/anzograph/io#>
INSERT {
  [ GRAPH <graph_name> { ]
    ?sub ?pred ?obj
  [ } ]
}
WHERE {
  { SELECT ?sub ?pred ?obj .
    WHERE {
      SERVICE io:load('<protocol://path_to_files[,protocol://path_to_files][,...]>') {}
    }
  }
}
```

Option	Description
GRAPH <graph_name>	This clause is optional. When loading files such as Turtle or N-Triple files without graph specifications, include this optional clause to specify the graph to load data into. If the graph does not exist, Graph Lakehouse automatically creates it and then loads the data into it. If you do not specify a graph, the data is loaded to the default

Option	Description
	graph. You can also include the GRAPH clause when loading quad files. If the quad files contain a mixture of quads and triples, Graph Lakehouse loads the triples into the specified graph. Quads are still loaded according to their graph specification. If you omit this option for quad files, any triples without graph specifications are loaded into the default graph.
?sub ?pred ?obj	This triple pattern is required and the variable names must be <code>?sub ?pred ?obj</code>. The WHERE clause requires a subquery that selects the same triple pattern.
SERVICE io:load	This is the required call to the IO load service. If your query omits the PREFIX clause, include the full URI in the call: <code>SERVICE <http://cambridgesemantics.com/anzograph/io#load></code>.
protocol	The service call includes a URI that specifies the load protocol to use and the path to the load file or directory of files. The protocol that you specify depends on the type of file system that hosts the files: • Local File System: Specify <code>file</code> to access files that are stored on the local Graph Lakehouse file system or a file system that is mounted to the Graph Lakehouse servers. Including the <code>file://</code> protocol is optional. When files are locally accessible, you can omit the protocol and specify only the path to the file or directory. • NFS: Specify <code>nfs</code> to access files on an NFS that is not mounted to the Graph Lakehouse servers. • Web Server: Specify <code>http</code> or <code>https</code> (for SSL connections) to access files on a web server. • Amazon S3: Specify <code>s3</code> or <code>s3crt</code> to access files on S3. Using <code>s3crt</code> is recommended when loading extremely large files. The <code>S3CrtClient</code>

Option	Description
	improves the throughput for transfers of large files to and from Amazon S3. For more information about s3crt, see Using S3CrtClient for Amazon S3 operations in the AWS documentation. • Google Storage: Specify gs to access files on Google storage. • Azure Storage: Specify az to access files on Azure blob storage. • Azure WebDAV: Specify webdav or webdavs (for SSL connections) to access files on Azure WebDAV.
path_to_files	After the protocol in the service call URI, specify server connection details, if necessary, and the path to the load file or directory of files. When loading a directory of files, make sure the directory name includes the same file type extension as the files in the directory (see Directory Name Requirements for more information). Graph Lakehouse loads all valid files in that directory as well as any subdirectories. Hidden files that are named with a leading period, such as <code>.file.ttl</code>, are not loaded. See Protocol and Path Examples below for example URIs.

Protocol and Path Examples

The following example URI, loads a directory of compressed TTL files from Amazon S3:

```
<s3://shared-data/load-files/emr.ttl.gz>
```

The example below connects to an NFS that is not mounted and loads a single NT file:

```
<nfs://10.10.100.10/shared-data/load-files/rdf/sales-2022.nt>
```

This example loads a TTL file from a Google object store and another TTL file from a web server:

```
<gs://shared-data/load-files/emr-  
data.ttl/patients.ttl,https://10.30.103.3/emr/medications.ttl>
```

The following two examples load a directory of compressed TTL files from the Graph Lakehouse file system. The second example omits the `file://` protocol since it is optional:

```
<file:///opt/data/airlines/airline-data.ttl.gz>
```

```
</opt/data/airlines/airline-data.ttl.gz>
```

Load Service Query Examples

The example query below loads a directory of compressed TTL files from an Azure blob store:

```
PREFIX io: <http://cambridgesemantics.com/anzograph/io#>
INSERT {
  GRAPH <http://anzograph.com/emr> {
    ?sub ?pred ?obj
  }
}
WHERE {
  { SELECT ?sub ?pred ?obj .
    WHERE {
      SERVICE io:load('<az://shared-data/load-files/emr.ttl.gz>') {} .
    }
  }
}
```

This query loads a directory of compressed N3 files from Amazon S3:

```
PREFIX io: <http://cambridgesemantics.com/anzograph/io#>
INSERT {
  GRAPH <http://anzograph.com/sales> {
    ?sub ?pred ?obj
  }
}
WHERE {
  { SELECT ?sub ?pred ?obj .
    WHERE {
      SERVICE io:load('<s3://shared-data/load-files/sales.ttl.n3>') {} .
    }
  }
}
```

Load Local RDF Files with SPARQL LOAD

If you have Turtle, N-Triple, N-Quad, or TriG files on the local Graph Lakehouse file system or a mounted NFS, you have the option to load the data by running a native SPARQL LOAD query. For instructions on loading RDF files from a remote store, such as cloud object storage or a web server, see [Load RDF Files with the IO Load Service](#).

This topic lists the syntax to use for SPARQL LOAD queries and provides some examples to follow.

- [LOAD Syntax](#)
- [LOAD Examples](#)

LOAD Syntax

Run the following query to load data from Turtle, N-Triple, N-Quad, or TriG files. The options that are links are described below.

```
LOAD [ SILENT ] [ WITH 'global' | 'leader' | 'compute' ] <URI> [...<URIn>] [ INTO GRAPH <graph_uri> ]
```

Option	Description
SILENT	Include this optional keyword if you want Graph Lakehouse to ignore "bad data" errors during the load. Data issues are problems such as dateTime values that are incorrectly formatted or strings that are tagged as double data types. The SILENT keyword does not silence syntax errors in the files. If a file is ill-formed, such as if it includes invalid characters in place of URIs, Graph Lakehouse cannot parse the data and the file must be corrected. When SILENT is omitted, Graph Lakehouse aborts the load upon hitting a data or syntax error and reports the error to the client. When SILENT is included and Graph Lakehouse encounters an error with the data, it logs the error to a graph and proceeds with the load. By default, any errors are captured in the <code><load_errors></code> graph. After a load completes, you can query the graph to review errors. To customize the load error graph, you can change the <code>load_errors_graph</code> system setting. See Change System

Option	Description
	Settings for instructions. Important When SILENT is specified, the load will still be aborted if there are syntax errors in the files. Graph Lakehouse cannot parse the data if there are syntax errors. The file or files must be corrected and loaded again.
WITH	The optional WITH clause can be used to specify which servers in the cluster have access to the load files. You can choose one of the following options: • global: Include <code>WITH 'global'</code> when all servers in the cluster will load a subset of the same files or directories on a mounted file system. Include this option when every Graph Lakehouse server in the cluster has visibility to the entire data set. Graph Lakehouse automatically divides file selection among the servers. • leader: Include <code>WITH 'leader'</code> when loading files that only the leader server can access. <code>WITH 'leader'</code> is the default value for the LOAD query. When the WITH clause is omitted, the load proceeds as if <code>WITH 'leader'</code> was specified. • compute: Include <code>WITH 'compute'</code> when all servers will load files from their local file systems. Use this option if you have arranged the files so that each Graph Lakehouse server has a unique subset of files on its local file system. Note The leader, compute, and global keywords are case-sensitive. Type the terms using lower case letters.

Option	Description
URI	Required clause that specifies the absolute path to the load file or files. To load a single file, the scheme of the URI should be <code>file:.</code> To load a directory of files, the scheme of the URI should be <code>dir:.</code> When loading a directory, make sure the directory name includes the same file type extension as the files in the directory, i.e., a directory of TTL files is named <code>name.ttl</code>, a directory of TriG files is named <code>name.trig</code>, and a directory of NQ files is named <code>name.nq</code>. When you specify a directory, Graph Lakehouse loads all valid files in that directory as well as any subdirectories. Graph Lakehouse does not load any hidden files that are named with a leading period, such as <code>.file.ttl</code>. For example, the following URI loads a single file from a shared directory: <pre><file:/shared-files/data/tickit.ttl></pre> This example URI loads a directory of <code>.ttl.gz</code> files: <pre><dir:/global/nfs/vpc_nfs_server/data/tickit_all.ttl.gz></pre> And this example URI statement loads multiple directories of <code>.ttl.gz</code> files: <pre><dir:/global/nfs/data/tickit_all.ttl.gz> <dir:/global/nfs/data/movies.ttl.gz></pre> Note If you specify more than one URI to load from, each URI must target the same file type, such as <code>.ttl</code>, <code>.trig</code>, etc. Also each URI must specify the same scheme, <code>file:</code> or <code>dir:.</code>
INTO GRAPH <graph_uri>	When loading files such as Turtle or N-Triple files without graph specifications, include this optional clause to specify the graph to load data into. If the graph does not exist, the system automatically creates it and then loads the data into it. If you do not specify a graph, Graph Lakehouse loads data into the default graph.

Option	Description
	Tip You can also include INTO GRAPH when loading N-Quad files. If the N-Quad files contain a mixture of quads and triples, Graph Lakehouse loads the triples into the specified graph. Quads are still loaded according to their graph specification. If you omit this option for N-Quad files, any triples without graph specifications are loaded into the default graph.

LOAD Examples

The following example query loads data from gzipped TTL files in a directory on a mounted file system. Since all servers in the cluster have access to the file system, `WITH 'global'` is specified. The data is loaded into a graph named `http://anzograph.com/sales`:

```
LOAD WITH 'global' <dir:/global/nfs/data/sales_data.ttl.gz> INTO GRAPH
<http://anzograph.com/sales>
```

The example query below loads a shared directory of N-Quad files. Since the files include graph specifications, the INTO GRAPH clause is omitted:

```
LOAD WITH 'global' <dir:/global/nfs/data/employees.nq>
```

Load or Virtualize Non-RDF Sources with SPARQL Queries

The topics in this section provide information about exploring, analyzing, virtualizing, and loading non-RDF data by writing federated SPARQL queries that invoke the Graph Data Interface (GDI), a plugin that enables you to connect directly to sources and control all aspects of the extract, load, and transform process. Depending on the type of query you write, you can load data into Graph Lakehouse or create a virtual graph that accesses the source only when it is needed without ingesting the data.

In this section:

- [Introduction to the Graph Data Interface](#) 188
- [GDI Concepts and Basic Usage](#) 191
- [Options for Data Types, Data Connections, and Models](#) 256
- [Advanced Usage by Data Source Type](#) 272
- [GDI Property Reference](#) 385

Introduction to the Graph Data Interface

The Graph Data Interface (GDI) is an extremely flexible and configurable plugin that enables users to access a variety of data sources via federated SPARQL queries. Depending on the type of query you write, i.e., whether it is an INSERT query against the GDI service or a CONSTRUCT query against the view or virtualized service, you can load source data into Graph Lakehouse or create a virtual graph that accesses the source only when it is needed without ingesting the data into Graph Lakehouse.

The GDI has built-in, native support for various file format types, HTTP/REST endpoints, and common database types. Internally, the GDI API has a records-oriented view of data. This view enables the GDI to bridge graph operations to operations for data in other formats. Though the GDI views the source as rows in a table, ultimately it has the capability to convert the records to graph format, enabling the data to be incorporated into data layers to augment existing data.

Tip

When you query a source such as a database, the GDI service leverages that source to retrieve only the data that it needs for the query. Unlike a JDBC driver, the GDI service does not need to retrieve all values and then complete an often time-consuming step to filter the results.

- [Supported Data Sources](#)
- [Data Source Connections and Authentication](#)

Supported Data Sources

This table below lists the data sources, file systems, and applications that the GDI supports.

Source	Description
HTTP/REST Endpoints	The GDI natively supports reading or ingesting data from HTTP/REST endpoints.
Databases	Altair supplies JDBC drivers for the following databases. For information about

Source	Description
	acquiring additional JDBC drivers for connecting to other databases, contact your Altair Customer Success manager. • Databricks • H2 • IBM DB2 • Microsoft SQL Server • MariaDB • Oracle • PostgreSQL • SAP Sybase (jTDS) • Snowflake
File Formats	The following file types are supported: • CSV and TSV • JSON and NDJSON • Parquet • SAS (SAS Transport XPT and SAS7BDAT formats) • XML • Raw text format
File Systems	The following types of file storage systems are supported: • Amazon S3 • FTP & FTPS • Google Cloud Storage

Source	Description
	• HDFS (Kerberized HDFS is not supported at this time.) • NFS • SFTP • WebDAV
Applications	Queries against Elasticsearch and Kafka applications are supported.

Data Source Connections and Authentication

When connecting to data sources, connection parameters like keys, tokens, and user credentials are provided as part of the query that you run against that source. To avoid including sensitive information in each request, however, Graph Lakehouse provides the option to create and manage **Query Contexts**. A context specifies all of the connection details for a source. Queries simply reference the context so that sensitive information is abstracted from the request. For more information about contexts, see [Use a Query Context](#).

GDI Concepts and Basic Usage

The topics in this section help you get to know the Graph Data Interface (GDI) by introducing you to the main concepts and giving a general overview of the query syntax, available properties, and functionality that is applicable across query and data source types.

In this section:

- [Getting Started with GDI Queries](#) 191
- [Generating a Knowledge Graph](#) 207
- [Reading Data Source Metadata](#) 222
- [Pagination Options](#) 239
- [Binding and Hierarchy Concepts](#) 243
- [Incremental Load Concepts](#) 251

Getting Started with GDI Queries

This topic provides details about the structure to use when writing GDI queries. It focuses on the properties that are common to all types of data sources. It also includes example queries that demonstrate the data integration capabilities for different types of sources.

Tip

Rather than manually writing complex queries, you can use the GDI to automatically generate graphs and ontologies by including a few key statements in a relatively simple query. For information, see [Generating a Knowledge Graph](#).

- [GDI Query Syntax](#)
- [GDI Query Examples](#)

GDI Query Syntax

The following query syntax shows the structure of a GDI query. The clauses, patterns, and placeholders that are links are described below.

```

# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
{
  [ GRAPH <target_graph> { ]
    triple_patterns
  [ ] ]
}
[ FROM Clause ]
WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>)

  {
    ?data a s:source_type ;
  }
}

# Based on the source_type, additional connection and input parameters are
# available. The options below are valid for all sources. For source-related
# options, see the GDI Property Reference.
s:url "string" ;
[ s:username "string" ; ]
[ s:password "string" ; ]
[ s:timeout int ; ]
[ s:maxConnections int ; ]
[ s:batching boolean | int ; ]
[ s:concurrency int | [ list_of_properties ] ; ]
[ s:rate int | "string" ; ]
[ s:locale "string" ; ]
[ s:sampling int ; ]
[ s:selector "string" | [ list ] ; ]
[ s:model "string" ; ]
[ s:key ("string") ; ]

```



```

[ s:reference [ s:model "string" ; s:using ("string") ]
[ s:formats [ datatype_formatting_options ] ; ]
[ s:normalize boolean | [ normalization_rules ] ; ]
[ s:count ?variable ; ]
[ s:offset int ; ]
[ s:limit int ; ]
# Mapping variables
?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
... ;
.
# Additional clauses such as BIND, VALUES, FILTER
}
}

```

Note

For readability, the parameters below exclude the base URI

<<http://cambridgesemantics.com/ontologies/DataToolkit#>> as well as the `s:` prefix. As shown in the examples, however, the `s:` prefix or full property URI does need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause defines the type of SPARQL query to run and the set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN]</code> < http://cambridgesemantics.com/services/DataToolkit > to invoke the GDI service when you are running a SELECT, INSERT, or CONSTRUCT query that is not creating a view. When

Option	Type	Description
		creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: <code>SERVICE <http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>)</code>. Using the <code>DataToolkitView</code> call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not necessarily returned.
source_type	object	The <code>?data a s:source_type</code> triple pattern specifies the type of data source that the query will run against. For example, <code>?data a s:DbSource</code>, specifies that the source type is a database. The list below describes the available types: • DbSource to connect to any type of database. • FileSource for flat files. The supported file types are CSV and TSV, JSON, NDJSON, XML, Parquet, and SAS (SAS Transport XPT and SAS7BDAT formats). The GDI automatically determines the file type from the file extensions.

Option	Type	Description
		• HttpSource to connect to HTTP endpoints. • ElasticSource to connect to Elasticsearch indexes on an Elasticsearch server. • KafkaSource to connect to Kafka streaming sources. • MetadataSource for metadata discovery. Tip Certain connection and input parameters are available based on the specified source type. For details about the options for your source, see GDI Property Reference.
url	string	This property specifies the URL for the data source, such as the database URL, Elasticsearch URL, or HTTP endpoint URL. For file-based sources, the <code>url</code> property specifies the file system location of the source file or directory of files. When specifying a directory (such as <code>s:url "/opt/shared-files/loads/"</code>), the GDI loads all of the file formats it recognizes. To specify a directory but limit the number or type of files that are read, you can include the <code>pattern</code> and/or <code>maxDepth</code> properties described in FileSource Properties. Important For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example, the triple patterns below reference keys from

Option	Type	Description
		a Query Context: <pre>?data a s:DbSource ; s:url "{{@Somedb.url}}" ; s:username "{{@Somedb.user}}" ; s:password "{{@Somedb.password}}" ;</pre>
username	string	If authentication is required to access the source, include this property to specify the user name.
password	string	This property lists the password for the given username.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
maxConnections	int	This property can be used to set a limit on the maximum number of active connections to the source. For example, <code>s:maxConnections 16</code> sets the limit to 16 connections. When not specified, the default value is 10.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or	This property can be included to configure the maximum level of

Option	Type	Description
	RDF list	concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code>. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate.

Option	Type	Description
		Given the example of <code>s:rate "90/minute"</code> , the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code> , the GDI would use 3 executors with a rate delay of 750ms between requests.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the SalesOrderHeader table in the Sales schema. For more information about binding components and the selector property, see Using Binding Trees and Selector Paths .
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For

Option	Type	Description
		example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or <code>dateTime</code> values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	boolean and/or RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.

Option	Type	Description
mapping_variable	variable	The mapping variables, in <code>?mapping_variable (["binding"] [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Note The parentheses around the binding, data type, and format specifications are not required but are included in this document for readability.
binding	string	The binding is a literal value that binds a <code>?mapping_variable</code> to a source column. If you specify a <code>?variable</code> that matches the source column name, then that variable name is the data selector and it is not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column, then the binding is needed to map the new variable to that source column. For example for a flat source like CSV, the following pattern simply binds the source column AIRLINE to the lowercase variable <code>?airline</code>: <code>?airline ("AIRLINE")</code>. For a database source, this example binds the <code>?subject</code> variable by navigating to the SUBJECT column in the FILM table in the dbo schema: <code>?subject ("dbo.FILM.SUBJECT")</code>. And for an HTTP source, this example binds the <code>?time</code> variable to the time object under the minutely data path: <code>?time ("minutely.data.time")</code>.

Option	Type	Description
		Note For FileSource and HttpSource, periods (.), forward slashes (/), and brackets ([]) are parsed as path notation. Therefore, if a source column name includes any of those characters they must be escaped in the binding. Use two backslashes (\\) as an escape character. For example, if a column name is average/day, the variable and binding pattern could be written as <code>?averagePerDay ("average\\/day")</code>. For DbSource, database, schema, and table names in bindings are parsed according to the specific rules for that database type. You do not need to escape characters in database names. However, database names with characters that do not match <code>(_ A-Z a-z)(_ A-Z a-z 0-9)*</code> should be quoted, such as <code>('Adventure.Works'.Sales.'Daily.Totals')</code>.
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double, xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration, xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI</pre>
datetime_format	string	This option is used to specify the format to use for date and time data types. The GDI supports Java date and time formats. Specify

Option	Type	Description
		days as "d," months as "M," and years as "y." For the time, specify "H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMMy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation ^nnnn (e.g., ^1900) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-yy^1900"</code> or <code>xsd:date "dd-MMM-yy^1990"</code>. When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

GDI Query Examples

The query below reads data from a sample HTTP source that compiles worldwide weather statistics. The source has several models available for retrieving data that is current, daily, historical, etc. To target current data, the query includes `s:selector "currently"` as an input parameter. In addition, the query demonstrates the use of the "topdown" functionality, where the query sends values to the source to narrow the results. The query includes the TOPDOWN keyword in the GDI service call, and the VALUES clause specifies the latitude and longitude values for the cities to return data for. In addition, since this sample source requires parameters to be specified in the connection URL, the `s:url` value includes `?lat` and `?long` as parameters as part of the value.

```
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
```

```

PREFIX owl:    <http://www.w3.org/2002/07/owl#>
PREFIX anzo:    <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:    <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX ex:      <http://example.org/ontologies/City#>

SELECT
    ?city ?state ?summary ?temp ?rainChance
    ?humidity ?pressure ?windSpeed
WHERE
{
    SERVICE TOPDOWN <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a s:HttpSource ;
        s:url "https://sampleEndpoint.com/forecast/{{?lat}},{{?long}}" ;
        s:selector "currently" ;
        ?lat ("latitude") ;
        ?long ("longitude") ;
        ?temp ("temperature") ;
        ?rainChance ( "precipProbability" ) ;
        ?humidity ( ) ;
        ?pressure ( ) ;
        ?windSpeed ( ) .
    }
    VALUES( ?city ?state ?lat ?long )
    {
        ( "Lakeway" "TX" 30.374563 -97.975892 )
        ( "Boston" "MA" 42.358043 -71.060415 )
        ( "Seattle" "WA" 47.590720 -122.307053 )
        ( "Chicago" "IL" 41.837741 -87.823296 )
        ( "Hilo" "HI" 19.702040 -155.090312 )
    }
}
ORDER BY ?city

```

The query returns the following results:

city	state	summary	temp	rainChance	humidity	pressure	windSpeed
Boston	MA	Overcast	79.81	0	0.6	1018.7	7.71
Chicago	IL	Clear	81.7	0	0.52	1021.1	5.13
Hilo	HI	Partly Cloudy	72.6	0.13	0.79	1018.6	4.86
Lakeway	TX	Partly Cloudy	92.43	0	0.48	1013.3	10.85

Seattle		WA		Mostly Cloudy		61.82		0		0.76		1018.2		4.57
5 rows														

The example below ingests data from a database source using an INSERT query.

```
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>
PREFIX :       <http://example.com/ontologies/kl_hosp#>
```

```
INSERT
{
  GRAPH <http://anzograph.com/orders>
  {
    ?InputEvent_cv a :InputEvent_cv ;
      :row_id ?row_id ;
      :subject_id ?subject_id ;
      :hadm_id ?hadm_id ;
      :icustay_id ?icustay_id ;
      :charttime ?charttime ;
      :itemid ?itemid ;
      :amount ?amount ;
      :amountuom ?amountuom ;
      :rate ?rate ;
      :rateuom ?rateuom ;
      :storetime ?storetime ;
      :cgid ?cgid ;
      :orderid ?orderid ;
      :linkorderid ?linkorderid ;
      :stopped ?stopped ;
      :newbottle ?newbottle ;
      :originalamount ?originalamount ;
      :originalamountuom ?originalamountuom ;
      :originalroute ?originalroute ;
      :originalrate ?originalrate ;
      :originalrateuom ?originalrateuom ;
      :originalsite ?originalsite .
  }
}
```

```

}
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a s:DbSource ;
        s:url "{{@db.eca4bfa83481f3638b93ab5fdf93ff9a.url}}" ;
        s:username "{{@db.eca4bfa83481f3638b93ab5fdf93ff9a.user}}"
        s:password "{{@db.eca4bfa83481f3638b93ab5fdf93ff9a.password}}"
        s:selector "kl_hosp_schema.inputevents_cv" ;
        ?row_id (xsd:int) ;
        ?subject_id (xsd:int) ;
        ?hadm_id (xsd:int) ;
        ?icustay_id (xsd:int) ;
        ?charttime (xsd:dateTime) ;
        ?itemid (xsd:int) ;
        ?amount (xsd:float) ;
        ?amountuom (xsd:string) ;
        ?rate (xsd:float) ;
        ?rateuom (xsd:string) ;
        ?storetime (xsd:dateTime) ;
        ?cgid (xsd:int) ;
        ?orderid (xsd:int) ;
        ?linkorderid (xsd:int) ;
        ?stopped (xsd:string) ;
        ?newbottle (xsd:int) ;
        ?originalamount (xsd:float) ;
        ?originalamountuom (xsd:string) ;
        ?originalroute (xsd:string) ;
        ?originalrate (xsd:float) ;
        ?originalrateuom (xsd:string) ;
        ?originalsite (xsd:string) ;
        BIND(IRI("http://example.com/inputevent_cv/{{?row_id}}") AS ?InputEvent_cv)
        BIND(IRI("http://example.com/patients/{{?subject_id}}") AS ?patient)
        BIND(IRI("http://example.com/admissions/{{?hadm_id}}") AS ?admission)
    }
}

```

The following query ingests airport-related data from a CSV file.

```

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>

```

```

PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:    <http://www.w3.org/2002/07/owl#>
PREFIX anzo:     <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:     <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:       <http://purl.org/dc/elements/1.1/>

INSERT
{
  GRAPH <http://anzograph.com/airports>
  {
    ?code a <http://anzograph.com/airport> ;
      <http://anzograph.com/airport/name> ?name ;
      <http://anzograph.com/airport/city> ?city ;
      <http://anzograph.com/airport/state> ?state ;
      <http://anzograph.com/airport/latitude> ?lat;
      <http://anzograph.com/airport/longitude> ?long.
  }
}
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource ;
      s:url "/opt/shared-files/airports.csv" ;
      ?iata_code ("IATA_CODE" xsd:string) ;
      ?name ("AIRPORT" xsd:string) ;
      ?city ("CITY" xsd:string) ;
      ?state ("STATE" xsd:string) ;
      ?lat ("LATITUDE" xsd:double) ;
      ?long ("LONGITUDE" xsd:double).
    BIND(IRI("http://anzograph.com/airport/{{?iata_code}}") as ?code)
  }
}

```

The query below creates a view of a database source.

```

PREFIX s:        <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:     <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:    <http://www.w3.org/2002/07/owl#>
PREFIX anzo:     <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:     <http://openanzo.org/ontologies/2009/05/AnzoOwl#>

```

```

PREFIX dc:    <http://purl.org/dc/elements/1.1/>
PREFIX ont:   <http://cambridgesemantics.com/ont/autogen/Rh/MIMIC-III-Data_Source/mimic_iii_schema#>

CONSTRUCT
{
    ?caregiversURI a ont:caregivers ;
    ont:caregivers_cgid ?cgid ;
    ont:caregivers_description ?description ;
    ont:caregivers_label ?label .
}
WHERE
{
    GRAPH ?g
    {
        SERVICE <http://cambridgesemantics.com/services/DataToolkitView>
        (<http://anzograph.com/caregivers>) {
            {
                ?data a s:DbSource ;
                s:url "{{@mimicdb.url}}" ;
                s:username "{{@mimicdb.user}}"
                s:password "{{@mimicdb.password}}"
                s:selector "mimic_iii_schema.caregivers" ;
                ?row_id (xsd:int) ;
                ?cgid (xsd:int) ;
                ?label (xsd:string) ;
                ?description (xsd:string) .
                BIND(IRI ("http://anzograph.com/class/caregivers/{{?row_id}}") AS ?caregiversURI)
            }
        }
    }
}

```

Generating a Knowledge Graph

With no mapping required, the Graph Data Interface (GDI) can automatically generate a graph and an ontology for a non-RDF data source. By running a relatively simple SPARQL query to invoke the RDF and Ontology Generators, the GDI determines the structure of a data source and automatically generates the necessary RDF statements.

Invoking the Generators is preferable to producing a hand-written query, especially when the structure of the data is very complex, such as a JSON data source with many inner repeating structures or a database with many tables and keys. When the source contains complex structures, the GDI will generate only the required statements and avoid cross-products, optimizing query execution and memory usage. In addition, the GDI Generator parallelizes the load across the Graph Lakehouse cluster so that a data source (such as a database) can be ingested with a single query. This topic provides details about invoking the GDI RDF and Ontology Generators. The Generators can be used with all of the supported data source types.

- [GDI Generator Query Syntax](#)
- [GDI Generator Example Queries](#)

GDI Generator Query Syntax

The following query syntax shows the structure of a GDI Generator query. The clauses, patterns, and placeholders that are links are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

#Result Clause
INSERT {
    GRAPH <target_graph> {
        ?s ?p ?o .
    }
}
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a s:source_type ;
# Based on the source_type, additional connection and input parameters are
# available. The options below are valid for all sources. For source-related
# options, see the GDI Property Reference.
        s:url "string" ;
```



```

[ s:model "class_name_for_this_source" ; ]
[ s:username "string" ; ]
[ s:password "string" ; ]
[ s:timeout int ; ]
[ s:maxConnections int ; ]
[ s:batching boolean | int ; ]
[ s:concurrency int | [ list_of_properties ] ; ]
[ s:rate int | "string" ; ]
[ s:locale "string" ; ]
[ s:sampling int ; ]
[ s:selector "string" | [ list ] ; ]
[ s:key ("string") ; ]
[ s:reference [ s:model "string" ; s:using ("string") ]
[ s:formats [ datatype_formatting_options ] ; ]
[ s:normalize boolean | [ source_normalization_rules ] ; ]
[ s:count ?variable ; ]
[ s:offset int ; ]
[ s:limit int ; ] .

```

Multiple data sources can be merged if they project a similar set
of output variables. Make sure each source has a unique subject variable.

```

[ ?unique_variable a s:source_type ;
...
. ]

```

```

?rdf a s:RdfGenerator, s:OntologyGenerator ;
s:as (?s ?p ?o);
s:ontology ontology_uri ;
s:base base_uri ;
[ s:normalize boolean | [ global_normalization_rules ] ; ]
.

```

Additional clauses such as BIND, FILTER

```

}
}

```

Note

For readability, the parameters below exclude the base URI

<<http://cambridgesemantics.com/ontologies/DataToolkit#>> as well as the s: prefix. As shown in the examples, however, the s: prefix or full property URI does need to be

included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause for GDI Generator queries is typically an INSERT query with the graph pattern in the template above.
source_type	object	The <code>?data a s:source_type</code> triple pattern specifies the type of data source that the query will run against. For example, <code>?data a s:DbSource</code>, specifies that the source type is a database. The list below describes the available types: • DbSource to connect to any type of database. • FileSource for flat files. The supported file types are CSV and TSV, JSON, NDJSON, XML, Parquet, and SAS (SAS Transport XPT and SAS7BDAT formats). The GDI automatically determines the file type from the file extensions. • HttpSource to connect to HTTP endpoints. • ElasticSource to connect to Elasticsearch indexes on an Elasticsearch server. • KafkaSource to connect to Kafka streaming sources.

Option	Type	Description
Tip Certain connection and input parameters are available based on the specified source type. For details about the options for your source, see GDI Property Reference.		
url	string	This property specifies the URL for the data source, such as the database URL, Elasticsearch URL, or HTTP endpoint URL. For file-based sources, the <code>url</code> property specifies the file system location of the source file or directory of files. When specifying a directory (such as <code>s:url "/opt/shared-files/loads/"</code>), the GDI loads all of the file formats it recognizes. To specify a directory but limit the number or type of files that are read, you can include the <code>pattern</code> and/or <code>maxDepth</code> properties described in FileSource Properties. Important For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example, the triple patterns below reference keys from a Query Context: <pre>?data a s:DbSource ; s:url "{{@Somedb.url}}" ;</pre>

Option	Type	Description
		<pre>s:username "{{@Somedb.user}}" ; s:password "{{@Somedb.password}}" ;</pre>
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
username	string	If authentication is required to access the source, include this property to specify the user name.
password	string	This property lists the password for the given username.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
maxConnections	int	This property can be used to set a limit on the maximum number of active connections to the source. For example, <code>s:maxConnections 16</code> sets the limit to 16 connections. When not specified, the default value is 10.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the

Option	Type	Description
		query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code>.
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code>. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values

Option	Type	Description
		below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code>, the GDI would use 3 executors with a rate delay of 750ms between requests.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the SalesOrderHeader table in the Sales schema. For more information about binding components and the selector property, see Using Binding Trees and Selector

Option	Type	Description
Paths.		
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
count	variable	If you want to turn the query into a COUNT query, you

Option	Type	Description
		can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
RdfGenerator	object	Include this property to invoke the RDF Generator. If you only want to generate a model without RDF, you can <code>exclude RdfGenerator</code> .
OntologyGenerator	object	Include this property to invoke the Ontology Generator. If you only want to generate RDF without a model, you can <code>exclude OntologyGenerator</code> .
as	N/A	This property provides the variable bindings for the RDF Generator's projection to RDF. Typically the value is <code>s:as (?s ?p ?o)</code> to match the variables in the result clause.
ontology	URI	This property specifies the URI to use as the base URI for any generated ontology artifacts. For example, <code>s:ontology <http://abc.com/ontologies/MyOntology></code> .
base	URI	This property specifies the base URI for instance data. The base value should NOT end in <code>#</code> . The Generator will add a trailing slash (<code>/</code>) if one does not exist. For example, <code>s:base <http://abc.com/></code> .

GDI Generator Example Queries

This section includes sample queries that may be useful as a starting point for writing your own RDF and Ontology Generator queries.

- [Basic Query that Generates RDF and Ontology for a JSON File](#)
- [Basic Query that Generates an Ontology for a Directory of CSV Files](#)
- [Query that Normalizes and Generates RDF and Ontology for a Database](#)
- [Query with Query Context that Normalizes and Generates RDF and Ontology for a Database](#)
- [Query for Multiple Sources that Generates RDF and Ontology with Resource Templates and Object Properties](#)

Basic Query that Generates RDF and Ontology for a JSON File

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
  GRAPH <http://anzograph.com/people> {
    ?s ?p ?o .
  }
}
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

    ?data a s:FileSource ;
      s:model "People" ;
      s:url "/opt/shared-files/json/people.json" .

    ?rdf a s:RdfGenerator , s:OntologyGenerator ;
      s:as (?s ?p ?o) ;
      s:ontology <http://anzograph.com/ontologies/People> ;
      s:base <http://anzograph.com/data/> .

  }
}
```

Basic Query that Generates an Ontology for a Directory of CSV Files

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
```

```

GRAPH <http://anzograph.com/sales> {
    ?s ?p ?o .
}
}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

        ?data a s:FileSource ;
        s:model "Sales" ;
        s:url "/opt/shared-files/csv/sales" ;
        s:format [
            s:delimiter "," ;
            s:headers true ;
            s:comment "#" ;
            s:quote "\"" ;
            s:maxColumns 22 ;
        ] .

        ?rdf a s:OntologyGenerator ;
        s:as (?s ?p ?o) ;
        s:ontology <http://anzograph.com/ontologies/Sales> ;
        s:base <http://anzograph.com/data/> .
    }
}

```

Query that Normalizes and Generates RDF and Ontology for a Database

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
    GRAPH <http://anzograph.com/emr> {
        ?s ?p ?o .
    }
}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

        ?data a s:DbSource ;
        s:url "jdbc:mysql://10.11.12.9/emrdbbig" ;
        s:username "root" ;
        s:password "sql1@#" ;
        s:normalize [
            s:model [

```

```

        s:removeStart "emr_" ;
        s:words "activity 'patient complaint' medication observation patient
specialty study" ;
    ] ;
    s:field [
        s:removePartialPrefix true ;
        s:words "provider description start end drug complaint date medication
normal code
                observation product active dose generic route admin strength
collection
                activity home first last status first year birth death directed
complex
                period age flag gender language" ;
    ] ;
] .

?rdf a s:RdfGenerator , s:OntologyGenerator ;
s:as (?s ?p ?o) ;
s:ontology <http://anzograph.com/ontologies/EMR> ;
s:base <http://anzograph.com/EMR> .
}
}

```

Query with Query Context that Normalizes and Generates RDF and Ontology for a Database

The query below references a Query Context to supply the username and password for the database connection.

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
    GRAPH <http://anzograph.com/adventureworks> {
        ?s ?p ?o .
    }
}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

        ?data a s:DbSource ;
        s:url "jdbc:sqlserver://localhost;databaseName=AdventureWorks2012" ;
        s:username "{{@adventureworksdb.username}}" ;
        s:password "{{@adventureworksdb.password}}" ;
        s:schema "Production", "HumanResources", "Person", "Sales", "Purchasing" ;
    }
}

```

```

s:normalize [
  s:model [
    s:localNamePrefix "C_" ;
    s:localNameSeparator "_" ;
    s:match [ s:pattern "(.+)"Enlarged" ; s:replace "$1" ] ;
  ] ;
  s:field [
    s:localNamePrefix "P_" ;
    s:localNameSeparator "_" ;
    s:ignore "rowguid ModifiedDate" ;
    s:match (
      [ s:pattern "(.+)"GUID$" ; s:replace "$1" ]
      [ s:pattern "(.+)"ID$" ; s:replace "$1" ]
    ) ;
  ] ;
] .

?rdf a s:RdfGenerator, s:OntologyGenerator ;
s:as (?s ?p ?o) ;
s:ontology <http://anzograph.com/ontologies/AdventureWorks> ;
s:base <http://anzograph.com/AdventureWorks> .
}

```

Query for Multiple Sources that Generates RDF and Ontology with Resource Templates and Object Properties

This query also includes global normalization rules for normalizing the data across all sources.

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
  GRAPH <http://anzograph.com/tickets> {
    ?s ?p ?o .
  }
}
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

    ?event a s:FileSource ;
    s:model "event" ;
    s:url "/opt/shared-files/csv/events.csv" ;
    s:key ("EVENT_ID") .
  }
}

```

```

?listing a s:FileSource ;
    s:model "listing" ;
    s:url " /opt/shared-files/csv/listings.csv" ;
    s:key ("LIST_ID") ;
    s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] .

?date a s:FileSource ;
    s:model "date" ;
    s:url " /opt/shared-files/csv/event_dates.csv" ;
    s:key ("DATE_ID") ;
    s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] .

?venue a s:FileSource ;
    s:model "venue" ;
    s:url " /opt/shared-files/csv/venues.csv" ;
    s:key ("VENUE_ID") ;
    s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] .

?sale a s:FileSource ;
    s:model "sale" ;
    s:url " /opt/shared-files/csv/sales.csv" ;
    s:key ("SALE_ID") ;
    s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] ;
    s:reference [ s:model "listing" ; s:using ("LIST_ID") ; s:key ("LIST_ID") ] .

?rdf a s:RdfGenerator, s:OntologyGenerator ;
    s:as (?s ?p ?o) ;
    s:ontology <http://anzograph.com/tickets> ;
    s:base <http://anzograph.com/data> ;
    s:normalize [
        s:all [
            s:casing s:UPPER ;
            s:localNameSeparator "_" ;
        ] ;
    ] .
}

```

Reading Data Source Metadata

If you want to retrieve instance data from a source but are unsure about the data model, schema, or the exact names of columns and their data types, you can use the Graph Data Interface (GDI) to explore the source's metadata. The GDI can be used to return a list of the catalogs (schemas), models, columns, data types, and other data source information.

This topic describes the metadata query syntax and provides several example queries.

- [Metadata Query Syntax](#)
- [Metadata Query Examples](#)

Metadata Query Syntax

The following query syntax shows the structure of a metadata query. The clauses, patterns, and placeholders in blue are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
SELECT *
WHERE
{
    # SERVICE Clause: Include the following service call
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:source_type ;
            s:url "string" ;
            [ s:username "string" ; ]
            [ s:password "string" ; ]
    }
}
```

```

    ?metadata a s:MetadataSource ;
      s:from ?data ;

# The metadata selector below specifies the type of metadata to return.
    ?catalogs | ?fields | ?models [
      ?metadata_type datatype ;
      ... ;
    ] .
  }
}

```

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause defines the results to return. For metadata queries, the result clause is typically <code>SELECT *</code> .
SERVICE Clause		Include the required GDI SERVICE call in the WHERE clause. The rest of the WHERE clause defines the patterns to look for in the source.
[] s:select ?metadata	N/A	Include this required triple pattern in metadata queries. The select property specifies the source that should be used to return data.
source_type	object	The <code>?data a s:source_type</code> triple pattern specifies the type of data source that the query will run against. For example, <code>?data a s:DbSource</code>, specifies that the source type is a database. The list below describes the available types: • DbSource to connect to any type of database. • FileSource for flat files. The supported file types are CSV and TSV, JSON, NDJSON, XML, Parquet, and SAS (SAS

Option	Type	Description
		Transport XPT and SAS7BDAT formats). The GDI automatically determines the file type from the file extensions. • HttpSource to connect to HTTP endpoints. • ElasticSource to connect to Elasticsearch indexes on an Elasticsearch server. • KafkaSource to connect to Kafka streaming sources. Tip Certain connection and input parameters are available based on the specified source type. For details about the options for your source, see GDI Property Reference.
url	string	This property specifies the URL for the data source, such as the database URL, Elasticsearch URL, or HTTP endpoint URL. For file-based sources, the <code>url</code> property specifies the file system location of the source file or directory of files. Important For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example, the triple patterns below reference keys from a Query Context: <pre>?data a s:DbSource ;</pre>

Option	Type	Description
		<pre>s:url "{{@Somedb.url}}" ; s:username "{{@Somedb.user}}" ; s:password "{{@Somedb.password}}" ;</pre>
username	string	If authentication is required to access the source, include this property to specify the user name.
password	string	This property lists the password for the given username.
catalogs	variable	This selector narrows the results to schema-related metadata such as the schema names. Even when additional metadata types (metadata_type datatype) are specified as objects, only catalog (schema) information is returned.
fields	variable	This selector is the broadest and most flexible option. Using the <code>fields</code> selector enables users to return any and all of the source metadata information, depending on the specified metadata types (metadata_type datatype).
models	variable	This selector narrows the results to model-related metadata such as the model names. Even when additional metadata types (metadata_type datatype) are specified as objects, only model information is returned.
metadata_type datatype	N/A	The triple patterns in the array for the metadata selector specify the type of metadata to return as well as the data type for the return value. The following list shows all of the valid options. You can include any combination of properties. The results that are returned depend on the type of data source and whether the information exists in the source. The parentheses around the data type are not required but are included in this document for readability.

Option	Type	Description
		• ?model (xsd:string): Returns model names in string format. For file sources, this property returns file names. • ?field (xsd:string): Returns column names. • ?catalog (xsd:string): Returns schema names. • ?datatype (owl:Thing): Returns the data types of the columns. • ?keys (xsd:string): Returns primary and foreign key columns. For compound keys, the GDI returns a comma-separated list of columns comprising the key. • ?format (xsd:string): Returns the format of the source. • ?cardinality (xsd:string): Returns the cardinality of relationships between tables: optional, many, or required. • ?count (xsd:int): Returns the number of times the field appears in the source. • ?order (xsd:int): Returns the order in which the field was encountered.

Metadata Query Examples

This section includes sample metadata queries that run against different types of data sources.

- [List Database Schemas](#)
- [Explore a Database Schema](#)
- [Explore a Directory of SAS Files](#)
- [Explore an HTTP Endpoint](#)
- [Explore a Directory of CSV Files](#)

List Database Schemas

The query below sends a metadata query to a MySQL database to return a list of the schemas that are available:

```
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:DbSource ;
            s:url "jdbc:mysql://10.100.2.9:5555/?user=root&password=Mysql11@#" .

        ?metadata a s:MetadataSource ;
            s:from ?data ;

        ?catalogs [
            ?catalog (xsd:string) ;
            ?order (xsd:int) ;
        ] .
    }
}
ORDER BY ?catalog
```

The query returns the following results:

catalog	order
BANKTEST_DB	1
EMR	4
GOLFCLUB_DB	8
NORTHWIND	10
SPORTDB	13

SQLPOCKET_DB		14
WORDPRESS_DB		16
classicmodels		2
crm_national_patients		3
emrdbbig		5
emrdbsmall		6
emrnational_schema		7
mysql		9
optum		11
performance_schema		12
sys		15
16 rows		

Explore a Database Schema

Using the list of schemas that were returned in the example above ([List Database Schemas](#)), the query below returns metadata about the columns in one of the schemas. To narrow the results to a schema, the schema name (NORTHWIND) is added to the connection URL.

```

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:DbSource ;
              s:url "jdbc:mysql://10.100.2.9:5555/NORTHWIND?user=root&password=Mysql1@#" .

        ?metadata a s:MetadataSource ;
                  s:from ?data ;

        ?fields [
            ?model (xsd:string) ;
            ?field (xsd:string) ;

```

```

        ?datatype (owl:Thing) ;
    ] .
}
}
ORDER BY ?model

```

The query returns the following results:

model	field	datatype
-----	-----	-----
Alphabetical list of products	CategoryID	
http://www.w3.org/2001/XMLSchema#int		
Alphabetical list of products	Discontinued	
http://www.w3.org/2001/XMLSchema#boolean		
Alphabetical list of products	SupplierID	
http://www.w3.org/2001/XMLSchema#int		
Alphabetical list of products	UnitPrice	
http://www.w3.org/2001/XMLSchema#decimal		
Alphabetical list of products	ProductName	
http://www.w3.org/2001/XMLSchema#string		
Alphabetical list of products	QuantityPerUnit	
http://www.w3.org/2001/XMLSchema#string		
Alphabetical list of products	UnitsOnOrder	
http://www.w3.org/2001/XMLSchema#short		
Alphabetical list of products	CategoryName	
http://www.w3.org/2001/XMLSchema#string		
Alphabetical list of products	ProductID	
http://www.w3.org/2001/XMLSchema#int		
Alphabetical list of products	ReorderLevel	
http://www.w3.org/2001/XMLSchema#short		
Alphabetical list of products	UnitsInStock	
http://www.w3.org/2001/XMLSchema#short		
Categories	CategoryID	
http://www.w3.org/2001/XMLSchema#int		
Categories	Description	
http://www.w3.org/2001/XMLSchema#string		
Categories	Picture	
http://www.w3.org/2001/XMLSchema#base64Binary		
Categories	CategoryName	
http://www.w3.org/2001/XMLSchema#string		
Categories	categoryid	
Category Sales for 1997	CategoryName	

```

http://www.w3.org/2001/XMLSchema#string
Category Sales for 1997          | CategorySales          |
http://www.w3.org/2001/XMLSchema#double
Current Product List            | ProductName            |
http://www.w3.org/2001/XMLSchema#string
Current Product List            | ProductID              |
http://www.w3.org/2001/XMLSchema#int
...
201 rows

```

Explore a Directory of SAS Files

The query below explores a directory of SAS files to return the model, catalog (schema), field, data type, and cardinality information. The query also orders the results by model name, which is the file name for file sources of a data model does not exist.

```

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:FileSource ;
              s:url "/opt/shared-files/sas" .

        ?metadata a s:MetadataSource ;
                 s:from ?data ;

        ?fields [
            ?model (xsd:string) ;
            ?field (xsd:string) ;
            ?catalog (xsd:string) ;
            ?datatype (owl:Thing) ;

```

```

        ?cardinality (xsd:string) ;

    ] .

}

}

ORDER BY ?model

```

The query returns the following results:

model	field	catalog	datatype	cardinality
-----+-----+-----+-----+-----				
-				
demand	P1	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demand	P2	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demand	P3	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demand	Y	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demand	Q1	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demand	Q2	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demand	Q3	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demo	YEAR	les/sas	http://www.w3.org/2001/XMLSchema#long	REQUIRED
demo	QTR	les/sas	http://www.w3.org/2001/XMLSchema#long	REQUIRED
demo	GDP	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demo	PR	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demo	M1	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
demo	RS	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
airline	YEAR	les/sas	http://www.w3.org/2001/XMLSchema#long	REQUIRED
airline	Y	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
airline	W	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
airline	R	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
airline	L	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
airline	K	les/sas	http://www.w3.org/2001/XMLSchema#double	REQUIRED
cars	MPG	les/sas	http://www.w3.org/2001/XMLSchema#long	REQUIRED
cars	CYL	les/sas	http://www.w3.org/2001/XMLSchema#long	REQUIRED
...				
50 rows				

Explore an HTTP Endpoint

The query below explores the metadata for a sample HTTP source that compiles worldwide weather statistics.

```

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>

```

```

PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:HttpSource ;
              s:url "https://sampleEndpoint.com/forecast/30.374563,-97.975892" .

        ?metadata a s:MetadataSource ;
                  s:from ?data ;

        ?fields [
            ?model (xsd:string) ;
            ?field (xsd:string) ;
            ?datatype (owl:Thing) ;
            ?cardinality (xsd:string) ;
            ?order (xsd:int) ;
        ] .
    }
}
ORDER BY ?model ?order

```

The query returns the following results:

model	field	datatype
cardinality	order	
-----+-----	-----+-----	-----+-----
currently	time	http://www.w3.org/2001/XMLSchema#int
REQUIRED	6	
currently	summary	http://www.w3.org/2001/XMLSchema#string
REQUIRED	7	
currently	icon	http://www.w3.org/2001/XMLSchema#string
REQUIRED	8	
currently	nearestStormDistance	http://www.w3.org/2001/XMLSchema#int

REQUIRED		9		
currently		nearestStormBearing		http://www.w3.org/2001/XMLSchema#int
REQUIRED		10		
currently		precipIntensity		http://www.w3.org/2001/XMLSchema#int
REQUIRED		11		
currently		precipProbability		http://www.w3.org/2001/XMLSchema#int
REQUIRED		12		
currently		temperature		http://www.w3.org/2001/XMLSchema#float
REQUIRED		13		
currently		apparentTemperature		http://www.w3.org/2001/XMLSchema#float
REQUIRED		14		
currently		dewPoint		http://www.w3.org/2001/XMLSchema#float
REQUIRED		15		
currently		humidity		http://www.w3.org/2001/XMLSchema#float
REQUIRED		16		
currently		pressure		http://www.w3.org/2001/XMLSchema#float
REQUIRED		17		
currently		windSpeed		http://www.w3.org/2001/XMLSchema#float
REQUIRED		18		
currently		windGust		http://www.w3.org/2001/XMLSchema#float
REQUIRED		19		
currently		windBearing		http://www.w3.org/2001/XMLSchema#int
REQUIRED		20		
currently		cloudCover		http://www.w3.org/2001/XMLSchema#float
REQUIRED		21		
currently		uvIndex		http://www.w3.org/2001/XMLSchema#int
REQUIRED		22		
currently		visibility		http://www.w3.org/2001/XMLSchema#int
REQUIRED		23		
currently		ozone		http://www.w3.org/2001/XMLSchema#float
REQUIRED		24		
daily		summary		http://www.w3.org/2001/XMLSchema#string
REQUIRED		75		
daily		icon		http://www.w3.org/2001/XMLSchema#string
REQUIRED		76		
daily		data		
MANY		77		
data		time		http://www.w3.org/2001/XMLSchema#int
REQUIRED		29		
data		precipIntensity		http://www.w3.org/2001/XMLSchema#float
REQUIRED		30		
data		precipProbability		http://www.w3.org/2001/XMLSchema#float
REQUIRED		31		

```

data          | summary                                | http://www.w3.org/2001/XMLSchema#string |
OPTIONAL      |      32
...
81 rows

```

The following query retrieves the model, field, and data type metadata for the United States from the publicly available [Data API Covid Tracking Project](#).

```

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:   <http://www.w3.org/2002/07/owl#>
PREFIX anzo:    <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:    <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>

SELECT *
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:HttpSource ;
              s:url "https://covidtracking.com/api/v1/us/current.csv" .

        ?metadata a s:MetadataSource ;
                  s:from ?data ;

        ?fields [
            ?model (xsd:string) ;
            ?field (xsd:string) ;
            ?datatype (owl:Thing) ;
        ] .
    }
}

```

The query returns the following results:

model	field	datatype
us	date	http://www.w3.org/2001/XMLSchema#string

us	states	http://www.w3.org/2001/XMLSchema#string
us	positive	http://www.w3.org/2001/XMLSchema#string
us	negative	http://www.w3.org/2001/XMLSchema#string
us	pending	http://www.w3.org/2001/XMLSchema#string
us	hospitalizedCurrently	http://www.w3.org/2001/XMLSchema#string
us	hospitalizedCumulative	http://www.w3.org/2001/XMLSchema#string
us	inIcuCurrently	http://www.w3.org/2001/XMLSchema#string
us	inIcuCumulative	http://www.w3.org/2001/XMLSchema#string
us	onVentilatorCurrently	http://www.w3.org/2001/XMLSchema#string
us	onVentilatorCumulative	http://www.w3.org/2001/XMLSchema#string
us	recovered	http://www.w3.org/2001/XMLSchema#string
us	dateChecked	http://www.w3.org/2001/XMLSchema#string
us	death	http://www.w3.org/2001/XMLSchema#string
us	hospitalized	http://www.w3.org/2001/XMLSchema#string
us	lastModified	http://www.w3.org/2001/XMLSchema#string
us	total	http://www.w3.org/2001/XMLSchema#string
us	totalTestResults	http://www.w3.org/2001/XMLSchema#string
us	posNeg	http://www.w3.org/2001/XMLSchema#string
us	deathIncrease	http://www.w3.org/2001/XMLSchema#string
us	hospitalizedIncrease	http://www.w3.org/2001/XMLSchema#string
us	negativeIncrease	http://www.w3.org/2001/XMLSchema#string
us	positiveIncrease	http://www.w3.org/2001/XMLSchema#string
us	totalTestResultsIncrease	http://www.w3.org/2001/XMLSchema#string
us	hash	http://www.w3.org/2001/XMLSchema#string

25 rows

Explore a Directory of CSV Files

The query below explores a directory of CSV files to return the model, field, and data type. The query also orders the results by model name, which is the file name for file sources of a data model does not exist. In addition, the query includes `s:sampling true`, which means the GDI will scan the entire file or files before returning results.

```
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>
```

```

SELECT *
WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?metadata .

        ?data a s:FileSource ;
              s:url "/opt/shared-files/movie-csv" .

        ?metadata a s:MetadataSource ;
                  s:from ?data ;

        # Sample the whole file
        s:sampling true ;

        # Sample the first N records #
        # s:sampling 1000 ;

        ?fields [
            ?model (xsd:string) ;
            ?field (xsd:string) ;
            ?datatype (owl:Thing) ;
        ] .
    }
}
ORDER BY ?model

```

The query returns the following results:

model	field	datatype
-----	-----	-----
MovieActors1	MovieID	
http://www.w3.org/2001/XMLSchema#int		
MovieActors1	MovieTitle	
http://www.w3.org/2001/XMLSchema#string		
MovieActors1	ActorID	
http://www.w3.org/2001/XMLSchema#int		
MovieActors1	ActorName	
http://www.w3.org/2001/XMLSchema#string		
MovieActors2	MovieID	
http://www.w3.org/2001/XMLSchema#int		

MovieActors2		MovieTitle	
http://www.w3.org/2001/XMLSchema#string			
MovieActors2		ActorID	
http://www.w3.org/2001/XMLSchema#int			
MovieActors2		ActorName	
http://www.w3.org/2001/XMLSchema#string			
MovieActors2		ActorCategory	
http://www.w3.org/2001/XMLSchema#string			
MovieCategory		MovieID	
http://www.w3.org/2001/XMLSchema#int			
MovieCategory		MovieTitle	
http://www.w3.org/2001/XMLSchema#string			
MovieCategory		MoveCategoryID	
http://www.w3.org/2001/XMLSchema#int			
MovieCategory		MovieCategory	
http://www.w3.org/2001/XMLSchema#string			
MovieCinematographers		MovieID	
http://www.w3.org/2001/XMLSchema#int			
MovieCinematographers		MovieTitle	
http://www.w3.org/2001/XMLSchema#string			
MovieCinematographers		MovieCinematographerID	
http://www.w3.org/2001/XMLSchema#int			
MovieCinematographers		MovieCinematographerName	
http://www.w3.org/2001/XMLSchema#string			
MovieComposers		MovieID	
http://www.w3.org/2001/XMLSchema#int			
MovieComposers		MovieTitle	
http://www.w3.org/2001/XMLSchema#string			
MovieComposers		MovieComposerID	
http://www.w3.org/2001/XMLSchema#int			
MovieComposers		MovieComposerName	
http://www.w3.org/2001/XMLSchema#string			
MovieDirectors		MovieID	
http://www.w3.org/2001/XMLSchema#int			
MovieDirectors		MovieTitle	
http://www.w3.org/2001/XMLSchema#string			
...			

79 rows

The following example shows a query that returns metadata for an Elasticsearch source.

```
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX anzo:  <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:  <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:    <http://purl.org/dc/elements/1.1/>
PREFIX ex:    <http://example.org/ontologies/City#>
PREFIX es:    <http://elastic.co/search/>
PREFIX :      <http://example.org/cities/>

SELECT *
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        [] s:select ?_fields .

        ?data a es:ElasticSource ;
            es:url "http://localhost:9200/" ;
            es:index "account" ;
            ?account_number xsd:long ;
            ?age (xsd:long) ;
            ?balance (xsd:long) ;
            ?address (xsd:string) ;
            ?city (xsd:string) ;
            ?state (xsd:string) ;
            ?email (xsd:string) ;
            ?employer (xsd:string) ;
            ?firstname (xsd:string) ;
            ?lastname (xsd:string) ;
            ?gender (xsd:string) .

        ?_fields a s:MetadataSource ;
            s:from ?data ;
            ?fields [
                ?catalog () ;
                ?model () ;
                ?field () ;
                ?cardinality () ;
                ?datatype () ;
                ?type () ;
                ?object () ;
            ] .
    }
}

```

```
}  
ORDER BY ?catalog ?model ?field
```

For instructions on querying the instance data based on the data source metadata, see [Getting Started with GDI Queries](#).

Pagination Options

The GDI exposes paging models that enable you to access large amounts of data across a number of smaller requests. Paging is configured by including the **paging** property in a query and configuring a combination of the pagination options described below. The GDI supports keyset-based, page-based, cursor-based, and offset-based pagination. Paging is supported for all data source types.

- [Paging Syntax](#)
- [Paging Examples](#)

Paging Syntax

```
s:paging [  
  s:key (?variable) ;  
  s:page ?variable ;  
  s:cursor ?variable ;  
  s:offset ?variable ;  
  s:size int ;  
  s:limit ?variable ;  
] ;
```

Option	Type	Description
key	variable	Include this property if you want to configure keyset-based pagination where a key is specified to act as a delimiter of the page. The <code>s:key</code> value is a variable that is bound to an expression that defines how to delimit the data. It is usually calculated by an aggregate expression and/or filter that can be pushed to the source. The aggregate expression is typically MAX, but MIN can also be used to page through data in reverse order, such as when working with temporal data. See Key-Based Examples below for examples that configure paging using

Option	Type	Description
		the <code>s:key</code> property.
page	variable	Include this property if you want to configure page-based pagination where the set is divided into pages. The <code>s:page</code> property value is a variable that the GDI can use to track the current page across requests. See Page-Based Example below for an example that configures paging using the <code>s:page</code> property.
cursor	variable	Include this property if you want to configure cursor-based pagination. The <code>s:cursor</code> property value is a variable that is bound against the source to capture the "cursor" value. The GDI uses this value as input to the source to deliver the next page of data. See Cursor-Based Example below for an example that configures paging with the <code>s:cursor</code> property.
offset	variable	Include this property along with the limit property if you want to configure offset-based pagination. The <code>s:offset</code> property value is a variable that the GDI can use to track the current offset across requests. See Offset-Based Example below for an example that configures paging using the <code>s:offset</code> property.
size	int	This property can be included with any of the paging models to configure the maximum size of each page. For example, <code>s:size 5000</code> limits the page size to 5,000 rows.
limit	variable	This property can be included to define the variable that the GDI should use to push the page size back to the source.

Paging Examples

- [Key-Based Examples](#)
- [Page-Based Example](#)

- [Cursor-Based Example](#)
- [Offset-Based Example](#)

Key-Based Examples

The example SERVICE clause below pages data based on the `?LastID` key, which is calculated by finding the maximum value of `SalesOrderID` and binding it to `?LastID`. A FILTER is used to filter for data where the `SalesOrderID` is greater than `?LastID`.

```
SERVICE <http://cambridgesemantics.com/services/DataToolkit>
{
  BIND(MAX(?SalesOrderID) AS ?LastID)
  FILTER(?SalesOrderID > ?LastID)

  ?SalesOrderHeaderEnlarged a s:DbSource ;
    s:url "jdbc:sqlserver://..." ;
    s:table "Sales.SalesOrderHeaderEnlarged" ;
    s:paging [
      s:key (?LastID) ;
      s:size 5000 ;
    ] ;
    ?SalesOrderID (xsd:int) ;
    ?RevisionNumber (xsd:int) ;
    ?OrderDate ("OrderDate" xsd:dateTime) ;
    ?DueDate (xsd:dateTime) .
}
```

The SERVICE clause below shows an example where key-based paging is configured to page through temporal data in reverse order. The `s:limit` property is configured on the `s:HttpSource` to limit the overall number of results returned across all pages. This query retrieves at most 1000 records (`s:limit 1000`), 100 rows (`s:size 100`) at a time.

```
SERVICE <http://cambridgesemantics.com/services/DataToolkit>
{
  BIND(MIN(?Timestamp) AS ?LastTimestamp)

  ?api a s:HttpSource ;
    s:url "http://slack.com/api/messages/latest" ;
    s:parameter [ s:name "before" ; s:value ?LastTimestamp ] ;
    s:parameter [ s:name "limit" ; s:value ?limit ] ;
    s:limit 1000 ;
}
```

```

s:paging [
  s:key (?LastTimestamp) ;
  s:limit ?limit ;
  s:size 100 ;
] ;
?Message (xsd:string) ;
?Author (xsd:string) ;
?Timestamp (xsd:dateTime) .
}

```

Page-Based Example

The SERVICE clause below shows an example that uses the `s:page` property to configure page-based paging where the page size is 100 rows. This query retrieves at most 1000 records (`s:limit 1000`), 100 rows (`s:size 100`) at a time.

```

SERVICE <http://cambridgesemantics.com/services/DataToolkit>
{
  ?api a s:HttpSource ;
  s:url "http://slack.com/api/messages" ;
  s:parameter [ s:name "page" ; s:value ?page ] ;
  s:parameter [ s:name "size" ; s:value ?limit ] ;
  s:limit 1000 ;
  s:paging [
    s:page ?page ;
    s:limit ?limit ;
    s:size 100 ;
  ] ;
  ?Message (xsd:string) ;
  ?Author (xsd:string) ;
  ?Timestamp (xsd:dateTime) .
}

```

Cursor-Based Example

The SERVICE clause below shows an example that uses the `s:cursor` property to configure cursor-based paging.

```

SERVICE <http://cambridgesemantics.com/services/DataToolkit>
{
  ?api a s:HttpSource ;
  s:url "http://slack.com/api/messages" ;
  s:parameter [ s:name "cursor" ; s:value ?cursor ] ;
}

```

```

s:parameter [ s:name "limit" ; s:value ?limit ] ;
s:limit 1000 ;
s:paging [
  s:cursor ?cursor ;
  s:limit ?limit ;
  s:size 100 ;
] ;
?Message (xsd:string) ;
?Author (xsd:string) ;
?Timestamp (xsd:dateTime) ;
?cursor ("next_cursor" xsd:string) .
}

```

Offset-Based Example

The SERVICE clause below shows an example that uses the `s:offset` property to configure offset-based paging.

```

SERVICE <http://cambridgesemantics.com/services/DataToolkit>
{
  ?api a s:HttpSource ;
  s:url "http://slack.com/api/messages" ;
  s:parameter [ s:name "offset" ; s:value ?offset ] ;
  s:parameter [ s:name "limit" ; s:value ?limit ] ;
  s:limit 1000 ;
  s:paging [
    s:offset ?offset ;
    s:limit ?limit ;
    s:size 100 ;
  ] ;
  ?Message (xsd:string) ;
  ?Author (xsd:string) ;
  ?Timestamp (xsd:dateTime) .
}

```

Binding and Hierarchy Concepts

As part of the Graph Data Interface's (GDI) flexibility, there are multiple ways to express binding hierarchies in queries. This topic describes the options for expressing hierarchies.

- [Using Binding Trees and Selector Paths](#)
- [Unpacking JSON with Bindings and Arrays](#)
- [Returning Hierarchies as JSON Strings](#)

Using Binding Trees and Selector Paths

One way to express hierarchies in queries is to use brackets ([]) to group objects into binding trees. For example, the WHERE clause snippet below organizes mapping variable objects into an hourly/data hierarchy by nesting the `?data` patterns inside the `?hourly` [] tree:

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:HttpSource;
    s:url "https://sampleEndpoint.com/forecast/" ;
    ?latitude (xsd:double) ;
    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    ?hourly
  [
    ?data
  [
    ?time (xsd:long) ;
    ?summary (xsd:string) ;
    ?rainIntensity ("precipIntensity" xsd:double) ;
    ?rainProbability ("precipProbability" xsd:double) ;
    ?temperature (xsd:double) ;
    ?feelsLike ("apparentTemperature" xsd:double) ;
    ?humidity (xsd:double) ;
    ?pressure (xsd:double) ;
    ?windSpeed (xsd:double) ;
  ] ;
  ] .
  }
}
```

When constructing object binding trees, if you choose to introduce the hierarchy with a variable name that is not an exact match to the source label, include a **selector** property to list the value from the source. For example, in the WHERE clause snippet below, `s:selector` is included to select `eventHeader` in the source as `?event` in the query and `statLocation` as `?location`.

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource ;
    s:url "/mnt/data/json/part_1.json" ;
    ?event
  [
    s:selector "eventHeader" ;
    ?eventId (xsd:string) ;
    ?eventName (xsd:string) ;
    ?eventVersion (xsd:string) ;
    ?eventTime (xsd:dateTime) ;
  ] ;
    ?location
  [
    s:selector "statLocation" ;
    ?locationId (xsd:string) ;
    ?lineNo (xsd:int) ;
    ?statNo (xsd:int) ;
    ?statId (xsd:int) ;
  ] .
  }
}
```

As an alternative to grouping objects in binding trees, the **selector** property also supports using dot notation to specify paths. For example, the WHERE clause snippet below rewrites the first example query to express the same `hourly/data` hierarchy as a path in the `s:selector` value:

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:HttpSource;
    s:url "https://sampleEndpoint.com/forecast/" ;
    ?latitude (xsd:double) ;
  }
```

```

    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    s:selector: "hourly.data" ;
    ?time (xsd:long) ;
    ?summary (xsd:string) ;
    ?rainIntensity ("precipIntensity" xsd:double) ;
    ?rainProbability ("precipProbability" xsd:double) ;
    ?temperature (xsd:double) ;
    ?feelsLike ("apparentTemperature" xsd:double) ;
    ?humidity (xsd:double) ;
    ?pressure (xsd:double) ;
    ?windSpeed (xsd:double) .
  }
}

```

You can also include the `$` character to anchor the selector at the root of the file. For example, `s:selector "data"` captures all data elements anywhere in the file. But `s:selector "$.data"` captures only the data elements that are at the root of the hierarchy.

Unpacking JSON with Bindings and Arrays

In addition to object binding trees and selectors, the GDI offers additional syntax for reading or ingesting JSON sources with nested objects and arrays. For example, following the JSON sample file below is a query that captures each value in the arrays:

```

{
  "payload" :
  {
    "IBP_IndEvent_MSR" :
    {
      "unit" : "ms",
      "value" : [ 0, 1 ]
    },
    "IBP_IndEvent_RMF" :
    {
      "unit" : "-",
      "value" : [ 0.012, 1.398, 3.1415 ]
    }
  }
}

```

To read the JSON file above, the following query uses an object binding (`?values [ ]`) to drill down to the `value` arrays in the source. An `@` selector is specified in the `?value` variable binding (`?value ("@" xsd:double)`) to retrieve each of the array values. For an array of primitive values, the `@` selector captures each value in the array. If the source `value` was an array of objects, the `@` selector would retrieve a JSON representation for each object in the array. In addition to creating a new binding context for the primitive array values, the `?values` object binding also includes `?index ("!array::index")` to capture the index array with the primitive value.

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT *
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
    ?data a s:FileSource ;
    s:url "/mnt/data/json/array-index.json" ;
    s:selector "payload.*" ;
    ?unit (xsd:string) ;
    ?values [
      s:selector "value" ;
      ?value ("@" xsd:double) ;
      ?index ("!array::index") ;
    ] .
  }
}
```

The results of the query are shown below:

unit	value	index
ms	0	0
ms	1	1
-	0.012	0
-	1.398	1
-	3.1415	2

If you do not want to retrieve all of the values in an array, you can include the specific index number to retrieve instead of using the `@` symbol. In the variable binding, the index number is appended in brackets (`[ ]`) to the binding column name. For example, the following variable binding retrieves the second index value (the third value in the array) from a "projects" array: `?project ("projects [2] ")`. The next example uses the following JSON file:

```
{
  "field1" : "value1" ,
  "arrayfield" : [
    "arrayvalue1",
    "arrayvalue2"
  ]
}
```

To retrieve only the second value in the array, the following query appends the index value 1 to the array column name, `arrayfield`:

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
SELECT *
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
    ?json a s:FileSource ;
    s:url "/mnt/data/json/array-index-2.json" ;
    ?field1 (xsd:string) ;
    ?arrayval ("arrayfield[1]" xsd:string) .
  }
}
```

The results of the query are shown below:

```
field1    | arrayval
-----+-----
value1    | arrayvalue2
```

Returning Hierarchies as JSON Strings

When working with schema-less sources, you can also capture a tree of data as a JSON string. For example, the query snippet below targets an HTTP endpoint. In this case, the properties under the **hourly** class of data are unknown. So the query binds all of the data below hourly to the **?hourly** variable by including empty parentheses. As a result, the GDI returns a JSON string representation of all of the properties and instance data under hourly:

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:HttpSource;
    s:url "https://sampleEndpoint.com/forecast/" ;
```



```

        ?latitude (xsd:double) ;
        ?longitude (xsd:double) ;
        ?timezone (xsd:string) ;
        ?hourly () .
    }
}

```

For example, the results look like this:

latitude	longitude	timezone	hourly
30.374563	-97.975892	America/Chicago	{ "summary": "\"Humid and partly cloudy throughout the day.\"\"", "icon": "\"partly-cloudy-day\"", "data": [{ "time": "1595559600", "summary": "\"Clear\"", "icon": "\"clear-night\"", "precipIntensity": "0", "precipProbability": "0", "temperature": "88.39", "apparentTemperature": "91.72", "dewPoint": "67.42", "humidity": "0.5", "pressure": "1011.7", "windSpeed": "7.48", "windGust": "16.71", "windBearing": "109", "cloudCover": "0.06", "uvIndex": "0", "visibility": "10", "ozone": "285.2"}, { "time": "1595563200", "summary": "\"Clear\"", "icon": "\"clear-night\"", "precipIntensity": "2.0E-4", "precipProbability": "0.01", "precipType": "\"rain\"", "temperature": "86.69", "apparentTemperature": "90.1", "dewPoint": "67.84", "humidity": "0.54", "pressure": "1012", "windSpeed": "7.05", "windGust": "17.56", "windBearing": "110", "cloudCover": "0.12", "uvIndex": "0", "visibility": "10", "ozone": "284.9"}], ...

Similar to the example above, you can write a query that specifically captures some of the properties in a hierarchy and then returns the rest of the properties and their values as a JSON string representation. To do so, use "@" as the binding path. For example:

```

WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:HttpSource;
    s:url
    "https://api.darksky.net/forecast/bdbe3f638eb908c9b94919537dad5945/30.374563,-
    97.975892" ;
    ?latitude (xsd:double) ;
    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    ?hourly [
      s:selector "hourly.data" ;
      ?time (xsd:long) ;
      ?summary (xsd:string) ;
    ]
  }
}

```

```

        ?hourly_data ("@" ) ;
    ] .
}
}

```

Sample results are shown below:

latitude	longitude	timezone	time	summary	hourly_data
30.374563	-97.975892	America/Chicago	1595559600	Clear	 { "time": "1595559600", "summary": "\"Clear\"", "icon": "\"clear- night\"", "precipIntensity": "0", "precipProbability": "0", "temperature": "88.39", "apparentTemperature": "91.72", "dewPoint": "67.42", "humidity": "0.5", "pressure": "1011.7", "windSpeed": "7.48", "windGust": "16.71", "windBearing": "109", "cloudCover": "0.06", "uvIndex": "0", "visibility": "10", "ozone": "285.2" }
30.374563	-97.975892	America/Chicago	1595563200	Clear	 { "time": "1595563200", "summary": "\"Clear\"", "icon": "\"clear-night\"", "precipIntensity": "2.0E-4", "precipProbability": "0.01", "precipType": "\"rain\"", "temperature": "86.69", "apparentTemperature": "90.1", "dewPoint": "67.84", "humidity": "0.54", "pressure": "1012", "windSpeed": "7.05", "windGust": "17.56", "windBearing": "110", "cloudCover": "0.12", "uvIndex": "0", "visibility": "10", "ozone": "284.9" }
30.374563	-97.975892	America/Chicago	1595566800	Partly Cloudy	 { "time": "1595566800", "summary": "\"Partly Cloudy\"", "icon": "\"partly-cloudy-night\"", "precipIntensity": "3.0E-4", "precipProbability": "0.01", "precipType": "\"rain\"", "temperature": "85.63", "apparentTemperature": "89.21", "dewPoint": "68.33", "humidity": "0.56", "pressure": "1012.6", "windSpeed": "6.48", "windGust": "17.92", "windBearing": "110", "cloudCover": "0.34", "uvIndex": "0", "visibility": "10", "ozone": "284.5" } ...

Incremental Load Concepts

When loading data from a database or file-based data source with a Graph Data Interface (GDI) query, you can add a few statements to the query to load a portion of the data incrementally rather than all of the data at once. As data is added or changed in the source, new data can be ingested without having to reload all of the previously ingested data. Because incremental ingestion is configured as a filter in a SPARQL query, it is extremely flexible, allowing for various conditions to be defined for diverse data sources. When the data is ingested, the GDI evaluates the current state of the data and then loads only the data that meets the conditions defined in the query. This topic provides example incremental queries to get you started.

- [Incremental DbSource Example](#)
- [Incremental FileSource Example](#)

Incremental DbSource Example

The following query ingests data from a database. All of the values for the requested columns in the ORDER_DETAILS table will be loaded.

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
  GRAPH <http://anzograph.com/northwind> {
    ?s ?p ?o .
  }
}

WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

    ?data a s:DbSource ;
      s:url "jdbc:oracle:thin:@10.10.10.10:1111/XE" ;
      s:username "northwind" ;
      s:password "NORTHWIND123" ;
      s:schema "NORTHWIND" ;
      s:table "ORDER_DETAILS" ;
      ?database ("!") ;
```

```

    ?schema ("!") ;
    ?table ("!") ;
    ?OrderID (xsd:int) ;
    ?ProductID (xsd:int) ;
    ?UnitPrice (xsd:double) ;
    ?Quantity (xsd:short) ;
    ?Discount xsd:double .

    BIND( IRI("http://anzograph.com/orders/{ {?OrderID}}") AS ?resource)

    ?rdf a s:RdfGenerator, s:OntologyGenerator ;
    s:as (?s ?p ?o) ;
    s:ontology <http://anzograph.com/ontologies/northwind> ;
    s:base <http://anzograph.com/data> .
}

```

The query below adds statements that configure the same query to ingest data incrementally. It captures the maximum order ID as the incremental value. When the source is updated with records that increase the order ID, only the records with larger order IDs than the previous maximum value will be ingested when the query is run. In the query:

- A `?MaxID` variable is bound to the result of `MAX(?OrderID)`: `BIND (MAX(?OrderID) AS ?MaxID)`.
- The `?MaxID` variable is defined as the incremental value: `?MaxID a s:IncrementalValue`.
- A filter clause is added to create a condition that ingests only the records where the order ID is greater than the previously ingested maximum ID: `FILTER (?OrderID > ?MaxID)`.

```

PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
  GRAPH <http://anzograph.com/northwind> {
    ?s ?p ?o .
  }
}

```

```

}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

        ?MaxID a s:IncrementalValue .
        FILTER (?OrderID > ?MaxID)
        BIND (MAX(?OrderID) AS ?MaxID)

        ?data a s:DbSource ;
            s:url "jdbc:oracle:thin:@10.10.10.10:1111/XE" ;
            s:username "northwind" ;
            s:password "NORTHWIND123" ;
            s:schema "NORTHWIND" ;
            s:table "ORDER_DETAILS" ;
            ?database ("!") ;
            ?schema ("!") ;
            ?table ("!") ;
            ?OrderID (xsd:int) ;
            ?ProductID (xsd:int) ;
            ?UnitPrice (xsd:double) ;
            ?Quantity (xsd:short) ;
            ?Discount xsd:double .

        BIND(IRI("http://anzograph.com/orders/{{?OrderID}}") AS ?resource)

        ?rdf a s:RdfGenerator, s:OntologyGenerator ;
        s:as (?s ?p ?o) ;
        s:ontology <http://anzograph.com/ontologies/northwind> ;
        s:base <http://anzograph.com/data> .
    }
}

```

Incremental FileSource Example

The following query ingests data from all of the CSV files in the `/nfs/data/fmcsa` directory:

```

PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {

```

```

GRAPH <http://anzograph.com/fmcsa> {
    ?s ?p ?o .
}
}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

        ?data a s:FileSource ;
            s:model "fmcsa" ;
            s:url "/nfs/data/fmcsa" ;
            s:pattern "*.csv" .

        ?rdf a s:RdfGenerator , s:OntologyGenerator ;
            s:as (?s ?p ?o) ;
            s:ontology <http://anzograph.com/ontologies/fmcsa> ;
            s:base <http://anzograph.com/data/> .
    }
}

```

The query below adds statements that configure the same query to ingest data incrementally. It uses a "last modified" strategy to determine what files are new or modified and should be ingested the next time the query is run. In the query:

- The modified timestamp metadata on the files is captured with `?Modified ("!")`.
- The `?LastRun` variable is bound to the result of the `NOW()` function: `BIND (NOW() AS ?LastRun)`.
- A filter clause is added to check whether the modified timestamp is later than the timestamp from the last time the query was run: `FILTER (?Modified > ?LastRun)`.
- `?LastRun` is defined as the incremental value: `?LastRun a s:IncrementalValue`.

```

PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>

INSERT {
    GRAPH <http://anzograph.com/fmcsa> {
        ?s ?p ?o .
    }
}

```

```

}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

        ?LastRun a s:IncrementalValue .
        FILTER (?Modified > ?LastRun)
        BIND (NOW() AS ?LastRun)

        ?data a s:FileSource ;
            s:model "fmcsa" ;
            s:url "/nfs/data/fmcsa" ;
            s:pattern "*.csv" ;
            ?Modified ("!") .

        ?rdf a s:RdfGenerator , s:OntologyGenerator ;
            s:as (?s ?p ?o) ;
            s:ontology <http://anzograph.com/ontologies/fmcsa> ;
            s:base <http://anzograph.com/data/> .
    }
}

```

Options for Data Types, Data Connections, and Models

The topics in this section describe the options that are available across data source types for controlling the way that strings are coerced to other data types, the way label and URI values are generated in the data model, and the relationships that define connections across multiple sources.

In this section:

- [Data Type Formatting Options](#) 256
- [Model Normalization Options](#) 259
- [Data Linking Options](#) 268

Data Type Formatting Options

To give you control over the data types that are used when coercing strings to other types, the **formats** property can be included in GDI queries to define the desired types. In addition, formats can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. You can also use the formats property to suppress the conversion so that the generated values are typed the same way as the source.

Tip

The GDI takes locale into account when formatting the generated date and time values.

For sources that do not include data type specifications and natively treat values as strings, the GDI Generator automatically converts the values to the appropriate type. For example, if a CSV file includes the value "Feb-18-2022," the GDI parses the string to an `xsd:date` with the format "2022-02-18". A column with numbers is converted to an `xsd:int` type and a column with a decimal value is converted to `xsd:float`. The formats property usage is described below.

- [Formats Syntax](#)
- [Formats Examples](#)

Formats Syntax

```
s:formats [  
  s:strict boolean ; [  
    xsd:data_type "format"  
  | xsd:data_type boolean ;  
    [ ... ; ]  
  ]  
] ;
```

Option	Type	Description
strict	boolean	This property enables or disables the automatic type conversion feature. By default, strict is set to false (<code>s:strict false</code>). When strict is false or not set, any formats specified in <code>s:formats []</code> augment the GDI's built-in date and time formats. In addition, when strict is false or not set, you can selectively disable certain type conversions by including <code>xsd:data_type false</code>. For example, <code>xsd:dateTime false</code> disables the parsing of strings to <code>dateTime</code>. When strict is true (<code>s:strict true</code>), you can selectively enable the default conversions. The GDI performs only the conversions that you enable with <code>xsd:data_type true</code> or define in <code>xsd:data_type "format"</code>. Values that do not match any of the formats provided will be typed as <code>xsd:string</code>. If strict is true and no other data type rules are specified, the auto conversion logic is essentially disabled and the generated data will be represented the same way it is in the source.
xsd:data_type "format"	N/A	Include <code>xsd:data_type "format"</code> when you want to describe the formats of date and time values in the source. The GDI supports Java date and time format notation. For example, if dates in the source are formatted like "yyyy-MM-dd," include the statement <code>xsd:date "yyyy-MM-dd"</code>. If the source uses multiple formats for dates, e.g., 18-MAR-1978 and 03/18/1978, you can list multiple

Option	Type	Description
		formats for <code>xsd:date</code>, such as <code>xsd:date "dd-MMM-yyyy"</code>, <code>"MM/dd/yyyy"</code>. Note The GDI's default base year is 2000. If the source data has years with only two digits, such as <code>02-04-99</code>, the GDI prepends 20 to the digits. The value <code>02-04-99</code> is parsed to <code>02-04-2099</code>. To specify an alternate base year to use for two-digit values, you can include the notation <code>^nnnn</code> (e.g., <code>^1900</code>) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-yy^1900"</code> or <code>xsd:date "dd-MMM-yy^1990"</code>. When one of those values is specified, <code>02-04-99</code> is parsed to <code>02-04-1999</code>.
<code>xsd:data_type boolean</code>	N/A	When <code>strict</code> is <code>false</code> or not set, you can disable specific type conversions by listing data types and setting their values to <code>false</code>. For example, if you want the GDI to convert strings to integers or floats when possible but you want the dates in the source to be preserved as strings, you can include <code>xsd:date false</code> to disable the conversion of strings to dates. When <code>strict</code> is <code>true</code>, you can enable specific type conversions by listing data types and setting their values to <code>true</code>. For example, if you want the GDI to preserve the strings in the source except for when the string is a number, you can include <code>xsd:int true</code> to enable the conversion of strings to integers.

Formats Examples

The example below sets `strict` to `true` and forces the GDI to parse values only to the data types that are enabled with `true`. It also defines the format to look for when converting strings to `dateTime`:

```
s:formats [
  s:strict true ;
  xsd:int true ;
  xsd:dateTime true ;
  xsd:dateTime "yyyy-MM-dd-HH-mm-ss" ;
] ;
```

The example below does not set `strict`, so the default value of `false` is used. The data type definitions specify the formats of the values to parse as date, time, and dateTime values. The example also disables the conversion from string to long:

```
s:formats [
  xsd:date "MM/dd/yyyy", "MMM dd", "MMM dd yyyy" ;
  xsd:time "HH[:mm][:ss][ ]a" ;
  xsd:dateTime "M/d/yyyy HH:mm:ss a", "yyyy-MM-dd-HH-mm-ss" ;
  xsd:long false ;
] ;
```

Model Normalization Options

To give users control over the labels and URIs that are generated in the data model, the GDI offers several options for normalizing the class and property fields that are created from the specified data source(s). Normalization rules can be specified at the source level to normalize the data from each source independently, or they can be used at the RDF Generator level to apply global rules across all specified data sources.

Note

Normalization rules are applied only at the model level. The rules to do affect the instance data values that are ingested.

Including the **normalize** parameter is optional. If you include it, you can specify any combination of rules. See [Default Normalization Behavior](#) below for details about the Generator's default behavior when normalization rules are not specified in your query.

- [Default Normalization Behavior](#)
- [Normalize Syntax](#)

- [Normalize Examples](#)

Default Normalization Behavior

The GDI Generator normalizes data according to the following rules by default. If you do not include the `s:normalize` parameter in your query, these are the rules that are applied:

```
s:normalize [  
  s:all [  
    s:removePrefix true ;  
    s:removePartialPrefix false ;  
    s:allowWhiteSpace false ;  
    s:allowPunctuation false ;  
    s:allowSymbols false ;  
    s:separator " " ;  
    s:singularize false ;  
    s:casing s:UpperCamel ;  
    s:localNameSeparator "." ;  
  ]  
]
```

Normalize Syntax

```
s:normalize boolean | [  
  s:model | s:field | s:all  
  [  
    s:removeStart "string" ;  
    s:removeEnd "string" ;  
    s:removePrefix boolean ;  
    s:removePartialPrefix boolean ;  
    s:match [ s:pattern "regex" ; s:replace "regex" ] ;  
    s:disambiguationLevel int ;  
    s:ignore "string" ;  
    s:words "string" ;  
    s:preserve "string" ;  
    s:split "string" ;  
    s:allowWhiteSpace boolean ;  
    s:allowPunctuation boolean ;  
    s:allowSymbols boolean ;  
    s:singularize boolean ;  
    s:casing property ;  
    s:separator "string" ;  
    s:localNamePrefix "string" ;  
  ]  
]
```

```
s:localNameSeparator "string" ;
] ;
] ;
```

Property	Type	Description
boolean	N/A	Normalize is enabled by default for all GDI Generator queries. If you want to disable normalization, you can include <code>s:normalize false</code> . If normalization is disabled, the names in the source will be used verbatim both for labeling and in generating the local names for property and class URIs. However, when normalization is disabled, the labels in the data source are used verbatim. In addition, the Generator creates hard-to-read, URL-encoded local names for property and class URIs.
s:model s:field s:all	N/A	This property defines whether the specified normalization rules should be applied across the model or only to the classes or properties. The list below describes each option: • s:model: Applies the rules to the file/table/class names only. • s:fields: Applies the rules to the column/property/field names only. • s:all: (Default) Applies the rules to both the class and property names. This is the default value if not specified.
removeStart	string	If you want to remove text from the beginning of identifiers, include the removeStart rule to specify the string to remove. For example, <code>s:removeStart "temp_"</code> .

Property	Type	Description
removeEnd	string	If you want to remove text from the end of identifiers, include the removeEnd rule to specify the string to remove. For example, <code>s:removeEnd "NEW"</code> .
removePrefix	boolean	If there are property identifiers that share a prefix with the class, the RDF Generator automatically removes the shared prefix from the property name; the removePrefix rule is set to <code>true</code> by default. For example, if there is an <code>EMPLOYEE</code> class with an <code>EMPLOYEE_ID</code> column, the shared prefix "EMPLOYEE" is removed from the generated property so that it becomes "ID." If you do not want the Generator to remove prefixes, you can include <code>s:removePrefix false</code> .
removePartialPrefix	boolean	If there are property identifiers that share a partial prefix with the class, you can enable removePartialPrefix to remove the partial prefix from the property name. The removePartialPrefix rule is set to <code>false</code> by default. If you want the Generator to remove partial prefixes, you can include <code>s:removePrefix true</code> .
match	RDF list	This rule provides a way to use regular expressions (REGEX) to match a pattern against source identifiers and replace the matched text in the normalized name. The <code>s:pattern</code> property defines the Java REGEX pattern to match against, and <code>s:replace</code> defines the Java REGEX replacement pattern. As shown in the example below, the match rule can also be configured with an <code>rdf:List</code> of objects to perform match evaluation in a certain order:

Property	Type	Description
		<pre>s:match ([s:pattern "(.+)"GUID\$" ; s:replace "\$1" ;] [s:pattern "(.+)"ID\$" ; s:replace "\$1" ;])</pre>
disambiguationLevel	int	This rule specifies the number of levels to use to resolve ambiguities between similarly named elements in a hierarchical source. For example, an element named "Data" appears in two contexts: "Currently" and "Hourly." By default, the Generator retains all levels, meaning two classes are generated: "Currently Data" and "Hourly Data." If <code>s:disambiguationLevel</code> is set to 0, a single class named "Data" is generated and both the Currently and Hourly classes have a "Data" property. The <code>disambiguationLevel</code> value is also used to determine the number of hierarchy levels to use when encoding the local name of the generated URI.
ignore	string	This rule can be used to list identifiers that should be ignored. Properties and classes will not be generated for identifiers that match the specified string(s). The ignore rule is a multi-valued property. For simplicity, you can enter a list by separating words with a space, rather than quoting each term and separating them with a comma. For multi-word identifiers, use single quotes. For example, <code>s:ignore "sample example 'test column' old"</code>.
words	string	Since many sources do not encode word boundaries

Property	Type	Description
		very well, the words rule can be used to list the set of words that should be separate identifiers. This rule tells the Generator which words may be encountered. The words rule is a multi-valued property. For simplicity, you can enter a list by separating words with a space, rather than quoting each term and separating them with a comma. For multi-word identifiers, use single quotes. For example: <pre>s:words "activity 'patient complaint' medication observation patient signal specialty study" ;</pre>
preserve	string	This rule can be used to identify any words whose casing should be preserved in the input identifiers. For example, if casing is set to <code>lower</code> but you want preserve the original upper casing of certain words, you can specify the words to preserve. The preserve rule is a multi-valued property. For simplicity, you can enter a list by separating words with a space, rather than quoting each term and separating them with a comma. For multi-word identifiers, use single quotes. For example: <code>s:preserve "ABC 'Laundry List' TriG"</code>. The preserve rule is case-insensitive. You do not have to match the casing of the words to preserve.
split	string	This rule specifies the string that should be used to split source identifiers into individual terms. If neither <code>split</code> nor <code>words</code> is specified, input identifiers are split on casing changes and character class changes.
allowWhiteSpace	boolean	This rule specifies whether or not white space should

Property	Type	Description
		be preserved in identifiers after they have been split into individual terms. This rule is set to <code>false</code> by default, meaning white space is not preserved. You can specify <code>s:allowWhiteSpace true</code> to preserve spaces.
allowPunctuation	boolean	This rule specifies whether or not punctuation should be preserved in identifiers after they have been split into individual terms. This rule is set to <code>false</code> by default, meaning punctuation is not preserved. You can specify <code>s:allowPunctuation true</code> to preserve punctuation.
allowSymbols	boolean	This rule specifies whether or not symbols should be preserved in identifiers after they have been split into individual terms. This rule is set to <code>false</code> by default, meaning symbols are not preserved. You can specify <code>s:allowSymbols true</code> to preserve symbols.
singularize	boolean	This rule specifies whether or not to change any plural identifiers to singular. This rule is set to <code>false</code> by default, meaning plural identifiers are preserved. You can specify <code>s:singularize true</code> to change plural terms to the singular version of the term.
casing	object	This rule specifies how the generated labels should be cased. By default, the Generator outputs labels in upper camel case (<code>s:casing s:UpperCamel</code>). To use a different casing, specify any of the following properties: • default: This object preserves the casing from the source. Labels will not be converted.

Property	Type	Description
		• UPPER: This object converts all characters to uppercase. For example, "uppercase" becomes "UPPERCASE." • lower: This object converts all characters to lowercase. For example, "Lower Case" becomes "lower case". • UpperCamel: This is the default casing value and converts labels to upper camel case, where terms are concatenated and the first letter of each word is capitalized. For example, "upper camel case" becomes "UpperCamelCase." • lowerCamel: This object converts labels to lower camel case, where terms are concatenated and the first letter of the first word is lower case. The first letter of subsequent terms is capitalized. For example, "lower camel case" becomes "lowerCamelCase."
separator	string	This rule specifies the character or characters to use to separate terms in the generated label. The default separator is a space (<code>s:separator " "</code>).
localNamePrefix	string	This rule specifies a string to use as the prefix for local names when generating a URI.
localNameSeparator	string	This rule specifies the string to use for separating local

Property	Type	Description
		names when encoding hierarchies according to the specified disambiguationLevel. By default, <code>localNameSeparator</code> is a period (<code>s:localNameSeparator "."</code>). If <code>localNameSeparator</code> is empty, hierarchical context will not be encoded into the local name of any properties or child classes. The result would be an ontology where only the class or property name is used to determine the local name. For example, a property URI would look like <code>ont:employeeID</code> rather than <code>ont:Employee.employeeID</code>. The result could lead to "conflicts" in the generated ontology, but those "conflicts" may be desired as properties with same name are reused across the generated ontology.

Tip

You can also specify normalization rules at both the source and global level in the same query. If you include multi-valued rules (such as `ignore`, `words`, or `preserve`) at both levels, the Generator combines all values from both instances of the rule. If you specify single value rules at both levels and the values are conflicting, the Generator applies the value at the source level.

Normalize Examples

The example below uses the `normalize` property to normalize data at both the model and field level.

```
s:normalize [
  s:model [
    s:localNamePrefix "C_" ;
    s:localNameSeparator "_" ;
    s:match [ s:pattern "(.+)"Enlarged" ; s:replace "$1" ] ;
  ] ;
  s:field [
    s:localNamePrefix "P_" ;
```

```

s:localNameSeparator "_" ;
s:ignore "rowguid ModifiedDate" ;
s:match (
  [ s:pattern "(.+)"GUID$" ; s:replace "$1" ]
  [ s:pattern "(.+)"ID$" ; s:replace "$1" ]
) ;
] ;
] ;

```

Data Linking Options

When a data source does not define keys (such as a CSV or JSON source), the GDI provides properties that enable you to create a connected knowledge graph by defining relationships, resource templates (primary keys) and object properties (foreign keys), when you are loading data from multiple sources. The properties that are available are described below.

- [Data Linking Syntax](#)
- [Data Linking Examples](#)

Data Linking Syntax

```

s:key ("column_name") ;
s:reference [
  s:model "table_to_reference" ;
  s:using ("foreign_key_column")
]

```

Option	Type	Description
key	string	Include this property when you want to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . If none of the columns contain unique values, you can specify a combination of columns that would create a unique value. For example, <code>s:key ("FlightNumber", "TailNumber")</code> .
reference	RDF list	Include this property when you want to specify a foreign key column.

Option	Type	Description
		The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column in the source table. <pre>s:reference [s:model "table_to_reference" ; s:using ("foreign_key_column")]</pre> Tip You can also include an optional <code>key</code> property within the <code>s:reference</code> list that defines the key column in the target table and can be used as a way to expose additional metadata that helps inform the GDI how to name the object property. For example: <pre>s:reference [s:model "Employees" ; s:using ("EMPLOYEE_ID") ; s:key ("EMPLOYEE_ID")]</pre>

Data Linking Examples

For example, the query snippet below defines two data sources. The `s:model` property defines the table/class for each source, and the `s:key` defines the primary key for each table/class. The `s:reference` property for the "venue" table defines a foreign key relationship from `venue.EVENT_ID` to `event.EVENT_ID`.

```
?event a s:FileSource ;
  s:model "event" ;
  s:url "/opt/shared-files/csv/events.csv" ;
  s:key ("EVENT_ID") .

?venue a s:FileSource ;
  s:model "venue" ;
  s:url " /opt/shared-files/csv/venues.csv" ;
```

```
s:key ("VENUE_ID") ;  
s:reference [ s:model "event" ; s:using ("EVENT_ID") ] .
```

The following query for multiple file sources generates RDF and an ontology with resource templates and object properties. The query also includes global normalization rules for normalizing the data across all sources (see [Model Normalization Options](#) for information about normalization).

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>  
  
INSERT {  
  GRAPH <http://anzograph.com/tickets> {  
    ?s ?p ?o .  
  }  
}  
  
WHERE {  
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {  
  
    ?event a s:FileSource ;  
      s:model "event" ;  
      s:url "/opt/shared-files/csv/events.csv" ;  
      s:key ("EVENT_ID") .  
  
    ?listing a s:FileSource ;  
      s:model "listing" ;  
      s:url " /opt/shared-files/csv/listings.csv" ;  
      s:key ("LIST_ID") ;  
      s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] .  
  
    ?date a s:FileSource ;  
      s:model "date" ;  
      s:url "/opt/shared-files/csv/event_dates.csv" ;  
      s:key ("DATE_ID") ;  
      s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] .  
  
    ?venue a s:FileSource ;  
      s:model "venue" ;  
      s:url " /opt/shared-files/csv/venues.csv" ;  
      s:key ("VENUE_ID") ;  
      s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] .  
  
    ?sale a s:FileSource ;  
      s:model "sale" ;  
      s:url " /opt/shared-files/csv/sales.csv" ;
```

```

s:key ("SALE_ID") ;
s:reference [ s:model "event" ; s:using ("EVENT_ID") ; s:key ("EVENT_ID") ] ;
s:reference [ s:model "listing" ; s:using ("LIST_ID") ; s:key ("LIST_ID") ] .

?rdf a s:RdfGenerator, s:OntologyGenerator ;
s:as (?s ?p ?o) ;
s:ontology <http://anzograph.com/tickets> ;
s:base <http://anzograph.com/data> ;
s:normalize [
    s:all [
        s:casing s:UPPER ;
        s:localNameSeparator "_" ;
    ] ;
] .
}
}

```

Advanced Usage by Data Source Type

The topics in this section provide more advanced GDI usage information by including descriptions for all of the query options for each type of supported data source.

Tip

Rather than manually writing complex queries, you can use the GDI to automatically generate graphs and ontologies by including a few key statements in a relatively simple query. For information, see [Generating a Knowledge Graph](#).

In this section:

Query a Database Source

Query an HTTP Source

Query an Elasticsearch Source

Query a File Source

272

286

307

331

Query a Database Source

This topic provides details about the structure to use when writing GDI queries to read or ingest data from database data sources. It also includes example queries that may be useful as a starting point for writing your own GDI queries.

- [Supported Databases](#)
- [Query Syntax](#)
- [Query Examples](#)

Supported Databases

The GDI supports querying any database through a JDBC connection. Graph Lakehouse installations include JDBC drivers for the following databases:

- Databricks
- H2

- IBM DB2
- Microsoft SQL Server
- MariaDB
- Oracle
- PostgreSQL
- SAP Sybase (jTDS)
- Snowflake

Query Syntax

The following query syntax shows the structure of a GDI query for database sources. The clauses, patterns, and placeholders that are links are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
{
  [ GRAPH <target_graph> { ]
    triple_patterns
  [ } ]
}
[ FROM Clause ]
WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView> (<target_graph>)
```

```

{
  ?data a s:DbSource ;
    s:url "string" ;
    s:username "string" ;
    s:password "string" ;
    [ s:token "string" ; ]
    [ s:driver "string" ; ]
    [ s:property [ s:name "string" ; s:value "string" ] ; ]
    [ s:timeout int ; ]
    [ s:maxConnections int ; ]
    [ s:batching boolean | int ; ]
    [ s:concurrency int | [ list_of_properties ] ; ]
    [ s:rate int | "string" ; ]
    [ s:partitionBy "string" | ?variable | s:auto ; ]
    [ s:locale "string" ; ]
    [ s:sampling int ; ]
    [ s:selector "string" | [ list ] ; ]
    [ s:model "string" ; ]
    [ s:key ("string") ; ]
    [ s:reference [ s:model "string" ; s:using ("string") ]
    [ s:formats [ datatype_formatting_options ] ; ]
    [ s:normalize boolean | [ normalization_rules ] ; ]
    [ s:query "string" ; ]
    [ s:database "string" ; ]
    [ s:schema "string" ; ]
    [ s:table "string" ; ]
    [ s:count ?variable ; ]
    [ s:offset int ; ]
    [ s:orderBy "string" | ?variable ; ]
    [ s:limit int ; ]
    # Mapping variables
    ?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
    ... ;
    .
    # Additional clauses such as BIND, VALUES, FILTER
}
}

```

Note

For readability, the parameters below exclude the base URI

`<http://cambridgesemantics.com/ontologies/DataToolkit#>` as well as the `s:` prefix. As shown in the examples, however, the `s:` prefix or full property URI does need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause defines the type of SPARQL query to run and the set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN]</code> <code><http://cambridgesemantics.com/services/DataToolkit></code> to invoke the GDI service when you are running a SELECT, INSERT, or CONSTRUCT query that is not creating a view. When creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.

Option	Type	Description
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: SERVICE <pre><http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>).</pre> Using the DataToolkitView call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not necessarily returned.
url	string	This property specifies the URL to use to access the database. Important For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example, the triple patterns below reference a Query Context and add a JDBC driver level connection property: <pre>?data a s:DbSource ; s:url "{{@Somedb.url}}" ; s:username "{{@Somedb.user}}" ; s:password "{{@Somedb.password}}" ; s:property [s:name "access" ; s:value "all"]</pre>
username	string	This property lists the user name to use for the connection to the database.

Option	Type	Description
		Tip If you want to group the username and password properties, you can wrap them with <code>s:credentials []</code>. For example: <pre>s:credentials [s:username "username" ; s:password "password" ;] ;</pre>
password	string	This property lists the password for the given username.
token	string	For connections that require a bearer token, this property can be included to specify the token.
driver	string	This property can be included to specify the JDBC driver to use.
property	RDF list	This property can be included to list any JDBC driver-specific connection properties. To incorporate <code>property</code>, use the following syntax: <pre>s:property [s:name "custom_driver_property_name" ; s:value "custom_value"]</pre>
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
maxConnection	int	This property can be used to set a limit on the maximum number

Option	Type	Description
s		of active connections to the source. For example, <code>s:maxConnections 16</code> sets the limit to 16 connections. When not specified, the default value is 10.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code>. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of

Option	Type	Description
		requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code>, the GDI would use 3 executors with a rate delay of 750ms between requests.
partitionBy	string, variable, object	The GDI attempts to partition queries automatically across the available cores (slices) in Graph Lakehouse. To determine how to partition the query, the GDI uses metadata from the source database. It looks for any column in an index, preferring the primary key column if it is interpolable. However, it only considers the first column in any index on the table. After determining the partition column, the GDI does a MIN/MAX on the column as well as a basic sizing query. To specify which column or columns the GDI should partition on, you can include the <code>partitionBy</code> property in the query. The property supports a list of source field names, bound variables, or the object <code>s:auto</code>, which forces the

Option	Type	Description
		GDI to partition the data when the source does not define partitioning metadata.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the SalesOrderHeader table in the Sales schema. As an alternative to including the selector property for identifying the target data, you could use the database , schema , and/or table properties.
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The

Option	Type	Description
		reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or <code>dateTime</code> values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
query	string	If you want to access the source data by running an SQL query, you can include this property to specify the query string to run. The language does not have to be SQL if the source supports another language. However, some GDI features where the query is dynamically altered may not work with a non-SQL language. Including <code>{{?variable}}</code> substitutions is supported within <code>s:query</code> strings. Note If you include <code>s:query</code> without also specifying table or partitionBy, the GDI may not partition the query and query execution may be less performant than if the partition

Option	Type	Description
		column was specified. When using <code>s:query</code>, specifying the table in <code>s:table</code> and the column to partition the table on in <code>s:partitionBy</code> is a good practice, especially when querying large tables.
database	string	This property can be used to specify the database to target in the source if the database is not listed in the <code>s:url</code> or <code>s:selector</code> strings.
schema	string	This property can be included to specify the target schema to query. If you include <code>s:schema "schema_name"</code> without specifying <code>s:table</code> (described below) or <code>s:query</code> , all tables in the schema are queried.
table	string	This property can be included to specify the target table or tables for the query.
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
orderBy	string, variable, list	You can include this property to order the result set by a field name, a bound variable, or a list of names or bound variables.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.

Option	Type	Description
mapping_variable	variable	The mapping variables, in <code>?variable (["binding"] [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Note The parentheses around the binding, data type, and format specifications are not required but are included in this document for readability.
binding	string	The binding is a literal value that binds a <code>?variable</code> to a source column. If you specify a <code>?variable</code> that matches the source column name, then that variable name is the data selector and it is not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column that is not already identified by the selector, database, schema, table, or query properties, then the binding is needed to map the new variable to that source column. For example, <code>?subject ("dbo.FILM.SUBJECT")</code> binds the <code>?subject</code> variable by navigating to the SUBJECT column in the FILM table in the dbo schema. Note Database, schema, and table names in bindings are parsed according to the specific rules for that database type. You do not need to escape characters in database names. However, database names with characters that do

Option	Type	Description
		not match <code>(_ A-Z a-z)(_ A-Z a-z 0-9)*</code> should be quoted, such as <pre>("Adventure.Works".Sales."Daily.Totals").</pre>
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double, xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration, xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI</pre>
datetime_ format	string	This option is used to specify the format to use for date and time data types. The GDI supports Java date and time formats. Specify days as "d," months as "M," and years as "y." For the time, specify "H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMyy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation <code>^nnnn</code> (e.g., <code>^1900</code>) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-</code>

Option	Type	Description
		yy^1900" or xsd:date "dd-MMM-yy^1990". When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

Query Examples

The example below selects data from the AdventureWorks2012 database. The `s:selector` property is used to specify the table (`salesOrderHeader` in the `Sales` schema) to target.

```

PREFIX s:    <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

SELECT (COUNT(*) as ?count)
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?SalesOrderHeader a s:DbSource ;
      s:url "{{@Somedb.url}}" ;
      s:username "{{@Somedb.user}}" ;
      s:password "{{@Somedb.password}}" ;
      s:selector "Sales.SalesOrderHeader" ;
      ?SalesOrderID (xsd:int) ;
      ?RevisionNumber (xsd:int) ;
      ?OrderDate (xsd:dateTime) ;
      ?DueDate (xsd:dateTime) ;
      ?TerritoryID (xsd:int) ;
      ?TotalDue (xsd:decimal) .
    FILTER(?TerritoryID IN (1, 2, 3))
    FILTER(?TotalDue < 11.0 || ?TotalDue > 250)
  }
}

```

The example below ingests data from a database. To define the data to target, the query includes the `s:query` property to run an SQL query. The `s:table` and `partitionBy` properties are also included to aid the GDI in partitioning the query.

```

PREFIX s:    <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX ont:
<http://cambridgesemantics.com/Layer/2f1e926b130a402db6fc10fa54199d49/Model#>

INSERT {
  GRAPH <http://anzograph.com/emr> {
    ?resource a ont:EmrPatient ;
    ont:EmrPatient.patientid ?PATIENTID ;
    ont:EmrPatient.gender ?GENDER ;
    ont:EmrPatient.language ?LANGUAGE ;
    ont:EmrPatient.patientfirstdocactivitydate ?PATIENTFIRSTDOCACTIVITYDATE .
  }
}
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
    ?data a s:DbSource ;
    s:url "{{@Somedb.url}}" ;
    s:username "{{@Somedb.user}}" ;
    s:password "{{@Somedb.password}}" ;
    s:query "select * from emrdbsmall.emr_patient where emr_patient.PATIENTID < 500"
;
    s:partitionBy "PATIENTID" ;
    s:table "emrdbsmall.emr_patient" ;
    ?PATIENTID (xsd:int) ;
    ?GENDER (xsd:string) ;
    ?LANGUAGE (xsd:string) ;
    ?PATIENTFIRSTDOCACTIVITYDATE (xsd:dateTime "M/d/yyyy HH:mm:ss") .

    BIND(IRI("http://cambridgesemantics.com/Layer/2f1e926b130a402db6fc10fa54199d49/
{{?PATIENTID}}") AS ?resource)
  }
}

```

Query an HTTP Source

This topic provides details about the structure to use when writing GDI queries to read or ingest data from HTTP data sources. It also includes example queries that may be useful as a starting point for writing your own GDI queries.

- [Query Syntax](#)
- [Mapping the Content Property to JSON](#)
- [Query Examples](#)

Query Syntax

The following query syntax shows the structure of a GDI query for HTTP sources. The clauses, patterns, and placeholders that are links are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
{
  [ GRAPH <target_graph> { ]
    triple_patterns
  [ ] ]
}
[ FROM Clause ]
WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView> (<target_graph>)

  {
    ?data a s:HttpSource ;
    s:url "string" ;
    [ s:authorization [
      a s:BearerToken ; s:token "string" ;
      | a s:AWSSignature ; s:accessKey "string" ; s:region "string" ;
      s:secretKey "string" ; s:serviceName "string" ;
    ]
  ]
  }
```

```

        s:sessionToken "string" ;
    | a s:BasicAuth ; s:username "string" ; s:password "string" ;
] ; ]
[ s:trust "string" ; ]
[ s:proxy "string" | [ s:host "string" ; s:port int ] ]
[ s:header [ s:name: "string" ; s:value "string" ] ; ]
[ s:mimetype "string" ; ]
[ s:contentType "string" ; ]
[ s:content ""string"" ; ]
[ s:parameter [ s:name "string" ; s:value "string" ] ; ]
[ s:method "string" ; ]
[ s:encoding "string" ; ]
[ s:form [ s:name: "string" ; s:value "string" ] ; ]
[ s:format [ source_format_options ; ] ; ]
[ s:timeout int ; ]
[ s:batching boolean | int ; ]
[ s:concurrency int | [ list_of_properties ] ; ]
[ s:rate int | "string" ; ]
[ s:partitionBy "string" | ?variable ; ]
[ s:locale "string" ; ]
[ s:sampling int ; ]
[ s:selector "string" | [ list ] ; ]
[ s:model "string" ; ]
[ s:key ("string") ; ]
[ s:reference [ s:model "string" ; s:using ("string") ]
[ s:formats [ datatype_formatting_options ] ; ]
[ s:normalize boolean | [ normalization_rules ] ; ]
[ s:count ?variable ; ]
[ s:offset int ; ]
[ s:orderBy "string" | ?variable ; ]
[ s:limit int ; ]
# Mapping variables
?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
... ;
.
# Additional clauses such as BIND, VALUES, FILTER
}
}

```


Note

For readability, the parameters below exclude the base URI

`<http://cambridgesemantics.com/ontologies/DataToolkit#>` as well as the `s:` prefix. As shown in the examples, however, the `s:` prefix or full property URI does need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause defines the type of SPARQL query to run and the set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN] <http://cambridgesemantics.com/services/DataToolkit></code> to invoke the GDI service when you are running a SELECT, INSERT, or CONSTRUCT query that is not creating a view. When creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.

Option	Type	Description
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: <code>SERVICE</code> <pre><http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>).</pre> Using the <code>DataToolkitView</code> call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not necessarily returned.
url	string	This property specifies the URL to use to access the source. Query binding variables can be inserted into the url string by surrounding the variable name with double curly braces. For example, "<code>{{?name}}</code>". Important For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example: <pre>?data a s:HttpSource ; s:url "{{@SomeHTTPSource.url}}" ; s:authorization [a s:BasicAuth ; s:username "{{@SomeHTTPSource.user}}" ; s:password "{{@SomeHTTPSource.password}}"]</pre>

Option	Type	Description
authorization	RDF list	This property specifies the type of authorization to use and the values for authentication. The options are BearerToken, AWSSignature, or BasicAuth. <pre>s:authorization [a s:BearerToken s:AWSSignature s:BasicAuth]</pre>
BearerToken	string	Specify this property when a bearer token is used for authentication, and include the token property. <pre>s:authorization [a s:BearerToken ; s:token "string"]</pre>
AWSSignature	RDF list	For authorization to AWS service endpoints, specify this property and include the appropriate authentication properties from the list below: • accessKey: Include this property to specify the AWS access key. • region: Include this property to specify the AWS region. • secretKey: Include this property to specify the AWS secret key. • serviceName: Include this property to specify the AWS service name. • sessionToken: Include this property to specify the AWS session token. <pre>s:authorization [a s:AWSSignature ; s:accessKey "string" ; s:region "string" ; s:secretKey "string" ;</pre>

Option	Type	Description
		<pre>s:serviceName "string" ; s:sessionToken "string" ;]</pre>
BasicAuth	RDF list	Specify this property when basic authentication is used, and include the username and password properties. <pre>s:authorization [a s:BasicAuth ; s:username "string" ; s:password "string" ;]</pre>
trust	string	Include this property to set the level of trust for the source's SSL certificate. The value can be either "system" or "all".
proxy	string or RDF list	Include this property to specify proxy information if a proxy is used. The value can be a string, such as <code>s:proxy "host_url:port_number"</code>, or an RDF list that includes <code>host</code> and <code>port</code> properties, such as <code>s:proxy [s:host "host_url" ; s:port port_number]</code>.
header	RDF list	You can use this property to specify name-value pairs to include as headers in the request. For example: <pre>s:header [s:name "Accept" ; s:value "application/json"]</pre> If you are creating a view, you can include variables in the <code>s:header</code> list. When another query is run against a view with variables, that query can map the variables through the view by including predicates in the <code>CONSTRUCT</code> clause.

Option	Type	Description
mimetype	string	You can include this property to specify the MIME type of the source. For example, <code>s:mimetype "text/html"</code> .
contentType	string	Include this property to specify the content type of the body of the request. For example, <code>s:contentType "application/sparql-query"</code> or <code>s:contentType "application/json"</code> .
content	string or RDF list	This property can be included to send content to the source in the body of the request. For example, <code>content</code> can be a SPARQL query, JSON arrays, or a list of key-value pairs. Content can also be configured with an inline object (blank node) that gets translated to JSON. For more information, see Mapping the Content Property to JSON below.
parameter	RDF list	You can include this property to list any URL parameters as name-value pairs. For example, the <code>s:parameter</code> property below adds <code>format</code> to return results in CSV format and the <code>named-graph-uri</code> parameter to target a specific layer in a graphmart. <pre>s:parameter [s:name "format" ; s:value "csv"] ,</pre> <pre>[s:name "named-graph-uri" ; s:value "http://cambridgesemantics.com/Layer/d541..."]</pre> If you are creating a view, you can include variables in the <code>s:parameter</code> list. When another query is run against a view with variables, that query can map the variables through the view by including predicates in the CONSTRUCT clause.
method	string	You can include this property to specify the HTTP method. For

Option	Type	Description
		example, <code>s:method "GET"</code> or <code>s:method "POST"</code> .
encoding	string	When targeting a file, you can include this property to specify the character encoding used by the file. The default value is <code>s:encoding "utf8"</code> .
form	RDF list	To send data to the HTTP endpoint, you can use this property to post the data. Form is a list of name-value pairs. When including <code>s:form</code> , you must also include <code>s:contentType "multipart/form-data"</code> . The GDI sends the form object as an <code>application/x-www-form-urlencoded</code> string that contains the specified parameters. See Query an HTTP Source below for sample usage.
format	RDF list	If the data is file-based, you can include the <code>format</code> property to add parameters that describe the source. See File Source Format Options for details about the supported parameters.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as

Option	Type	Description
		<code>s:concurrency</code> 8. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate.

Option	Type	Description
		Given the example of <code>s:rate "90/minute"</code> , the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code> , the GDI would use 3 executors with a rate delay of 750ms between requests.
partitionBy	string, variable, list	The GDI attempts to partition queries automatically across the available cores (slices) in Graph Lakehouse. To determine how to partition the query, the GDI uses metadata from the source. It looks for any column in an index, preferring the primary key column if it is interpolable. However, it only considers the first column in any index on the table. After determining the partition column, the GDI does a MIN/MAX on the column as well as a basic sizing query. To specify which column or columns the GDI should partition on, you can include the <code>partitionBy</code> property in the query. The property supports a list of source field names, bound variables, or the object <code>s:auto</code> , which forces the GDI to partition the data when the source does not define partitioning metadata.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the <code>SalesOrderHeader</code> table in the <code>Sales</code> schema. For more information about binding components and the selector property, see Using Binding Trees and Selector Paths .

Option	Type	Description
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .

Option	Type	Description
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
orderBy	string, variable, list	You can include this property to order the result set by a field name, a bound variable, or a list of names or bound variables.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
mapping_variable	variable	The mapping variables, in <code>?mapping_variable (["binding"] [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Note The parentheses around the binding, data type, and format specifications are not required but are included in this document for readability.
binding	string	The <code>binding</code> is a literal value that binds a ?mapping_variable to a source column. If you specify a <code>?variable</code> that matches the source column name, then that variable name is the data selector and it is

Option	Type	Description
		not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column, then the binding is needed to map the new variable to that source column. For example for CSV, the following pattern simply binds the source column AIRLINE to the lowercase variable ?airline: <pre>?airline ("AIRLINE").</pre> Note For FileSource, periods (.), forward slashes (/), and brackets ([]) are parsed as path notation. Therefore, if a source column name includes any of those characters they must be escaped in the binding. Use two backslashes (\\) as an escape character. For example, if a column name is average/day, the variable and binding pattern could be written as ?averagePerDay ("average\\/day").
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double, xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration, xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI</pre>
datetime_ format	string	This option is used to specify the format to use for date and time data types. The GDI supports Java date and time formats. Specify days as "d," months as "M," and years as "y." For the time, specify

Option	Type	Description
		"H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMMyy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation ^nnnn (e.g., ^1900) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-yy^1900"</code> or <code>xsd:date "dd-MMM-yy^1990"</code>. When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

Mapping the Content Property to JSON

The `s:content` property can be configured with an inline object (blank node) that gets translated to JSON in the request body. This mapping allows for creation of embedded objects and arrays as well as a mechanism for iterating over all available input so that HTTP endpoints that support batching can be used more effectively.

Using Blank Nodes

Blank nodes are used to create an object in the output JSON. The local name of any predicate used within `content` becomes a key in the generated JSON object. Blank nodes can be embedded within each other, allowing the hierarchical nature of JSON to be represented. For example:

```
s:content [ ex:firstName "Mary" ; ex:lastName "Barry" ] ;
```

Or

```
s:content [ ex:person [ ex:firstName "Mary" ] ] ;
```

Using Variables

Variables can be also used in the object position to construct a request from input at runtime. For example:

```
s:content [ ex:firstName ?firstName ; ex:lastName ?lastName ] ;
```

The values for the variables can come from a TOPDOWN variable, a VALUES clause in the SERVICE block, or another data source. Any unbound variables in the input will not be added to the generated JSON object.

Using RDF Lists

An RDF list can also be used to create an array in the output JSON. For example:

```
s:content [ ex:allKnownNames ( ?firstName ?lastName ?nickName ) ]
```

An RDF list can also be embedded inside another list to create an array in the output JSON and populate it with items evaluated against a repeating pattern across all available input rows for a slice. That pattern can be a variable, which generates an array of primitive values, or a blank node, which generates an array of mapped JSON objects. For example:

```
s:content [ ex:documents ((?id)) ] ;
```

Or

```
s:content [ ex:documents ([ ex:id ?id ; ex:title ?title ]) ] ;
```

Example

The following example query demonstrates the use of `s:content` to generate JSON. The query also includes the `s:concurrency` property to restrict execution to a single slice. Without limiting execution when there are a small number of inputs (as in the VALUES clause), each input row gets executed on its own. As the inputs increase, each slice operates over a larger number of inputs until the default `s:batching 5000` is applied.

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX api: <http://contoso.com/api/>

SELECT *
```

```

WHERE {
  SERVICE TOPDOWN <http://cambridgesemantics.com/services/DataToolkit>
  {
    VALUES (?firstName ?lastName ?dob ?email)
    {
      ("Gray" "Hay" "1978-03-18"^^xsd:date "gray@abc.com")
      ("Ana" "Bana" "1974-10-20"^^xsd:date "ana@abc.com")
      ("George" "Forge" "1975-08-13"^^xsd:date "george@abc.com")
      ("Miles" "Giles" "1977-04-12"^^xsd:date "miles@abc.com")
    }

    ?data a s:HttpSource ;
    s:url "https://postman-echo.com/post" ;
    s:header [ s:name "Accept" ; s:value "application/json" ] ;
    s:concurrency 1 ;
    s:content
      ([[
        api:dateOfBirth ?dob ;
        api:email ?email ;
        api:year 2020 ;
        api:person [ api:firstName ?firstName ; api:lastName ?lastName ] ;
      ]]) ;
    s:selector "data" ;
    ?firstName ("person.firstName" xsd:string) ;
    ?lastName ("person.lastName" xsd:string) ;
    ?dob ("dateOfBirth" xsd:date) ;
    ?email ("email" xsd:string) ;
    ?year ("year" xsd:int) .
  }
}

```

The content portion of the request that the query generates is shown below:

```

[[{
  "firstName": "Gray" ,
  "lastName": "Hay" ,
  "dateOfBirth": "1978-03-18" ,
  "email": "gray@abc.com" ,
  "year": 2020
},
{
  "firstName": "Ana" ,
  "lastName": "Bana" ,

```

```

    "dateOfBirth": "1974-10-20" ,
    "email": "ana@abc.com" ,
    "year": 2020
  },
  {
    ...
  }
}]

```

Query Examples

- [Topdown Query with URL Parameters](#)
- [Generator Query against a SPARQL Endpoint](#)
- [API Queries](#)

Topdown Query with URL Parameters

The query below reads data from a sample HTTP source that compiles worldwide weather statistics. The source has several models available for retrieving data that is current, daily, historical, etc. To target current data, the query includes `s:selector "currently"`. In addition, the query demonstrates the use of the "topdown" functionality, where the query sends values to the source to narrow the results. The `VALUES` clause specifies the latitude and longitude values for the cities to return data for. In addition, since this sample source requires parameters to be specified in the connection URL, the `s:url` value includes `?lat` and `?long` as parameters as part of the value.

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.org/ontologies/City#>

SELECT
    ?city ?state ?temp ?rainChance
    ?humidity ?pressure ?windSpeed
WHERE
{
    SERVICE TOPDOWN <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a s:HttpSource ;
        s:url "https://sampleEndpoint.com/forecast/{{?lat}},{{?long}}" ;
        s:selector "currently" ;
    }
}

```

```

    ?lat ("latitude") ;
    ?long ("longitude") ;
    ?temp ("temperature") ;
    ?rainChance ( "precipProbability" ) ;
    ?humidity () ;
    ?pressure () ;
    ?windSpeed () .
}
VALUES( ?city ?state ?lat ?long )
{
    ( "Lakeway" "TX" 30.374563 -97.975892 )
    ( "Boston" "MA" 42.358043 -71.060415 )
    ( "Seattle" "WA" 47.590720 -122.307053 )
    ( "Chicago" "IL" 41.837741 -87.823296 )
    ( "Hilo" "HI" 19.702040 -155.090312 )
}
}
ORDER BY ?city

```

Generator Query against a SPARQL Endpoint

The example below is a GDI Generator query that retrieves data from a remote SPARQL endpoint.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

INSERT {
    GRAPH <http://anzograph.com/something> {
        ?s ?p ?o }
}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a s:HttpSource ;
            s:url "https://10.10.0.10/sparql/http%3A%2F%2Fsomething.com%2Fdata";
            s:trust "all" ;
            s:username "user";
            s:password "pass";
            s:contentType "application/sparql-query" ;
            s:header [ s:name "Accept" ; s:value "text/csv" ] ;
    }
}

```



```

s:content """
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  SELECT ?s ?p ?o
  WHERE {
    ?s ?p ?o .
    FILTER(ISLITERAL(?o))
  }
  """ .
  ?rdf a s:RdfGenerator, s:OntologyGenerator ;
  s:as (?s ?p ?o) ;
  s:ontology <http://anzograph.com/ontologies/TopMovies> ;
  s:base <http://anzograph.com/data> .
}
}

```

API Queries

The following example queries the Google Recognize API to request transcriptions for voice recordings that are stored in a Google bucket.

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX anzo: <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl: <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

INSERT {
  GRAPH <http://anzograph.com/transcriptions> {
    ?record <http://google.com/transcript> ?transcript .
    ?record <http://google.com/confidence> ?confidence .
    ?record <http://google.com/file> ?file .
  }
}
WHERE {
  BIND(<gs://csi-se/demo/emergency-test.mp3> as ?file)
  BIND(UUID() as ?record)
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:HttpSource ;

```

```

        s:selector "results.alternatives" ;
        s:url "https://speech.googleapis.com/v1p1beta1/speech:recognize" ;
        s:authorization [ a s:BearerToken ; s:token ""ya29..." ] ;
        s:content ""
    {
        "config": {
            "encoding":"MP3",
            "sampleRateHertz": 16000,
            "languageCode": "en-US",
            "enableWordTimeOffsets": false
        },
        "audio": {
            "uri":"gs://csi-se/demo/emergency-test.mp3"}
    }
    "" ;
        ?confidence ("confidence") ;
        ?transcript ("transcript") .
    }
}
}

```

The example below includes the header and content properties to send a request that contains small text snippets for sentiment analysis.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ont: <http://cambridgesemantics.com/ontologies/Sentiment_Analysis#>

INSERT {
    GRAPH <http://anzograph.com/sentiment> {
        ?requirement a ont:Sentiment ;
        ont:p_Sentiment_Type ?sentiment ;
        ont:p_Sentiment_Score ?polarity .
    }
}
WHERE {
    ?requirement a <http://cambridgesemantics.com/Requirements> ;
    <http://cambridgesemantics.com/Requirements.reqText> ?requirement_text .

    SERVICE TOPDOWN <http://cambridgesemantics.com/services/DataToolkit> {
        ?data a s:HttpSource ;
    }
}

```

```

s:url "https://text-analysis12.p.rapidapi.com/sentiment-analysis/api/v1.1" ;
s:method "POST" ;
s:header [ s:name "Accept" ; s:value "application/json" ] ,
          [ s:name "X-RapidAPI-Key" ; s:value "key" ] ,
          [ s:name "X-RapidAPI-Host" ; s:value "text-analysis12.p.rapidapi.com"
] ;

s:contentType "application/json" ;
s:content ""{ "text": "{?requirement_text}" , "language": "english" }"" ;
?polarity ("aggregate_sentiment/compound" xsd:double);
?sentiment () .

}

}

```

Query an Elasticsearch Source

This topic provides details about the structure to use when writing GDI queries to read or ingest data from Elasticsearch data sources. It also includes example queries that may be useful as a starting point for writing your own GDI queries.

- [Query Syntax](#)
- [Query DSL and Filter Mapping](#)
- [Query Examples](#)

Query Syntax

The following query syntax shows the structure of a GDI query for Elasticsearch sources. The clauses, patterns, and placeholders that are links are described below.

```

# PREFIX Clause

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX es:     <http://elastic.co/search/>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause

```

```

{
  [ GRAPH <target_graph> { ]
  triple_patterns
[ ] ]
}

[ FROM Clause ]

WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView>(${targetGraph})

  {
    ?data a s:ElasticSource ;
      s:url "string" ;
      [ s:username "string" ; ]
      [ s:password "string" ; ]
      [ s:property [ s:name "string" ; s:value "string" ; ]
      [ es:aggregations [ rdf_list ] ; ]
      [ es:document "string" ; ]
      [ es:field "string" | ?variable ; ]
      [ es:highlight [ rdf_list ] ; ]
      [ es:html boolean ; ]
      [ es:index "string" ; ]
      [ es:minScore float ; ]
      [ es:query "string" | [ rdf_list ] ; ]
      [ es:routing "string" ; ]
      [ es:searchAfter [ rdf_list ] ; ]
      [ es:size int ; ]
      [ es:source boolean | [ rdf_list ] ; ]
      [ s:timeout int ; ]
      [ s:batching boolean | int ; ]
      [ s:concurrency int | [ list_of_properties ] ; ]
      [ s:rate int | "string" ; ]
      [ s:partitionBy "string" | ?variable ; ]
      [ s:locale "string" ; ]
      [ s:sampling int ; ]
      [ s:selector "string" | [ list ] ; ]
      [ s:model "string" ; ]
      [ s:key ("string") ; ]
  }
}

```

```

[ s:reference [ s:model "string" ; s:using ("string") ]
[ s:formats [ datatype_formatting_options ] ; ]
[ s:normalize boolean | [ normalization_rules ] ; ]
[ s:count ?variable ; ]
[ s:offset int ; ]
[ s:orderBy "string" | ?variable ; ]
[ s:limit int ; ]
# Mapping variables
?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
... ;
.
# Additional clauses such as BIND, VALUES, FILTER
}
}

```

Note

For readability, the parameters below exclude the base URIs

<<http://cambridgesemantics.com/ontologies/DataToolkit#>> and
 <<http://elastic.co/search/>> as well as the `s:` and `es:` prefixes. As shown in the examples, however, the prefixes or full property URIs do need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause defines the type of SPARQL query to run and the set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN]</code> < http://cambridgesemantics.com/services/DataToolkit > to invoke the GDI service when you are running a SELECT,

Option	Type	Description
		INSERT, or CONSTRUCT query that is not creating a view. When creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: <code>SERVICE</code> <pre><http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>).</pre> Using the <code>DataToolkitView</code> call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not necessarily returned.
url	string	This property specifies the URL to use to access the source. Important For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example: <pre>?data a s:ElasticSource ;</pre>

Option	Type	Description
		<pre>s:url "{{@es.hostname}}:{{@es.port}}" ; s:username "{{@es.username}}" ; s:password "{{@es.password}}" ;</pre>
username	string	This property lists the user name to use for the connection to the Elasticsearch. Tip If you want to group the username and password properties, you can wrap them with <code>s:credentials []</code>. For example: <pre>s:credentials [s:username "username" ; s:password "password" ;]</pre>
password	string	This property lists the password for the given username.
property	RDF list	This property can be included to list any source-specific configuration values. <pre>s:property [s:name "custom_property_name" ; s:value "custom_value"]</pre>
aggregations	RDF list	You can include this property to calculate aggregations over the specified bindings. For information about aggregations, see Aggregations in the Elasticsearch documentation.
document	string	This property lists the document(s) to search.

Option	Type	Description
field	string or variable	This property defines the field to operate on. The value can be a string or bound variable.
highlight	RDF list	You can include this property to define how results are highlighted. For information about the available properties, see Highlighting Elasticsearch Results .
html	boolean	This property controls whether to output HTML for highlighted results. Defaults to <code>true</code> .
index	string	This property can be included to specify the index to search.
minScore	float	This property defines the minimum score for matching documents. Documents with a lower score are not included in the search results.
query	string or RDF list	This property defines the query to execute. The value can be a string or a query object that maps to the Elasticsearch Query DSL . To generate the final query, the GDI combines <code>es:query</code> with any filters it can push to the Elasticsearch DSL. For more information about the <code>query</code> property and mapping Elasticsearch filters to SPARQL FILTER clauses, see Query DSL and Filter Mapping below.
routing	string	This property can be included to route a document to a specific shard or to limit the search to a particular shard.
searchAfter	RDF list	You can include this property to define the key values to start searching from.
size	int	This property maps to the <code>size</code> parameter in the Elasticsearch Search API and configures the batch size or maximum number of hits to return in a single call. Defaults to <code>10</code> and typically does not

Option	Type	Description
		need to be changed.
source	boolean or RDF list	This property can be included to specify the source data to include in results. The value can be a boolean, list of fields, or a list of variable bindings. When <code>true</code> , all source data is returned. When <code>false</code> , no source data is returned.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code>. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ;</pre>

Option	Type	Description
		<pre>s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code>, the GDI would use 3 executors with a rate delay of 750ms between requests.
partitionBy	string, variable, list	The GDI attempts to partition queries automatically across the available cores (slices) in Graph Lakehouse. To determine how to partition the query, the GDI uses metadata from the source database. It looks for any column in an index, preferring the primary key column if it is interpolable. However, it only considers the first

Option	Type	Description
		column in any index on the table. After determining the partition column, the GDI does a MIN/MAX on the column as well as a basic sizing query. To specify which column or columns the GDI should partition on, you can include the <code>partitionBy</code> property in the query. The property supports a list of source field names, bound variables, or the object <code>s:auto</code> , which forces the GDI to partition the data when the source does not define partitioning metadata.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the <code>SalesOrderHeader</code> table in the <code>Sales</code> schema. For more information about binding components and the selector property, see Using Binding Trees and Selector Paths .
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example,

Option	Type	Description
		<code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The <code>reference</code> property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or <code>dateTime</code> values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> . The GDI runs an Elasticsearch value count aggregation .
offset	int	This property can be used to offset the data that is returned by a number of rows.
orderBy	string, variable,	You can include this property to order the result set by a field name, a bound variable, or a list of names or bound variables.

Option	Type	Description
	list	
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
mapping_variable	variable	The mapping variables, in <code>?mapping_variable ("binding" [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Note The parentheses around the binding, data type, and format specifications are not required but are included in this document for readability.
binding	string	The <code>binding</code> is a literal value that binds a ?mapping_variable to a source column. If you specify a <code>?variable</code> that matches the source column name, then that variable name is the data selector and it is not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column, then the binding is needed to map the new variable to that source column.
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double,</pre>

Option	Type	Description
		<code>xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration, xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI</code>
datetime_ format	string	This option is used to specify the format to use for date and time data types. The GDI supports Java date and time formats. Specify days as "d," months as "M," and years as "y." For the time, specify "H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMMy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation <code>^nnnn</code> (e.g., <code>^1900</code>) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-yy^1900"</code> or <code>xsd:date "dd-MMM-yy^1990"</code>. When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

Query DSL and Filter Mapping

The vocabulary used in GDI queries against an ElasticSource closely mimics the Elasticsearch [Query DSL](#). The table below shows a side-by-side view of a DSL query that is mapped to SPARQL using the `es:query` property:

DSL	SPARQL
<pre> { "query": { "bool" : { "must" : { "term" : { "user.id" : "kimchy" } }, "filter": { "term" : { "tags" : "production" } }, "must_not" : { "range" : { "age" : { "gte" : 10, "lte" : 20 } } }, "should" : [{ "term" : { "tags" : "env1" } }, { "term" : { "tags" : "deployed" } }], "minimum_should_match" : 1, "boost" : 1.0 } } }</pre>	<pre> es:query [a es:BoolQuery ; es:must [a es:TermQuery ; es:field "user.id" ; es:value "kimchy" ;] ; es:filter [a es:TermQuery ; es:field "tags" ; es:value "production" ;] ; es:mustNot [a es:RangeQuery ; es:field "age" ; es:gte 10 ; es:lte 20 ;] ; es:should ([a es:TermQuery ; es:field "tags" ; es:value "env1"] [a es:TermQuery ; es:field "tags" ; es:value "deployed"]) ; es:minimumShouldMatch 1 ; es:boost 1.0 ;]</pre>

The following example SERVICE clause with comments provides details about how the GDI `es:query` property can be mapped to DSL:

```

SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
  ?data a es:ElasticSource ;
```

```

s:url "http://localhost:9200/" ;
# When the value of es:query is a simple literal,
# it is mapped to an Elastic query string query.
  es:query "literal"

# When the value of es:query is an RDF list,
# you can specify other query types,
# such as a match query:
  es:query [
    a es:MatchQuery ;
    es:field "title" | ?title ; # field can be a literal or bound variable
    es:query "moby dick" ;
  ] ;
# or a boolean query:
  es:query [
    a es:BoolQuery ;
    es:should ([
      a es:RangeQuery ;
      es:field ?amount ;
      es:gt 500 ;
      es:lt 1000 ;
    ] [
      a es:TermQuery ;
      es:field ?status ;
      es:value 'late' ;
    ]) ;
  ] ;
}

```

Filter Mapping

Filtering can be performed inside the `es:query` list or you can add a `FILTER` clause to the query. For example, the table below shows the SPARQL snippet above expressed as a `FILTER` clause.

SPARQL Query	FILTER Clause
<pre> es:query [a es:BoolQuery ; es:must [a es:TermQuery ; </pre>	<pre> FILTER(?user_id = "kimchy" && ?tags = "production" && !(?age >= 10 && ?age <= 20) && (?tags == "env1" ?tags </pre>

SPARQL Query	FILTER Clause
<pre> es:field "user.id" ; es:value "kimchy" ;] ; es:filter [a es:TermQuery ; es:field "tags" ; es:value "production" ;] ; es:mustNot [a es:RangeQuery ; es:field "age" ; es:gte 10 ; es:lte 20 ;] ; es:should ([a es:TermQuery ; es:field "tags" ; es:value "env1"] [a es:TermQuery ; es:field "tags" ; es:value "deployed"]) ; es:minimumShouldMatch 1 ; es:boost 1.0 ;]</pre>	<pre> == "deployed"))</pre>

The table below shows each of the supported ElasticSource FILTER translations. Only expressions matching the list below will be translated by the GDI. If the expression is of the form `value <= ?field`, the inequality is flipped to `?field > value` before translating.

es:query Expression	FILTER Clause Expression
<code>es:query [a es:BoolQuery ; es:mustNot expr]</code>	<code>!expr</code>

es:query Expression	FILTER Clause Expression
es:query [a es:BoolQuery ; es:must (left right)]	left && right
es:query [a es:BoolQuery ; es:should (left right)]	left right
es:query [a es:RangeQuery ; es:field ?field ; es:lt value]	?field < value
es:query [a es:RangeQuery ; es:field ?field ; es:lte value]	?field <= value
es:query [a es:TermQuery ; es:field ?field ; es:value value]	?field = value
es:query [a es:BoolQuery ; es:mustNot [a es:TermQuery ; es:field ?field ; es:value value]]	?field != value
es:query [a es:RangeQuery ; es:field ?field ; es:gte value]	?field >= value
es:query [a es:RangeQuery ; es:field ?field ; es:gt value]	?field > value
es:query [a es:QueryStringQuery ; es:field ?field ; es:query pattern ; es:defaultOperator "AND"]	REGEX(?field, pattern, "q")
es:query [a es:TermsQuery ; es:field ?field ; es:value value, ...]	IN(?field, value, ...)
es:query [a es:BoolQuery ; es:mustNot [a es:TermsQuery ; es:field ?field ; es:value value, ...]]	NOT IN(?field, value, ...)
es:query [a es:MatchQuery ; es:field ?field ; es:query	CONTAINS

es:query Expression	FILTER Clause Expression
value ; es:lenient true]	(?field, value)
es:query [a es:PrefixQuery ; es:field ?field ; es:value value]	STRSTARTS (?field, value)
es:query [a es:ExistsQuery ; es:field ?field]	BOUND(?field)

Query Examples

- [Aggregations](#)
- [Highlighting](#)

Aggregations

The following example query performs terms aggregations.

```
PREFIX es: <http://elastic.co/search/>
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT DISTINCT *
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
    ?data a es:ElasticSource;
    s:url "https://{{@es.hostname}}:{{@es.port}}/" ;
    s:username "{{@es.username}}" ;
    s:password "{{@es.password}}" ;
    es:index "templated_consumption_es" ;
    es:query "*ELM*" ;
    ?instance () ;
    es:aggregations [
      ?artifactTypes [
        a es:TermsAggregation ;
        es:field ?artifactType ;
        es:meta [
          ?label "artifactType" ;
        ] ;
      ] ;
    ] ;
  }
}
```

```

        ?value () ;
        ?count () ;
    ] ;
    ?fileTypes [
        a es:TermsAggregation ;
        es:field ?fileType ;
        es:meta [
            ?label "fileType" ;
        ] ;
        ?value () ;
        ?count () ;
    ] ;
    ?managedBys [
        a es:TermsAggregation ;
        es:field ?managedBy ;
        es:meta [
            ?label "managedBy" ;
        ] ;
        ?value () ;
        ?count () ;
    ] ;
] .
}
}

```

Highlighting

The following example configures highlighting for fragments from the actor field.

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX es: <http://elastic.co/search/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT *
WHERE {
    SERVICE TOPDOWN <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a es:ElasticSource ;
        es:url "http://localhost:9200/" ;
        es:index "films" ;
        es:html false ;
        es:query "Clint" ;
        es:field ?actor, ?director ;
        es:highlight [

```

```

        es:field ?actor ;
        es:type "plain" ;
        es:fragmentSize 200 ;
        es:numberOfFragments 10 ;
        es:preTags "<mark hit='true'>" ;
        es:postTags "</mark>" ;
    ] ;
    s:selector "film" ;
    ?actor (xsd:string) ;
    ?awards (xsd:string) ;
    ?director (xsd:string) ;
    ?image (xsd:string) ;
    ?length (xsd:long) ;
    ?popularity (xsd:long) ;
    ?subject (xsd:string) ;
    ?title (xsd:string) ;
    ?year (xsd:long) ;
    ?score () ;
    ?id () ;
    ?highlights [
        ?field () ;
        ?fragment () ;
    ] .
    FILTER(?year = 1990 || ?length > 103)
    FILTER(REGEX(?title, "Manhattan", "q") || REGEX(?subject, "Comedy", "q") || REGEX
(?subject, "Drama", "q"))
}
}

```

Highlighting Elasticsearch Results

By including the **highlight** property in ElasticSource GDI queries, you can configure the response to include highlights for search results. For general information about highlighting Elasticsearch responses, see [Highlighting](#) in the Elasticsearch documentation. Highlight property usage is described below.

- [Highlight Syntax](#)
- [Highlight Examples](#)

Highlight Syntax

```
es:highlight [
  es:boundaryChars "string" ;
  es:boundaryMaxScan int ;
  es:boundaryScannerLocale "string" ;
  es:boundaryScannerType "string" ;
  es:field "string" ;
  es:forceSource boolean ;
  es:fragmentSize int ;
  es:fragmenter "string" ;
  es:highlightFilter boolean ;
  es:highlightQuery "string" | [ rdf_list ] ;
  es:highlighterType "string" ;
  es:noMatchSize int ;
  es:numberOfFragments int ;
  es:order "string" ;
  es:phraseLimit int ;
  es:postTags "string" ;
  es:preTags "string" ;
  es:requireFieldMatch boolean ;
] ;
```

Option	Type	Description
boundaryChars	string	This property can be used to define the boundary characters to look for. Defaults to <code>. , ! ? \t \n</code> .
boundaryMaxScan	int	This property can be used to place a limit on the number of characters to scan when looking for boundary characters. Defaults to 20.
boundaryScannerLocale	string	This property defines the language tag (such as "en-US" or "fr-FR") to apply when searching for sentence and word boundaries.
boundaryScannerType	string	If highlighterType is <code>unified</code> or <code>fvh</code> , this property can be used to specify how to break the highlighted fragments. This property is ignored when the

Option	Type	Description
		highlighter type is <code>plain</code>. The list below describes the valid values: chars: Valid when the highlighter type is fast vector highlighter (fvh) (<code>es:highlighterType "fvh"</code>). Specifies that the highlighting boundaries are the characters specified by boundaryChars. The boundaryMaxScan value controls how far to scan for boundary characters. This is the default value for fvh. sentence: This is the default value for the unified highlighter. It configures highlighted fragments to break at the next sentence boundary. You can specify the locale to use with boundaryScannerLocale. When used with the unified highlighter, the sentence scanner splits sentences bigger than fragmentSize at the first word boundary next to <code>fragmentSize</code>. You can set <code>fragmentSize</code> to 0 to avoid splitting sentences. word: Configures highlighted fragments to break at the next word boundary. You can specify the locale to use with boundaryScannerLocale.
field	string or variable	This property specifies the field to retrieve highlights for. It can include a <code>?variable</code> (which the GDI maps to the full path of the field in the

Option	Type	Description
		Elasticsearch document), a field name, or a field name pattern. For example: <pre>es:highlight [es:field ?actor ; es:field "film.actor" ; es:field "film.*" ; es:field "*" ;]</pre>
forceSource	boolean	This property controls whether to highlight based on the source even if the field is stored separately. Defaults to <code>false</code> .
fragmentSize	int	This property specifies the number of characters to include in highlighted fragments. Defaults to <code>100</code> .
fragmenter	string	If highlighterType is <code>plain</code>, this property can be used to specify how to break up text in highlight snippets. The list below describes the valid values: • simple: Breaks text into fragments that are the same size (as specified by fragmentSize). • span: The default value. Breaks text into fragments that are the same size but tries to avoid breaking text between highlighted terms.
highlightFilter	boolean	This property controls whether to highlight filter results.

Option	Type	Description
highlightQuery	string or object	This property specifies the highlight query. The value can be a string or a query object that maps to the Elasticsearch query DSL.
highlighterType	string	This property defines the type of highlighter to use, "plain", "unified", or "fvh".
noMatchSize	int	This property specifies the number of characters to return from the beginning of the field if there are no matching fragments to highlight. Defaults to 0 (nothing is returned).
numberOfFragments	int	This property can be used to set the maximum number of fragments to generate. If this property is set to 0, no fragments are returned. Instead, the entire field contents are highlighted and returned, which can be useful if you want to highlight short text (such as a title or address) for which fragmentation is not required. Defaults to 5. If the number of fragments is 0, fragmentSize is ignored.
order	string	This property can be included to sort highlighted fragments by score. When <code>es:order "score"</code> , the most relevant fragments are output first. Defaults to "none"; fragments are output in the order they appear in the field.
phraseLimit	int	If highlighterType is <code>fvh</code> , this property can be used to limit the number of matching phrases to consider. Limiting the number of phrases prevents the <code>fvh</code> highlighter from analyzing too many phrases and consuming too much memory. Defaults to 256.

Option	Type	Description
postTags	string	This property is used in conjunction with preTags to define the HTML tags to use for the highlighted elements. This property defines the closing tag to use after the highlighted text. Defaults to <code></code> .
preTags	string	This property is used in conjunction with postTags to define the HTML tags to use for the highlighted elements. This property defines the opening tag to use before the highlighted text. Defaults to <code></code> .
requireFieldMatch	boolean	This property controls whether to highlight only the fields that contain a query match. Defaults to <code>true</code> . If <code>false</code> , all fields are highlighted.

Highlight Examples

The following example configures highlighting for fragments from the actor field.

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX es: <http://elastic.co/search/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT *
WHERE {
  SERVICE TOPDOWN <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a es:ElasticSource ;
    es:url "http://localhost:9200/" ;
    es:index "films" ;
    es:html false ;
    es:query "Clint" ;
    es:field ?actor, ?director ;
    es:highlight [
      es:field ?actor ;
      es:type "plain" ;
      es:fragmentSize 200 ;
      es:numberOfFragments 10 ;
      es:preTags "<mark hit='true'>" ;
```

```

        es:postTags "</mark>" ;
    ] ;
    s:selector "film" ;
    ?actor (xsd:string) ;
    ?awards (xsd:string) ;
    ?director (xsd:string) ;
    ?image (xsd:string) ;
    ?length (xsd:long) ;
    ?popularity (xsd:long) ;
    ?subject (xsd:string) ;
    ?title (xsd:string) ;
    ?year (xsd:long) ;
    ?score () ;
    ?id () ;
    ?highlights [
        ?field () ;
        ?fragment () ;
    ] .
FILTER(?year = 1990 || ?length > 103)
FILTER(REGEX(?title, "Manhattan", "q") || REGEX(?subject, "Comedy", "q") || REGEX
(?subject, "Drama", "q"))
}
}

```

Query a File Source

The Graph Data Interface (GDI) uses the [Apache Commons VFS](#) library to work with file systems. Many of the properties specified in queries against file sources reflect the requirements of the VFS library for that source. The topics in this section provide guidance on writing GDI queries for each of the supported file types.

In this section:

Query CSV and TSV Files

This topic provides details about the structure to use when writing GDI queries to read or ingest data from CSV or TSV files. It also includes example queries that may be useful as a starting point for writing your own GDI queries.

- [Query Syntax](#)
- [Query Examples](#)

Query Syntax

The following query syntax shows the structure of a GDI query for CSV and TSV sources. The clauses, patterns, and placeholders that are links are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
{
  [ GRAPH <target_graph> { ]
    triple_patterns
  [ ] ]
}
[ FROM Clause ]
WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView> (<target_graph>)

  {
    ?data a s:FileSource ;
    s:url "string" ;
    [ s:options [ file_storage_connection_options ] ; ]
    [ s:pattern "string" ; ]
    [ s:maxDepth int ; ]
    [ s:format [ source_format_options ; ] ; ]
    [ s:mimetype "string" ; ]
    [ s:username "string" ; ]
  }
}
```

```

[ s:password "string" ; ]
[ s:timeout int ; ]
[ s:batching boolean | int ; ]
[ s:concurrency int | [ list_of_properties ] ; ]
[ s:rate int | "string" ; ]
[ s:locale "string" ; ]
[ s:sampling int ; ]
[ s:selector "string" | [ list ] ; ]
[ s:model "string" ; ]
[ s:key ("string") ; ]
[ s:reference [ s:model "string" ; s:using ("string") ]
[ s:formats [ datatype_formatting_options ] ; ]
[ s:normalize boolean | [ normalization_rules ] ; ]
[ s:count ?variable ; ]
[ s:offset int ; ]
[ s:limit int ; ]
# Mapping variables
?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
... ;
.
# Additional clauses such as BIND, VALUES, FILTER
}
}

```

Note

For readability, the parameters below exclude the base URI

<<http://cambridgesemantics.com/ontologies/DataToolkit#>> as well as the `s:` prefix. As shown in the examples, however, the `s:` prefix or full property URI does need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result	N/A	The result clause defines the type of SPARQL query to run and the

Option	Type	Description
Clause		set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN] <http://cambridgesemantics.com/services/DataToolkit></code> to invoke the GDI service when you are running a SELECT, INSERT, or CONSTRUCT query that is not creating a view. When creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: <code>SERVICE <http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>)</code>. Using the <code>DataToolkitView</code> call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not necessarily returned.
url	string	This property specifies the file system location of the source file or directory of files. When specifying a directory (such as <code>s:url "/opt/shared-files/loads/"</code>), the GDI loads all of the file formats it recognizes. To specify a directory but limit the number or

Option	Type	Description
type of files that are read, you can include the pattern and/or maxDepth properties.		
options	RDF list	If additional connection information needs to be provided to access the file storage system, include the <code>options</code> property to list any storage-specific connection parameters. See File Storage Connection Options for information about the supported properties for each storage type.
pattern	string	This property can be used to specify a wildcard pattern for matching file names. For example, <code>s:pattern "common_prefix*.csv"</code> . You can include one <code>s:pattern</code> property per <code>FileSource</code> . The GDI supports Unix file globbing syntax outside of parentheses. Within parentheses, full Java regular expression language is supported. For example, including <code>s:pattern "data/**/customer_*.csv"</code> tells the GDI to load all files that match the pattern "customer_*.csv" from any number of subdirectories under the <code>data</code> directory. Similarly <code>s:pattern "(\\d+)/transaction_*.csv"</code> tells the GDI to load all files that match the pattern "transaction_*.csv" in all subdirectories.
maxDepth	int	This property can be used to limit the directory traversal depth. By default, when <code>s:url</code> specifies a directory (and a <code>s:pattern</code> that limits that traversal depth is not specified), all subdirectories are processed. To process only the files in the top level directory, set <code>maxDepth</code> to 0 (<code>s:maxDepth 0</code>). To process the files in the top level directory plus the first-level subdirectories, set <code>maxDepth</code> to 1 (<code>s:maxDepth 1</code>), and so on.
format	RDF list	You can include the <code>format</code> property to add parameters that describe the source files. See File Source Format Options for details about the supported parameters.

Option	Type	Description
mimetype	string	If you are querying TSV files that do not have a .tsv file extension, include the <code>mimetype</code> property with a value of <code>text/tsv</code> (<code>s:mimetype "text/tsv"</code>).
username	string	If authentication is required to access the source, include this property to specify the user name.
password	string	This property lists the password for the given username.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code>. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [</pre>

Option	Type	Description
		<pre>s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code>, the GDI would use 3 executors with a rate delay of 750ms between requests.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.

Option	Type	Description
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the SalesOrderHeader table in the Sales schema. For more information about binding components and the selector property, see Using Binding Trees and Selector Paths .
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about key, see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about reference, see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI

Option	Type	Description
		queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
mapping_variable	variable	The mapping variables, in <code>?mapping_variable (["binding"] [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Note The parentheses around the binding, data type, and format

Option	Type	Description
		specifications are not required but are included in this document for readability.
binding	string	The <code>binding</code> is a literal value that binds a <code>?mapping_variable</code> to a source column. If you specify a <code>?variable</code> that matches the source column name, then that variable name is the data selector and it is not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column, then the binding is needed to map the new variable to that source column. For example for CSV, the following pattern simply binds the source column AIRLINE to the lowercase variable <code>?airline</code>: <code>?airline ("AIRLINE")</code>. Note For FileSource, periods (.), forward slashes (/), and brackets ([]) are parsed as path notation. Therefore, if a source column name includes any of those characters they must be escaped in the binding. Use two backslashes (\\) as an escape character. For example, if a column name is average/day, the variable and binding pattern could be written as <code>?averagePerDay ("average\\/day")</code>.
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double, xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration,</pre>

Option	Type	Description
		xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI
datetime_ format	string	This option is used to specify the format to use for date and time data types. The GDI supports Java date and time formats. Specify days as "d," months as "M," and years as "y." For the time, specify "H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMyy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation ^nnnn (e.g., ^1900) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as xsd:date "dd-MMM-yy^1900" or xsd:date "dd-MMM-yy^1990". When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

Query Examples

The example below ingests a directory of CSV files. The pattern property (`s:pattern "post_[0-9]*_[0-9]*.csv"`) is used to narrow down the set of files to load. Since the files use a pipe (|) as the delimiter rather than a comma (,), the delimiter property is also included to specify the | character.

```
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX snvoc:  <http://www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>
PREFIX sntag:  <http://www.ldbc.eu/ldbc_socialnet/1.0/tag/>
```

```

PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

INSERT {
  GRAPH <http://anzograph.com/vocab>
  {
    ?postIRI a snvoc:Post, snvoc:Message ;
    snvoc:creationDate ?creationDate ;
    snvoc:id ?id ;
    snvoc:imageFile ?imageFile ;
    snvoc:locationIP ?locationIP ;
    snvoc:browserUsed ?browserUsed ;
    snvoc:language ?language ;
    snvoc:content ?content ;
    snvoc:length ?length ;
  }
}
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource ;
    s:url "/opt/shared-files/data/csv/post_6_0/" ;
    s:pattern "post_[0-9]*_[0-9]*.csv" ;
    s:format [ s:delimiter "|" ] ;
    ?creationDate (xsd:dateTime) ;
    ?id (xsd:string) ;
    ?imageFile (xsd:string) ;
    ?locationIP (xsd:string) ;
    ?browserUsed (xsd:string) ;
    ?language (xsd:string) ;
    ?content (xsd:string) ;
    ?length(xsd:string) .
    BIND(IRI("http://www.ldbc.eu/ldbc_socialnet/1.0/data/Post/{?id}") as ?postIRI)
  }
}

```

The example below is similar to the query above but it specifies the formats to use for the `xsd:date` values.

```

PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>

```

```

PREFIX zowl: <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX kd: <http://cambridgesemantics.com/ont/autogen/Rh/Kaggle_Diabetes#>

INSERT {
  GRAPH <http://anzograph.com/diagnoses>
  {
    ?URI a kd:Diagnosis ;
    kd:Diagnosis_DiagnosisGuid ?diagnosis_guid ;
    kd:Diagnosis_PatientGuid ?patient_guid ;
    kd:Diagnosis_ICD9Code ?icd9Code ;
    kd:Diagnosis_DiagnosisDescription ?diagnosisDescription ;
    kd:Diagnosis_StartDate ?cus_start_date ;
    kd:Diagnosis_EndDate ?Date_End ;
    kd:Diagnosis_Acute ?acute ;
    kd:Diagnosis_UserGuid ?UserGuid .
  }
}
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?csv a s:FileSource ;
    s:url "/opt/shared-files/source_data/kaggle_diabetes/" ;
    s:pattern "Diagnosis.csv" ;
    s:format [ s:delimiter "," ] ;
    ?diagnosis_guid ("DiagnosisGuid" xsd:string) ;
    ?patient_guid ("PatientGuid" xsd:string) ;
    ?icd9Code ("ICD9Code" xsd:string) ;
    ?diagnosisDescription ("DiagnosisDescription" xsd:string) ;
    ?acute ("Acute" xsd:int ) ;
    ?UserGuid ("UserGuid" xsd:string) ;
    ?cus_start_date ("CUSTOMER_START_DATE" xsd:date "yyyy-MM-dd") ;
    ?Date_End ("Date End" xsd:date "MM/dd/yy") .
  }
  BIND(IRI(CONCAT("urn://anzograph.com/kaggle_diabetes/patient/",ENCODE_FOR_URI
(?diagnosis_guid)))) as ?URI)
}

```

Query JSON and NDJSON Files

This topic provides details about the structure to use when writing GDI queries to read or ingest data from JSON or NDJSON files. It also includes example queries that may be useful as a starting point for writing your own GDI queries.

- [Query Syntax](#)
- [Hierarchical Bindings and Arrays](#)
- [Capturing Property Keys](#)
- [Query Examples](#)

Query Syntax

The following query syntax shows the structure of a GDI query for JSON sources. The clauses, patterns, and placeholders that are links are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
{
  [ GRAPH <target_graph> { ]
    triple_patterns
  [ ] ]
}
[ FROM Clause ]
WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView> (<target_graph>)
```



```

{
  ?data a s:FileSource ;
    s:url "string" ;
    [ s:options [ file_storage_connection_options ] ; ]
    [ s:pattern "string" ; ]
    [ s:maxDepth int ; ]
    [ s:format [ source_format_options ; ] ; ]
    [ s:mimetype "string" ; ]
    [ s:username "string" ; ]
    [ s:password "string" ; ]
    [ s:timeout int ; ]
    [ s:batching boolean | int ; ]
    [ s:concurrency int | [ list_of_properties ] ; ]
    [ s:rate int | "string" ; ]
    [ s:locale "string" ; ]
    [ s:sampling int ; ]
    [ s:selector "string" | [ list ] ; ]
    [ s:model "string" ; ]
    [ s:key ("string") ; ]
    [ s:reference [ s:model "string" ; s:using ("string") ]
    [ s:formats [ datatype_formatting_options ] ; ]
    [ s:normalize boolean | [ normalization_rules ] ; ]
    [ s:count ?variable ; ]
    [ s:offset int ; ]
    [ s:limit int ; ]
    # Mapping variables and hierarchical bindings
    ?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
    ... ;
    .
    # Additional clauses such as BIND, VALUES, FILTER
}
}

```

Note

For readability, the parameters below exclude the base URI

<<http://cambridgesemantics.com/ontologies/DataToolkit#>> as well as the `s:` prefix. As shown in the examples, however, the `s:` prefix or full property URI does need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific declarations.
Result Clause	N/A	The result clause defines the type of SPARQL query to run and the set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN] <http://cambridgesemantics.com/services/DataToolkit></code> to invoke the GDI service when you are running a SELECT, INSERT, or CONSTRUCT query that is not creating a view. When creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: <code>SERVICE <http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>)</code>. Using the <code>DataToolkitView</code> call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not

Option	Type	Description
		necessarily returned.
url	string	This property specifies the file system location of the source file or directory of files. When specifying a directory (such as <code>s:url "/opt/shared-files/loads/"</code>), the GDI loads all of the file formats it recognizes. To specify a directory but limit the number or type of files that are read, you can include the pattern and/or maxDepth properties.
options	RDF list	If additional connection information needs to be provided to access the file storage system, include the <code>options</code> property to list any storage-specific connection parameters. See File Storage Connection Options for information about the supported properties for each storage type.
pattern	string	This property is used to specify a wildcard pattern for matching file names. For example, <code>s:pattern "common_prefix*.json"</code> . You can include one <code>s:pattern</code> property per FileSource. The GDI supports Unix file globbing syntax outside of parentheses. Within parentheses, full Java regular expression language is supported. For example, including <code>s:pattern "data/**/customer_*.json"</code> tells the GDI to load all files that match the pattern "customer_*.json" from any number of subdirectories under the <code>data</code> directory. Similarly <code>s:pattern "(\\d+)/transaction_*.json"</code> tells the GDI to load all files that match the pattern "transaction_*.json" in all subdirectories.
maxDepth	int	This property can be used to limit the directory traversal depth. By default, when <code>s:url</code> specifies a directory (and a <code>s:pattern</code> that limits that traversal depth is not specified), all subdirectories are processed. To process only the files in the top level directory, set <code>maxDepth</code> to 0 (<code>s:maxDepth 0</code>). To process the files in the top level

Option	Type	Description
		directory plus the first-level subdirectories, set <code>maxDepth</code> to 1 (<code>s:maxDepth 1</code>), and so on.
format	RDF list	You can include the <code>format</code> property to add parameters that describe the source files. See File Source Format Options for details about the supported parameters.
mimetype	string	If you are querying NDJSON files that do not have an <code>.ndjson</code> file extension, include the <code>mimetype</code> property with a value of <code>application/x-ndjson</code> (<code>s:mimetype "application/x-ndjson"</code>).
username	string	If authentication is required to access the source, include this property to specify the user name.
password	string	This property lists the password for the given username.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code> . If the value is an integer, it configures a

Option	Type	Description
		maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a

Option	Type	Description
		rate delay of 666ms between requests. If <code>s:rate "240/minute"</code> , the GDI would use 3 executors with a rate delay of 750ms between requests.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used for JSON path extraction to traverse nested structures and target specific data. For example, <code>s:selector "projects"</code> targets the <code>projects</code> class of data. To express a hierarchy, use dot notation. For example, <code>s:selector "region.state.city"</code> navigates a hierarchy to target <code>city</code> data. For more information about binding components and the selector property, see Using Binding Trees and Selector Paths .
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .

Option	Type	Description
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
mapping_variable	variable	The mapping variables, in <code>?mapping_variable (["binding"] [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the

Option	Type	Description
		source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Tip See Hierarchical Bindings and Arrays below for more information about configuring mapping variables and unpacking JSON files with nested objects and arrays.
binding	string	The <code>binding</code> is a literal value that binds a <code>?mapping_variable</code> to a source column. If you specify a <code>?variable</code> that matches the source column name, then that variable name is the data selector and it is not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column, then the binding is needed to map the new variable to that source column. For example for CSV, the following pattern simply binds the source column AIRLINE to the lowercase variable <code>?airline</code>: <code>?airline ("AIRLINE")</code>. Note For FileSource, periods (<code>.</code>), forward slashes (<code>/</code>), and brackets (<code>[]</code>) are parsed as path notation. Therefore, if a source column name includes any of those characters they must be escaped in the binding. Use two backslashes (<code>\\</code>) as an escape character. For example, if a column name is average/day, the variable and binding pattern could be written as <code>?averagePerDay ("average\\/day")</code>.

Option	Type	Description
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double, xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration, xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI</pre>
datetime_ format	string	This option is used to specify the format to use for date and time data types. The GDI supports Java date and time formats. Specify days as "d," months as "M," and years as "y." For the time, specify "H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMMyy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation <code>^nnnn</code> (e.g., <code>^1900</code>) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-yy^1900"</code> or <code>xsd:date "dd-MMM-yy^1990"</code>. When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

Hierarchical Bindings and Arrays

When configuring the mapping variables in a query, the GDI provides syntax for unpacking JSON files with nested objects and arrays. One way to express hierarchies in queries is to use brackets ([]) to group objects into binding trees. For example, the WHERE clause snippet below organizes mapping variable objects into an `hourly/data` hierarchy by nesting the `?data` patterns inside the `?hourly [ ]` tree:

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource;
    s:url "/mnt/data/json/weather.json" ;
    ?latitude (xsd:double) ;
    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    ?hourly
  [
    ?data
  [
    ?time (xsd:long) ;
    ?rainProbability ("precipProbability" xsd:double) ;
    ?temperature (xsd:double) ;
    ?feelsLike ("apparentTemperature" xsd:double) ;
    ?windSpeed (xsd:double) ;
  ] ;
  ] .
  }
}
```

When constructing object binding trees, if you choose to introduce the hierarchy with a variable name that is not an exact match to the source label, include a **selector** property to list the value from the source. For example, in the WHERE clause snippet below, `s:selector` is included to **select** `eventHeader` in the source as `?event` in the query and `statLocation` as `?location`.

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource ;
```

```

s:url "/mnt/data/json/part_1.json" ;
?event
[
    s:selector "eventHeader" ;
    ?eventId (xsd:string) ;
    ?eventName (xsd:string) ;
    ?eventVersion (xsd:string) ;
    ?eventTime (xsd:dateTime) ;
] ;
?location
[
    s:selector "statLocation" ;
    ?locationId (xsd:string) ;
    ?lineNo (xsd:int) ;
    ?statNo (xsd:int) ;
    ?statId (xsd:int) ;
] .
}
}

```

As an alternative to grouping objects in binding trees, the **selector** property also supports using dot notation to specify paths. For example, the WHERE clause snippet below rewrites the first example query to express the same `hourly/data` hierarchy as a path in the `s:selector` value:

```

WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource;
    s:url "/mnt/data/json/weather.json" ;
    ?latitude (xsd:double) ;
    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    s:selector: "hourly.data" ;
    ?time (xsd:long) ;
    ?rainProbability ("precipProbability" xsd:double) ;
    ?temperature (xsd:double) ;
    ?feelsLike ("apparentTemperature" xsd:double) ;
    ?windSpeed (xsd:double) .
  }
}

```

In addition to object binding trees and selectors, the GDI offers additional syntax for reading or ingesting JSON sources with nested objects and arrays. For example, following the JSON sample file below is a query that captures each value in the arrays:

```
{
  "payload" :
  {
    "IBP_IndEvent_MSR" :
    {
      "unit" : "ms",
      "value" : [ 0, 1 ]
    },
    "IBP_IndEvent_RMF" :
    {
      "unit" : "-",
      "value" : [ 0.012, 1.398, 3.1415 ]
    }
  }
}
```

To read the JSON file above, the following query uses an object binding (`?values [ ]`) to drill down to the `value` arrays in the source. An `@` selector is specified in the `?value` variable binding (`?value ("@" xsd:double)`) to retrieve each of the array values. For an array of primitive values, the `@` selector captures each value in the array. If the source `value` was an array of objects, the `@` selector would retrieve a JSON representation for each object in the array. In addition to creating a new binding context for the primitive array values, the `?values` object binding also includes `?index ("!array::index")` to capture the index array with the primitive value.

```
PREFIX s:    <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
SELECT *
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
    ?data a s:FileSource ;
    s:url  "/mnt/data/json/array-index.json" ;
    s:selector "payload.*" ;
    ?unit (xsd:string) ;
    ?values [
      s:selector "value" ;
```

```

        ?value ("@" xsd:double) ;
        ?index ("!array:index") ;
    ] .
}
}

```

The results of the query are shown below:

unit	value	index
ms	0	0
ms	1	1
-	0.012	0
-	1.398	1
-	3.1415	2

If you do not want to retrieve all of the values in an array, you can include the specific index number to retrieve instead of using the @ symbol. In the variable binding, the index number is appended in brackets ([]) to the binding column name. For example, the following variable binding retrieves the second index value (the third value in the array) from a "projects" array:

?project ("projects[2]"). The next example uses the following JSON file:

```

{
  "field1" : "value1" ,
  "arrayfield" : [
    "arrayvalue1",
    "arrayvalue2"
  ]
}

```

To retrieve only the second value in the array, the following query appends the index value 1 to the array column name, arrayfield:

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
SELECT *
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
    ?json a s:FileSource ;
    s:url "/mnt/data/json/array-index-2.json" ;
    ?field1 (xsd:string) ;
    ?arrayval ("arrayfield[1]" xsd:string) .
  }
}

```

```
}  
}
```

The results of the query are shown below:

```
field1      | arrayval  
-----+-----  
value1      | arrayvalue2
```

Capturing Property Keys

In GDI Generator queries, the names of property keys can be captured from files by including a variable as the `s:selector` and using the same variable as the `s:key`. For example, the GDI query below ingests the following simple JSON file.

```
# company.json  
{  
  "AAPL": {  
    "name": "Apple Corp"  
  },  
  "MSFT": {  
    "name": "Microsoft"  
  },  
  "IBM": {  
    "name": "IBM"  
  }  
}
```

In the query, the keys "AAPL," "MSFT," and "IBM" are selected as the `?TickerSymbol` variable and the key is set to the same value.

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>  
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>  
  
INSERT { GRAPH <http://anzograph.com/companies> {  
  ?s ?p ?o .  
}  
}  
  
WHERE {  
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {  
    ?data a s:FileSource ;  
    s:url "/opt/shared/data/company.json" ;  
    s:selector "?TickerSymbol" ;
```

```

    s:key (?TickerSymbol) ;
    s:model "Company" ;
    ?TickerSymbol (xsd:string) ;
    ?name (xsd:string) .

?rdf a s:RdfGenerator, s:OntologyGenerator;
  s:as (?s ?p ?o) ;
  s:ontology <http://anzograph.com/ontologies/companies> ;
  s:base ${targetGraph} .
}
}

```

Selecting the predicates and objects from the graph shows the tickerSymbol predicate and value.

```

SELECT ?p ?o
FROM <http://anzograph.com/companies>
WHERE { ?s ?p ?o . }
ORDER BY desc(?o)

```

p	o
http://anzograph.com/ontologies/company#Company.name	Microsoft
http://anzograph.com/ontologies/company#Company.tickerSymbol	MSFT
http://anzograph.com/ontologies/company#Company.name	IBM
http://anzograph.com/ontologies/company#Company.tickerSymbol	IBM
http://anzograph.com/ontologies/company#Company.name	Apple Corp
http://anzograph.com/ontologies/company#Company.tickerSymbol	AAPL
...	

Query Examples

The example query below reads a JSON file that contains data about weather. Since the file is hierarchical, the `s:selector` property is included to specify the path to `data` in the `hourly/data` hierarchy:

```

PREFIX s:    <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
SELECT *
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource;

```

```

    s:url "/mnt/data/json/weather.json" ;
    ?latitude (xsd:double) ;
    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    s:selector: "hourly.data" ;
    ?time (xsd:long) ;
    ?rainProbability ("precipProbability" xsd:double) ;
    ?temperature (xsd:double) ;
    ?feelsLike ("apparentTemperature" xsd:double) ;
    ?windSpeed (xsd:double) .
}
}

```

The following example query ingests data from a JSON file that contains data about the New York Times best selling books.

```

PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX books:  <http://cambridgesemantics.com/ontologies/NYT_Bestsellers_Ontology#>
INSERT {
  GRAPH <http://anzograph.com/books> {
    ?book a books:Book ;
    books:p_Title ?title ;
    books:p_Description ?description ;
    books:p_Author ?author ;
    books:p_Publisher ?publisher ;
    books:p_Date ?rawdate .
  }
}
WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource ;
    s:url "/mnt/data/json/nyt_best_sellers.json" ;
    ?title () ;
    ?author () ;
    ?description () ;
    ?publisher () ;
    ?price() ;
    ?rawdate ("bestsellers_date.$date.$numberLong") .
  }
  BIND(IRI(CONCAT("http://anzograph.com/ontologies/NYT_Bestsellers_Ontology/", ENCODE_
FOR_URI(?title))) AS ?book) .
}

```


A snippet of the file's contents is shown below:

```
{
  "_id": {
    "$oid": "5b4aa4ead3089013507db18b"
  },
  "bestsellers_date": {
    "$date": {
      "$numberLong": "1211587200000"
    }
  },
  "published_date": {
    "$date": {
      "$numberLong": "1212883200000"
    }
  },
  "amazon_product_url": "http://www.amazon.com/Odd-Hours-Dean-
Koontz/dp/0553807056?tag=NYTBS-20",
  "author": "Dean R Koontz",
  "description": "Odd Thomas, who can communicate with the dead, confronts evil forces
in a California coastal town.",
  "price": {
    "$numberInt": "27"
  },
  "publisher": "Bantam",
  "title": "ODD HOURS",
  "rank": {
    "$numberInt": "1"
  },
  "rank_last_week": {
    "$numberInt": "0"
  },
  "weeks_on_list": {
    "$numberInt": "1"
  }
}
```

Query XML Files

This topic provides details about the structure to use when writing GDI queries to read or ingest data from XML files. It also includes example queries that may be useful as a starting point for writing your own GDI queries.

- [Query Syntax](#)
- [Hierarchical Bindings and Arrays](#)
- [Query Examples](#)

Query Syntax

The following query syntax shows the structure of a GDI query for XML sources. The clauses, patterns, and placeholders that are links are described below.

```
# PREFIX Clause
PREFIX s:      <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX anzo:   <http://openanzo.org/ontologies/2008/07/Anzo#>
PREFIX zowl:   <http://openanzo.org/ontologies/2009/05/AnzoOwl#>
PREFIX dc:     <http://purl.org/dc/elements/1.1/>

# Result Clause
{
  [ GRAPH ${targetGraph} { ]
    triple_patterns
  [ ] ]
}
[ ${usingSources} ]

WHERE
{
  # SERVICE Clause: Include the following service call when reading or inserting data.
  SERVICE [ TOPDOWN ] <http://cambridgesemantics.com/services/DataToolkit>

  # View SERVICE Clause: Or use the service call below when constructing a view.
  SERVICE <http://cambridgesemantics.com/services/DataToolkitView> (${targetGraph})

  {
    ?data a s:FileSource ;
      s:url "string" ;
      [ s:options [ file_storage_connection_options ] ; ]
      [ s:pattern "string" ; ]
      [ s:maxDepth int ; ]
  }
}
```

```

[ s:format [ source_format_options ; ] ; ]
[ s:mimetype "string" ; ]
[ s:username "string" ; ]
[ s:password "string" ; ]
[ s:timeout int ; ]
[ s:batching boolean | int ; ]
[ s:paging [ pagination_options ; ]
[ s:concurrency int | [ list_of_properties ] ; ]
[ s:rate int | "string" ; ]
[ s:locale "string" ; ]
[ s:sampling int ; ]
[ s:selector "string" | [ list ] ; ]
[ s:model "string" ; ]
[ s:key ("string") ; ]
[ s:reference [ s:model "string" ; s:using ("string") ]
[ s:formats [ datatype_formatting_options ] ; ]
[ s:normalize boolean | [ normalization_rules ] ; ]
[ s:count ?variable ; ]
[ s:offset int ; ]
[ s:limit int ; ]
# Mapping variables and hierarchical bindings
?mapping_variable ( [ "binding" ] [ datatype ] [ "datetime_format" ] ) ;
... ;
.
# Additional clauses such as BIND, VALUES, FILTER
}
}

```

Note

For readability, the parameters below exclude the base URI

<<http://cambridgesemantics.com/ontologies/DataToolkit#>> as well as the `s:` prefix. As shown in the examples, however, the `s:` prefix or full property URI does need to be included in queries.

Option	Type	Description
PREFIX Clause	N/A	The PREFIX clause declares the standard and custom prefixes for GDI service queries. Generally, queries include the prefixes from the query template (or a subset of them) plus any data-specific

Option	Type	Description
		declarations.
Result Clause	N/A	The result clause defines the type of SPARQL query to run and the set of results to return, i.e., whether you want to read (SELECT or CONSTRUCT) from the source or ingest the data into Graph Lakehouse (INSERT).
GRAPH \${targetGraph}	N/A	Include the GRAPH keyword and target graph parameter \${targetGraph} when writing an INSERT query to ingest data into a graphmart. Graph Lakehouse automatically populates the query with the appropriate target URIs when the query runs.
\${usingSources}	N/A	Include the source graph parameter \${usingSources} when writing a "topdown" query that passes values from the data that is in the graphmart to the data source. Graph Lakehouse automatically populates the query with the appropriate FROM clauses when the query runs. When passing literal values to the remote source, you do not need to include the source graph parameter. The SERVICE Clause description below includes more information about passing input to data sources.
SERVICE Clause	N/A	Include the SERVICE call <code>SERVICE [TOPDOWN] <http://cambridgesemantics.com/services/DataToolkit></code> to invoke the GDI service when you are running a SELECT, INSERT, or CONSTRUCT query that is not creating a view. When creating a view, use the <code>DataToolkitView</code> service call, as described below in View SERVICE Clause. Include the optional TOPDOWN keyword when you want to pass input values from Graph Lakehouse to the data source. When you include TOPDOWN in the service call, it indicates that the rest of the query produces values to send to the source. In this

Option	Type	Description
		case, the GDI makes repeated calls to pass in each of the specified values and retrieve the data that is based on those values.
View SERVICE Clause	N/A	When writing a CONSTRUCT query that creates a view of the data, include the following SERVICE call: <code>SERVICE <http://cambridgesemantics.com/services/DataToolkitView>(<target_graph>)</code> . Using the <code>DataToolkitView</code> call optimizes query execution because it tells the GDI to inspect the query and determine which filters to push to the data source. It also limits the result set and retrieves only the data that is needed, i.e., the source data is fully mapped but all of the mapped data is not necessarily returned.
url	string	This property specifies the file system location of the source file or directory of files. When specifying a directory (such as <code>s:url "/opt/shared-files/loads/"</code>), the GDI loads all of the file formats it recognizes. To specify a directory but limit the number or type of files that are read, you can include the pattern and/or maxDepth properties.
options	RDF list	If additional connection information needs to be provided to access the file storage system, include the <code>options</code> property to list any storage-specific connection parameters. See File Storage Connection Options for information about the supported properties for each storage type.
pattern	string	This property is used to specify a wildcard pattern for matching file names. For example, <code>s:pattern "common_prefix*.xml"</code> . You can include one <code>s:pattern</code> property per <code>FileSource</code> . The GDI supports Unix file globbing syntax outside of parentheses. Within parentheses, full Java regular expression language is

Option	Type	Description
		supported. For example, including <code>s:pattern "data/**/customer_*.xml"</code> tells the GDI to load all files that match the pattern "customer_*.xml" from any number of subdirectories under the <code>data</code> directory. Similarly <code>s:pattern "(\\d+)/transaction_*.xml"</code> tells the GDI to load all files that match the pattern "transaction_*.xml" in all subdirectories.
maxDepth	int	This property can be used to limit the directory traversal depth. By default, when <code>s:url</code> specifies a directory (and a <code>s:pattern</code> that limits that traversal depth is not specified), all subdirectories are processed. To process only the files in the top level directory, set <code>maxDepth</code> to 0 (<code>s:maxDepth 0</code>). To process the files in the top level directory plus the first-level subdirectories, set <code>maxDepth</code> to 1 (<code>s:maxDepth 1</code>), and so on.
format	RDF list	You can include the <code>format</code> property to add parameters that describe the source files. See File Source Format Options for details about the supported parameters.
mimetype	string	This property can be included to specify the MIME type of the data.
username	string	If authentication is required to access the source, include this property to specify the user name.
password	string	This property lists the password for the given username.
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to

Option	Type	Description
		5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
paging	RDF list	This property can be used to configure paging so that the GDI can access large amounts of data across a number of smaller requests. For details about the <code>paging</code> property, see Pagination Options .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code>. If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with <code>limit</code>, <code>nodes</code>, and/or <code>executorsPerNode</code> properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the

Option	Type	Description
		rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code>, the GDI would use 3 executors with a rate delay of 750ms between requests.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
sampling	int	This property can be used to configure the number of records in the source to examine for data type inferencing.
selector	string or RDF list	This property can be used for XML path extraction to traverse nested structures and target specific data. For example, <code>s:selector "projects"</code> targets the <code>projects</code> class of data. To express a hierarchy, use dot notation. For example, <code>s:selector "region.state.city"</code> navigates a hierarchy to target <code>city</code> data. For more information about binding components

Option	Type	Description
		and the selector property, see Hierarchical Bindings and Arrays below.
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. For details about the <code>formats</code> property, see Data Type Formatting Options .
normalize	RDF list	To give users control over the labels and URIs that are generated,

Option	Type	Description
		the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code> .
offset	int	This property can be used to offset the data that is returned by a number of rows.
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
mapping_variable	variable	The mapping variables, in <code>?mapping_variable (["binding"] [datatype] ["datetime_format"])</code> format, define the triple patterns to output. When the specified <code>?variable</code> matches the source column name, the GDI uses the variable as the source data selector. If you specify an alternate variable name, a binding needs to be specified to map the new variable to the source. You also have the option to transform the data using the datatype and datetime_format options. Tip See Hierarchical Bindings and Arrays below for more information about configuring mapping variables and unpacking JSON files with nested objects and arrays.
binding	string	The <code>binding</code> is a literal value that binds a ?mapping_variable to a source column. If you specify a <code>?variable</code> that matches the source

Option	Type	Description
		column name, then that variable name is the data selector and it is not necessary to specify a binding. If you specify an alternate variable name or there is a hierarchical path to the source column, then the binding is needed to map the new variable to that source column. For example for CSV, the following pattern simply binds the source column AIRLINE to the lowercase variable ?airline: <pre>?airline ("AIRLINE").</pre> Note For FileSource, periods (.), forward slashes (/), and brackets ([]) are parsed as path notation. Therefore, if a source column name includes any of those characters they must be escaped in the binding. Use two backslashes (\\) as an escape character. For example, if a column name is average/day, the variable and binding pattern could be written as ?averagePerDay ("average\\/day").
datatype	URI	The <code>datatype</code> is the data type to convert the column to. If you do not specify a data type, the GDI infers the type. The GDI supports the following types: <pre>xsd:int, xsd:long, xsd:float, xsd:double, xsd:boolean, xsd:time, xsd:dateTime, xsd:date, xsd:duration, xsd:dayTimeDuration, xsd:yearMonthDuration, xsd:gMonthDay, xsd:gMonth, xsd:gYearMonth, xsd:anyURI</pre>
datetime_	string	This option is used to specify the format to use for date and time

Option	Type	Description
format		data types. The GDI supports Java date and time formats. Specify days as "d," months as "M," and years as "y." For the time, specify "H" for hours, "m" for minutes, and "s" for seconds. For example, "yyyyMMdd HH:mm:ss" or "ddMMMy" to display date values such as "01JAN19." Note The GDI's default base year is 2000. If the source data has years with only two digits, such as 02-04-99, the GDI prepends 20 to the digits. The value 02-04-99 is parsed to 02-04-2099. To specify an alternate base year to use for two-digit values, you can include the notation ^nnnn (e.g., ^1900) in the format value. For example, to set the base year to 1900 instead of 2000, use a format value such as <code>xsd:date "dd-MMM-yy^1900"</code> or <code>xsd:date "dd-MMM-yy^1990"</code>. When one of those values is specified, 02-04-99 is parsed to 02-04-1999.

Hierarchical Bindings and Arrays

When configuring the mapping variables in a query, the GDI provides syntax for unpacking XML files with nested objects and arrays. One way to express hierarchies in queries is to use brackets ([]) to group objects into binding trees. For example, the WHERE clause snippet below organizes mapping variable objects into an `hourly/data` hierarchy by nesting the `?data` patterns inside the `?hourly [ ]` tree:

```
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource;
    s:url "/mnt/data/xml/weather.xml" ;
    ?latitude (xsd:double) ;
```

```

        ?longitude (xsd:double) ;
        ?timezone (xsd:string) ;
        ?hourly
    [
        ?data
    [
        ?time (xsd:long) ;
        ?rainProbability ("precipProbability" xsd:double) ;
        ?temperature (xsd:double) ;
        ?feelsLike ("apparentTemperature" xsd:double) ;
        ?windSpeed (xsd:double) ;
    ] ;
    ] .
    }
}

```

When constructing object binding trees, if you choose to introduce the hierarchy with a variable name that is not an exact match to the source label, include a **selector** property to list the value from the source. For example, in the WHERE clause snippet below, `s:selector` is included to select `eventHeader` in the source as `?event` in the query and `statLocation` as `?location`.

```

WHERE
{
    SERVICE <http://cambridgesemantics.com/services/DataToolkit>
    {
        ?data a s:FileSource ;
        s:url "/mnt/data/xml/part_1.xml" ;
        ?event
    [
        s:selector "eventHeader" ;
        ?eventId (xsd:string) ;
        ?eventName (xsd:string) ;
        ?eventVersion (xsd:string) ;
        ?eventTime (xsd:dateTime) ;
    ] ;
        ?location
    [
        s:selector "statLocation" ;
        ?locationId (xsd:string) ;
        ?lineNo (xsd:int) ;
        ?statNo (xsd:int) ;
        ?statId (xsd:int) ;
    ] ;
    }
}

```

```

    ] .
  }
}

```

As an alternative to grouping objects in binding trees, the **selector** property also supports using dot notation to specify paths. For example, the WHERE clause snippet below rewrites the first example query to express the same `hourly/data` hierarchy as a path in the `s:selector` value:

```

WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:FileSource;
    s:url "/mnt/data/xml/weather.xml" ;
    ?latitude (xsd:double) ;
    ?longitude (xsd:double) ;
    ?timezone (xsd:string) ;
    s:selector: "hourly.data" ;
    ?time (xsd:long) ;
    ?rainProbability ("precipProbability" xsd:double) ;
    ?temperature (xsd:double) ;
    ?feelsLike ("apparentTemperature" xsd:double) ;
    ?windSpeed (xsd:double) .
  }
}

```

Query Examples

The following example query ingests data from an XML file that contains hierarchies.

```

PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX fmcsa: <http://census.gov/ontologies/FMCSA#>

INSERT {
  GRAPH <http://anzograph.com/define> {
    ?s ?p ?o
  }
}

WHERE {
  SERVICE <http://cambridgesemantics.com/services/DataToolkit> {

```

```

?data a s:FileSource ;
  s:url "file:///opt/shared/data/xml/define.xml" ;

  ?ItemGroupDef [
    ?OID (xsd:string) ;
    ?Name (xsd:string) ;
    ?Repeating (xsd:string) ;
    ?IsReferenceData (xsd:string) ;
    ?Purpose (xsd:string) ;
    ?Label (xsd:string) ;
    ?Structure (xsd:string) ;
    ?DomainKeys (xsd:string) ;
    ?Class (xsd:string) ;
    ?ArchiveLocationID (xsd:string) ;
    ?Comment (xsd:string) ;
    ?ItemRef [
      ?ItemOID (xsd:string) ;
      ?OrderNumber (xsd:int) ;
      ?Mandatory (xsd:string) ;
    ] ;
  ] .
}

```

File Source Format Options

For file sources, you can include the **format** property to list additional parameters that describe the source. The supported format parameters are described below.

```

s:format [
  s:delimiter "string" ;
  s:headers boolean ;
  s:columns "string" ;
  s:start int ;
  s:skip int ;
  s:comment "string" ;
  s:quote "string" ;
  s:escape "string" ;
  s:maxColumns int ;
  s:segment boolean ;
] ;

```

Option	Type	Description
delimiter	string	This property specifies the string that is used to delimit columns in the file(s). For example, <code>s:delimiter " "</code> .
headers	boolean	This property indicates whether or not the file(s) include headers. By default the headers value is true (<code>s:headers true</code>). For files that do not have headers, specify <code>s:headers false</code> .
columns	string	If you want the GDI to target only certain columns in the source file(s), you can include the columns property to list the names of columns to include. The value is a single string that is a comma-separated list. For example, <code>s:columns "employee_id, name, address, start date, title"</code> .
start	int	If the file includes headers that take up more than one row, include the <code>start</code> property to specify the row number where the data starts to exclude headers. For example, <code>s:start 8</code> .
skip	int	This property can be used to specify the number of rows/records to skip before reading or ingesting the file(s). By default, skip is set to 0 (<code>s:skip 0</code>).
comment	string	This property specifies the string that is used as the comment character in the file(s). The comment value is set to # by default (<code>s:comment "#"</code>).
quote	string	This property is used to specify the string that is used as the quote character.
escape	string	This property is used to specify the escape string that is used in the file(s). For example, <code>s:escape "\"</code> .

Option	Type	Description
maxColumns	int	This property can be used to set a limit on the maximum number of columns to read or ingest. The maxColumns property is set to -1 (unlimited) by default (<code>s:maxColumns -1</code>).
segment	boolean	This property indicates whether or not the file(s) can be segmented. For example, some CSV files that contain embedded newlines cannot be segmented. By default, segment is not set.

File Storage Connection Options

If you are querying a **FileSource** and additional connection information needs to be provided to access the file storage system, include the **options** property in the query and define the necessary storage-specific connection parameters. The parameters that the GDI supports for each type of storage system are pulled directly from the Java API for that system. The supported properties for each storage type are listed below.

- [Amazon S3](#)
- [FTP & FTPS](#)
- [Google Cloud Storage](#)
- [HDFS](#)
- [SFTP](#)
- [WebDAV](#)

Amazon S3

```
s:options [
  s:accessKey "string" ;
  s:region "string" ;
  s:secretKey "string" ;
  s:serviceName "string" ;
  s:sessionToken "string" ;
  s:createBucket boolean ;
```

```
s:disableChunkedEncoding boolean ;
s:serverSideEncryption boolean ;
s:useHttps boolean ;
] ;
```

Option	Type	Description
accessKey	string	The <code>accessKey</code> property can be included to specify the access key.
region	string	The <code>region</code> property can be included to specify the region.
secretKey	string	The <code>secretKey</code> property can be included to specify the secret key.
serviceName	string	For connections to AWS service endpoints, the <code>serviceName</code> property can be included to specify the service name.
sessionToken	string	The <code>sessionToken</code> property can be included to specify the session token.
createBucket	boolean	Refer to the S3 API documentation.
disableChunkedEncoding	boolean	For increased performance, Amazon S3 requests use chunked encoding by default. To disable chunked encoding, you can include <code>s:disableChunkedEncoding true</code> in the query.
serverSideEncryption	boolean	Refer to the S3 API documentation.
useHttps	boolean	Refer to the S3 API documentation.

FTP & FTPS

```
s:options [  
  s:autodetectUtf8 boolean ;  
  s:connectTimeout int ;  
  s:controlEncoding "string" ;  
  s:dataTimeout int ;  
  s:defaultDateFormat "string" ;  
  s:entryParser "string" ;  
  s:fileType "string" ;  
  s:passiveMode boolean ;  
  s:proxy "string" ;  
  s:recentDateFormat "string" ;  
  s:remoteVerification boolean ;  
  s:serverLanguageCode "string" ;  
  s:serverTimeZoneId "string" ;  
  s:shortMonthNames "string" ;  
  s:socketTimeout int ;  
  s:userDirIsRoot boolean ;  
  s:dataChannelProtectionLevel "string" ;  
  s:ftpsMode "string" ;  
  s:keyManager "string" ;  
  s:trustManager "string" ;  
] ;
```

Option	Type	Description
autodetectUtf8	boolean	For FTP connections, the <code>autodetectUtf8</code> property can be included to indicate whether the FTP server is set to UTF-8 mode or Auto-detect encoding.
connectTimeout	int	For FTP connections, you can include the <code>connectTimeout</code> property to specify the maximum number of seconds to hold a connection before timing out.
controlEncoding	string	Refer to the FTP API documentation.

Option	Type	Description
dataTimeout	int	For FTP connections, you can include the <code>dataTimeout</code> property to specify the maximum number of seconds to transfer data before timing out.
defaultDateFormat	string	Refer to the FTP API documentation.
entryParser	string	Refer to the FTP API documentation.
fileType	string	Refer to the FTP API documentation.
passiveMode	boolean	For FTP connections, the <code>passiveMode</code> property can be included to indicate whether the data transfer mode is passive or active. If you use passive mode, set <code>passiveMode</code> to <code>true</code> (<code>s:passiveMode true</code>).
proxy	string	If you are using an FTP proxy, include the <code>proxy</code> property to specify the proxy connection details.
recentDateFormat	string	Refer to the FTP API documentation.
remoteVerification	boolean	For FTP connections, the <code>remoteVerification</code> property can be included to indicate whether remote authentication is enabled. If you use remote authentication, set <code>remoteVerification</code> to <code>true</code> (<code>s:remoteVerification true</code>).
serverLanguageCode	string	If the FTP server language is not set to English, include the <code>serverLanguageCode</code> property

Option	Type	Description
		to specify the language code for the server. For example, <code>s:serverLanguageCode "ES"</code> .
serverTimeZoneId	string	For FTP connections, the <code>serverTimeZoneId</code> property can be included to specify the timezone ID for the server.
shortMonthNames	string	Refer to the FTP API documentation.
socketTimeout	int	For FTP connections, you can include the <code>socketTimeout</code> property to specify the maximum number of seconds to transfer data before timing out.
userDirIsRoot	boolean	Refer to the FTP API documentation.
dataChannelProtectionLevel	string	For FTPS connections, the <code>dataChannelProtectionLevel</code> property specifies the Data Channel Protection Level for the server.
ftpsMode	string	For FTPS connections, the <code>ftpsMode</code> property specifies whether the FTPS is in implicit or explicit mode.
keyManager	string	For FTPS connections, the <code>keyManager</code> property specifies the KeyManager value for making an SSL connection to the server.
trustManager	string	For FTPS connections, the <code>trustManager</code> property specifies the TrustManager value for the SSL connection to the server.

Google Cloud Storage

```
s:options [  
  s:serviceAccountKey "string" ;  
] ;
```

Option	Type	Description
serviceAccountKey	string	For connections to GCS, the <code>serviceAccountKey</code> property can be included to specify the key for the service account.

HDFS

```
s:options [  
  s:configName "string" ;  
  s:configPath "string" ;  
  s:configURL "string" ;  
] ;
```

Option	Type	Description
configName	string	For connections to HDFS, the <code>configName</code> property can be included to specify the name of the configuration file to read.
configPath	string	For connections to HDFS, the <code>configPath</code> property can be included to list the path to the specified configuration file.
configURL	string	Refer to the HDFS API documentation.

SFTP

```
s:options [  
  s:compression "string" ;  
  s:configRepository "string" ;  
  s:fileNameEncoding "string" ;  
  s:identityProvider "string" ;  
  s:identityRepositoryFactory "string" ;  
  s:keyExchangeAlgorithm "string" ;  
]
```

```

s:knownHosts "string" ;
s:loadOpenSSHConfig boolean ;
s:preferredAuthentications "string" ;
s:sessionTimeout int ;
s:strictHostKeyChecking "string" ;
s:userInfo "string" ;
] ;

```

Option	Type	Description
compression	string	Refer to the SFTP API documentation.
configRepository	string	Refer to the SFTP API documentation.
fileNameEncoding	string	Refer to the SFTP API documentation.
identityProvider	string	Refer to the SFTP API documentation.
identityRepositoryFactory	string	Refer to the SFTP API documentation.
keyExchangeAlgorithm	string	For SFTP connections, you can include the <code>keyExchangeAlgorithm</code> property to specify the key exchange algorithm to use.
knownHosts	string	Refer to the SFTP API documentation.
loadOpenSSHConfig	boolean	For SFTP connections, the <code>loadOpenSSHConfig</code> property indicates whether to read the <code>~/ .ssh/config</code> file.
preferredAuthentications	string	For SFTP connections, the <code>preferredAuthentications</code> property can be included to specify the authentication order to use.

Option	Type	Description
sessionTimeout	int	For SFTP connections, you can include the <code>sessionTimeout</code> property to specify the maximum number of seconds to leave the session open before timing out.
strictHostKeyChecking	string	For SFTP connections, you can include the <code>strictHostKeyChecking</code> property to specify how host keys are checked.
userInfo	string	Refer to the SFTP API documentation.

WebDAV

```
s:options [
  s:creatorName "string" ;
  s:versioning boolean ;
] ;
```

Option	Type	Description
creatorName	string	For WebDAV connections, the <code>creatorName</code> property can be included to add a description of the creator of the resource.
versioning	boolean	Refer to the WebDAV API documentation.

GDI Property Reference

This topic describes the Graph Data Interface (GDI) properties that are available to use in queries. The first section describes the options that are available regardless of data source type, and the subsequent sections describe the source-specific options.

- [Universal Properties](#)
- [DbSource Properties](#)
- [FileSource Properties](#)
- [HttpSource Properties](#)
- [ElasticSource Properties](#)

Universal Properties

The table below lists the properties that are valid in queries against all data source types.

Option	Type	Description
batching	boolean or int	This property can be used to disable batching, or it can be used to change the default the batch size. By default, batching is set to 5000 (<code>s:batching 5000</code>). To disable batching, you can include <code>s:batching false</code> in the query. Typically users do not change the batching size. However, it can be useful to control the batch size when performing updates. To configure the size, include <code>s:batching int</code> in the query. For example, <code>s:batching 3000</code> .
concurrency	int or RDF list	This property can be included to configure the maximum level of concurrency for the query. The value can be an integer, such as <code>s:concurrency 8</code> . If the value is an integer, it configures a maximum limit on the number of slices that can execute the query. For finer-grained control over the number of nodes and slices to use, concurrency can also be included as an object with

Option	Type	Description
		limit, nodes, and/or executorsPerNode properties. For example, the following object configures a concurrency model that allows a maximum of 24 executors distributed across 4 nodes with 8 executors per node: <pre>s:concurrency [s:limit 24 ; s:nodes 4 ; s:executorsPerNode 8 ;] ;</pre>
count	variable	If you want to turn the query into a COUNT query, you can include this property with a <code>?variable</code> to perform a count. For example, <code>s:count ?count</code>.
errors	boolean	Controls whether the GDI ignores errors (such as query or file errors) or stops processing the query when an error is encountered. This property is set to <code>true</code> by default (<code>s:errors true</code>). Processing stops when an error is encountered. To ignore errors, you can include <code>s:errors false</code>.
formats	RDF list	To give users control over the data types that are used when coercing strings to other types, this property can be included in GDI queries to define the desired types. In addition, it can be used to describe the formats of date and time values in the source to ensure that they are recognized and parsed to the appropriate date, time, and/or dateTime values. For details about the <code>formats</code> property, see Data Type Formatting Options.
key	string	This property can be used to define the primary key column for the source file or table. This column is leveraged in a resource

Option	Type	Description
		template for the instances that are created from the source. For example, <code>s:key ("EMPLOYEE_ID")</code> . For more information about <code>key</code> , see Data Linking Options .
limit	int	You can include this property to limit the number of results that are returned. <code>s:limit</code> maps to the SPARQL LIMIT clause.
locale	string	This property can be used to specify the locale to use when parsing locale-dependent data such as numbers, dates, and times.
model	string	This property defines the class (or table) name for the type of data that is generated from the specified data source. For example, <code>s:model "employees"</code> . Model is optional when querying a single source. If your query targets multiple sources, however, and you want to define resource templates (primary keys) and object properties (foreign keys), you must specify the model value for each source.
normalize	boolean and/or RDF list	To give users control over the labels and URIs that are generated, the GDI offers several options for normalizing the model and/or the fields that are created from the specified data source(s). For details about the <code>normalize</code> property, see Model Normalization Options .
offset	int	This property can be used to offset the data that is returned by a number of rows.
paging	RDF list	This property can be used to configure paging so that the GDI can access large amounts of data across a number of smaller requests. For details about the <code>paging</code> property, see Pagination Options .

Option	Type	Description
password	string	This property lists the password for the given username.
rate	int or string	This property can be included to control the frequency with which a request is sent to the source. The limit applies to the number of requests a single slice can make. If you specify an integer for the rate, then the value is treated as the maximum number of requests to issue per minute. If you specify a string, you have more flexibility in configuring the rate. The sample values below show the types of values that are supported: <pre>s:rate "90/minute" ; s:rate "90 per minute" ; s:rate "200000 every week" ; s:rate "10000 every 6 hours" ;</pre> To enforce the rate limit, the GDI introduces a sleep between requests that is equal to the rate delay. The more executing slices, the longer the rate delay needs to be to enforce the limit in aggregate. Given the example of <code>s:rate "90/minute"</code>, the GDI would optimize the concurrency and only use 1 slice for execution with a rate delay of 666ms between requests. If <code>s:rate "240/minute"</code>, the GDI would use 3 executors with a rate delay of 750ms between requests.
reference	RDF list	This property can be used to specify a foreign key column. The reference property is an RDF list that includes the <code>model</code> property to list the target table and a <code>using</code> property that defines the foreign key column. For more information about <code>reference</code> , see Data Linking Options .
sampling	int	This property can be used to configure the number of records in

Option	Type	Description
		the source to examine for data type inferencing.
selector	string or RDF list	This property can be used as a binding component to identify the path to the source objects. For example, <code>s:selector "Sales.SalesOrderHeader"</code> targets the SalesOrderHeader table in the Sales schema. For more information about binding components and the selector property, see Using Binding Trees and Selector Paths .
strict	boolean	This property can be used to force the GDI to limit the data to strictly what is stated in the query. For example, when ingesting data from a CSV file, you can include <code>s:strict true</code> on the <code>s:FileSource</code> to ensure that the GDI only ingests columns for which a variable binding exists in the query. In addition, this property can be included in <code>s:formats</code> to control the automatic data type conversion feature (as described in Data Type Formatting Options). The default value is <code>false</code> .
timeout	int	This property can be used to specify the timeout (in milliseconds) to use for requests against the source. For example, <code>s:timeout 5000</code> configures a 5 second timeout.
url	string	This property specifies the URL for the data source, such as the database URL, Elasticsearch URL, or HTTP endpoint URL. For file-based sources, the <code>url</code> property specifies the file system location of the source file or directory of files. When specifying a directory (such as <code>s:url "/opt/shared-files/loads/"</code>), the GDI loads all of the file formats it recognizes. To specify a directory but limit the number or type of files that are read, you can include the pattern and/or maxDepth properties. Important

Option	Type	Description
		For security, it is a best practice to reference connection information (such as the url, username, and password) from a Query Context so that the sensitive details are abstracted from any requests. In addition, using a Query Context makes connection details reusable across queries. See Use a Query Context for more information. For example, the triple patterns below reference a Query Context and add a JDBC driver level connection property: <pre>?data a s:DbSource ; s:url "{{@Somedb.url}}" ; s:username "{{@Somedb.user}}" ; s:password "{{@Somedb.password}}" ; s:property [s:name "access" ; s:value "all"]</pre>
username	string	If authentication is required to access the source, include this property to specify the user name.

DbSource Properties

The table below lists the properties that are available for queries against database data sources. For more information about database sources, see [Query a Database Source](#).

Option	Type	Description
database	string	This property can be used to specify the database to target in the source if the database is not listed in the <code>s:url</code> or <code>s:selector</code> strings.

Option	Type	Description
driver	string	This property can be included to specify the JDBC driver to use.
orderBy	string, variable, list	You can include this property to order the result set by a field name, a bound variable, or a list of names or bound variables.
maxConnections	int	This property can be used to set a limit on the maximum number of active connections to the source. For example, <code>s:maxConnections 16</code> sets the limit to 16 connections. When not specified, the default value is 10.
partitionBy	string, variable, list	The GDI attempts to partition queries automatically across the available cores (slices) in Graph Lakehouse. To determine how to partition the query, the GDI uses metadata from the source database. It looks for any column in an index, preferring the primary key column if it is interpolable. However, it only considers the first column in any index on the table. After determining the partition column, the GDI does a MIN/MAX on the column as well as a basic sizing query. To specify which column or columns the GDI should partition on, you can include the <code>partitionBy</code> property in the query. The property supports a list of source field names, bound variables, or the object <code>s:auto</code> , which forces the GDI to partition the data when the source does not define partitioning metadata.
property	RDF list	This property can be included to list any JDBC driver-specific connection properties. To incorporate <code>property</code> , use the following syntax: <code>s:property [</code>

Option	Type	Description
		<pre>s:name "custom_driver_property_name" ; s:value "custom_value"]</pre>
query	string	If you want to access the source data by running an SQL query, you can include this property to specify the query string to run. The language does not have to be SQL if the source supports another language. However, some GDI features where the query is dynamically altered may not work with a non-SQL language. Including <code>{{?variable}}</code> substitutions is supported within <code>s:query</code> strings. Important If you include <code>s:query</code>, you must also specify <code>table</code> and <code>partitionBy</code>. Specify the table name in <code>s:table</code> and the column to partition the table on in <code>s:partitionBy</code>. If the table and partition column are not specified, the GDI will not partition the query and query execution may fail or perform very poorly.
schema	string	This property can be included to specify the target schema to query. If you include <code>s:schema "schema_name"</code> without specifying <code>s:table</code> (described below) or <code>s:query</code>, all tables in the schema are queried.
table	string	This property can be included to specify the target table or tables for the query.

FileSource Properties

The table below lists the properties that are available for queries against file-based data sources. For more information about file sources, see [Query a File Source](#).

Option	Type	Description
format	RDF list	You can include the <code>format</code> property to add parameters that describe the source files. See File Source Format Options for details about the supported parameters.
maxDepth	int	This property can be used to limit the directory traversal depth. By default, when <code>s:url</code> specifies a directory (and a <code>s:pattern</code> that limits that traversal depth is not specified), all subdirectories are processed. To process only the files in the top level directory, set <code>maxDepth</code> to 0 (<code>s:maxDepth 0</code>). To process the files in the top level directory plus the first-level subdirectories, set <code>maxDepth</code> to 1 (<code>s:maxDepth 1</code>), and so on.
mimetype	string	This property can be included to specify the MIME type of the data. If you are querying TSV files that do not have a <code>.tsv</code> file extension, include the <code>mimetype</code> property with a value of <code>text/tsv</code> (<code>s:mimetype "text/tsv"</code>).
options	RDF list	If additional connection information needs to be provided to access the file storage system, include the <code>options</code> property to list any storage-specific connection parameters. See File Storage Connection Options for information about the supported properties for each storage type.
pattern	string	This property can be used to specify a wildcard pattern for matching file names. For example, <code>s:pattern "common_prefix*.csv"</code> . You can include one <code>s:pattern</code> property per FileSource. The GDI supports Unix file globbing syntax outside of parentheses. Within

Option	Type	Description
		parentheses, full Java regular expression language is supported. For example, including <code>s:pattern "data/**/customer_*.csv"</code> tells the GDI to load all files that match the pattern "customer_*.csv" from any number of subdirectories under the <code>data</code> directory. Similarly <code>s:pattern "(\\d+)/transaction_*.csv"</code> tells the GDI to load all files that match the pattern "transaction_*.csv" in all subdirectories.

HttpSource Properties

The table below lists the properties that are available for queries against HTTP data sources. For more information about HTTP sources, see [Query an HTTP Source](#).

Option	Type	Description
authorization	RDF list	This property specifies the type of authorization to use and the values for authentication. The options are BearerToken, AWSSignature, or BasicAuth. <pre>s:authorization [a s:BearerToken s:AWSSignature s:BasicAuth]</pre>
AWSSignature	RDF list	For authorization to AWS service endpoints, specify this property and include the appropriate authentication properties from the list below: • accessKey: Include this property to specify the AWS access key. • region: Include this property to specify the AWS region. • secretKey: Include this property to specify the AWS secret key. • serviceName: Include this property to specify the

Option	Type	Description
		AWS service name. • sessionToken: Include this property to specify the AWS session token. <pre>s:authorization [a s:AWSSignature ; s:accessKey "string" ; s:region "string" ; s:secretKey "string" ; s:serviceName "string" ; s:sessionToken "string" ;]</pre>
BasicAuth	RDF list	Specify this property when basic authentication is used, and include the username and password properties. <pre>s:authorization [a s:BasicAuth ; s:username "string" ; s:password "string" ;]</pre>
BearerToken	string	Specify this property when a bearer token is used for authentication, and include the token property. <pre>s:authorization [a s:BearerToken ; s:token "string"]</pre>
content	string or RDF list	This property can be included to send content to the source in the body of the request. For example, <code>content</code> can be a SPARQL query, JSON arrays, or a list of key-value pairs. Content can also be configured with an inline object (blank node) that gets translated to JSON. For more information, see Mapping the Content Property to JSON.

Option	Type	Description
contentType	string	Include this property to specify the content type of the body of the request. For example, <code>s:contentType "application/sparql-query"</code> or <code>s:contentType "application/json"</code> .
encoding	string	When targeting a file, you can include this property to specify the character encoding used by the file. The default value is <code>s:encoding "utf8"</code> .
form	RDF list	To send data to the HTTP endpoint, you can use this property to post the data. Form is a list of name-value pairs. When including <code>s:form</code> , you must also include <code>s:contentType "multipart/form-data"</code> . The GDI sends the form object as an <code>application/x-www-form-urlencoded</code> string that contains the specified parameters. See GDI Property Reference below for sample usage.
format	RDF list	If the data is file-based, you can include the <code>format</code> property to add parameters that describe the source. See File Source Format Options for details about the supported parameters.
header	RDF list	You can use this property to specify name-value pairs to include as headers in the request. For example: <pre>s:header [s:name "Accept" ; s:value "application/json"]</pre> If you are creating a view, you can include variables in the <code>s:header</code> list. When another query is run against a view with variables, that query can map the variables through the view by including predicates in the CONSTRUCT clause.
method	string	You can include this property to specify the HTTP method. For

Option	Type	Description
		example, <code>s:method "GET"</code> or <code>s:method "POST"</code> .
mimetype	string	You can include this property to specify the MIME type of the source. For example, <code>s:mimetype "text/html"</code> .
orderBy	string, variable, list	You can include this property to order the result set by a field name, a bound variable, or a list of names or bound variables.
parameter	RDF list	You can include this property to list any URL parameters as name-value pairs. For example, the <code>s:parameter</code> property below adds <code>format</code> to return results in CSV format and the <code>named-graph-uri</code> parameter to target a specific layer in a graphmart. <pre>s:parameter [s:name "format" ; s:value "csv"] , [s:name "named-graph-uri" ; s:value "http://cambridgesemantics.com/Layer/d541..."]</pre> If you are creating a view, you can include variables in the <code>s:parameter</code> list. When another query is run against a view with variables, that query can map the variables through the view by including predicates in the CONSTRUCT clause.
partitionBy	string, variable, list	The GDI attempts to partition queries automatically across the available cores (slices) in Graph Lakehouse. To determine how to partition the query, the GDI uses metadata from the source. It looks for any column in an index, preferring the primary key column if it is interpolable. However, it only considers the first

Option	Type	Description
		column in any index on the table. After determining the partition column, the GDI does a MIN/MAX on the column as well as a basic sizing query. To specify which column or columns the GDI should partition on, you can include the <code>partitionBy</code> property in the query. The property supports a list of source field names, bound variables, or the object <code>s:auto</code> , which forces the GDI to partition the data when the source does not define partitioning metadata.
proxy	string or RDF list	Include this property to specify proxy information if a proxy is used. The value can be a string, such as <code>s:proxy "host_url:port_number"</code> , or an RDF list that includes <code>host</code> and <code>port</code> properties, such as <code>s:proxy [s:host "host_url" ; s:port port_number]</code> .
trust	string	Include this property to set the level of trust for the source's SSL certificate. The value can be either <code>"system"</code> or <code>"all"</code> .

ElasticSource Properties

The table below lists the properties that are available for queries against Elasticsearch data sources. For more information about Elasticsearch sources, see [Query an Elasticsearch Source](#).

Option	Type	Description
aggregations	object	You can include this property to calculate aggregations over the specified bindings. For information about aggregations, see Aggregations in the Elasticsearch documentation.
config	string	To enable you to use explicit mappings, you can include this property to specify the URL to the index configuration file to employ. For example, <code>es:config</code>

Option	Type	Description
		<code>"/opt/shared/elastic/mapping.json".</code>
document	string	This property lists the document(s) to search.
field	string or variable	This property defines the field to operate on. The value can be a string or bound variable.
highlight	RDF list	You can include this property to define how results are highlighted. For information about the available properties, see Highlighting Elasticsearch Results .
html	boolean	This property controls whether to output HTML for highlighted results. Defaults to <code>true</code> .
index	string	This property can be included to specify the index to search.
minScore	float	This property defines the minimum score for matching documents. Documents with a lower score are not included in the search results.
query	string or RDF list	This property defines the query to execute. The value can be a string or a query object that maps to the Elasticsearch Query DSL . To generate the final query, the GDI combines <code>es:query</code> with any filters it can push to the Elasticsearch DSL. For more information about the <code>query</code> property and mapping Elasticsearch filters to SPARQL FILTER clauses, see Query DSL and Filter Mapping .
routing	string	This property can be included to route a document to a specific shard or to limit the search to a particular shard.
searchAfter		You can include this property to define the key values to start

Option	Type	Description
		searching from.
size	int	This property maps to the <code>size</code> parameter in the Elasticsearch Search API and configures the batch size or maximum number of hits to return in a single call. Defaults to <code>10</code> and typically does not need to be changed.
source	boolean or RDF list	This property can be included to specify the source data to include in results. The value can be a boolean, list of fields, or a list of variable bindings. When <code>true</code> , all source data is returned. When <code>false</code> , no source data is returned.
url	string	The Elasticsearch endpoint URL.

Use a Query Context

When accessing data sources that require sensitive connection and authorization information such as keys, tokens, and user credentials, you can create Query Contexts for storing the sensitive information. A Query Context has a number of key-value pairs, such as username, password, and connection URL. Queries can then reference the keys from a context file and the connection values are abstracted from the requests. This topic provides instructions for creating contexts and referring to a context in a query.

- [Creating a Query Context](#)
- [Referencing Context Keys in a Query](#)

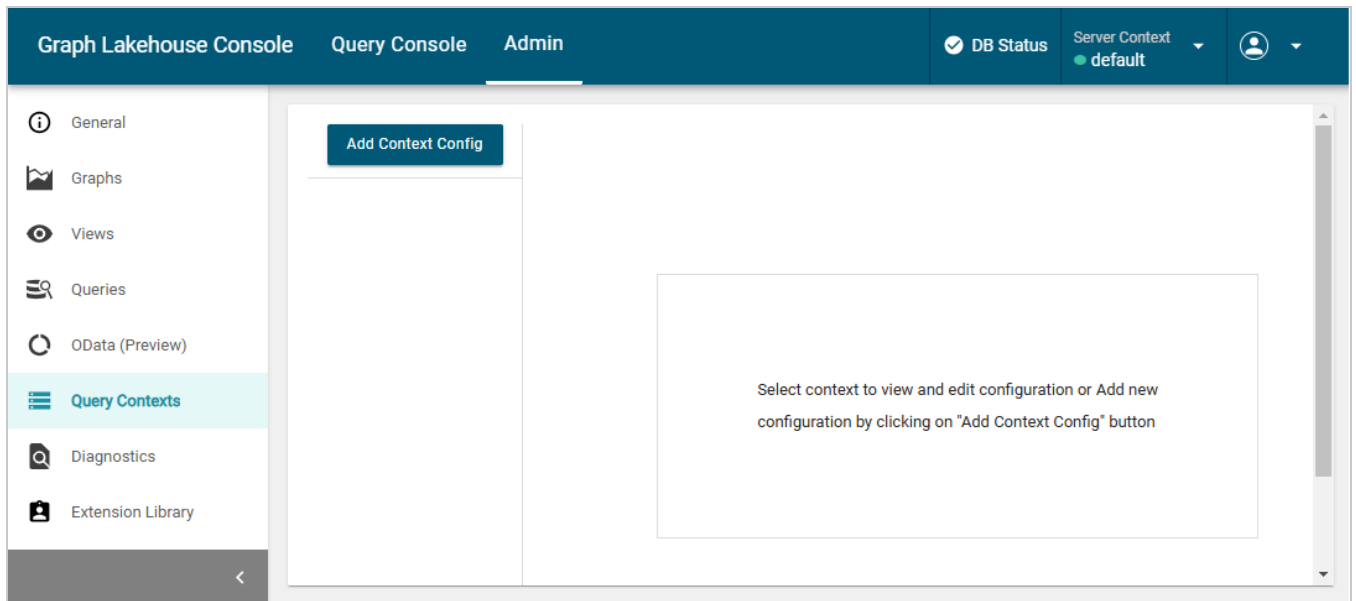
Creating a Query Context

Follow the steps below to create a query context from the user interface.

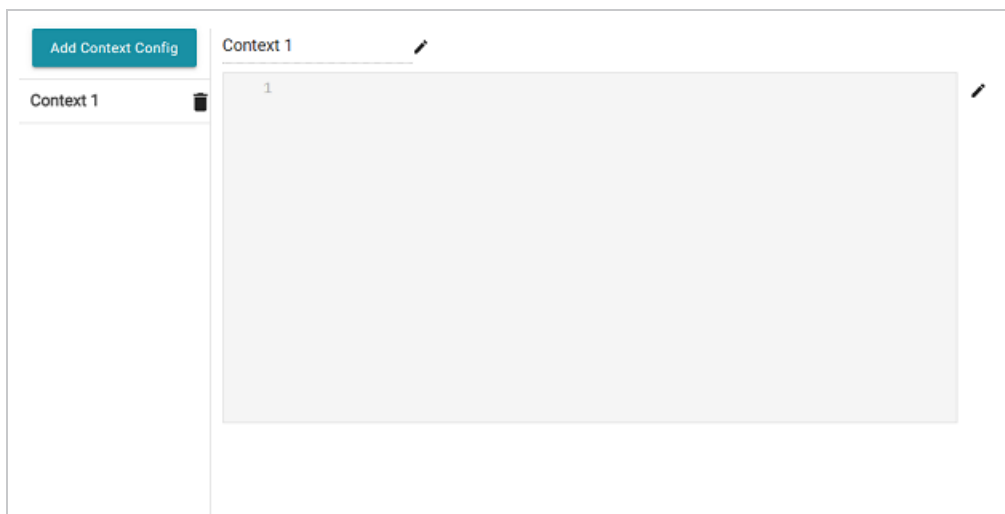
Tip

You can also create a context file in JSON format and save it on the Graph Lakehouse leader server. To reference a context file when using the AZGI command line interface, use the `-context <filename>.json` option.

1. In the Query & Admin Console, click the **Admin** tab. Then click the **Query Contexts** menu item. The Context Configuration screen is displayed.



2. Click the **Add Context Config** button to create a new context. A new context, named **Context N**, is added. For example:



3. At the top of the screen, click the edit icon (✎) next to the context title and specify a name for this context. Queries that connect to this source will use this name when referencing keys in the file. Click **Save** to save the change.
4. Click the edit icon (✎) next to the gray contents field. In the field, specify the appropriate key-value pairs to use to connect to the data source. The contents must be in valid JSON format. For example:

```
{
  "url": "jdbc:mysql://10.111.4.9:3306/NORTHWIND",
  "username": "sysadmin",
  "password": "admin123"
}
```

- When you have finished adding key-value pairs, click the checkmark icon (✓) to save the changes. For example:



- If you want to create additional Query Contexts, click the **Add Context Config** button and repeat the steps above. For details about referencing contexts in queries, see [Referencing Context Keys in a Query](#) below.

Referencing Context Keys in a Query

To reference the keys from a Query Context in a query, you use the following syntax to specify a variable in the object of a triple pattern:

```
"{@context_name:key_name}"
```

Where **context_name** is the title of the context file, and **key_name** is the key whose value should be used as the object. For example, the following query excerpt from a Graph Data Interface (GDI) query refers to the sample context that was created in [Creating a Query Context](#):

```
PREFIX s: <http://cambridgesemantics.com/ontologies/DataToolkit#>
SELECT DISTINCT *
WHERE
{
  SERVICE <http://cambridgesemantics.com/services/DataToolkit>
  {
    ?data a s:DbSource ;
    s:url "{@NorthwindDB:url}" ;
```

```
s:username "{{@NorthwindDB:username}}" ;  
s:password "{{@NorthwindDB:password}}" ;  
...  
}  
}
```

At runtime, the GDI refers to the context file to find the values that are associated with the specified keys. For more information about running GDI queries, see [Getting Started with GDI Queries](#).

Create a Labeled Property Graph (RDF-star)

Graph Lakehouse supports the Labeled Property Graph (LPG) model for adding metadata about the relationships in your graphs. Properties that express values such as start and end dates, data provenance tracking, or the weight, score, or veracity of the data can be added to a graph to further define any of the relationships in the data.

Note

Graph Lakehouse's LPG implementation follows the proposed RDF-star and SPARQL-star extension to the W3C SPARQL query language and RDF data model specifications. The proposal, called [RDF-star and SPARQL-star](#), is a work in progress. The syntax described in the document may not be included in the final specification, and Graph Lakehouse does not support all of the examples included in the proposal at this time.

This topic provides information about loading and inserting properties and querying property graphs.

- [Defining Properties in Turtle Load Files](#)
- [Defining Properties in INSERT Queries](#)
- [Querying Property Graphs](#)

Defining Properties in Turtle Load Files

This section provides information about how to create a property graph by defining relationship properties in a Turtle load file. For instructions on creating properties in INSERT queries, see [Defining Properties in INSERT Queries](#) below.

Note

There is a limit of 255 total property values per edge. Graph Lakehouse returns an `Element larger than allowed - too many properties` error if you attempt to load or insert more than 255 property values for the same relationship.

To define a relationship property in a Turtle file, wrap the triplet in double arrow heads (`<< >>`), and then specify the property URI and value at the end of the triplet:

```
<< <subject> <predicate> <object> >> <property_URI> <property_value> .
```

For example, the TTL file contents below include properties that further define the **like**, **dislike**, and **friend** relationships in the triples. The file adds a **weight** property to define how much `person3` likes or dislikes certain types of events, and the file adds **startDate** and **endDate** properties to `friend` predicates to define the start and end dates of friendships.

Tip

By default, the sample Ticket data set already includes `startDate` and `endDate` properties for the `friend` predicates. The example below defines start and end date properties only for illustrative purposes.

```
@prefix tickit: <http://anzograph.com/tickit/> .
```

```
tickit:person3
  rdf:type tickit:person ;
  tickit:card "4984932249480735"^^xsd:long ;
  tickit:birthday "1963-07-02"^^xsd:date ;
  tickit:ssn 503703220 ;
  tickit::firstname "Lars" ;
  tickit:lastname "Ratliff" ;
  tickit:city "High Point" ;
```

```

tickit:state "NY" ;
tickit:email "amet.faucibus.ut@condimentumegetvolutpat.ca" ;
tickit:phone "(624) 767-2465" .

<< tickit:person3 tickit:like "sports" >> tickit:weight 8 .
<< tickit:person3 tickit:like "rock" >> tickit:weight 9 .
<< tickit:person3 tickit:like "musicals" >> tickit:weight 4 .
<< tickit:person3 tickit:dislike "theatre" >> tickit:weight 5 .
<< tickit:person3 tickit:dislike "jazz" >> tickit:weight 9 .
<< tickit:person3 tickit:dislike "opera" >> tickit:weight 10 .
<< tickit:person3 tickit:friend tickit:person8563 >> tickit:startDate "1990-01-04"^^xsd:date .
<< tickit:person3 tickit:friend tickit:person38436 >> tickit:startDate "2000-04-27"^^xsd:date .
<< tickit:person3 tickit:friend tickit:person11979 >> tickit:startDate "2004-11-09"^^xsd:date .
<< tickit:person3 tickit:friend tickit:person11979 >> tickit:endDate "2012-07-17"^^xsd:date .
    tickit:person3 tickit:friend
tickit:person8639,tickit:person18536,tickit:person42975,tickit:person47376,

tickit:person1692,tickit:person2556,tickit:person11979,tickit:person20860,tickit:person
21259,tickit:person26586,

tickit:person27529,tickit:person31735,tickit:person36264,tickit:person38436,tickit:pers
on42306,tickit:person42975 .

```

The example above contains both compact and long Turtle notation. When defining properties in files, tuples that contain properties must include the complete reference triple (subject, predicate, and object). Properties cannot be added to triples specified in compact notation. In addition, specify one property per triplet. To define multiple properties for the same triplet, list the triplet multiple times. For example, the following lines in the example above define two properties (startDate and endDate) for the person3 friend person11979 triple:

```

<< tickit:person3 tickit:friend tickit:person11979 >> tickit:startDate "2004-11-09"^^xsd:date .
<< tickit:person3 tickit:friend tickit:person11979 >> tickit:endDate "2012-07-17"^^xsd:date .

```

Defining Properties in INSERT Queries

Users can create property graphs using [INSERT](#) and [INSERT DATA](#) syntax to insert triples and properties or add properties to existing triples. To define properties in INSERT statements, use the same syntax as Turtle files: wrap triplets in double arrow heads (<< >>), and then specify the property URI and value for that triple at the end of the triplet.

```
<< <subject> <predicate> <object> >> <property_URI> <property_value> .
```

Note

There is a limit of 255 total property values per edge. Graph Lakehouse returns an `Element larger than allowed - too many properties` error if you attempt to load or insert more than 255 property values for the same relationship.

For example, the INSERT DATA statement below adds weight properties to the like and dislike predicates for person3. This example specifies literal values for weight property.

```
PREFIX tickit: <http://anzograph.com/tickit/>
INSERT DATA { GRAPH <http://anzograph.com/tickit> {
  << tickit:person3 tickit:dislike "jazz" >> tickit:weight 9 .
  << tickit:person3 tickit:dislike "theatre" >> tickit:weight 5 .
  << tickit:person3 tickit:dislike "opera" >> tickit:weight 10 .
  << tickit:person3 tickit:like "sports" >> tickit:weight 8 .
  << tickit:person3 tickit:like "rock" >> tickit:weight 9 .
  << tickit:person3 tickit:like "musicals" >> tickit:weight 4 .
}
```

The following example INSERT statement queries the Tickit graph to find the sellers whose total sales amount is greater than or equal to \$20,000. For each seller who meets the requirement, the INSERT clause inserts an **earned** predicate with a property named **score** and a score value of **10**:

```
PREFIX tickit: <http://anzograph.com/tickit/>
INSERT {GRAPH <http://anzograph.com/tickit> {
  << ?person tickit:earned ?earned >> tickit:score 10
}
}
WHERE {GRAPH <http://anzograph.com/tickit> {
```



```

{ SELECT ?person (SUM(?dollars) AS ?earned)
  WHERE {
    ?person tickit:firstname ?first .
    ?person tickit:lastname ?last .
    ?sale tickit:sellerid ?person .
    ?sale tickit:pricepaid ?dollars .
  }
  GROUP BY ?person
}
FILTER(?earned >= 20000)
}

```

Selecting the newly created triples shows that 52 people met the requirement and were assigned a `<score>` property with a value of 10:

```

PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?person ?earned ?score
FROM <http://anzograph.com/tickit>
WHERE {
  << ?person tickit:earned ?earned >> tickit:score ?score .
}
ORDER BY ?person

```

person	earned	score
http://anzograph.com/tickit/person11168	21036	10
http://anzograph.com/tickit/person1140	32399	10
http://anzograph.com/tickit/person12263	20320	10
http://anzograph.com/tickit/person12646	22194	10
http://anzograph.com/tickit/person13385	28495	10
http://anzograph.com/tickit/person15976	20929	10
http://anzograph.com/tickit/person16008	20515	10
http://anzograph.com/tickit/person16335	20160	10
http://anzograph.com/tickit/person18005	20918	10
http://anzograph.com/tickit/person19231	22636	10
http://anzograph.com/tickit/person19814	20465	10
http://anzograph.com/tickit/person20029	20103	10
http://anzograph.com/tickit/person23635	20265	10
http://anzograph.com/tickit/person2372	27159	10
http://anzograph.com/tickit/person24980	24857	10
http://anzograph.com/tickit/person25433	27653	10
http://anzograph.com/tickit/person26198	21243	10

```
...
52 rows
```

The following example shows how to create properties and assign values based on data that exists in a source file. The data for the example is a CSV file with the following columns and data:

```
Airline,FlightNumber,TailNumber,OriginAirport,DestinationAirport,Distance
AS,98,N407AS,ANC,SEA,1448
AA,2336,N3KUAA,LAX,PBI,2330
US,840,N171US,SFO,CLT,2296
AA,258,N3HYAA,LAX,MIA,2342
AS,135,N527AS,SEA,ANC,1448
DL,806,N3730B,SFO,MSP,1589
NK,612,N635NK,LAS,MSP,1299
US,2013,N584UW,LAX,CLT,2125
```

The example INSERT query for the file above defines the Distance column as a property and adds the Distance value as the value for the property:

```
PREFIX s:    <http://cambridgesemantics.com/ontologies/DataToolkit#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

INSERT { GRAPH <http://anzograph.com/flights> {
    ?OriginIRI a <Airport> .
    ?DestinationIRI a <Airport> .
    << ?OriginIRI <Destination> ?DestinationIRI >> <Distance> ?Distance .
    ?FlightIRI a <Flight> ;
    <Airline> ?Airline ;
    <FlightNumber> ?FlightNumber ;
    <TailNumber> ?TailNumber .
}
}
WHERE {
    SERVICE <http://cambridgesemantics.com/services/DataToolkit> {
        ?data a s:FileSource ;
        s:url "/home/erin/air-lpg.csv" ;
        ?Airline (xsd:string);
        ?FlightNumber (xsd:string);
        ?TailNumber (xsd:string);
        ?OriginAirport (xsd:string);
        ?DestinationAirport (xsd:string);
        ?Distance (xsd:long) .
    }
    BIND(IRI("http://anzograph.com/flights/Flight/{{?FlightNumber}}") as ?FlightIRI)
```

```

    BIND( IRI ("http://anzograph.com/flights/origin/{{?OriginAirport}}") as ?OriginIRI)
    BIND( IRI ("http://anzograph.com/flights/destination/{{?DestinationAirport}}") as
?DestinationIRI)
  }
}

```

The following query returns the origin and destination airports for the flights as well as the distance property value:

```

SELECT ?from ?to ?distance
FROM <http://anzograph.com/flights>
WHERE {
    << ?from ?p ?to >> ?property ?distance
}
ORDER BY DESC(?distance)

```

from	to
distance	
http://anzograph.com/flights/origin/LAX	http://anzograph.com/flights/destination/MIA
2342	
http://anzograph.com/flights/origin/LAX	http://anzograph.com/flights/destination/PBI
2330	
http://anzograph.com/flights/origin/SFO	http://anzograph.com/flights/destination/CLT
2296	
http://anzograph.com/flights/origin/LAX	http://anzograph.com/flights/destination/CLT
2125	
http://anzograph.com/flights/origin/SFO	http://anzograph.com/flights/destination/MSP
1589	
http://anzograph.com/flights/origin/ANC	http://anzograph.com/flights/destination/SEA
1448	
http://anzograph.com/flights/origin/SEA	http://anzograph.com/flights/destination/ANC
1448	
http://anzograph.com/flights/origin/LAS	http://anzograph.com/flights/destination/MSP
1299	

8 rows

Querying Property Graphs

To return properties and their values when analyzing data sets, include the following property graph syntax in graph and triple patterns:

```
<< <subject> <predicate> <object> >> <property_URI> <property_value> .
```

The following example query returns the properties that were defined in the INSERT DATA query above.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT *
FROM <http://anzograph.com/tickit>
WHERE {
  << tickit:person3 ?p ?likes_or_dislikes >> tickit:weight ?value.
  FILTER(?p=tickit:like || ?p=tickit:dislike)
}
ORDER BY ?p
```

p	likes_or_dislikes	value
http://anzograph.com/tickit/dislike	jazz	9
http://anzograph.com/tickit/dislike	opera	10
http://anzograph.com/tickit/dislike	theatre	5
http://anzograph.com/tickit/like	musicals	4
http://anzograph.com/tickit/like	rock	9
http://anzograph.com/tickit/like	sports	8

6 rows

This example returns a list of the properties in the Tickit graph and lists the number of times each property is referenced in the graph. Note that in addition to the properties that were defined above, the results shown below also include the properties that are defined by default in the sample Tickit data set. See [Working with SPARQL and the Tickit Data](#) for instructions on loading the full data set.

```
SELECT ?property (COUNT(?property) AS ?times_used)
FROM <http://anzograph.com/tickit>
WHERE {
  << ?s ?p ?o >> ?property ?value
}
GROUP BY ?property
ORDER BY desc(?times_used)
```

property	times_used
startDate	1445832
score	241949
endDate	144706
http://anzograph.com/tickit/score	52
http://anzograph.com/tickit/weight	6
5 rows	

Return Edges and Vertexes as JSON Objects

Graphs consists of nodes or *vertexes* connected in pairs by relationships or *edges*. Information about vertexes includes labels such as Person or Employee. Information about edges includes the type of edge, such as "knows" or "friend," and properties of an edge, such as a "startDate" or "endDate."

Labeled property graphs can be queried with SPARQL. However, SPARQL does not provide a construct to extract and combine the vertex or edge information as a unit, as might be needed for certain applications. Graph Lakehouse provides the EDGE and VERTEX functions for returning edge and vertex data as a JSON object.

- [Constructing Edges and Vertexes](#)
- [VERTEX Function](#)
- [EDGE Function](#)
- [Examples](#)

Important

Prior to loading the data for which you want to use the VERTEX function, Graph Lakehouse must be configured to register vertex labels as predicates. To configure the system to register vertices as predicates, add the following line to `<install_path>/config/settings.conf` and then restart Graph Lakehouse:

```
auto_predicate=true
```

For more information about changing settings, see [Change System Settings](#).

Constructing Edges and Vertexes

Both the EDGE and VERTEX functions return Blob type objects, in JSON format, that represent the edges and vertexes in a graph, along with all their associated attributes and properties. Following the standard subject-predicate-object representation of triples, the criteria for how various data is handled to generate edges and vertexes is the following:

Vertexes

- The URI in the subject or object position of a triple can be used to construct a vertex.
- The `rdf:type` predicates define the label of vertexes.
- Triples with non-URI object values are treated as vertex properties. Predicates in those triples are used as the property name and the objects are the values of properties. For example:

```
<person1> <age> 20
```

Edges

- Triples where both the subject and object are URIs can be used to construct an edge.
- An edge's property name and property value is obtained from the RDF-star triple for that edge. Non-URI object values are identified as property values and predicates are treated as the property name. For example:

```
<< <person1> <works_at> <Company1> >> <startDate> "2000-04-27"^^xsd:date .
```

Note

You cannot directly pass a vertex or edge constructed by the VERTEX or EDGE functions as a parameter or filter in a query. However you can use them as arguments to other functions or expressions within the same query.

VERTEX Function

The VERTEX function returns labels and properties of nodes or vertexes.

Syntax

```
VERTEX(?URI_variable) as ?variable
```

Where `?URI_variable` is the variable that represents the targeted subject or object URI.

The VERTEX function returns an `<http://anzograph.com/blobtype/vertex>` Blob type object formatted as a JSON string with the following elements:

- **id**: A unique identifier for the vertex in the database.
- **labels**: An array of the labels for the vertex.
- **properties**: A list of the properties that are mapped to the vertex.

For example:

```
{
  "id":4294967405,
  "labels":[
    "Actor",
    "Person"
  ],
  "properties":{
    "born":{
      "type":"typed-literal",
      "datatype":"http://www.w3.org/2001/XMLSchema#int",
      "value":"1956"
    },
    "name":{
      "type":"typed-literal",
      "datatype":"http://www.w3.org/2001/XMLSchema#string",
      "value":"Tom Hanks"
    }
  }
}
```

EDGE Function

The EDGE function returns properties and values of relationships or edges. There are two options for calling the EDGE function, both of which provide a way to create the same edge objects. The first syntax method is more straightforward, however, it requires that the database be in a pristine and unvarying or unaltered state. It may return an error in some cases when there are ongoing transactions. In that case, you can use the second method, which does not require the same pristine state of the database.

- [Syntax 1: No Transactions in Progress](#)
- [Syntax 2: Aggregate \(Database Updates can be Ongoing\)](#)

Syntax 1: No Transactions in Progress

You can use the following syntax to call the EDGE function when the database is at rest—there are no transactions in progress.

```
BIND(EDGE(?start_vertex, ?edge, ?end_vertex) as ?variable)
```

Argument	Data Type	Description
start_vertex	URI as a variable	The variable that represents the vertex at the start of the edge.
edge	URI as a variable	The variable that represents the edge or predicate.
end_vertex	URI as a variable	The variable that represents the vertex at the end of the edge.

Note

Using this syntax, the EDGE function must appear in a BIND clause that immediately follows the triple pattern that specifies the subject, predicate, and object variables for the edge. The WHERE clause should not include any clause other than a simple FILTER on the subject, predicate, or object. For example:

```
SELECT ?acted_in
FROM <Movies>
WHERE {
  ?s a <Actor> .
  {
    ?s ?p ?o .
    BIND (EDGE(?s,?p,?o) as ?acted_in)
  }
  FILTER(?p = <ACTED_IN>)
}
```

The EDGE function returns an `<http://anzograph.com/blobtype/edge>` Blob type object formatted as a JSON string with the following elements:

- **start:** The ID for the starting vertex of the edge.
- **end:** The ID for the end vertex of the edge.
- **type:** The type of edge.
- **properties:** A list of the properties that are mapped to the edge.

For example:

```
{
  "start":114,
  "end":4294967403,
  "type":"ACTED_IN",
  "properties":{
    "roles":{
      "type":"typed-literal",
      "datatype":"http://www.w3.org/2001/XMLSchema#string",
      "value":"Neo"
    }
  }
}
```

Syntax 2: Aggregate (Database Updates can be Ongoing)

You can use the following syntax to call the EDGE function when the database is at rest or when transactions are in progress.

```
EDGE(?start_vertex, ?edge, ?end_vertex, ?edge_property, ?property_value) as ?variable)
...
GROUP BY ?start_vertex ?edge ?end_vertex
```

Argument	Data Type	Description
start_vertex	URI as a variable	The variable that represents the vertex at the start of the edge.
edge	URI as a variable	The variable that represents the edge or predicate.
end_vertex	URI as a	The variable that represents the vertex at the end of the

Argument	Data Type	Description
	variable	edge.
edge_property	URI as a variable	The variable that represents the edge property URI.
property_value	variable	The variable that represents the value of the property.

The query must include a GROUP BY clause that groups on the first three fields. For example:

```
SELECT ?s ?p ?o (EDGE(?s,?p,?o,?pp,?pv) as ?edge)
FROM <Movies>
WHERE {
    ?s ?p ?o .
    OPTIONAL { << ?s ?p ?o >> ?pp ?pv }
    FILTER (?p = <ACTED_IN>)
}
GROUP BY ?s ?p ?o
```

The EDGE function returns an `<http://anzograph.com/blobtype/edge>` Blob type object formatted as a JSON string with the following elements:

- **start:** The ID for the starting vertex of the edge.
- **end:** The ID for the end vertex of the edge.
- **type:** The type of edge.
- **properties:** A list of the properties that are mapped to the edge.

For example:

```
{
  "start":114,
  "end":4294967403,
  "type":"ACTED_IN",
  "properties":{
    "roles":{
      "type":"typed-literal",
```

```

    "datatype": "http://www.w3.org/2001/XMLSchema#string",
    "value": "Neo"
  }
}
}

```

Examples

Tip

Make sure that `auto_predicate=true` is set in `<install_path>/config/settings.conf` before inserting the sample data for the example queries.

The following INSERT DATA query creates a graph named `http://anzograph.com/Movies` that inserts a small sample of data you can use to test the VERTEX and EDGE functions:

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX movies: <http://anzograph.com/Movies/data/>
PREFIX ont: <http://anzograph.com/ontologies/Movies#>
INSERT DATA {
  GRAPH <http://anzograph.com/Movies> {
    # triples for "Tom Hanks" vertex
    movies:TomHanks rdf:type ont:Person .
    movies:TomHanks rdf:type ont:Actor .
    movies:TomHanks ont:name "Tom Hanks" .
    movies:TomHanks ont:born 1956 .
    # triples for "Forrest Gump" vertex
    movies:ForrestGump rdf:type ont:Movie .
    movies:ForrestGump ont:title "Forrest Gump" .
    movies:ForrestGump ont:release 1994 .
    # edge with properties
    <<movies:TomHanks ont:ACTED_IN movies:ForrestGump>> ont:roles "Forrest" .
  }
}

```

VERTEX Example

The following example uses the VERTEX function to return the vertexes for actors defined in the sample Movies graph:

```

SELECT (VERTEX(?s) as ?actor)
FROM <http://anzograph.com/Movies>
WHERE {
    ?s a <http://anzograph.com/ontologies/Movies#Actor> .
}

```

```

{
  "id":4294967435,
  "labels":[
    "http://anzograph.com/ontologies/Movies#Person",
    "http://anzograph.com/ontologies/Movies#Actor"
  ],
  "properties":{
    "http://anzograph.com/ontologies/Movies#name":{
      "type":"typed-literal",
      "datatype":"http://www.w3.org/2001/XMLSchema#string",
      "value":"Tom Hanks"
    },
    "http://anzograph.com/ontologies/Movies#born":{
      "type":"typed-literal",
      "datatype":"http://www.w3.org/2001/XMLSchema#int",
      "value":"1956"
    }
  }
}

```

EDGE Example (Syntax 1)

The following example uses the EDGE function to return information about the ACTED_IN edge:

```

SELECT ?acted_in
FROM <http://anzograph.com/Movies>
WHERE {
    ?s a <http://anzograph.com/ontologies/Movies#Actor> .
    {
        ?s ?p ?o .
        BIND (EDGE(?s,?p,?o) as ?acted_in)
    }
    FILTER(?p = <http://anzograph.com/ontologies/Movies#ACTED_IN>)
}

```

```

{
  "start":4294967435,
  "end":8589934735,

```

```

"type":"http://anzograph.com/ontologies/Movies#ACTED_IN",
"properties":{
  "http://anzograph.com/ontologies/Movies#roles":{
    "type":"typed-literal",
    "datatype":"http://www.w3.org/2001/XMLSchema#string",
    "value":"Forrest"
  }
}
}
}

```

Aggregate EDGE Example (Syntax 2)

The following example uses the aggregate EDGE function to return information about the ACTED_IN edge.

```

SELECT ?s ?p ?o (EDGE(?s,?p,?o,?pp,?pv) as ?edge)
FROM <http://anzograph.com/Movies>
WHERE {
  ?s ?p ?o .
  OPTIONAL { << ?s ?p ?o >> ?pp ?pv }
  FILTER (?p = <http://anzograph.com/ontologies/Movies#ACTED_IN>)
}
GROUP BY ?s ?p ?o

```

s	p	o	edge
http://anzograph.com/Movies/data/TomHanks	http://anzograph.com/ontologies/Movies#ACTED_IN	http://anzograph.com/Movies/data/ForrestGump	{ "start":4294967435,
			"end":8589934735,
			"type":"http://anzograph.com/ontologies/Movies#ACTED_IN",
			"properties":{
			"http://anzograph.com/ontologies/Movies#roles":{

```
"type": "typed-literal",  
  
"datatype": "http://www.w3.org/2001/XMLSchema#string",  
  
"value": "Forrest"}}
```

1 rows

Infer New Data (RDFS+ Inferencing)

Graph Lakehouse includes an inference engine that can create new relationships based on the vocabularies or ontologies in the existing data.

The following example from the [W3C Semantic Web Inference](#) documentation illustrates the inference concept:

A data set might include the relationship `Flipper isA Dolphin`. An ontology might declare that "every Dolphin is also a Mammal." An inference program that understands the notion of "X is also Y" adds the statement `Flipper isA Mammal` to the set of relationships even though it was not specified in the original data.

When Graph Lakehouse creates inferences, it scans the specified graph for any of the RDFS-plus and supported OWL 2 RL ontologies and generates new triples according to the W3C [OWL 2 RL](#) rules, or rules specified with the optional `WITH RULES` clause. This topic provides instructions for generating inferences with Graph Lakehouse and describes the supported inference vocabularies.

- [Generating Inferences](#)
- [Inference Rule Reference](#)
- [Inference Example](#)

Generating Inferences

Graph Lakehouse generates inferences as a batch command. Run the following command to generate inferences from one or more existing graphs:

```
CREATE INFERENCES FROM source_graph1 [ source_graph2 ... ] INTO GRAPH target_graph  
[ WITH RULES 'list_of_rules' ]
```

Where *list_of_rules* in the optional `WITH RULES` clause is any of the following arguments. Specify multiple options in a comma-separated list:

Option	Description
<code>all</code>	Run all rules.
<code>rdfsplus</code>	Run only the RDFS-plus rules.
<i>rule list</i>	List specific rules to run. For a list of available rule names, see Inference Rule Reference below.
<code>-rulename</code>	Specify a hyphen (-) in front of a rule name to exclude that rule. For example, <code>-scm-svf2</code> excludes the <code>scm-svf2</code> rule.

For example, the following WITH RULES clause runs all of the inference rules except **prp-fp** and **prp-ifp**:

```
... WITH RULES 'all,-prp-fp,-prp-ifp'
```

When you run the CREATE INFERENCES command, Graph Lakehouse runs rules for each of the RDFS-plus ontologies that it finds in the source graphs, or rules specified with the optional WITH RULES clause, and inserts the inferred triples into the specified target graph.

Note

Certain inference rules are coupled. Specifying either of the rules in the pair automatically runs the coupled rule. The list below describes the paired rules:

- `scm-dom1` and `scm-rng1`
- `scm-dom2` and `scm-rng2`
- `prp-inv1` and `prp-inv2`

In addition, running `scm-eqc1` and `cax-sco` also runs `cax-eqc1` and `cax-eqc2`. And running `scm-eqp1` and `prp-spo1` also runs `prp-eqp1` and `prp-eqp2`.

Inference Rule Reference

The tables below describe the RDFS-plus rules as well as the additional subset of OWL 2 RL rules that Graph Lakehouse supports.

- [RDFS-Plus Rules](#)
- [OWL 2 RL Rules](#)

RDFS-Plus Rules

The tables below define the RDFS-plus inference rules.

Semantics of Class Axioms

Note

Because **cax-eqc1** and **cax-eqc2** (described in the table below) are implied rules that are coupled with scm-eqc1 and cax-sco, including **cax-eqc1** or **cax-eqc2** in the WITH RULES clause will result in an `invalid inference rule name` error. To run the **cax-eqc1** and **cax-eqc2** rules, specify **scm-eqc1** and **cax-sco** (**scm-eqc1,cax-sco**) in the WITH RULES clause.

Rule	Description	IF	THEN
cax- eqc1	Two classes are synonymous.	T(?c1, owl:equivalentClass, ?c2) T(?x, rdf:type, ?c1)	T(?x, rdf:type, ?c2)
cax- eqc2	Two classes are synonymous.	T(?c1, owl:equivalentClass, ?c2) T(?x, rdf:type, ?c2)	T(?x, rdf:type, ?c1)
cax-sco	Members of a subclass are also members	T(?c1, rdfs:subClassOf,	T(?x,

Rule	Description	IF	THEN
	of the superclass.	?c2) T(?x, rdf:type, ?c1)	rdf:type, ?c2)

Semantics of Axioms about Properties

Note

Because **prp-eqp1** and **prp-eqp2** (described in the table below) are implied rules that are coupled with **scm-eqp1** and **prp-spo1**, including **prp-eqp1** or **prp-eqp2** in the WITH RULES clause will result in an `invalid inference rule name` error. To run the **prp-eqp1** and **prp-eqp2** rules, specify **scm-eqp1** and **prp-spo1** (**scm-eqp1,prp-spo1**) in the WITH RULES clause.

Rule	Description	IF	THEN
prp-dom	Infer the subject's type from the predicate's domain.	T(?p, rdfs:domain, ?c) T(?x, ?p, ?y)	T(?x, rdf:type, ?c)
prp-eqp1	Two properties are synonymous.	T(?p1, owl:equivalentProperty, ?p2) T(?x, ?p1, ?y)	T(?x, ?p2, ?y)
prp-eqp2	Two properties are synonymous.	T(?p1, owl:equivalentProperty, ?p2) T(?x, ?p2, ?y)	T(?x, ?p1, ?y)
prp-fp	If predicate p is a functional property, then a subject can be related to only one specific object by p.	T(?p, rdf:type, owl:FunctionalProperty) T(?x, ?p, ?y1) T(?x, ?p, ?y2)	T(?y1, owl:sameAs, ?y2)
prp-ifp	If predicate p is an inverse	T(?p, rdf:type,	T(?x1,

Rule	Description	IF	THEN
	functional property, then a specific object can be related to only one subject by p.	owl:InverseFunctionalProperty) T(?x1, ?p, ?y) T(?x2, ?p, ?y)	owl:sameAs, ?x2)
prp-inv1	Two properties are the inverse of each other.	T(?p1, owl:inverseOf, ?p2) T(?x, ?p1, ?y)	T(?y, ?p2, ?x)
prp-inv2	Two properties are the inverse of each other.	T(?p1, owl:inverseOf, ?p2) T(?x, ?p2, ?y)	T(?y, ?p1, ?x)
prp-rng	Infer the object's type from the predicate's range.	T(?p, rdfs:range, ?c) T(?x, ?p, ?y)	T(?y, rdf:type, ?c)
prp-spo1	Relationships that are described by a subproperty also hold for the superproperty.	T(?p1, rdfs:subPropertyOf, ?p2) T(?x, ?p1, ?y)	T(?x, ?p2, ?y)
prp-symp	The inverse is true for a property.	T(?p, rdf:type, owl:SymmetricProperty) T(?x, ?p, ?y)	T(?y, ?p, ?x)
prp-trp	Chains of relationships collapse into a single relationship.	T(?p, rdf:type, owl:TransitiveProperty) T(?x, ?p, ?y) T(?y, ?p, ?z)	T(?x, ?p, ?z)

Semantics of Schema Vocabulary

Rule	Description	IF	THEN
scm-cls	Every class is its own	T(?c, rdf:type, owl:Class)	T(?c, rdfs:subClassOf, ?c)

Rule	Description	IF	THEN
	subclass and equivalent class, and it is a subclass of owl:Thing.		T(?c, owl:equivalentClass, ?c) T(?c, rdfs:subClassOf, owl:Thing) T(owl:Nothing, rdfs:subClassOf, ?c)
scm-dom1	A property with domain c also has domain c's superclasses.	T(?p, rdfs:domain, ?c1) T(?c1, rdfs:subClassOf, ?c2)	T(?p, rdfs:domain, ?c2)
scm-dom2	A subproperty inherits the domains of the superproperties.	T(?p2, rdfs:domain, ?c) T(?p1, rdfs:subPropertyOf, ?p2)	T(?p1, rdfs:domain, ?c)
scm-ecq1	Equivalent classes are subclasses of each other.	T(?c1, owl:equivalentClass, ?c2)	T(?c1, rdfs:subClassOf, ?c2) T(?c2, rdfs:subClassOf, ?c1)
scm-ecq2	If two classes are subclasses, they are also equivalent classes.	T(?c1, rdfs:subClassOf, ?c2) T(?c2, rdfs:subClassOf, ?c1)	T(?c1, owl:equivalentClass, ?c2)
scm-epq1	Equivalent properties are subproperties of each other.	T(?p1, owl:equivalentProperty, ?p2)	T(?p1, rdfs:subPropertyOf, ?p2) T(?p2, rdfs:subPropertyOf, ?p1)
scm-epq2	If two properties are subproperties, they are	T(?p1, rdfs:subPropertyOf, ?p2)	T(?p1, owl:equivalentProperty,

Rule	Description	IF	THEN
	also equivalent properties.	$T(?p2, \text{rdfs:subPropertyOf}, ?p1)$	$?p2$
scm-rng1	A property with range c also has range c's superclasses.	$T(?p, \text{rdfs:range}, ?c1)$ $T(?c1, \text{rdfs:subClassOf}, ?c2)$	$T(?p, \text{rdfs:range}, ?c2)$
scm-rng2	A subproperty inherits the ranges of its superproperties.	$T(?p2, \text{rdfs:range}, ?c)$ $T(?p1, \text{rdfs:subPropertyOf}, ?p2)$	$T(?p1, \text{rdfs:range}, ?c)$
scm-sco	owl:subClassOf relationships are transitive	$T(?c1, \text{rdfs:subClassOf}, ?c2)$ $T(?c2, \text{rdfs:subClassOf}, ?c3)$	$T(?c1, \text{rdfs:subClassOf}, ?c3)$
scm-spo	owl:subPropertyOf relationships are transitive.	$T(?p1, \text{rdfs:subPropertyOf}, ?p2)$ $T(?p2, \text{rdfs:subPropertyOf}, ?p3)$	$T(?p1, \text{rdfs:subPropertyOf}, ?p3)$

Note

The scm-dp and scm-op schema vocabulary rules are not run. Those rules add significant compute overhead but do not result in meaningful inference results.

OWL 2 RL Rules

The tables below define the subset of OWL 2 RL inference rules that are supported.

Semantics of Equality

Rule	Description	IF	THEN
eq-rep-o	Describes the replacement property of the owl:sameAs axiom.	T(?o, owl:sameAs, ?o') T(?s, ?p, ?o)	T(?s, ?p, ?o')
eq-rep-p	Describes the replacement property of the owl:sameAs axiom.	T(?p, owl:sameAs, ?p') T(?s, ?p, ?o)	T(?s, ?p', ?o)
eq-rep-s	Describes the replacement property of the owl:sameAs axiom.	T(?s, owl:sameAs, ?s') T(?s, ?p, ?o)	T(?s', ?p, ?o)
eq-sym	Describes the symmetric property of the owl:sameAs axiom.	T(?x, owl:sameAs, ?y)	T(?y, owl:sameAs, ?x)
eq-trans	Describes the transitive property of the owl:sameAs axiom.	T(?x, owl:sameAs, ?y) T(?y, owl:sameAs, ?z)	T(?x, owl:sameAs, ?z)

Semantics of Schema Vocabulary

Rule	Description	IF	THEN
scm-svf1	A property restriction c1 is a subclass of c2 if they are both someValuesFrom restrictions on the same property and c1's target class is a subclass of c2's target class.	T(?c1, owl:someValuesFrom, ?y1) T(?c1, owl:onProperty, ?p) T(?c2, owl:someValuesFrom,	T(?c1, rdfs:subClassOf, ?c2)

Rule	Description	IF	THEN
		?y2) T(?c2, owl:onProperty, ?p) T(?y1, rdfs:subClassOf, ?y2)	
scm-svf2	A property restriction c1 is a subclass of c2 if they are both someValuesFrom restrictions on the same class where c1's target property is a subproperty of c2's target property.	T(?c1, owl:someValuesFrom, ?y) T(?c1, owl:onProperty, ?p1) T(?c2, owl:someValuesFrom, ?y) T(?c2, owl:onProperty, ?p2) T(?p1, rdfs:subPropertyOf, ?p2)	T(?c1, rdfs:subClassOf, ?c2)
scm-int		T(?c, owl:intersectionOf, ?x) LIST[?x, ?c1, ..., ?cn]	T(?c, rdfs:subClassOf, ?c1) T(?c, rdfs:subClassOf, ?c2) ... T(?c, rdfs:subClassOf, ?cn)

Semantics of Classes

Rule	Description	IF	THEN
cls-svf1	At least one object of a property is a member of the specified class.	$T(?x,$ $owl:someValuesFrom,$ $?y)$ $T(?x, owl:onProperty, ?p)$ $T(?u, ?p, ?v)$ $T(?v, rdf:type, ?y)$	$T(?u,$ $rdf:type,$ $?x)$
cls-int1	An instance belongs to every one of the specified classes.	$T(?c, owl:intersectionOf,$ $?x)$ $LIST[?x, ?c1, ..., ?cn]$ $T(?y, rdf:type, ?c1)$ $T(?y, rdf:type, ?c2)$ $...$ $T(?y, rdf:type, ?cn)$	$T(?y,$ $rdf:type,$ $?c)$

Inference Example

The following simple example demonstrates the inferences that Graph Lakehouse generates to infer friendships in the sample Tickit data set. The example uses the following subset of triples, and it describes the friend relationships using the `owl:TransitiveProperty` vocabulary:

```
# friends.ttl
PREFIX owl: <http://www.w3.org/2002/07/owl#>
<friend> a owl:TransitiveProperty .
<person1>
rdf:type <person>
; <name> "Rafael Taylor"
; <like> "sports", "theatre", "classical", "vegas", "musicals"
; <dislike> "jazz", "broadway"
; <friend> <person2>, <person4>
.
<person2>
rdf:type <person>
; <name> "Vladimir Humphrey"
```

```

; <like> "jazz", "classical", "vegas", "musicals"
; <dislike> "broadway"
; <friend> <person3>
.
<person3>
rdf:type <person>
; <name> "Lars Ratliff"
; <like> "sports", "rock", "musicals"
; <dislike> "theatre", "jazz", "opera"
; <friend> <person1>
.
<person4>
rdf:type <person>
; <name> "Barry Roy"
; <like> "theatre"
; <dislike> "sports", "jazz", "musicals"
; <friend> <person5>
.
<person5>
rdf:type <person>
; <name> "Reagan Hodge"
; <like> "concerts", "rock", "vegas", "musicals"
; <dislike> "jazz", "broadway"
; <friend> <person1>
.

```

Loading `friends.ttl` into a graph named **friends** and querying the new graph for a list of friendships produces the following results. The query returns 6 friendships:

```

SELECT *
FROM <friends>
WHERE { ?person <friend> ?friend . }
ORDER BY ?person

```

```

person  | friend
-----+-----
person1 | person4
person1 | person2
person2 | person3
person3 | person1
person4 | person5
person5 | person1
6 rows

```

The query below generates inferences for the friends graph based on the rules for `owl:TransitiveProperty`. The query creates the inferences in a graph named **more-friends**:

```
CREATE INFERENCES FROM <friends> INTO GRAPH <more-friends>
```

When the inferencing is complete, the following query returns the friend triples in the more-friends graph. The query filters out triples for which the same person is the subject (`?person`) and object (`?friend`):

```
SELECT *
FROM <more-friends>
WHERE {
    ?person <friend> ?friend.
    FILTER(?person != ?friend).
}
ORDER BY ?person
```

```
person  | friend
-----+-----
person1 | person5
person1 | person3
person2 | person5
person2 | person4
person2 | person1
person3 | person5
person3 | person2
person3 | person4
person4 | person2
person4 | person3
person4 | person1
person5 | person2
person5 | person4
person5 | person3
14 rows
```

Following the OWL 2 RL rules for `owl:TransitiveProperty`, Graph Lakehouse inferred 8 new friendships from the 6 friendships in the original friend graph.

Validate Data with SHACL (Preview)

Graph Lakehouse supports using the W3C standard [Shapes Constraint Language](#) (SHACL) to describe and validate your knowledge graphs.

Note

The SHACL feature is a **Preview** release, which means the implementation has recently been completed, does not support the complete specification, and could be unstable. The feature is available for trial usage, but Altair recommends that you do not rely on Preview features in production environments.

This section provides an introduction to SHACL, information about shapes graph requirements and constraints, instructions on creating and configuring shapes graphs and validating your data graphs against the shapes, and interpreting the resulting validation graphs.

In this section:

Introduction to SHACL	437
Constraint Component Reference	438
Create a Shapes Graph	453
Validate a Data Graph	458

Introduction to SHACL

SHACL is a modeling language for describing a set of conditions and constraints that data in knowledge graphs must follow. The conditions are defined in structures called SHACL shapes, which are in the form of RDF graphs called *shapes graphs*. The graphs that are validated against shapes graphs are called *data graphs*.

Targets in the shapes graphs define the nodes, classes, and/or properties in the data graphs that must conform to the shape, and constraints define how to validate the targeted data. There are two types of shapes graphs: *node shapes* and *property shapes*. Node shapes define constraints on focus nodes, and property shapes define constraints on the values for properties that are connected to the focus nodes.

Shape Requirements

A shape is a URI or blank node that meets at least one of the following conditions in the shapes graph:

- The shape is an instance of `sh:NodeShape` or `sh:PropertyShape`.
- The shape has at least one of the following predicates: `sh:targetClass`, `sh:targetNode`, `sh:targetObjectsOf` or `sh:targetSubjectsOf`.
- The shape has a `sh:property [ parameter_list ] predicate`.
- The shape is a value of any of the constraint components described in [Constraint Component Reference](#).

Data Validation

The validation processor in Graph Lakehouse is invoked by running a SPARQL query. The processor validates one or more data graphs against the constraints defined in one or more shapes graphs and produces a report in the form of a *validation graph*. For more information, see [Validate a Data Graph](#).

Constraint Component Reference

This section describes each of the constraint components that Graph Lakehouse supports for node and property shapes. Certain constraints are valid only in property shapes. And other constraints are valid in both node and property shapes. When a constraint is applied to a particular property—by including the `sh:path <property_uri>` predicate in the shape—the specified condition applies only to that property. For example the following snippet from a shapes graph creates a condition that requires values for the `age` property in the `Person` class to be between 0 and 130 (inclusive):

```
ex:PersonShape a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    sh:path ex:age ;
    sh:minInclusive 0 ;
    sh:maxInclusive 130 ;
  ]
```

If a constraint is applied to a node shape (the `sh:path <property_uri>` predicate is excluded), the condition applies to all properties associated with the focus node. For example the following constraints apply to all properties related to the `Person` node. All properties are required to have values between 0 and 130 (inclusive):

```
ex:PersonShape a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    sh:minInclusive 0 ;
    sh:maxInclusive 130 ;
  ]
```

Constraint Types

- [Cardinality Constraints](#)
- [Logical Constraints](#)
- [Other Constraints](#)
- [Property Pair Constraints](#)

- [Shape-Based Constraints](#)
- [String-Based Constraints](#)
- [Value Range Constraints](#)
- [Value Type Constraints](#)

Cardinality Constraints

Constraint	Shape Type	Data Type	Description
sh:maxCount	property	int	This constraint sets a limit on the maximum number of values for a property. The following example limits the <code>lastName</code> property to one value. <pre>sh:property [sh:path ex:lastName ; sh:maxCount 1; sh:datatype xsd:string;]</pre>
sh:minCount	property	int	This constraint requires a minimum number of values for a property. The following example requires the <code>lastName</code> property to have one value. <pre>sh:property [sh:path ex:lastName ; sh:minCount 1; sh:maxCount 1; sh:datatype xsd:string]</pre>

Logical Constraints

Constraint	Shape Type	Data Type	Description
sh:or	node, property	URI list	This constraint requires a node or property to conform to at least one of the listed shapes. The following example requires the <code>child</code> property to contain a value that conforms to the <code>biological</code> or <code>adopted</code> shapes. <pre>sh:property [sh:path ex:child ; sh:or (ex:biological ex:adopted)]</pre>
sh:and	node, property	URI list	This constraint requires a node or property to conform to all of the listed shapes. The following example requires the <code>employee</code> property to conform to the <code>person</code> and <code>organization</code> shapes. <pre>sh:property [sh:path ex:employee ; sh:and (ex:person ex:organization)]</pre>
sh:not	node, property	URI	This constraint defines a condition where a node or property must not conform to any of the listed shapes. The following example specifies that the <code>president</code> property cannot conform to the <code>felon</code> shape. <pre>sh:property [</pre>

Constraint	Shape Type	Data Type	Description
			<pre> sh:path ex:president ; sh:not ex:felon ;] </pre>
sh:xone	node, property	URI list	This constraint defines a condition where a node or property must conform to one and only one of the listed shapes. The following example specifies that the <code>child</code> property must conform to either the <code>biological</code> or <code>adopted</code> shape but cannot conform to both shapes. <pre> sh:property [sh:path ex:child ; sh:xone (ex:biological ex:adopted)] </pre>

Other Constraints

Constraint	Shape Type	Data Type	Description
sh:in	node, property	URI or literal list	This constraint restricts a node or property value to be one of those specified. The following example restricts the <code>continent</code> property to contain one of three possible values. <pre> sh:property [sh:path ex:continent ; sh:in (ex:Asia, ex:Europe, ex:NorthAmerica); </pre>

Constraint	Shape Type	Data Type	Description
			]
sh:closed, sh:ignoredProperties	node	boolean, URI list	The <code>closed</code> and optional <code>ignoredProperties</code> constraints can be used to limit the properties that are allowed for a node. If <code>sh:closed true</code>, only the properties that are described in the shape are valid. You can include <code>ignoredProperties</code> if you want to list any properties that are not described in the shape but should be allowed for the target. In the following example, the only allowed properties for <code>Employee</code> are <code>rdf:type</code>, <code>id</code>, and <code>record</code>. <pre> ex:EmployeeShape a sh:NodeShape; sh:targetClass ex:Employee; sh:closed true; sh:ignoredProperties (rdf:type) sh:property [sh:path ex:id ; sh:datatype xsd:long;] sh:property [sh:path ex:record ; sh:class sh:EmployeeRecord;] </pre>

Constraint	Shape Type	Data Type	Description
sh:hasValue	node, property	URI, literal	This constraint requires a node or property to have at least one value that matches the specified <code>sh:hasValue</code>. The following example requires the <code>genre</code> property for the <code>Book</code> node to have at least one value that is <code>ex:Mystery</code>. <pre> ex:BookShape a sh:NodeShape ; sh:targetClass ex:Book ; sh:property [sh:path ex:genre ; sh:hasValue ex:Mystery ;] </pre>
sh:sparql, sh:select	node, property	URI, string	This SPARQL-based constraint can be used to set restrictions based on the specified SPARQL SELECT query. The following pre-bound variables are known in the SPARQL query: • \$this: subject • \$PATH: predicate • ?value: object The following example creates a requirement for email addresses to be strings that end in <code>.com</code>. <pre> sh:property [sh:path ex:email ; </pre>

Constraint	Shape Type	Data Type	Description
			<pre> sh:sparql [a sh:SPARQLConstraint ; sh:message "Email is a string ending in .com" ; sh:select """ SELECT \$this ?value WHERE { \$this \$PATH ?value . FILTER(DATATYPE (?value) != xsd:string (lcase(substr(str (?value),strlen(str(?value))- 3)) not in (".com",".net",".gov",".edu"))). } """ ;];] </pre>

Property Pair Constraints

Constraint	Shape Type	Data Type	Description
sh:equals	property	URI	This constraint requires a property to have a value that is equal to the specified value (value1 = value2). The

Constraint	Shape Type	Data Type	Description
			following example requires the value for the <code>firstName</code> property to equal the value of <code>givenName</code>. <pre>sh:property [sh:path ex:firstName ; sh>equals ex:givenName;]</pre>
sh:disjoint	property	URI	This constraint requires a property to have a value that is not equal to the specified value (<code>value1 != value2</code>). The following example specifies that the prefix label must not equal the <code>label</code> value. <pre>sh:property [sh:path ex:prefLabel ; sh:disjoint ex:label ;]</pre>
sh:lessThan	property	URI	This constraint requires a property to have a value that is less than the specified value (<code>value1 < value2</code>). The following example requires the <code>startDate</code> value to be less than the <code>endDate</code> value. <pre>sh:property [sh:path ex:startDate ;</pre>

Constraint	Shape Type	Data Type	Description
			<pre> sh:lessThan ex:endDate;] </pre>
sh:lessThanOrEquals	property	URI	This constraint requires a property to have a value that is less than or equal to the specified value (<code>value1 <= value2</code>). The following example requires the <code>startDate</code> value to be less than or equal to the <code>endDate</code> value. <pre> sh:property [sh:path ex:startDate ; sh:lessThanOrEquals ex:endDate;] </pre>

Shape-Based Constraints

Constraint	Shape Type	Data Type	Description
sh:node	node, property	URI list	This constraint requires a node or property to conform to the specified shape. The following example requires that the <code>address</code> property conforms to the <code>AddressShape</code>. <pre> sh:property [sh:path ex:address ; sh:minCount 1 ; </pre>

Constraint	Shape Type	Data Type	Description
			<pre>sh:node ex:AddressShape ;]</pre>
sh:property	node, property	URI list	This constraint is used to define the property shape for a node or property.

String-Based Constraints

Constraint	Shape Type	Data Type	Description
sh:minLength	node, property	int	This constraint requires a literal value or URI to meet a minimum character length. The following example requires the <code>password</code> property to have a value that is at least 8 characters. <pre>sh:property [sh:path ex:password ; sh:minLength 8 ;]</pre>
sh:maxLength	node, property	int	This constraint sets a limit on the number of characters a literal value or URI can have. The following example limits values for the <code>country</code> property to 60 characters. <pre>sh:property [sh:path ex:country ; sh:maxLength 60 ;]</pre>

Constraint	Shape Type	Data Type	Description
sh:pattern, sh:flags	node, property	string, string	The <code>pattern</code> and optional <code>flags</code> constraint can be included to require a property or node to match a regular expression pattern. For the supported regex syntax, see the Regular Expression Syntax section of the W3C XQuery 1.0 and XPath 2.0 Functions and Operators specification. Include <code>flags</code> if you want to include optional modifier flags that further define the pattern. See the Flags section of the W3C Functions and Operators specification. The following example requires the <code>zipcode</code> to match a string that consists of exactly five digits (0-9). <pre>sh:property [sh:path ex:zipcode ; sh:pattern "^\\d{5}\$";]</pre>
sh:languageIn	property	string list	This constraint limits the language tags that are allowed for a property. The following example limits the <code>description</code> property to English, German, or French. <pre>sh:property [sh:path ex:description ; sh:languageIn ("en", "de",</pre>

Constraint	Shape Type	Data Type	Description
			<pre>"fr");]</pre>
sh:uniqueLang	property	boolean	This constraint creates a condition where no two values can have the same language tag. Each value must have a unique tag. The following example requires each <code>label</code> value to have a unique language tag. <pre>sh:property [sh:path ex:label ; sh:uniqueLang true ;]</pre>

Value Range Constraints

Constraint	Shape Type	Data Type	Description
sh:minExclusive	node, property	literal	This constraint sets the minimum value for a node or property, excluding the value that is specified. The following example requires the minimum value for the <code>length</code> property to be greater than 0. <pre>sh:property [sh:path ex:length ; sh:minExclusive 0;]</pre>
sh:maxExclusive	node, property	literal	This constraint sets the maximum

Constraint	Shape Type	Data Type	Description
			value for a node or property, excluding the value that is specified. The following example requires the maximum value for the <code>price</code> property to be less than 100.00. <pre>sh:property [sh:path ex:price ; sh:maxExclusive 100.00;]</pre>
sh:minInclusive	node, property	literal	This constraint sets the minimum value for a node or property, including the value that is specified. The following example requires the minimum value for the <code>age</code> property to be greater than or equal to 0. <pre>sh:property [sh:path ex:age ; sh:minInclusive 0;]</pre>
sh:maxInclusive	node, property	literal	This constraint sets the maximum value for a node or property, including the value that is specified. The following example requires the maximum value for the <code>age</code> property to be less than or equal to 120. <pre>sh:property [sh:path ex:age ; sh:maxInclusive 120;</pre>

Constraint	Shape Type	Data Type	Description
			]

Value Type Constraints

Constraint	Shape Type	Data Type	Description
sh:nodeKind	node, property	URI	This constraint requires the values for a node or property to be of a certain type. Valid <code>nodeKind</code> values are: • sh:IRI: The value must be an IRI. • sh:BlankNode: The value must be a blank node. • sh:Literal: The value must be a literal. • sh:BlankNodeOrIRI: The value must be a blank node or IRI. • sh:BlankNodeOrLiteral: The value must be a blank node or literal. The following example requires the <code>birthDate</code> property to contain literal values. <pre>sh:property [sh:path ex:birthDate ; sh:nodeKind sh:Literal ;]</pre>

Constraint	Shape Type	Data Type	Description
sh:datatype	node, property	URI	This constraint requires each node or property value to be the specified data type. The following example requires the <code>age</code> property to be an integer. <pre> sh:property [sh:path ex:age ; sh:datatype xsd:integer ;]</pre>
sh:class	node, property	URI list	This constraint requires each node or property to have an <code>rdf:type</code> that matches one of the values specified in <code>sh:class</code>. The following example requires the <code>address</code> property to be a <code>PostalAddress</code>. <pre> sh:property [sh:path ex:address ; sh:class ex:PostalAddress;]</pre>

Create a Shapes Graph

You can create a shapes graph by writing and loading a Turtle file or by running a SPARQL INSERT query. This topic provides guidance and examples for creating and loading a shapes graph. For information about the constraints that are supported in shapes graphs, see [Constraint Component Reference](#).

Important

Each shapes graph can contain up to 60 shapes. To load more than 60 shapes, create multiple graphs.

- [Defining Shapes in a TTL File](#)
- [Creating Shapes with an INSERT Query](#)

Defining Shapes in a TTL File

The following example shows a .ttl file that configures shape constraints on employee data. The file includes comments and messages that explain the constraints.

```
# employee_shapes.ttl
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .

ex:EmployeeShape
  a sh:NodeShape ;
  sh:targetClass ex:Employee ;
  sh:property [
    sh:path ex:hasID ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
    sh:pattern "^[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]$" ;
    sh:message "Every employee must have an ID that matches the pattern" ;
  ] ;
  sh:property [
```

```

    sh:path ex:employeeType ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
    sh:in ("Manager" "Worker" "Contractor") ;
    sh:message "Every employee is a manager, worker, or contractor" ;
] ;
sh:property [
    sh:path ex:birthYear ;
    sh:maxInclusive 2007 ;
    sh:datatype xsd:integer ;
    sh:message "Birth year must be 2007 or earlier" ;
] ;
sh:property [
    sh:path ex:hasTitle ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:message "Must have a title but may have more than one" ;
] ;
sh:or (
# All employees must have a supervisor except for the President
[
    sh:path ex:hasSupervisor ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:class ex:Employee ;
]
[
    sh:path ex:hasTitle ;
    sh:hasValue "President" ;
]
) ;
sh:or (
# Every employee must have an hourly wage or salary
[
    sh:path ex:hasSalary ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
]
[
    sh:path ex:hasWage ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
]
) ;

```

```

    ]
  ) ;
  sh:property [
    sh:path ex:hasSalary ;
    sh:datatype xsd:double ;
    sh:minInclusive 30000.00 ;
    sh:message "Salary must be 30,000 or higher" ;
  ] ;
  sh:property [
    sh:path ex:hasWage ;
    sh:datatype xsd:double ;
    sh:minInclusive 15.00 ;
    sh:message "Wage must be at least 15.00" ;
  ] .

```

To create the shapes graph, load the file to Graph Lakehouse. For example, the following query loads the TTL file from a mounted file system into a graph named

<http://anzograph.com/employeeShapes>:

```

LOAD <file:/mnt/shared/data/employee_shapes.ttl> INTO GRAPH
<http://anzograph.com/employeeShapes>

```

For more information about loading files, see [Load RDF Data from Files](#).

Creating Shapes with an INSERT Query

The example below shows an INSERT query that configures the same shape constraints as the TTL example above. Running the query creates a shapes graph called

<http://anzograph.com/employeeShapes>.

```

# employee_shapes.rq
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.org/>

INSERT DATA { GRAPH <http://anzograph.com/employeeShapes> {
  ex:EmployeeShape
    a sh:NodeShape ;
    sh:targetClass ex:Employee ;
    sh:property [

```

```

    sh:path ex:hasID ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
    sh:pattern "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" ;
    sh:message "Every employee must have an ID that matches the pattern" ;
] ;
sh:property [
    sh:path ex:employeeType ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:string ;
    sh:in ("Manager" "Worker" "Contractor") ;
    sh:message "Every employee is a manager, worker, or contractor" ;
] ;
sh:property [
    sh:path ex:birthYear ;
    sh:maxInclusive 2007 ;
    sh:datatype xsd:integer ;
    sh:message "Birth year must be 2007 or earlier" ;
] ;
sh:property [
    sh:path ex:hasTitle ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:message "Must have a title but may have more than one" ;
] ;
sh:or (
# All employees must have a supervisor except for the President
[
    sh:path ex:hasSupervisor ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:class ex:Employee ;
]
[
    sh:path ex:hasTitle ;
    sh:hasValue "President" ;
]
) ;
sh:or (
# Every employee must have an hourly wage or salary
[

```



```

    sh:path ex:hasSalary ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ]
  [
    sh:path ex:hasWage ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ]
) ;
sh:property [
  sh:path ex:hasSalary ;
  sh:datatype xsd:double ;
  sh:minInclusive 30000.00 ;
  sh:message "Salary must be 30,000 or higher" ;
] ;
sh:property [
  sh:path ex:hasWage ;
  sh:datatype xsd:double ;
  sh:minInclusive 15.00 ;
  sh:message "Wage must be at least 15.00" ;
] .
}
}

```

Validate a Data Graph

Data graphs are validated by running a SPARQL query that lists the data graphs to validate and the shapes graphs to validate the data against. Depending on the type of query that you run, Graph Lakehouse returns tabular results or a validation graph that uses [SHACL Validation Report Vocabulary](#) to report on any conformance and constraint violations. This topic describes the validation query syntax and includes examples.

- [Validation Query Syntax](#)
- [Validation Examples](#)

Validation Query Syntax

There are two modes in which you can run a validation query: **query** mode and **report** mode. In query mode, tabular results are returned. If 0 results are returned in query mode, that means the data graphs conform to the shapes graphs. In report mode, results are inserted into a specified graph, and you query the graph to review the validation results. The syntax for each mode is described below.

Query Mode

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX ...
USING
    <shapes_graph_uri>
    [ <shapes_graph2_uri> ]
    [ ... ]
VALIDATE
    <data_graph_uri>
    [ <data_graph2_uri> ]
    [ ... ]
```

For example:

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX azg: <http://anzograph.com/>
USING azg:personShapes
VALIDATE azg:personData
```

Report Mode

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX ...
USING
  <shapes_graph_uri>
  [ <shapes_graph2_uri> ]
  [ ... ]
VALIDATE
  <data_graph_uri>
  [ <data_graph2_uri> ]
  [ ... ]
CREATE REPORT GRAPH <report_graph_uri>
```

For example:

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX azg: <http://anzograph.com/>
USING azg:personShapes
VALIDATE azg:personData
CREATE REPORT GRAPH azg:personReport
```

Validation Examples

- [Sample Data Graph](#)
- [Query Mode Example](#)
- [Report Mode Example](#)

Sample Data Graph

The examples below validate the data graph that is defined in the following INSERT query. The data is validated against the shapes graph example in [Create a Shapes Graph](#).

```
# employee-data.rq
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.org/>

INSERT DATA {
```

```

GRAPH <http://anzograph.com/employeeData>
{
  ex:Employee
    a rdfs:Class .
  ex:emp001
    a ex:Employee ;
    ex:hasID "000-12-3456" ;
    ex:hasTitle "President" ;
    ex:employeeType "Manager" ;
    ex:birthYear "1953"^^xsd:integer ;
    ex:hasSalary "100000"^^xsd:double .
  ex:emp002
    a ex:Employee ;
    ex:hasID "000-56-3456" ;
    ex:hasTitle "Foreman" ;
    ex:employeeType "Worker" ;
    ex:birthYear "1966"^^xsd:integer ;
    ex:hasSupervisor ex:emp003 ;
    ex:hasWage "20.20"^^xsd:double .
  ex:emp003
    a ex:Employee ;
    ex:hasID "000-77-3232" ;
    ex:hasTitle "Production Manager" ;
    ex:employeeType "Manager" ;
    ex:birthYear "1968"^^xsd:integer ;
    ex:hasSupervisor ex:emp001 ;
    ex:hasSalary "4000"^^xsd:double .
  ex:emp004
    a ex:Employee ;
    ex:hasID "0" ;
    ex:hasTitle "Fitter" ;
    ex:employeeType "Worker" ;
    ex:birthYear "1979"^^xsd:integer ;
    ex:hasSupervisor ex:emp002 ;
    ex:hasWage "17.20"^^xsd:double .
  ex:emp005
    a ex:Employee ;
    ex:hasID "000-99-3492" ;
    ex:hasTitle "Fitter" ;
    ex:employeeType "Worker" ;
    ex:hasSupervisor ex:emp002 ;
    ex:birthYear "2000"^^xsd:integer ;
    ex:hasWage "17.60"^^xsd:double .
}

```

```

ex:emp006
  a ex:Employee ;
  ex:hasID "000-78-5592" ;
  ex:hasTitle "Filer" ;
  ex:employeeType "Intern" ;
  ex:birthYear "2003"^^xsd:integer ;
  ex:hasSupervisor ex:emp002 ;
  ex:hasWage "14.20"^^xsd:double .
ex:emp007
  a ex:Employee ;
  ex:hasID "000-77-3232" ;
  ex:hasTitle "Sales Manager" ;
  ex:hasTitle "Vice President" ;
  ex:employeeType "Manager" ;
  ex:birthYear "1962"^^xsd:integer ;
  ex:hasSupervisor ex:emp001 ;
  ex:hasSalary "80000"^^xsd:double .
ex:emp008
  a ex:Employee ;
  ex:hasID "000-31-4868" ;
  ex:hasTitle "Fitter" ;
  ex:employeeType "Worker" ;
  ex:birthYear "2008"^^xsd:integer ;
  ex:hasSupervisor ex:emp002 ;
  ex:hasWage "15.00"^^xsd:double .
ex:emp009
  a ex:Employee ;
  ex:hasID "000-56-3336" ;
  ex:hasTitle "Fitter" ;
  ex:employeeType "Contractor" ;
  ex:birthYear "2001"^^xsd:integer ;
  ex:hasSupervisor ex:emp002 ;
  ex:hasWage "15.00"^^xsd:double .
}
}

```

Query Mode Example

The following example performs validation on the sample data graph above. The validation is done in query mode, where results are returned in tabular format:

```

PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX azg: <http://anzograph.com/>

```

```

USING azg:employeeShapes
VALIDATE azg:employeeData

```

The results show that there are 5 violations:

focusNode	resultPath	value	constraint
sourceShape	message	violation	
-----+-----+-----+-----			
-----+-----+-----+-----			
-----+-----+-----+-----			
http://example.org/emp003	http://example.org/hasSalary	4000	
http://www.w3.org/ns/shacl#MinInclusiveConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b10737418398	Salary must be 30,000 or higher	
http://example.org/emp004	http://example.org/hasID	0	
http://www.w3.org/ns/shacl#PatternConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b15032385677	Every employee must have an ID that matches the pattern	
http://example.org/emp006	http://example.org/employeeType	Intern	
http://www.w3.org/ns/shacl#InConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b6442451086	Every employee is a manager, worker, or contractor	
http://example.org/emp006	http://example.org/hasWage	14.2	
http://www.w3.org/ns/shacl#MinInclusiveConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b10737418399	Wage must be at least 15.00	
http://example.org/emp008	http://example.org/birthYear	2008	
http://www.w3.org/ns/shacl#MaxInclusiveConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b6442451087	Birth year must be 2007 or earlier	
5 rows			

For each violation, the `focusNode` (subject), `resultPath` (predicate), `value`, `constraint`, `violation`, `sourceShape`, and `message` (if one exists for the shape) is shown. In the first row, employee 3 has a salary of \$4,000, which violates the `MinInclusiveConstraintComponent` that says salaries must be at least \$30,000. In the second row, employee 4 has an ID value that violates `PatternConstraintComponent` because it is too short. Rows 3 and 4 show that employee 6 has an invalid employee type and a wage that is too low. And row 5 shows that employee 8 does not meet the age requirement.

Report Mode Example

The following example performs validation on the sample data graph above. The validation is done in report mode, where the results are saved to a graph rather than returned in tabular format:

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX azg: <http://anzograph.com/>
USING azg:employeeShapes
VALIDATE azg:employeeData
CREATE REPORT GRAPH azg:employeeReport
```

When the query is complete, you can query the new graph to view the results. First you can run a simple ASK query to see whether or not the graph conforms to the shapes. For example, the query below asks whether the value of `<http://www.w3.org/ns/shacl#conforms>` is `t` (true). If the value is `f` (false), the ASK query returns `false`:

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
ASK FROM <http://anzograph.com/employeeReport> { ?s sh:conforms "t" .}

false
```

If the data graph does not conform to the shapes graph, you can write additional queries to return information about the violations. For example:

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
SELECT ?focusNode ?resultPath ?value ?constraint ?violation ?sourceShape ?message
FROM <http://anzograph.com/employeeReport>
WHERE {
  ?s sh:focusNode ?focusNode ;
    sh:resultPath ?resultPath ;
    sh:value ?value ;
    sh:sourceConstraintComponent ?constraint ;
    sh:resultSeverity ?violation ;
    sh:sourceShape ?sourceShape ;
    sh:resultMessage ?message .
}
ORDER BY ?focusNode
LIMIT 100
```

The results show that there are 5 violations:

focusNode	resultPath	value	constraint
sourceShape	message	violation	
-----+-----+-----+-----			
-----+-----+-----+-----			
-----+-----+-----+-----			
http://example.org/emp003	http://example.org/hasSalary	4000	
http://www.w3.org/ns/shacl#MinInclusiveConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b10737418398	Salary must be 30,000 or higher	
http://example.org/emp004	http://example.org/hasID	0	
http://www.w3.org/ns/shacl#PatternConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b15032385677	Every employee must have an ID that matches the pattern	
http://example.org/emp006	http://example.org/employeeType	Intern	
http://www.w3.org/ns/shacl#InConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b6442451086	Every employee is a manager, worker, or contractor	
http://example.org/emp006	http://example.org/hasWage	14.2	
http://www.w3.org/ns/shacl#MinInclusiveConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b10737418399	Wage must be at least 15.00	
http://example.org/emp008	http://example.org/birthYear	2008	
http://www.w3.org/ns/shacl#MaxInclusiveConstraintComponent			
http://www.w3.org/ns/shacl#Violation	_:b6442451087	Birth year must be 2007 or earlier	
5 rows			

For each violation, the `focusNode` (subject), `resultPath` (predicate), `value`, `constraint`, `violation`, `sourceShape`, and `message` (if one exists for the shape) is shown. In the first row, employee 3 has a salary of \$4,000, which violates the `MinInclusiveConstraintComponent` that says salaries must be at least \$30,000. In the second row, employee 4 has an ID value that violates `PatternConstraintComponent` because it is too short. Rows 3 and 4 show that employee 6 has an invalid employee type and a wage that is too low. And row 5 shows that employee 8 does not meet the age requirement.

Copy Graphs to Files

This topic provides instructions for using the COPY command to copy graphs from Graph Lakehouse to compressed or uncompressed files on disk. You can copy a graph to a file if you added or updated the data in a graph and want to be able to load that updated graph into another Graph Lakehouse instance. Or, you may want to create a backup to restore data to a previous state after upgrading or installing a new version of Graph Lakehouse.

By default, when you restart Graph Lakehouse, it automatically reloads the last state of graph data from that stored in the `<install_path>/persistence` directory.

Note

Copying graph data to a file or directory does not remove the copied data from Graph Lakehouse.

COPY Syntax

Copy data to files by running the following SPARQL query. Each of the options are described below.

```
COPY ALL | graph_uri_list TO <single_file_uri> | <directory_uri>
```

Argument	Description
ALL	Include the ALL keyword if you want to copy all graphs to files rather than listing specific graphs. If you do not want to copy all graphs, specify a graph_uri_list .
graph_uri_list	Use the format below if you want to copy a single graph or a list of graphs. Separate multiple graphs with a space. <graph_URI> [<graph2_URI> <graphN_URI> ...]
single_file_uri	If you want to copy a graph or graphs to a single file, specify a file location URI in the format below. When generating a single file on a cluster, the leader node writes the file.

Argument	Description
	<pre data-bbox="370 205 932 233"><file:/path/filename.filetype[.gz]></pre> Where <code>filetype</code> is the file format to generate. Supported types are .ttl, .n3, .nt, .nq, .quads, and .trig. If you want to compress the files, include the .gz suffix. Note When copying from multiple graphs, make sure that you specify a quad format such as .nq, .quads, or .trig to preserve the graph name information in the data.
directory_uri	If you want to copy a graph or graphs to many smaller files, specify a directory location URI in the format below. When generating a directory of multiple files on a cluster, each node creates files that contain the data that is stored in its slices. It is important to choose a directory location that is shared between the nodes in the cluster. Otherwise you have to retrieve the files from each node separately. <pre data-bbox="370 1056 899 1083"><dir:/path/dirname.filetype[.gz]></pre> Where <code>filetype</code> is the file format to generate. Supported types are .ttl, .n3, .nt, .nq, .quads, and .trig. If you want to compress each of the files in the directory, include the .gz suffix. Note When copying from multiple graphs, make sure that you specify a quad format such as .nq, .quads, or .trig to preserve the graph name information in the data. Tip By default, Graph Lakehouse creates 5 MB .gz files in the specified directory. To configure Graph Lakehouse to create a different file size,

Argument	Description
	you can change the settings file, settings.conf, to add <code>copy_file_size=<number_of_MB></code> to the file. For instructions on changing settings, see Change System Settings.

COPY Examples

The example below copies data from the **flights** graph to a single **flights.ttl.gz** file on a shared file system.

```
COPY <http://anzograph.com/flights> TO <file:/mnt/shared/data/flights.ttl.gz>
```

The example below copies data from two graphs, **flights** and **airports**, to a **flight-data.trig.gz** directory on a shared file system. Using .trig format ensures that the graph names are included in the files.

```
COPY <http://anzograph.com/flights> <http://anzograph.com/airports> TO
<dir:/mnt/shared/data/flight-data.trig.gz>
```

The example below copies the data from all graphs to a directory on a shared file system:

```
COPY ALL TO <dir:/mnt/shared/data/allgraphs.trig.gz>
```

Schedule Automated Data Updates

There are often data update operations that must be performed on a regular or periodic basis, such as retrieving updates from external data sources or exporting data. Graph Lakehouse provides a CRON-like mechanism to automatically perform these repetitive operations. These operations are managed entirely within the database rather than being controlled by the configuration of external control files.

There are two primary aspects to creating and configuring automated or scheduled operations within Graph Lakehouse:

1. Create and define the contents of one or more Cron graphs, each of which specify the database operations to perform for one or more Cron jobs. Each Cron graph is defined as a collection of RDF triples, with each triple specifying a particular scheduled job attribute or parameter. The Cron graph includes configuration settings that control other aspects of each scheduled job, such as a job's scheduled execution time (particular dates and times or intervals), retry options, error handling policies, and so on.
2. Update the scheduled Cron graph job settings in the Graph Lakehouse **settings.conf** file to include the Cron graphs you want to execute. The settings.conf file contains two settings to control the scheduling and execution of Cron graphs, **cron_graphs** and **cron_graphs_recheck**.

This topic provides instructions for setting up automated database operations and describes the configuration options and best practices available to control the scheduling, prioritization, error handling, and other aspects of running jobs.

- [Create a Cron Graph](#)
- [Load a Cron Graph](#)
- [Configure Graph Lakehouse to Run Cron Jobs](#)
- [Monitor Job Execution and Errors](#)

Create a Cron Graph

A Cron graph is defined in a TTL file that contains a collection of RDF triples that define configuration and scheduling information for one or more Cron jobs. A Cron graph can contain any number of Cron jobs, and each job can have custom scheduling and error-handling policies.

The content below shows the syntax for a Cron graph file. Descriptions of each job parameter are provided below.

```
# filename.ttl

PREFIX azg: <http://www.anzograph.com/> .

<job_name> azg:Schedule | Delay "<duration_value>"^^xsd:duration ;
          azg:Statement "<statement>" ;
          azg:ErrorPolicy "<policy>" ;
          [ azg:BaseTime "<datetime_value>"^^xsd:dateTime ; ]
          [ azg:RetryInterval "<duration_value>"^^xsd::duration ; ]
          [ azg:RetryCount <integer_value> ; ]
          [ azg:RunAfterStartup "true | false" ] .

[ <job2>   azg:Schedule | Delay "<duration_value>"^^xsd:duration ;
          azg:Statement "<statement>" ;
          azg:ErrorPolicy "<policy>" ;
          ...
]
```

Note

If any required triples are missing or invalid, the associated Cron graph job is rejected and returns an error. (See [Monitor Job Execution and Errors](#).)

Parameter	Description
Schedule Delay	Each job is required to include either azg:Schedule or azg:Delay . Both parameters accept an <code>xsd:duration</code> data type value, as described in xsd:duration in the W3C specification. If BaseTime is also specified, the azg:Schedule duration is added to the BaseTime value to produce the job's next scheduled execution time. If azg:Delay is specified, the

Parameter	Description
	execution of the associated job is delayed by the specified interval (in seconds) from the time the job last completed. Note These options set "scheduled request times," not guaranteed start times. If the system is busy enough that a given job would have multiple outstanding requested start times, only the last one is executed. If Graph Lakehouse is stopped and subsequently restarted, Cron jobs return to their normal scheduled interval times. For example, for nightly jobs scheduled for execution at midnight, a skipped midnight job will not be performed until midnight of the next day.
Statement	This required parameter defines the database operation (any valid SPARQL statement) to be performed when the corresponding job is executed. You can specify multiple statements by separating each statement by double semicolon characters (;;), e.g., <code>statement1;;statement2</code>. Each statement is executed as a separate transaction following ACID principles. Note Specifying SPARQL statements in a separate file rather than as a text string (for example, <file:/path/job1.rq>) is not currently supported.
ErrorPolicy	To specify what happens when Graph Lakehouse encounters an error in processing a job, all jobs require an azg:ErrorPolicy parameter. Each job must contain exactly one error policy. The list below describes the valid policy values: • AbortDatabase: The most conservative policy. Any critical job

Parameter	Description
	failure produces a crash-dump Xray. • Ignore: The most liberal policy. The error information is recorded in the <code>sth_errors</code> system table and included in manually generated Xrays, but no feedback is returned while the job is processed. • Disable: If an error occurs, this value directs Graph Lakehouse not to attempt to run the Cron job again. • BlockUsers: Similar to <code>AbortDatabase</code>, this policy causes all subsequent user-issued <code>SELECT</code> queries to error out with a "Cron failed, contact your system administrator" message. To unblock users from running <code>SELECT</code> queries, you can restart the database or run a <code>SET selects_blocked TO false</code> query. For example: <pre>azgi -c "SET selects_blocked TO false"</pre> Note If RetryInterval is specified in for a job, error policy actions are postponed until the number of retries is reached.
BaseTime	This optional parameter specifies the time to use as the base for other timing-related settings. If <code>azg:BaseTime</code> is not specified, Graph Lakehouse's start time is used to determine the Cron job's first start time. For example, if Graph Lakehouse was started at 2 PM on Sunday, May 12, and <code>azg:Schedule "1Day"^^xsd:duration</code> was set, then the job would run every day at 2 PM.
RetryInterval	This optional parameter specifies the duration to wait before retrying the job if the job errors out. The job will be continually retried until the first success

Parameter	Description
	or until the RetryCount value is reached. Afterwards, the job returns to its normal scheduled time.
RetryCount	This optional parameter specifies the number of times to retry a job if it errors out. If azg:RetryCount is specified, RetryInterval must also be specified. When the number of retries reaches the retry count, the specified ErrorPolicy for this job is performed. If the Ignore error policy is specified, the associated job resumes its normal schedule time.
RunAfterStartup	This optional parameter accepts a "true" or "false" value that indicates whether the associated job should run shortly after Graph Lakehouse startup. If azg:RunAfterStartup "true" , the azg:Schedule value is ignored.

Example Cron Graph File

The following content provides a simple example of a Cron graph file, named `cron1.ttl`, which schedules two jobs in the same graph:

```
PREFIX azg: <http://www.anzograph.com/> .
<job1> azg:BaseTime "2020-04-07:11:32"^^xsd:dateTime .
<job1> azg:Schedule "1Day"^^xsd:duration .
<job1> azg:ErrorPolicy "AbortDatabase" .
<job1> azg:RetryInterval "1Hour"^^xsd:duration .
<job1> azg:RetryCount 23 .
<job1> azg:Statement "REFRESH VIEW <testView1>" .
<job2> azg:BaseTime "2020-07-08:00:00"^^xsd:dateTime .
<job2> azg:Schedule "1Day"^^xsd:duration .
<job2> azg:ErrorPolicy "Ignore" .
<job2> azg:Statement "REFRESH VIEW <testView2>" .
```

In this example, the subject defines the job names: **job1** for scheduling and configuration of one scheduled job, and **job2** for the scheduling and configuration of a second job. Each predicate specifies a particular attribute or parameter of a scheduled job.

Tip

Each Cron graph is assigned a different Cron thread. The Cron thread acts as a "virtual user" that evaluates when to run the next job defined within the same graph. Each Cron thread runs only one job at a time per graph. If two jobs are scheduled for the same time, they are run sequentially. To execute Cron jobs concurrently, you can define Cron jobs in separate graphs, since jobs in different graphs are run using different Cron threads. For example, you could create one graph named "quickjobs" that defines many shorter jobs and create another graph that runs longer-executing jobs. Then the jobs from the two graphs could be run concurrently.

Load a Cron Graph

Once you have created a Cron graph file, you load the Cron graph into Graph Lakehouse using the following LOAD command:

```
LOAD <file:/<path>/<filename>.ttl> INTO GRAPH <graph_name>
```

For example:

```
LOAD <file:/tmp/cron1.ttl> INTO GRAPH <CronGraph1>
```

In this example, the triples stored in the **cron1.ttl** file are loaded into a graph named **CronGraph1**. It is this name, **CronGraph1**, that is added to the **cron_graphs** setting in `<install_path>/config/settings.conf` to run the scheduled jobs defined in **CronGraph1**. More details about configuring Graph Lakehouse to run Cron jobs are included in [Configure Graph Lakehouse to Run Cron Jobs](#).

Tip

As an alternative to specifying the graph name as part of the LOAD query, you can specify the name of the Cron graph within the triples file. For example:

```
PREFIX azg: <http://www.anzograph.com/> .
GRAPH <CronGraph1> {
  <job1> azg:BaseTime "2020-04-07:11:32"^^xsd:dateTime .
  ...
  <job1> azg:Statement "REFRESH VIEW <testView1>" .
  <job2> azg:BaseTime "2020-07-08:00:00"^^xsd:dateTime .
```

```
...
<job2> azg:Statement "REFRESH VIEW <testView2>" .
}
```

You could then load the Cron graph using the following LOAD command:

```
LOAD <file:/path/cron1.ttl>
```

Configure Graph Lakehouse to Run Cron Jobs

To configure Graph Lakehouse to run the jobs within Cron graphs, edit the `<install_path>/config/settings.conf` configuration file to specify values for the following two settings:

- **cron_graphs:** A comma-separated list of the Cron graph names to enable. For example, `cron_graphs=CronGraph1, CronGraph2`.
- **cron_graphs_recheck:** The interval (in number of seconds) to wait before re-checking the `cron_graphs` value to determine if there are changes, i.e, new or deleted graph names. For example, `cron_graphs_recheck=300`.

If a Cron graph is non-existent or empty, the associated Cron thread periodically checks at the specified interval whether the named graph is now loaded and has new jobs.

After changing `settings.conf`, restart Graph Lakehouse to apply the configuration changes.

Monitor Job Execution and Errors

Details about scheduled job run are logged to the following Graph Lakehouse system tables.

System Table	Logging Details
sth_query	SPARQL statements executed from jobs are logged to this table. To identify Cron job queries, look for the text cron: in the <code>label</code> column.
sth_cron_events	Activities related to execution of Cron jobs by their associated Cron threads are logged to this table. You can monitor this table for failed entries in the <code>event</code> column and take corrective action based on the failures.

System Table	Logging Details
sth_cron_graph	All scans of the Cron graphs (including Cron graph refreshes) are logged to this table.
sth_errors	All errors arising from scheduled job execution are logged to this table. If an error is caused by one of a Cron graph's job configuration settings, the <code>basic_text</code> column value will begin with Cron: . The Cron Graph Errors section below includes a list of Cron graph related errors.

You can query Graph Lakehouse's system tables using SPARQL queries in the following format:

```
SELECT * | list_of_variables
WHERE { table 'table_name' }
```

For example:

```
SELECT *
WHERE { table 'sth_cron_events' }
LIMIT 100
```

Tip

Entries in the `sth_cron_events` and `sth_errors` system tables are, by default, also spooled to disk so that they are incorporated into Crashdumps and Xrays.

Cron Graph Errors

The table below lists the errors that are returned for errors related to Cron job processing.

Cron Graph Error	Error Message
CronInvalidPredicate	"Cron: Invalid predicate"
CronOneDuration	"Cron: Multiple durations are being requested"

Cron Graph Error	Error Message
CronOneStatement	"Cron: Multiple statements are being requested"
CronOneFirstTime	"Cron: Multiple base/first times are being requested"
CronMissingStatement	"Cron: Missing Statement to execute"
CronMissingErrorPolicy	"Cron: Missing ErrorPolicy to execute"
CronConflictingErrorPolicy	"Cron: ErrorPolicy must be 'Ignore' if no RetryCount is specified"
CronUnknownErrorPolicy	"Cron: Unknown ErrorPolicy"
CronSingleErrorPolicy	"Cron: Only a single ErrorPolicy allowed per subject"
CronSingleStatement	"Cron: Only a single Statement allowed per subject"
CronSingleFirstTime	"Cron: Only a single FirstTime allowed per subject"
CronMustBeLiteralNotIRI	"Cron: Object must be a literal, cannot be an IRI"
CronWrongType	"Cron: Object wrong type"
CronStatementFailed	"Cron: Statement failed to execute, see system table sth_errors for more information"
CronRetryCountIncons	"Cron: Specifying an RetryCount requires a RetryInterval"
CronRetryCountPos	"Cron: RetryCount must be greater than 0"
CronIntervalPos	"Cron: Schedule, Delay, RetryInterval must be greater than 0"

Cron Graph Error	Error Message
CronMissingRetryCount	"Cron: Retry requires a RetryCount unless ErrorPolicy is Ignore"
CronBlockingUsers	"All SELECTS blocked, contact your system administrator" To unblock users from running SELECT queries, you can restart the database or run a <code>SET selects_blocked TO false</code> query.

Access & Analyze Data

This section includes information about the ways you can access the data that is stored in Graph Lakehouse.

In this section:

- [Use the Query & Admin Console](#) 479
- [Use the Graph Lakehouse CLI](#) 493
- [Use Third-Party Visualization Tools](#) 498
- [Access the SPARQL and RDF Endpoints](#) 509
- [Access Data with OData Protocol](#) 518
- [Create and Save Views](#) 537
- [Save Queries for Reuse](#) 544
- [SPARQL Query Language Reference](#) 552
- [Cypher Query Language Reference](#) 964

Use the Query & Admin Console

This topic provides information about using the Graph Lakehouse front end user interface, referred to here as the Query & Admin Console.

- [Log in to the Console](#)
- [Tour the Console](#)

Log in to the Console

The user interface supports the latest Safari, Google Chrome, Mozilla Firefox, and Microsoft Edge browsers.

1. Depending on whether you deployed Graph Lakehouse using Docker, Kubernetes with Helm, or the RHEL/Rocky installer, follow the appropriate instructions below to access the user interface:

Deployment	Instructions
Desktop Container Engine	You can use the desktop application to open the Graph Lakehouse container in a browser, or open a browser and go to the following URL: <code>http://127.0.0.1</code>. If you specified a port other than 80 for the host HTTP port when you deployed Graph Lakehouse, include that port in the URL. For example, <code>http://127.0.0.1:8888</code>.
Linux Container Engine	If you are accessing a container image on a remote Linux host, note the IP address of the host, and then open a browser and go to the following URL: <code>https://<host_IP_address></code>. If you mapped the container's HTTPS (8443) port to port 443 on the host when you deployed Graph Lakehouse, you do not need to specify a port. If you specified a port other than 443, include the port in the

Deployment	Instructions
	URL. For example, <code>https://10.100.0.1:8888</code>. Tip If you are using Docker locally on a Linux machine and need to know the IP address of the Graph Lakehouse container, you can run the following command: <pre>sudo docker inspect <container_name> grep '"IPAddress"' head -n 1</pre> For example: <pre>sudo docker inspect anzograph grep '"IPAddress"' head -n 1 "IPAddress": "172.17.0.2"</pre>
Kubernetes with Helm	Using the Graph Lakehouse cluster or external IP obtained from the <code>kubectl get service</code> command, open a browser and go to the following URL: <code>https://<IP_address></code> .
EL9 Installer	Use the following URL to access the console: <code>https://<host_IP_address>:<https_port></code> .

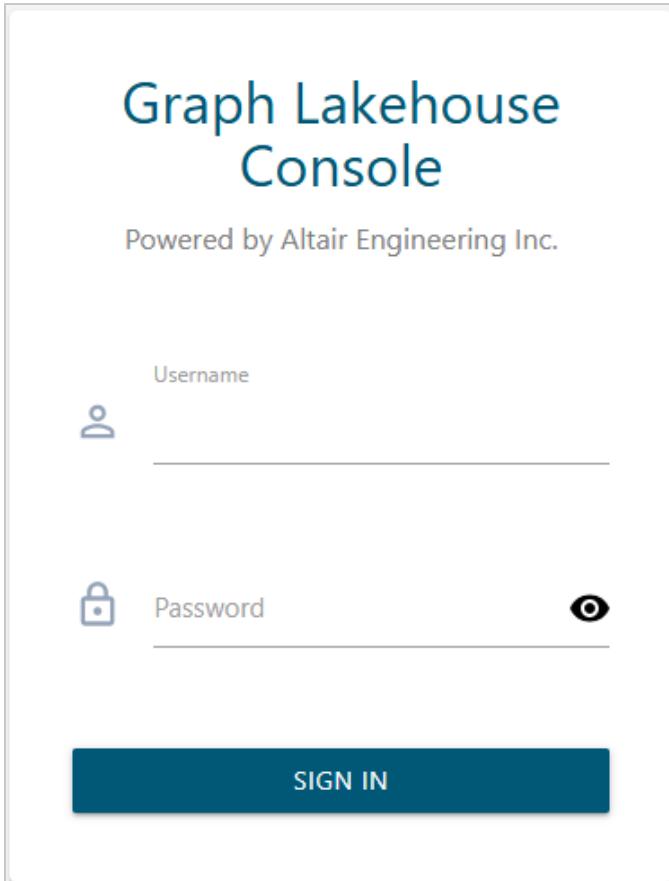
Note

If you use the HTTPS endpoint, your browser may warn you that the connection is not private. The warning is normal behavior. Graph Lakehouse servers use self-signed certificates, and browsers automatically trust only the certificates from well-known certificate authorities. For more information about certificate warnings, see [Security Certificate Errors](#) on the DigiCert website. Depending on your browser, follow the

appropriate instructions below to either bypass the warning and continue to the console or configure the browser to trust the certificate:

- On Chrome, click the **Advanced** link at the bottom of the page and then click the **Proceed to *ip* (unsafe)** link.
- On Safari, click the **Show Details** button and then click **Visit Website** to import the certificate.
- On Firefox, click **Advanced** and then click **Add Exception**. On the next screen, click **Add Security Exception** to confirm the exception for the endpoint.

The browser displays the login screen.

The image shows a login screen for the Graph Lakehouse Console. At the top, the text "Graph Lakehouse Console" is displayed in a large, blue, sans-serif font. Below this, in a smaller, grey font, it says "Powered by Altair Engineering Inc.". The login form consists of two input fields. The first field is labeled "Username" in a small, grey font, with a person icon to its left. The second field is labeled "Password" in a small, grey font, with a lock icon to its left and a toggle icon (an eye) to its right. Below the password field is a dark blue button with the text "SIGN IN" in white, uppercase letters.

2. On the login screen, type the username and password for the admin user that you set up during the deployment. For Docker installs, type **admin** as the user name and **Passw0rd1** as the password.

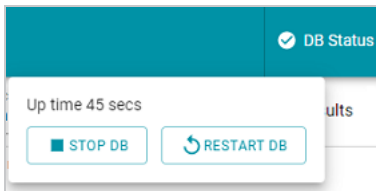
3. Then click **Sign In**. After successful authentication, the Query Console tab is displayed.

Tour the Console

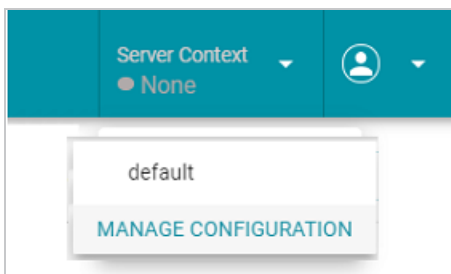
The Query & Admin Console application top menu bar provides two tab selections, **Query Console** ([Query Console Tab](#)) and **Admin** ([Admin Tab](#)). On the right side of the screen, the top menu provides the **DB Status** option, the Server Context drop-down menu, and the user drop-down menu. The list below describes each item.



- **DB Status** shows the status of the database. A check mark icon indicates that the database is running, and an X icon indicates the database is stopped. Click **DB Status** to access the options to start, stop, or restart the database:

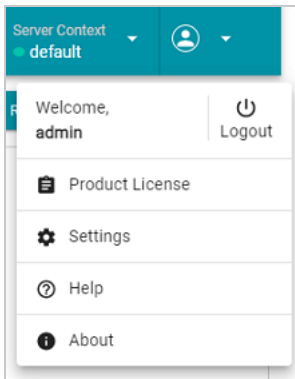


- The **Server Context** menu enables you to manage the connections to multiple Graph Lakehouse instances as well as set up an LDAP configuration for authentication.



- The user menu includes options to log out of the console, view the product license, and access the documentation (**Help**). The **Settings** option takes you to the Server Configuration page for server context and LDAP configuration, and the **About** option displays information about the current Graph Lakehouse database and front end versions running in your environment. The Product License option displays information about the current Graph Lakehouse license and allotted resources. In addition, this option provides links to request

new or enhanced licenses and upload new license keys. See [Install or Upgrade a License](#) for more information about licensing options and requesting a new license.



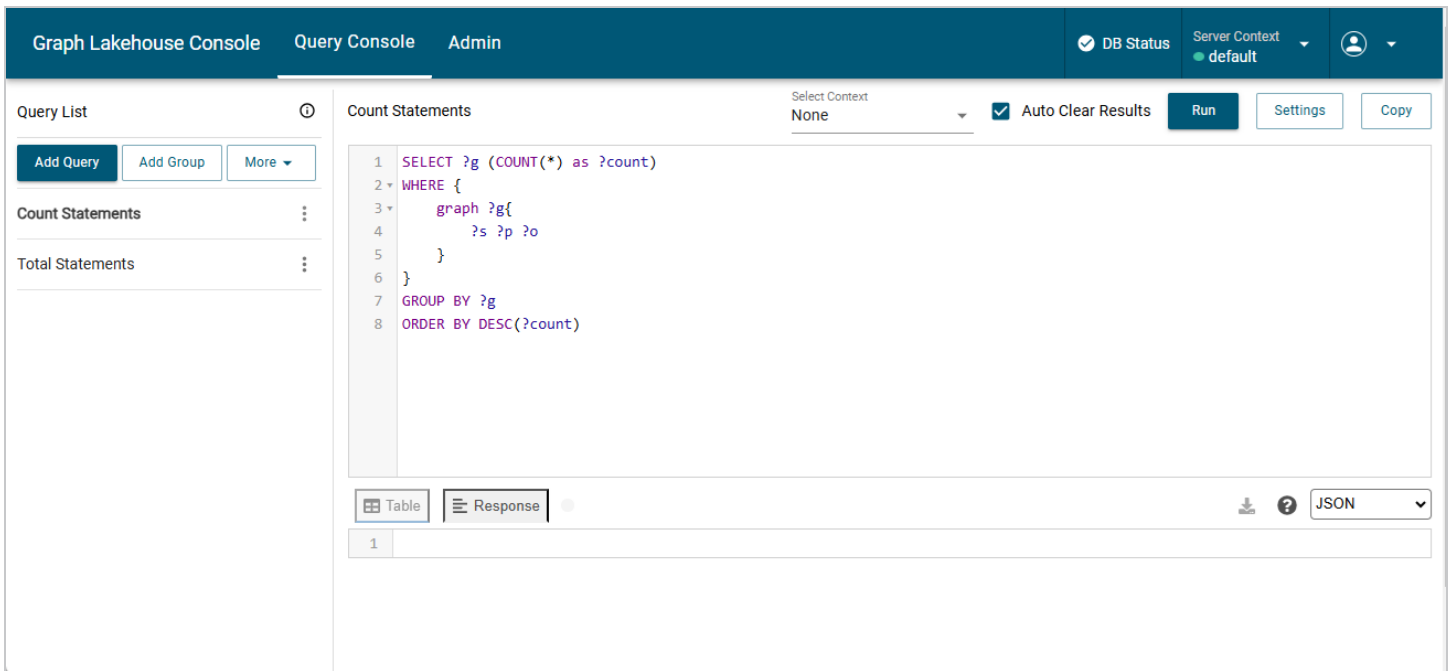
Query Console Tab

When you open the Query Console tab, the left sidebar navigation pane, labeled Query List, displays a number of predefined menu options and groups.

The Console provides two default queries, **Count Statements** and **Total Statements**. The Count Statements query returns a list of each named graph and the number of statements or triples in the graph. The Total Statements query returns the total number of statements in all named graphs.

Clicking the **Add Query** button lets you enter and run new queries and other SPARQL statements. The **Add Group** button lets to define new menu option groups to organize queries, and the **More** button lets you import from and export queries to your local file system environment. Clicking on the three-stacked dot icon (⋮) next to an existing query lets you rename, export, or delete the query.

In the right side window, the Console provides an editing and display window along with various button, checkboxes, and windows, pertaining to your current sidebar menu or query selection. For example, if you've already loaded the sample tickit graph into Graph Lakehouse, selecting the Count Statements option would display the following:



At the top of the right-side window, the Console displays the **Server Context** drop-down menu, the **Auto Clear Results** checkbox and **Run**, **Settings**, and **Copy** buttons, which let you run the displayed query or perform other actions. If you choose the Add Query option, the Console clears the right-side query window, allowing you to enter a new query or SPARQL query.

Note

As you enter a new query, the Console validates the syntax of the SPARQL statement you are entering. If you specify syntax that is invalid, the Console displays the invalid syntax in red. In that case, you can click on the red Info (ⓘ) icon to see suggested or allowed syntax elements you can enter at a specific position in your SPARQL statement.

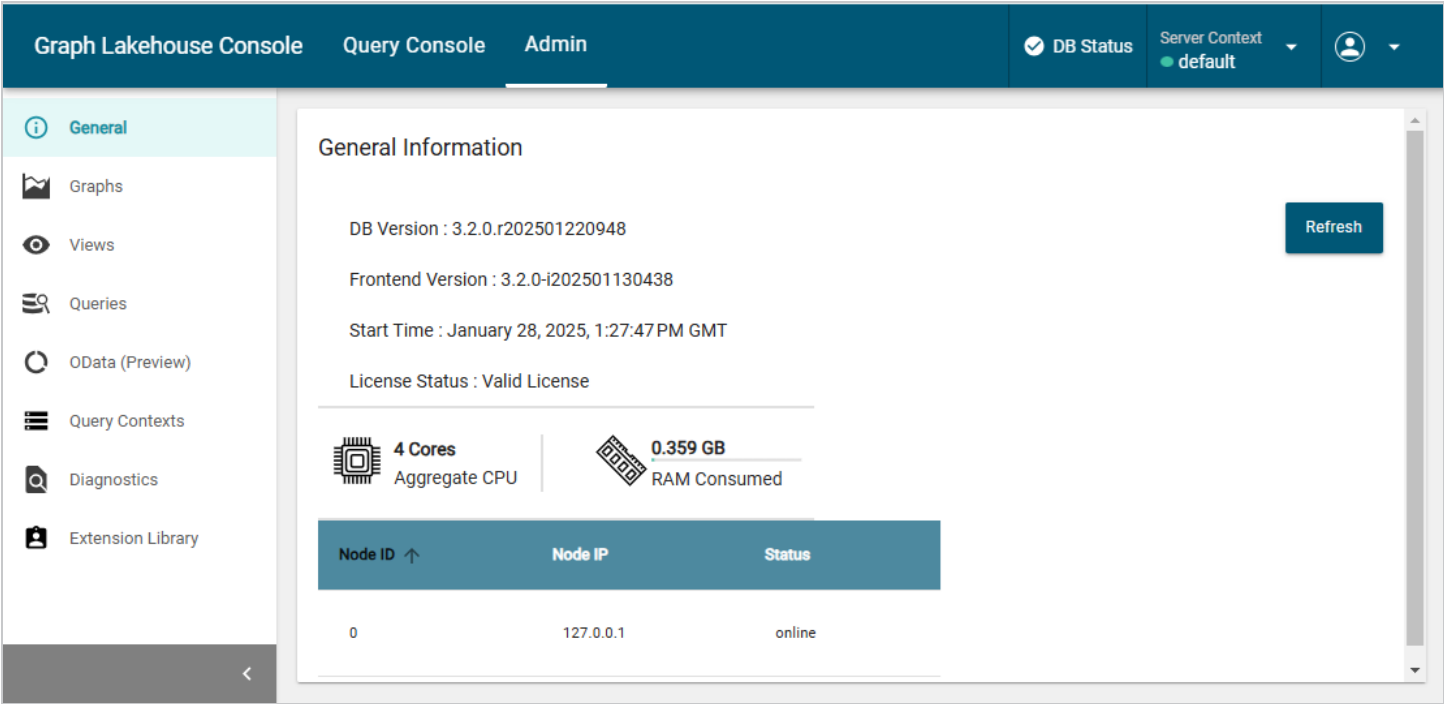
Below the SPARQL query window, the Console displays the results from running a query, along with options to control the results format, limits to page size, and so on. You can toggle between **Table** and **Response** options, to show the query result in either a tabular format, or when you click **Response**, view the query result in the specified format, by default, JSON.

Admin Tab

When you open the Admin tab, the left sidebar navigation panel displays a variety of menu options pertaining to the common operations that an Graph Lakehouse administrator or analyst might perform. This section provides a basic description of each option and provides references to additional information.

General

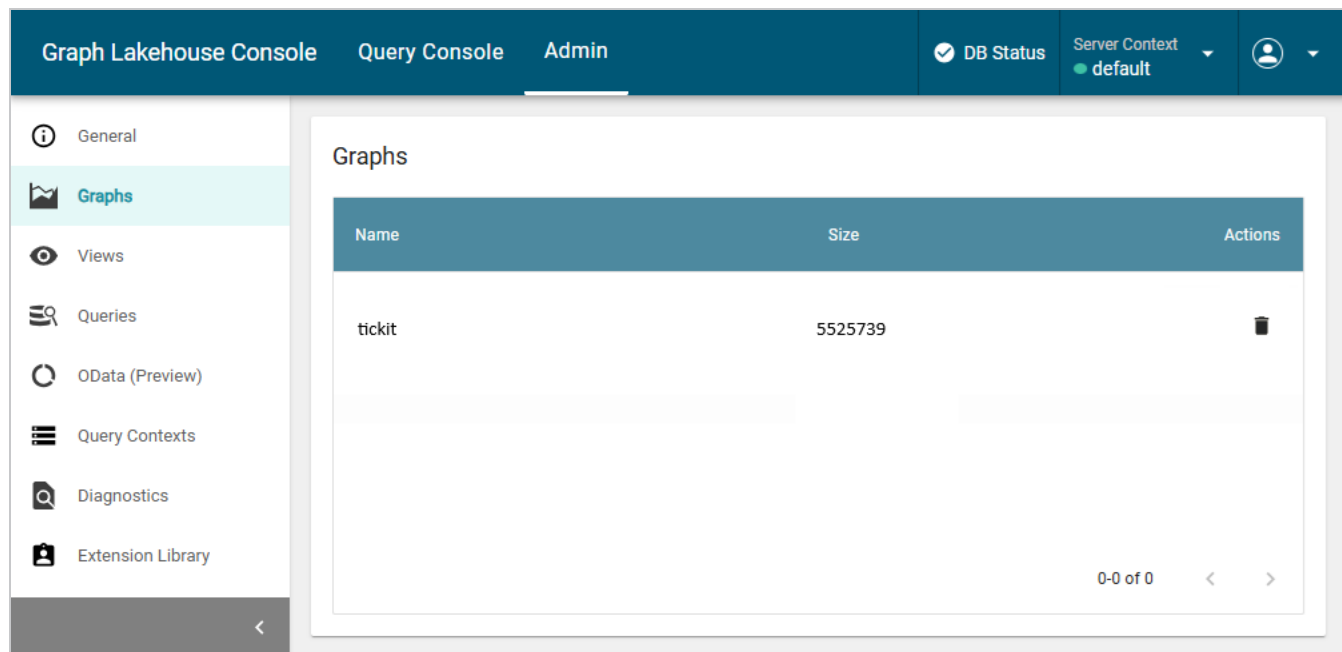
The General tab selection displays the database and console version information, database start time, and license status. It also provides details such as the number of cores utilized on the Graph Lakehouse server or cluster, memory usage, and total memory resources available to Graph Lakehouse.



Clicking the **Upgrade License** button takes you to the Graph Lakehouse License Offerings web site, allowing you to view various product download and licensing options. For instructions on upgrading the license, see [Install or Upgrade a License](#).

Graphs

The **Graphs** menu option displays a list of the graphs in memory as well as a count of the number of triples that each graph contains. For example:



The screenshot shows the Graph Lakehouse Console interface. The top navigation bar includes 'Graph Lakehouse Console', 'Query Console', and 'Admin'. On the right, there are links for 'DB Status', 'Server Context' (set to 'default'), and a user profile icon. The left sidebar contains a menu with options: 'General', 'Graphs' (highlighted), 'Views', 'Queries', 'OData (Preview)', 'Query Contexts', 'Diagnostics', and 'Extension Library'. The main content area is titled 'Graphs' and displays a table with the following data:

Name	Size	Actions
tickit	5525739	

At the bottom right of the table, there is a pagination indicator showing '0-0 of 0' and navigation arrows.

If you want to delete a graph from the database, click the trash can icon (🗑️) in the row for the graph that you want to drop. The console displays a dialog box to confirm that you want to delete the graph. Click **OK** to remove the graph.

Views

The **Views** menu option displays a list of the views that have currently been defined in Graph Lakehouse. For example:

Graph Lakehouse Console

Query Console

Admin

DB Status

Server Context

default

General

Graphs

Views

Queries

OData (Preview)

Query Contexts

Diagnostics

Extension Library

Views

Type	Name	Query	Actions
Named Query	stc_aclobj	select ?objname ?objtype ?objid ?container ?aclinfo ?recordtin	
Named Query	sth_acl	select ?xrowid ?query ?time ?user ?action_type ?detail where {	
Named Query	sth_aggr	select ?xrowid ?query ?pnid ?slice ?segment ?step ?starttime ?	
Named Query	sth_aggr_distinct	select ?xrowid ?query ?slice ?segment ?step ?col ?rows ?bytes	

1-20 of 179

For each view, the display shows the view name, the type (materialized or non-materialized), and the query on which the view was defined. If you want to delete a view from the database, click the trash can icon (🗑️) in the row for the view that you want to delete. The console displays a dialog box to confirm that you want to delete the view. Click **OK** to remove the view.

Queries

The Admin **Queries** menu option provides access to the query history log, which shows a list of the queries that have been run against Graph Lakehouse. To view the list, select the **Queries** menu option. The following display hows the query history and provides the option to cancel all running queries. For example:

Graph Lakehouse ConsoleQuery ConsoleAdmin

DB StatusServer Contextdefault

General

Graphs

Views

Queries

OData (Preview)

Query Contexts

Diagnostics

Extension Library

Queries

Cancel all running queriesReload

ID	Label	Status	Start Time	Duration	Query	Actions
446	TotalStatements:...	COMPLETED	Jan 28, 2025 08:28:07	159 ms	SELECT (COUNT(*) as ?count) WHERE	

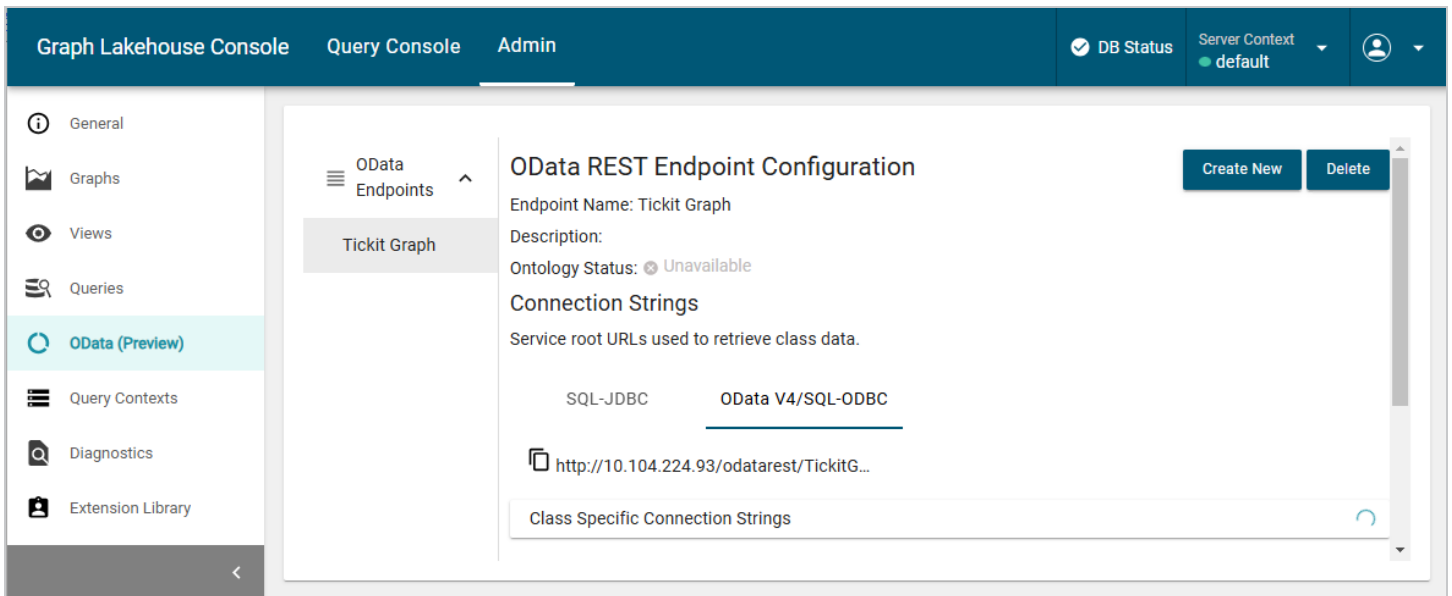
1-1 of 1<>

For each query, the screen shows the query ID, label, and status, as well as the start time and duration in milliseconds. The last column shows the query text. You can click the new window icon () next to a query to copy that query into the Query editing window where you can edit and/or re-run the query.

OData (Preview)

Graph Lakehouse provides a "Data on Demand" service that allows users to define RESTful API endpoints using Open Data Protocol (OData)-based data feeds, which allow web-based access to Graph Lakehouse graph data. The RESTful API endpoints allow web clients to use simple HTTP messages to publish and edit resources that are identified using URLs and are defined in a data model.

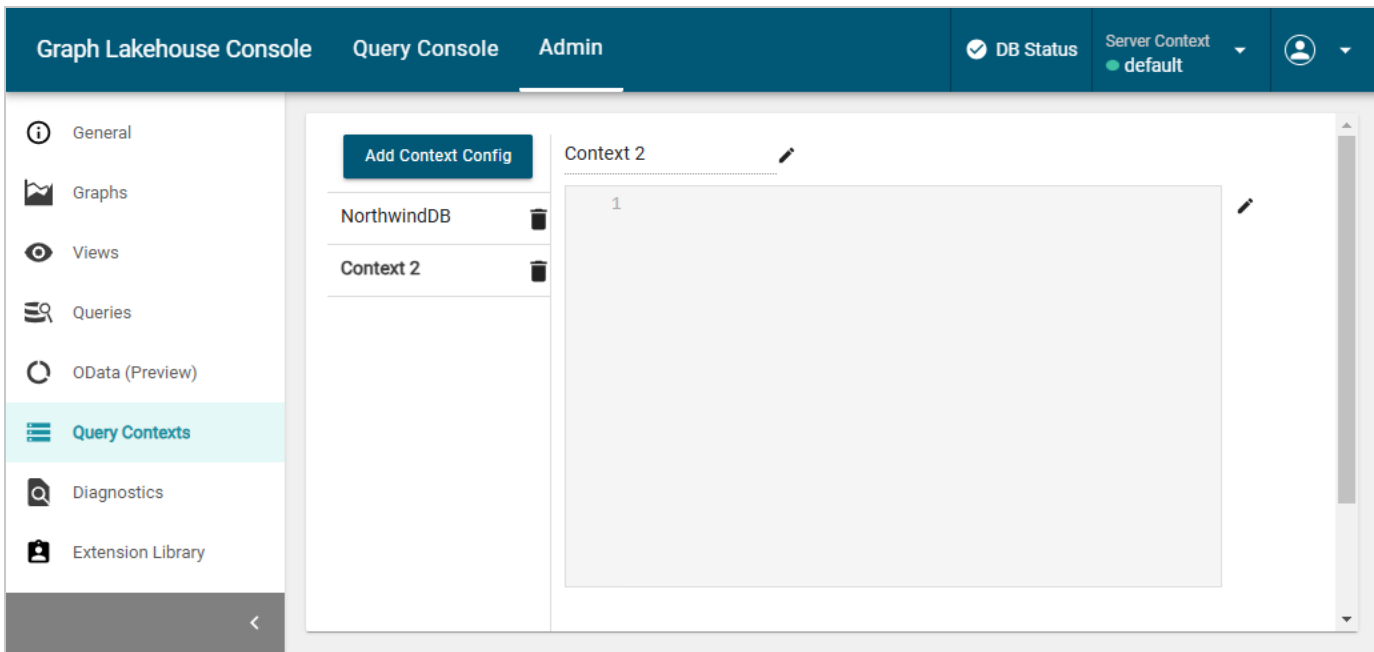
The **OData (Preview)** menu option lets you view existing Data on Demand endpoints as well as create new endpoints. The following screen shows an example of a Data on Demand endpoint that provides access to a Tickit graph.



For more information and instructions on creating Data on Demand endpoints, see [Access Data with OData Protocol](#).

Query Contexts

The Admin **Query Contexts** menu option lets you manage query contexts, which define sensitive data source connection details like keys, tokens, and user credentials. Queries that are run against a source can simply reference the keys in a context to avoid including sensitive information in the request. The following screen shows the display of currently defined query contexts and the options to add new context configurations as well as edit or delete contexts.



For more information and instructions on creating Query Contexts, see [Use a Query Context](#).

Diagnostics

The Admin **Diagnostics** menu option displays and lets you download any existing Xray snapshot diagnostic files that Graph Lakehouse has generated in response to an error or database crash. When Altair Support requests Graph Lakehouse diagnostic files for troubleshooting an issue, you can quickly retrieve the files here.

There are two types of Graph Lakehouse diagnostic files:

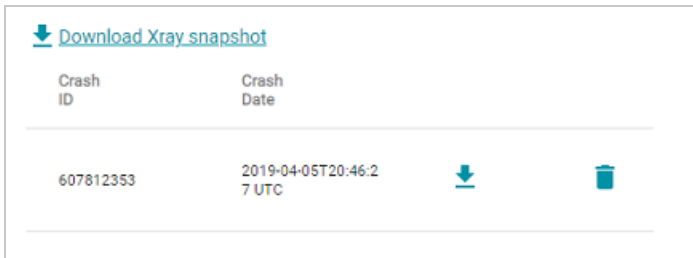
- **Xray:** Xrays are generated on-demand. If you encounter an error and the database remains running, you generate an Xray to produce the diagnostic files.
- **Crash:** If you encounter an error that crashes the database, Graph Lakehouse automatically generates a crash file that contains diagnostic information about the crash.

Note

See [Getting Support](#) for more information about the files, troubleshooting issues, and obtaining Altair Engineering Inc. support.

To retrieve an Xray file:

1. Select the **Diagnostics** menu option from the Admin sidebar panel. The console displays the available options. For example:



The screenshot shows a web interface with a link 'Download Xray snapshot' at the top. Below it is a table with two columns: 'Crash ID' and 'Crash Date'. The table contains one row with the values '607812353' and '2019-04-05T20:46:27 UTC'. To the right of the table, there are two icons: a download arrow and a trash can.

Crash ID	Crash Date
607812353	2019-04-05T20:46:27 UTC

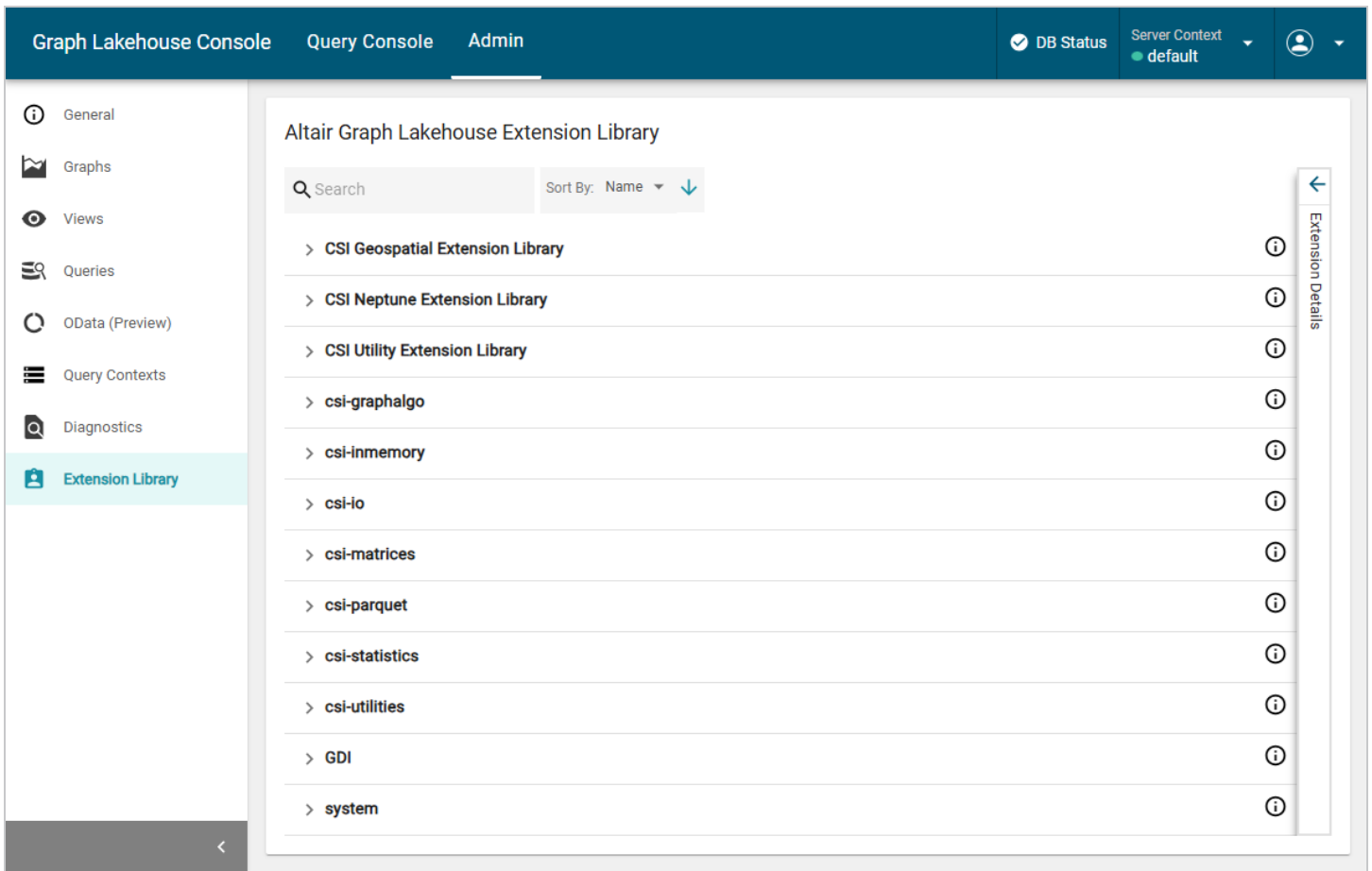
2. If you want to retrieve an xray, click the **Download Xray snapshot** link. Graph Lakehouse creates the xray and produces a tarball with a .xray extension. The console downloads the .xray file to your computer.
3. If you want to retrieve a crash dump, click the **Refresh** button to refresh the list of available crash dump .xray files. Click the file name that you want to download. The console downloads the .xray file to your computer.

Note

Xray and crash dump files that you download are already compressed. Do not compress the files before sending them to Altair when they are requested for troubleshooting an issue.

Extension Library

The Admin **Extensions** menu option provides a display of extension libraries and extensions currently installed in Graph Lakehouse. From the list of libraries, you can expand or collapse library items to show all the extensions defined within those libraries. For example:



For the library and extension display, you can click the Info icon next to an item to obtain additional information about that library or extension.

User Role Management

This option only appears if you have enabled Graph Lakehouse access control. Selecting the **User Role Management** option lets you create roles and define permissions that control access to Graph Lakehouse data and determine the operations users can perform after they log in. For more information, see [Create and Manage Roles from the Console](#).

Use the Graph Lakehouse CLI

You can use the AZGI command line interface to run commands and access data in Graph Lakehouse. AZGI uses SSL protocol to interact with the database. The client exists in the `<install_path>/bin` directory. In a container deployment, the installation path is `/opt/anzograph`. On RHEL/Rocky deployments, the installation path is customizable. The default path is `/opt/altair/anzograph`. In a cluster, use `azgi` on the leader node only.

Tip

Graph Lakehouse also includes an alternate command line interface (AZGBOLT), which uses the Bolt protocol and enables you to run Cypher queries. For information, see [Using the Cypher CLI \(AZGBOLT\)](#).

AZGI Syntax

This section describes the available `azgi` command options. To view the list of options from the command line, run `azgi -help`.

```
azgi [-f <filename>] [-c "<command>"] [-set <param>=<value>] [-h <host_url>] [-p  
<port>]  
      [-u <username>:<password>] [-v] [-timer] [-raw] [-csv] [-json] [-xml] [-silent]  
      [-nohead] [-noprogress] [-maxwid <width>] [-wide]  
      [-nossll] [-o <file>] [-certs <directory>] [-context <json_file>]
```

-f <filename>

Runs the specified SPARQL query file. For example, the following command runs the query or queries in the `query.rq` file:

```
azgi -f /home/user/query.rq
```

-c "<command>"

Runs the command in quotation marks. For example, this command runs a query:

```
azgi -c "select distinct ?eventname from <http://anzograph.com/tickit>  
where {?event <http://anzograph.com/tickit/eventname> ?eventname} limit 100"
```

You can include multiple `-c` options to run multiple commands. For example, this command runs two queries:

```
azgi -c "select * from <http://anzograph.com/tickit> where {?s ?p ?o}"  
-c "select distinct ?likes from <http://anzograph.com/tickit> where  
{?person <http://anzograph.com/like> ?likes}"
```

And this command sets the `query_label` setting to "events" before running the query:

```
azgi -c "set query_label to 'events'" -c "select distinct ?event  
from <http://anzograph.com/tickit> where  
{?event <http://anzograph.com/eventname> ?event} limit 100"
```

-set <param>=<value>

Sets or changes parameter values in query files. For example this command runs the query in the `query_summary.rq` file with the `$query` parameter set to 2:

```
azgi -set query=2 -f query_summary.rq
```

-h <host_url>

Connects to a remote Graph Lakehouse server. For example, the following statement runs a query against Graph Lakehouse on host 10.104.55.27:

```
azgi -h 10.104.55.27 -c "select * from <http://anzograph.com/tickit>  
where {?s ?p ?o} limit 100"
```

-p <port>

Used to connect to the database on a non-default port. The default azgi port is 8256.

-u <username>:<password>

Connects to the database with credentials (basic authentication). If you type `-u <username>` and exclude the password, the client prompts for the password. For example, the following command uses basic authentication to run a query:

```
azgi -u admin:Passw0rd1 -c "select ?g where {graph ?g {?s ?p ?o}} limit 10"
```

-v

Displays verbose output such as client connection details. For example:

```
azgi -v -c "select distinct ?p from <http://anzograph.com/tickit>  
where {<http://anzograph.com/tickit/person1> ?p ?o}"
```

```
Connecting to host=localhost port=8256  
IPv4: connected  
POST /sparql HTTP/1.1  
Host: Anon  
Accept: application/sparql-results+xml  
User-Agent: azgi  
Connection: keep-alive  
Content-Length: 106  
Content-Type: application/sparql-query  
select distinct ?p from ...  
  
HTTP/1.1 200 OK  
Date: Fri, 14 Apr 2023 21:37:16 GMT  
Server: AnzoGraph  
Access-Control-Allow-Origin: *  
X-AnzoGraph-QueryExecution-Time: 2837  
Connection: close  
Content-Type: application/sparql-results+xml; charset=utf-8  
...
```

-timer

Reports query execution time in milliseconds.

-raw

Returns query results in raw XML, JSON, or CSV format, depending on what format you request.

-csv

Returns results in CSV format.

-json

Returns results in JSON format.

-xml

Returns results in XML format.

-silent

Suppresses the query output.

-nohead

Suppresses headings in query results.

-noprogess

Suppresses the progress messages that are displayed for queries that are inflight.

-maxwid <width>

Overrides the default maximum column width of 50 characters for tabular query results. Using the [-wide](#) option described below is equivalent to `maxwid 60000`.

-wide

Increases the column width for tabular query results from the default 50 characters to 60,000 characters. Equivalent to `-maxwid 60000`.

-noss1

Instructs the client to make a non-SSL (HTTP) connection to the database. When using AZGI to send a request to a remote Graph Lakehouse server, include the `-h <host_url>` and `-p <port>` options when using `-noss1`. The default HTTP port is 7070. For example:

```
azgi -noss1 -h 10.100.0.20 -p 7070 -c "select (count(*) as ?cnt) where {?s ?p ?o}"
```

-o <file>

Writes the response to the specified file. If the file exists, it is overwritten.

Note

When you include this option to redirect output to a file, all progress messages will also be written to the file unless you also specify the [-noprogess](#) option. Altair recommends that you include `-noprogess` any time you output results to a file.

-certs <directory>

Instructs the client to make a certified secure connection to the database. The Graph Lakehouse certificates are **ca.crt**, **serv.crt** (public key), and **serv.key** (private key) in the `<install_path>/config` directory. When sending requests to a remote Graph Lakehouse server, you can copy the certificates to the server where you are using AZGI. For example, the following command runs a query on a remote Graph Lakehouse server. The command makes a certified connection using the Graph Lakehouse certificates, which were copied to the `/home/user/certs` directory:

```
azgi -h 10.10.10.01 -certs /home/user/certs  
-c "select ?g where {graph ?g {?s ?p ?o}} limit 100"
```

This command runs the same query from the Graph Lakehouse server.

```
azgi -certs /opt/altair/anzograph/config  
-c "select ?g where {graph ?g {?s ?p ?o}} limit 100"
```

-context <json_file>

Specifies the query context file on the Graph Lakehouse server file system to use with the request. Context files are JSON-formatted files with key-value pairs that provide connection details, such as user credentials, keys, and tokens, for authentication against data sources. For more information, see [Use a Query Context](#).

Use Third-Party Visualization Tools

A variety of graph visualization applications can be used to access data in Graph Lakehouse via the SPARQL endpoint (see [Access the SPARQL and RDF Endpoints](#) for information). For demonstrations, Altair utilizes two third-party applications:

- **Apache Zeppelin:** Altair offers an Apache Zeppelin Docker image for download. The Zeppelin image includes a custom SPARQL interpreter for securely connecting to Graph Lakehouse.
- **Jupyter Notebook:** Existing Jupyter Notebook or JupyterLab environments can run queries against the Graph Lakehouse SPARQL endpoint.

This topic provides information about integrating Zeppelin with Graph Lakehouse. It also provides instructions for accessing Graph Lakehouse from your existing Jupyter installation.

- [Zeppelin Notebook Integration](#)
- [Jupyter Notebook Integration](#)

Zeppelin Notebook Integration

This section provides instructions for deploying the [Cambridge Semantics Apache Zeppelin image](#) with Docker, connecting to Graph Lakehouse, and optionally downloading and running the tutorial notebook. The Zeppelin deployment includes an integrated SPARQL interpreter that enables users to make a secure, authenticated connection to Graph Lakehouse using gRPC protocol.

1. [Deploying Zeppelin](#)
2. [Connecting to Graph Lakehouse](#)
3. [Downloading the Tutorial Notebook](#)

Deploying Zeppelin

Tip

If you use Docker on Linux, you might want to follow the steps in [Post-installation steps for Linux](#) to make sure that a non-root user can run Docker commands and you do not need to include "sudo" in the commands below.

1. If necessary, start Docker for Linux or the Docker Desktop application for Mac or Windows. If you are on Mac, open the Terminal app. If you are on Windows, open PowerShell.

Note

Docker caches images on the docker host. If you have deployed a Zeppelin container previously, that image is cached on the host and will be used to redeploy Zeppelin. If you want to deploy the latest release, first pull the latest image. To do so, run the following command from the command line, and then proceed to the next step.

```
docker pull cambridgesemantics/contrib-zeppelin:latest
```

You can deploy alternate Zeppelin versions by replacing the "latest" tag with any of the tags that are available on the Cambridge Semantics [Zeppelin Docker Hub](#) site.

2. If you are deploying the Zeppelin container for the first time, Altair recommends that you create a directory on the local file system where Zeppelin notebooks can be saved. When you deploy Zeppelin, you can map the notebook directory in the container to the notebook directory on the local file system. This way the notebooks are shared, and if you remove the Zeppelin container, the local file system retains a copy of any notebooks you created. If you redeploy Zeppelin later, the new container can be mapped to the same local directory and access the existing notebooks. To create the directory, navigate to a location on the host and run the following command to create a notebook directory in the current directory:

```
mkdir $PWD/notebook
```

Note

On Mac and Linux, Docker is configured by default to allow local directories to be shared with containers. On Mac, the /Users, /Volumes, /private, and /tmp directories are shared. If necessary, you can configure additional locations in Docker **Preferences > Resources > File Sharing**. On Windows, Docker is not configured to share local directories by default. Configure sharing by going to Docker **Settings > Resources > File Sharing** and selecting the **C** checkbox to share the C drive. Then click **Apply & Restart** to apply the change.

3. Run the following Docker command to deploy the Zeppelin container. The command instructs Docker to start Zeppelin and configure HTTP access to the application by mapping the container port to the HTTP port on the local host. In addition, the `-v $PWD/notebook:/notebook` statement creates the bridge between the notebook directory you created on the local file system and the container's /notebook directory.

Note

The last line in the command varies. Choose one option depending on whether you want only the application that is empty of notebooks or the image that contains several Altair-supplied sample notebooks that demonstrate Graph Lakehouse's Data Science functions.

```
docker run -p <http_host_port>:8080 --name=<container_name> \
-v $PWD/notebook:/notebook \
-e ZEPPELIN_NOTEBOOK_DIR='/notebook' \
-e ZEPPELIN_WEBSOCKET_MAX_TEXT_MESSAGE_SIZE=10240000 \
-d cambridgesemantics/contrib-zeppelin:<tag> \
# To deploy Zeppelin without notebooks, end the command with the following
line:
    /zeppelin/bin/zeppelin.sh
# To deploy Zeppelin with sample notebooks, end the command with this line:
/bin/bash /docker-entrypoint.sh
```

The list below describes each of the parameters:

- **host_http_port** is the port on the local host to use for HTTP access to the Zeppelin user interface. In the container, the user interface binds to port 8080 for HTTP access. Altair recommends that you specify **8080** to map the container's HTTP port to port 8080 on the local host. If port 8080 is in use, specify an alternate port for **host_http_port**.
- **container_name** is the short name to use to identify the Zeppelin container. For example, **zeppelin**.
- **tag** is the tag from the Cambridge Semantics [Zeppelin Docker Hub](#) site that identifies the version of Zeppelin to deploy. If you pulled an image in the first step, this tag should match the tag from the pull command. Usually the **latest** tag is specified so the most recent release is deployed.
- **/zeppelin/bin/zeppelin.sh**: Ending the command with this line excludes Altair's sample notebooks from the image. The base Zeppelin application will be deployed, and it will not include sample notebooks.
- **/bin/bash /docker-entrypoint.sh**: Ending the command with this line includes Altair's sample notebooks in the image. The Zeppelin application will be deployed and pre-loaded with many sample notebooks that demonstrate the data science functions of Graph Lakehouse.

For example, the following command deploys Zeppelin with all of the sample notebooks:

```
docker run -p 8080:8080 --name=zeppelin \
  -v $PWD/notebook:/notebook \
  -e ZEPPELIN_NOTEBOOK_DIR='/notebook' \
  -e ZEPPELIN_WEBSOCKET_MAX_TEXT_MESSAGE_SIZE=10240000 \
  -d cambridgesemantics/contrib-zeppelin:latest \
  /bin/bash /docker-entrypoint.sh
```

This command deploys an empty Zeppelin container without any sample notebooks:

```
docker run -p 8080:8080 --name=zeppelin \
  -v $PWD/notebook:/notebook \
  -e ZEPPELIN_NOTEBOOK_DIR='/notebook' \
  -e ZEPPELIN_WEBSOCKET_MAX_TEXT_MESSAGE_SIZE=10240000 \
  -d cambridgesemantics/contrib-zeppelin:latest \
  /zeppelin/bin/zeppelin.sh
```

Note

Windows PowerShell will not run the command above in its current format. Copy the command and paste it into a text editor. In the editor, remove the line breaks and \ characters. Then paste the edited version into PowerShell and run it.

When the prompt returns the container ID, the container is running. For example:

```
6c67e9f111cb55fb9a44208ce1802256acf459acbce1e0250b70646492d32642
```

Zeppelin is now installed and ready to use. On Mac and Windows, you can open Zeppelin from the Docker Dashboard, or you can open a browser and go to the following URL:

```
http://127.0.0.1:port
```

Where **port** is the HTTP port that you specified in the Docker run command, typically **8080**.

On Linux, open a browser and go to the following URL:

```
https://host_IP_address:port
```

Where **host_ip_address** is the IP address of the host server, and **port** is the HTTP port that you specified in the Docker run command, typically **8080**.

Note

If you are using Docker locally on a Linux machine or you have the Graph Lakehouse and Zeppelin containers in the same Docker instance and need to know the IP address of the Zeppelin container, you can run the following command:

```
docker inspect container_name | grep '"IPAddress"' | head -n 1
```

For example:

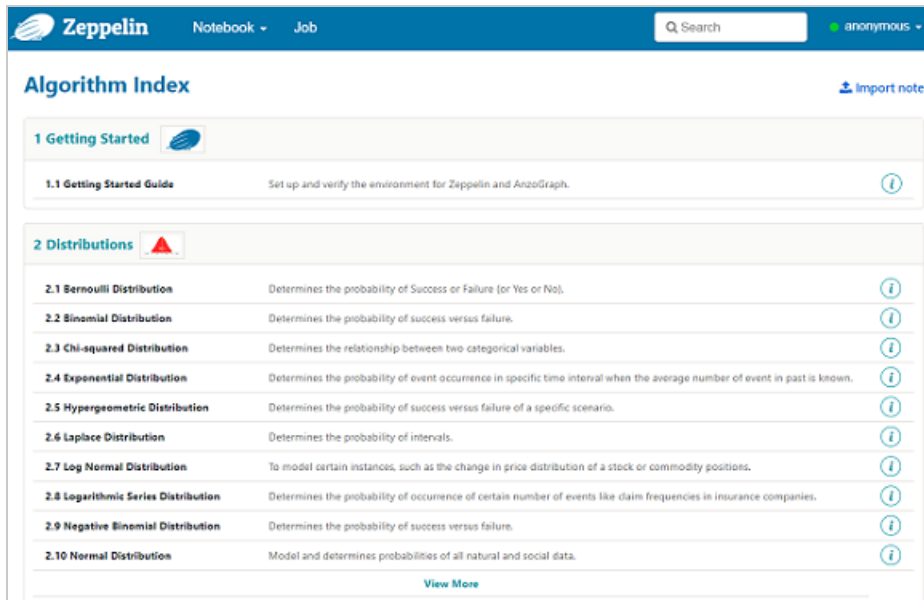
```
docker inspect zeppelin | grep '"IPAddress"' | head -n 1
```

```
"IPAddress": "172.17.0.3"
```

If you are running Docker locally on a Windows computer, you can run the following command:

```
docker inspect zeppelin | select-string '"IPAddress"'
```

- If you included the sample notebooks, the Cambridge Semantics index page is displayed, which lists the sample notebooks that are included in the image. For example:



Each notebook provides details and example usage for one of the Graph Lakehouse Data Science functions. You can click a notebook name to open the file. It might take some time to load all of the contents. To run the queries in a sample notebook or write and run your own queries run against Graph Lakehouse, connect Zeppelin to Graph Lakehouse by following the instructions in [Connecting to Graph Lakehouse](#) below. For more information about the data science functions, see [Data Science Library](#). For information about using Zeppelin, see the [Zeppelin Documentation](#).

- If you excluded notebooks, the Zeppelin Welcome page is displayed:



Connect Zeppelin to your Graph Lakehouse deployment by following the instructions in [Connecting to Graph Lakehouse](#) below. If you want to get started with a sample notebook, see [Downloading the Tutorial Notebook](#).

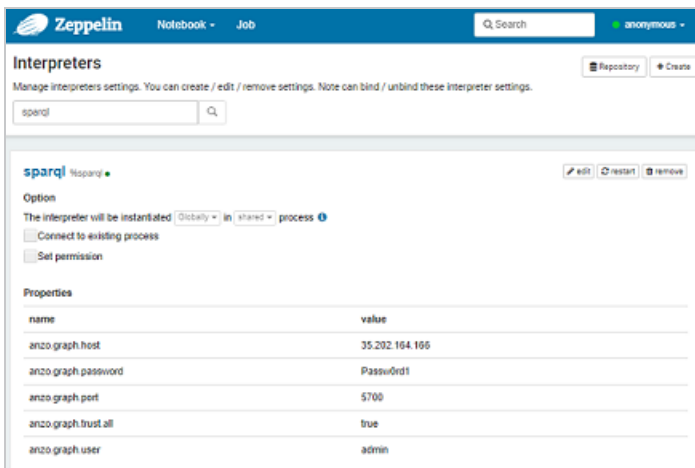
Connecting to Graph Lakehouse

To connect Zeppelin to Graph Lakehouse, you first choose an interpreter for Zeppelin to use to connect with Graph Lakehouse and then specify the connection parameter settings needed to establish the interpreter connection with the Graph Lakehouse database.

Important

To connect to an Graph Lakehouse database, you must first have installed Graph Lakehouse, and the database must be accessible by the Zeppelin Docker container. For more information, see [Deploy](#).

1. On the top right of the Zeppelin screen, click the **anonymous** drop-down list and select **Interpreter**. The Interpreters screen opens.
2. In the **Search interpreters** field at the top of the screen, type "sparql" and find the SPARQL interpreter. For example:



The screenshot shows the Zeppelin web interface. At the top, there's a header with the Zeppelin logo, 'Notebook - Job', a search bar, and a user dropdown set to 'anonymous'. Below the header, the 'Interpreters' section is active. It has a sub-header 'Manage interpreters settings. You can create / edit / remove settings. Note can bind / unbind these interpreter settings.' and buttons for 'Repository' and 'Create'. A search bar contains 'sparql'. Below the search bar, the 'sparql' interpreter is selected. It has buttons for 'edit', 'restart', and 'remove'. Under 'Option', there's a dropdown for 'The interpreter will be instantiated' set to 'Globally', and checkboxes for 'Connect to existing process' and 'Set permission'. Below that is a 'Properties' table.

name	value
anzo.graph.host	35.202.164.166
anzo.graph.password	Password1
anzo.graph.port	5700
anzo.graph.trust.all	true
anzo.graph.user	admin

3. Click the **edit** button and modify the interpreter to enter your Graph Lakehouse deployment details and make a secure connection to the database. The list below describes each interpreter setting:

- **anzo.graph.host:** The IP address of the Graph Lakehouse host. If Graph Lakehouse is running in the same Docker instance as Zeppelin, run the following command to return the Graph Lakehouse container IP address:

```
docker inspect container_name | grep '"IPAddress"' | head -n 1
```

For example:

```
docker inspect anzograph | grep '"IPAddress"' | head -n 1
```

This command returns the IP address of the Graph Lakehouse Docker container as shown below:

```
"IPAddress": "172.17.0.2",
```

Note

If you are running Docker locally on a Windows computer, you can run the following command:

```
docker inspect anzograph | select-string '"IPAddress"'
```

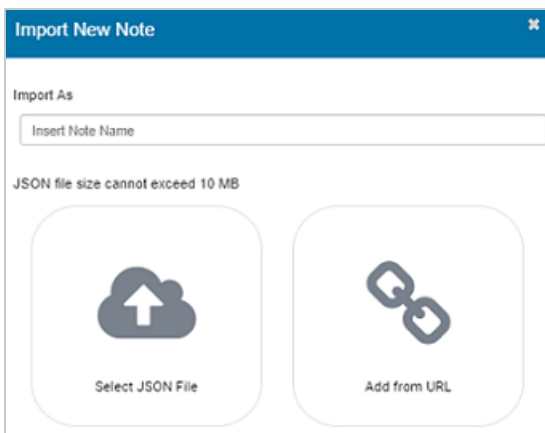
- **anzo.graph.password:** The password for the user in the `anzo.graph.user` field. On Docker deployments, specify **Passw0rd1**.
RHEL/Rocky Deployments: Use the Admin username and password that was created during the installation.
- **anzo.graph.port:** The gRPC port for Graph Lakehouse. The default value is **5700**. Do not change this value.
- **anzo.graph.trust.all:** Instructs Zeppelin to trust the Graph Lakehouse SSL certificates. Accept the default value of **true**.
- **anzo.graph.user:** The username to use to log in to Graph Lakehouse. On Docker deployments, specify **admin**.

4. When you finish adding the connection details, click **Save** at the bottom of the screen. Zeppelin displays a dialog box that asks if you want to restart the interpreter with the new settings. Click **OK** to configure the connection.
5. When the interpreter restart is complete, click the Zeppelin logo at the top of the screen to return to the index screen.

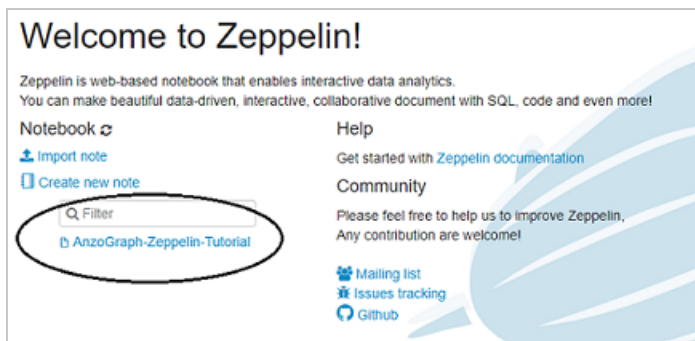
Downloading the Tutorial Notebook

If you deployed the Zeppelin image that does not contain sample notebooks and you want to import a notebook to get started with, follow the instructions in this section to download and import the Graph Lakehouse Tutorial Notebook.

1. Click the link below to download the Tutorial Notebook to your computer.
 [Download the Graph Lakehouse Tutorial Zeppelin Notebook](#)
2. Extract the downloaded notebook ZIP file on your computer. The ZIP file contains **AnzoGraph-Zeppelin-Tutorial.json**.
3. On the Zeppelin Welcome or the Algorithm Index screen if you have the sample notebooks, click **Import note**. Zeppelin displays the Import New Note screen.



4. On the import screen, click **Select JSON File**, and then select the AnzoGraph-Zeppelin-Tutorial.json file to import. Zeppelin imports the note and lists the new file under the filter text box on the home page or in the Notebook menu. For example:



Click the link to open the notebook. It might take some time to load all of the contents. To run a query in the file, click the run (▶) button for the paragraph. For more information about using Zeppelin, see the [Zeppelin Documentation](#).

Jupyter Notebook Integration

This section provides information about accessing Graph Lakehouse with your existing Jupyter Notebook or JupyterLab installation. If you do not have Jupyter Notebook or JupyterLab installed, follow the instructions in [Installing Jupyter](#) on the Jupyter website to install Jupyter Notebook and its prerequisites.

Accessing Graph Lakehouse from a Jupyter Notebook

1. Since Graph Lakehouse runs SPARQL queries, make sure that the Jupyter SPARQL kernel is installed. To install the kernel, run the following commands:

```
pip install sparqlkernel
jupyter sparqlkernel install --user
```

2. Connect to the Graph Lakehouse SPARQL endpoint by adding the following text to a cell in the notebook and then running the cell.

```
%endpoint http://hostname/sparql
%auth basic admin Passw0rd1
```

Where **hostname** is the IP address of the Graph Lakehouse instance. If Graph Lakehouse is running in a Docker container on the same server as Jupyter, you can run the following command to return the container IP address:

```
docker container inspect container_name | grep IP
```

For example:

```
docker container inspect anzograph | grep IP
```

Once the notebook is connected to the Graph Lakehouse endpoint, you can run SPARQL queries against Graph Lakehouse.

Access the SPARQL and RDF Endpoints

Graph Lakehouse supports the standard W3C SPARQL 1.1 Protocol and SPARQL 1.1 Graph Store HTTP Protocol for sending and receiving SPARQL requests between client applications and Graph Lakehouse. Since Graph Lakehouse adheres to RDF and SPARQL standards, developers do not need to learn a proprietary protocol or query language to incorporate Graph Lakehouse into their existing graph-based infrastructure. There are thousands of available [SPARQL client libraries](#) for querying SPARQL and graph store endpoints. In addition, tutorials, such as [Bob DuCharme's weblog](#), provide helpful information about SPARQL HTTP protocol.

Note

Graph Lakehouse also provides Bolt protocol support for execution of Cypher-based queries, either from Graph Lakehouse's Cypher-based CLI (AZGBOLT) or from other Cypher-based applications that use the Bolt protocol. For more information, see [Using the Cypher CLI \(AZGBOLT\)](#) and [Using Bolt Protocol](#).

This topic provides information about the Graph Lakehouse SPARQL and RDF Graph Store endpoints and describes the supported HTTP methods and parameters.

- [Endpoint Types and HTTP Methods](#)
- [Endpoint Base URLs](#)
- [Authentication and Request Parameters](#)
- [Example HTTP Requests](#)

Endpoint Types and HTTP Methods

The table below describes the SPARQL and RDF Graph Store HTTP endpoints. Both of the endpoints can be used to send requests through the Graph Lakehouse front end or directly to the database (back end).

Important

The front end requires that you use Basic Authentication to connect. The back end does not support authentication. When deciding whether to access an endpoint via the front end or back end, consider whether the client application supports authentication. See [Authentication and Request Parameters](#) below for more information.

Endpoint	Description
SPARQL	The SPARQL endpoint accepts HTTP GET and POST methods. Use GET to read data from the endpoint (SELECT, ASK, CONSTRUCT, DESCRIBE queries), and use POST to update data via the endpoint (INSERT, INSERT DATA, CREATE, DELETE, DELETE DATA, DROP queries). Update queries must use the POST method, but read-only queries can be submitted using GET or POST.
RDF Graph Store	The RDF graph store endpoint supports create, read, update, and delete (CRUD) operations and enables programmers to work with RDF graphs in a way that is similar to REST-style interfaces. The graph store endpoint supports GET, POST, UPDATE, and DELETE HTTP methods.

Tip

Typically, users configure client applications to connect to the SPARQL endpoint as it supports GET operations and update operations via POST. However, to use DELETE and UPDATE methods specifically, connect to the RDF graph store endpoint.

Endpoint Base URLs

This base URL that you use to connect to an Graph Lakehouse endpoint depends on whether you want to connect to the SPARQL endpoint or the RDF Graph Store endpoint. This section provides details about the base URLs for each endpoint:

- [SPARQL Endpoint Base URL](#)
- [RDF Graph Store Endpoint Base URL](#)

SPARQL Endpoint Base URL

To connect to the Graph Lakehouse SPARQL endpoint, use the following base URL:

```
protocol://hostname:port/sparql
```

The table below describes each of the base URL components:

Component	Description
protocol	The protocol to use for the connection: http for HTTP protocol or https for SSL protocol. Tip SPARQL HTTP or HTTPS protocol can be enabled and disabled via the enable_sparql_protocol and enable_ssl_protocol settings.
hostname	The DNS name or IP address of the Graph Lakehouse host server. For clusters, this is the name or IP address of the leader server.
port	The port number for the endpoint. The port that you specify depends on the protocol and whether the request is sent to the Graph Lakehouse front end or back end. Note The front end requires that you use Basic Authentication to connect. The back end does not support authentication. Consider whether the client application supports authentication when specifying the port. • Front end: If the front end ports were mapped to the default HTTP (80) and HTTPS (443) ports on the local host when the front end was

Component	Description
	deployed, do not specify a port. If the front end ports were mapped to non-default HTTP and HTTPS ports, specify the appropriate port based on the protocol. • Back end: The port is either the HTTP <code>sparql_protocol_port</code> or the HTTPS <code>ssl_protocol_port</code>. By default, the HTTP SPARQL protocol port is 7070, and the HTTPS SSL protocol port is 8256.
sparql	The path for the SPARQL endpoint.

For example, the following base URLs connect to the front end HTTP and HTTPS SPARQL endpoints. Because the ports for this deployment are mapped to the default HTTP and HTTPS ports on the local host, the port does not need to be specified in the URL:

```
http://10.100.10.20/sparql
```

```
https://10.100.10.20/sparql
```

The example URLs below connect to the back end HTTP and HTTPS SPARQL endpoints. In the examples, Graph Lakehouse is using the default SPARQL protocol and SSL protocol ports:

```
http://10.100.10.20:7070/sparql
```

```
https://10.100.10.20:8256/sparql
```

RDF Graph Store Endpoint Base URL

To connect to the Graph Lakehouse RDF graph store endpoint, use the following base URL:

```
protocol://hostname:port/endpoint_path
```

The table below describes each of the base URL components:

Component	Description
protocol	The protocol to use for the connection: http for HTTP protocol or https for SSL

Component	Description
	protocol. Tip SPARQL HTTP or HTTPS protocol can be enabled and disabled via the enable_sparql_protocol and enable_ssl_protocol settings.
hostname	The DNS name or IP address of the Graph Lakehouse host server. For clusters, this is the name or IP address of the leader server.
port	The port number for the endpoint. The port that you specify depends on the protocol and whether the request is sent to the Graph Lakehouse front end or back end. Note The front end requires that you use Basic Authentication to connect. The back end does not support authentication. Consider whether the client application supports authentication when specifying the port. • Front end: If the front end ports were mapped to the default HTTP (80) and HTTPS (443) ports on the local host when the front end was deployed, do not specify a port. If the front end ports were mapped to non-default HTTP and HTTPS ports, specify the appropriate port based on the protocol. • Back end: The port is either the HTTP sparql_protocol_port or the HTTPS ssl_protocol_port. By default, the HTTP SPARQL protocol port is 7070, and the HTTPS SSL protocol port is 8256.
endpoint_path	The path for the RDF graph store endpoint. The path that you specify depends on whether the request is being sent to the front end or back end: • Front end: The front end path is data.

Component	Description
	• Back end: The back end path is rdf-graph-store.

For example, the following base URLs connect to the front end HTTP and HTTPS RDF graph store endpoints. Because the ports for this deployment are mapped to the default HTTP and HTTPS ports on the local host, the port does not need to be specified in the URL:

```
http://10.100.10.20/data
```

```
https://10.100.10.20/data
```

The example URLs below connect to the back end HTTP and HTTPS RDF graph store endpoints. In the examples, Graph Lakehouse is using the default SPARQL protocol and SSL protocol ports:

```
http://10.100.10.20:7070/rdf-graph-store
```

```
https://10.100.10.20:8256/rdf-graph-store
```

Authentication and Request Parameters

The Graph Lakehouse front end SPARQL and RDF Graph Store endpoints require that you use Basic Authentication. The default credentials for the front end are Username:**admin** and Password:**Passw0rd1**. The back end does not require user authentication.

For information about the supported HTTP header and request parameters for the SPARQL endpoint, see the [SPARQL 1.1 Protocol](#) specification. For information about HTTP parameters for the RDF graph store endpoint, see the [SPARQL 1.1 Graph Store HTTP Protocol](#) specification.

Note

CONSTRUCT query results are always returned in RDF format. Accept, Content-Type, and format parameters are ignored.

Example HTTP Requests

This section includes sample HTTP requests using different languages:

- [cURL Example](#)
- [Python Example](#)

cURL Example

The following example, using the `curl` command, shows how you can specify a SPARQL query using Graph Lakehouse endpoint URLs. To query Graph Lakehouse using cURL, the queries must be [URL encoded](#). The command statement below shows the general syntax you could use to make cURL HTTP requests:

```
curl base_endpoint_url [ --insecure ] -u username:password [ -H "header_argument" ]
--data-urlencode "query=query_text" | query@file.rq
```

For example, the following cURL request is sent to the front end SPARQL endpoint and returns 10 triples from the sample Tickit graph. The statement returns results in CSV format:

```
curl https://10.10.10.100/sparql --insecure -u admin:Passw0rd1 -H "Accept: text/csv"
--data-urlencode "query=select ?s ?p ?o from <tickit> where {?s ?p ?o} order by ?p
limit 10"
```

```
s,p,o
person39003,birthday,1986-09-12
person33946,birthday,1988-12-24
person10199,birthday,1953-04-13
person41860,birthday,1976-06-25
person13789,birthday,1981-06-23
person30637,birthday,1978-11-26
person38857,birthday,1960-01-07
person24661,birthday,1992-11-21
person17029,birthday,1993-03-08
person43904,birthday,1962-01-06
```

The following cURL request sent to the back end SPARQL endpoint (without the **insecure** or **username** options) would return the same result.

```
curl http://10.10.10.100:7070/sparql -H "Accept: application/sparql-results+csv"
--data-urlencode "query=select ?s ?p ?o from <tickit> where {?s ?p ?o} order by ?p
limit 10"
```

This example uses cURL to run a query contained in a file. Since the statement does not include a header argument, Graph Lakehouse returns results in XML format:

```
curl https://10.10.10.100/sparql --insecure -u admin:Passw0rd1 --data-urlencode
query@/home/user/queries/sales_totals.rq
```

Python Example

You can also use the Graph Lakehouse SPARQL endpoint URL to execute SPARQL commands from programs such as Python or Javascript. Graph Lakehouse provides a Python library file, **azg3.py**, located in the `<install_path>/lib/py_modules` directory, to help you get started using Python to run SPARQL queries. You can click the following link to see the contents of the Python library file:

📄 [azg3.py](#)

The **azg3.py** module includes two functions for executing SPARQL queries:

```
azg3.run_query(sparql_endpoint, SPARQL_query_string [, format])
```

Runs the specified SPARQL query "*query_string*" at the SPARQL endpoint host location "*sparql_endpoint*". By default, this function returns the results as a Python dictionary map in the SPARQL1.1 results format (see <https://www.w3.org/TR/sparql11-results-json>). To output results using an alternate "raw string" format, you may also specify a different *format*, with possible format options **xml**, **json**, or **csv**.

```
azg3.create_dataframe(sparql_endpoint, SPARQL_query_string)
```

Runs the specified SPARQL query "*query_string*" on the SPARQL endpoint host "*sparql_endpoint*". This function returns results as a Pandas data frame object, for example:

```
df = azg3.create_dataframe("10.102.0.56:7070" , "SELECT... ")
```

To execute SPARQL queries from Python applications, you need to first have installed **Python 3**, and the **NumPy** and **Pandas** libraries on your client machine with network access to a deployed Graph Lakehouse environment. The easiest way to install these packages is to install the Anaconda distribution, which automatically installs Python 3 along with a number of scientific and data analysis

packages (including NumPy and Pandas) that may be helpful in performing analytic queries and calculations. (See <https://www.anaconda.com/distribution> for information about installing the Anaconda distribution.)

After creating the Python environment on a client machine:

1. Copy the `<install_path>/lib/py_modules` directory to your client machine:

```
$HOME/py_modules
```

2. Update your PYTHONPATH environment variable to include the Python module location:

```
export PYTHONPATH=$HOME/py_modules
```

Note

If you already have PYTHONPATH defined on your computer, copy the Python `az3.py` library file there instead.

For Python programs from which you want to run SPARQL queries, you will need to import the **azg3.py** library file. You can then use either of the two functions, **azg3.run_query** or **azg3.create_dataframe** to specify the SPARQL query to run as well as the location of the Graph Lakehouse server to target, for example, `"10.102.0.56:7070"`.

Tip

Along with the `azg3.py` file, the Graph Lakehouse distribution also includes a demo program, **azg3run.py** in the `<install_path>/lib/py_modules` directory, which allows you to execute SPARQL queries against an existing Graph Lakehouse database.

Access Data with OData Protocol

The Graph Lakehouse front end application includes a Data on Demand service that enables users to generate Open Data Protocol (OData)-based feeds that can be used to access data programmatically via a RESTful API or from third-party business intelligence tools such as TIBCO Spotfire, Tableau, and Microsoft Power BI.

OData facilitates the creation and consumption of queryable and interoperable RESTful APIs in a simple and standard manner. The protocol enables web clients to use simple HTTP messages to publish and edit resources that are identified using URLs and are defined in a data model. OData shares some similarities with JDBC and ODBC. Like ODBC, OData is not limited to relational databases. The Graph Lakehouse Data on Demand service follows the OData Version 4.0 specification, which defines the standard URL conventions, query options, and a metadata schema that describes the data model. The topics in this section provide information about creating and accessing Data on Demand endpoints.

In this section:

Create a Data on Demand Endpoint	519
Access a Data on Demand Endpoint	522
OData Reference	529

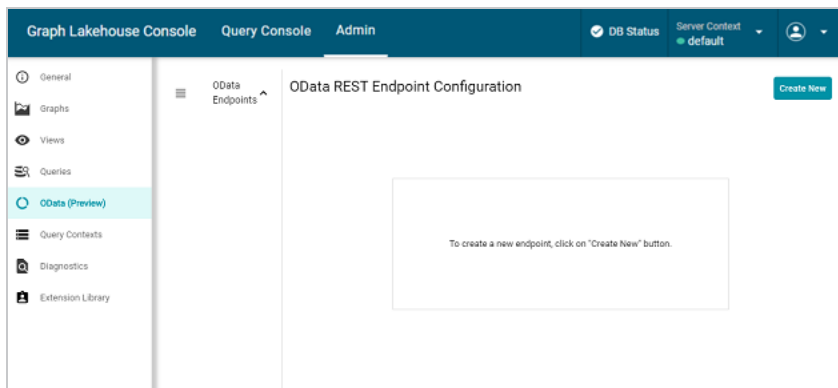
Create a Data on Demand Endpoint

This topic provides instructions for creating a Data on Demand endpoint for a graph or view. When you create an endpoint, the Data on Demand service generates a service root URL that you can use to connect to the endpoint programmatically or from applications.

Note

The Data on Demand service is included in the Graph Lakehouse front end, and endpoints must be created from the Query & Admin Console. For instructions on deploying the front end if it is not installed, see [Deploy the Frontend Container](#).

1. In the Query & Admin Console, click the **Admin** tab. Then click the **OData (Preview)** menu item. The OData REST Endpoint Configuration screen is displayed. For example:



2. Click the **Create New** button at the top of the screen. The Create New Data On Demand Configuration dialog box is displayed. For example:

The dialog box is titled 'Create New Data On Demand Configuration'. It contains the following fields: 'Endpoint Name *' (text input), 'EndPoint Description' (text input), 'Select Graphs or Views to expose as Endpoint*' (a list box with 'ticketit' and 'ghib' selected), and 'Select Graph or View containing Ontology' (a dropdown menu). At the bottom right, there are 'CANCEL' and 'SAVE' buttons.

3. Configure the Data on Demand endpoint by specifying the following options:

- **Endpoint Name:** Required field that specifies the name of this endpoint. This value is added to the service root URL that is generated for the endpoint.
- **Endpoint Description:** Optional field that lists a short description of the endpoint.
- **Select Graphs or Views to Expose:** Required field that specifies the graph or view to expose in the endpoint. You can press Ctrl and click to select multiple graphs or views.

Note

Labeled Property Graphs are not available for selection. OData is based on OWL and OWL does not support RDF*.

- **Select Graph or View Containing Ontology:** Optional field that specifies the graph or view that contains the ontology that you want to expose in the endpoint.

Note

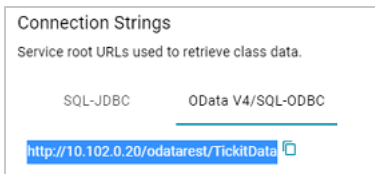
If an ontology does not exist, leave the field blank. The Data on Demand service automatically generates one when you save the configuration.

4. When you have finished configuring the endpoint, click **Save** to create the endpoint. The service generates an ontology, if needed, and then displays the endpoint details. For example:

The screenshot shows the 'OData REST Endpoint Configuration' window. On the left, there is a sidebar with 'OData Endpoints' and a list containing 'Ticket Data'. The main panel displays the configuration for 'Ticket Data'. It includes fields for 'Endpoint Name' (Ticket Data), 'Description' (Exposes the ticket graph), and 'Ontology Status' (Available). Under 'Connection Strings', it shows 'Service root URLs used to retrieve class data.' with two options: 'SQL-JDBC' and 'OData V4/SQL-ODBC'. The 'OData V4/SQL-ODBC' option is selected and underlined. Below this, a text box shows the URL 'http://10.102.0.20/odatarest/TicketData' with a download icon and a link to 'Download data access drivers'. At the bottom, there is a section for 'Class Specific Connection Strings' with a dropdown arrow.

The Data on Demand endpoint is ready for access via OData protocol.

5. To get the connection string for the entire graph or view that was exposed, use the **OData V4/SQL-ODBC** service URL that is listed under Connection Strings. For example:



6. If your application requires you to view one table at a time or you are only interested in viewing a specific class, click **Class Specific Connection Strings** to expand the field and view the connection strings for each class in the ontology. For example:

The screenshot shows a 'Class Specific Connection Strings' window. It has a search bar and a 'For URL Format' dropdown menu set to 'XML'. Below is a table with two columns: 'Class Title' and 'URL String'. The table lists five classes: person, date, event, listing, and sales, each with its corresponding OData URL. At the bottom, there is a pagination control showing 'Rows per page: 5' and '1-5 of 7'.

For convenience, the class-specific connection strings include the format parameter for specifying the output format. You can click the **For URL Format** drop-down list to set the output format to CSV, JSON, or XML.

For information about accessing OData endpoints, see [Access a Data on Demand Endpoint](#).

Access a Data on Demand Endpoint

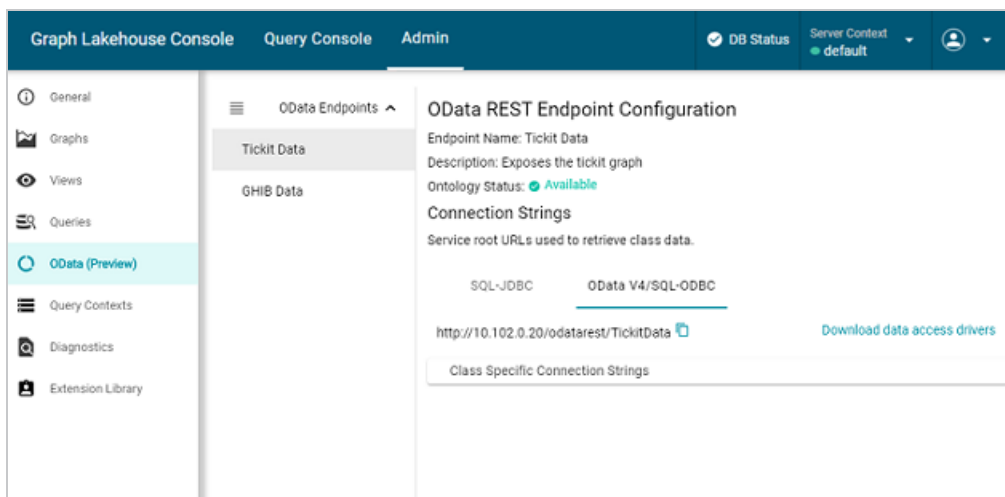
Since Graph Lakehouse's Data on Demand service conforms to the OData standard, any tool that supports the OData V4 REST API can access a Data on Demand endpoint to leverage data in Graph Lakehouse. This topic provides information about getting an OData connection string, endpoint authentication, and it includes examples of accessing endpoints from applications and programs. For instructions on creating endpoints, see [Create a Data on Demand Endpoint](#).

- [Retrieving an OData Connection String](#)
- [Using Authentication](#)
- [Accessing an Endpoint from an Application](#)
- [Accessing an Endpoint Programmatically](#)

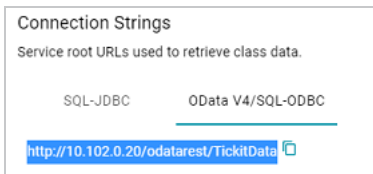
Retrieving an OData Connection String

Follow the instructions below to retrieve the connection string to use for a Data on Demand endpoint.

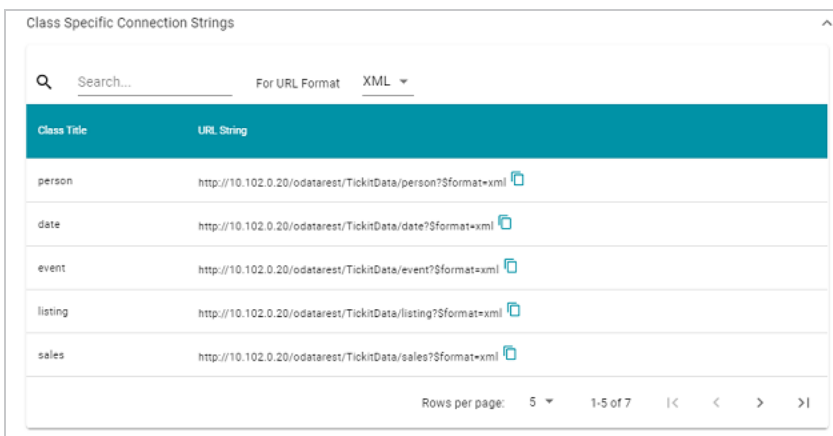
1. In the Graph Lakehouse user interface, click the **Admin** tab. Then click the **OData (Preview)** menu item. The OData REST Endpoint Configuration screen is displayed, which lists the existing endpoints. For example:



2. If necessary, click the name of the endpoint for which you want to retrieve the connection string.
3. To get the connection string for the entire graph or view that was exposed, use the **OData V4/SQL-ODBC** service URL that is listed under Connection Strings. For example:



4. If your application requires you to view one table at a time or you are only interested in viewing a specific class, click **Class Specific Connection Strings** to expand the field and view the connection strings for each class in the ontology. For example:



For convenience, the class-specific connection strings include the format parameter for specifying the output format. You can click the **For URL Format** drop-down list to set the output format to CSV, JSON, or XML.

Using Authentication

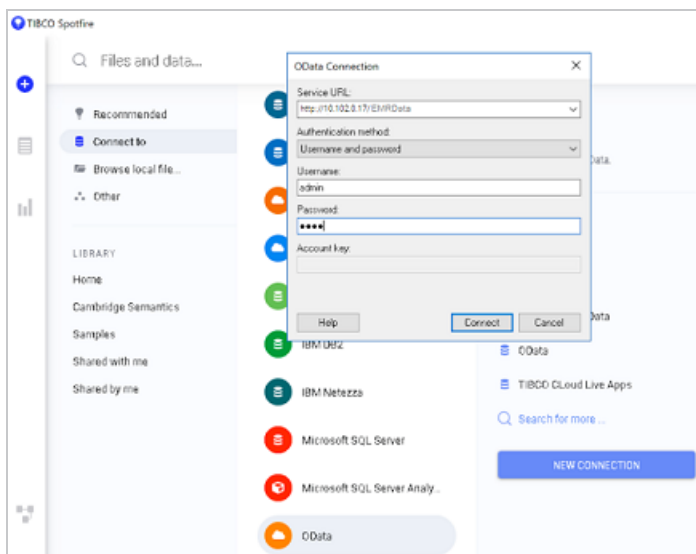
Connections to Data on Demand endpoints must be authenticated using Basic Authentication. For Docker deployments, the default credentials are:

- Username: **admin**
- Password: **Passw0rd1**

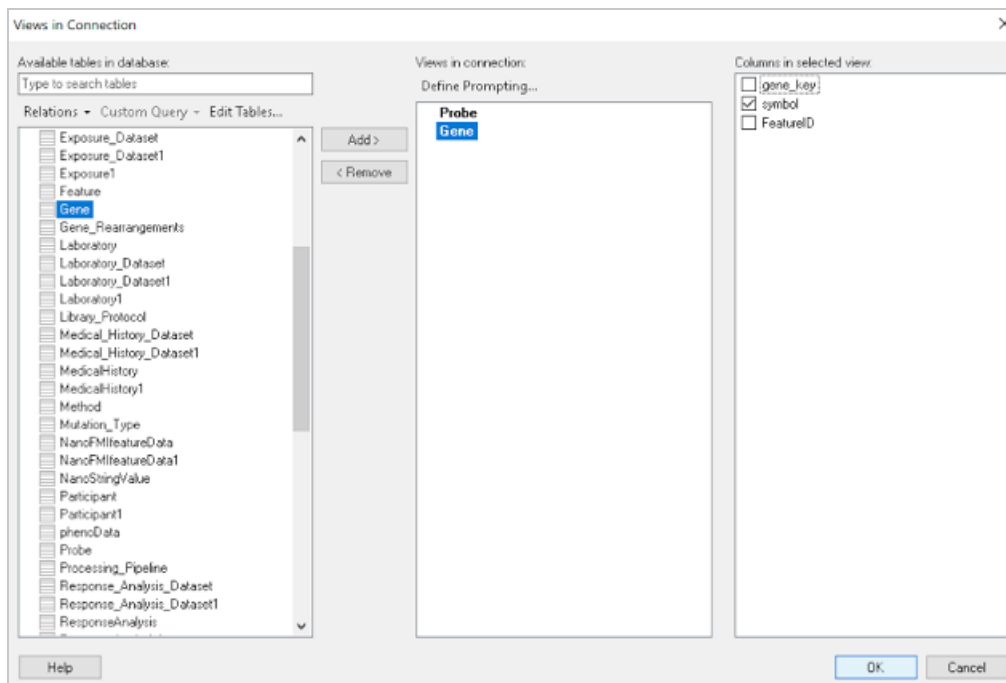
Accessing an Endpoint from an Application

This section provides guidance on accessing a Data on Demand endpoint from an application that supports the OData REST API. It includes an example that configures an OData connection in TIBCO Spotfire. The example steps can also be applied to OData connections in other similar business intelligence tools.

The first step is to connect to the OData endpoint using the Spotfire Data sources user interface. When setting up the OData connection, the Service URL is the OData/ODBC URL from the Data on Demand endpoint configuration details in the user interface. The OData connection uses the Graph Lakehouse user interface credentials for authentication.



Once the connection is established, Sportfire prompts the user to select the classes and properties to work with. In this example, the **FeatureID** property from the **Probe** class and the **symbol** property from the **Gene** class are selected:



Once the properties are chosen, the data is loaded in Spotfire and can be used to inform existing analytics and data visualizations or create new ones.

Accessing an Endpoint Programmatically

This topic provides guidance on accessing Data on Demand endpoints programmatically by showing some example implementations using R and Python.

- [Accessing an Endpoint with R \(Through RStudio\)](#)
- [Accessing an Endpoint with Python \(Through a Linux Terminal\)](#)

Accessing an Endpoint with R (Through RStudio)

The following example shows how to connect to an OData endpoint from RStudio. The example uses the R programming language to access a Data on Demand endpoint and pull in data via a standard dataframe. New or existing R scripts can then be used with the data.

The first step in accessing data from RStudio is to prepare the R script that will construct the target URL and retrieve the resulting information via HTTP. The example script below accesses a pre-configured "SampleData" endpoint. The script has sections for filtering the results as well as expanding the selection to include information from multiple classes:

```

require("httr")
require("jsonlite")
require("rstudioapi")

user  <- rstudioapi::showPrompt("Username", "Enter AnzoGraph username", "admin")
pw    <- rstudioapi::askForPassword(paste("Enter password for",user,sep=" "))

## Data on Demand endpoint
odata  <- "http://10.100.0.10/odatarest/SampleData"

## Start from Probe class
startClass  <- "Probe?"

## Filter results for Homo sapiens species
filterKw    <- "$filter="
filterVal   <- "Species eq 'Hs'"
urlify      <- URLencode(filterVal)
filterStr   <- paste(filterKw,urlify,sep="")

## Select properties of interest (FeatureID) from base class
selectKw    <- "&$select="
selectVal   <- "FeatureID"
selectStr   <- paste(selectKw,selectVal,sep="")

## Select properties of interest (symbol) from Gene class
## via corresponds_to property on base Probe class
expandKw    <- "&$expand="
expandClass <- "corresponds_to"
expandProps <- "symbol"
expSelStr   <- "$select="
expandStr    <- paste(expandKw,expandClass,"(",expSelStr,expandProps,")",sep="")

## Specify format
format  <- "&$format=json"

## Generate OData URL using fragments above
url <- paste(odata,startClass,filterStr,selectStr,expandStr,format,sep="")

## Access OData endpoint
resultRaw <- GET(url, (authenticate(user,pw, type = "basic")))
resultTxt <- content(resultRaw, "text")
resultJson <- fromJSON(resultTxt, flatten = TRUE)

```

```
print(url)

## Read results into dataframe
resultDataFrame <- as.data.frame(resultJson)
View(resultDataFrame)
```

Executing the above R script from RStudio results in a dataframe that represents columns from the **Probe** and **Gene** classes.

Accessing an Endpoint with Python (Through a Linux Terminal)

Many users have existing Python scripts to use with data in Graph Lakehouse or a familiarity with Python that would make exploring, retrieving, and leveraging the data easier. The following example shows how to connect to an OData endpoint by executing a Python script from a Linux terminal.

The first step in accessing data using Python is to prepare the Python script that will construct the target URL and retrieve the resulting information via HTTP. The example script below accesses a pre-configured "SampleData" endpoint. The script has sections for filtering the results as well as expanding the selection to include information from multiple classes (the same filter and class properties that were used in the R example above).

```
import requests
import getpass
from urllib.parse import urlparse

un = getpass.getpass(prompt='Username: ')
pw = getpass.getpass(prompt='Password: ')

## OData endpoint
odata = 'https://10.100.0.10/odatarest/SampleData/'
# data on demand url

## Start from Lease class
startClass = "Probe?"

## Filter results
filterKw = "$filter="
filterVal = "Species eq 'Hs'"
urlify = urlparse(filterVal)
filterStr = filterKw + urlify.geturl()
```

```

## Select properties of interest (start date, missed payments, lease status) from base
class
selectKw  = "&$select="
selectVal = "FeatureID"
selectStr = selectKw + selectVal

## Select properties of interest (name, social security number, credit score) from
Individual class
expandKw  = "&$expand="
expandClass = "corresponds_to"
expandProps = "symbol"
expSelStr  = "$select="
expandStr = expandKw + expandClass + "(" + expSelStr + expandProps + ")"

## Specify format
format = "&$format=text/csv"

## Generate OData URL using fragments above
url = odata + startClass + filterStr + selectStr + expandStr + format

## Access OData endpoint
r = requests.get(url, auth=(un, pw), verify=False)

print("URL")
print(url)
print("CONTENT")
print(r.content.decode('unicode_escape'))
print(type(r))
print(type(r.content))

```

In this example, the output is returned in CSV format (rather than JSON, as in the R example).

OData Reference

The Data on Demand service follows the [OData Version 4.0 specification](#), which defines the standard URL conventions and query options. This topic provides a quick reference for learning OData basics and viewing the supported string operators and output formats. It also provides some example queries.

- [OData URL Conventions](#)
- [Supported Query Operators](#)
- [Example OData Requests](#)

OData URL Conventions

An OData service URL has three main parts:

1. The **Service Root URL** that AnzoGraph provides. The service root URL is the metadata that describes all of the available feeds (tables). For information about viewing the service root URL for an endpoint, see [Retrieving an OData Connection String](#).
2. The optional **Resource Path** that narrows the scope of the available data to the table (class) level, property level, or the schema.
3. The **Query Options** for analyzing the data.

For example, the following OData URL shows the service root from the Data on Demand service, a resource path that narrows the scope of the data to the Employees table (class), and query options that filter the result set to show data for the NA region only:

```
http://10.100.0.10/odatarest/Northwind/Employees?$filter=contains(Region, 'NA')
```

\ _____ /	\ _____ /	\ _____ /
Service Root URL	Resource Path	Query Options

Note

OData requests need to be URL-encoded. Typically you can configure programs to encode requests automatically. And browsers encode URLs that are pasted into the address bar.

Supported Query Operators

OData query options are used to dynamically query data via the endpoint and control the amount and order of the data returned. The Data on Demand service supports the following OData query operators. See [Example OData Requests](#) below for example queries that employ the operators.

Operator	Description
\$count	Used to count the number of matching resources in the result set.
\$expand	Used to retrieve related data and include it in the results. When you query data via OData, the default response does not include related entities. The \$expand option provides flexibility for exploring data across the data model. It allows the related information to be embedded in the response.
\$filter	Used to filter a result set. The expression specified with \$filter is evaluated for each resource identified by the resource path, and only items where the expression evaluates to true are included in the response.
\$format	Used to specify the output format for the results. The supported formats are text/CSV, JSON, and XML. For example: \$format=json
\$metadata	Used to return the schema, entity set, and property metadata.
\$orderby	Used to return results in ascending (asc) or descending (desc) order. If asc or desc is not specified, solutions are returned in ascending order.
\$select	Used to specify the subset of properties to include in the result set.
\$skip	Used to specify the number of solutions to exclude in the results. The \$top and \$skip OData query options are similar to the LIMIT and OFFSET clauses in SPARQL queries.
\$top	Used to limit the number of solutions that are returned.

Example OData Requests

This section demonstrates the use of OData query operators by providing examples of common types of OData requests.

The examples below are run against a sample graph, called **LeagueGM**, that contains data about the teams and players in a small local baseball league. The Data on Demand endpoint is named **LeagueData**. The following service root URL was created by the Data on Demand service:

```
http://10.100.0.10/odatarest/LeagueGM
```

Note

For readability, the examples below abbreviate "http://10.100.0.10/odatarest" to **dataondemand**. In addition, the examples are not URL-encoded.

The data has Leagues, Teams, Players, and Positions classes (or entities in OData). The image below shows a graph view of the data model:



To view details about the properties and values for each class, you can click a link below to view the data for that class. The data is in JSON format.

[!\[\]\(5361750c22c4e047a52f4eac1ec2d4cc_img.jpg\) Leagues](#) [!\[\]\(f276343e5e0d2402c20fdc9e8443c0dd_img.jpg\) Teams](#) [!\[\]\(f63d0a0c6c21d1cd8465081c8a0d79d6_img.jpg\) Players](#) [!\[\]\(c7aa0a73772f58306e4c9858ef30418b_img.jpg\) Positions](#)

- [Counting an Entity](#)
- [Counting a Property of an Entity](#)

- [Filtering Data via Text Search](#)
- [Selecting Properties and Ordering Results](#)
- [Expanding the Results to Include Related Entities](#)

Counting an Entity

The request below returns the number of teams in the graph. Adding the resource path **Teams** to the request narrows the scope to the Teams entity (or class).

```
dataondemand/LeagueGM/Teams/$count
```

Result

```
4
```

This request returns the number of players:

```
dataondemand/LeagueGM/Players/$count
```

Result

```
12
```

Counting a Property of an Entity

The request below counts the number of players on the Al Thomas team. The request uses the `team_key` to identify the team and the `TeamToPlayer` to identify each player.

```
dataondemand/LeagueGM/Teams('aHR0cDovL2NzaS5jb20vVGh0bXMvMQ')/TeamToPlayer/$count
```

Result

```
3
```

This request counts the number of positions played by James Smith:

```
dataondemand/LeagueGM/Players  
('aHR0cDovL2NzaS5jb20vUGxheWVycy8y')/PlayerToPosition/$count
```

Result

```
2
```

Filtering Data via Text Search

The request below filters the results to show data for the TeamName that equals "Black Sox." The request also returns results in JSON format:

```
dataondemand/LeagueGM/Teams?$filter=TeamName eq 'Black Sox'&$format=json
```

Result

```
{
  "@odata.context": "https://10.100.0.10/odatarest/LeagueGM/$metadata#Teams",
  "value": [
    {
      "teams_key": "aHR0cDovL2NzaS5jb20vVGVhbXMvMg",
      "TeamId": 2,
      "teamtleague_key": [
        "aHR0cDovL2NzaS5jb20vTGVhZ3Vlcy8x"
      ],
      "TeamName": "Black Sox",
      "teamtplayer_key": [
        "aHR0cDovL2NzaS5jb20vUGxheWVycy80",
        "aHR0cDovL2NzaS5jb20vUGxheWVycy81",
        "aHR0cDovL2NzaS5jb20vUGxheWVycy82"
      ]
    }
  ]
}
```

This request filters the data to find the players whose name contains "Ted."

```
dataondemand/LeagueGM/Players?$filter=contains(PlayerName,'Ted')
```

The request can also use "startswith" in place of contains to filter specifically for player names that start with "Ted."

```
dataondemand/LeagueGM/Players?$filter=startswith(PlayerName,'Ted')
```

Result

```
{
  "@odata.context": "https://10.100.0.10/odatarest/LeagueGM/$metadata#Players",
  "value": [
    {
```

```

    "players_key": "aHR0cDovL2NzaS5jb20vUGxheWVycy8xMA",
    "playertoposition_key": [
      "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzM",
      "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzI"
    ],
    "PlayerId": 10,
    "playertoteam_key": [
      "aHR0cDovL2NzaS5jb20vVG9hbmVhbmVNA"
    ],
    "PlayerName": "Ted James",
    "DefensiveRating": 92.55
  },
  {
    "players_key": "aHR0cDovL2NzaS5jb20vUGxheWVycy84",
    "playertoposition_key": [
      "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzI",
      "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzEw"
    ],
    "PlayerId": 8,
    "playertoteam_key": [
      "aHR0cDovL2NzaS5jb20vVG9hbmVhbmVMw"
    ],
    "PlayerName": "Ted Sale",
    "DefensiveRating": 77.33
  }
]
}

```

Selecting Properties and Ordering Results

The request below selects player names and their defensive ratings. The results are ordered by defensive rating in descending order so that the player with the highest defensive rating is listed first. The request also formats the results in text/csv.

```

dataondemand/LeagueGM/Players?$select=PlayerName,DefensiveRating&$orderby=DefensiveRating desc&$format=text/csv

```

Result

```

PlayerName,DefensiveRating
James Smith,98.33
Alex Granderson,98.22
Matt Butler,95.66

```

Tim Hooper, 93.43
Steve Jones, 93.28
Ted James, 92.55
Fred Wynn, 88.68
Jared Bonds, 86.34
Billy Roper, 83.44
Mike Magazine, 78.33
Ted Sale, 77.33
Chris Underwood, 66.22

Expanding the Results to Include Related Entities

The request below uses the \$expand operator to retrieve data from the Players entity and include the related Positions data for each player. For this example, the request limits the number of results returned to 2 players by adding \$top=2:

dataondemand/LeagueGM/Players?\$expand=PlayerToPosition&\$top=2

Result

```
{
  "@odata.context": "https://10.100.0.10/odataorest/LeagueGM/$metadata#Players",
  "value": [
    {
      "players_key": "aHR0cDovL2NzaS5jb20vUGxheWVycy8x",
      "playertoposition_key": [
        "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzg"
      ],
      "PlayerId": 1,
      "playertoteam_key": [
        "aHR0cDovL2NzaS5jb20vVGhvbXMvMQ"
      ],
      "PlayerName": "Steve Jones",
      "DefensiveRating": 93.28,
      "PlayerToPosition": [
        {
          "positions_key": "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzg",
          "PositionId": 8,
          "ShortName": "CF",
          "positiontoplayer_key": [
            "aHR0cDovL2NzaS5jb20vUGxheWVycy8xMg",
            "aHR0cDovL2NzaS5jb20vUGxheWVycy8x"
          ],

```

```

        "Description": "Centerfield"
    }
]
},
{
    "players_key": "aHR0cDovL2NzaS5jb20vUGxheWVycy8xMA",
    "playertoposition_key": [
        "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzI",
        "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzM"
    ],
    "PlayerId": 10,
    "playertoteam_key": [
        "aHR0cDovL2NzaS5jb20vVGVhbXMvNA"
    ],
    "PlayerName": "Ted James",
    "DefensiveRating": 92.55,
    "PlayerToPosition": [
        {
            "positions_key": "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzI",
            "PositionId": 2,
            "ShortName": "C",
            "positiontoplayer_key": [
                "aHR0cDovL2NzaS5jb20vUGxheWVycy84",
                "aHR0cDovL2NzaS5jb20vUGxheWVycy8xMA"
            ],
            "Description": "Catcher"
        },
        {
            "positions_key": "aHR0cDovL2NzaS5jb20vUG9zaXRpb25zLzM",
            "PositionId": 3,
            "ShortName": "1B",
            "positiontoplayer_key": [
                "aHR0cDovL2NzaS5jb20vUGxheWVycy83",
                "aHR0cDovL2NzaS5jb20vUGxheWVycy8xMA"
            ],
            "Description": "First Base"
        }
    ]
}
]
}
]
}

```


Create and Save Views

Graph Lakehouse includes support for creating views. Views enable users to simplify logic by hiding the underlying complexity of the data or SPARQL operations, combine data from one or more graphs or other views, or mask sensitive information from some users. You can create virtual views, where Graph Lakehouse stores only the view definition, or materialized views, where Graph Lakehouse stores a copy of the data that the view creates as well as the view definition. This topic provides information about creating and using views.

Tip

Graph Lakehouse also enables you to create named query definitions. See [Save Queries for Reuse](#) for more information.

There are two ways to create a named view, depending on whether you want to create a view definition that is saved in the triplestore and can be referenced in various queries or whether you want to write a view inline to be referenced once in the query that immediately follows the inline view definition. This topic provides instructions for creating each type of named view:

- [Create and Save a View for Reuse](#): Follow these instructions to create and save a named view for future use.
- [Create a View Inline for One-Time Use](#): Follow these instructions to create a named view inline for single use.

Create and Save a View for Reuse

Use the following syntax to create a view and save the view definition for future use:

```
CREATE [ OR REPLACE ] [ MATERIALIZED ] VIEW <view_uri> AS
CONSTRUCT {
    query
}
```

Include the OR REPLACE keywords when you want to replace a previously defined view with the same name. Include the MATERIALIZED keyword if you want Graph Lakehouse to store a copy of the data that the view constructs. If you exclude MATERIALIZED, Graph Lakehouse stores only the view definition.

To reference a view in subsequent queries, use `view_uri` as a graph URI in FROM clauses or GRAPH patterns. For example:

```
SELECT *
FROM <view_uri>
WHERE { ?s ?p ?o . }
```

Or

```
SELECT *
FROM <ticket>
FROM NAMED <view_uri>
WHERE {
    ?person <birthday> ?bday .
    GRAPH <view_uri> { ?person <age> ?age. }
}
```

Create a View Inline for One-Time Use

If you want to create a named view on-the-fly to reference in a query that you are writing, you can include a `WITH` clause to define a named view at the beginning of that query.

Note

Graph Lakehouse does not save the view definition for named views that are defined in a `WITH` clause. The named view can only be referenced in the query that immediately follows the `WITH` clause; it is not available to use in subsequent queries. To create a named view whose definition is persisted and can be referenced in future queries, use the `CREATE OR REPLACE` syntax.

WITH Syntax

```
WITH ( VIEW <view_uri> AS construct_query )
[ ( ... ) ]
```

Where `construct_query` is the query that constructs the view that you want to name and reference as a graph URI in the main query. You can define multiple named views in one `WITH` clause. For example, the `WITH` clause below defines a view named **friends**. The friends view is listed as a graph in the `FROM` clause in the main query:

```
WITH
( VIEW <friends> AS
CONSTRUCT { ?s <friend> ?friend }
WHERE { GRAPH <tickit> {
  ?s <friend> ?friend .
  filter (?s = <person1> || ?s = <person2>)
}
}
)
SELECT *
FROM <friends>
WHERE { ?s ?p ?o }
ORDER BY ?s
```

```
s      | p      | o
-----+-----+-----
```

```
person1 | friend | person20018
person1 | friend | person11678
person1 | friend | person12081
person1 | friend | person12316
person1 | friend | person11549
person1 | friend | person13826
person1 | friend | person26733
person1 | friend | person3005
person1 | friend | person27710
person1 | friend | person29554
person1 | friend | person14472
...
73 rows
```

Examples

The example queries in this section run against the Graph Lakehouse sample Tickit data set, which captures sales activity for a fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts. You can load and explore this data set. For more information, see [Working with SPARQL and the Tickit Data](#).

In the sample Tickit data set, the sales1 data includes values for the following properties or predicates:

```
SELECT ?p
FROM <tickit>
WHERE {
    <sales1> ?p ?o .
}
```

```
p
-----
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
dateid
sellerid
eventid
commission
saletime
listid
pricepaid
qtysold
buyerid
10 rows
```

A sales manager might want to create a view so that the sales team can review ticket sales data without viewing the commission paid to their team members. This query creates a view that suppresses the commission values for sales1:

```
CREATE VIEW <no-commission> AS
CONSTRUCT {?s ?p ?o}
WHERE { GRAPH <tickit> {
    ?s ?p ?o .
    FILTER(?p != <commission>)
    FILTER(?s = <sales1>)
```

```
}
}
```

Querying the sales1 data using the new view shows the following results:

```
SELECT ?p ?o
FROM <no-commission>
WHERE { ?s ?p ?o . }
```

p	o
-----	-----
buyerid	person21191
listid	listing1
pricepaid	728.000000
sellerid	person36861
eventid	event7872
qtysold	4
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	sales
saletime	2008-02-18T02:36:48Z
dateid	date1875
9 rows	

The example below creates a materialized view called "ages." The view constructs a new age predicate and calculates the approximate age value for each person in the sample Tickit data set.

```
CREATE MATERIALIZED VIEW <ages> AS
CONSTRUCT { ?person <age> ?age . }
WHERE { GRAPH <tickit> {
    SELECT ?person ((YEAR(?date))-(YEAR(xsd:dateTime(?birthdate)))) AS ?age
    WHERE {
        ?person <birthday> ?birthdate .
        BIND(xsd:dateTime(NOW()) AS ?date)
    }
}
}
```

Running the following query against the view, shows the approximate age of each person.

```
SELECT *
FROM <ages>
WHERE { ?s ?p ?o . }
LIMIT 10
```

```
s      | p  | o
-----+-----+-----
person40149 | age | 39
person30658 | age | 23
person6893  | age | 30
person12131 | age | 22
person33027 | age | 69
person24690 | age | 55
person9306  | age | 76
person4808  | age | 54
person45368 | age | 25
person34994 | age | 59
10 rows
```

Save Queries for Reuse

Similar to the [Create and Save Views](#) functionality, Graph Lakehouse enables you to create query definitions that you can reference as subqueries in other queries. Naming queries for later use enables you to simplify complex queries and quickly add commonly used subqueries to other queries. Using named queries can also increase query performance since Graph Lakehouse can identify and execute repetitive patterns once and then reuse the results.

There are two ways to create a named query, depending on whether you want to create a query definition that is saved in the triplestore and can be used as a subquery in various queries or whether you want to write a query inline to be used once in the query that immediately follows the inline named query definition. This topic provides instructions for creating each type of named query:

- [Create and Save a Query for Reuse](#): Follow these instructions to create and save a named query for future use.
- [Create a Query Inline for One-Time Use](#): Follow these instructions to create a named query inline for single use.

Create and Save a Query for Reuse

Use the following syntax to create a query and save the query definition so that you can run the query as a subquery in subsequent queries:

```
CREATE [ OR REPLACE ] QUERY <query_URI> AS
    query_text
```

Include the OR REPLACE keywords when you want to replace a previously defined query with the same name. For example, the query below creates a named query called **total_profit**.

```
PREFIX tickit: <http://anzograph.com/tickit/>
CREATE OR REPLACE QUERY <total_profit> AS
SELECT ?event (sum(?qty) as ?tickets) (sum(?comm) as ?commission_paid)
    (sum(?price) as ?total_paid)
FROM <http://anzograph.com/tickit>
WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?sales tickit:commission ?comm .
    ?sales tickit:pricepaid ?price .
}
GROUP BY ?event
ORDER BY ?event
```

To reference a predefined query as a subquery, use the following syntax in the WHERE clause:

```
{ QUERY <query_URI> }
```

For example, the query below includes the predefined query, **total_profit**, as a subquery:

```
SELECT ?event ?tickets ((?total_paid - ?commission_paid) as ?profit)
FROM <http://anzograph.com/tickit>
WHERE { QUERY <total_profit> }
ORDER BY desc(?profit)
LIMIT 10
```

event	tickets	profit
Mamma Mia!	3658	965135.900000
Spring Awakening	3025	826926.750000
The Country Girl	2871	773978.550000

Macbeth		2733		733193.000000
Jersey Boys		2781		690095.450000
Legally Blonde		2272		683895.550000
Chicago		2535		672344.050000
Spamalot		2199		607160.950000
Hedda Gabler		1891		561865.300000
Thurgood		1894		543895.450000

10 rows

Note

When you include a FROM clause in a named query, Graph Lakehouse always applies that FROM list to the query. If the named query becomes a subquery in another query, the subquery does not inherit the FROM clause from the main query. If the named query does not include a FROM clause, Graph Lakehouse applies the FROM clause from the main query.

Create a Query Inline for One-Time Use

If you want to create a named query on-the-fly to use in a query that you are writing, you can include a **WITH** clause to define a named query at the beginning of that query.

Note

Graph Lakehouse does not save the query definition for named queries that are defined in a **WITH** clause. The named query can only be referenced in the query that immediately follows the **WITH** clause; it is not available to use in subsequent queries. To create a named query whose definition is persisted and can be referenced in future queries, use the **CREATE OR REPLACE** syntax..

WITH Syntax

```
WITH ( QUERY <query_name> AS select_query )  
[ ( ... ) ]
```

Where `select_query` is the query that you want to name and reference in the **WHERE** clause of the main query. You can define multiple named queries in one **WITH** clause. For example, the **WITH** clause below defines a query named **profit**. The profit query is then referenced as a subquery in the main query:

```
PREFIX tickit: <http://anzograph.com/tickit/>  
WITH  
( QUERY <profit> AS  
  SELECT ?event (sum(?qty) as ?tickets) (sum(?comm) as ?commission_paid)  
          (sum(?price) as ?total_paid)  
FROM <http://anzograph.com/tickit>  
  WHERE {  
    ?sales tickit:qtysold ?qty .  
    ?sales tickit:eventid ?eventid .  
    ?eventid tickit:eventname ?event .  
    ?sales tickit:commission ?comm .  
    ?sales tickit:pricepaid ?price .  
  }  
  GROUP BY ?event  
)  
SELECT ?event ?tickets ((?total_paid - ?commission_paid) as ?profit)
```

```
FROM <ticket>  
WHERE { QUERY <profit> }  
ORDER BY desc(?profit)  
LIMIT 10
```

Examples

The example queries in this section run against the Graph Lakehouse sample Tickit data set, which captures sales activity for a fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts. You can load and explore this data set. For more information, see [Working with SPARQL and the Tickit Data](#).

The example below creates a named query and uses it to query the sample Tickit data set to identify possible ticket scalpers by calculating the average price per ticket for events and then finding cases where tickets are listed for a higher price.

```
PREFIX tickit: <http://anzograph.com/tickit/>
WITH ( QUERY <avg_price> AS
  SELECT ?eventname (avg(?priceperticket) as ?avg_price)
  WHERE {
    ?listing tickit:eventid ?eventid .
    ?eventid tickit:eventname ?eventname .
    ?listing tickit:priceperticket ?priceperticket .
  }
  GROUP BY ?eventname
)
SELECT ?sellername ?avg_price ?priceperticket ?eventname ?listtime
FROM <http://anzograph.com/tickit>
WHERE {
  { QUERY <avg_price> }
  ?listing tickit:listtime ?listtime .
  ?listing tickit:priceperticket ?priceperticket .
  ?listing tickit:sellerid ?seller .
  ?seller tickit:firstname ?firstname .
  ?seller tickit:lastname ?lastname .
  BIND(CONCAT(?firstname, " ", ?lastname) AS ?sellername)
  FILTER (?priceperticket > ?avg_price)
}
ORDER BY ?avg_price ?listtime ?sellername ?eventname
LIMIT 10
```

sellername	avg_price	priceperticket	eventname	listtime
Garrett Rasmussen	249.181818	277.000000	White Christmas	2008-01-01T01:03:16Z
Ivan Trevino	249.181818	415.000000	White Christmas	2008-01-

```

01T01:03:17Z
Liberty Hopkins      | 249.181818 |      2120.000000 | White Christmas | 2008-01-
01T01:03:53Z
Jenette Norton       | 249.181818 |     1031.000000 | White Christmas | 2008-01-
01T01:04:15Z
Tana Mcguire         | 249.181818 |      455.000000 | White Christmas | 2008-01-
01T01:07:02Z
Lee Prince           | 249.181818 |      377.000000 | White Christmas | 2008-01-
01T01:14:34Z
Aileen Nicholson     | 249.181818 |      413.000000 | White Christmas | 2008-01-
01T01:14:40Z
Wylie Kemp           | 249.181818 |      372.000000 | White Christmas | 2008-01-
01T01:14:48Z
Allistair Yang       | 249.181818 |      491.000000 | White Christmas | 2008-01-
01T01:15:54Z
Quinn Porter         | 249.181818 |      977.000000 | White Christmas | 2008-01-
01T01:17:19Z
10 rows

```

The example below uses a **WITH** clause to define the subquery, locations, which queries the example Tickit data to return a list of the locations for events that took place in February:

```

PREFIX tickit: <http://anzograph.com/tickit/>
WITH ( QUERY <locations> AS
  SELECT ?name ?where
  WHERE {
    ?e tickit:venueid ?v .
    ?v tickit:venueid ?where .
    ?e tickit:dateid ?d .
    ?d tickit:month ?when .
    ?e tickit:eventname ?name .
    filter (?when = "FEB")
  }
)
SELECT *
FROM <http://anzograph.com/tickit>
WHERE {
  {QUERY <locations>}
}
ORDER BY ?where ?name
LIMIT 100

```

name	where
G. Love and Special Sauce	ARCO Arena
Oasis	ARCO Arena
Rush	ARCO Arena
Smash Mouth	ARCO Arena
Steve Miller Band	ARCO Arena
The Guess Who	ARCO Arena
Tokio Hotel	ARCO Arena
Projekt Revolution	AT&T Center
Chromeo	AT&T Park
Citizen Cope	AT&T Park
Counting Crows and Maroon 5	AT&T Park
Extreme	AT&T Park
Oliver Dragojevic	AT&T Park
Nashville Star	Air Canada Centre
Stone Temple Pilots	Air Canada Centre
Taylor Swift	Air Canada Centre
Wallflowers	Air Canada Centre
ZZ Top	Air Canada Centre
Dirty Dancing	Al Hirschfeld Theatre
...	
100 rows	

SPARQL Query Language Reference

Graph Lakehouse implements the standard SPARQL forms and functions described in the [W3C SPARQL 1.1 Query Language](#) specification. In addition to supporting the standard functions, Graph Lakehouse also provides several SQL-like and Microsoft Excel-like functions as well as support for more advanced operations like window aggregates, advanced grouping sets, and graph algorithms. In addition to the built-in functions, Graph Lakehouse includes extension libraries that offer advanced Data Science, Geospatial, Matrix, Apache Arrow, and other utility packages. The topics in this section describe the supported built-in functions and extension libraries.

Tip

Most examples included in this section run against the Graph Lakehouse sample Tickit data set, which captures sales activity for a fictional Tickit website where people buy and sell tickets for sporting events, shows, and concerts. For more information on loading and analyzing this data set, see [Working with SPARQL and the Tickit Data](#).

In this section:

Built-in Functions	553
Extension Libraries	704

Built-in Functions

The topics in this section provide descriptions, usage information, and examples for the Graph Lakehouse standard and advanced built-in functions.

In this section:

- Aggregate Functions 553
- Casting Functions 568
- Date and Time Functions 579
- Graph Algorithms 597
- Hash Functions 614
- Informational or Testing Functions 619
- Logical Functions 625
- Math Functions 636
- Property Paths 657
- String Functions 658
- Update Functions 681
- Window Aggregate and Ranking Functions 686
- Advanced Grouping Sets 702

Aggregate Functions

This topic describes the aggregate functions in Graph Lakehouse. For information about window aggregates, see [Window Aggregate and Ranking Functions](#).

- **AVG**: Calculates the average (arithmetic mean) value for a group of numbers.
- **CHOOSE_BY_MAX**: Returns the value from a group that corresponds to the maximum value from another group.
- **CHOOSE_BY_MIN**: Returns the value from a group that corresponds to the minimum value from another group.

- **COUNT**: Counts the number of values that exist for a group.
- **GROUP_CONCAT**: Concatenates a group of strings into a single string.
- **MAX**: Returns the maximum value from a group of values.
- **MEDIAN**: Returns the median number out of a group of numbers.
- **MIN**: Returns the minimum value from a group of values.
- **MODE**: Returns the mode (the value that occurs most frequently) from a group of values.
- **MODE_PERCENT**: Calculates the percentage of values in a group that belong to the mode.
- **SAMPLE**: Returns an arbitrary value from the specified group of values.
- **SUM**: Calculates the sum of the numbers within a group.
- **VAR**: Calculates the unbiased (sample) variance of a group of numbers.
- **VARP**: Calculates the biased (population) variance of a group of numbers.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

Note

A GROUP BY statement is required for queries that contain aggregate functions if the results clause lists non-aggregate variables. Include all non-aggregated variables in the GROUP BY statement.

AVG

This function calculates the average (arithmetic mean) value for a group of numbers.

Syntax

```
AVG (number)
```

Argument	Type	Description
number	numeric	The numeric value for which to calculate the average.

Returns

Type	Description
number	The arithmetic mean of the input values.

Examples

The following example queries the sample Tickit data set to determine the average number of seats in the venues in each state. Since the results clause contains a non-aggregated variable (?state), a **GROUP BY** clause is required for grouping on ?state.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?state (ROUND(AVG(?seats)) AS ?avg_seats)
FROM <http://anzograph.com/tickit>
WHERE {
    ?s tickit:venuestate ?state .
    ?s tickit:venueSeats ?seats .
}
GROUP BY ?state
ORDER BY ?state
```

```
state | avg_seats
-----+-----
CA    |      50309
CO    |      63285
DC    |      41888
FL    |      62603
GA    |      60620
IL    |      48244
IN    |      63000
```

LA		72000
MA		54342
MD		70229
MI		53391
MN		64035
MO		59217
NC		73298
NJ		80242
NY		48764
OH		56035
ON		50516
PA		53931
TN		68804
TX		56915
WA		57058
WI		57561

23 rows

The query below calculates the average total price for all of the listings in the sample Tickit data set:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT (AVG(?numtickets*?priceperticket) AS ?avg_total_price)
FROM <http://anzograph.com/tickit>
WHERE {
    ?listing tickit:priceperticket ?priceperticket .
    ?listing tickit:numtickets ?numtickets .
}
```

```
avg_total_price
-----
3034.42
1 rows
```

CHOOSE_BY_MAX

This function calculates the maximum value for one group and returns the value from another group that corresponds to the maximum from the first group.

Syntax

```
CHOOSE_BY_MAX(test, value)
```

Argument	Type	Description
test	any type	The group of values from which to find the maximum value.
value	any type	The group of values from which to return the value that corresponds to the maximum value of <code>test</code> .

Returns

Type	Description
input type	The value from the <code>value</code> group that corresponds to the maximum value from the <code>test</code> group.

CHOOSE_BY_MIN

This function calculates the minimum value for one group and returns the value from another group that corresponds to the minimum from the first group.

Syntax

```
CHOOSE_BY_MIN(test, value)
```

Argument	Type	Description
test	any type	The group of values from which to find the minimum value.
value	any type	The group of values from which to return the value that corresponds to the minimum value of <code>test</code> .

Returns

Type	Description
input	The value from the <code>value</code> group that corresponds to the minimum value from the <code>test</code>

Type	Description
type	group.

COUNT

This function counts the number of values that exist for a group.

Syntax

```
COUNT[ ( DISTINCT ( ) (value)
```

Argument	Type	Description
DISTINCT	N/A	Include the optional DISTINCT keyword to limit the results to the unique values.
value	any type	The group of values to count.

Returns

Type	Description
long	The number of values in the group.

Example

The following example queries the sample Tickit data set to count the number of people who have the same last name:

```
SELECT ?lastname (COUNT(?person) AS ?count)
FROM <http://anzograph.com/tickit>
WHERE {
    ?person <http://anzograph.com/tickit/lastname> ?lastname .
}
GROUP BY ?lastname
ORDER BY desc(?count)
LIMIT 10
```

```

lastname | count
-----+-----
Harding  |      72
Ashley   |      70
Stein    |      70
Mason    |      70
Fuentes  |      69
Christian |      69
Murphy   |      69
Madden   |      69
Clements |      68
Chandler |      68
10 rows

```

GROUP_CONCAT

This function concatenates a group of strings into a single string.

Syntax

```

GROUP_CONCAT (group ; [ SEPARATOR = "separator_char" ] ; [ ROW_LIMIT = max_rows ] ;
[ PRE = "prefix" ] ; [ VALUE_SERIALIZE = serialize ] ; [ DELIMIT_BLANKS = separate_
blanks ] ;
[ MAX_LENGTH = string_length ] ; [ SUFFIX = "suffix" ] )

```

Argument	Type	Description
group	string	The group of strings to concatenate.
separator_char	string	Optional argument that defines the separator to use between the values in returned strings. When SEPARATOR is omitted, Graph Lakehouse separates values with a space.
max_rows	int	Optional argument that puts a maximum limit on the number of rows to retrieve for the group. When ROW_LIMIT is omitted, the default is <code>unlimited</code> . Note that Graph Lakehouse performs the GROUP_CONCAT for each slice separately and combines the results from each slice. The ROW_LIMIT is applied to each slice, not the total result. Therefore, the total number of values that are

Argument	Type	Description
		concatenated will be larger than the specified limit, proportional to the number of slices in the cluster.
prefix	string	Optional string to add as a prefix to the resulting string.
serialize	boolean	Optional argument that indicates whether returned values should be serialized with the value's data type. When <code>VALUE_SERIALIZE</code> is omitted, the default is <code>false</code> .
separate_blanks	boolean	Optional argument that indicates whether to delimit blanks with the <code>SEPARATOR</code> value. When <code>DELIMIT_BLANKS</code> is omitted, the default is <code>false</code> .
string_length	int	Optional argument that limits the resulting strings to a maximum character length. Graph Lakehouse has a 2MB (~2,000,000 characters) limit on the length of strings and displays an error if <code>GROUP_CONCAT</code> returns a string that is longer than 2000000. When <code>MAX_LENGTH</code> is omitted, the default is <code>unlimited</code> .
suffix	string	Optional argument that defines a suffix to add to the resulting strings. When <code>SUFFIX</code> is omitted, Graph Lakehouse adds an empty string as the suffix.

Returns

Type	Description
string	The concatenated string.

Example

The query below concatenates the list of friends for 10 people in the sample Tickit data set. Since the GROUP_CONCAT expression includes ROW_LIMIT=2, Graph Lakehouse limits the records to two for each slice (or shard) of data.

```
SELECT ?person (GROUP_CONCAT(?id;SEPARATOR=",";ROW_LIMIT=2) AS ?friends)
FROM <http://anzograph.com/tickit>
WHERE {
    ?person <http://anzograph.com/tickit/friend> ?friend .
    BIND(STRATER(STR(?friend), "http://anzograph.com/tickit/") as ?id)
}
GROUP BY ?person
ORDER BY ?person
LIMIT 10
```

person	friends
http://anzograph.com/tickit/person1	person2894, person20624, person33618, person47127
http://anzograph.com/tickit/person10	person3136, person22714, person2509, person24535
http://anzograph.com/tickit/person100	person42775, person29725, person27334, person24553
http://anzograph.com/tickit/person1000	person19040, person39066, person2236, person9089
http://anzograph.com/tickit/person10000	person43706, person37085, person18874, person31270
http://anzograph.com/tickit/person10001	person3389, person44830, person4720, person307
http://anzograph.com/tickit/person10002	person46462, person43989, person46491, person31130
http://anzograph.com/tickit/person10003	person31544, person19595, person23460, person28465
http://anzograph.com/tickit/person10004	person11070, person19845, person11172, person24252
http://anzograph.com/tickit/person10005	person33888, person9467, person35761, person47709

10 rows

MAX

This function returns the maximum value from a group of values.

Syntax

```
MAX (value)
```

Argument	Type	Description
value	any type except boolean	The group of values for which to return the maximum value.

Returns

Type	Description
input type	The maximum value from the group.

Example

The following example queries the sample Tickit data to list the top 10 events with the highest number of tickets sold in one transaction:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event (MAX(?tickets) AS ?max_tickets)
FROM <http://anzograph.com/tickit>
WHERE {
    ?listing tickit:numtickets ?tickets .
    ?listing tickit:eventid ?id .
    ?id tickit:eventname ?event .
}
GROUP BY ?event
ORDER BY desc(?max_tickets)
LIMIT 10
```

event	max_tickets
Akon	30
Beatles LOVE	30
The Country Girl	30
Sarah Brightman	30
Jesse Lacey	30

```
Spring Awakening | 30
Le Reve         | 30
Das Rheingold   | 30
Macbeth         | 30
King Lear       | 30
10 rows
```

MEDIAN

This function returns the median value from a group of numbers. The median is the number in the group where half of the numbers are greater than the number and half are less than the number.

Syntax

```
MEDIAN (number)
```

Argument	Type	Description
number	numeric	The group of numeric values for which to calculate the median.

Returns

Type	Description
number	The median for the group.

MIN

This function returns the minimum value from a group of values.

Syntax

```
MIN (value)
```

Argument	Type	Description
value	any type except boolean	The group of values for which to return the minimum value.

Returns

Type	Description
input type	The minimum value from the group.

Example

The following example queries the sample Tickit data to list the 10 events with the lowest price paid for tickets:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event (MIN(?paid) AS ?min_paid)
FROM <http://anzograph.com/tickit>
WHERE {
    ?s tickit:pricepaid ?paid .
    ?s tickit:eventid ?id .
    ?id tickit:eventname ?event .
}
GROUP BY ?event
ORDER BY ?min_paid
LIMIT 10
```

event	min_paid
Legally Blonde	20
Spring Awakening	20
Green Day	20
Keb Mo	20
Thurgood	20
King Lear	20
Macbeth	20
The Country Girl	20
Ringo Starr	20
August: Osage County	20

10 rows

MODE

This function returns the mode from a group of values. The mode is the value that occurs most frequently in the group.

Syntax

```
MODE (value)
```

Argument	Type	Description
value	any type	The group of values for which to return the mode.

Returns

Type	Description
input type	The mode from the group.

MODE_PERCENT

This function calculates the percentage of values in a group that belong to the mode.

Syntax

```
MODE_PERCENT (value)
```

Argument	Type	Description
value	numeric	The group of values for which to calculate the mode percent.

Returns

Type	Description
double	The percentage of values that belong to the mode.

SAMPLE

This function returns an arbitrary value from the specified group of values.

Syntax

```
SAMPLE (value)
```

Argument	Type	Description
value	any type	The group of values from which to choose a sample value.

Returns

Type	Description
input type	The arbitrary value from the group.

SUM

This function calculates the sum of the numbers within a group.

Syntax

```
SUM (number)
```

Argument	Type	Description
number	numeric	The group of numbers to sum.

Returns

Type	Description
number	The sum of the values in the group.

Example

The following example queries the sample Tickit data set to determine the most unpopular events by returning the 10 events with the least number of ticket sales. The query uses the SUM aggregate function to calculate the total tickets for each event.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event ?category (SUM(?qty) AS ?total_tickets)
FROM <http://anzograph.com/tickit>
WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?eventid tickit:catid ?catid .
    ?catid tickit:catname ?category .
}
GROUP BY ?event ?category
ORDER BY ?total_tickets
LIMIT 10
```

event	category	total_tickets
White Christmas	Musicals	35
Joshua Radin	Pop	75
Martina McBride	Pop	101
Beach Boys	Pop	112
Linda Ronstadt	Pop	116
Teena Marie	Pop	124
Indigo Girls	Pop	125
Billy Idol	Pop	141
Mogwai	Pop	146
Stephenie Meyer	Pop	151

10 rows

VAR

This function calculates the unbiased (sample) variance for a group of numbers.

Syntax

```
VAR(value)
```

Argument	Type	Description
value	numeric	The numeric value that defines the set of numbers for which to measure the variance.

Returns

Type	Description
number	The unbiased variance for the group.

VARP

This function calculates the biased (population) variance for a group of numbers.

Syntax

```
VARP (value)
```

Argument	Type	Description
value	number	The value that defines the set of numbers for which to measure the population variance.

Returns

Type	Description
double	The biased variance for the group.

Casting Functions

This topic describes the functions that are available for coercing data types in Graph Lakehouse.

- **BNODE**: Creates a blank node.
- **DATATYPE**: Returns the data type of the given value.

- **DATETIME (or xsd:dateTime)**: Returns a dateTime value from the given string, long, or dateTime.
- **DUR_TO_USECS**: Casts a duration value to microseconds.
- **ENCODE_FOR_URI**: Encodes the specified string as a URI.
- **FORMATDATETIME**: Converts a value to a string in the specified dateTime format.
- **FORMATDURATION**: Converts a value into a string in the specified duration format.
- **HEX**: Converts a long value to a hexadecimal string.
- **HEX2DEC**: Converts a hexadecimal string to a long value.
- **PARSEDATE**: Attempts to convert the given string to a date, time, or dateTime value.
- **RADIANS**: Converts to radians an angle value that is in degrees.
- **SERIALIZE**: Creates a string representation of the input value.
- **STR**: Casts a value to a string.
- **URI**: Casts a string to a URI.
- **USECS_TO_DUR**: Converts a microseconds value to a duration.
- **UUID**: Generates a Universally Unique Identifier (UUID).

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

BNODE

This function creates a blank node.

Syntax

```
BNODE([ value ])
```

Argument	Type	Description
value	string	An optional string value from which to create the blank node.

Returns

Type	Description
blank node	The generated blank node.

DATATYPE

This function returns the data type of the given value.

Syntax

```
DATATYPE(value)
```

Argument	Type	Description
value	any	The value for which to return the data type.

Returns

Type	Description
data type URI	The data type.

DATETIME (or xsd:dateTime)

This function returns a dateTime value from the given long, double, date, or time value.

Syntax

```
DATETIME (value)
```

Argument	Type	Description
value	long, double, date, time	The value from which to return a dateTime.

Returns

Type	Description
dateTime	The dateTime value.

DUR_TO_USECS

This function calculates the time in microseconds from a duration value.

Syntax

```
DUR_TO_USECS (value)
```

Argument	Type	Description
value	duration	The duration value from which to calculate the time in microseconds.

Returns

Type	Description
long	The number of microseconds.

Example

```
SELECT (DUR_TO_USECS ("PT2H11M48.376S"^^xsd:duration) as ?microseconds)
```

```
microseconds
-----
7908376000
1 rows
```

ENCODE_FOR_URI

This function encodes the specified string as a URI and returns a string in URI format.

Syntax

```
ENCODE_FOR_URI (text)
```

Argument	Type	Description
text	string	The string value to encode as a URI.

Returns

Type	Description
string	The string as a URI.

Example

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT DISTINCT (ENCODE_FOR_URI(?eventname) as ?event)
FROM <http://anzograph.com/tickit>
WHERE {
    ?s tickit:eventid ?eventid .
    ?eventid tickit:eventname ?eventname .
}
ORDER BY ?event
LIMIT 10
```

```
event
-----
.38%20Special
3%20Doors%20Down
70s%20Soul%20Jam
```

```
A%20Bronx%20Tale
A%20Catered%20Affair
A%20Chorus%20Line
A%20Christmas%20Carol
A%20Doll%27s%20House
A%20Man%20For%20All%20Seasons
A%20Midsummer%20Night%27s%20Dream
```

10 rows

HEX

This function converts a long value to a hexadecimal string.

Syntax

```
HEX (value)
```

Argument	Type	Description
value	long	The value to convert to a hexadecimal string.

Returns

Type	Description
string	The hex value.

HEX2DEC

This function converts a hexadecimal string to a long value.

Syntax

```
HEX2DEC (value)
```

Argument	Type	Description
value	string	The hexadecimal string to convert.

Returns

Type	Description
long	The converted value.

FORMATDATETIME

This function converts a value into a string with the specified date`Time` format.

Syntax

```
FORMATDATE (value, format)
```

Argument	Type	Description
value	long, double, date, time, date <code>Time</code>	The value to convert to a string in the specified <code>format</code> .
format	string	The format to use for the resulting date <code>Time</code> string. Graph Lakehouse supports <code>YYYY-MM-DDThh:mm:ss</code> format.

Returns

Type	Description
string	The date <code>Time</code> as a string.

FORMATDURATION

This function converts a value into a string with the specified duration format.

Syntax

```
FORMATDURATION (value, format)
```

Argument	Type	Description
value	long, string, duration	The value to convert to a string in the specified <code>format</code> .
format	string	The format to use for the resulting duration string. Graph Lakehouse supports <code>PnYnMnDTnHnMnS</code> format.

Returns

Type	Description
string	The duration as a string.

PARSEDATE

This function attempts to convert the given string to a date, time, or dateTime value.

Syntax

```
PARSEDATE(value [, output_type ])
```

Argument	Type	Description
value	string	The string or plain literal value to convert to a date, time, or dateTime.
output_type	URI	An optional URI (<code>xsd:date</code> , <code>xsd:time</code> , or <code>xsd:dateTime</code>) that specifies the type of value to return. If <code>output_type</code> is not specified, <code>dateTime</code> is returned.

Returns

Type	Description
date, time, or dateTime	The conversion of the string to the desired type.

RADIANS

This function converts to radians an angle value that is in degrees.

Syntax

```
RADIANS (angle)
```

Argument	Type	Description
angle	double	The angle value to convert to radians.

Returns

Type	Description
double	The angle in radians.

SERIALIZE

This function returns a string representation of the input value.

Syntax

```
SERIALIZE (value)
```

Argument	Type	Description
value	any type except URI	The value for which to generate a string representation.

Returns

Type	Description
string	The string representation of the input value.

STR

This function casts the specified value to a string.

Syntax

```
STR(value)
```

Argument	Type	Description
value	any	The value to convert to a string.

Returns

Type	Description
string	The value as a string.

URI

This function casts the specified string to a URI.

Syntax

```
URI(value)
```

Argument	Type	Description
value	string	The value to convert to a URI.

Returns

Type	Description
URI	The value as a URI.

USECS_TO_DUR

This function converts a number of microseconds in long, duration, or string format to a duration value.

Syntax

```
USECS_TO_DUR(value)
```

Argument	Type	Description
value	long, duration, string	The microseconds value to convert to a duration.

Returns

Type	Description
duration	The input value as a duration.

Example

```
SELECT (USECS_TO_DUR(76555373888) as ?duration)
```

```
duration
-----
PT21H15M55.373888S
1 rows
```

UUID

This function generates a Universally Unique Identifier (UUID).

Syntax

```
UUID()
```

Returns

Type	Description
URI	The UUID.

Date and Time Functions

This topic describes the date, time, and duration functions in Graph Lakehouse.

- **DATE**: Returns an `xsd:date` value based on the specified year, month, and day.
- **DATETIME (or `xsd:dateTime`)**: Returns a `dateTime` value from the given long, double, date, or time value.
- **DAY**: Returns the day of the month from the specified date or `dateTime`.
- **DAYSFROMDURATION**: Returns the days portion of a duration value.
- **DUR_TO_MILLIS**: Calculates the time in milliseconds from a duration value.
- **DUR_TO_USECS**: Converts a duration value to microseconds.
- **FORMATDATETIME**: Converts a value to a string in the specified `dateTime` format.
- **FORMATDURATION**: Converts a value into a string in the specified duration format.
- **HOURS**: Returns the hour portion of the given time or `dateTime` value.
- **MASKEDDATETIME**: Replaces the year, month, day, hour, minute, second, and millisecond values for the given date or `dateTime` value with the new date and time values that you specify.
- **MILLIS_TO_DUR**: Converts milliseconds to a duration value.
- **MINUTES**: Returns the minutes portion of the given time or `dateTime` value.
- **MONTH**: Returns the month portion of the given date or `dateTime` value.
- **NOW**: Returns the current server date and time.
- **NOWMILLIS**: Returns the current server date and time in epoch milliseconds.

- **PARSEDATE**: Attempts to convert the given string to a date, time, or dateTime value.
- **SECONDS_DBL**: Returns the seconds portion of the given dateTime value.
- **TIME**: Returns an xsd:time value based on the specified hour, minute, and second values.
- **TIMEZONE**: Returns as a duration the timezone from a dateTime value.
- **TODAY**: Returns today's date based on the server date.
- **TOMILLIS**: Converts a date or dateTime value to milliseconds.
- **TZ**: Returns as a string the timezone from a dateTime value.
- **USECS_TO_DUR**: Converts a microseconds value to a duration.
- **WEEKDAY**: Returns the day of the week from a date or dateTime value.
- **WEEKNUM**: Returns the week of the year in which the given date or dateTime occurs.
- **YEAR**: Returns the year portion of the given dateTime value.
- **YEARDAY**: Returns the day of the year in which the given date or dateTime occurs.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

DATE

This function returns an xsd:date value based on the specified year, month, and day values.

Syntax

```
DATE(year, month, day)
```

Argument	Type	Description
year	long	A number that represents the year.
month	long	A number that represents the month.
day	long	A number that represents the day.

Returns

Type	Description
date	The date according to the input values.

DATETIME (or xsd:dateTime)

This function returns a dateTime value from the given long, double, date, or time value.

Syntax

```
DATETIME (value)
```

Argument	Type	Description
value	long, double, date, time	The value from which to return a dateTime.

Returns

Type	Description
dateTime	The dateTime value.

DAY

This function returns the day of the month from the specified date or dateTime value.

Syntax

```
DAY (value)
```

Argument	Type	Description
value	date, dateTime	The value from which to return the day of the month.

Returns

Type	Description
int	The day of the month.

DAYSFROMDURATION

This function returns the days portion of a duration value.

Syntax

```
DAYSFROMDURATION (value)
```

Argument	Type	Description
value	duration	The duration value from which to return the days.

Returns

Type	Description
long	The number of days in the duration.

DUR_TO_MILLIS

This function calculates the time in milliseconds from a duration value.

Syntax

```
DUR_TO_MILLIS (value)
```

Argument	Type	Description
value	duration	The duration value from which to calculate the time in milliseconds.

Returns

Type	Description
long	The number of milliseconds.

DUR_TO_USECS

This function calculates the time in microseconds from a duration value.

Syntax

```
DUR_TO_USECS (value)
```

Argument	Type	Description
value	duration	The duration value from which to calculate the time in microseconds.

Returns

Type	Description
long	The number of microseconds.

Example

```
SELECT (DUR_TO_USECS ("PT2H11M48.376S"^^xsd:duration) as ?microseconds)
```

```
microseconds
-----
7908376000
1 rows
```

FORMATDATETIME

This function converts a value into a string with the specified date`Time` format.

Syntax

```
FORMATDATE (value, format)
```

Argument	Type	Description
value	long, double, date, time, date <code>Time</code>	The value to convert to a string in the specified <code>format</code> .
format	string	The format to use for the resulting date <code>Time</code> string. Graph Lakehouse supports <code>YYYY-MM-DDThh:mm:ss</code> format.

Returns

Type	Description
string	The date <code>Time</code> as a string.

FORMATDURATION

This function converts a value into a string with the specified duration format.

Syntax

```
FORMATDURATION (value, format)
```


Argument	Type	Description
value	long, string, duration	The value to convert to a string in the specified <code>format</code> .
format	string	The format to use for the resulting duration string. Graph Lakehouse supports <code>PnYnMnDTnHnMnS</code> format.

Returns

Type	Description
string	The duration as a string.

HOURS

This function returns the hour portion of the given `dateTime` value.

Syntax

```
HOURS (value)
```

Argument	Type	Description
value	time, <code>dateTime</code>	The <code>dateTime</code> value from which to return the hours portion.

Returns

Type	Description
int	The hour.

MASKEDDATETIME

This function replaces the year, month, day, hour, minute, second, and millisecond values for the given date or `dateTime` value with the new date and time values that you specify.

Syntax

```
MASKEDDATETIME (value, year, month, day, hour, minute, second, milliseconds)
```

Argument	Type	Description
value	date, dateTime	The date or dateTime for which to replace the year, month, date, hour, minute, second, and milliseconds values.
year	int	The year to include in the resulting dateTime value.
month	int	The month to include in the resulting dateTime value.
day	int	The day to include in the resulting dateTime value.
hour	int	The hour to include in the resulting dateTime value.
minute	int	The minutes value to include in the resulting dateTime value.
second	int	The seconds value to include in the resulting dateTime value.
milliseconds	int	The milliseconds value to include in the resulting dateTime value.

Returns

Type	Description
dateTime	The dateTime value with the specified input values.

MILLIS_TO_DUR

This function converts milliseconds to a duration value.

Syntax

```
MILLIS_TO_DUR(value)
```

Argument	Type	Description
value	long, double, duration, string	The number of milliseconds.

Returns

Type	Description
duration	The duration value.

MINUTES

This function returns the minutes portion of the given time or dateTime value.

Syntax

```
MINUTES(value)
```

Argument	Type	Description
value	time, dateTime	The value from which to return the minutes portion.

Returns

Type	Description
int	The minutes portion of the input value.

MONTH

This function returns the month portion of the given date or dateTime value.

Syntax

MONTH (value)

Argument	Type	Description
value	date, dateTime	The value from which to return the month portion.

Returns

Type	Description
int	The month number.

Example

The query below uses the MONTH function to determine the most popular month to hold events, based on the number of events that occur in each month.

```
SELECT ?month (COUNT(?eventid) AS ?num_events)
FROM <http://anzograph.com/tickit>
WHERE {
  { SELECT ?eventid (MONTH(?eventtime) AS ?month)
    WHERE {
      ?eventid <http://anzograph.com/tickit/starttime> ?eventtime .
      ?sale <http://anzograph.com/tickit/eventid> ?eventid .
    }
  }
}
GROUP BY ?month
ORDER BY DESC(?num_events)
```

```
month | num_events
-----+-----
  3 |      34935
  9 |      34346
 10 |      33856
  7 |      33770
  5 |      33638
```

11		33599
8		33542
12		33022
4		32864
6		32418
2		22503
1		6460

12 rows

NOW

This function returns the current server date and time.

Syntax

```
NOW ( )
```

Returns

Type	Description
dateTime	The current server date and time.

NOWMILLIS

This function returns the current server date and time in epoch milliseconds.

Syntax

```
NOWMILLIS ( )
```

Returns

Type	Description
long	The current server date and time in milliseconds.

PARSEDATE

This function attempts to convert the given string to a date, time, or dateTime value.

Syntax

```
PARSEDATE (value [, output_type ])
```

Argument	Type	Description
value	string	The string or plain literal value to convert to a date, time, or dateTime.
output_type	URI	An optional URI (<code>xsd:date</code> , <code>xsd:time</code> , or <code>xsd:dateTime</code>) that specifies the type of value to return. If <code>output_type</code> is not specified, <code>dateTime</code> is returned.

Returns

Type	Description
date, time, or dateTime	The conversion of the string to the desired type.

SECONDS_DBL

This function returns the seconds portion of the given time or dateTime value.

Syntax

```
SECONDS_DBL (value)
```

Argument	Type	Description
value	time, dateTime	The value from which to return the seconds portion.

Returns

Type	Description
double	The seconds portion of the input value.

TIME

This function returns an `xsd:time` value based on the specified hour, minute, and second values.

Syntax

```
TIME(hour, minute, second)
```

Argument	Type	Description
hour	long	A number that represents the hour.
minute	long	A number hat represents the minute.
second	long, double	A number that represents the seconds.

Returns

Type	Description
time	The time according to the input values.

TIMEZONE

This function returns the timezone part of a `dateTime` value as a duration.

Syntax

```
TIMEZONE(value)
```

Argument	Type	Description
value	dateTime	The value from which to retrieve the timezone.

Returns

Type	Description
duration	The timezone.

TODAY

This function returns today's date based on the server date.

Syntax

```
TODAY ()
```

Returns

Type	Description
date	Today's date according to the server.

TOMILLIS

This function converts a date or dateTime value to the number of milliseconds.

Syntax

```
TOMILLIS (value)
```

Argument	Type	Description
value	date, dateTime	The value to convert to milliseconds.

Returns

Type	Description
long	The number of milliseconds.

TZ

This function returns the timezone part of a dateTime value as a string.

Syntax

```
TZ (value)
```

Argument	Type	Description
value	dateTime	The value from which to retrieve the timezone.

Returns

Type	Description
string	The timezone.

USECS_TO_DUR

This function converts a number of microseconds in long, duration, or string format to a duration value.

Syntax

```
USECS_TO_DUR (value)
```

Argument	Type	Description
value	long, duration, string	The microseconds value to convert to a duration.

Returns

Type	Description
duration	The input value as a duration.

Example

```
SELECT (USECS_TO_DUR(76555373888) as ?duration)
```

```
duration
-----
PT21H15M55.373888S
1 rows
```

WEEKDAY

This function returns the day of the week from a date or dateTime value.

Syntax

```
WEEKDAY (value [, day_number_start ])
```

Argument	Type	Description
value	date, dateTime	The date or dateTime value from which to return the day of the week.
day_ number_ start	long	An optional value of 1, 2, or 3 that defines how the days of the week are represented as numbers. • 1 means Sunday is day 1. Saturday is day 7. • 2 means Monday is day 1. Sunday is day 7. • 3 means Monday is day 0. Sunday is day 6. If <code>day_number_start</code> is not specified, the default value is 1.

Returns

Type	Description
int	The day of the week from the input values.

WEEKNUM

This function returns the week of the year in which the given date or dateTime occurs.

Syntax

```
WEEKNUM(value [, day_week_begins ])
```

Argument	Type	Description
value	date, dateTime	The date or dateTime value from which to return the week number.
day_week_begins	long	An optional value of 1 or 2 that defines which day the weeks start on. • 1 means a new week starts on Sunday. • 2 means a new week starts on Monday. If <code>day_week_begins</code> is not specified, the default value is 1.

Returns

Type	Description
int	The week of the year the input value falls in.

YEAR

This function returns the year portion of the given dateTime value.

Syntax

```
YEAR(value)
```

Argument	Type	Description
value	dateTime	The dateTime value to return the year from.

Returns

Type	Description
int	The year portion of the input values.

Example

The example below uses the NOW and YEAR functions to calculate the approximate ages of 10 people in the sample Tickit data set. The resulting age values are approximations because the calculation excludes days and months.

```
SELECT ?person ((YEAR(?date))-(YEAR(xsd:dateTime(?birthdate))) AS ?age)
FROM <http://anzograph.com/tickit>
WHERE {
    ?person <http://anzograph.com/tickit/birthday> ?birthdate .
    BIND(xsd:dateTime(NOW()) AS ?date)
}
ORDER BY ?person
LIMIT 10
```

person	age
http://anzograph.com/tickit/person1	55
http://anzograph.com/tickit/person10	75
http://anzograph.com/tickit/person100	32
http://anzograph.com/tickit/person1000	38
http://anzograph.com/tickit/person10000	77
http://anzograph.com/tickit/person10001	27
http://anzograph.com/tickit/person10002	75
http://anzograph.com/tickit/person10003	69
http://anzograph.com/tickit/person10004	50
http://anzograph.com/tickit/person10005	72

10 rows

YEARDAY

This function returns the day of the year from the specified date or dateTime value.

Syntax

```
YEARDAY (value)
```

Argument	Type	Description
value	date, dateTime	The value to return the day of the year from.

Returns

Type	Description
int	The day of the year.

Graph Algorithms

Graph Lakehouse offers graph algorithms for exploring and computing metrics for graphs, nodes, and relationships. This section describes each of the algorithms.

Tip

All graph algorithm example queries are run against a sample dataset that is available for download. See [Sample Data for Graph Algorithm Queries](#) for information.

In this section:

Centrality Algorithms

Centrality algorithms identify important nodes in a graph:

- [PageRank](#): Ranks the nodes in a graph by their relative importance or influence. Google uses PageRank to rank websites in their search engine results.

- **Betweenness Centrality:** Detects the amount of influence a vertex has over the flow of information in a graph.

PageRank

The PageRank algorithm ranks the nodes in a graph by their relative importance or influence. PageRank determines each node's ranking by identifying the number of links to the node, the outbound links from the node, and the quality of the links. The quality of a link is determined by the importance (PageRank) of the connected nodes. For labeled property graphs, the PageRank algorithm accepts an edge property that can be considered a relationship weight to factor into the PageRank calculation.

Syntax

The PageRank algorithm is available in the `graphalgo` extension library (http://cambridgesemantics.com/anzograph/graphalgo#page_rank) and is implemented as a procedure. To incorporate the PageRank algorithm in a query, use the following syntax in the FROM clause. The arguments that are links are described below.

```
SELECT triple_patterns_and_expressions
# The FROM clause lists the URI for the page_rank algorithm and
# includes in parentheses the input parameters for the algorithm.
FROM <http://cambridgesemantics.com/anzograph/graphalgo#page_rank>
(
  <graph_URI>,
  <edge_URI>,
  [ <weighted_property>, ]
  [ damping_factor, ]
  [ max_iterations, ]
  [ error_tolerance, ]
  [ normalized ]
)
WHERE {
  ...
}
```

Argument	Range	Description
graph_URI	URI	The URI of the target graph.

Argument	Range	Description
edge_URI	URI	The URI of the edge that connects the nodes to rank.
weighted_property	URI	Optional argument that defines the URI of the relationship property on the edge_URI that contains values to be factored in as a weight when calculating the importance of the nodes. When this property is omitted, <code>page_rank</code> is unweighted.
damping_factor	0.0 - 1.0	Optional argument that represents the edge traversal probability. The damping factor is an estimate of the probability that a user will stay on the page rather than follow the link (edge). The <code>damping_factor</code> value is subtracted from 1.0 in the calculation. The default value is <code>0.85</code> .
max_iterations	1 - 100	Optional argument that specifies the maximum number of times to iterate through the graph to adjust approximate PageRank values. The default value is <code>40</code> .
error_tolerance	0.0 - 0.1	Optional argument that specifies the error tolerance to use. If the sum of the error values for all nodes is below this tolerance value, PageRank iterations are stopped. The default value is <code>1e-8</code> .
normalized	boolean	Optional argument that controls whether to produce PageRank values that are between 0 and 1. The default value is <code>false</code> .

Examples

The example below uses the unweighted PageRank algorithm to find the 10 most connected airports. The edge to operate on is defined as the `hasRouteTo` URI, which links the airport nodes.

```
SELECT ?airport ?rank
FROM <http://cambridgesemantics.com/anzograph/graphalgo#page_rank>(
  <http://anzograph.com/airline_flight_network>,
  <http://anzograph.com/flights/hasRouteTo>
```

```

)
WHERE
{
    ?airport ?p ?rank .
}
ORDER BY desc(?rank)
LIMIT 10

```

The results show that ORD (Chicago O'Hare) has the highest PageRank. It has the highest number of links to other airports.

airport	rank
http://anzograph.com/flights/Airport/ORD	13.4122
http://anzograph.com/flights/Airport/DFW	13.1632
http://anzograph.com/flights/Airport/ATL	12.8073
http://anzograph.com/flights/Airport/DEN	10.3813
http://anzograph.com/flights/Airport/IAH	8.38149
http://anzograph.com/flights/Airport/SLC	6.90182
http://anzograph.com/flights/Airport/MSP	6.24396
http://anzograph.com/flights/Airport/SFO	5.50728
http://anzograph.com/flights/Airport/PHX	5.27483
http://anzograph.com/flights/Airport/LAX	5.24546

10 rows

By including the edge property, `<http://anzograph.com/flights/distanceMiles>`, the example below uses the weighted PageRank algorithm to find the 10 most connected airports where distance between the airports is factored into the calculation.

```

SELECT ?airport ?rank
FROM <http://cambridgesemantics.com/anzograph/graphalgo#page_rank>(
    <http://anzograph.com/airline_flight_network>,
    <http://anzograph.com/flights/hasRouteTo>,
    <http://anzograph.com/flights/distanceMiles>
)
WHERE
{
    ?airport ?p ?rank .
}
ORDER BY desc(?rank)
LIMIT 10

```


airport	rank
http://anzograph.com/flights/Airport/ORD	2.445
http://anzograph.com/flights/Airport/ATL	1.935
http://anzograph.com/flights/Airport/DFW	1.68
http://anzograph.com/flights/Airport/SLC	1.17
http://anzograph.com/flights/Airport/MSP	0.915
http://anzograph.com/flights/Airport/DTW	0.66
http://anzograph.com/flights/Airport/SFO	0.66
http://anzograph.com/flights/Airport/FLL	0.5325
http://anzograph.com/flights/Airport/ANC	0.5325
http://anzograph.com/flights/Airport/DEN	0.5325

10 rows

Betweenness Centrality

Betweenness centrality is a measure of the amount of influence a vertex has over the flow of information in a graph. The amount of influence is determined by the number of shortest paths that pass through a vertex. The Betweenness Centrality algorithm computes the shortest path between each pair of vertices in a graph and assigns a score to each vertex based on the number of shortest paths that intersect it. Vertices that frequently lie on the shortest paths between other nodes have higher betweenness centrality scores.

Syntax

To incorporate the Betweenness Centrality algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:betweenness>
{
  [] <csi:binding-vertex> ?binding_vertex_variable ;
  <csi:binding-centrality> ?centrality_variable ;
  <csi:edge-label> <edge_uri> .
}
```

Argument	Range	Description
binding-vertex	variable	Required argument that defines the name to use for the result column that lists the source nodes (vertices).

Argument	Range	Description
binding- centrality	variable	Required argument that defines the name to use for the result column that lists the computed centrality score.
edge-label	URI	Required argument that lists the edge URI that defines the graph to operate on. The graph is the set of vertices that are connected by this URI.

Example

The example below uses the Betweenness Centrality algorithm to find the 10 airports with the highest centrality. The edge to operate on is defined as the `hasRouteTo` URI, which links the airport vertices.

```
prefix : <http://anzograph.com/data#>
prefix fl: <http://anzograph.com/flights/>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix skos: <http://www.w3.org/2004/02/skos/core#>

SELECT *
FROM <http://anzograph.com/airline_flight_network>
WHERE
{
  SERVICE <csi:betweenness>
  {
    [] <csi:binding-vertex> ?airport ;
    <csi:binding-centrality> ?centrality ;
    <csi:edge-label> <http://anzograph.com/flights/hasRouteTo> .
  }
}
ORDER BY desc(?centrality)
LIMIT 10
```

airport	centrality
http://anzograph.com/flights/Airport/ORD	18985.371039
http://anzograph.com/flights/Airport/DFW	17718.273941
http://anzograph.com/flights/Airport/ATL	16286.345255
http://anzograph.com/flights/Airport/DEN	12421.967745

```

http://anzograph.com/flights/Airport/SLC | 8304.111011
http://anzograph.com/flights/Airport/MSP | 6835.823598
http://anzograph.com/flights/Airport/IAH | 6022.854234
http://anzograph.com/flights/Airport/SEA | 4895.147059
http://anzograph.com/flights/Airport/SFO | 4696.686488
http://anzograph.com/flights/Airport/DTW | 3978.338284
10 rows

```

Community Detection Algorithms

Community detection algorithms evaluate clusters of nodes and determine whether they have a tendency to strengthen or break apart:

- **Connected Components:** Identifies the connected nodes in an undirected graph.
- **Label Propagation:** Detects structures in a graph by propagating labels throughout the graph and forming groups based on the label propagation.
- **Triangle Enumeration:** Identifies each triangle in a graph.
- **Triangle Count:** Determines the number of triangles that a graph includes and calculates the average clustering coefficient for the resulting network of nodes.
- **Vertex Triangle Count:** Determines the number of triangles that a vertex is a member of and computes the clustering coefficient for the vertex.

Connected Components

The Connected Components algorithm identifies the connected vertices in an undirected graph. The algorithm returns a unique connected component ID for each vertex in the graph.

Syntax

To incorporate the Connected Components algorithm in a query, include the following SERVICE call in the WHERE clause.

```

SERVICE <csi:connected_components>
{
  [] <csi:binding-vertex>    ?binding_vertex_variable ;
  <csi:binding-id>          ?component_id_variable ;
  <csi:edge-label>          <edge_uri> ;
}

```

```
[ <csi:graph> <graph_uri> ] .
}
```

Argument	Range	Description
binding-vertex	variable	Required argument that defines the name to use for the result column that lists the source nodes (vertices).
binding-id	variable	Required argument that defines the name to use for the result column that lists the assigned component identifiers.
edge-label	URI	Required argument that lists the edge URI that connects the nodes or vertices.
graph	URI	Optional argument that specifies the graph to query.

Label Propagation

The Label Propagation algorithm detects structures in a graph by propagating labels throughout the graph and forming groups based on the label propagation.

Syntax

To incorporate the Label Propagation algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:label_propagation>
{
  [] <csi:binding-vertex> ?vertex_variable ;
  <csi:binding-label> ?label_variable ;
  <csi:edge-label> <edge_uri> ;
  [ <csi:max-iterations> number_of_iterations ] .
}
```

Argument	Range	Description
binding-	variable	Required argument that defines the name to use for the result

Argument	Range	Description
vertex		column that lists the source nodes (vertices).
binding-label	variable	Required argument that defines the name to use for the result column that lists the assigned label values.
edge-label	URI	Required argument that lists the edge URI that defines the graph to operate on. The graph is the set of vertices that are connected by this URI.
max-iterations	1 - 100	Optional argument that specifies the maximum number of times to iterate through the graph. The default value is 5.

Triangle Enumeration

The Triangle Enumeration algorithm identifies each of the triangles that exist in the specified graph. A triangle is defined as three nodes that are connected by three edges (a-b, b-c, c-a).

Syntax

To incorporate the Triangle Enumeration algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:triangles>
{
  [] <csi:binding-vertex1> ?vertex1_variable ;
    <csi:binding-vertex2> ?vertex2_variable ;
    <csi:binding-vertex3> ?vertex3_variable ;
    <csi:edge-label>      <edge_uri> .
}
```

Argument	Range	Description
binding-vertex1	variable	Required argument that defines the name to use for the result column that lists the first node in the triangle.

Argument	Range	Description
binding-vertex2	variable	Required argument that defines the name to use for the result column that lists the second node in the triangle.
binding-vertex3	variable	Required argument that defines the name to use for the result column that lists the third node in the triangle.
edge-label	URI	Required argument that lists the edge URI that defines the graph to operate on. The graph is the set of vertices that are connected by this URI.

Triangle Count

The Triangle Count algorithm determines the number of triangles that a graph includes and calculates the average clustering coefficient for the resulting network of nodes. A triangle is defined as three nodes that are connected by three edges (a-b, b-c, c-a).

Syntax

To incorporate the Triangle Count algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:triangles>
{
  [] <csi:binding-average-clustering-coefficient> ?binding_avg_cc_variable ;
    <csi:binding-triangle-count> ?triangle_count_variable ;
    <csi:edge-label> <edge_uri> .
}
```

Argument	Range	Description
binding-average-clustering-coefficient	variable	Required argument that defines the name to use for the result column that lists the average clustering coefficient value. The algorithm returns a double value that indicates the average degree to which the nodes in the network tend to cluster.

Argument	Range	Description
binding-triangle-count	variable	Required argument that defines the name to use for the result column that lists the number of triangles in the graph.
edge-label	URI	Required argument that lists the edge URI that defines the graph to operate on. The graph is the set of vertices that are connected by this URI.

Vertex Triangle Count

The Vertex Triangle Count algorithm counts number of triangles that a vertex is a member of and computes the clustering coefficient for the vertex. A triangle is defined as three nodes that are connected by three edges (a-b, b-c, c-a).

Syntax

To incorporate the Vertex Triangle Count algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:triangles>
{
  [] <csi:binding-vertex>                ?vertex_variable ;
    <csi:binding-vertex-triangle-count>   ?triangle_count_variable ;
    <csi:binding-clustering-coefficient>   ?binding_cc_variable ;
    <csi:edge-label>                      <edge_uri> .
}
```

Argument	Range	Description
binding-vertex	variable	Required argument that defines the name to use for the result column that lists each vertex.
binding-vertex-triangle-	variable	Required argument that defines the name to use for the result column that lists the number of triangles that a vertex belongs to.

Argument	Range	Description
count		
binding-clustering-coefficient	variable	Required argument that defines the name to use for the result column that lists the clustering coefficient value for each node. The algorithm returns a double value that indicates the degree to which the node tends to cluster with other nodes.
edge-label	URI	Required argument that lists the edge URI that defines the graph to operate on. The graph is the set of vertices that are connected by this URI.

Path Finding Algorithms

Path finding algorithms identify the shortest path or evaluate the availability and quality of paths:

- **All Paths**: Lists all of the paths that exist between two nodes in a graph.
- **Shortest Path**: Finds the shortest path (the path with the least cost) from a source node to the other nodes in a graph.

All Paths

The All Paths algorithm finds all of the paths that exist between a source node and destination node in a graph.

Syntax

To incorporate the All Paths algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:all_paths>
{
  [] <csi:binding-vertex>          ?vertex_variable ;
    <csi:binding-edge>             ?edge_variable ;
    <csi:binding-successor>        ?successor_variable ;
    <csi:source>                   <source_node_uri> ;
    <csi:destination>              <destination_node_uri> ;
}
```



```

[ <csi:graph>                <graph_uri> ; ]
[ <csi:edge-label>           "<edge_uri>" ; ]
[ <csi:binding-path-index>   ?path_index_variable ; ]
[ <csi:binding-path>         ?path_variable ; ]
[ <csi:min-length>           min_length ; ]
[ <csi:max-length>           max_length ; ]
[ <csi:undirected>           undirected ; ]
[ <csi:binding-orientation> ?orientation_variable ] .
}

```

Argument	Range	Description
binding-vertex	variable	Required argument that defines the name to use for the result column that lists the nodes that are reachable by the source node.
binding-edge	variable	Required argument that defines the name to use for the result column that lists the edges that exist between the nodes.
binding-successor	variable	Required argument that defines the name to use for the result column that lists the destination nodes.
source	URI	Required argument that specifies the URI of the source node.
destination	URI	Required argument that specifies the URI for the destination node.
graph	URI	Optional argument that specifies the graph to query.
edge-label	string	Optional argument that further defines the path to operate on. This property accepts complex path specifications such as W3C Property Paths . When specifying edge URIs, prefix notation is not supported. Include the full URI.
binding-path-index	variable	Optional argument that defines the name to use for the result column that lists the unique identifiers for the edges found in the path. The identifier represents the order for edges.

Argument	Range	Description
binding-path	variable	Optional argument that defines the name to use for the result column that lists the unique identifiers for the paths that are found.
min-length	int	Optional argument that specifies the minimum number of paths to evaluate.
max-length	int	Optional argument that specifies the maximum number of paths to evaluate. The default value is unlimited.
undirected	boolean	Optional argument that specifies whether to treat edges as undirected. When <code>true</code> , the algorithm assumes that paths can be traversed in both directions. The default value is <code>false</code> .
binding-orientation	variable	Optional argument to be used in conjunction with <code>undirected</code> . If <code>csi:undirected</code> is <code>true</code> , you can include this property to add a result column that lists the orientation of each path. In the results, <code>csi:binding-orientation</code> returns <code>t</code> (true) when the direction of the edge goes from the source node to the successor. <code>csi:binding-orientation</code> returns <code>f</code> (false) when the direction of the edge goes from the successor to the source node.

Shortest Path

The Shortest Path algorithm finds the shortest path from a source node to the other reachable nodes in a graph.

Syntax

To incorporate the Shortest Path algorithm in a query, include the following SERVICE call in the WHERE clause.

```
SERVICE <csi:shortest_path>
{
  [] <csi:binding-vertex>           ?binding_vertex_variable ;
```

```

    <csi:binding-predecessor>      ?predecessor_variable ;
    <csi:graph>                    <graph_uri> ;
    <csi:source-vertex>           <source_node_uri> ;
    <csi:edge-label>              <edge_uri> ;
  [ <csi:binding-distance>       ?binding_distance_variable ; ]
  [ <csi:weight>                 <property_uri> ; ]
  [ <csi:destination-vertex>     <destination_node_uri> ] .
}

```

Argument	Range	Description
binding-vertex	variable	Required argument that defines the name to use for the result column that lists the nodes that are reachable by the source-vertex.
binding-predecessor	variable	Required argument that defines the name to use for the result column that lists the nodes that are on the path between the source and destination vertices.
graph	URI	Required argument that specifies the graph to query.
source-vertex	URI	Required argument that specifies the URI of the source node.
edge-label	URI	Required argument that lists the edge URI that defines the graph to operate on. The graph is the set of vertices that are connected by this URI.
binding-distance	variable	Optional argument that adds a result column that lists the number of hops in the path between the source node and destination nodes. If <code>csi:weight</code> is specified, the binding-distance column displays the weight value.
weight	URI	Optional argument that specifies an edge property URI whose value can be used to determine the shortest path. When you do not specify a weight, the algorithm calculates the shortest path

Argument	Range	Description
		according the number of hops required to reach the destination vertex (or all nodes identified by the edge-label if destination-vertex is not specified). When weight is specified, the weight value is used as the shortest path measurement instead of the number of hops. For more information about properties, see Create a Labeled Property Graph (RDF-star) .
destination-vertex	URI	Optional argument that specifies the URI of the destination node. Include the destination-vertex URI when you want to find the shortest path between two specific vertices. When destination-vertex is excluded, Graph Lakehouse returns the shortest path between all vertices that are accessible by the edge-label. Note The value for this property is only applied when weight is included in the query. It is ignored when <code><csi:weight></code> is excluded.

Example

The following example finds the shortest path for flights from Boston (BOS) to Honolulu (HNL). In the query, the starting point or source node (`csi:source-vertex`) is the URI for BOS. To find the shortest path between BOS and HNL instead of the shortest path between BOS and all other airports that are reachable by the `edge-label`, the query includes the `csi:destination-vertex` property, which specifies the URI for HNL. The `csi:weight` property is also included and specifies the URI for the `distanceMiles` edge property. That tells the algorithm to calculate the shortest path by distance rather than number of hops.

```

prefix : <http://anzograph.com/data#>
prefix fl: <http://anzograph.com/flights/>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix skos: <http://www.w3.org/2004/02/skos/core#>

```

```

SELECT ?destination ?distance
FROM <http://anzograph.com/airline_flight_network>
WHERE
{
    SERVICE <csi:shortest_path>
    {
        [] <csi:binding-vertex> ?airport ;
        <csi:binding-predecessor> ?predecessor ;
        <csi:binding-distance> ?distance ;
        <csi:graph> <http://anzograph.com/airline_flight_network> ;
        <csi:source-vertex> <http://anzograph.com/flights/Airport/BOS> ;
        <csi:destination-vertex> <http://anzograph.com/flights/Airport/HNL> ;
        <csi:edge-label> <http://anzograph.com/flights/hasRouteTo> ;
        <csi:weight> <http://anzograph.com/flights/distanceMiles> .
    }
    ?airport fl:terminalCode ?destination .
}
ORDER BY ?distance

```

The results (shown below) determine that the shortest path from BOS to HNL is through Salt Lake City (SLC). The binding-vertex column (?distance) shows the distance for each leg of the flight.

destination	distance
BOS	0
SLC	2105
HNL	5099

3 rows

Sample Data for Graph Algorithm Queries

The graph algorithm example queries run against a sample airline flight network data set, which includes a 10,000 row subset of flight data from the Department of Transportation and airport data with information about airports and their locations. If you would like to load the sample data so that you can run the example queries, click the link below to download the `airline_flight_network_lpg.zip` file, which contains the load file, `airline_flight_network_lpg.ttl`.

📎 [Download the Airline Flight Network Sample Data](#)

Extract the .zip file and place `airline_flight_network_lpg.ttl` in a location on the Graph Lakehouse file system. Then run the following query to load the sample data into a graph called `<http://anzograph.com/airline_flight_network>`.

```
LOAD <file:/path_to_file/airline_flight_network_lpg.ttl> INTO GRAPH
<http://anzograph.com/airline_flight_network>
```

Hash Functions

This topic describes the hash functions in Graph Lakehouse.

- **HASH32**: Returns a 32-bit hash value of a string.
- **MD5**: Returns the MD5 checksum as a hexadecimal string.
- **SHA1**: Calculates the SHA-1 digest of a value.
- **SHA224**: Calculates the SHA-224 digest of a value.
- **SHA256**: Calculates the SHA-256 digest of a value.
- **SHA384**: Calculates the SHA-384 digest of a value.
- **SHA512**: Calculates the SHA-512 digest of a value.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

HASH32

This function uses a hash algorithm to encrypt strings and return a 32-bit hash value.

Syntax

```
HASH32 (text)
```

Argument	Type	Description
text	string	The string for which to return a 32-bit hash.

Returns

Type	Description
int	The 32-bit hash.

MD5

This function returns the MD5 checksum of the specified value as a hexadecimal string.

Syntax

```
MD5 (value)
```

Argument	Type	Description
value	string, boolean, int, long, float, double, date, time, dateTime	The value for which to return the MD5 checksum.

Returns

Type	Description
string	The hexadecimal string.

SHA1

This function calculates the SHA-1 digest of a value.

Syntax

```
SHA1 (value)
```

Argument	Type	Description
value	string, boolean, int, long, float, double, date, time, dateTime	The value for which to calculate the SHA-1 digest.

Returns

Type	Description
string	The SHA-1 digest.

SHA224

This function calculates the SHA-224 digest of a value.

Syntax

```
SHA224 (value)
```

Argument	Type	Description
value	string, boolean, int, long, float, double, date, time, dateTime	The value for which to calculate the SHA-224 digest.

Returns

Type	Description
string	The SHA-224 digest.

SHA256

This function calculates the SHA-256 digest of a value.

Syntax

```
SHA256 (value)
```


Argument	Type	Description
value	string, boolean, int, long, float, double, date, time, dateTime	The value for which to calculate the SHA-256 digest.

Returns

Type	Description
string	The SHA-256 digest.

Example

The example below queries the sample Tickit data set to convert social security and credit card numbers to SHA 256-bit hash values.

```
SELECT (SHA256(?card) AS ?sha2_card) (SHA256(?ssn) AS ?sha2_ssn)
FROM <http://anzograph.com/tickit>
WHERE {
    ?person <http://anzograph.com/tickit/ssn> ?ssn ;
           <http://anzograph.com/tickit/card> ?card .
}
ORDER BY ?sha2_card
LIMIT 10
```

sha2_card	sha2_ssn
-----+-----	
000046293f44419e08ddb59b5ce9593c5b14fb80e2e49924d0	
9cbee8c1c7f8b526ddf6bbe62d5c9ad6c6abdfcc674475a57c	
00010cfc90dbc6ce312002ef5072118a772462075b5199caa7	
7e1f18583d167c6d82fe737650c5aaa3e83350330ed3099f3c	
000365bf7e342feaед1a5b2b9ab9d0643570089278af12a242	
2a79e83495384866ba5520daff432ba3dca6f4a8d18989b0dd	
00037e28f93315b2b9a821c93d7261bee1fed421ca7125f0cf	
3ef7706f78662a401e9fa9448d8cbb470f41d65ffbe1b0f8cd	
0004740568f42f9eb8ac71a95d672a89895a5331e62d8bd506	
ee48859e41cd3ef64539f0cc09fd377ab18662f6fc31012855	
0006406ace683e967a338d41ed04fdcefd8637c50243fa450c	
faa394399dea5cc8b56bcb209abe787ec9d2d07d196fa9e270	

```
00091fec35b7b07f881bd56b7a5e96c6da8ceb2f7fc18dc89f |
4fdb3c8afd2281e95cacb16867946ae74fa55f9a8e33143d93
000b33a49dfc953ce1b98de194394dc60498ad94d2d4e168c7 |
b822e2758ab80f1866d31981dc883d1b628f1ef90b5ab4ea9a
000b4fa0c93c04a44f230bfb0545d83f9a70ed7e9897d621b3 |
ad03804fa16e6de63bcaff1bfff1b5754381f80c9d4fb436a05
000b78f1bb238dc1af52bbc30c6947c225274c3d3114e9b9b6 |
a28d6b0fe5c2506ce3fd0828cb9d6afd090b68084ea00291f0
10 rows
```

SHA384

This function calculates the SHA-384 digest of a value.

Syntax

```
SHA384 (value)
```

Argument	Type	Description
value	string, boolean, int, long, float, double, date, time, dateTime	The value for which to calculate the SHA-384 digest.

Returns

Type	Description
string	The SHA-384 digest.

SHA512

This function calculates the SHA-512 digest of a value.

Syntax

```
SHA512 (value)
```

Argument	Type	Description
value	string, boolean, int, long, float, double, date, time, dateTime	The value for which to calculate the SHA-512 digest.

Returns

Type	Description
string	The SHA-512 digest.

Informational or Testing Functions

This topic describes the functions in that retrieve information from your values and let you ask questions about them or test whether the values match expectations.

- **CONTAINS**: Evaluates whether the specified string contains the given pattern.
- **ISBLANK**: Evaluates whether the given RDF term is a blank node.
- **ISIRI**: Evaluates whether the given RDF term is an IRI.
- **ISLITERAL**: Evaluates whether the given RDF term is a literal value.
- **ISNUMERIC**: Evaluates whether the given RDF term is a numeric literal value.
- **ISURI**: Evaluates whether the given RDF term is a URI.
- **LANG**: Returns any language tags that are included with strings.
- **LANGMATCHES**: Evaluates whether a string includes a language tag that matches the specified language range.
- **LOCALNAME**: Retrieves the local name from the given URI.
- **NAMESPACE**: Retrieves the namespace for the specified URI.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS:** Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

CONTAINS

This function evaluates whether the specified strings contain the given pattern.

Syntax

```
CONTAINS(text, pattern)
```

Argument	Type	Description
text	string	The string value that you want to check against the specified pattern.
pattern	string	The string pattern that you want to look for in the supplied text.

Returns

Type	Description
boolean	<code>True</code> if the strings contain the pattern and <code>false</code> if they do not.

ISBLANK

This function evaluates whether the given RDF term value is a blank node. It returns `true` if it is a blank node or `false` if it is not.

Syntax

```
ISBLANK(value)
```

Argument	Type	Description
value	RDF term	The literal, URI, or blank node value to test and determine if it is a blank node.

Returns

Type	Description
boolean	<code>True</code> if the term is a blank node and <code>false</code> if it is not.

ISIRI

This function evaluates whether the given RDF term type value is an IRI. It returns `true` if the value is an IRI or `false` if it is not.

Syntax

```
ISIRI (term)
```

Argument	Type	Description
term	RDF term	The literal, URI, or blank node value to evaluate whether it is an IRI.

Returns

Type	Description
boolean	<code>True</code> if the term is an IRI and <code>false</code> if it is not.

ISLITERAL

This function evaluates whether the given RDF term type value is a literal value. It returns `true` if the value is a literal or `false` if it is not.

Syntax

```
ISLITERAL (term)
```

Argument	Type	Description
term	RDF term	The literal, URI, or blank node value to evaluate whether it is a literal.

Returns

Type	Description
boolean	<code>True</code> if the term is a literal value and <code>false</code> if it is not.

ISNUMERIC

This function evaluates whether the given RDF term type value is a numeric literal. It returns `true` if the value is a numeric literal or `false` if it is not.

Syntax

```
ISNUMERIC (term)
```

Argument	Type	Description
term	RDF term	The literal, URI, or blank node value to evaluate whether it is a numeric literal.

Returns

Type	Description
boolean	<code>True</code> if the term is a numeric literal and <code>false</code> if it is not.

ISURI

This function evaluates whether the given value is a URI. It returns `true` if the value is a URI or `false` if it is not.

Syntax

```
ISURI (term)
```

Argument	Type	Description
term	RDF term	The literal, URI, or blank node value to evaluate whether it is a URI.

Returns

Type	Description
boolean	<code>True</code> if the term is a URI and <code>false</code> if it is not.

LANG

This function returns any language tags that are included in the string. The results are grouped by each language tag or by "blank" if a value does not have a language tag.

Syntax

```
LANG (text)
```

Argument	Type	Description
text	string	The string to search for language tags.

Returns

Type	Description
string	The found language tags.

LANGMATCHES

This function tests whether a string includes a language tag that matches the specified language range.

Syntax

```
LANGMATCHES (text, language_range)
```

Argument	Type	Description
text	string	The string to evaluate.
language_range	string	The language tag to match in the <code>text</code> .

Returns

Type	Description
boolean	<code>True</code> if strings include a language tag that matches the range and <code>false</code> if they do not.

LOCALNAME

This function retrieves the local name from the given URI.

Syntax

```
LOCALNAME (uri)
```


Argument	Type	Description
uri	URI	The URI from which to retrieve the local name.

Returns

Type	Description
string	The local name.

NAMESPACE

This function retrieves the namespace for the given URI.

Syntax

```
NAMESPACE(uri)
```

Argument	Type	Description
uri	URI	The URI from which to retrieve the namespace.

Returns

Type	Description
string	The namespace.

Logical Functions

This topic describes the logical functions in Graph Lakehouse.

- **AND**: Evaluates two logical expressions and returns true if both expressions are true.
- **BOUND**: Evaluates whether an RDF term type is bound.
- **CASE**: Evaluates a series of conditions and returns the matching result.

- **COALESCE**: Evaluates a number of expressions and returns the results for the first expression that is bound and does not raise an error.
- **EXISTS**: Evaluates whether the specified pattern exists.
- **IF**: Evaluates a condition and returns the specified result depending on the outcome of the test.
- **IN**: Evaluates whether the specified RDF term is found in any of the given test values.
- **NOT**: Evaluates whether the specified logical expression is not true.
- **OR**: Evaluates two logical expressions and returns true if at least one of the expressions is true.
- **PARTITIONINDEX**: Returns the zero-based index of the bucket in which the specified value falls.
- **SAMETERM**: Evaluates whether two RDF term type values are the same.
- **UNBOUNDED**: Extends the SPARQL UNDEF functionality to enable users to include an undefined value as a function argument.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

AND

This function evaluates two logical expressions. If both expressions are true, the function returns `true`. If one or both arguments are false, the function returns `false`.

Syntax

```
AND(logical_expression1, logical_expression2)
```

Argument	Type	Description
<code>logical_expression1</code>	evaluates to boolean	The first logical expression to evaluate.
<code>logical_expression2</code>	evaluates to boolean	The second logical expression to evaluate.

Returns

Type	Description
boolean	<code>True</code> if both conditions are true and <code>false</code> if either condition is false.

BOUND

This function evaluates whether the specified RDF term has a value bound to it.

Syntax

```
BOUND (term)
```

Argument	Type	Description
<code>term</code>	RDF term	The literal, URI, or blank node value to evaluate.

Returns

Type	Description
boolean	<code>True</code> if the term is bound and <code>false</code> if it is not.

CASE

This function enables you to add IF/THEN logic to a query. A CASE expression evaluates a series of conditions and returns the matching result. You can use CASE expressions wherever expressions are valid in SPARQL queries.

Syntax

There are two variations of CASE statements: simple and generic. Use the simple form to compare the results of an expression with a series of tests and return a result when a test returns `true`. Use the generic form when evaluating a larger range of tests with multiple conditions.

Simple Form

```
CASE expression_to_compare
WHEN expression1 THEN result1
WHEN expression2 THEN result2
[ WHEN expressionN THEN resultN ]
[ ELSE result_when_false ]
END
```

Argument	Type	Description
expression_to_compare	evaluates to boolean	The expression to evaluate and compare its results with the subsequent expressions.
expression1–N	evaluates to boolean	The expressions to evaluate against <code>expression_to_compare</code> .
result1–N	any	The result to return when the corresponding expression is true.
result_when_false	any	An optional value to be returned if none of the specified expressions are true.

Generic Form

```
CASE WHEN condition1 THEN result1
WHEN condition2 THEN result2
[ WHEN conditionN THEN resultN ]
[ ELSE result_when_false ]
END
```

Argument	Type	Description
condition1–N	evaluates to boolean	The conditions to test.
result1–N	any	The result to return when the corresponding condition passes.
result_when_false	any	An optional value to be returned if none of the specified conditions pass.

Returns

Type	Description
The type of the specified result	The specified results according to the evaluation of the conditions.

Example

The following example uses a CASE statement to determine and report on whether the top 10 events (with the most tickets sold) sold out.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event ?venue ?seats
(( CASE WHEN (?seats <= (sum(?qty))) then "yes"
        WHEN (?seats > (sum(?qty))) then "no"
    END ) as ?sold_out)
FROM <http://anzograph.com/tickit>
WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?eventid tickit:venueid ?venueid .
    ?venueid tickit:venueid ?venue .
    ?venueid tickit:venueseats ?seats.
}
GROUP BY ?event ?venue ?seats ?qty
```

```
ORDER BY desc(?qty)
LIMIT 10
```

event	venue	seats	sold_out
Simple Plan	Hubert H. Humphrey Metrodome	64035	no
Black Crowes	Yankee Stadium	52325	no
Hot Tuna	Turner Field	50091	no
Marc Anthony	Georgia Dome	71149	no
Mark Knopfler	Edward Jones Dome	66965	no
Spoon	Dolphin Stadium	74916	no
Armando Manzanero	Texas Stadium	65595	no
Missy Higgins	Great American Ball Park	42059	no
Zombies	Lambeau Field	72922	no
Hannah Montana	Monster Park	69843	no

10 rows

COALESCE

This function evaluates a number of expressions and returns the results for the first expression that is bound and does not raise an error.

Syntax

```
COALESCE(expression1 [, expression2 ] [, expressionN ] )
```

Argument	Type	Description
expression1–N	RDF term	The literal, URI, or blank node expressions to evaluate.

Returns

Type	Description
RDF term	The result of the first expression that is bound and does not error.

EXISTS

This function evaluates whether the specified pattern exists in the data.

Syntax

```
EXISTS { graph_pattern }
```

Returns

Type	Description
boolean	True if the pattern exists and false if it does not.

IF

This function evaluates a condition and returns the specified result depending on the outcome of the test. If the condition evaluates to true, the first result is returned. If the condition evaluates to false, the second result is returned. And if the condition results in an error, the third result is returned.

Syntax

```
IF(logical_expression, true_result, false_result [, error_result ])
```

Argument	Type	Description
logical_expression	evaluates to boolean	The condition that evaluates to true or false.
true_result	RDF term	The value that defines the result to return if the condition evaluates to true.
false_result	RDF term	The value that defines the result to return if the condition evaluates to false.
error_result	RDF term	An optional value that defines the result to return if the condition evaluates to an error. If the condition results in an error and <code>error_result</code> is not specified, <code>logical_expression (error)</code> is returned.

Returns

Type	Description
RDF term	The result based on the evaluation of the condition.

IN

This function evaluates whether the specified RDF term type value is found in any of the given test values.

Syntax

```
IN(term, test_value1 [, test_value2 ] [, test_valueN])
```

Argument	Type	Description
term	RDF term	The literal, URI, or blank node value to look for in the test values.
test_value1–N	RDF term	The literal, URI, or blank node values to look for the specified <code>term</code> in.

Returns

Type	Description
boolean	<code>True</code> if the given term is found in the test values and <code>false</code> if it is not.

Example

The example below queries the sample Tickit data set to return the names of people who were born in the year 1975.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?birthday (concat(?fname, ?lname) AS ?name)
```



```
FROM <http://anzograph.com/tickit>
WHERE {
    ?s tickit:firstname ?fname .
    ?s tickit:lastname ?lname .
    ?s tickit:birthday ?birthday.
    FILTER ((YEAR(?birthday)) IN (1975))
}
ORDER BY ?birthday
```

```
birthday    | name
-----+-----
1975-01-01  | MaryamWeeks
1975-01-01  | OliverHammond
1975-01-01  | MacKenzieBaldwin
1975-01-01  | XenosBaxter
1975-01-01  | XanderWilson
1975-01-01  | HadleyBush
1975-01-01  | RajaTodd
1975-01-02  | AlecFitzgerald
1975-01-02  | QuinnBuckley
1975-01-03  | RhiannonBooth
1975-01-03  | LanaLeonard
1975-01-03  | DeirdreWheeler
...
763 rows
```

NOT

This function evaluates whether the specified logical expression is not true.

Syntax

```
NOT(logical_expression)
```

Argument	Type	Description
logical_expression	evaluates to boolean	The condition to evaluate.

Returns

Type	Description
boolean	<code>True</code> if the condition is false and <code>false</code> if it is true.

OR

This function evaluates two logical expressions. If at least one expression is true, the function returns `true`. If both expressions are false, the function returns `false`.

Syntax

```
OR(logical_expression1, logical_expression2)
```

Argument	Type	Description
<code>logical_expression1</code>	evaluates to boolean	The first logical expression to evaluate.
<code>logical_expression2</code>	evaluates to boolean	The second logical expression to evaluate.

Returns

Type	Description
boolean	<code>True</code> if one or both conditions are true and <code>false</code> if both conditions are false.

PARTITIONINDEX

This function returns the zero-based index of the bucket in which the specified value falls. Buckets start at the specified `start` value and are sized according to the specified `interval`. The first bucket is `[start, start+interval)`. That means it is closed on the low end and open on the high end. `PARTITIONINDEX` returns less than 0 if the value does not fall into any bucket, such as when the given `value` is less than `start` or if the comparison is indeterminate for date and time data types.

Syntax

```
PARTITIONINDEX (value, start, interval)
```

Argument	Type	Description
value	literal	The literal value for which to determine the zero-based index.
start	literal	The literal value that indicates the start of the first bucket.
interval	literal	The literal value that specifies the size of the bucket.

Returns

Type	Description
long	The zero-based index of the bucket in which the specified value exists.

SAMETERM

This function evaluates whether two RDF term type values are the same.

Syntax

```
SAMETERM (term1, term2)
```

Argument	Type	Description
term1	RDF term	The first literal, URI, or blank node value to compare.
term2	RDF term	The literal, URI, or blank node value to compare to <code>term1</code> .

Returns

Type	Description
boolean	<code>True</code> if the terms are the same and <code>false</code> if they are not.

UNBOUNDED

This function is like the SPARQL UNDEF keyword but extends that functionality to enable users to include an undefined value as a function argument, as UNDEF is only supported in VALUES clauses.

Syntax

```
UNBOUNDED ( )
```

Returns

Type	Description
RDF term	The specified result according to the evaluation of the condition.

Example

The following example statement incorporates UNBOUNDED to return null if the specified condition (`?x > 5`) fails:

```
BIND (IF (?x > 5 , "Win", UNBOUNDED ()) as ?testResult)
```

In this case, `?testResult` is bound if `?x` is greater than 5. If `?x` is not greater than 5, `?testResult` is not bound.

Math Functions

This topic describes the mathematical functions in Graph Lakehouse.

- **ABS**: Calculates the absolute value of the specified number.
- **ADD**: Adds two numeric values.
- **AVG**: Calculates the average (arithmetic mean) value for a group of numbers.
- **BASE**: Converts a number to the specified base and returns a text representation.
- **CEIL**: Rounds up a numeric value to the nearest integer.
- **COS**: Calculates the cosine of an angle.
- **EXP**: Raises e to the specified power.
- **FACT**: Calculates the factorial of the specified number.
- **FLOOR**: Rounds down a numeric value to the nearest integer.
- **HAMMING_DIST**: Calculates the hamming distance between two values.
- **HAVERSINE_DIST**: Computes the haversine distance between two latitude and longitude values.
- **LN**: Calculates the natural logarithm of a double value.
- **LOG**: Calculates the specified base logarithm of a double value.
- **LOG2**: Calculates the base two logarithm of a double value.
- **MOD**: Calculates the modulo of the division between two numbers.
- **PI**: Returns the value for PI.
- **POWER**: Raises the specified number to the specified power.
- **RADIANS**: Converts to radians an angle value that is in degrees.
- **RAND**: Returns a random double value between 0 and 1.
- **RANDBETWEEN**: Returns a random integer that falls between two specified integers.
- **ROUND**: Rounds a numeric value to the nearest integer.
- **ROUNDDOWN**: Rounds a numeric value down to the specified number of digits.
- **ROUNDUP**: Rounds a numeric value up to the specified number of digits.

- **SIN**: Calculates the sine of an angle.
- **SQRT**: Calculates the square root of a number.
- **TAN**: Calculates the tangent of an angle.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

ABS

This function calculates the absolute value of the specified number.

Syntax

```
ABS (number)
```

Argument	Type	Description
number	numeric	The numeric value for which to calculate the absolute value.

Returns

Type	Description
number	The absolute value.

Example

The following example queries the sample Tickit data to find the absolute value of the price per ticket minus the total price paid for each of the ticket listings.

```

PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?listing (ABS(?priceper - ?total) AS ?absolute_value)
FROM <http://anzograph.com/tickit>
WHERE {
    ?listing tickit:priceperticket ?priceper .
    ?listing tickit:totalprice ?total .
}
ORDER BY ?listing
LIMIT 10

```

listing	absolute_value
http://anzograph.com/tickit/listing1	1638
http://anzograph.com/tickit/listing10	2955
http://anzograph.com/tickit/listing100	3059
http://anzograph.com/tickit/listing1000	928
http://anzograph.com/tickit/listing10000	1350
http://anzograph.com/tickit/listing100001	410
http://anzograph.com/tickit/listing100002	5502
http://anzograph.com/tickit/listing100003	3146
http://anzograph.com/tickit/listing100004	368
http://anzograph.com/tickit/listing100006	6960

10 rows

ADD

This function adds two numeric values.

Syntax

```
ADD(value1, value2)
```

Argument	Type	Description
value1	numeric	The first numeric value to add.
value2	numeric	The second numeric value to add.

Returns

Type	Description
number	The result of the addition operation.

AVG

This function calculates the average (arithmetic mean) value for a group of numbers.

Syntax

```
AVG (number)
```

Argument	Type	Description
number	numeric	The numeric value for which to calculate the average.

Returns

Type	Description
number	The arithmetic mean of the input values.

Examples

The following example queries the sample Tickit data set to determine the average number of seats in the venues in each state. Since the results clause contains a non-aggregated variable (?state), a GROUP BY clause is required for grouping on ?state.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?state (ROUND(AVG(?seats)) AS ?avg_seats)
FROM <http://anzograph.com/tickit>
WHERE {
  ?s tickit:venuestate ?state .
  ?s tickit:venueSeats ?seats .
}
```



```
GROUP BY ?state
ORDER BY ?state
```

```
state | avg_seats
-----+-----
CA    |      50309
CO    |      63285
DC    |      41888
FL    |      62603
GA    |      60620
IL    |      48244
IN    |      63000
LA    |      72000
MA    |      54342
MD    |      70229
MI    |      53391
MN    |      64035
MO    |      59217
NC    |      73298
NJ    |      80242
NY    |      48764
OH    |      56035
ON    |      50516
PA    |      53931
TN    |      68804
TX    |      56915
WA    |      57058
WI    |      57561
23 rows
```

The query below calculates the average total price for all of the listings in the sample Ticket data set:

```
PREFIX tickit: <http://anzograph.com/ticket/>
SELECT (AVG(?numtickets*?priceperticket) AS ?avg_total_price)
FROM <http://anzograph.com/ticket>
WHERE {
    ?listing tickit:priceperticket ?priceperticket .
    ?listing tickit:numtickets ?numtickets .
}
```

```
avg_total_price
-----
```

```
3034.42
```

```
1 rows
```

BASE

This function converts a number into the specified base and returns a text representation of the calculated value.

Syntax

```
BASE(number, base [, min_length ])
```

Argument	Type	Description
number	int	The integer to convert. Valid values are 0–2 ⁵³ .
base	int	The radix to convert the <code>number</code> to. Valid values are 2–36.
min_length	int	Optional argument that specifies the minimum length of the returned string. If the result is shorter than the minimum length, leading zeros are added to the result so that it reaches the minimum length. Valid values are 0–255.

Returns

Type	Description
string	The text representation of the base.

CEIL

This function rounds up a numeric value to the nearest integer if the value has a fractional part. CEILING returns the value itself if it is a whole number.

Syntax

```
CEIL(number)
```

Argument	Type	Description
number	numeric	The numeric value to round up.

Returns

Type	Description
number	The rounded up value.

COS

This function calculates the cosine of the specified angle.

Syntax

```
COS (angle)
```

Argument	Type	Description
angle	double	The angle in radians (double data type) to calculate the cosine for. If you have angle values in degrees, you can use RADIANS to convert the degrees to radians.

Returns

Type	Description
double	The cosine of the angle.

EXP

This function raises the base of the natural logarithms, e, to the specified power.

Syntax

```
EXP (power)
```

Argument	Type	Description
power	double	The number to raise e to.

Returns

Type	Description
double	E raised to the specified power.

FACT

This function calculates the factorial of the specified number.

Syntax

```
FACT (number)
```

Argument	Type	Description
number	int	The number for which to calculate the factorial.

Returns

Type	Description
int	The factorial of the input values.

FLOOR

This function rounds down a numeric value to the nearest integer if the value has a fractional part. FLOOR returns the value itself if it is a whole number.

Syntax

```
FLOOR (number)
```

Argument	Type	Description
number	numeric	The numeric value to round down.

Returns

Type	Description
number	The rounded down value.

HAMMING_DIST

This function calculates the hamming distance between two values.

Syntax

```
HAMMING_DIST(number1, number2)
```

Argument	Type	Description
number1	long	The first number.
number2	long	The second number.

Returns

Type	Description
int	The hamming distance.

HAVERSINE_DIST

This function computes the haversine distance between two latitude and longitude values and returns the distance in kilometers.

Syntax

```
HAVERSINE_DIST(latitude1, longitude1, latitude2, longitude2)
```

Argument	Type	Description
latitude1	double	The first latitude value.
longitude1	double	The first longitude value.
latitude2	double	The second latitude value.
longitude2	double	The second longitude value.

Returns

Type	Description
double	The distance in kilometers.

LN

This function calculates the natural logarithm of a double value.

Syntax

```
LN (number)
```

Argument	Type	Description
number	double	The double value for which to calculate the natural logarithm.

Returns

Type	Description
double	The natural logarithm of the input value.

LOG

This function calculates the specified base logarithm of a double value.

Syntax

```
LOG (number [, base ])
```

Argument	Type	Description
number	double	The double value for which to calculate the <code>base</code> logarithm.
base	double	An optional double value that specifies the base for the logarithm. If omitted, base e is used.

Returns

Type	Description
double	The base logarithm of the input value.

LOG2

This function calculates the base two logarithm of a double value.

Syntax

```
LOG2 (number)
```

Argument	Type	Description
number	double, float	The double value for which to calculate the base 2 logarithm.

Returns

Type	Description
double, float	The base two logarithm of the input value.

Example

The example below determines the base two logarithm of the quantity of tickets sold for each ticket listing.

```
SELECT ?sale ?qtysold (LOG2(?qtysold) AS ?qtylog)
FROM <http://anzograph.com/tickit>
WHERE {
    ?sale <http://anzograph.com/tickit/qtysold> ?qtysold .
}
ORDER BY ?sale
LIMIT 10
```

sale	qtysold	qtylog
http://anzograph.com/tickit/sales1	4	2
http://anzograph.com/tickit/sales10	1	0
http://anzograph.com/tickit/sales100	2	1
http://anzograph.com/tickit/sales1000	3	1.58496
http://anzograph.com/tickit/sales10000	1	0
http://anzograph.com/tickit/sales100000	1	0
http://anzograph.com/tickit/sales100001	2	1
http://anzograph.com/tickit/sales100002	1	0
http://anzograph.com/tickit/sales100003	2	1
http://anzograph.com/tickit/sales100004	2	1

10 rows

MOD

This function calculates the modulo or remainder of the division between two numbers.

Note

The calculation of negative operands depends on C++ and your underlying hardware. Graph Lakehouse uses FMOD for floating point operands and % for all other data types.

Syntax

```
MOD(number, divisor)
```

Argument	Type	Description
number	numeric	The number that is the dividend in the equation.
divisor	numeric	The number to divide the dividend by.

Returns

Type	Description
number	The modulo between the input numbers.

Example

The following example queries the sample Tickit dataset to find the modulo between the number of seats in each venue and the population of the city the venue is in.

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?venue ?city (MOD(?pop, ?seats) AS ?modulo)
FROM <http://anzograph.com/tickit>
WHERE {
  ?s tickit:venueName ?venue .
  ?s tickit:venueCityPop ?pop .
  ?s tickit:venueCity ?city .
  ?s tickit:venueSeats ?seats .
}
ORDER BY ?venue
LIMIT 10
```

venue	city	modulo
ARCO Arena	Sacramento	638
AT&T Park	San Francisco	16678
Angel Stadium of Anaheim	Anaheim	26011
Arrowhead Stadium	Kansas City	62532
Bank of America Stadium	Charlotte	71742
Busch Stadium	St. Louis	20109
Citizens Bank Park	Philadelphia	42008
Cleveland Browns Stadium	Cleveland	30815
Comerica Park	Detroit	3483
Coors Field	Denver	45263
10 rows		

PI

This function returns the value for PI.

Syntax

```
PI ()
```

Returns

Type	Description
double	The PI value.

POWER

This function raises the specified number to the specified power.

Syntax

```
POWER (value, power)
```

Argument	Type	Description
value	numeric	The number to raise by the power.

Argument	Type	Description
power	numeric	The number to raise <code>value</code> by.

Returns

Type	Description
number	The result of <code>value</code> raised to the specified <code>power</code> .

Example

The following example queries the sample Tickit dataset to raise the total number of tickets sold for each event by the power of 2:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event (POWER(?tickets, 2) AS ?power_total)
FROM <http://anzograph.com/tickit>
WHERE {
  SELECT ?event (sum(?qty) as ?tickets)
  WHERE {
    ?sales tickit:qtysold ?qty .
    ?sales tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
  }
}
GROUP BY ?event
ORDER BY desc(?tickets)
LIMIT 10
}
```

event	power_total
Mamma Mia!	1.3381e+07
Spring Awakening	9.15062e+06
The Country Girl	8.24264e+06
Jersey Boys	7.73396e+06
Macbeth	7.46929e+06
Chicago	6.42622e+06
Legally Blonde	5.16198e+06
Spamalot	4.8356e+06

Rhinoceros		3.6481e+06
Thurgood		3.58724e+06
10 rows		

RADIANS

This function converts to radians an angle value that is in degrees.

Syntax

RADIANS (angle)

Argument	Type	Description
angle	double	The angle value to convert to radians.

Returns

Type	Description
double	The angle in radians.

RAND

This function returns a random double value between 0 and 1, including 0 and excluding 1.

Syntax

RAND ()

Returns

Type	Description
double	The random value between 0 and 1.

RANDBETWEEN

This function returns a random integer that falls between the two specified integers. The two integers are included as options to be returned.

Syntax

```
RANDBETWEEN(low_number, high_number)
```

Argument	Type	Description
low_number	int	The lowest integer in the range of values.
high_number	int	The highest integer in the range of values.

Returns

Type	Description
int	The random value between the given low and high numbers.

ROUND

This function rounds a numeric value to the nearest integer.

Syntax

```
ROUND(number)
```

Argument	Type	Description
number	numeric	The number to round to the nearest integer.

Returns

Type	Description
number	The rounded value.

ROUNDDOWN

This function rounds a numeric value down to the specified number of digits.

Syntax

```
ROUNDDOWN(number, num_digits)
```

Argument	Type	Description
number	numeric	The number to round down.
num_digits	int	An integer that specifies the number of digits to round down to.

Returns

Type	Description
number	The rounded down value.

ROUNDUP

This function rounds a numeric value up to the specified number of digits.

Syntax

```
ROUNDUP(number, num_digits)
```

Argument	Type	Description
number	numeric	The number to round up.
num_digits	int	An integer that specifies the number of digits to round up to.

Returns

Type	Description
number	The rounded up value.

SIN

This function calculates the sine of the specified angle.

Syntax

```
SIN(angle)
```

Argument	Type	Description
angle	double	The angle in radians to calculate the sine for. If you have angle values in degrees, you can use RADIANS to convert the degrees to radians.

Returns

Type	Description
double	The sine of the angle.

SQRT

This function calculates the square root of the specified number.

Syntax

```
SQRT (number)
```

Argument	Type	Description
number	numeric	The number for which to calculate the square root.

Returns

Type	Description
double	The square root of the input value.

TAN

This function calculates the tangent of the specified angle.

Syntax

```
TAN (angle)
```

Argument	Type	Description
angle	double	The angle in radians to calculate the tangent for. If you have angle values in degrees, you can use RADIANS to convert the degrees to radians.

Returns

Type	Description
double	The tangent of the angle.

Property Paths

SPARQL property paths enable users to examine the patterns between properties in the data. Property paths reveal the routes between nodes in a graph.

Syntax

Specify property paths in the predicate part of a triple pattern. Combine predicates using the operators described in the table below. For more information, see the [W3C Property Path](#) specification.

Construct	Expression Name	Description
<URI>	PredicatePath	A predicate URI in a triple pattern matches a path length of one.
^path1	InversePath	Matches on backwards paths--subject to object.
path1/path2	SequencePath	Matches on forward paths--path1 followed by path2.
path1 path2	AlternativePath	Matches on either path1 or path2. Graph Lakehouse finds all possibilities.
path1*	ZeroOrMorePath	Connects the subject and object of the path by zero or more matches of path1, i.e., path1 repeated zero or more times.
path1+	OneOrMorePath	Connects the subject and object of the path by one or more matches of path1, i.e., path1 repeated one or more times.
path1?	ZeroOrOnePath	Connects the subject and object of the path by zero or one matches of path1, i.e., path1 is optional.

Construct	Expression Name	Description
(path)	N/A	Specifies groups of paths. Use parentheses around groups to control precedence.
!URI or !(URI1 ... URIn)	NegatedPropertySet	A negated property path where matches are excluded. The order of URIs is not significant.
!^URI or !(^URI1 ... ^URIn)	NegatedPropertySet	Negated property path where the excluded matches are based on a reversed path. The order of URIs is not significant. You can include a combination of forward and reverse properties in a negated property set: !(URI1 ... URIj ^URIj+1 ... ^URIn)

String Functions

This topic describes the Graph Lakehouse functions that operate on string data types.

- **CONCAT**: Concatenates a list of strings.
- **CONTAINS**: Evaluates whether the specified string contains the given pattern.
- **ENCODE_FOR_URI**: Encodes the specified string as a URI.
- **ESCAPEHTML**: Escapes the specified string for use in HTML.
- **FIND**: Returns the position—from left to right—of a string within another string.
- **FINDREVERSE**: Returns the position—from right to left—of a string within another string.
- **GROUP_CONCAT**: Concatenates a group of strings into a single string.
- **LANG**: Returns any language tags that are included with strings.
- **LANGMATCHES**: Evaluates whether a string includes a language tag that matches the specified language range.

- **LCASE**: Converts the letters in a string to lower case.
- **LEFT**: Returns the specified number of characters starting from the beginning (left side) of the string.
- **LEVENSHTEIN_DIST**: Calculates the Levenshtein distance or measure of similarity between two strings.
- **LTRIM_WS**: Trims white space from the left side of a string.
- **REGEX**: Evaluates whether a string matches the specified regular expression pattern.
- **REGEXP_SUBSTR**: Searches a string for the specified regular expression pattern and returns the substring that matches the pattern.
- **REPLACE**: Extends the REGEX function to provide the ability to find a pattern in a string and replace it with another pattern.
- **RIGHT**: Returns the specified number of characters starting from the end (right side) of the string.
- **RPAD**: Pads the right side of a string with the specified number of spaces.
- **RTRIM_WS**: Trims white space from the right side of a string
- **STRAFTER**: Returns the portion of a string that comes after the specified substring.
- **STRBEFORE**: Returns the portion of a string that comes before the specified substring.
- **STREND**: Evaluates whether the specified string ends with the specified substring.
- **STRLANG**: Constructs a literal value with the specified language tag.
- **STRLEN**: Returns the number of characters in the specified string.
- **STRSTARTS**: Evaluates whether the specified string starts with the specified substring.
- **STRUUID**: Returns a string that is the result of generating a Universally Unique Identifier (UUID).
- **SUBSTR**: Returns a substring from a string value.
- **TRIM**: Removes all spaces from a string except for any single spaces between words.

- **UCASE**: Converts the letters in a string to upper case.
- **URI**: Casts a string to a URI.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

CONCAT

This function concatenates two or more strings and returns the result as a string.

Syntax

```
CONCAT(text1, text2 [, textN ])
```

Argument	Type	Description
text1–N	string	The strings that you want to concatenate to form a single string.

Returns

Type	Description
string	The concatenated string.

CONTAINS

This function evaluates whether the specified strings contain the given pattern.

Syntax

```
CONTAINS(text, pattern)
```

Argument	Type	Description
text	string	The string value that you want to check against the specified pattern.
pattern	string	The string pattern that you want to look for in the supplied text.

Returns

Type	Description
boolean	<code>True</code> if the strings contain the pattern and <code>false</code> if they do not.

ENCODE_FOR_URI

This function encodes the specified string as a URI and returns a string in URI format.

Syntax

```
ENCODE_FOR_URI (text)
```

Argument	Type	Description
text	string	The string value to encode as a URI.

Returns

Type	Description
string	The string as a URI.

Example

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT DISTINCT (ENCODE_FOR_URI(?eventname) as ?event)
FROM <http://anzograph.com/tickit>
WHERE {
```

```

?s tickit:eventid ?eventid .
?eventid tickit:eventname ?eventname .
}
ORDER BY ?event
LIMIT 10

```

```

event
-----
.38%20Special
3%20Doors%20Down
70s%20Soul%20Jam
A%20Bronx%20Tale
A%20Catered%20Affair
A%20Chorus%20Line
A%20Christmas%20Carol
A%20Doll%27s%20House
A%20Man%20For%20All%20Seasons
A%20Midsummer%20Night%27s%20Dream
10 rows

```

ESCAPEHTML

This function escapes the specified string for use in HTML.

Syntax

```
ESCAPEHTML (text)
```

Argument	Type	Description
text	string	The string value to escape for HTML.

Returns

Type	Description
string	The string escaped for HTML.

FIND

This function returns the position—from left to right—of a string within another string.

Tip

You can use [FINDREVERSE](#) to find the character or substring position from right to left.

Syntax

```
FIND(find_text, within_text, start_num)
```

Argument	Type	Description
find_text	string	The string to look for in the <code>within_text</code> .
within_text	string	The string to search within.
start_num	int	An integer that indicates the position to start from when looking for the <code>find_text</code> . The starting position is at the beginning of the <code>within_text</code> value and characters are counted from left to right.

Returns

Type	Description
int	The character position (from left to right) where the substring starts.

FINDREVERSE

Similar to [FIND](#), this function returns the position—from right to left—of a string within another string.

Syntax

```
FINDREVERSE(find_text, within_text, start_num)
```

Argument	Type	Description
find_text	string	The string to look for in the <code>within_text</code> value.

Argument	Type	Description
within_text	string	The string to search within.
start_num	int	An integer that indicates the position to start from when looking for the <code>find_text</code> . The starting position is the end of the <code>within_text</code> value and characters are counted from right to left.

Returns

Type	Description
int	The character position (from right to left) where the substring starts.

GROUP_CONCAT

This function concatenates a group of strings into a single string.

Syntax

```
GROUP_CONCAT (group ; [ SEPARATOR = "separator_char" ] ; [ ROW_LIMIT = max_rows ] ;
[ PRE = "prefix" ] ; [ VALUE_SERIALIZE = serialize ] ; [ DELIMIT_BLANKS = separate_
blanks ] ;
[ MAX_LENGTH = string_length ] ; [ SUFFIX = "suffix" ] )
```

Argument	Type	Description
group	string	The group of strings to concatenate.
separator_char	string	Optional argument that defines the separator to use between the values in returned strings. When SEPARATOR is omitted, Graph Lakehouse separates values with a space.
max_rows	int	Optional argument that puts a maximum limit on the number of rows to retrieve for the group. When ROW_LIMIT is omitted, the default is <code>unlimited</code> . Note that Graph Lakehouse performs the

Argument	Type	Description
		GROUP_CONCAT for each slice separately and combines the results from each slice. The ROW_LIMIT is applied to each slice, not the total result. Therefore, the total number of values that are concatenated will be larger than the specified limit, proportional to the number of slices in the cluster.
prefix	string	Optional string to add as a prefix to the resulting string.
serialize	boolean	Optional argument that indicates whether returned values should be serialized with the value's data type. When VALUE_SERIALIZE is omitted, the default is <code>false</code> .
separate_blanks	boolean	Optional argument that indicates whether to delimit blanks with the SEPARATOR value. When DELIMIT_BLANKS is omitted, the default is <code>false</code> .
string_length	int	Optional argument that limits the resulting strings to a maximum character length. Graph Lakehouse has a 2MB (~2,000,000 characters) limit on the length of strings and displays an error if GROUP_CONCAT returns a string that is longer than 2000000. When MAX_LENGTH is omitted, the default is <code>unlimited</code> .
suffix	string	Optional argument that defines a suffix to add to the resulting strings. When SUFFIX is omitted, Graph Lakehouse adds an empty string as the suffix.

Returns

Type	Description
string	The concatenated string.

Example

The query below concatenates the list of friends for 10 people in the sample Tickit data set. Since the GROUP_CONCAT expression includes ROW_LIMIT=2, Graph Lakehouse limits the records to two for each slice (or shard) of data.

```
SELECT ?person (GROUP_CONCAT(?id;SEPARATOR=",";ROW_LIMIT=2) AS ?friends)
FROM <http://anzograph.com/tickit>
WHERE {
    ?person <http://anzograph.com/tickit/friend> ?friend .
    BIND(STRATER(STR(?friend), "http://anzograph.com/tickit/") as ?id)
}
GROUP BY ?person
ORDER BY ?person
LIMIT 10
```

person	friends
http://anzograph.com/tickit/person1	person2894, person20624, person33618, person47127
http://anzograph.com/tickit/person10	person3136, person22714, person2509, person24535
http://anzograph.com/tickit/person100	person42775, person29725, person27334, person24553
http://anzograph.com/tickit/person1000	person19040, person39066, person2236, person9089
http://anzograph.com/tickit/person10000	person43706, person37085, person18874, person31270
http://anzograph.com/tickit/person10001	person3389, person44830, person4720, person307
http://anzograph.com/tickit/person10002	person46462, person43989, person46491, person31130
http://anzograph.com/tickit/person10003	person31544, person19595, person23460, person28465
http://anzograph.com/tickit/person10004	person11070, person19845, person11172, person24252
http://anzograph.com/tickit/person10005	person33888, person9467, person35761, person47709

10 rows

LANG

This function returns any language tags that are included in the string. The results are grouped by each language tag or by "blank" if a value does not have a language tag.

Syntax

```
LANG (text)
```

Argument	Type	Description
text	string	The string to search for language tags.

Returns

Type	Description
string	The found language tags.

LANGMATCHES

This function tests whether a string includes a language tag that matches the specified language range.

Syntax

```
LANGMATCHES (text, language_range)
```

Argument	Type	Description
text	string	The string to evaluate.
language_range	string	The language tag to match in the <code>text</code> .

Returns

Type	Description
boolean	<code>True</code> if strings include a language tag that matches the range and <code>false</code> if they do not.

LCASE

This function converts the letters in a string literal to lower case.

Tip

To convert the characters in a string according to a specific locale, you can use the [LCASE](#) utility extension.

Syntax

```
LCASE (text)
```

Argument	Type	Description
text	string	The string literal to convert to lower case.

Returns

Type	Description
string	The string with lower case letters.

LEFT

This function returns the specified number of characters starting from the beginning (left side) of the string.

Syntax

```
LEFT (text, num_chars)
```

Argument	Type	Description
text	string	The string from which to return the specified number of characters.
num_chars	int	An integer that specifies the number of characters to return, starting

Argument	Type	Description
		from the left side of the <code>text</code> .

Returns

Type	Description
string	The specified number of characters from the string.

LEVENSHTEIN_DIST

This function calculates the Levenshtein distance or measure of similarity between two strings. The distance is the smallest number of insertions, deletions, and/or substitutions required to transform the first string into the second string.

Syntax

```
LEVENSHTEIN_DIST(text1, text2)
```

Argument	Type	Description
text1	string	The string that would be transformed into <code>text2</code> .
text2	string	The string to measure <code>text1</code> against.

Returns

Type	Description
int	The Levenshtein distance between the strings.

LTRIM_WS

This function removes all spaces from the left side of a string.

Syntax

```
LTRIM_WS(text)
```

Argument	Type	Description
text	string	The string to trim.

Returns

Type	Description
string	The string with spaces removed.

REGEX

This function tests whether a string matches the specified regular expression pattern.

Syntax

```
REGEX(text, pattern [, flags ])
```

Argument	Type	Description
text	string	The string to test against the <code>pattern</code> .
pattern	string	The regular expression pattern to look for in the <code>text</code> . For information about the supported regular expression syntax, see the Regular Expression Syntax section of the W3C XQuery 1.0 and XPath 2.0 Functions and Operators specification.
flags	string	You can include one or more optional modifier flags that further define the pattern. For information about flags, see the Flags section of the W3C Functions and Operators specification.

Returns

Type	Description
boolean	<code>True</code> if the string matches the regular expression pattern and <code>false</code> if it does not.

REGEXP_SUBSTR

This function searches a string for the specified regular expression pattern and returns the substring that matches the pattern.

Syntax

```
REGEXP_SUBSTR(text, pattern [, start_position ] [, nth_appearance ])
```

Argument	Type	Description
text	string	The string to test against the <code>pattern</code> .
pattern	string	The regular expression pattern to look for in the <code>text</code> . For information about the supported regular expression syntax, see the Regular Expression Syntax section of the W3C XQuery 1.0 and XPath 2.0 Functions and Operators specification.
start_position	int	An optional integer that specifies the number of characters from the beginning of the string to start searching for matches (the default value is 1).
nth_appearance	int	An optional integer that specifies which occurrence of the pattern to match (the default value is 1).

Returns

Type	Description
string	The substring that matches the regular expression pattern.

REPLACE

This function extends the REGEX function to provide the ability to find a pattern in a string and replace it with another pattern. The function returns the replaced string.

Syntax

```
REPLACE(text, pattern, replacement_pattern [, flags ])
```

Argument	Type	Description
text	string	The string to test against the <code>pattern</code> .
pattern	string	The regular expression pattern to look for in the <code>text</code> . For information about the supported regular expression syntax, see the Regular Expression Syntax section of the W3C XQuery 1.0 and XPath 2.0 Functions and Operators specification.
replacement_pattern	string	The pattern to replace the <code>pattern</code> with.
flags	string	You can include one or more optional modifier flags that further define the pattern. For information about flags, see the Flags section of the W3C Functions and Operators specification.

Returns

Type	Description
string	The string that contains the replacement pattern.

RIGHT

This function returns the specified number of characters starting from the end (right side) of the string.

Syntax

```
RIGHT(text, num_chars)
```

Argument	Type	Description
text	string	The string from which to return the specified number of characters.
num_chars	int	An integer that specifies the number of characters to return, starting from the right side of the <code>text</code> .

Returns

Type	Description
string	The specified characters from the string.

RPAD

This function pads the end (right side) of a string with the number of spaces that you specify.

Syntax

```
RPAD(text, num_spaces)
```

Argument	Type	Description
text	string	The string to add the spaces to.
num_spaces	int	An integer that specifies the number of spaces to add to the end of the <code>text</code> .

Returns

Type	Description
string	The value with the specified number of spaces.

RTRIM_WS

This function removes all spaces from the right side of a string.

Syntax

```
RTRIM_WS(text)
```

Argument	Type	Description
text	string	The string to trim.

Returns

Type	Description
string	The string with spaces removed.

STRAFTER

This function returns the portion of a string that comes after the specified substring.

Syntax

```
STRAFTER(text, substring)
```

Argument	Type	Description
text	string	The string from which to return the characters that follow the substring.
substring	string	The string to match in the <code>text</code> . The function will return the part of the text that comes after this substring.

Returns

Type	Description
string	The part of the string that comes after the substring.

Example

The following example query uses STRAFTER to return only the unique portion of each event ID in the sample Tickit data set. The query uses BIND to convert the event URIs to strings and bind them to the `?str_event` variable.

```
SELECT (STRAFTER(?str_event, "event") AS ?event_number) ?name
FROM <http://anzograph.com/tickit>
WHERE {
    ?event <http://anzograph.com/tickit/eventname> ?name .
    BIND (STR(?event) AS ?str_event)
}
ORDER BY ?event_number
```

```
event_number | name
-----+-----
1            | Gotterdammerung
10           | Rigoletto
100          | Siegfried
```

```
1000      | Gypsy
1001      | Chicago
1002      | The King and I
1003      | Pal Joey
1004      | Grease
...
8798 rows
```

STRBEFORE

This function returns the portion of a string that comes before the specified substring.

Syntax

```
STRAFTER(text, substring)
```

Argument	Type	Description
text	string	The string from which to return the characters that precede the <code>substring</code> .
substring	string	The string to match in the <code>text</code> . The function will return the part of the text that comes before this substring.

Returns

Type	Description
string	The part of the string that comes before the substring.

STREND

This function evaluates whether the specified string ends with the specified substring.

Syntax

```
STREND(text, substring)
```

Argument	Type	Description
text	string	The string to search for the <code>substring</code> .
substring	string	The string to match at the end of <code>text</code> . The function returns <code>true</code> if the text ends in the specified substring and <code>false</code> if it does not.

Returns

Type	Description
boolean	<code>True</code> if strings end with the specified substring and <code>false</code> if they do not.

STRLANG

This function constructs a literal value with the specified language tag.

Syntax

```
STRLANG(text, language_tag)
```

Argument	Type	Description
text	string	The string to add the language tag to.
language_tag	string	The language tag to add to the <code>text</code> .

Returns

Type	Description
string	The literal value with the language tag.

STRLEN

This function calculates the length (in characters) of a string value.

Syntax

```
STRLEN (text)
```

Argument	Type	Description
text	string	The string for which to return the length.

Returns

Type	Description
long	The number of characters in the string.

STRSTARTS

This function evaluates whether the specified string starts with the specified substring.

Syntax

```
STREND (text, substring)
```

Argument	Type	Description
text	string	The string to search for the <code>substring</code> .
substring	string	The string to match at the beginning of <code>text</code> . The function returns <code>true</code> if the text starts with the specified substring and <code>false</code> if it does not.

Returns

Type	Description
boolean	<code>True</code> if strings begin with the specified substring and <code>false</code> if they do not.

STRUUID

This function returns a string that is the result of generating a Universally Unique Identifier (UUID).

Syntax

```
STRUUID()
```

Returns

Type	Description
string	The UUID.

SUBSTR

This function returns a substring from a string value.

Syntax

```
SUBSTR(text, start [, length ])
```

Argument	Type	Description
text	string	The string to find the substring in.
start	int	An integer that specifies the number of the character in the <code>text</code> that should be the start of the substring.
length	int	An optional integer that specifies the total number of characters to include in the substring. If not specified, the substring will end at the end of the <code>text</code> value.

Returns

Type	Description
string	The substring.

TRIM

This function removes all spaces from a string except for any single spaces between words.

Syntax

```
TRIM(text)
```

Argument	Type	Description
text	string	The string to trim.

Returns

Type	Description
string	The string with spaces removed.

UCASE

This function converts all letters in a string to upper case.

Tip

To convert the characters in a string according to a specific locale, you can use the [UCASE](#) utility extension.

Syntax

```
UPPER(text)
```


Argument	Type	Description
text	string	The string value to convert to upper case.

Returns

Type	Description
string	The string with upper case characters.

URI

This function casts the specified string to a URI.

Syntax

```
URI (value)
```

Argument	Type	Description
value	string	The value to convert to a URI.

Returns

Type	Description
URI	The value as a URI.

Update Functions

This topic describes the SPARQL functions that are used to load, insert, or update data.

- **CLEAR**: Deletes all of the triples in a graph without deleting the graph.
- **COPY**: Copies graph data from the database to disk.
- **CREATE**: Creates a new empty graph.

- **DELETE and DELETE DATA:** Deletes the specified graph or triple patterns or specific triples from the database.
- **DROP:** Deletes a graph and all of its triples.
- **INSERT and INSERT DATA:** Inserts the specified graph or triple patterns or specific triples to the database.
- **LOAD:** Loads data to the database from RDF files that are on the Graph Lakehouse file system. For information, see [Load RDF Data from Files](#).

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS:** Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

CLEAR

The CLEAR function deletes all of the triples in a graph without deleting the graph.

Syntax

```
CLEAR [ SILENT ] GRAPH <graph_URI> | DEFAULT | NAMED | ALL
```

The optional SILENT keyword tells Graph Lakehouse not to return an error if an error occurs.

Statement	Description
CLEAR GRAPH <graph_URI>	Deletes all of the triples from the named graph.
CLEAR DEFAULT	Deletes all of the triples from the default graph.
CLEAR NAMED	Deletes all of the triples from all of the named graphs in the

Statement	Description
	database.
CLEAR ALL	Deletes all of the triples from all graphs.

COPY

In Graph Lakehouse, the COPY operation is used to copy graph data from the database to files on disk. For information on using the Graph Lakehouse COPY command, see [Copy Graphs to Files](#).

CREATE

The CREATE function creates a new empty graph.

Syntax

```
CREATE [ SILENT ] GRAPH <graph_uri>
```

The optional SILENT keyword tells Graph Lakehouse not to return an error if an error occurs.

DELETE and DELETE DATA

The DELETE function deletes the specified graph or triple patterns from the database. The DELETE DATA function deletes specific triples from the database. DELETE DATA statements cannot include variables.

DELETE Syntax

Use the following syntax to delete graph and triple patterns with the DELETE function.

```
DELETE { graph_and_triple_patterns }
WHERE { graph_and_triple_patterns }
```

DELETE DATA Syntax

Use the following syntax to delete specific triples with the DELETE DATA function.

```
DELETE DATA { [ GRAPH <graph_uri> { } triples } [ ] }
```

The optional GRAPH statement specifies the graph to delete the *triples* from. The triples that you list must include URIs, literal, values, or blank nodes. You cannot specify triple patterns with variables. For example, the query below uses DELETE DATA to remove the person0 triples from the tickit graph:

```
DELETE { GRAPH <http://anzograph.com/tickit> {
  <person0> <http://anzograph.com/tickit/firstname> "Jay" .
  <person0> <http://anzograph.com/tickit/lastname> "Stevens" .
  <person0> <http://anzograph.com/tickit/state> "CA" .
}
```

DROP

The DROP function deletes a graph and all of its triples.

Syntax

Use the following syntax to delete graphs and their triples using the DROP function.

```
DROP [ SILENT ] GRAPH <graph_uri> | DEFAULT | NAMED | ALL
```

The optional SILENT keyword tells Graph Lakehouse not to return an error if an error occurs.

Option	Description
DROP GRAPH <graph_URI>	Deletes the named graph.
DROP DEFAULT	Since a graph database must always have a default graph, the DROP DEFAULT operation deletes the triples from the default graph; it does not remove the graph. DROP DEFAULT is synonymous with CLEAR DEFAULT.
DROP NAMED	Deletes all of the named graphs in the database.
DROP ALL	Deletes all of the graphs and their triples from the database. Since the default graph cannot be removed, the triples from the default graph are

Option	Description
	deleted but the default graph remains.

INSERT and INSERT DATA

The INSERT function inserts the specified graph or triple patterns into the database. The INSERT DATA function inserts specific triples into the database. INSERT DATA statements cannot include variables.

INSERT Syntax

Use the following syntax to insert data using graph and triple patterns. The syntax below inserts triples into the default graph:

```
INSERT { triple_patterns }
WHERE { triple_patterns }
```

The following syntax inserts triples into a named graph. The WHERE clause specifies the named graph to find triple patterns.

```
INSERT { GRAPH <graph_uri> { triple_patterns } }
WHERE { GRAPH <graph_uri> { triple_patterns } }
```

As an alternative, you can include one or more USING clauses to specify named graphs for the WHERE clause. USING acts like a FROM clause in a SELECT query.

```
INSERT { GRAPH <graph_uri> { triple_patterns } }
USING <graph_uri>
WHERE { triple_patterns }
```

INSERT DATA Syntax

Use the following syntax to insert specific triples with the INSERT DATA function.

```
INSERT DATA { triples }
```

Use the following syntax to insert specific triples into a graph with the INSERT DATA function.

```
INSERT DATA { GRAPH <graph_uri> { triples } }
```

The GRAPH statement specifies the graph to insert the triples in. The triples that you list must include URIs, literal, values, or blank nodes. You cannot specify triple patterns with variables. For example, the query below uses INSERT DATA to add a new user to the sample tickit data set:

```
INSERT DATA { GRAPH <http://anzograph.com/tickit> {  
  <person0> <http://anzograph.com/tickit/firstname> "Jay" .  
  <person0> <http://anzograph.com/tickit/lastname> "Stevens" .  
  <person0> <http://anzograph.com/tickit/state> "CA" .  
}  
}
```

The query below inserts a graph named "friends" using data from the tickit graph.

```
INSERT { GRAPH <friends> {  
  ?person <http://anzograph.com/tickit/friendOf> ?friend .  
}  
USING <http://anzograph.com/tickit>  
WHERE { ?person <http://anzograph.com/tickit/friend> ?friend .  
}
```

Window Aggregate and Ranking Functions

Window aggregate functions enable you to compute aggregate values on a particular partition or window of the result set. Unlike grouped aggregate functions that group the results and return a single value, window aggregates return a value for each row in the specified window. For example, using the grouped aggregate SUM function to add up the total number of tickets sold in a year returns one value: the total number of tickets sold for the year. By using the SUM window aggregate instead, the results could be partitioned by month so that the query returns 12 values: the sum of the number of tickets sold in each month of the year.

This topic describes the window aggregate and ranking functions in Graph Lakehouse:

- **AVG**: Calculates the average value of each group of values.
- **COUNT**: Counts the number of values in each group of values.
- **MAX**: Calculates the maximum value of each group of values.
- **MIN**: Calculates the minimum value of each group of values.

- **NTILE**: Divides the rows in the partition into the specified number of ranked groups and returns the group that each value belongs to.
- **PERCENTILE**: Divides the rows in the partition into 100 ranked groups and returns the group that each value belongs to.
- **PRODUCT**: Calculates the product of each group of values.
- **QUARTILE**: Divides the rows in the partition into four ranked groups and returns the group that each value belongs to.
- **ROW_NUMBER**: Assigns unique numbers to each row in the partition.
- **SUM**: Calculates the sum of each group of values.

Typographical Conventions

The following list describes the conventions used to document function syntax:

- **CAPS**: Although SPARQL is case-insensitive, SPARQL keywords in this section are written in uppercase for readability.
- `[ argument ]`: Brackets indicate an optional argument or keyword.
- `|`: Means OR. Indicates that you can use one or more of the specified options.

AVG

This function calculates the average value of each group of values.

Syntax

```
(AVG(value) OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
# frame clause
  [ [ ROWS ] frame_start |
    [ ROWS ] BETWEEN frame_start AND frame_end
  ]
)
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.
frame clause	The reference point for all window frames is the current row. The optional frame clause further defines the frame by specifying the rows in a partition to combine with the current row. There are two types of window frames: • A fixed frame with two moving endpoints: Each row becomes the current row as the window frame slides forward in the partition. This type of frame is ideal for computing aggregations over moving time frames. • A resizing frame with one anchored endpoint: One row is a fixed endpoint and the frame resizes up (PRECEDING) or down (FOLLOWING). This type of frame is ideal for computing running totals. The frame clause can be one of the following options: <pre>[ROWS] frame_start [ROWS] BETWEEN frame_start AND frame_end</pre> When a frame clause is not included, the window frame is unbounded: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
frame_start	The starting point of the frame. This argument can be one of the following values: <pre>UNBOUNDED PRECEDING positive_int PRECEDING</pre>

Argument	Description
	<pre>CURRENT ROW positive_int FOLLOWING</pre>
frame_end	The end of the frame. This argument can be one of the following values: <pre>positive_int PRECEDING CURRENT ROW positive_int FOLLOWING UNBOUNDED FOLLOWING</pre>

COUNT

This function counts the number of values in each group of values.

Syntax

```
(COUNT(value) OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
# frame clause
  [ [ ROWS ] frame_start |
    [ ROWS ] BETWEEN frame_start AND frame_end
  ]
)
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.

Argument	Description
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.
frame clause	The reference point for all window frames is the current row. The optional frame clause further defines the frame by specifying the rows in a partition to combine with the current row. There are two types of window frames: • A fixed frame with two moving endpoints: Each row becomes the current row as the window frame slides forward in the partition. This type of frame is ideal for computing aggregations over moving time frames. • A resizing frame with one anchored endpoint: One row is a fixed endpoint and the frame resizes up (PRECEDING) or down (FOLLOWING). This type of frame is ideal for computing running totals. The frame clause can be one of the following options: <pre>[ROWS] frame_start [ROWS] BETWEEN frame_start AND frame_end</pre> When a frame clause is not included, the window frame is unbounded: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
frame_start	The starting point of the frame. This argument can be one of the following values: <pre>UNBOUNDED PRECEDING positive_int PRECEDING CURRENT ROW positive_int FOLLOWING</pre>
frame_end	The end of the frame. This argument can be one of the following values: <pre>positive_int PRECEDING CURRENT ROW positive_int FOLLOWING UNBOUNDED FOLLOWING</pre>

MAX

This function calculates the maximum value of each group of values.

Syntax

```
(MAX(value) OVER (  
  [ PARTITION BY partition_value ]  
  [ ORDER BY order_value ]  
# frame clause  
  [ [ ROWS ] frame_start |  
    [ ROWS ] BETWEEN frame_start AND frame_end  
  ]  
)  
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.
frame clause	The reference point for all window frames is the current row. The optional frame clause further defines the frame by specifying the rows in a partition to combine with the current row. There are two types of window frames: • A fixed frame with two moving endpoints: Each row becomes the current row as the window frame slides forward in the partition. This type of frame is ideal for computing aggregations over moving time frames. • A resizing frame with one anchored endpoint: One row is a fixed

Argument	Description
	endpoint and the frame resizes up (PRECEDING) or down (FOLLOWING). This type of frame is ideal for computing running totals. The frame clause can be one of the following options: <pre>[ROWS] frame_start [ROWS] BETWEEN frame_start AND frame_end</pre> When a frame clause is not included, the window frame is unbounded: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
frame_start	The starting point of the frame. This argument can be one of the following values: <pre>UNBOUNDED PRECEDING positive_int PRECEDING CURRENT ROW positive_int FOLLOWING</pre>
frame_end	The end of the frame. This argument can be one of the following values: <pre>positive_int PRECEDING CURRENT ROW positive_int FOLLOWING UNBOUNDED FOLLOWING</pre>

MIN

This function calculates the minimum value of each group of values.

Syntax

```
(MIN(value) OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
# frame clause
  [ [ ROWS ] frame_start |
    [ ROWS ] BETWEEN frame_start AND frame_end
  ]
)
```

```
)
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.
frame clause	The reference point for all window frames is the current row. The optional frame clause further defines the frame by specifying the rows in a partition to combine with the current row. There are two types of window frames: • A fixed frame with two moving endpoints: Each row becomes the current row as the window frame slides forward in the partition. This type of frame is ideal for computing aggregations over moving time frames. • A resizing frame with one anchored endpoint: One row is a fixed endpoint and the frame resizes up (PRECEDING) or down (FOLLOWING). This type of frame is ideal for computing running totals. The frame clause can be one of the following options: <pre>[ROWS] frame_start [ROWS] BETWEEN frame_start AND frame_end</pre> When a frame clause is not included, the window frame is unbounded: <code>ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING</code>.

Argument	Description
frame_start	The starting point of the frame. This argument can be one of the following values: <pre> UNBOUNDED PRECEDING positive_int PRECEDING CURRENT ROW positive_int FOLLOWING </pre>
frame_end	The end of the frame. This argument can be one of the following values: <pre> positive_int PRECEDING CURRENT ROW positive_int FOLLOWING UNBOUNDED FOLLOWING </pre>

NTILE

This function divides the rows in the partition into the specified number of ranked groups and returns the group that each value belongs to.

Syntax

```

(NTILE(number_of_groups) OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
)
AS ?variable )

```

Argument	Description
number_of_groups	Required argument that defines the number of ranking groups.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each

Argument	Description
	partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.

PERCENTILE

This function divides the rows in the partition into 100 ranked groups and returns the group that each value belongs to.

Syntax

```
(PERCENTILE(value) OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
)
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.

PRODUCT

This function calculates the product of each group of values.

Syntax

```
(PRODUCT(value) OVER (  
  [ PARTITION BY partition_value ]  
  [ ORDER BY order_value ]  
# frame clause  
  [ [ ROWS ] frame_start |  
    [ ROWS ] BETWEEN frame_start AND frame_end  
  ]  
)  
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.
frame clause	The reference point for all window frames is the current row. The optional frame clause further defines the frame by specifying the rows in a partition to combine with the current row. There are two types of window frames: • A fixed frame with two moving endpoints: Each row becomes the current row as the window frame slides forward in the partition. This type of frame is ideal for computing aggregations over moving time frames. • A resizing frame with one anchored endpoint: One row is a fixed endpoint and the frame resizes up (PRECEDING) or down (FOLLOWING). This type of frame is ideal for computing running totals.

Argument	Description
	The frame clause can be one of the following options: <pre>[ROWS] frame_start [ROWS] BETWEEN frame_start AND frame_end</pre> When a frame clause is not included, the window frame is unbounded: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
frame_start	The starting point of the frame. This argument can be one of the following values: <pre>UNBOUNDED PRECEDING positive_int PRECEDING CURRENT ROW positive_int FOLLOWING</pre>
frame_end	The end of the frame. This argument can be one of the following values: <pre>positive_int PRECEDING CURRENT ROW positive_int FOLLOWING UNBOUNDED FOLLOWING</pre>

QUARTILE

This function divides the rows in the partition into four ranked groups and returns the group that each value belongs to.

Syntax

```
(QUARTILE(value) OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
)
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.

ROW_NUMBER

This function assigns unique numbers to each row in the partition.

Syntax

```
(ROW_NUMBER() OVER (
  [ PARTITION BY partition_value ]
  [ ORDER BY order_value ]
)
AS ?variable )
```

Argument	Description
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.

SUM

This function calculates the sum of each group of values.

Syntax

```
(SUM(value) OVER (  
  [ PARTITION BY partition_value ]  
  [ ORDER BY order_value ]  
# frame clause  
  [ [ ROWS ] frame_start |  
    [ ROWS ] BETWEEN frame_start AND frame_end  
  ]  
)  
AS ?variable )
```

Argument	Description
value	Required argument that defines the group of values to operate on.
partition_value	The optional PARTITION BY clause forms the groups of rows, dividing the result set into the partitions defined by the given <code>partition_value</code> . If you do not include PARTITION BY, the partition becomes the entire result set. When PARTITION BY is included, the function is applied to the group of rows in each partition.
order_value	The optional ORDER BY clause defines the order or sequence of rows within each partition.
frame clause	The reference point for all window frames is the current row. The optional frame clause further defines the frame by specifying the rows in a partition to combine with the current row. There are two types of window frames: • A fixed frame with two moving endpoints: Each row becomes the current row as the window frame slides forward in the partition. This type of frame is ideal for computing aggregations over moving time frames. • A resizing frame with one anchored endpoint: One row is a fixed

Argument	Description
	endpoint and the frame resizes up (PRECEDING) or down (FOLLOWING). This type of frame is ideal for computing running totals. The frame clause can be one of the following options: <pre>[ROWS] frame_start [ROWS] BETWEEN frame_start AND frame_end</pre> When a frame clause is not included, the window frame is unbounded: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
frame_start	The starting point of the frame. This argument can be one of the following values: <pre>UNBOUNDED PRECEDING positive_int PRECEDING CURRENT ROW positive_int FOLLOWING</pre>
frame_end	The end of the frame. This argument can be one of the following values: <pre>positive_int PRECEDING CURRENT ROW positive_int FOLLOWING UNBOUNDED FOLLOWING</pre>

Examples

This example queries the sample Tickit data set to return the percentage of a salesperson's total sales that came from the "Gypsy" event:

```
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event_name ?fname ?lname
((?dollars * 100.0/(SUM(?dollars) OVER(PARTITION BY ?event))) as ?percent_of_sales)
FROM <http://anzograph.com/tickit>
WHERE {
    ?sale tickit:eventid ?event .
    ?event tickit:eventname ?event_name .
    ?sale tickit:sellerid ?salesperson .
```

```

?sale tickit:pricepaid ?dollars .
?salesperson tickit:firstname ?fname .
?salesperson tickit:lastname ?lname .
FILTER(?event_name = "Gypsy").
}
ORDER BY ?event_name desc(?percent_of_sales)

```

event_name	fname	lname	percent_of_sales
Gypsy	Zoe	Sosa	100
Gypsy	Xaviera	Jacobson	50.9415
Gypsy	Brianna	Mcfarland	50.5076
Gypsy	Alexa	Baird	45.7926
Gypsy	Roanna	Wood	42.0408
Gypsy	Colette	Clay	36.9388
Gypsy	Amela	Holman	35.7277
Gypsy	Aubrey	Terrell	32.2457
Gypsy	Bruno	Griffin	31.8139
Gypsy	Damian	Berger	31.2459
Gypsy	Zelenia	Woods	31.1616
Gypsy	Imogene	McClean	31.0005
...			

857 rows

This example queries the sample Tickit data set to return a running total of the number of tickets sold for the event "Mamma Mia!":

```

PREFIX tickit: <http://anzograph.com/tickit/>
SELECT ?event ?month (SUM(?qty) OVER (PARTITION BY ?month ORDER BY ?event
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS ?tickets)
FROM <http://anzograph.com/tickit>
WHERE {
    ?s tickit:qtysold ?qty .
    ?s tickit:eventid ?eventid .
    ?eventid tickit:eventname ?event .
    ?s tickit:dateid ?date .
    ?date tickit:month ?month .
    filter(?event="Mamma Mia!")
}
ORDER BY ?tickets
LIMIT 100

```

event	month	tickets
Mamma Mia!	FEB	1
Mamma Mia!	AUG	1
Mamma Mia!	MAR	1
Mamma Mia!	MAY	1
Mamma Mia!	JUN	2
Mamma Mia!	SEP	2
Mamma Mia!	AUG	2
Mamma Mia!	DEC	2
Mamma Mia!	NOV	2
Mamma Mia!	OCT	2
Mamma Mia!	JAN	2
Mamma Mia!	MAR	2
Mamma Mia!	NOV	3
Mamma Mia!	JAN	3
Mamma Mia!	JUN	4
Mamma Mia!	JUL	4
Mamma Mia!	SEP	4
...		
100 rows		

Advanced Grouping Sets

Graph Lakehouse supports creating advanced reports using grouping set extensions in the GROUP BY clause. Advanced grouping expressions enable users to conduct multidimensional analysis using a single statement in a single query to calculate different combinations of aggregations. This topic describes the Graph Lakehouse grouping set extensions:

- **CUBE:** Use CUBE expressions to generate subtotals for all combinations of different dimensions. For example, use CUBE to report on revenue by various dimensions such as region, time, and department.
- **ROLLUP:** Use ROLLUP expressions to generate subtotals for hierarchical levels of the same dimension, such as time or geography. For example, use ROLLUP to report on revenue by year, month, and day or by country, state, and city.
- **GROUPING SETS:** Use GROUPING SETS expressions to group the GROUP BY list into subsets.

- **GROUPING:** If a GROUP BY clause includes a CUBE, ROLLUP, or GROUPING SETS expression, the results of the expression might include unbound (NULL) values. Unbound values can result from either the WHERE clause operations (which are input to the GROUP BY clause) or the UNION of the individual grouping results in the GROUP BY clause. Using a GROUPING expression in conjunction with CUBE, ROLLUP, and GROUPING SETS enables users to determine the reason for an unbound result. GROUPING denotes the cause of an unbound result by returning **1** if the unbound value is the result of the grouping operation in the GROUP BY clause or **0** if the unbound value originated as input to the GROUP BY clause.

Extension Libraries

The topics in this section provide descriptions, usage information, and examples for the Graph Lakehouse extension libraries.

In this section:

Apache Arrow Library

Apache Arrow is a software development platform for building high performance applications that process and transport large data sets. It is designed to improve both the performance of analytical algorithms and the efficiency of moving data from one system to another. One important feature of Apache Arrow is its in-memory columnar format, a standardized, language-agnostic specification for representing structured, table-like data sets in memory.

Graph Lakehouse can act as an Apache Arrow client for graph query driven exports and imports of in-memory data sets. Graph Lakehouse provides a collection of services that support the Arrow Flight protocol for integration with leading ML and other Big Data Ecosystems, including Python Pandas, Spark MLLIB and Google Tensorflow, Cassandra, Kudu, and Hadoop. This topic provides details about the Arrow services:

- [arrow_flight_get](#): Returns data about a specific flight from the flight server.
- [arrow_flight_get_info](#): Returns metadata about a specific flight.
- [arrow_flight_list](#): Lists all of the available flights known to the flight server.
- [arrow_flight_push](#): Pushes query results directly to the flight server.
- [arrow_flight_push_csv](#): Pushes a CSV file to the flight server.

arrow_flight_get

This service returns data for a specific flight from the flight server.

Syntax

```
SERVICE <http://cambridgesemantics.com/anzograph/inmemory#arrow_flight_get>
("IP", port, flight_type, "flight_name", "root_cert_file")
```


Argument	Type	Description
IP	string	Flight server IP address.
port	int	Server port number.
flight_type	int	Flight descriptor's type: 1-PATH, 2-CMD .
flight_name	string	Name of the flight.
root_cert_file	string	Path to a trusted TLS root certificate. For non-TLS operation, specify an empty path.

The function returns the column data for the specified flight.

Example 1

Returns CMD flight data using a custom name.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT *
WHERE {
  SERVICE exfun:arrow_flight_get("10.117.2.36", 5005, 2, "tickit_total_price",
"/tmp/arrow/keys/root-ca.pem") { }
}
ORDER BY ?ticketPrice ?numtickets
LIMIT 3
```

```
ticketPrice | numtickets | totalPrice
-----+-----+-----
20.000000   |      1      | 20.000000
20.000000   |      1      | 20.000000
20.000000   |      1      | 20.000000
```

Example 2

Returns PATH flight.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT *
```

```

WHERE {
  SERVICE exfun:arrow_flight_get("10.117.2.36", 5005, 1, "/tmp/arrow/data/iris.csv",
"/tmp/arrow/keys/root-ca.pem") { }
}
ORDER BY ?sepal_length ?sepal_width ?petal_length ?petal_width
LIMIT 3

```

sepal_length	sepal_width	petal_length	petal_width	variety
4.300000	3.000000	1.100000	0.100000	Setosa
4.400000	2.900000	1.400000	0.200000	Setosa
4.400000	3.000000	1.300000	0.200000	Setosa

Example 3

Returns CMD flight using the query name.

```

PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT *
WHERE {
  SERVICE exfun:arrow_flight_get("10.117.2.36", 5005, 2, "SELECT ?ticketPrice
?numtickets ?totalPrice WHERE { ?sell <priceperticket> ?ticketPrice; <numtickets>
?numtickets; <totalprice> ?totalPrice . }", "/tmp/arrow/keys/root-ca.pem") { }
}
ORDER BY ?ticketPrice ?numtickets
LIMIT 3

```

ticketPrice	numtickets	totalPrice
20.000000	1	20.000000
20.000000	1	20.000000
20.000000	1	20.000000

Example 4

Create a graph from a CMD flight with a custom name.

```

PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
INSERT
{
  graph <flight_graph1> {
    ?insUri <ticketPrice> ?ticketPrice;
    <numtickets> ?numtickets;
    <totalPrice> ?totalPrice.
  }
}

```

```

    }
}
WHERE {
    SERVICE exfun:arrow_flight_get("10.117.2.36", 5005, 2, "tickit_total_price",
"/tmp/arrow/keys/root-ca.pem") {
    }
    BIND(IRI(CONCAT("http://CSI.COM/", STRUUID()))) as ?insUri)
}

```

To display the graph data, you can run the following query:

```

SELECT ?ticketPrice ?numtickets ?totalPrice
FROM <flight_graph1>
WHERE {
    ?insUri <ticketPrice> ?ticketPrice;
    <numtickets> ?numtickets;
    <totalPrice> ?totalPrice.
}
ORDER BY ?ticketPrice ?numtickets
LIMIT 3

```

ticketPrice	numtickets	totalPrice
20.000000	1	20.000000
20.000000	1	20.000000
20.000000	1	20.000000

Example 5

Creates a graph from a PATH flight.

```

PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
INSERT
{
    graph <flight_graph2> {
        ?insUri <sepal_length> ?sepal_length;
        <sepal_width> ?sepal_width;
        <petal_length> ?petal_length;
        <petal_width> ?petal_width;
        <variety> ?variety.
    }
}
WHERE {
    SERVICE exfun:arrow_flight_get("10.117.2.36", 5005, 1, "/tmp/arrow/data/iris.csv",

```

```
"/tmp/arrow/keys/root-ca.pem") { }
  BIND(IRI(CONCAT("http://CSI.COM/", STRUUID()))) as ?insUri)
}
```

Example 6

Creates a graph from a CMD flight with a query name.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
INSERT
{
  graph <flight_graph3> {
    ?insUri <ticketPrice> ?ticketPrice;
    <numtickets> ?numtickets;
    <totalPrice> ?totalPrice.
  }
}
WHERE {
  SERVICE exfun:arrow_flight_get("10.117.2.36", 5005, 2, "SELECT ?ticketPrice
?numtickets ?totalPrice WHERE { ?sell <priceperticket> ?ticketPrice; <numtickets>
?numtickets; <totalprice> ?totalPrice . }", "/tmp/arrow/keys/root-ca.pem") { }
  BIND(IRI(CONCAT("http://CSI.COM/", STRUUID()))) as ?insUri)
}
```

arrow_flight_get_info

This service returns metadata about a specific flight.

Syntax

```
SERVICE <http://cambridgesemantics.com/anzograph/inmemory#arrow_flight_get_info>
("IP", port, "type", "path", "root_cert_file")
```

Argument	Type	Description
IP	string	Flight server IP address.
port	int	Server port number.
type	string	Flight descriptor's type: 1-PATH, 2-CMD .

Argument	Type	Description
path	string	Flight command or path.
root_cert_file	string	Path to a trusted TLS root certificate. For non-TLS operation, specify an empty path.

Returns

Type	Description
string	Flight's schema.
long	Total number of records (rows) in the dataset.
long	Total number of bytes in the dataset.

Example 1

This example returns metadata.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT (row_opt_string(?flight_info,0) as ?schema) (row_get_long(?flight_info,1) as
?records) (row_get_long(?flight_info,2) as ?bytes)
WHERE {
  SELECT (exfun:arrow_flight_get_info("10.117.2.36", 5005, 2, "SELECT ?ticketPrice
    ?numtickets ?totalPrice WHERE { ?sell <priceperticket> ?ticketPrice;
    <numtickets> ?numtickets; <totalprice> ?totalPrice . }",
    "/tmp/arrow/keys/root-ca.pem") as ?flight_info)
}
```

```
schema | records | bytes
-----+-----+-----
?ticketPrice:double ?numtickets:int ?totalPrice:double | 192497 | 3942600
```

Example 2

Returns PATH name flight's metadata.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT (row_opt_string(?flight_info,0) as ?schema)
        (row_get_long(?flight_info,1) as ?records)
        (row_get_long(?flight_info,2) as ?bytes)
WHERE {
    SELECT (exfun:arrow_flight_get_info("10.117.2.36", 5005, 1,
"/tmp/arrow/data/iris.csv", "/tmp/arrow/keys/root-ca.pem") as ?flight_info)
}
```

```
schema
        | records | bytes
-----+-----+-----
?sepal_length:double ?sepal_width:double ?petal_length:double ?petal_width:double
?variety:String      |    150    |    7368
```

Example 3

Returns custom name flight's metadata.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT (row_opt_string(?flight_info,0) as ?schema)
        (row_get_long(?flight_info,1) as ?records)
        (row_get_long(?flight_info,2) as ?bytes)
WHERE {
    SELECT (exfun:arrow_flight_get_info("10.117.2.36", 5005, 2, "tickit_total_price",
"/tmp/arrow/keys/root-ca.pem") as ?flight_info)
}
```

```
schema
        | records | bytes
-----+-----+-----
?ticketPrice:double ?numtickets:int ?totalPrice:double |    192497    |    3942600
```

arrow_flight_list

This service lists all of the available flights known to the flight server.

Syntax

```
SERVICE <http://cambridgesemantics.com/anzograph/inmemory#arrow_flight_list>
("IP", port, "root_cert_file")
```

Argument	Type	Description
IP	string	Flight server IP address.
port	int	Server port number.
root_cert_file	string	Path to a trusted TLS root certificate. For non-TLS operation, specify an empty path.

Returns

Type	Description
long	Flight number.
long	Total number of endpoints associated with the flight (dataset).
int	Endpoint number in the flight.
string	Type of flight. CMD-Command, PATH-Path.
string	Name of the flight.
string	List of locations where ticket can be redeemed.

Example

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT *
WHERE {
    SERVICE exfun:arrow_flight_list("10.117.2.36", 5005, "/tmp/arrow/keys/root-ca.pem")
    {}
}
```

```
flight | total_endpoints | endpoint_num | flight_type | flight_name
```

```

| location
-----+-----+-----+-----+-----+
|
|
|
1      |      1      |      1      |  PATH      |  /tmp/arrow/data/iris.csv

| grpc+tls://10.117.2.36:5005
2      |      1      |      1      |  CMD       |  tickit_total_price

| grpc+tls://10.117.2.36:5005
3      |      1      |      1      |  CMD       |  SELECT ?ticketPrice
?numtickets ?totalPrice WHERE { ?sell <priceperticket> ?ticketPrice; <numtickets>
?numtickets; <totalprice> ?totalPrice . } | grpc+tls://10.117.2.36:5005

```

arrow_flight_push

This service pushes out the result of a query directly to the flight server.

Syntax

```
SERVICE <http://cambridgesemantics.com/anzograph/inmemory#arrow_flight_push>
("IP", port, "cmd_name", "root_cert_file")
```

Argument	Type	Description
IP	string	Server IP address.
port	int	Server port number.
cmd_name	string	Flight name that can be a SPARQL command or custom name. Specify "CMD" to use SPARQL command as name, otherwise provide a custom name.
root_cert_file	string	Path to a trusted TLS root certificate. For non-TLS operation, specify an empty path.

Returns

Type	Description
long	The number of rows pushed to the server.
string	Name of the flight.

Example 1

Returns the flight name as user-defined name, "tickit_total_price".

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT ?num_rows ?flight_name
FROM <tickit>
WHERE {
  {
    SELECT ?ticketPrice ?numtickets ?totalPrice
    WHERE
    {
      ?sell <priceperticket>    ?ticketPrice;
      <numtickets>              ?numtickets;
      <totalprice>              ?totalPrice .
    }
  }
  SERVICE exfun:arrow_flight_push("10.117.2.36", 5005,
    "tickit_total_price", "/tmp/arrow/keys/root-ca.pem") { }
```

```
num_rows | flight_name
-----+-----
192497   | tickit_total_price
```

Example 2

This example returns the flight name as a SPARQL query.

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT ?num_rows ?flight_name
FROM <tickit>
WHERE {
  {
```

```

SELECT ?ticketPrice ?numtickets ?totalPrice
WHERE
{
  ?sell <priceperticket>    ?ticketPrice;
  <numtickets>              ?numtickets;
  <totalprice>              ?totalPrice .
}
}
SERVICE exfun:arrow_flight_push("10.117.2.36", 5005, "CMD",
  "/tmp/arrow/keys/root-ca.pem") { }
}

```

```

num_rows | flight_name
-----+-----
192497   | SELECT ?ticketPrice ?numtickets ?totalPrice WHERE { ?sell <priceperticket>
?ticketPrice;
          <numtickets> ?numtickets; <totalprice> ?totalPrice . }

```

arrow_flight_push_csv

This service pushes a CSV file to the flight server.

Syntax

```

SERVICE <http://cambridgesemantics.com/anzograph/inmemory#arrow_flight_push_csv>
("IP", port, "file", "root_cert_file")

```

Argument	Type	Description
IP	string	Server IP address.
port	int	Server port number.
file	string	CSV file path.
root_cert_file	string	Path to a trusted TLS root certificate. For non-TLS operation, specify an empty path.

Returns

Type	Description
long	The number of rows pushed to the server.
string	Flight path.

Example

```
PREFIX exfun: <http://cambridgesemantics.com/anzograph/inmemory#>
SELECT *
WHERE {
  SERVICE exfun:arrow_flight_push_csv("10.117.2.36", 5005,
    "/tmp/arrow/data/iris.csv", "/tmp/arrow/keys/root-ca.pem") {}
}
```

```
num_rows | path
-----+-----
150      | /tmp/arrow/data/iris.csv
```

Data Science Library

The topics in this section provide details about the Graph Lakehouse Data Science library.

Tip

Altair offers an Apache Zeppelin Docker image that includes a collection of notebooks with details and example usage of each of the Data Science functions. See [Zeppelin Notebook Integration](#) for more information.

In this section:

Correlation Aggregates

The correlation aggregates determine the relationship between elements.

- [Matthews Correlation Coefficient \(MCC\)](#): Provides a measure of the quality of binary classifications of a condition with observed versus predicted scoring.

- **Pearson Correlation Coefficient (PCC)**: Determines the extent to which two variables are linearly related: positive, negative, or no relationship.
- **Spearman's Correlation Coefficient (SCC)**: Determines how well the relationship between two variables can be described using a monotonic function.

Note

The URI for the data science functions is

`<http://cambridgesemantics.com/anzograph/statistics#>`. For readability, the syntax for each function below includes the prefix `stats:`, defined as `PREFIX stats: <http://cambridgesemantics.com/anzograph/statistics#>`.

Matthews Correlation Coefficient (MCC)

The [Matthews correlation coefficient](#) aggregate returns a coefficient value between observed and predicted binary classifications.

Syntax

```
stats:mcc(x, y)
```

Parameter	Type	Description
x	boolean	First variable column data.
y	boolean	Second variable column data.

Returns

Type	Description
double	Coefficient value that shows the extent to which observed and predicted binary classifications are related.

Pearson Correlation Coefficient (PCC)

The [Pearson correlation coefficient](#) aggregate determines the extent to which two variables are linearly related: positive, negative, or no relationship.

Syntax

```
stats:pcc(x, y)
```

Parameter	Type	Description
x	boolean	First variable column data.
y	boolean	Second variable column data.

Returns

Type	Description
double	Coefficient that shows the extent to which two variables are linearly related.

Spearman's Correlation Coefficient (SCC)

The [Spearman's rank correlation coefficient](#) aggregate determines how well the relationship between two variables can be described using a monotonic function.

Syntax

```
stats:scc(rank_X, rank_Y)
```

Parameter	Type	Description
rank_X	double	First set of ranked data.
rank_Y	double	Second set of ranked data.

Returns

Type	Description
double	Coefficient between ranked datasets.

Distribution Functions

The distribution functions calculate the probability of a given value over a random distribution.

- [Cumulative Distribution Functions \(CDF\)](#): Calculate the probability of a random variable X taking on a value less than or equal to Y.
- [Bernoulli Distribution \(BERNDIST\)](#): Determines the probability of a specific event occurring, or not occurring, in tests that have only two possible outcomes: success (1) or failure (0).
- [Beta-Binomial Distribution \(BETABINDIST\)](#): Computes probability using a combination of both binomial and beta probability distributions.
- [Hypergeometric Distribution \(HYPGEODIST\)](#): Calculates probability from a distribution that is often used to predict the outcome of a process in which different elements are randomly drawn from a collection and not replaced.
- [Logarithmic \(Series\) Distribution \(LOGSERDIST\)](#): Calculates probability using a discrete probability distribution derived from the Maclaurin series expansion.
- [Skellam Distribution \(SKELLAMDIST\)](#): Calculates probability using the Skellam distribution which models the difference between two independent Poisson distributed variables.

Note

The URI for the data science functions is

`<http://cambridgesemantics.com/anzograph/statistics#>`. For readability, the syntax for each function below includes the prefix `stats:`, defined as `PREFIX stats: <http://cambridgesemantics.com/anzograph/statistics#>`.

Cumulative Distribution Functions (CDF)

A [Cumulative distribution function](#) function calculates the probability of a random variable X taking on a value less than or equal to Y . The following functions produce cumulative distribution calculations:

- [Binomial Distribution \(BINOMDIST\)](#): Calculates the probability for X successes in N trials given a probability of success P for each trial.
- [Chi-Squared Distribution \(CHISQDIST\)](#): Calculates probability often used in hypothesis testing to compare an observed distribution with a theoretical one. Also provides a way to show a relationship between two categorical variables.
- [Continuous Uniform Distribution \(CONUNIDIST\)](#): Calculates probability using continuous probability distribution concerned with events that are equally likely to occur.
- [Discrete Uniform Distribution \(DISCUNIDIST\)](#): Calculates probability using symmetric probability distribution where a finite number of values are equally likely to be observed and every one of n values has equal probability.
- [Exponential Distribution \(EXPDIST\)](#): Calculates probability using a distribution that describes time between events in a Poisson point process (where events occur continuously and independently at a constant average rate).
- [Laplace Distribution \(LAPLACEDIST\)](#): Calculates probability using a distribution that represents differences between two independent variables that have identical exponential distributions (also called double exponential distribution).
- [Log Normal Distribution \(LOGNORDIST\)](#): Calculates probability using a distribution of a random variable whose logarithm follows a normal distribution. Log normal distributions are widely used in risk analysis.
- [Negative Binomial Distribution \(NEGBINDIST\)](#): Calculates probability using a discrete probability distribution that concerns the number of trials which must occur in order to have a predetermined number of successes.

- **Normal Distribution (NORMDIST)**: Calculates probability using a continuous probability distribution of data in which the majority of data points are relatively similar, within a small range of values having few outliers.
- **Poisson Distribution (POISDIST)**: Calculates probability using a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time or space and those events occur with a known constant rate and occur independently of the time since the last event.
- **Student's T-Distribution (TDIST)**: Calculates probability using the Student's t-distribution and associated t scores. Often used in hypothesis testing when the sample size is small and/or when the population variance is unknown.
- **TDigest Metric (TDIGEST)**: Creates an estimate of the median (and more generally, any percentile) from either distributed data or streaming data, using a t-Digest probabilistic data structure.
- **Weibull Distribution (WEIBULDIST)**: Calculates probability from a continuous probability distribution that is commonly used to assess product reliability, analyze product life data and failure times.

Binomial Distribution (BINOMDIST)

The [Binomial distribution](#) aggregate calculates the probability for X successes in N trials given a probability of success P for each trial.

Syntax

```
stats:binomdist(data, n, k, "success_string")
```

Parameter	Type	Description
data	string	Column data.
n	long	Number of trials.

Parameter	Type	Description
k	long	Number of successes in n trials.
success_string	string	Defines the success string.

Returns

Type	Description
double	Probability mass function value.
double	Lower cumulative distribution: probability ($\leq k$) under the area of distribution.
double	Upper cumulative distribution: probability ($> k$) under the area of distribution.

Chi-Squared Distribution (CHISQDIST)

The [Chi-squared distribution](#) aggregate calculates probability that is often used in hypothesis testing to compare an observed distribution with a theoretical one. It also provides a way to show a relationship between two categorical variables.

Syntax

```
stats:chisqdist(data, s)
```

Parameter	Type	Description
data	double	Sample data.
s	double	Population standard deviation.

Returns

Type	Description
double	Mean of the distribution.
double	Standard deviation of the distribution.
double	Variance of the distribution.
double	Chi-squared statistic: $[(n - 1) * s^2] / d^2$ where d is the standard deviation of the population, s is the standard deviation of the sample, and n is the sample size.
long	Number of samples: the degrees of freedom(k) is $(count-1)$.
double	Probability mass function value.
double	Cumulative distribution: the probability for $\leq$ the chi-squared statistic.

Continuous Uniform Distribution (CONUNIDIST)

The [Continuous uniform distribution](#) aggregate calculates probability using a continuous probability distribution concerned with events that are equally likely to occur.

Syntax

```
stats:conunidist(data, a, b)
```

Parameter	Type	Description
data	double	Column data.
a	double	Minimum value of the probability interval.

Parameter	Type	Description
b	double	Maximum value of the probability interval.

Returns

Type	Description
double	Cumulative distribution: probability under the area of distribution.
double	Probability density function value.
double	Differential entropy in nats.

Discrete Uniform Distribution (DISCUNIDIST)

The [Discrete uniform distribution](#) aggregate calculates probability using symmetric probability distribution where a finite number of values are equally likely to be observed and every one of n values has equal probability.

Syntax

```
stats:discunidist(data, k)
```

Parameter	Type	Description
data	long	Column data.
k	long	The number of outcomes.

Returns

Type	Description
double	Cumulative distribution: probability under the area of distribution.

Type	Description
double	Probability density function value.
double	Differential entropy in nats.

Exponential Distribution (EXPDIST)

The [Exponential distribution](#) aggregate calculates probability using a distribution that describes time between events in a Poisson point process (where events occur continuously and independently at a constant average rate).

Syntax

```
stats:expdist(data, x)
```

Parameter	Type	Description
data	long	Column data.
x	double	The probability for the interval.

Returns

Type	Description
double	Lower cumulative distribution: probability ($\leq k$) under the area of distribution.
double	Upper cumulative distribution: probability ($> k$) under the area of distribution.
double	Probability density function value.
double	Differential entropy in nats.

Laplace Distribution (LAPLACEDIST)

The [Laplace distribution](#) aggregate calculates probability using a distribution that represents differences between two independent variables that have identical exponential distributions (also called *double exponential distribution*).

Syntax

```
stats:laplacedist(data, "c", x1, x2)
```

Parameter	Type	Description
data	double	Column data.
c	string	"below", "above", "bet" (between), or "out" (outside).
x1	double	Lower number (>0) to find the probability.
x2	double	Upper number (>0) to find the probability.

Returns

Type	Description
double	Mean of the distribution.
double	Scale parameter of the distribution.
double	Standard deviation of the distribution.
double	Variance of the distribution.
double	Differential entropy in nats.

Type	Description
double	Cumulative distribution: probability under the area of distribution.
double	Probability density function value for x1.
double	Probability density function value for x2.

Log Normal Distribution (LOGNORDIST)

The [Log-normal distribution](#) aggregate calculates probability using distribution of a random variable whose logarithm follows a normal distribution. The log normal distribution widely used in risk analysis.

Syntax

```
stats:lognordist(data, "c", x1, x2)
```

Parameter	Type	Description
data	double	Column data.
c	string	"below", "above", "bet" (between), or "out" (outside).
x1	double	Lower number (>0) to find the probability.
x2	double	Upper number (>0) to find the probability.

Returns

Type	Description
double	Mean of the distribution of natural logarithms distribution.

Type	Description
double	Standard deviation of the distribution of natural logarithms distribution.
double	Variance of the distribution.
double	Differential entropy in nats.
double	Cumulative distribution: probability under the area of distribution.
double	Probability density function value for x1.
double	Probability density function value for x2.

Negative Binomial Distribution (NEGBINDIST)

The [Negative binomial distribution](#) aggregate calculates probability using a discrete probability distribution that concerns the number of trials which must occur in order to have a predetermined number of successes.

Syntax

```
stats:negbindist("data", k, r, "success_string")
```

Parameter	Type	Description
data	string	Column data.
k	long	Number of successes.
r	long	Number of failures.
success_string	string	Defines the success string.

Returns

Type	Description
double	Probability mass function value.
double	Lower cumulative distribution: probability ($\leq k$) under the area of distribution.
double	Upper cumulative distribution: probability ($> k$) under the area of distribution.

Normal Distribution (NORMDIST)

The [Normal distribution](#) aggregate calculates probability using a continuous probability distribution of data in which the majority of data points are relatively similar, within a small range of values with few outliers.

Syntax

```
stats:normdist(data, "c", x1, x2)
```

Parameter	Type	Description
data	double	Column data.
c	string	"below", "above", "bet" (between), or "out" (outside).
x1	double	Lower number (>0) to find the probability.
x2	double	Upper number (>0) to find the probability.

Returns

Type	Description
double	Mean of the distribution.
double	Standard deviation of the distribution.
double	Variance of the distribution.
double	Differential entropy in nats.
double	Cumulative distribution: probability under the area of distribution.
double	Probability density function value for x1.
double	Probability density function value for x2.

Poisson Distribution (POISDIST)

The [Poisson distribution](#) function calculates probability using discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time or space, given these events occur with a known constant rate and occur independently of the time since the last event.

Syntax

```
stats:poisdist(data, k)
```

Parameter	Type	Description
data	long	Column data.
k	long	Probability of observing k events in an interval.

Returns

Type	Description
double	Probability mass function value.
double	Lower cumulative distribution: probability ($\leq k$) under the area of distribution.
double	Upper cumulative distribution: probability ($> k$) under the area of distribution.

Student's T-Distribution (TDIST)

The [Student's t-distribution](#) function calculates probability using the Student's t-distribution (and associated t scores) which are often used in hypothesis testing when the sample size is small and/or when the population variance is unknown.

Syntax

```
stats.tdist(data, m)
```

Parameter	Type	Description
data	double	Sample data.
m	double	Population mean.

Returns

Type	Description
double	Mean of the distribution.
double	Standard deviation of the distribution.

Type	Description
double	Variance of the distribution.
double	T-statistics: $t = [u - M] / [s / \sqrt{N}]$ where u is the sample mean, M is the population mean, s is the standard deviation of the sample, and N is the sample size.
long	Number of samples: the degrees of freedom is $(count-1)$.
double	Probability mass function value.
double	Cumulative distribution: the probability for $\leq$ t-statistics.

TDigest Metric (TDIGEST)

This function creates an estimate of the median (and more generally, any percentile) from either distributed data or streaming data, using a t-Digest probabilistic data structure. For background information about this function, see [Computing quantiles using t-Digests](#).

Syntax

```
stats:tdigest(data, p, q, cdf)
```

Parameter	Type	Description
data	double	Column data.
p	double	The percentile (0 - 100) to compute.
q	double	The quantile (0.0 - 1.0) to compute.
cdf	double	The CDF to use.

Returns

Type	Description
double	Percentile: the value below which a given percentage of observations falls.
double	Quantile: Cut point to dividing the observations in a sample.
double	The computation of $F(x)$ where F is the CDF of the distribution.

Weibull Distribution (WEIBULDIST)

The [Weibull distribution](#) function calculates probability from a continuous probability distribution commonly used to assess product reliability and analyze product life data and failure times.

Syntax

```
stats:weibuldist(data, k, x)
```

Parameter	Type	Description
data	double	Sample data.
k	double	The initial starting value for the shape parameter. A good guess is crucial to quick convergence.
x	double	The probability for a random variable.

Returns

Type	Description
double	The mean of the distribution.

Type	Description
double	The standard deviation of the distribution.
double	The variance of the distribution.
long	The count of the number of samples.
double	The estimated shape parameter(k) of the distribution from the mean and variance using the root finding method.
double	The estimated scale parameter(a) of the distribution from the mean and variance using the root finding method.
double	Differential entropy in nats.
double	Probability density function value.
double	Lower cumulative distribution: probability ($\leq x$) under the area of distribution.
double	Upper cumulative distribution: probability ($> x$) under the area of distribution.
long	The actual number of iterations performed to get an estimate of the k value.
double	The mean calculated using estimated values of k and a.
double	The variance calculated using estimated values of k and a.

Bernoulli Distribution (BERNDIST)

The [Bernoulli distribution](#) function determines the probability of success or failure (or Yes or No) in tests that have only two possible outcomes.

Syntax

```
stats:berndist("data", prob, "success_string")
```

Parameter	Type	Description
data	string	Column data.
prob	boolean	Probability of success (true) or failure (false).
success_string	string	The success message.

Returns

Type	Description
double	The Bernoulli distribution probability.

Beta-Binomial Distribution (BETABINDIST)

The [Beta-binomial distribution](#) function computes probability using a combination of both binomial and beta probability distributions.

Syntax

```
stats:betabindist(k, n, alpha, beta)
```

Parameter	Type	Description
k	double	The probability for the number.
n	double	The number of trials.
alpha, beta	double	Shape parameters.

Returns

Type	Description
double	The probability of occurrence k for a beta binomial n, alpha, beta.

Hypergeometric Distribution (HYPGEODIST)

The [Hypergeometric distribution](#) function calculates probability from a distribution often used to predict the outcome of a process in which different elements are randomly drawn from a collection and not replaced.

Syntax

```
stats:hypgeodist("data", n, k, "success_string")
```

Parameter	Type	Description
data	string	Column data.
n	int	The number of trials.
k	int	The number of success in n trials.
success_string	string	The success message.

Returns

Type	Description
double	The hypergeometric distribution probability.

Logarithmic (Series) Distribution (LOGSERDIST)

The [Logarithmic \(series\) distribution](#) function calculates probability using a discrete probability distribution derived from the Maclaurin series expansion.

Syntax

```
stats:logserdist("data", k, "success_string")
```

Parameter	Type	Description
data	string	Column data.
k	long	The probability for the number.
success_string	string	The success message.

Returns

Type	Description
double	The logarithmic distribution probability.

Skellam Distribution (SKELLAMDIST)

The [Skellam distribution](#) function calculates probability using the Skellam distribution which models the difference between two independent Poisson distributed variables.

Syntax

```
stats:skellamdist(n1_data, n2_data, k)
```

Parameter	Type	Description
n1_data	long	N1 column data.

Parameter	Type	Description
n2_data	long	N2 column data.
k	long	Probability for the number.

Returns

Type	Description
double	The Skellam probability.

Entropy Functions

The entropy functions determine variance and probability density across a given distribution.

- [Cross Entropy \(CROSSENTROPY\)](#): Computes cross-entropy, which is commonly used to quantify the difference between two probability distributions.
- [Discrete Entropy Metric \(DISCENTROPY\)](#): Calculates discrete entropy for maps on finite sets.
- [Differential Entropy or Continuous Entropy Metrics](#): These functions compute differential entropy (also referred to as continuous entropy), which is entropy defined for distributions with a continuous random variable.

Note

The URI for the data science functions is

`<http://cambridgesemantics.com/anzograph/statistics#>`. For readability, the syntax for each function below includes the prefix `stats:`, defined as `PREFIX stats: <http://cambridgesemantics.com/anzograph/statistics#>`.

Cross Entropy (CROSSENTROPY)

The [Cross-entropy](#) function computes cross-entropy, which is commonly used to quantify the difference between two probability distributions.

Syntax

```
stats:crossentropy(p, q)
```

Parameter	Type	Description
p	double	True probabilities for x.
q	double	Predicted probabilities for x.

Returns

Type	Description
double	The cross entropy value.

Discrete Entropy Metric (DISCENTROPY)

The [Discrete entropy](#) function calculates entropy for maps on finite sets, referred to as *discrete entropy*.

Syntax

```
stats:discentropy("data")
```

Parameter	Type	Description
data	string	Column data.

Returns

Type	Description
double	The discrete entropy value.

Differential Entropy or Continuous Entropy Metrics

[Differential entropy](#) (also referred to as continuous entropy) is entropy that can be computed for distributions with a continuous random variable.

The following functions produce entropy calculations. For details about the functions, see [Distribution Functions](#).

- Continuous Uniform Distribution (CONUNIDIST)
- Discrete Uniform Distribution (DISCUNIDIST)
- Exponential Distribution (EXPDIST)
- Laplace Distribution (LAPLACEDIST)
- Log Normal Distribution (LOGNORDIST)
- Normal Distribution (NORMDIST)
- Weibull Distribution (WEIBULDIST)

Profiling Metrics

The profiling metrics produce statistical metrics such as percentile, geometric mean, or skew on a given dataset.

- [Geometric Mean Metric \(GMEAN\)](#): Calculates geometric mean, defined as the n th root of the product of n positive numbers.
- [Percentile Metric \(PERCENTILE\)](#): Calculates 1 to 100 percentile of numeric values.
- [Skew Metric \(SKEWCOEFF\)](#): Calculates Pearson's coefficient of skewness on numeric values.

Note

The URI for the data science functions is

`<http://cambridgesemantics.com/anzograph/statistics#>`. For readability, the syntax for each function below includes the prefix `stats:`, defined as `PREFIX stats: <http://cambridgesemantics.com/anzograph/statistics#>`.

Geometric Mean Metric (GMEAN)

The [Geometric mean](#) function calculates geometric mean, defined as the nth root of the product of n positive numbers.

Syntax

```
stats:gmean (data)
```

Parameter	Type	Description
data	double	Column data.

Returns

Type	Description
double	The geometric mean value.

Percentile Metric (PERCENTILE)

The [Percentile metric](#) function calculates the percentile (1 to 100) of numeric values.

Syntax

```
stats:percentile (data, p)
```

Parameter	Type	Description
data	double	The dataset.
p	double	The percentile (0 - 100) to compute.

Returns

Type	Description
double	The percentile value.

Skew Metric (SKEWCOEFF)

The [Skewness metric](#) function calculates the Pearson's coefficient of skewness on numeric values.

Syntax

```
stats.skewcoeff(data, dp)
```

Parameter	Type	Description
data	double	The dataset.
dp	int	Number of decimal points to consider for the input data.

Returns

Type	Description
double	The mode (value that appears most frequently).
double	The median number in an ordered set of data.
double	The average value.
double	The standard deviation.
double	Pearson mode skewness or first skewness coefficient.
double	Pearson median skewness or second skewness coefficient.

Geospatial Library

Graph Lakehouse offers two packages of pre-built geospatial functions: **Geospatial** and **GeoSPARQL**. The geospatial functions follow geospatial operations implemented using ESRI's widely-used public domain geometry library of API functions (<https://github.com/Esri/geometry-api-java/wiki>). And the geoSPARQL functions follow the world-wide geospatial standard (<https://www.ogc.org/standards/geosparql/>) developed and promoted by the Open Geospatial Consortium (OGC) community to represent geospatial data in RDF format and query that data using the SPARQL query language.

The spatial features offer advanced capabilities for developing large scale location intelligence and geospatial applications to use along with rich data SPARQL analytics. Both sets of functions have been developed in compliance with OGC standards (<https://www.ogc.org/standards>).

In this section:

Geospatial Functions

The geospatial functions follow operations implemented using ESRI's widely-used public domain geometry library of API functions (<https://github.com/Esri/geometry-api-java/wiki>). This topic describes each function.

Data Types and Arguments

Arguments and return values are transient objects that are internal to Graph Lakehouse. The values may contain arbitrarily long sequences of raw data bytes. These objects cannot be directly persisted to the graph store, however, a *Custom* object (return value of a function) can be bound to a variable (using a BIND expression) and can be passed to functions as arguments.

The life span of a Custom object handle cannot exceed one query. A Custom object has to be serialized into some form of text string or a URI of a user-defined data type to return as query results or save in the graph store.

Note

Some functions include arguments labeled as *x*, *y*, and *z*. These arguments correspond to longitude (*x*), latitude (*y*), and height or altitude (*z*) values in standard geospatial coordinate systems. Coordinates that specify an *x*, *y* location in a Cartesian coordinate system or an *x*, *y*, *z* coordinate in a three dimensional system represent locations on the Earth's surface relative to other locations. In addition to the Cartesian coordinate system, the Graph Lakehouse geospatial functions support other coordinate systems including Spherical, Cylindrical, and Elliptical.

Representations of function syntax included in this topic use braces (`[arg]`) to represent optional arguments. In addition, the convention, `arg1 [, ..., argN]` is used to indicate when you can include any number of arguments, 1 to *N*. Similarly, for functions that require an index argument, such as `ST_PointN`, the index values also range from 1 to *N*.

Geospatial functions use geometry inputs provided in the following forms:

- WKT String geometry
- WKTLiteral geometry
- GMLLiteral geometry
- Graph Lakehouse **Custom** geometry data type (wrapper encapsulation of the OGCGeometry standard data type from the ESRI library)

Important

The Graph Lakehouse geospatial functions follow the standards below for reading and parsing particular geometry values:

- For GML readers and GML literal values, the functions support GML version 2.0 and use the CRS 84 coordinate system by default.
- The Shape (`.shp`) file reader supports version 1000.
- For WKT geometry parsing, the functions follow the WKT version 1 standard and use

the 4326 coordinate system by default.

The KML reader follows the version 2.2.0 standard.

Functions

The geospatial functions are grouped by the following categories:

- **Point and Multipoint Functions:** These functions operate on Point (a single location in space that has, at a minimum, an x-coordinate and y-coordinate) and MultiPoint (an ordered collection of points) shapes.
- **LineString and MultiLineString Functions:** These functions operate on LineString (a sequence of points with boundary endpoints) and MultiLineString (a collection of LineStrings) shapes.
- **Polygon Functions:** These functions operate on Polygons, MultiPolygons, Circle and Circle Arc Polygons, Ellipse and Elliptical Arc Polygons, Rectangle Polygons, and Squire Polygon.
- **Utility Functions:** These functions are common to all shapes and provide operations such as conversion, translation, or conditional testing.
- **Aggregator (UDA) Functions:** Aggregators construct new geometric shapes from multiple aggregated geometries.
- **Services (UDS) Functions:** The services extract geometric information from source files such as .shp, .gml, .kml, and .json.

Note

The URI for the geospatial functions is

`<http://www.opengis.net/def/function/geosparql/>`. For readability, the syntax for each function below includes the prefix `geof:`, defined as `PREFIX geof:`
`<http://www.opengis.net/def/function/geosparql/>`.

Point and Multipoint Functions

- **ST_Point**: Constructs a Point from a given set of x, y, z, and m coordinates.
- **ST_X**: Returns the X coordinate of a Point.
- **ST_SetX**: Sets the X coordinate of a Point.
- **ST_Y**: Returns the Y coordinate of a Point.
- **ST_SetY**: Sets the Y coordinate of a Point.
- **ST_Z**: Returns the Z coordinate of a Point.
- **ST_SetZ**: Sets the Z coordinate of a Point.
- **ST_M**: Returns the M coordinate of a Point.
- **ST_SetM**: Sets the M coordinate of a Point.
- **ST_Bin**: Returns the bin ID of a Point.
- **ST_BinEnvelope**: Returns the bin envelope for a Point or bin ID.
- **ST_MultiPoint**: Constructs a MultiPoint geometry from a set of x and y coordinate pairs.
- **ST_PointN**: Returns a Point that is at the Nth index position in a MultiPoint.
- **ST_GeomFromText**: Constructs a Point from a given set of well known text (WKT) coordinates.
- **ST_GeomFromWKB**: Constructs a Point from a given set of well known binary (WKB) coordinates.

ST_Point

This function returns a Point as a custom object constructed from a given set of x, y, z, and m coordinates. You can specify two coordinates (x and y), three coordinates (x, y, and z), or four coordinates (x, y, z, and m).

Syntax

```
geof:ST_Point(x, y [, z ] [, m ])
```

Argument	Type	Description
x	double	The X coordinate of the point.
y	double	The Y coordinate of the point.
z	double	The optional Z coordinate of the point.
m	double	The optional M coordinate of the point.

Returns

Type	Description
Custom object	The point.

Example

```
SELECT (geof:ST_AsText(geof:ST_SetM(geof:ST_Point(1,2,3),4)) as ?point)
```

ST_X

This function returns the X coordinate of a Point geometry.

Syntax

```
geof:ST_X(geom)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
double	The X coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (MIN(?point_x) as ?3d_min)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
  BIND(geof:ST_X(?point_2d) as ?point_x)
  BIND(geof:ST_Y(?point_2d) as ?point_y)
}
```

ST_SetX

This function sets the X coordinate of a Point geometry.

Syntax

```
geof:ST_SetX(geom, x)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
x	double	The X coordinate point value to set.

Returns

Type	Description
Custom object	The X coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geo:ST_SetX(geo:ST_Point(1,2),4)) as ?point)
```

ST_Y

This function returns the Y coordinate of a Point geometry.

Syntax

```
geof:ST_Y(geom)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
double	The Y coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (MIN(?point_x) as ?3d_min)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
```

```

<http://csi.com/geologic_units_24k/shape> ?shape;
BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
BIND(geof:ST_X(?point_2d) as ?point_x)
BIND(geof:ST_Y(?point_2d) as ?point_y)
}

```

ST_SetY

This function sets the Y coordinate of a Point geometry.

Syntax

```
geof:ST_SetY(geom, y)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
y	double	The Y coordinate point value to set.

Returns

Type	Description
Custom object	The Y coordinate.

Example

```

PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (MIN(?point_y) as ?3d_min)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
  BIND(geof:ST_X(?point_2d) as ?point_x)
}

```

```
    BIND(geof:ST_Y(?point_2d) as ?point_y)
}
```

ST_Z

This function returns the Z coordinate of a Point geometry.

Syntax

```
geof:ST_Z(geom)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
double	The Z coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (MAX(geof:ST_Z(geof:ST_Point(?point_x, ?point_y, ?point_z))) as ?3d_max)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
  BIND(geof:ST_X(?point_2d) as ?point_x)
  BIND(geof:ST_Y(?point_2d) as ?point_y)
  BIND(32 as ?point_z)
}
```

ST_SetZ

This function sets the Z coordinate of a Point geometry.

Syntax

```
geof:ST_SetZ(geom, z)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
z	double	The Z coordinate point value to set.

Returns

Type	Description
Custom object	The Z coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_SetZ(geof:ST_Point(1,2),4))) as ?point_with_z_coordinate)
```

ST_M

This function returns the M coordinate of a Point geometry.

Syntax

```
geof:ST_M(geom)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
double	The M coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (MIN(geof:ST_M(geof:ST_Point(?point_x, ?point_y, ?point_z, ?point_m))) as
?result)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
  BIND(geof:ST_X(?point_2d) as ?point_x)
  BIND(geof:ST_Y(?point_2d) as ?point_y)
  BIND(25 as ?point_z)
  BIND(30 as ?point_m)
}
```

ST_SetM

This function sets the M coordinate of a Point geometry.

Syntax

```
geof:ST_SetM(geom, m)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
m	double	The M coordinate point value to set.

Returns

Type	Description
Custom object	The M coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_SetM(geof:ST_Point(1,2,3),4)) as ?point)
```

ST_Bin

This function returns the bin ID of a Point geometry.

Syntax

```
geof:ST_Bin(binSize, geom)
```

Argument	Type	Description
binSize	double	The bin size.
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
long	The bin ID.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```

SELECT (geof:ST_Bin(10, ?point_2d) as ?union)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  <http://csi.com/geologic_units_24k/id> "0"^^xsd:long.
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
}

```

ST_BinEnvelope

This function returns the bin envelope for a Point geometry or bin ID. You can also return the bin envelope for multiple Points.

Syntax

```
geof:ST_BinEnvelope(binSize, geom)
```

Argument	Type	Description
binSize	double	The bin size.
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The bin envelope.

Example

```

PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT (geof:ST_AsText(geof:ST_BinEnvelope(10.0, ?point_2d)) as ?union)
FROM <point>
WHERE {

```

```

?point a <http://csi.com/geologic_units_24k/point>;
<http://csi.com/geologic_units_24k/shape> ?shape;
<http://csi.com/geologic_units_24k/id> "0"^^xsd:long.
BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
}

```

ST_MultiPoint

This function returns a MultiPoint as a custom object constructed from a set of X and Y coordinate pairs. You can specify any number of X and Y coordinates.

Syntax

```
geof:ST_MultiPoint(x, y [, xx ] [, yy ])
```

Argument	Type	Description
x	double	The X coordinate value.
y	double	The Y coordinate value.
xx	double	Optional X coordinate values.
yy	double	Optional Y coordinate values.

Returns

Type	Description
Custom object	The multipoint.

Example

```

PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_MultiPoint(?x,?y,?x1,?y1,?x2,?y2)) as ?multipoint_pairs)
WHERE {

```

```
values (?x ?y ?x1 ?y1 ?x2 ?y2) {(1.1 7.0 2 3 5 -6.7) (-2 3.4 -7.8 9.3 5.4 -1.0)}
```

ST_PointN

This function returns the Point that is at the Nth index in a MultiPoint.

Syntax

```
geof:ST_PointN(geom, index)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
index	int	The index of the point to retrieve from the specified geometry.

Returns

Type	Description
Custom object	The point at the Nth index.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_PointN(geof:ST_GeomFromText(?x) , 2)) as ?envelope)
WHERE {
  VALUES (?x) {
    ('MULTILINESTRING M((10 10 10, 20 20 20, 10 40 30),(40 40 20, 30 30 30, 40 20 10, 30 10 20))')
    ('MULTILINESTRING M((10 10 20, 20 20 30, -10 -40 -30),(10 40 20, 30 30 30, 10 20 10, 30 10 20))')
```

```
}  
}
```

ST_GeomFromText

This function returns a Point as a custom object constructed from a set of well known free text (WKT) coordinates.

Syntax

```
geof:ST_GeomFromText (wkt [, SRID ] )
```

Parameter	Type	Description
wkt	string	Input geometry in WKT string format. The geometry shape value can be a Point, Multipoint, LineString, MultiLineString, Polygon, or Multipolygon.
SRID	int	Optional well known reference ID. If not specified, the default SRID is 0.

Returns

Type	Description
Custom object	The point.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ST_AsText(geof:ST_MultiLineString(geof:ST_GeomFromText(?l1),geof:ST_GeomFromText(?l2))) as ?MultiLineString)  
WHERE {  
  VALUES (?l1 ?l2) {  
    ('LINESTRING (8 7, 7 8)' 'LINESTRING (1 7, 7 8)')  
    ('LINESTRING (18 17, -17 8)' 'LINESTRING (-1 7, 7 -8)')  
  }
```

```
}  
}
```

ST_GeomFromWKB

This function returns a Point as a custom object constructed from a set of well known binary (WKB) coordinates.

Syntax

```
geof:ST_GeomFromWKB(wkb_hexstr [, wkid ] )
```

Parameter	Type	Description
wkb_hexstr	string	Input geometry in WKB Hex String format. The geometry shape value can be a Point, Multipoint, LineString, MultiLineString, Polygon, or Multipolygon.
wkid	int	Optional well known reference ID. If not specified, the default is 0.

Returns

Type	Description
Custom object	The point.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ST_AsText(geof:ST_GeomFromWKB(?val)) as ?geom)  
WHERE {  
  VALUES (?val) { ('00000000001400000000000000004010000000000000') }  
}
```

LineString and MultiLineString Functions

- [ST_LineString](#): Constructs a LineString from a number of Points in an array.
- [ST_StartPoint](#): Returns the start point of a line.
- [ST_EndPoint](#): Returns the end point of a line.
- [ST_IsClosed](#): Determines whether a line geometry is a ring.
- [ST_IsRing](#): Determines whether a geometry is closed.
- [ST_MultiLineString](#): Returns a MultiLine geometry constructed from a list of Line geometries.

ST_LineString

This function returns a LineString as a custom object constructed from a number of Points. Any number of Point geometries can be specified to form a new line.

Syntax

```
geof:ST_LineString(point1 [, ..., pointN])
```

Argument	Type	Description
point1–N	Object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The LineString.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_LineString(geof:ST_GeomFromText(?pt1), geof:ST_GeomFromText(?pt2))) as ?line_from_points)
```

```
WHERE {
  VALUES (?pt1 ?pt2) {
    ('Point(1 1)' 'Point(-1 -2)') ('Point Z(-11 1 4)' 'Point ZM(0 9.2 -1 -2)')
  }
}
```

ST_StartPoint

This function returns the start point of a given line.

Syntax

```
geof:ST_StartPoint(line)
```

Argument	Type	Description
line	Object	LineString geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The start point of the line.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsGeoJSON(geof:ST_StartPoint(geof:ST_GeomFromText(?line_str_wkt))) as
?start_point) (geof:ST_AsGeoJSON(geof:ST_EndPoint(geof:ST_GeomFromText(?line_str_wkt)))
as ?end_point)
FROM <linestring>
WHERE {
  ?ln_str_ins a <http://csi.com/road_centerline/linestring>;
  <http://csi.com/road_centerline/shape> ?line_str_wkt;
  <http://csi.com/road_centerline/no_of_points> ?no_pt_ln.
}
ORDER BY desc(?no_pt_ln) desc(?line_str_wkt)
LIMIT 1
```


ST_EndPoint

This function returns the end point of a given line.

Syntax

```
geof:ST_EndPoint(line)
```

Argument	Type	Description
line	Object	LineString geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The end point of the line.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsGeoJSON(geof:ST_StartPoint(geof:ST_GeomFromText(?line_str_wkt))) as
?start_point)
      (geof:ST_AsGeoJSON(geof:ST_EndPoint(geof:ST_GeomFromText(?line_str_wkt))) as
?end_point)
FROM <linestring>
WHERE {
  ?ln_str_ins a <http://csi.com/road_centerline/linestring>;
  <http://csi.com/road_centerline/shape> ?line_str_wkt;
  <http://csi.com/road_centerline/no_of_points> ?no_pt_ln.
}
ORDER BY desc(?no_pt_ln) desc(?line_str_wkt)
LIMIT 1
```

ST_IsClosed

This function evaluates whether a geometry is closed.

Syntax

```
geof:ST_IsClosed(line)
```

Argument	Type	Description
line	Object	LineString or MultiLineString geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
boolean	True if the geometry is closed. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsClosed(geof:ST_GeomFromText(?x)) as ?ring)
WHERE {
  VALUES (?x) {
    ('LINESTRING (30 10, 10 10, 40 40, 30 10)') ('LINESTRING Z(-30.56 12 45, 10 1 -5, -
67 -0.56 68, -30.56 12 45)')
  }
}
```

ST_IsRing

This function evaluates whether a Line geometry is a ring.

Syntax

```
geof:ST_IsRing(line)
```

Argument	Type	Description
line	Object	LineString geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
boolean	True if the line is a ring. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsRing(geof:ST_GeomFromText(?x,0)) as ?ring_simple)
WHERE {
  VALUES (?x) {
    ('LINESTRING (0 0, 1 1, 1 2, 2 1, 1 1, 0 0)')
    ('LINESTRING (0 0, 3 4)')
  }
}
```

ST_MultiLineString

This function returns the MultiLine geometry constructed from a list of Line geometries. Any number of line geometries can be specified to form a new MultiLine.

Syntax

```
geof:ST_MultiLineString(line1 [, ..., lineN])
```

Argument	Type	Description
line1–N	Object	LineString geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The multiline geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_MultiLineString(geof:ST_GeomFromText(?l1),geof:ST_GeomFromText(?l2))) as ?MultiLineString)
WHERE {
  VALUES (?l1 ?l2) {
    ('LINESTRING (8 7, 7 8)' 'LINESTRING (1 7, 7 8)')
    ('LINESTRING (18 17, -17 8)' 'LINESTRING (-1 7, 7 -8)')
  }
}
```

Polygon Functions

- [ST_Polygon](#): Returns a Polygon constructed from multiple Point geometries.
- [ST_ExteriorRing](#): Returns the exterior ring of a Polygon.
- [ST_NumInteriorRing](#): Returns the number of interior rings that exist in a Polygon.
- [ST_InteriorRingN](#): Returns the interior ring at the Nth index position in a Polygon.
- [ST_MultiPolygon](#): Returns a MultiPolygon constructed from a list of Polygons.
- [ST_Circle](#): Returns a Circle Polygon from a radius and x, y, and z coordinates.
- [ST_CircleArc](#): Returns a Circle Arc Polygon from a radius and Point geometry.
- [ST_Arc](#): Returns an Arc line from a start angle and size angle.
- [ST_Ellipse](#): Returns an Ellipse Polygon from a radius and Point.
- [ST_EllipticalArc](#): Returns an Elliptical Arc Polygon from a radius and Point.
- [ST_Rectangle](#): Returns a Rectangle Polygon from a height, width, and Point.

- [ST_Squircle](#): Returns a Squircle Polygon from a radius and Point.
- [ST_GeomFromGML](#): Returns geometry based on a GML specification.

ST_Polygon

This function returns Polygon geometry constructed from multiple Point geometries.

Syntax

```
geof:ST_Polygon(point1 [, ..., pointN])
```

Argument	Type	Description
point1–N	Object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. Any number of point geometries can be passed as input to form a new polygon but all points should have the same dimensions. That is, the specified input geometries should all be either 2D point geometries or 3D point geometries.

Returns

Type	Description
Custom object	The polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Polygon(geof:ST_Point(?x,?y),geof:ST_Point(?x1,?y1),
                                     geof:ST_Point(?x2,?y2))) as ?line_from_points)
WHERE {
  VALUES (?x ?y ?x1 ?y1 ?x2 ?y2) {
    (1.1 7.0 2 3 5 -6.7)
    (-2 3.4 -7.8 9.3 5.4 -1.0)
  }
}
```

ST_ExteriorRing

This function returns the exterior ring of a given Polygon geometry.

Syntax

```
geof:ST_ExteriorRing (polygon)
```

Argument	Type	Description
polygon	Object	Polygon geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The exterior ring.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_ExteriorRing('POLYGON((0 0,0 3,3 3,3 0,0 0),(1 1,1 2,2 2,2 1,1 1)')))) as ?exterior_ring)
```

ST_NumInteriorRing

This function returns the number of interior rings that exist in a given Polygon.

Syntax

```
geof:ST_NumInteriorRing (polygon)
```

Argument	Type	Description
polygon	Object	Polygon geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
int	The number of interior rings.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
SELECT (geof:ST_NumInteriorRing(geof:ST_GeomFromText(?multi_pt_wkt)) as ?polygon)
FROM <polygon>
WHERE {
  ?multi_pt a <http://csi.com/national_weather_service_wind_gust_forecast/polygon>;
  <http://csi.com/national_weather_service_wind_gust_forecast/shape> ?multi_pt_wkt;
  <http://csi.com/national_weather_service_wind_gust_forecast/id> "0"^^xsd:int.
}
```

ST_InteriorRingN

This function returns the interior ring at the Nth index position in a Polygon.

Syntax

```
geof:ST_InteriorRingN(polygon, n)
```

Argument	Type	Description
polygon	Object	Polygon geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
n	int	The index of the interior ring to retrieve from the specified polygon.

Returns

Type	Description
Custom object	The interior ring.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_InteriorRingN(geof:ST_GeomFromText(?x),1)) as
?interiorRing)
WHERE {
  VALUES (?x) {
    ('POLYGON((0 0,0 3,3 3,3 0,0 0),(1 1,1 2,2 2,2 1,1 1))')
    ('POLYGON((2 2, 6 4, 6 10, -2 10, -2 4, 2 2),(1 5, 3 5, 3 7, 1 7))')
  }
}
```

ST_MultiPolygon

This function returns MultiPolygon constructed from a list of Polygons. Any number of Polygon geometries can be specified as input.

Syntax

```
geof:ST_MultiPolygon(polygon1 [, ..., polygonN])
```

Argument	Type	Description
polygon1–N	Object	Polygon geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	The multipolygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_MultiPolygon(geof:ST_GeomFromText(?p1),geof:ST_GeomFromText(?p2))) as ?agg_multipolygon)
WHERE {
  VALUES (?p1 ?p2) {
    ('POLYGON ((2 0, 2 3, 3 0))' 'POLYGON ((11 12,-11 -12,11 -12))')
    ('POLYGON ((1 2,-2 -3,9.2 -4.5))' 'POLYGON ((-1 -2, -4.5 -17, -5.6 -78))')
  }
}
```

ST_Circle

This function returns a Circle Polygon from a given radius and Point.

Syntax

```
geof:ST_Circle(geom, radius, numOfPoints)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
radius	double	The radius of the circle polygon.
numOfPoints	int	The number of points to use to represent the new circle polygon.

Returns

Type	Description
Custom object	The circle polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Circle((geof:ST_Point(1,2)),3,10)) as ?circle_polygon)
```

ST_CircleArc

This function returns a Circular Arc Polygon from the given radius parameters and Point arguments.

Syntax

```
geof:ST_CircleArc(geom, startRad, sizeRad, radius, numOfPoints)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
startRad	double	The start angle in radians.
sizeRad	double	The size of the angle in radians.
radius	double	The radius of the circular arc polygon.
numOfPoints	int	The number of points to use to represent the new polygon.

Returns

Type	Description
Custom object	The circular arc polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_CircleArc((geof:ST_Point(1,2)),5,10,3,10)) as ?circle_
```

```
arc_polygon)
```

ST_Arc

This function returns an Arc given a center point, radius, start and size angle, and number of Points arguments.

Syntax

```
geof:ST_Arc(x, y, radius, startRad, sizeRad, numOfPoints)
```

Argument	Type	Description
x	double	The X coordinate of the center used to draw the arc.
y	double	The Y coordinate of the center used to draw the arc.
radius	double	The radius of the arc.
startRad	double	The start angle in radians.
sizeRad	double	The size of the angle in radians.
numOfPoints	int	The number of points to use to represent the new arc.

Returns

Type	Description
Custom object	The arc.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ST_AsText(geof:ST_Arc(1,2,5,10,20,4)) as ?Arc)
```

ST_Ellipse

This function returns an Ellipse Polygon based on a given radius and Point geometry.

Syntax

```
geof:ST_Ellipse(geom, xRadius, yRadius, numOfPoints)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
xRadius	double	The X radius for the ellipse polygon.
yRadius	double	The Y radius for the ellipse polygon.
numOfPoints	int	The number of points to use to represent the new ellipse polygon.

Returns

Type	Description
Custom object	The ellipse polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Ellipse((geof:ST_Point(1,2)),2,3,10)) as ?ellipse_
polygon)
```

ST_EllipticalArc

This function returns an Elliptical Arc Polygon from the given radius and Point geometry.

Syntax

```
geof:ST_EllipseArc(geom, startRad, sizeRad, width, height, numOfPoints)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
startRad	double	The start angle in radians.
sizeRad	double	The size of the angle in radians.
width	double	The width of the elliptical arc.
height	double	The height of the elliptical arc.
numOfPoints	int	The number of points to use to represent the new elliptical arc polygon.

Returns

Type	Description
Custom object	The elliptical arc polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_EllipticalArc((geof:ST_Point(1,2)),5,10,3,10,10)) as
?elliptical_arc_polygon)
```

ST_Rectangle

This function returns a Rectangle Polygon from a given height, width, and Point geometry.

Syntax

```
geof:ST_Rectangle(geom, width, height, numOfPoints)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
width	double	The width of the rectangle.
height	double	The height of the rectangle.
numOfPoints	int	The number of points to use to represent the new rectangle polygon.

Returns

Type	Description
Custom object	The rectangle polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Rectangle((geof:ST_Point(1,2)),3,2,10)) as ?rectangle_
polygon)
```

ST_Squircle

This function returns a Squircle Polygon from a given radius and Point geometry.

Syntax

```
geof:ST_Squircle(geom, radius, numOfPoints)
```

Argument	Type	Description
geom	object	Point geometry in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.
raduis	double	The radius of the squircle polygon.
numOfPoints	int	The number of points to use to represent the new squircle polygon.

Returns

Type	Description
Custom object	The squircle polygon.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Squircle((geof:ST_Point(1,2)),3,10)) as ?squircle_
polygon)
```

ST_GeomFromGML

This function returns geometry based on a GML specification.

Syntax

```
geof:ST_GeomFromGML(gml)
```

Parameter	Type	Description
gml	string	Input geometry in GMLLiteral format. The geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, or Multipolygon.

Returns

Type	Description
Custom object	The geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_GeomFromGML(geof:ST_AsGML(geof:ST_GeomFromText('POINT(1 2)')))) as ?geometryGml)
```

Utility Functions

- [ST_MinX](#): Returns the minimum value of the X coordinate for a geometry.
- [ST_MaxX](#): Returns the maximum value of the X coordinate for a geometry.
- [ST_MinY](#): Returns the minimum value of the Y coordinate for a geometry.
- [ST_MaxY](#): Returns the maximum value of the Y coordinate for a geometry.
- [ST_MinZ](#): Returns the minimum value of Z coordinate for a geometry.
- [ST_MaxZ](#): Returns the maximum value of Z coordinate for a geometry.
- [ST_MinM](#): Returns the minimum measure value for a geometry.
- [ST_MaxM](#): Returns the maximum measure value for a geometry.

Serialization

- [ST_AsGeoJSON](#): Returns a GeoJSON representation of a shape.
- [ST_AsJSON](#): Returns a JSON representation of a shape.
- [ST_AsBinary](#): Returns a binary object from a geometry.
- [ST_AsText](#): Returns a well known text (WKT) representation of a shape.
- [ST_AsGML](#): Returns geometry from a GML representation of a shape.

- [ST_AsWktLiteral](#): Returns a URL from WKT representation of a geometry.
- [ST_AsGmlLiteral](#): Returns a URL from GML representation of a geometry.
- [ST_AsWKB_HEX](#): Returns a hex string from WKB representation of a geometry.

Logical, Comparison, and Relational Operations

- [ST_Is3D](#): Determines if a geometry object is three-dimensional.
- [ST_IsEmpty](#): Determines if a geometry object is empty.
- [ST_IsMeasured](#): Determines if a geometry object is measured.
- [ST_IsSimple](#): Determines if a geometry object is simple.
- [ST_Crosses](#): Determines if geometry1 crosses geometry2.
- [ST_Contains](#): Determines if geometry1 contains geometry2.
- [ST_Disjoint](#): Determines if geometry1 has any Points in common with geometry2.
- [ST_Equals](#): Determines if geometry1 equals geometry2.
- [ST_Intersects](#): Determines if geometry1 intersects geometry2.
- [ST_Overlaps](#): Determines if geometry1 overlaps geometry2.
- [ST_Touches](#): Determines if geometry1 touches geometry2.
- [ST_Within](#): Determines if geometry1 is within geometry2.
- [ST_EnvIntersects](#): Determines if the envelopes of geometry1 and geometry2 intersect.
- [ST_Relate](#): Determines if geometry1 has the specified DE-9IM relationship with geometry2.

Conversion, Calculation, and Translation

- [ST_Distance](#): Returns the distance between two geometry objects.
- [ST_Boundary](#): Returns the boundary of a given geometry.
- [ST_Intersection](#): Returns the intersection of two geometry objects.
- [ST_Difference](#): Returns the difference between two geometry objects.

- [ST_SymDifference](#): Returns the symmetric difference between two geometry objects.
- [ST_SRID](#): Returns the spatial reference ID of a geometry.
- [ST_SetSRID](#): Sets the spatial reference ID of a geometry and returns its coordinates.
- [ST_Dimension](#): Returns the spatial dimension of a geometry.
- [ST_Length](#): Returns the length of a Line.
- [ST_Area](#): Returns the area of a Polygon or MultiPolygon.
- [ST_CoordDim](#): Returns the count of coordinate components.
- [ST_Envelope](#): Returns the envelope of a geometry.
- [ST_GeometryType](#): Returns the geometry type of a geometry.
- [ST_Union](#): Returns the union of one or more geometries.
- [ST_NumPoints](#): Returns the number of Points in a geometry.
- [ST_NumGeometries](#): Returns the number of geometries in a multi-geometry shape.
- [ST_GeometryN](#): Returns the Nth geometry in a multi-geometry shape.
- [ST_Centroid](#): Returns the centroid of a geometry.
- [ST_Buffer](#): Returns the geometry buffered by distance.
- [ST_ConvexHull](#): Returns the convex hull of one or more geometries.
- [ST_GeodesicLengthWGS84](#): Returns the distance (in meters) along a Line of a WGS84 spheroid for geographic coordinates.
- [ST_GeomFromJSON](#): Returns the geometry from a JSON representation of a shape.
- [ST_GeomFromGeoJSON](#): Returns the geometry from a GeoJSON representation of a shape.
- [ST_LatLonFromDMSToDD](#): Returns latitude and longitude in decimal degrees.
- [ST_GeomFromWktLiteral](#): Returns geometry object from a WKT representation of a shape.
- [ST_GeomFromGmlLiteral](#): Returns geometry object from a GML representation of a shape.

- [ST_ConvertCoordinates](#): Return coordinates converted from one coordinate system to another.
- [ST_IsValidWKT](#): Validates whether a given WKT representation is correct.
- [ST_IsValidGeoJSON](#): Validates whether a given geoJSON representation is correct.
- [ST_Translate](#): Transforms a geometry with given shift values.
- [ST_Scale](#): Scales a geometry to a new size using scale factor arguments.
- [ST_Shear](#): Shears a geometry around an axis using shearing arguments.
- [ST_Rotate](#): Rotates geometry around the origin using rotation arguments.

ST_MinX

This function returns the minimum value of the X coordinate for a geometry.

Syntax

```
geof:ST_MinX (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The minimum value of the X coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_MinX(?z1) as ?minX)
WHERE {
  SELECT (geof:ST_Point(?x, ?y, ?z, ?m) as ?z1)
  WHERE {
    VALUES (?x ?y ?z ?m) {
      (1.1 2.2 3.2 1) (1.2 3.1 1.1 2) (8.2 3.2 9.2 3)
    }
  }
}
```

ST_MaxX

This function returns the maximum value of the X coordinate for a geometry.

Syntax

```
geof:ST_MaxX(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The maximum value of the X coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT ?no_pt_ln ?xmin ?xmax ?ymin ?ymax
FROM <multipoint>
WHERE {
  ?ln_str_ins a <http://csi.com/wind_speed_at_block_group_level/multipoint>;
  <http://csi.com/wind_speed_at_block_group_level/shape> ?multi_pt_wkt;
  <http://csi.com/wind_speed_at_block_group_level/no_of_points> ?no_pt_ln;
  BIND(geof:ST_GeomFromText(?multi_pt_wkt) as ?geo_shp)
  BIND(geof:ST_MinX(?geo_shp) as ?xmin)
  BIND(geof:ST_MaxX(?geo_shp) as ?xmax)
  BIND(geof:ST_MinY(?geo_shp) as ?ymin)
  BIND(geof:ST_MaxY(?geo_shp) as ?ymax)
}
ORDER BY desc(?no_pt_ln)
LIMIT 1
```

ST_MinY

This function returns the minimum value of the Y coordinate for a geometry.

Syntax

```
geof:ST_MinY(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The minimum value of the Y coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT ?no_pt_ln ?xmin ?xmax ?ymin ?ymax
FROM <multipoint>
WHERE {
  ?ln_str_ins a <http://csi.com/wind_speed_at_block_group_level/multipoint>;
  <http://csi.com/wind_speed_at_block_group_level/shape> ?multi_pt_wkt;
  <http://csi.com/wind_speed_at_block_group_level/no_of_points> ?no_pt_ln;
  BIND(geof:ST_GeomFromText(?multi_pt_wkt) as ?geo_shp)
  BIND(geof:ST_MinX(?geo_shp) as ?xmin)
  BIND(geof:ST_MaxX(?geo_shp) as ?xmax)
  BIND(geof:ST_MinY(?geo_shp) as ?ymin)
  BIND(geof:ST_MaxY(?geo_shp) as ?ymax)
}
ORDER BY desc(?no_pt_ln)
LIMIT 1
```

ST_MaxY

This function returns the maximum value of the Y coordinate for a geometry.

Syntax

```
geof:ST_MaxY(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or

Parameter	Type	Description
GeometryCollection.		

Returns

Type	Description
double	The maximum value of the Y coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT ?no_pt_ln ?xmin ?xmax ?ymin ?ymax
FROM <multipoint>
WHERE {
  ?ln_str_ins a <http://csi.com/wind_speed_at_block_group_level/multipoint>;
  <http://csi.com/wind_speed_at_block_group_level/shape> ?multi_pt_wkt;
  <http://csi.com/wind_speed_at_block_group_level/no_of_points> ?no_pt_ln;
  BIND(geof:ST_GeomFromText(?multi_pt_wkt) as ?geo_shp)
  BIND(geof:ST_MinX(?geo_shp) as ?xmin)
  BIND(geof:ST_MaxX(?geo_shp) as ?xmax)
  BIND(geof:ST_MinY(?geo_shp) as ?ymin)
  BIND(geof:ST_MaxY(?geo_shp) as ?ymax)
}
ORDER BY desc(?no_pt_ln)
LIMIT 1
```

ST_MinZ

This function returns the minimum value of the Z coordinate for a geometry.

Syntax

```
geof:ST_MinZ(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The minimum value of the Z coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_MinZ(geof:ST_GeomFromText("LINESTRING Z(10 12 1, 22 20 2, 1 40 3)")) as
?minZ)
```

ST_MaxZ

This function returns the maximum value of the Z coordinate for a geometry.

Syntax

```
geof:ST_MaxZ(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The maximum value of the Z coordinate.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_MaxZ(geof:ST_GeomFromText("LINESTRING Z(10 12 1, 22 20 2, 1 40 3)")) as
?maxZ)
```

ST_MinM

This function returns the minimum value of measure for a geometry.

Syntax

```
geof:ST_MinM(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The minimum value of measure.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_MinM(geof:ST_GeomFromText("LINESTRING ZM(10 12 1 2.1, 22 20 2 1.2, 1 40
3 4.3)")) as ?maxZ)
```

ST_MaxM

This function returns the maximum value of measure for a geometry.

Syntax

```
geof:ST_MaxM(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The maximum value of measure.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_MaxM(geof:ST_GeomFromText("LINESTRING ZM(10 12 1 2.1, 22 20 2 1.2, 1 40
3 4.3)")) as ?maxZ)
```

ST_AsGeoJSON

This function returns a GeoJSON representation of a geometric shape.

Syntax

```
geof:ST_AsGeoJSON (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
string	The GeoJSON representation of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsValidGeoJSON(geof:ST_AsGeoJSON(geof:ST_GeomFromText('LINESTRING(4 6,7 10)')))) as ?is_valid_geojson)
```

ST_AsJSON

This function returns a JSON representation of a geometric shape.

Syntax

```
geof:ST_AsJSON (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data

Parameter	Type	Description
		type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
string	The JSON representation of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_GeomFromJSON(geof:ST_AsJSON(geof:ST_GeomFromText(?x))))
as ?line_from_json)
WHERE {
  VALUES (?x) {
    ('LINESTRING (0 0, 2 2)') ('LINESTRING Z(8 -7 -1, -7 -1 8)')
  }
}
```

ST_AsBinary

This function returns a binary object from a given geometry.

Syntax

```
geof:ST_AsBinary(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or

Parameter	Type	Description
GeometryCollection.		

Returns

Type	Description
Custom object	The binary object.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsBinary(geof:ST_GeomFromText(?x)) as ?binary)
WHERE {
  VALUES (?x) {
    ('GeometryCollection(Point(1 1),LineString(2 2, 3 3),Point(4 0))')
  }
}
```

ST_AsText

This function returns the well known text (WKT) representation of a geometric shape.

Syntax

```
geof:ST_AsText (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
string	The WKT representation of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Circle((geof:ST_Point(1,2)),3,10)) as ?circle_polygon)
```

ST_AsGML

This function returns geometry from the GML representation of a shape.

Syntax

```
geof:ST_AsGML(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
string	The geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsGML(geof:ST_Point(1.1,-6.7)) as ?point)
```

ST_AsWktLiteral

This function returns a URI from a well known text (WKT) representation of a shape.

Syntax

```
geof:ST_AsWktLiteral (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The WKT URI representation of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfContains(?x,geof:ST_AsWktLiteral(geof:ST_GeomFromText(?y))) as ?url)
WHERE {
  VALUES (?x ?y) {
    ('<http://www.opengis.net/def/crs/OGC/1.3/CRS84>POLYGON ((1 1, 1 4, 4 4, 4 1))'
    'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('<http://www.opengis.net/def/crs/OGC/1.3/CRS84>POLYGON ((1 1, 1 4, 4 4, 4 1))'
    'Point (1 2)')
```

```
}  
}
```

ST_AsGmlLiteral

This function returns a URI from the GML representation of a shape.

Syntax

```
geof:ST_AsGmlLiteral (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The GmlLiteral URI representation of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
SELECT (geof:sfContains(?x,geof:ST_AsGmlLiteral(?y)) as ?url)  
WHERE {  
  VALUES (?x ?y) {  
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4 1))'  
'<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4  
1))'^^geo:wktLiteral)  
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4 1))'  
'<http://www.opengis.net/def/crs/EPSG/0/4326>Point (1 2)'^^geo:wktLiteral)
```



```
}  
}
```

ST_AsWKB_HEX

This function returns a hex string from the WKB representation of a geometry.

Syntax

```
geof:ST_AsWKB_HEX (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
string	The hex string.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ST_AsWKB_HEX(geof:ST_GeomFromText(?x)) as ?hex)  
WHERE {  
  VALUES (?x) {  
    ('POLYGON ((2 0, 2 1, 3 1))')  
  }  
}
```

ST_Is3D

This function evaluates whether the specified geometry object is three-dimensional.

Syntax

```
geof:ST_Is3D(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the geometry is 3D. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Is3D(geof:ST_Point(?x,?y,?z)) as ?is3DGeometry)
WHERE {
  VALUES (?x ?y ?z) {
    (1.1 2.2 3.2) (1.2 3.1 -1.1) (-0.6 8.2 3.2)
  }
}
```

ST_IsEmpty

This function evaluates whether the specified geometry object is empty.

Syntax

```
geof:ST_IsEmpty(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the geometry is empty. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsEmpty(geof:ST_GeomFromText(?x)) as ?empty)
WHERE {
  VALUES (?x) {('point empty')}
}
```

ST_IsMeasured

This function evaluates whether a given geometry object is measured.

Syntax

```
geof:ST_IsMeasured(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the geometry is measured. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsMeasured(geof:ST_Point(?x,?y,?z,?m)) as ?measured)
WHERE {
  VALUES (?x ?y ?z ?m) {
    (1.1 2.2 3.2 3)
    (1.2 3.1 -1.1 1)
    (2 -0.6 8.2 3.2)
  }
}
```

ST_IsSimple

This function evaluates whether a given geometry object is simple.

Syntax

```
geof:ST_IsSimple(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the geometry is simple. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsSimple(geof:ST_Point(?x, ?y)) as ?simple)
WHERE {
  VALUES (?x ?y) {
    (2 3)
    (-1000 -234234)
  }
}
```

ST_Crosses

This function evaluates whether geometry1 crosses geometry2.

Syntax

```
geof:ST_Crosses(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry crosses the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Crosses(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as ?crosses)
WHERE {
  VALUES (?x ?y) {('LINESTRING (0 0, 1 1)' 'LINESTRING (1 0, 0 1)')
                  ('LINESTRING (0 2, 0 1)' 'LINESTRING (2 0, 1 0)')}
}
```

ST_Contains

This function evaluates whether geometry1 contains geometry2.

Syntax

```
geof:ST_Contains(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry contains the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Contains(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as
?contains)
WHERE {
  VALUES (?x ?y) {('POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))' 'Point (-
106.4453583 39.11775)')
                  ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (2 3)')
                  ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (7 8)')}
}
```

ST_Disjoint

This function evaluates whether geometry1 is disjoint with geometry2, i.e., whether geometry1 has any points in common with geometry2.

Syntax

```
geof:ST_Disjoint(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry is disjoint with the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Disjoint(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as
?disjoint)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (0 0, 0 1)' 'LINESTRING (1 1, 1 0)')
    ('LINESTRING (0 0, 0 1)' 'LINESTRING (1 0, 0 1)')
  }
}
```

ST_Equals

This function evaluates whether geometry1 equals geometry2.

Syntax

```
geof:ST_Equals (geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry equals the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Equals(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as ?equals)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (0 0, 1 1)' 'LINESTRING (1 1, 0 0)')
    ('LINESTRING (0 0, 0 1)' 'LINESTRING (1 0, 0 1)')
  }
}
```

ST_Intersects

This function determines if geometry1 intersects geometry2.

Syntax

```
geof:ST_Intersects(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry intersects the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Intersects(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as
```

```

?intersects)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (8 7, 7 8)' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('LINESTRING (2 0, 2 3)' 'POLYGON ((1 1, 4 1, 4 4, 1 4))')
  }
}

```

ST_Overlaps

This function evaluates whether geometry1 overlaps geometry2.

Syntax

```
geof:ST_Overlaps (geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry overlaps the second geometry. False if not.

Example

```

PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Overlaps(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as
?overlaps)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 0, 2 1, 1 3))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}

```

```

('POLYGON ((2 0, 2 1, 3 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((2 0, 2 3, 3 0))')
}
}

```

ST_Touches

This function evaluates whether geometry1 touches geometry2, i.e., whether they have any points in common.

Syntax

```
geof:ST_Touches (geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry touches the second geometry. False if not.

Example

```

PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Touches(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as ?touches)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))' 'Point (-106.4453583 39.11775)')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (1 2)')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (7 8)')
  }
}

```

```
}  
}
```

ST_Within

This function evaluates whether geometry1 is within geometry2.

Syntax

```
geof:ST_Within(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the first geometry is within the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ST_Within(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as ?is_point_within_polygon)  
WHERE {  
  VALUES (?x ?y) {  
    ('Point (-106.4453583 39.11775)' 'POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')  
    ('Point (2 3)' 'POLYGON ((1 1, 1 4, 4 4, 4 1))' )  
    ('Point (7 8)' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')  
  }  
}
```

ST_EnvIntersects

This function evaluates whether the envelopes of geometry1 and geometry2 intersect.

Syntax

```
geof:ST_EnvIntersects(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the envelopes of geometry1 and geometry2 intersect. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_EnvIntersects(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as
?env_intersects)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (0 0, 1 1)' 'LINESTRING (1 3, 2 2)')
    ('LINESTRING (0 0, 2 2)' 'LINESTRING (1 0, 3 2)')
  }
}
```

ST_Relate

This function evaluates whether geometry1 has the specified DE-9IM relationship with geometry2.

Syntax

```
geof:ST_Relate(geom1, geom2, "pattern_matrix")
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
pattern_ matrix	string	Represents a DE-9IM intersection pattern consisting of T (true) and F (false) values. For example, "T*F***FF*".

Returns

Type	Description
boolean	True if the first geometry has the specified DE-9IM relationship with the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Relate(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y),?z) as
?relate)
WHERE {
  VALUES (?x ?y ?z) {
    ('LINESTRING (0 0, 3 3)' 'LINESTRING (1 1, 4 4)' 'T*****')
    ('LINESTRING (0 0, 3 3)' 'LINESTRING (1 1, 4 4)' '*****T****')
  }
}
```

ST_Distance

This function returns the distance between two geometry objects.

Syntax

```
geof:ST_Distance(geom1, geom2 [ , units ] [, isSpherical ])
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
units	URI	Optional Unit URI having one of the following values: uom:minute, uom:second, uom:centimeter, uom:centimetre, uom:degree, uom:foot, uom:grad, uom:inch, uom:kilometer, uom:kilometre, uom:meter, uom:metre, uom:microRadian, uom:mile, uom:millimeter, uom:millimetre, uom:nauticalMile, uom:radian, uom:statuteMile, uom:surveyFootUS, or uom:yard.
isSpherical	boolean	This flag allows the ST_Distance function to explicitly control computation of the shortest spherical distance between two geometries. If isSpherical is true, the function computes the shortest spherical distance between the two geometries using only the geom1 SRID to detect axis order. The function then performs spherical calculation based on the Earth mean radius (6,371,008.7714).

Returns

Type	Description
double	The distance between the geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
SELECT (geof:ST_Distance(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y)) as
?distance)
WHERE {
  VALUES (?x ?y) {
    ('Point (0 0)' 'Point (3 4)')
    ('Point (0 0)' 'Point (2 3)')
  }
}
```

ST_Boundary

This function returns the boundary of a given geometry.

Syntax

```
geof:ST_Boundary (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The boundary of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Boundary(geof:ST_GeomFromText(?x))) as ?boundary)
WHERE {
  VALUES (?x) {
    ('POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')
    ('POLYGON ((1 1, 1 4, 4 1))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}
```

ST_Intersection

This function returns the intersection of two geometry objects.

Syntax

```
geof:ST_Intersection(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The intersection between the geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Intersection(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y))) as ?intersection)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (8 7, 7 8)' 'LINESTRING (2 0, 2 3)')
    ('LINESTRING (0 2, 0 0, 2 0)' 'LINESTRING (0 3, 0 1, 1 0, 3 0)')
  }
}
```

ST_Difference

This function returns the difference between two geometry objects.

Syntax

```
geof:ST_Difference(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The difference between the geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Difference(geof:ST_GeomFromText(?x),geof:ST_GeomFromText
(?y))) as ?difference)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))' 'POLYGON ((30 10 , 40 40, 20 40,
10 20, 30 10))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')
    ('POLYGON ((0 0, 0 10, 10 10, 10 0))' 'POLYGON ((0 0, 0 5, 5 5, 5 0))')
    ('POLYGON ((1 1,4 1, 4 4,1 4))' 'POLYGON ((2 0, 2 1, 3 1))')
  }
}
```

ST_SymDifference

This function returns the symmetric difference between two geometry objects.

Syntax

```
geof:ST_SymDifference(geom1, geom2)
```

Parameter	Type	Description
geom1, geom2	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The symmetric difference.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_SymDifference(geof:ST_GeomFromText(?x),geof:ST_GeomFromText(?y))) as ?difference)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))' 'POLYGON ((1 1, 3 1, 3 3, 1 3, 1 1))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')
    ('POLYGON ((0 0, 0 10, 10 10, 10 0))' 'POLYGON ((0 0, 0 5, 5 5, 5 0))')
    ('POLYGON ((1 1,4 1, 4 4,1 4))' 'POLYGON ((2 0, 2 1, 3 1))')
  }
}
```

ST_SRID

This function returns the spatial reference ID of a geometry.

Syntax

```
geof:ST_SRID(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
int	The SRID.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_SRID(geof:ST_SetSRID(geof:ST_GeomFromText(?x) , 4326)) as ?SRID)
WHERE {
  VALUES (?x) {
    ('Point (1.5 2.5)') ('Point (23.45 -78.90)')
  }
}
```

ST_SetSRID

This function sets the spatial reference ID of a geometry and returns its coordinates.

Syntax

```
geof:ST_SetSRID(geom, wkid)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
wkid	int	The well known reference ID.

Returns

Type	Description
Custom object	The coordinates.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_SRID(geof:ST_SetSRID(geof:ST_GeomFromText(?x) , 4326)) as ?SRID)
WHERE {
  VALUES (?x) {
    ('Point (1.5 2.5)') ('Point (23.45 -78.90)')
  }
}
```

ST_Dimension

This function returns the spatial dimension of a given geometry.

Syntax

```
geof:ST_Dimension(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
int	The spatial dimension.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Dimension(geof:ST_GeomFromText(?x)) as ?Dimension)
WHERE {
```

```
VALUES (?x) {
  ('Point (1.5 2.5)') ('Point (23.45 -78.90)')
}
}
```

ST_Length

This function returns the length of a Line.

Syntax

```
geof:ST_Length(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The length of the shape.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Length(geof:ST_GeomFromText(?x)) as ?geometry_length)
WHERE {
  VALUES (?x) {
    ('MULTILINESTRING ((1 1, 1 2),(10 10, 20 10))')
    ('MULTILINESTRING ((10 30, 20 20, 30 40),(40 20, 30 30, 40 20, 30 10))')
  }
}
```

ST_Area

This function returns the area of a Polygon or MultiPolygon.

Syntax

```
geof:ST_Area (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The area of geometry as follows: • The planar area is returned if a geometry is passed without an SRID. • The geodesic area is returned if a geometry is passed with SRID 4326 /CRS84. • The spherial area is returned if a geometry is passed with SRID 4047.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_Area(geof:ST_GeomFromText(?x)) as ?Area)
WHERE {
  VALUES (?x) {
    ('POLYGON ((0 0, 0 8, 8 0, 0 0), (1 1, 1 5, 5 1, 1 1))')
    ('multipolygon (((10 40, 20 10, 50 10, 60 40, 50 70, 20 70), (25 20, 45 20, 45 60, 25 60)), ((30 30, 40 30, 35 50)))')
  }
```



```
}  
}
```

ST_CoordDim

This function returns the count of coordinate components.

Syntax

```
geof:ST_CoordDim(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
double	The count of coordinate components.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ST_CoordDim(geof:ST_GeomFromText(?x)) as ?coordinate_dimension)  
WHERE {  
  VALUES (?x) {  
    ('Point (1.5 2.5)') ('Point Z(23.45 -78.90 3)')  
  }  
}
```

ST_Envelope

This function returns the envelope of a given geometry.

Syntax

```
geof:ST_Envelope (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The envelope.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText (geof:ST_Envelope (geof:ST_GeomFromText (?x))) as ?envelope)
WHERE {
  VALUES (?x) {
    ('POLYGON ((2 0, 2 3, 3 0))') ('POLYGON ((2 0, 2 1, 3 1))')
  }
}
```

ST_GeometryType

This function returns the geometry type of a geometry.

Syntax

```
geof:ST_GeometryType (geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
string	The geometry type.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_GeometryType(geof:ST_GeomFromText(?x)) as ?geometry_type)
WHERE {
  VALUES (?x) {
    ('POLYGON ((2 0, 2 3, 3 0))') ('POLYGON ((2 0, 2 1, 3 1))')
  }
}
```

ST_Union

This function returns the union of one or more geometries. Any number of geometries can be specified as input.

Syntax

```
geof:ST_Union(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data

Parameter	Type	Description
		type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The union of geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_NumPoints(geof:ST_Union(?point_2d, ?point_2d1)) as ?union)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/id> "0"^^<http://www.w3.org/2001/XMLSchema#long>;
  <http://csi.com/geologic_units_24k/shape> ?shape.
  ?point1 a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/id> "1"^^<http://www.w3.org/2001/XMLSchema#long>;
  <http://csi.com/geologic_units_24k/shape> ?shape1;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
  BIND(geof:ST_GeomFromText(?shape1) as ?point_2d1)
}
```

ST_NumPoints

This function returns the number of Points in a geometry.

Syntax

```
geof:ST_NumPoints(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
int	The number of points.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_NumPoints(geof:ST_Union(?point_2d, ?point_2d1)) as ?union)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/id> "0"^^<http://www.w3.org/2001/XMLSchema#long>;
  <http://csi.com/geologic_units_24k/shape> ?shape.
  ?point1 a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/id> "1"^^<http://www.w3.org/2001/XMLSchema#long>;
  <http://csi.com/geologic_units_24k/shape> ?shape1;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
  BIND(geof:ST_GeomFromText(?shape1) as ?point_2d1)
}
```

ST_NumGeometries

This function returns the number of geometries in a multi-geometry shape.

Syntax

```
geof:ST_NumGeometries(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
int	The number of geometries.

Example

```

PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
SELECT (geof:ST_NumGeometries(geof:ST_GeomFromText(?multi_pt_wkt)) as ?multi_polygon)
FROM <multipolygon>
WHERE {
  ?multi_pt a <http://csi.com/maryland_Shellfish__historic_oyster_
plantings/multipolygon>;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/shape> ?multi_pt_wkt;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/id> "0"^^xsd:int.
}
```

ST_GeometryN

This function returns the Nth geometry in a multi-geometry shape.

Syntax

```
geof:ST_GeometryN(geom, index)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
index	int	The index of the geometry to be retrieved.

Returns

Type	Description
Custom object	The Nth geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>#
SELECT (geof:ST_AsText(geof:ST_GeometryN(geof:ST_GeomFromText(?multi_pt_wkt), 1)) as
?multi_polygon)
FROM <multipolygon>
WHERE {
  ?multi_pt a <http://csi.com/maryland_Shellfish__historic_oyster_
plantings/multipolygon>;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/shape> ?multi_pt_wkt;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/id> "0"^^xsd:int.
}
```

ST_Centroid

This function returns the centroid of a given geometry.

Syntax

```
geof:ST_Centroid(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The centroid.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Centroid(geof:ST_GeomFromText(?multi_pt_wkt))) as
?multi_polygon)
FROM <multipolygon>
WHERE {
  ?multi_pt a <http://csi.com/maryland_Shellfish__historic_oyster_
plantings/multipolygon>;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/shape> ?multi_pt_wkt;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/id> "0"^^xsd:int.
}
```

ST_Buffer

This function returns the geometry buffered by distance.

Syntax

```
geof:ST_Buffer(geom, distance)
```


Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
distance	double	The distance value to use to get the buffer of geometry.

Returns

Type	Description
Custom object	Geometry buffered by distance.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT (geof:ST_AsText(geof:ST_Buffer(geof:ST_GeomFromText(?multi_pt_wkt), 1)) as
?multi_polygon)
FROM <multipolygon>
WHERE {
  ?multi_pt a <http://csi.com/maryland_Shellfish__historic_oyster_
plantings/multipolygon>;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/shape> ?multi_pt_wkt;
  <http://csi.com/maryland_Shellfish__historic_oyster_plantings/id> "0"^^xsd:int.
}
```

ST_ConvexHull

This function returns the convex hull of one or more geometries. Any number of geometries can be specified as input.

Syntax

```
geof:ST_ConvexHull(geom1 [, ..., geomN ])
```

Parameter	Type	Description
geom1–N	Object	Geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	Convex hull.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_ConvexHull(geof:ST_GeomFromText('point(0 0)'),
    geof:ST_GeomFromText('point(0 1)'), geof:ST_GeomFromText('point(1 1)'))
    as ?convex_hull)
```

ST_GeodesicLengthWGS84

This function returns the distance (in meters) along a line of a WGS84 spheroid for geographic coordinates.

Syntax

```
geof:ST_GeodesicLengthWGS84(geom)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or

Parameter	Type	Description
GeometryCollection.		

Returns

Type	Description
double	Distance in meters.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_GeodesicLengthWGS84(geof:ST_GeomFromText(?x)) as ?geodesic_length)
WHERE {
  VALUES (?x) {
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>LineString(0 0, 0.03 0.04)')
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>MultiLineString((0 80, 0.03 80.04))')
  }
}
```

ST_GeomFromJSON

This function returns geometry from a JSON representation of a shape.

Syntax

```
geof:ST_GeomFromJSON(json)
```

Parameter	Type	Description
json	string	Input geometry in JSON string format. The Geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, or Multipolygon.

Returns

Type	Description
Custom object	Geometry object.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_GeomFromJSON(geof:ST_AsJSON(geof:ST_GeomFromText(?x))))
as ?line_from_json)
WHERE {
  VALUES (?x) {
    ('LINESTRING (0 0, 2 2)') ('LINESTRING Z(8 -7 -1, -7 -1 8)')
  }
}
```

ST_GeomFromGeoJSON

This function returns geometry from a GeoJSON representation of a shape.

Syntax

```
geof:ST_GeomFromGeoJSON(geojson)
```

Parameter	Type	Description
geojson	string	Input geometry in GeoJSON string format. The Geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, or Multipolygon.

Returns

Type	Description
Custom object	Geometry object.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_GeomFromGeoJSON(geof:ST_AsGeoJSON(geof:ST_GeomFromText
(?x)))) as ?point_from_geojson)
WHERE {
  VALUES (?x) {
    ('POINT(10 40)') ('POINT Z(-2 30 -11)')
  }
}
```

ST_LatLonFromDMSToDD

This function returns the latitude and longitude in decimal degrees.

Syntax

```
geof:ST_LatLonFromDMSToDD(degrees, minutes, seconds, direction)
```

Parameter	Type	Description
degrees	double	Latitude and longitude degrees value.
minutes	double	Latitude and longitude minutes value.
seconds	double	Latitude and longitude seconds value.
direction	URI	Direction values (E/W/S/N) in URI format: <code>geof:E</code> , <code>geof:W</code> , <code>geof:S</code> , or <code>geof:N</code> .

Returns

Type	Description
double	Latitude and longitude in degrees.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_LatLonFromDMSToDD(137,24,13,geof:S) as ?longitude_decimal_degree)
```

ST_GeomFromWktLiteral

This function returns a Custom object from a WKT literal string.

Syntax

```
geof:ST_GeomFromWktLiteral (geom)
```

Parameter	Type	Description
geom	WKT Literal	Input geometry in WKT Literal format. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	Geometry object.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_GeomFromWktLiteral(?x)) as ?is_contains)
WHERE {
  VALUES (?x) {
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>Point(33.95 -83.38)'^^geo:wktLiteral)
  }
}
```

ST_GeomFromGmlLiteral

This function returns geometry from a GML representation of a shape.

Syntax

```
geof:ST_GeomFromGmlLiteral (geom)
```

Parameter	Type	Description
geom	GMLLiteral	Input geometry shape value in GMLLiteral format. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, or Multipolygon.

Returns

Type	Description
Custom object	Geometry object.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_GeomFromGmlLiteral(?x)) as ?is_contains)
WHERE {
  VALUES (?x) {
    ('<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral)
  }
}
```

ST_ConvertCoordinates

This function converts coordinates from one coordinate system to another.

Syntax

```
geof:ST_ConvertCoordinates(x, y, z, sourceSys, destSys)
```

Parameter	Type	Description
x, y, z	double	The x, y, z inputs depend on the type of conversion performed: • Cartesian to Spherical Coordinates: x, y, z • Spherical to Cartesian Coordinates: rad, latitude, longitude • Cartesian to Elliptical Coordinates: x, y, z • Elliptical to Cartesian Coordinates: latitude, longitude,height • Cartesian to Cylindrical Coordinates: x,y,z • Cylindrical to Cartesian Coordinates: radius, longitude, z • Cylindrical to Spherical Coordinates: radius, longitude, height • Spherical to Cylindrical Coordinates: radius, latitude, longitude
sourceSys, destSys	URI	String constants indicating source and destination coordinate systems. The sourceSys and destSys arguments can be passed as constants such as <code>geof: CARTESIAN</code> , <code>geof: SPHERICAL</code> , <code>geof: CYLINDRICAL</code> , and <code>geof: ELLIPTICAL</code> .

Returns

Type	Description
string	Converted coordinates.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_ConvertCoordinates(2,3,8,geof:CARTESIAN,geof:SPHERICAL) as
?transformed_point)
```

ST_IsValidWKT

This function validates whether a given WKT representation is correct.

Syntax

```
geof:ST_IsValidWKT(wkt)
```

Parameter	Type	Description
wkt	string	Geometry shape value in WKT string format. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the WKT representation is correct. False if it is not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsValidWKT('LINESTRING(4 6,7 10)') as ?is_valid_wkt)
```

ST_IsValidGeoJSON

This function validates whether a given geoJSON representation is correct.

Syntax

```
geof:ST_IsValidGeoJSON(geojson)
```

Parameter	Type	Description
geojson	string	Geometry shape value in GeoJson string format. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
boolean	True if the geoJSON representation is correct. False if it is not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_IsValidGeoJSON(geof:ST_AsGeoJSON(geof:ST_GeomFromText('LINESTRING(4 6,7 10)')))) as ?is_valid_geojson)
```

ST_Translate

This function transforms a geometry with given shift values.

Syntax

```
geof:ST_Translate(geom, x_delta, y_delta [ , z_delta ])
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
x_delta	double	Shift value for the X coordinate.

Parameter	Type	Description
y_delta	double	Shift value for the Y coordinate.
z_delta	double	Optional shift value for the Z coordinate. For 2D translations, only the x_delta and y_delta arguments are required.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Translate(geof:ST_GeomFromText('POINT Z(14 12 10)'),1.5,2.3, -13.7)) as ?transformed_point)
```

ST_Scale

This function returns Point geometry translated with given scaling factor values.

Syntax

```
geof:ST_Scale(geom, x_sf, y_sf [, z_sf])
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
x_sf	double	Scaling factor for the X coordinate.
y_sf	double	Scaling factor for the Y coordinate.
z_sf	double	Optional scaling factor for the Z coordinate. For 2D scaling, only the x_sf and y_sf arguments are required.

Returns

Type	Description
Custom object	Point geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Scale(geof:ST_GeomFromText('POINT (14 12)'),2,3)) as
?scaled_point)
```

ST_Shear

This function returns Point geometry translated with given shear values.

Syntax

```
geof:ST_Shear(geom, x_shear, y_shear [, z_shear])
```

Values can be passed as string constants like `geo:X`, `geo:Y`, or `geo:Z`. The shearing factor for x, y, and z coordinates should be passed in X, Y, and Z order.

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
x_shear	double	Shearing factor for the X coordinate. This value is the axis along which shearing is to be done.
y_shear	double	Shearing factor for the Y coordinate.

Parameter	Type	Description
z_shear	double	Optional shearing factor for the Z coordinate. For 2D shearing, only the x_shear and y_shear arguments are required.

Returns

Type	Description
Custom object	Point geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Shear(geof:ST_GeomFromText('POINT (1 1)'),geof:X,2,2))
as ?scaled_point)
```

ST_Rotate

This function returns Point geometry translated with given rotate values.

Syntax

```
geof:ST_Rotate(geom, angle_rads, axis)
```

Parameter	Type	Description
geom	Object	Geometry shape value in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.
angle_rads	double	The angle (in radians) that represents the rotational angle.
axis	URI	The axis on which the 2d or 3d rotation is performed. This is the X, Y,

Parameter	Type	Description
		or Z axis depending on the direction to rotate. The value can be passed as a string constant like <code>geo:X</code> , <code>geo:Y</code> , or <code>geo:Z</code> .

Returns

Type	Description
Custom object	Point geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Rotate(geof:ST_GeomFromText('POINT Z(1 2
3)'),1.5708,geof:Z)) as ?rotate_point)
```

Aggregator (UDA) Functions

- [ST_Aggr_Union](#): Returns the union of geometries.
- [ST_Aggr_LineString](#): Returns Line geometry constructed from a number of Point geometries.
- [ST_Aggr_MultiLineString](#): Returns MultiLine geometry constructed from a number of Line geometries.
- [ST_Aggr_MultiPoint](#): Returns MultiPoint geometry constructed from a number of Point geometries.
- [ST_Aggr_Polygon](#): Returns Polygon geometry constructed from a number of Point geometries.
- [ST_Aggr_MultiPolygon](#): Returns MultiPolygon geometry constructed from multiple Polygon geometries.
- [ST_Aggr_ConvexHull](#): Returns the convex hull for the specified geometries.
- [ST_Aggr_Intersection](#): Returns the intersection of the specified geometries.

ST_Aggr_Union

This aggregate returns the union of geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_Union(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The union of geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_NumPoints(geof:ST_Aggr_Union(?point_2d)) as ?union)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
}
```

ST_Aggr_LineString

This aggregate returns Line geometry constructed from a number of Point geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_LineString(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	Line geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Aggr_LineString(?point)) as ?ST_Aggr_LineString)
WHERE {
  SELECT (geof:ST_Point(?x, ?y) as ?point)
  WHERE {
    VALUES (?x ?y) {(1 2) (2 3) (-1 2) (-2 -3) (-2 -1)}
  }
}
```

ST_Aggr_MultiLineString

This aggregate returns MultiLine geometry constructed from a number of line geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_MultiLineString(geom1 [, ..., geomN])
```


Parameter	Type	Description
geom1–N	Object	LineString or MultiLineString geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	MultiLine geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Aggr_MultiLineString(?point)) as ?agg_multiline)
WHERE {
  SELECT (geof:ST_GeomFromText(?x) as ?point)
  WHERE {
    VALUES (?x) {('LINESTRING (8 7, 7 8)') ('LINESTRING (1 7, 7 8)')}
  }
}
```

ST_Aggr_MultiPoint

This aggregate returns MultiPoint geometry constructed from a number of Point geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_MultiPoint(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Point geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	MultiPoint geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Aggr_Multipoint(?point)) as ?agg_multipoint)
WHERE {
  SELECT (geof:ST_Point(?x, ?y) as ?point)
  WHERE {
    VALUES (?x ?y) {(1 2)(2 3)}
  }
}
```

ST_Aggr_Polygon

Using aggregate Point geometries, this aggregate returns Polygon geometry constructed from a number of Point geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_Polygon(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Point or Polygon geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	Polygon geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Aggr_Polygon(?point)) as ?agg_polygon)
WHERE {
  SELECT (geof:ST_Point(?x, ?y) as ?point)
  WHERE {
    VALUES (?x ?y) {(1 2) (2 3) (-1 2) (-2 -1) (-2 -3)}
  }
}
```

ST_Aggr_MultiPolygon

This aggregate returns MultiPolygon geometry constructed from multiple polygon geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_MultiPolygon(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Polygon geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type.

Returns

Type	Description
Custom object	MultiPolygon geometry.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_AsText(geof:ST_Aggr_MultiPolygon(?point)) as ?agg_multipolygon)
WHERE {
  SELECT (geof:ST_GeomFromText(?x) as ?point)
  WHERE {
    VALUES (?x) {('POLYGON ((2 0, 2 3, 3 0))')('POLYGON ((11 12,-11 -12,11 -12))')}
  }
}
```

ST_Aggr_ConvexHull

This aggregate returns the convex hull from the specified geometries. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_ConvexHull(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The convex hull from the geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_NumPoints(geof:ST_Aggr_ConvexHull(?point_2d)) as ?convex_hull)
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
}
```

ST_Aggr_Intersection

This aggregate returns the intersection of given geometries that have the same SRID. Any number of geometries can be passed as input.

Syntax

```
geof:ST_Aggr_Intersection(geom1 [, ..., geomN])
```

Parameter	Type	Description
geom1–N	Object	Geometry shape values in WKT string format, WKTLiteral, GMLLiteral, or the Graph Lakehouse Custom OGCGeometry data type. The input geometry shape can be a Point, Multipoint, LineString, MultiLineString, Polygon, Multipolygon, or GeometryCollection.

Returns

Type	Description
Custom object	The intersection between geometries.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ST_NumPoints(geof:ST_Aggr_Intersection(?point_2d)) as ?union
FROM <point>
WHERE {
  ?point a <http://csi.com/geologic_units_24k/point>;
  <http://csi.com/geologic_units_24k/shape> ?shape;
  BIND(geof:ST_GeomFromText(?shape) as ?point_2d)
}
```

Services (UDS) Functions

- **ST_ReadSHP**: Returns geometry objects extracted from a Shapefile (SHP).
- **ST_ReadKML**: Returns geometry objects extracted from a Keyhole Markup Language (KML) file.
- **ST_ReadGeoJSON**: Returns geometry objects extracted from a JSON file.
- **ST_ReadText**: Returns geometry objects extracted from a Well-Known Text (WKT) file.
- **ST_ReadGML**: Returns geometry objects extracted from a Geography Markup Language (GML) file.
- **ST_ReadWKB**: Returns geometry objects extracted from an OpenGIS Well-known Binary (WKB) file.

ST_ReadSHP

This function returns geometry objects extracted from a .shp file.

Syntax

```
geof:ST_ReadSHP("/path/file")
```

Parameter	Type	Description
/path/file	string	The location and file name of the .shp file to read.

Returns

Type	Description
Custom object	Objects extracted from the file.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT *
WHERE {
  SERVICE <http://www.opengis.net/def/function/geosparql/ST_ReadSHP>("{CURRENT_
DIR}/../data/Road_Centerline.shp") {}
}
```

ST_ReadKML

This function returns geometry objects and properties extracted from a .kml file.

Syntax

```
geof:ST_ReadKML("/path/file")
```

Parameter	Type	Description
/path/file	string	The location and file name of the .kml file to read.

Returns

Type	Description
Custom object	Objects extracted from the file.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT * WHERE {
    SERVICE <http://www.opengis.net/def/function/geosparql/ST_ReadKML> (" {CURRENT_
DIR}/../data/Snow_Emergency_Routes.kml") {}
}
```

ST_ReadGeoJSON

This function returns geometry objects and properties extracted from a .json file.

Syntax

```
geof:ST_ReadGeoJSON("/path/file")
```

Parameter	Type	Description
/path/file	string	The location and file name of the .json file to read.

Returns

Type	Description
Custom object	Objects extracted from the file.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT * WHERE {
    SERVICE <http://www.opengis.net/def/function/geosparql/ST_ReadGeoJSON> (" {CURRENT_
```



```
DIR}/../data/geometry_collection.json") {}  
}
```

ST_ReadText

This function returns geometry objects and properties extracted from a .wkt file.

Syntax

```
geof:ST_ReadText("/path/file")
```

Parameter	Type	Description
/path/file	string	The location and file name of the .wkt file to read.

Returns

Type	Description
Custom object	Objects extracted from the file.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT * WHERE {  
  SERVICE <http://www.opengis.net/def/function/geosparql/ST_ReadText> (" {CURRENT_  
DIR}/../data/single_geometryWkt.wkt") {}  
}
```

ST_ReadGML

This function returns geometry objects and properties extracted from a .gml file.

Syntax

```
geof:ST_ReadGML("/path/file")
```

Parameter	Type	Description
/path/file	string	The location and file name of the .gml file to read.

Returns

Type	Description
Custom object	Objects extracted from the file.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT * WHERE {
  SERVICE <http://www.opengis.net/def/function/geosparql/ST_ReadGML> (" {CURRENT_
DIR}/../data/single_geometryGml.gml") {}
}
```

ST_ReadWKB

This function returns geometry objects and properties extracted from .wkb file.

Syntax

```
geof:ST_ReadWKB("/path/file")
```

Parameter	Type	Description
/path/file	string	The location and file name of the .wkb file to read.

Returns

Type	Description
Custom object	Objects extracted from the file.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT * WHERE {
  SERVICE <http://www.opengis.net/def/function/geosparql/ST_ReadWKB>("{CURRENT_
DIR}/../data/single_geometryWkb.wkb") {}
}
```

GeoSPARQL Functions

The geoSPARQL functions follow the world-wide geospatial standard (<https://www.ogc.org/standards/geosparql/>) developed and promoted by the Open Geospatial Consortium (OGC) community to represent geospatial data in RDF format and query that data using the SPARQL query language.

Classes, Data Types, and Properties

This section provides a summary of the geometry shape classes, subclasses, data types, properties, and relationships that geoSPARQL functions operate on.

OWL classes and subclasses

The GeoSPARQL OWL specification describes various classes that are supported (for example, `geo:SpatialObject` and `geo:Feature`), their subclasses (`geo:Geometry`), and associated properties and relationships (`geo:hasGeometry`, `geo:hasDefaultGeometry`, `geo:dimension`, `geo:coordinateDimension`, `geo:spatialDimension`, `geo:isEmpty`, `geo:isSimple`, `geo:hasSerialization`).

GeoSPARQL RDF data must also conform to the OWL representation and implement the features and properties as described in the OGC GeoSPARQL specification.

Data Types

GeoSPARQL uses some custom data types, namely, `geo:wktLiteral` and `geo:gmlLiteral` to represent serialized geometry shapes in text form. Function arguments that specify geometries may pass those objects as `wktLiteral`, `gmlLiteral`, or string literals defined in

the `<http://www.opengis.net/ont/geosparql#>` namespace. Units of measurement (UOM) such as kilometer, meter, mile, degree, and radian are defined in the `<http://www.opengis.net/def/uom/OGC/1.0/>` namespace.

In Graph Lakehouse, these literals are represented as `RDFLiteral` objects, which is a combination of a string and a data type URI. All of the geometries are represented in the graph as either WKT (well-known-text format) or GML (Geometry Markup Language) serialization forms. They may also be represented as objects or string literals.

Relational Properties

GeoSPARQL introduces a set of properties to be used in SPARQL graph patterns. There are three types of relational families: **Simple Features** (sf), **Egenhofer** (eh), and **RCC8** (rcc8).

For additional information on the operation of functions, see the [Geographic Query Language for RDF Data](#) specification.

Functions

The geoSPARQL functions are grouped by the following categories:

- [Non-Topological Functions](#)
- [Simple Feature Family \(Topological\) Functions](#)
- [Egenhofer Family \(Topological\) Functions](#)
- [RCC8 Family \(Topological\) Functions](#)

Note

The URI for the geoSPARQL functions is

`<http://www.opengis.net/def/function/geosparql/>`. For readability, the syntax for each function below includes the prefix `geof:`, defined as `PREFIX geof: <http://www.opengis.net/def/function/geosparql/>`.

Non-Topological Functions

- **distance**: Computes the shortest distance between two geometries.
- **buffer**: Returns a geometric object that represents all points whose distance from a geometry is less than or equal to the specified radius.
- **convexHull**: Returns a geometric object that represents all points in the convex hull of the specified geometry.
- **intersection**: Returns all points that intersect two geometries.
- **union**: Returns all points in the union of two geometries.
- **difference**: Returns all points in the set of difference between two geometries.
- **symDifference**: Returns all points in the set of symmetric difference between two geometries.
- **envelope**: Returns the minimum bounding box of the specified geometry.
- **boundary**: Returns the closure of the boundary of the specified geometry.
- **getSRID**: Returns the spatial reference system URI for the specified geometry.
- **relate**: Evaluates whether the spatial relationship between the specified geometries corresponds to the specified pattern matrix.

distance

This function takes two geometries as input and computes the shortest distance between them based on the SRID of the first geometry. The function converts the distance in meters to the specified unit of measure.

Syntax

```
geof:distance(geom1, geom2, units)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.

Parameter	Type	Description
geom2	geomLiteral	The second geometry.
units	URI	The unit of measure in OGC format.

If no SRID is specified in geom1 or geom2, the function assumes `CRS84` as the default unit of measure and assumes coordinates in Long, Lat format. If geom1 and geom2 are specified as WKT strings and no SRID details are provided, the function computes the Euclidean distance between the two geometries regardless of what unit of measure is provided.

To find the spherical distance between two geometries, you can use `4047` as the SRID. The 4047 value specifies a spherical coordinate system number. The spherical distance gets computed based on Great circle algorithms, and Planer geodesics distances are computed using ellipsoidal formulas.

Returns

Type	Description
double	The shortest distance.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>
SELECT (geof:distance(?x,?y,?z) as ?distance)
WHERE {
  VALUES (?x ?y ?z) {
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>Point (0 0) '
    '<http://www.opengis.net/def/crs/EPSG/0/4326>Point (3 4) ' uom:millimeter)
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>Point (0 0) '
    '<http://www.opengis.net/def/crs/EPSG/0/4326>Point (3 4) ' uom:kilometer)
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>Point (0 0) '
    '<http://www.opengis.net/def/crs/EPSG/0/4326>Point (3 4) ' uom:metre)
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>Point (0 0) '
    '<http://www.opengis.net/def/crs/EPSG/0/4326>Point (3 4) ' uom:foot)
```

```
}  
}
```

buffer

This function takes a `geomLiteral` and `radius` argument in a given unit of measure and produces a polygon of points that has a distance less than or equal to the given radius from a central `geomLiteral` position. The input radius is converted from the source unit of measure based on 1° equal to 111 km and returns the polygon of points less than or equal to the distance from the `geomLiteral` position on an XY plane.

Syntax

```
geof:buffer(geom, radius, units)
```

Parameter	Type	Description
geom	geomLiteral	The geometry.
radius	double	The radius of the geometry.
units	URI	The unit of measure in OGC format.

Returns

Type	Description
geomLiteral	Polygon of points.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
PREFIX uom: <http://www.opengis.net/def/uom/OGC/1.0/>  
SELECT (geof:buffer(?x,?y,?z) as ?buffer)  
WHERE {
```

```
VALUES (?x ?y ?z) {
  ('Point (0 0)' 2 uom:meter)
  ('POLYGON ((1 1 , 1 4, 4 4, 4 1))' ^^geo:wktLiteral 20 uom:millimeter)
}
}
```

convexHull

This function returns a geometric object that represents all points in the convex hull of the geometry. Calculations are in the spatial reference system of the specified geometry.

Syntax

```
geof:convexHull (geom)
```

Parameter	Type	Description
geom	geomLiteral	The geometry.

Returns

Type	Description
geomLiteral	All points in the convex hull.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:convexHull(?x) as ?convexHull)
WHERE {
  VALUES (?x) {
    ('POINT(1 2)')
    ('<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>' ^^geo:gmlLiteral)
    ('POLYGON ((1 1 , 1 4, 4 4, 4 1))' ^^geo:wktLiteral)
  }
}
```


intersection

This function returns a geometric object that represents all points in the intersection of the input geometries. Calculations are in the spatial reference system of the first geometry.

Syntax

```
geof:intersection(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
geomLiteral	All points in the intersection.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:intersection(?x,?y) as ?intersection)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (8 7, 7 8)' 'LINESTRING (2 0, 2 3)')
    ('LINESTRING (0 2, 0 0, 2 0)' 'LINESTRING (0 3, 0 1, 1 0, 3 0)')
  }
}
```

union

This function returns a geometric object that represents all points in the union of two geometries. Calculations are in the spatial reference system of the first geometry.

Syntax

```
geof:union(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
geomLiteral	All points in the union.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:union(?x,?y) as ?union)
WHERE {
  VALUES (?x ?y) {
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>LINESTRING (8 7, 7
8)'^^geo:wktLiteral '<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral)
    ('LINESTRING (0 2, 0 0, 2 0)'^^geo:wktLiteral 'LINESTRING (0 3, 0 1, 1 0, 3
0)'^^geo:wktLiteral)
  }
}
```

difference

This function returns a geometric object that represents all points in the set of difference between two geometries. Calculations are in the spatial reference system of the first geometry.

Syntax

```
geof:difference(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
geomLiteral	All points in the set of difference.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:difference(?x,?y) as ?intersection)
WHERE {
  VALUES (?x ?y) {
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>LINESTRING (8 7, 7
8)'^^geo:wktLiteral '<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral)
    ('LINESTRING (0 2, 0 0, 2 0)'^^geo:wktLiteral 'LINESTRING (0 3, 0 1, 1 0, 3
0)'^^geo:wktLiteral)
  }
}
```

symDifference

This function returns a geometric object that represents all points in the set of symmetric difference between two geometries. Calculations are in the spatial reference system of the first geometry.

Syntax

```
geof:symDifference(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
geomLiteral	All points in the symmetric difference.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:symDifference(?x,?y) as ?symdiff)
WHERE {
  VALUES (?x ?y) {
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>LINESTRING (8 7, 7
8)'^^geo:wktLiteral '<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral)
    ('LINESTRING (0 2, 0 0, 2 0)'^^geo:wktLiteral 'LINESTRING (0 3, 0 1, 1 0, 3
0)'^^geo:wktLiteral)
  }
}
```

envelope

This function returns the minimum bounding box of the specified geometry.

Syntax

```
geof:envelope (geom)
```

Parameter	Type	Description
geom	geomLiteral	The geometry.

Returns

Type	Description
geomLiteral	The minimum bounding box.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:envelope(?x) as ?envelope)
WHERE {
  VALUES (?x) {
    ('POLYGON ((2 0, 2 3, 3 0))'^^geo:wktLiteral) ('POLYGON ((2 0, 2 1, 3 1))')
  }
}
```

boundary

This function returns the closure of the boundary of the specified geometry.

Syntax

```
geof:boundary (geom)
```

Parameter	Type	Description
geom	geomLiteral	The geometry.

Returns

Type	Description
geomLiteral	The closure of the boundary.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:boundary(?x) as ?boundary)
WHERE {
  VALUES (?x) {
    ('POLYGON ((2 0, 2 3, 3 0))'^^geo:wktLiteral) ('<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral)
  }
}
```

getSRID

This function returns the spatial reference system URI for the specified geometry.

Syntax

```
geof:getSRID(geom)
```

Parameter	Type	Description
geom	geomLiteral	The geometry.

Returns

Type	Description
URI	The spatial reference system URI.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:getSRID(?x) as ?srid)
WHERE {
  VALUES (?x) {
    ('POLYGON ((2 0, 2 3, 3 0))'^^geo:wktLiteral) ('<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral)
  }
}
```

relate

This function evaluates whether the spatial relationship between the specified geometries relates to the specified pattern matrix. The spatial reference system for the first geometry is used for spatial calculations.

Syntax

```
geof:relate(geom1, geom2, "pattern_matrix")
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.
pattern_matrix	string	Represents a DE-9IM intersection pattern consisting of T (true) and F (false) values. For example, "T*F***FF*".

Returns

Type	Description
boolean	True if the spatial relationship relates to the specified pattern matrix. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:relate(?x,?y,"T*F**FFF*") as ?relate)
WHERE {
  VALUES (?x ?y) {
    ( '<http://www.opengis.net/def/crs/EPSG/0/4326>LINESTRING (2 0, 2
3)' ^ geo:wktLiteral '<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>' ^ geo:gmlLiteral)
    ('LINESTRING (0 2, 0 0, 2 0)' ^ geo:wktLiteral 'LINESTRING (0 3, 0 1, 1 0, 3
0)' ^ geo:wktLiteral)
  }
}
```

Simple Feature Family (Topological) Functions

The Simple Feature Family relation functions test DE-9IM intersection patterns between two geometries. Each function tests a different pattern matrix and returns true or false depending on whether the specified relation exists or not. Multi-row intersection patterns should be interpreted as a logical OR of each row. Click a function name in the list below to view the syntax and see details about function arguments and return values.

- [sfEquals](#): Tests whether the specified geometries are equal.
- [sfDisjoint](#): Tests whether the specified geometries are disjoint.
- [sfIntersects](#): Tests whether the specified geometries intersect.
- [sfTouches](#): Tests whether the specified geometries touch.
- [sfCrosses](#): Tests whether the first geometry spatially crosses the second geometry.
- [sfWithin](#): Tests whether the first geometry is spatially within the second geometry.
- [sfContains](#): Tests whether the first geometry spatially contains the second geometry.
- [sfOverlaps](#): Tests whether the first geometry spatially overlaps the second geometry.

sfEquals

This function tests whether the specified geometries are equal. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TFFFTFFFT`.

Syntax

```
geof:sfEquals (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries are equal. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfEquals(?x,?y) as ?is_equal)
WHERE {
  VALUES (?x ?y) {
    ('Point (2 3)' 'Point (2 3)')
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4
1))'^^geo:wktLiteral '<gml:Point gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
<gml:coordinates>2,3</gml:coordinates></gml:Point>'^^geo:gmlLiteral)
  }
}
```

sfDisjoint

This function tests whether the specified geometries are disjoint. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `FF*FF****`.

Syntax

```
geof:sfDisjoint (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries are disjoint. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfDisjoint(?x,?y) as ?is_disjoint)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (0 0, 0 1)' 'LINESTRING (1 0, 0 1)')
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>LINESTRING (1 1, 1
0)' '^geo:wktLiteral '<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>0,0
0,1</gml:coordinates></gml:LineString>' '^geo:gmlLiteral)
  }
}
```

sfIntersects

This function tests whether the specified geometries intersect. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is as follows:

```
T*****
*T*****
***T*****
****T*****
```

Syntax

```
geof:sfIntersects (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries intersect. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfIntersects(?x,?y) as ?intersects)
WHERE {
  VALUES (?x ?y) {
    ('LINESTRING (8 7, 7 8)' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 4 1, 4 4, 1 4))' ^^geo:wktLiteral '<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>' ^^geo:gmlLiteral)
```

```
}  
}
```

sfTouches

This function tests whether the specified geometries touch. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is as follows:

```
F*****  
F**T*****  
F***T*****
```

Syntax

```
geof:sfTouches (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries touch. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:sfTouches(?x,?y) as ?touches)  
WHERE {  
  VALUES (?x ?y) {  
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (1 2)')  
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))'^^geo:wktLiteral '<gml:Point gml:id="p21"'
```

```

srsName="http://www.opengis.net/def/crs/EPSG/0/4326"> <gml:coordinates>-
106.4453583,39.11775</gml:coordinates></gml:Point>'^^geo:gmlLiteral)
  }
}

```

sfCrosses

This function tests whether the first geometry spatially crosses the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is as follows:

```

For P/L, P/A, L/A:
T*T**T**
For L/L:
0*T**T**

```

Syntax

```
geof:sfCrosses(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry spatially crosses the second geometry. False if not.

Example

```

PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfCrosses(?x,?y) as ?crosses)
WHERE {

```

```
VALUES (?x ?y) {
  ('LINESTRING (2 0, 2 3)' 'POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')
  ('<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSSG/0/4326"><gml:coordinates>2,0
2,3</gml:coordinates></gml:LineString>'^^geo:gmlLiteral
'<http://www.opengis.net/def/crs/EPSSG/0/4326>POLYGON ((1 1 , 1 4, 4 4, 4
1))'^^geo:wktLiteral)
}
```

sfWithin

This function tests whether the first geometry is spatially within the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `T**F**F***`.

Syntax

```
geof:sfWithin(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry is spatially within the second geometry. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfWithin(?x,?y) as ?is_within)
```

```

WHERE {
  VALUES (?x ?y) {
    ('Point (-106.4453583 39.11775)' 'POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')
    ('<gml:Point gml:id="p21" srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
<gml:coordinates>2,3</gml:coordinates></gml:Point>'^^geo:gmlLiteral
'<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4
1))'^^geo:wktLiteral)
  }
}

```

sfContains

This function tests whether the first geometry spatially contains the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `T*****FF*`.

Syntax

```
geof:sfContains(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry spatially contains the second geometry. False if not.

Example

```

PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfContains(?x,?y) as ?contains)

```

```

WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))'^^geo:wktLiteral 'Point (2 3)'^^geo:wktLiteral)
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4
1))'^^geo:wktLiteral '<gml:Point gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
<gml:coordinates>2,3</gml:coordinates></gml:Point>'^^geo:gmlLiteral)
    ('<gml:LineString gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>45.67,88.56
55.56,89.44</gml:coordinates></gml:LineString>'^^geo:gmlLiteral '<gml:Point
gml:id="p21" srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
<gml:coordinates>45.67,88.56</gml:coordinates></gml:Point>'^^geo:gmlLiteral)
  }
}

```

sfOverlaps

This function tests whether the first geometry spatially overlaps the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is as follows:

```

For A/A, P/P:
T**T***T**
For L/L:
1*T***T**

```

Syntax

```
geof:sfOverlaps(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry spatially overlaps the second geometry. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:sfOverlaps(?x,?y) as ?overlaps)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 0, 2 1, 1 3))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('POLYGON ((2 0, 2 1, 3 1))'^^geo:wktLiteral 'POLYGON ((1 1, 1 4, 4 4, 4
1))'^^geo:wktLiteral)
  }
}
```

Egenhofer Family (Topological) Functions

The Egenhofer Family relation functions test DE-9IM intersection patterns between two geometries. Each function tests a different pattern matrix and returns true or false depending on whether the specified relation exists or not. Multi-row intersection patterns should be interpreted as a logical OR of each row. Click a function name in the list below to view the syntax and see details about function arguments and return values.

- [ehEquals](#): Tests whether the specified objects are equal.
- [ehDisjoint](#): Tests whether the specified objects are disjoint.
- [ehMeet](#): Tests whether the specified geometries meet.
- [ehOverlap](#): Tests whether the specified geometries overlap.
- [ehCovers](#): Tests whether the first geometry spatially covers the second geometry.
- [ehCoveredBy](#): Tests whether the first geometry is covered by the second geometry.

- [ehInside](#): Tests whether the first geometry is inside the second geometry.
- [ehContains](#): Tests whether the first geometry is contained in the second geometry.

ehEquals

This function tests whether the specified objects are equal based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TFFFTFFFT`.

Syntax

```
geof:ehEquals (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the objects are equal. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ehEquals(?x,?y) as ?is_equals)
WHERE {
  VALUES (?x ?y) {
    ('Point (2 3)' 'Point (2 3)')
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4 1))' '^geo:wktLiteral '<gml:Point gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
<gml:coordinates>2,3</gml:coordinates></gml:Point>' '^geo:gmlLiteral)
```

```
}  
}
```

ehDisjoint

This function tests whether the specified objects are disjoint based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `FF*FF****`.

Syntax

```
geof:ehDisjoint (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the objects are disjoint. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ehDisjoint(?x,?y) as ?is_disjoint)  
WHERE {  
  VALUES (?x ?y) {  
    ('LINESTRING (0 0, 0 1)' 'LINESTRING (1 0, 0 1)')  
    ('<http://www.opengis.net/def/crs/EPSG/0/4326> LINESTRING (1 1, 1  
0)' '^geo:wktLiteral '<gml:LineString gml:id="p21"  
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"><gml:coordinates>0,0  
0,1</gml:coordinates></gml:LineString>' '^geo:gmlLiteral)
```

```
}  
}
```

ehMeet

This function tests whether the specified geometries meet based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is as follows:

```
F T*****  
F**T*****  
F***T*****
```

Syntax

```
geof:ehMeet (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries meet. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ehMeet(?x,?y) as ?meets)  
WHERE {  
  VALUES (?x ?y) {  
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (1 2)')  
    ('<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((30 10 , 40 40, 20 40, 10
```

```
20, 30 10))'^^geo:wktLiteral '<gml:Point gml:id="p21"
srsName="http://www.opengis.net/def/crs/EPSG/0/4326"> <gml:coordinates>-
106.4453583,39.11775</gml:coordinates></gml:Point>'^^geo:gmlLiteral)
}
```

ehOverlap

This function tests whether the specified geometries overlap based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `T*T***T**`.

Syntax

```
geof:ehOverlap(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries overlap. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ehOverlap(?x,?y) as ?overlaps)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 0, 2 1, 1 3))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('LINESTRING(0 0, 4 4)'^^geo:wktLiteral 'POLYGON ((1 1, 1 4, 4 4, 4
```

```
1)) '^geo:wktLiteral)
  }
}
```

ehCovers

This function tests whether the first geometry spatially covers the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `T*TFT*FF*`.

Syntax

```
geof:ehCovers (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry spatially covers the second geometry. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ehCovers(?x,?y) as ?covers)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 0, 2 1, 1 3))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))'^^geo:wktLiteral 'LINESTRING(1 1, 4
4)^geo:wktLiteral)
    ('LINESTRING(1 1, 4 4)' 'LINESTRING(2 2, 4 4)')
```

```

('LINESTRING(3 3, 4 4)' 'LINESTRING(2 2, 4 4)')
}
}

```

ehCoveredBy

This function tests whether the first geometry is covered by the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TF*TF**`.

Syntax

```

geof:ehCoveredBy(geom1, geom2)

```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry is covered by the second geometry. False if not.

Example

```

PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:ehCoveredBy(?x,?y) as ?is_covered)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 0, 2 1, 1 3))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('LINESTRING(1 1, 4 4)' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('LINESTRING(1 1, 4 4)' 'LINESTRING(2 2, 4 4)')
    ('LINESTRING(3 3, 4 4)' 'LINESTRING(2 2, 4 4)')
  }
}

```

```
}  
}
```

ehInside

This function tests whether the first geometry is inside the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TFF*FFT**`.

Syntax

```
geof:ehInside(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry is inside the second geometry. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>  
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ehInside(?x,?y) as ?is_inside)  
WHERE {  
  VALUES (?x ?y) {  
    ('Point (-106.4453583 39.11775)' 'POLYGON ((30 10 , 40 40, 20 40, 10 20, 30 10))')  
    ('<gml:Point gml:id="p21" srsName="http://www.opengis.net/def/crs/EPSG/0/4326">  
<gml:coordinates>2,3</gml:coordinates></gml:Point>' '^geo:gmlLiteral  
'<http://www.opengis.net/def/crs/EPSG/0/4326>POLYGON ((1 1, 1 4, 4 4, 4  
1))' '^geo:wktLiteral)
```



```
}  
}
```

ehContains

This function tests whether the first geometry is contained in the second geometry. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `T*TF*FF*`.

Syntax

```
geof:ehContains (geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the first geometry is contained in the second geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>  
SELECT (geof:ehContains (geof:ST_GeomFromText (?x), geof:ST_GeomFromText (?y)) as  
?contains)  
WHERE {  
  VALUES (?x ?y) {  
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (2 3)')  
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'Point (7 8)')  
  }  
}
```

RCC8 Family (Topological) Functions

The RCC8 Family relation functions test DE-9IM intersection patterns between two geometries. Each function tests a different pattern matrix and returns true or false depending on whether the specified relation exists or not. Click a function name in the list below to view the syntax and see details about function arguments and return values.

- [rcc8eq](#): Tests whether the specified geometries are equal.
- [rcc8dc](#): Tests whether the specified geometries are disjoint.
- [rcc8ec](#): Tests whether the specified geometries are externally connected.
- [rcc8po](#): Tests whether the specified geometries overlap.
- [rcc8tpp](#): Tests whether one geometry is a tangential proper part of another geometry.
- [rcc8tppi](#): Tests whether one geometry is a tangential proper part inverse of another geometry.
- [rcc8ntpp](#): Tests whether one geometry is a non-tangential proper part of another geometry.
- [rcc8ntppi](#): Tests whether one geometry is a non-tangential proper part inverse of another geometry.

rcc8eq

This function tests whether the specified objects are equal based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TFFFTFFFT`.

Syntax

```
geof:rcc8eq(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.

Parameter	Type	Description
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the objects are equal. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8eq(?x,?y) as ?is_eq)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))'^^geo:wktLiteral 'POLYGON ((1 1, 1 4, 4 4, 4 1))'^^geo:wktLiteral)
  }
}
```

rcc8dc

This function tests whether the specified objects are disjoint based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `FFTFFTTTT`.

Syntax

```
geof:rcc8dc(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the objects are disjoint. False if not.

Example

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8dc(?x,?y) as ?is_dc)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((11 11, 11 14, 14 14, 14 11))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))'^^geo:wktLiteral 'POLYGON ((1 1, 1 4, 4 4, 4
1))'^^geo:wktLiteral)
  }
}
```

rcc8ec

This function tests whether the specified objects are externally connected based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `FFTFTTTTT`.

Syntax

```
geof:rcc8ec(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the objects are externally connected. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8ec(?x,?y) as ?is_ec)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((4 1, 6 1, 6 4, 4 4))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}
```

rcc8po

This function tests whether the specified geometries overlap based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TTTTTTTTT`.

Syntax

```
geof:rcc8po(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the geometries overlap. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8po(?x,?y) as ?is_po)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 0, 2 1, 1 3,1 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((4 1, 6 1, 6 4, 4 4))')
  }
}
```

rcc8tpp

This function tests whether one geometry is a tangential proper part of another geometry based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is **TFFTTFTTT**.

Syntax

```
geof:rcc8tpp(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the second geometry is a tangential proper part of the first geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8tpp(?x,?y) as ?is_tpp)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 2, 5 2, 5 4, 2 4))' 'POLYGON ((1 1, 1 6, 4 1, 4 6))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}
```

rcc8tppi

This function tests whether one geometry is a tangential proper part inverse of another geometry based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TTTF TTFFFT`.

Syntax

```
geof:rcc8tppi(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the second geometry is a tangential proper part inverse of the first geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8tppi(?x,?y) as ?is_tppi)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 2, 5 2, 5 4, 2 4))' 'POLYGON ((1 1, 1 6, 4 1, 4 6))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}
```

rcc8ntpp

This function tests whether one geometry is a non-tangential proper part of another geometry based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is `TFFTFFTTT`.

Syntax

```
geof:rcc8ntpp(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the second geometry is a non-tangential proper part of the first geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8ntpp(?x,?y) as ?is_ntpp)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 2, 5 2, 5 4, 2 4))' 'POLYGON ((1 1, 1 6, 4 1, 4 6))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}
```

rcc8ntppi

This function tests whether one geometry is a non-tangential proper part inverse of another geometry based on their associated primary geometry objects. The spatial reference system of the first geometry is used for spatial calculations. The defining DE-9IM intersection pattern is TTTFFTTFFT.

Syntax

```
geof:rcc8ntppi(geom1, geom2)
```

Parameter	Type	Description
geom1	geomLiteral	The first geometry.
geom2	geomLiteral	The second geometry.

Returns

Type	Description
boolean	True if the second geometry is a non-tangential proper part inverse of the first geometry. False if not.

Example

```
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>
SELECT (geof:rcc8ntppi(?x,?y) as ?is_ntppi)
WHERE {
  VALUES (?x ?y) {
    ('POLYGON ((2 2, 5 2, 5 4, 2 4))' 'POLYGON ((1 1, 1 6, 4 1, 4 6))')
    ('POLYGON ((1 1, 1 4, 4 4, 4 1))' 'POLYGON ((1 1, 1 4, 4 4, 4 1))')
  }
}
```

Matrix Utilities Library

The matrix utilities return information on various attributes of vector space mapping and related matrix tensors.

- [Matrix Information](#): These utilities are used to retrieve information from a given matrix, vector, or tensor.
- [Matrix Properties](#): These utilities are used to evaluate the characteristics of a matrix or vector.
- [Matrix and Vector Construction](#): These utilities are used to create a matrix or vector.
- [Submatrix and Subvector Extraction](#): These utilities are used to extract elements from a matrix or vector.
- [Correlation and Similarity](#): These utilities are used to calculate correlation and similarity between variables or row vectors.

- **Distance and Vector Flattening:** These utilities are used to calculate distance or flatten vectors.
- **Dimensionality Reduction:** These utilities are used to perform dimensionality reduction using Linear Discriminant Analysis, Principal Component Analysis, or Singular Value Decomposition.
- **Mathematical Operations:** These utilities are used to perform mathematical operations on vectors.
- **Relational Condition Evaluation:** These utilities are used to evaluate conditions on a vector or matrix.

Note

The URI for the matrix utilities is

`<http://cambridgesemantics.com/anzograph/matrices#>`. For readability, the syntax for each function below includes the prefix `matrices:`, defined as `PREFIX matrices: <http://cambridgesemantics.com/anzograph/matrices#>`.

Matrix Information

- **dump_tensor:** Displays the Armadillo header and the first few elements of the matrix or vector as a string.
- **dump_vec:** Returns the string representation of a row or column vector.
- **get_cols:** Returns the number of columns present in a tensor.
- **get_diag:** Extracts a diagonal from a matrix or sparse matrix.
- **get_elem:** Accesses one or more elements that are stored in a tensor.
- **get_max_val:** Retrieves the maximum value from a tensor.
- **get_min_val:** Retrieves the minimum value from a tensor.
- **get_nonzero:** Returns the number of non-zero elements that are present in a sparse matrix.
- **get_order:** Returns the tensor order.

- `get_rows`: Returns the number of rows present in a tensor.
- `get_shape`: Formats the shape of a tensor as a string.
- `get_slices`: Returns the number of slices present in a tensor.
- `get_subvec`: Extracts a range of elements from a row or column vector.
- `get_total_elem`: Returns the total number of elements that are present in a tensor.

dump_tensor

This function displays the Armadillo header and the first few elements of the matrix or vector as a string.

Syntax

```
matrices:dump_tensor(b [, type ] [, isRowWise ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	A tensor of matrix/row vector/column vector.
type	int	Optional argument that specifies the type of tensor: 0=row vector, 1=column vector, 2=matrix. Default is 2.
isRowWise	Boolean	Optional argument that indicates whether the display matrix is column- or row- wise: <code>false</code> =column-wise, <code>true</code> =row-wise. Default is <code>true</code> .

Returns

Type	Description
string	Row- or column- wise string representation of the vector or matrix.

dump_vec

This function returns the string representation of a row or column vector.

Syntax

```
matrices:dump_vec(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The row or column vector to convert to a string.

Returns

Type	Description
string	The string representation of the row or column vector.

get_cols

This function returns the number of columns present in a tensor.

Syntax

```
matrices:get_cols(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor to evaluate.

Returns

Type	Description
long	The number of columns.

get_diag

This function extracts a diagonal from a matrix or sparse matrix.

Syntax

```
matrices:get_diag(b [, k ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix or sparse matrix.
k	long	Optional diagonal number. By default, the main diagonal is accessed ($k=0$). For $k>0$, the k th super-diagonal is accessed (top right corner). For $k<0$, the k th sub-diagonal is accessed (bottom left corner).

Returns

Type	Description
http://anzograph.com/matrices#tensor	The tensor representation of the diagonal as a column vector.

get_elem

This function accesses one or more elements that are stored in a tensor.

Syntax

```
matrices:get_elem(b, i [, j ] [, k ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor.
i	long	The element stored in the <i>i</i> th row.
j	long	Optional argument that lists the element stored in the <i>j</i> th column.
k	long	Optional argument that lists the element stored in the <i>k</i> th slice.

Returns

Type	Description
double	The element value.

get_max_val

This function retrieves the maximum value from a tensor.

Syntax

```
matrices:getmax_val(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor from which to return the maximum value.

Returns

Type	Description
double	The maximum value in the tensor.

get_min_val

This function retrieves the minimum value from a tensor.

Syntax

```
matrices:getmin_val(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor from which to return the minimum value.

Returns

Type	Description
double	The minimum value from the tensor.

get_nonzero

This function gets the number of non-zero elements that are present in a sparse matrix.

Syntax

```
matrices:get_nonzero(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The sparse matrix.

Returns

Type	Description
long	The number of non-zero elements.

get_order

This function returns the tensor order.

Syntax

```
matrices:get_order(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor to evaluate.

Returns

Type	Description
long	The tensor order.

get_rows

This function returns the number of rows present in a tensor.

Syntax

```
matrices:get_rows(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor for which to return the number of rows.

Returns

Type	Description
long	The number of rows.

get_shape

This function formats the shape of a tensor as a string.

Syntax

```
matrices:get_shape(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor to format.

Returns

Type	Description
string	The shape of the tensor.

get_slices

This function gets the number of slices present in a tensor.

Syntax

```
matrices:get_slices(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor for which to return the number of slices.

Returns

Type	Description
long	The number of slices.

get_subvec

This function extracts a range of elements from a row or column vector.

Syntax

```
matrices:get_subvec(b, i, j)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The row or column vector.
i	long	The start index.
j	long	The end index.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a row or column vector.

get_total_elem

This function returns the total number of elements that are present in a tensor.

Syntax

```
matrices:get_total_elem(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The tensor for which to return the total number of elements.

Returns

Type	Description
long	The total number of elements.

Matrix Properties

- [has_nan](#): Evaluates whether a matrix is not a number (NaN).
- [is_colvec](#): Evaluates whether the given matrix is a column vector.
- [is_diag_mat](#): Evaluates whether a matrix is diagonal.
- [is_hermitian](#): Evaluates whether the matrix is hermitian (self-adjoint).
- [is_rowvec](#): Evaluates whether the given matrix is a row vector.
- [is_sorted](#): Evaluates whether a vector or matrix is sorted.
- [is_square](#): Evaluates whether a matrix is square.
- [is_symmetric](#): Evaluates whether a matrix is symmetrical.
- [is_tri_mat_lower](#): Evaluates whether a matrix is lower triangular.
- [is_tri_mat_upper](#): Evaluates whether a matrix is upper triangular.
- [is_vec](#): Evaluates whether the given matrix is a row or column vector.

has_nan

This function evaluates whether a matrix is not a number (NaN).

Syntax

```
matrices:has_nan(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if at least one of the elements is NaN and <code>false</code> if all elements are numbers.

is_colvec

This function evaluates whether the given matrix is a column vector.

Syntax

```
matrices.is_colvec(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	<code>True</code> if the matrix can be interpreted as a column vector. <code>False</code> if the matrix does not have exactly one column.

is_diag_mat

This function evaluates whether a matrix is diagonal, i.e., all elements outside of the main diagonal are zero.

Syntax

```
matrices:is_diag_mat(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if the matrix is diagonal and <code>false</code> if it is not.

is_hermitian

This function evaluates whether a matrix is hermitian (self-adjoint).

Syntax

```
matrices:is_hermitian(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if the matrix is hermitian and <code>false</code> if it is not.

is_rowvec

This function evaluates whether the given matrix is a row vector.

Syntax

```
matrices:is_rowvec(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	<code>True</code> if the matrix can be interpreted as a row vector. <code>False</code> if the matrix does not have exactly one row.

is_sorted

This function evaluates whether a vector or matrix is sorted.

Syntax

```
matrices:is_sorted(b [, t ] [, d ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.
t	boolean	Optional argument that specifies the sort dimension for the matrix. Set to <code>true</code> if elements are sorted row-wise and <code>false</code> if they are sorted column-wise. Default is <code>false</code> .
d	int	Optional argument that specifies the sort direction for the matrix. Allowed

Parameter	Type	Description
		arguments are: • 0: ascend (default). Elements are ascending; consecutive elements can be equal. • 1: descend. Elements are descending; consecutive elements can be equal. • 2: strictascend. Elements are strictly ascending; consecutive elements cannot be equal. • 3: strictdescend. Elements are strictly descending; consecutive elements cannot be equal.

Returns

Type	Description
boolean	True if the elements are sorted. False if they are not.

is_square

This function evaluates whether a matrix is square, i.e., the number of rows is equal to the number of columns.

Syntax

```
matrices:is_square(b)
```


Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if the matrix is square and <code>false</code> if it is not.

is_symmetric

This function evaluates whether a matrix is symmetrical.

Syntax

```
matrices:is_symmetric(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if the matrix is symmetrical and <code>false</code> if it is not.

is_tri_mat_lower

This function evaluates whether a matrix is lower triangular, i.e., the matrix is square sized and all elements above the main diagonal are zero.

Syntax

```
matrices:is_tri_mat_lower(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if the matrix is lower triangular and <code>false</code> if it is not.

is_tri_mat_upper

This function evaluates whether a matrix is upper triangular, i.e., the matrix is square sized and all elements below the main diagonal are zero.

Syntax

```
matrices:is_tri_mat_upper(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	Returns <code>true</code> if the matrix is upper triangular and <code>false</code> if it is not.

is_vec

This function evaluates whether the given matrix is a row or column vector.

Syntax

```
matrices:is_vec(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Returns

Type	Description
boolean	<code>True</code> if the matrix can be interpreted as a column or row vector. <code>False</code> if the matrix does not have exactly one column or one row.

Matrix and Vector Construction

- [gramian](#): Creates a Gramian matrix that is commonly used to compute linear independence.
- [make_matrix](#): Creates a matrix of doubles with the given dimensions and values.
- [make_tensor_from_string](#): Constructs a tensor from the given dimensions in a string.
- [make_vec](#): Constructs a row vector with the given index and value to be stored in the index.
- [string_from_vector](#): Formats a row vector as a plain string.
- [vector_from_string](#): Returns a vector from a string representation of a vector.

gramian

The [Gramian matrix](#) linear algebra aggregate creates a Gramian matrix commonly used to compute linear independence.

Syntax

```
matrices:gramian(x1, x2, ..., xn)
```

Parameter	Type	Description
x1–xn	double	The feature column datasets.

Returns

Type	Description
http://anzograph.com/matrices#tensor	The Gramian matrix.

make_matrix

This function creates a matrix of doubles with the given dimensions and values.

Syntax

```
matrices:make_matrix(m, n [, v1, v2, ..., vn ])
```

Parameter	Type	Description
m	int	The number of rows for the new matrix.
n	int	The number of columns for the new matrix.
v1–vn	double	Optional arguments that specify the row-wise matrix elements to include. Default value is 0 for all elements.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation for $m \times n$ matrix of doubles.

make_tensor_from_string

This function constructs a tensor from the given dimensions in a string.

Syntax

```
matrices:make_tensor_from_string(s [, n ])
```

Parameter	Type	Description
s	string	The string that contains the row-wise elements for constructing the tensor.
n	int	Optional argument that specifies the number of columns to include in the tensor. The default value is 0, which constructs a row vector. A value of 1 constructs a column vector. A value that is greater than 1 constructs a matrix with the specified number of columns.

Returns

Type	Description
http://anzograph.com/matrices#tensor	A tensor of doubles.

make_vec

This aggregate constructs a row vector with the given index and value to be stored in the index.

Syntax

```
matrices:make_vec(n, v)
```

Parameter	Type	Description
n	int	The index into the vector.
v	double	The value to be stored in the vector at the nth index.

Returns

Type	Description
http://anzograph.com/matrices#tensor	A row vector.

string_from_vector

This function formats a row vector as a plain string.

Syntax

```
matrices:string_from_vector(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The row vector to format.

Returns

Type	Description
string	The row vector.

vector_from_string

This function returns a vector from a string representation of a vector.

Syntax

```
matrices:vector_from_string(s)
```

Parameter	Type	Description
s	string	The string representation of a vector.

Returns

Type	Description
http://anzograph.com/matrices#tensor	The vector as a tensor.

Submatrix and Subvector Extraction

- `subvec_head`: Extracts starting elements from a row or column vector.
- `subvec_tail`: Extracts tailing elements from a row or column vector.
- `subview_col`: Extracts a column from a matrix or sparse matrix.
- `subview_cols`: Extracts a range of columns from a matrix or sparse matrix.
- `subview_head_cols`: Extracts starting columns from a matrix or sparse matrix.
- `subview_head_rows`: Extracts starting rows from a matrix or sparse matrix.
- `subview_mat`: Extracts a submatrix from a matrix or sparse matrix.
- `subview_row`: Extracts a row from a matrix or sparse matrix.
- `subview_rows`: Extracts a range of rows from a matrix or sparse matrix.
- `subview_tail_cols`: Extracts tailing columns from a matrix or sparse matrix.
- `subview_tail_rows`: Extracts tailing rows from a matrix or sparse matrix.

`subvec_head`

This function extracts starting elements from a row or column vector.

Syntax

```
matrices:subvec_head(b, n)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	A row or column vector.
n	long	The number of elements to extract from the beginning of the vector.

Returns

Type	Description
<code>http://anzograph.com/matrices#tensor</code>	Tensor representation of a row or column vector with elements from 0 to $n-1$.

subvec_tail

This function extracts tailing elements from a row or column vector.

Syntax

```
matrices:subvec_tail(b, n)
```

Parameter	Type	Description
b	<code>http://anzograph.com/matrices#tensor</code>	A row or column vector.
n	long	The number of elements to extract from the end of the vector.

Returns

Type	Description
<code>http://anzograph.com/matrices#tensor</code>	Tensor representation of a row or column vector with n elements from the tail.

subview_col

This function extracts a column from a matrix or sparse matrix.

Syntax

```
matrices:subview_col(b, n)
```


Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	A matrix or sparse matrix.
n	long	The column index.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a column vector.

subview_cols

This function extracts a range of columns from a matrix or sparse matrix.

Syntax

```
matrices.subview_cols(b, c1, ..., cn)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	A matrix or sparse matrix.
c1–n	long	The start column index to the end column index.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of the matrix with columns from c1 to cn .

subview_head_cols

This function extracts starting columns from a matrix or sparse matrix.

Syntax

```
matrices:subview_head_cols(b, n)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to extract starting columns from.
n	long	The number of columns to extract from the beginning of the matrix.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a matrix with columns from 0 to $n-1$.

subview_head_rows

This function extracts starting rows from a matrix or sparse matrix.

Syntax

```
matrices:subview_head_rows(b, n)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to extract starting rows from.

Parameter	Type	Description
n	long	The number of rows to extract from the beginning of the matrix

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a matrix with rows from 0 to n-1.

subview_mat

This function extracts a submatrix from a matrix or sparse matrix.

Syntax

```
matrices:subview_mat(b, r1, ..., rn, c1, ..., cn)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to extract a submatrix from.
r1–n	long	The start row index to the end row index.
c1–n	long	The start column index to the end column index.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a matrix of size $[1+(rn-r1)] \times [1+(cn-c1)]$.

subview_row

This function extracts a row from a matrix or sparse matrix.

Syntax

```
matrices:subview_row(b, n)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to extract the row from.
n	long	The row index.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a row vector.

subview_rows

This function extracts a range of rows from a matrix or sparse matrix.

Syntax

```
matrices:subview_rows(b, r1, ..., rn)
```

Parameter	Type	Description
b	<code>http://anzograph.com/matrices#tensor</code>	The matrix to extract the rows from.
r1–rn	<code>long</code>	The start row index to the end row index.

Returns

Type	Description
<code>http://anzograph.com/matrices#tensor</code>	Tensor representation of the matrix with rows from <code>r1</code> to <code>rn</code> .

subview_tail_cols

This function extracts tailing columns from a matrix or sparse matrix.

Syntax

```
matrices:subview_tail_cols(b, n)
```

Parameter	Type	Description
b	<code>http://anzograph.com/matrices#tensor</code>	The matrix to extract trailing columns from.
n	<code>long</code>	The number of columns to extract from the end of the matrix.

Returns

Type	Description
<code>http://anzograph.com/matrices#tensor</code>	Tensor representation of a matrix with <code>n</code> columns from

Type	Description
	the tail.

subview_tail_rows

This function extracts tailing rows from a matrix or sparse matrix.

Syntax

```
matrices:subview_tail_rows(b, n)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to extract tailing rows from.
n	long	The number of rows to extract from the end of the matrix.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a matrix with n rows from the tail.

Correlation and Similarity

- [cancor](#): Calculates the overall correlation between two sets of variables.
- [cosine_similarity](#): Calculates the cosine similarity between two row vectors.
- [covariance](#): Provides a measure of the strength of the correlation between two or more sets of random variables.

cancor

The [Canonical correlation](#) aggregate calculates the canonical correlation between two sets of variables.

Syntax

```
matrices:cancor(lc, m, x1, x2, ..., xn, y1, y2, ..., yn)
```

Parameter	Type	Description
lc	int	The number of linear combinations for the first canonical correlation.
m	int	The number of columns in the first set.
x1–xn	double	The feature columns from the first dataset.
y1–yn	double	The feature columns from the second dataset.

Returns

Type	Description
string	Canonical correlation.
string	Square of the canonical correlation.
string	Canonical coefficient.

cosine_similarity

This function calculates the cosine similarity between two row vectors.

Note

The `cosine_similarity` function is not compatible with column or matrix vectors. The input must be row vectors.

Syntax

```
matrices:cosine_similarity(m, n)
```

Parameter	Type	Description
m	http://anzograph.com/matrices#tensor	A row vector.
n	http://anzograph.com/matrices#tensor	The row vector to compare to the vector in argument <code>m</code> .

Returns

Type	Description
double	Results range from -1 to 1 : -1 is perfectly dissimilar and 1 is perfectly similar.

covariance

The [Covariance](#) aggregate provides a measure of the strength of the correlation between two or more sets of random variables (or *variates*).

Syntax

```
matrices:covariance(x1, x2, ..., xn)
```

Parameter	Type	Description
x1–xn	double	Feature columns from the dataset.

Returns

Type	Description
http://anzograph.com/matrices#tensor	The covariance matrix.

Distance and Vector Flattening

- [euclidean_distance](#): Returns the euclidean distance between two vectors.
- [flatten_as_col](#): Returns a flattened version of a matrix as a column vector.
- [flatten_as_row](#): Returns a flattened version of a matrix as a row vector.

euclidean_distance

This function returns the euclidean distance between two vectors.

Syntax

```
matrices.euclidean_distance(b, c)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The first vector in the calculation.
c	http://anzograph.com/matrices#tensor	The vector to calculate the distance from vector b .

Returns

Type	Description
double	The euclidean distance between the input vectors.

flatten_as_col

This function returns a flattened version of a matrix as a column vector.

Syntax

```
matrices:flatten_as_col(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to flatten.

Returns

Type	Description
http://anzograph.com/matrices#tensor	The tensor representation of the matrix as a column vector.

flatten_as_row

This function returns a flattened version of a matrix as a row vector.

Syntax

```
matrices:flatten_as_row(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to flatten.

Returns

Type	Description
http://anzograph.com/matrices#tensor	The tensor representation of the matrix as a row vector.

Dimensionality Reduction

- [Linear Discriminant Analysis \(LDA\)](#): These functions use dimensionality reduction to make predictions.
- [Principal Component Analysis \(PCA\)](#): These functions reduce a high-dimensional dataset into fewer dimensions while retaining important information.
- [Singular Value Decomposition \(SVD\)](#): These functions are similar to PCA except that the factorization is done on the data matrix.
- [transform](#): Applies PCA or SVD to transform the samples onto the new subspace.

Linear Discriminant Analysis (LDA)

[Linear discriminant analysis](#) functions apply linear discriminant analysis (LDA) to create combined eigenvalues and vectors that characterize or separate two or more classes of objects or events. The following functions are available for LDA operations:

- [lda::create](#)
- [lda::get_eigvec](#)
- [lda::get_eigval](#)
- [lda::get_raw_eigval](#)
- [lda::predict](#)
- [lda::transform](#)

lda::create

This aggregate applies LDA to create combined eigenvalues and eigenvectors.

Syntax

```
matrices::lda::create(y, x1, x2, ..., xn)
```

Parameter	Type	Description
y	double	The class of the feature tuple.

Parameter	Type	Description
x1–xn	double	The feature column datasets.

Returns

Type	Description
http://anzograph.com/matrices#lda_result	The combined eigenvalues, eigenvectors, class mean, count, and class map.

lda::get_eigvec

Given LDA data, this function returns LDA's eigenvectors as a matrix.

Syntax

```
matrices:lda::get_eigvec(lda_data)
```

Parameter	Type	Description
lda_data	http://anzograph.com/matrices#lda_result	Linear discriminant analysis data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Eigenvectors as a matrix.

lda::get_eigval

Given LDA data, this function gets LDA's eigenvalues as a column vector.

Syntax

```
matrices:lda::get_eigval(lda_data)
```

Parameter	Type	Description
lda_data	http://anzograph.com/matrices#lda_result	Linear discriminant analysis data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Eigenvalues in descending order as a column vector.

lda::get_raw_eigval

Given LDA data, this function gets LDA's unsorted eigenvalues.

Syntax

```
matrices:lda::get_raw_eigval(lda_data)
```

Parameter	Type	Description
lda_data	http://anzograph.com/matrices#lda_result	LDA data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Eigenvalues in unsorted order as a column vector.

lda::predict

This function predicts the class for the samples using LDA as the classifier.

Syntax

```
matrices:lda::predict(lda_data, p1, p2, ..., pn)
```

Parameter	Type	Description
lda_data	http://anzograph.com/matrices#lda_result	LDA data.
p1–pn	double	The data sample that contains the class to predict.

Returns

Type	Description
string	The class name to which data tuple belongs.

lda::transform

This function applies LDA to transform samples onto the new subspace.

Syntax

```
matrices::lda::transform(lda_data, d, x1, x2, ..., xn)
```

Parameter	Type	Description
lda_data	http://anzograph.com/matrices#lda_result	Linear discriminant analysis data.
d	int	The number of eigenvectors to consider from the start.
x1–xn	double	The feature column datasets.

Returns

Type	Description
double	The original data transformed into the tuple of lower dimensional space.

Principal Component Analysis (PCA)

Applies [Principal component analysis](#) (PCA) to create combined eigenvalues and vectors that highlight patterns in a dataset, making it easier to explore and visualize data. The following functions are available for PCA operations:

- [pca::create](#)
- [pca::get_eigval](#)
- [pca::get_eigvec](#)
- [pca::get_raw_eigval](#)
- [transform](#)

pca::create

This aggregate applies PCA to create combined eigenvalues and eigenvectors.

Syntax

```
matrices:pca::create(x1, x2, ..., xn)
```

Parameter	Type	Description
x1–xn	double	The feature column datasets.

Returns

Type	Description
http://anzograph.com/matrices#feature_	PCA data containing eigenvalues and

Type	Description
result	eigenvectors.

pca::get_eigval

This function retrieves PCA's eigenvalues as a column vector from PCA data.

Syntax

```
matrices:pca::get_eigval(pca_data)
```

Parameter	Type	Description
pca_data	http://anzograph.com/matrices#feature_result	Principal Component Analysis data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Eigenvalues in descending order as column vectors.

pca::get_eigvec

This function retrieves PCA's eigenvectors as a matrix from the PCA data.

Syntax

```
matrices:pca::get_eigvec(pca_data)
```

Parameter	Type	Description
pca_data	http://anzograph.com/matrices#feature_result	Principal Component Analysis data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Eigenvectors as a matrix.

`pca::get_raw_eigval`

This function gets the PCA's unsorted eigenvalues from the PCA data.

Syntax

```
matrices:pca::get_raw_eigval(pca_data)
```

Parameter	Type	Description
<code>pca_data</code>	http://anzograph.com/matrices#feature_result	Principal Component Analysis data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Eigenvalues in unsorted order as column vectors.

Singular Value Decomposition (SVD)

The [Singular value decomposition](#) (SVD) matrix factorization method creates combined singular values and right singular vectors.

The following functions are available for SVD operations:

- [svd::create](#)
- [svd::get_sigval](#)

- [svd::get_sigvec](#)
- [transform](#)

svd::create

This aggregate applies SVD to create combined singular values and right singular vectors.

Syntax

```
matrices::svd::create(x1, x2, ..., xn)
```

Parameter	Type	Description
x1–xn	double	The feature column datasets.

Returns

Type	Description
http://anzograph.com/matrices#feature_result	SVD data containing singular values and right singular vectors.

svd::get_sigval

This function gets SVD's singular values as a column vector from the SVD data.

Syntax

```
matrices::svd::get_sigval(svd_data)
```

Parameter	Type	Description
svd_data	http://anzograph.com/matrices#feature_result	SVD data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Singular values in descending order as a column vector.

svd::get_sigvec

This function gets SVD's singular vector as a matrix from the SVD data.

Syntax

```
matrices::svd::get_sigvec(svd_data)
```

Parameter	Type	Description
svd_data	http://anzograph.com/matrices#feature_result	SVD data.

Returns

Type	Description
http://anzograph.com/matrices#tensor	Right singular vectors as a matrix.

transform

This function applies PCA or SVD (depending on the input) to transform the samples onto the new subspace.

Syntax

```
matrices::transform(data, d, x1, x2, ..., xn)
```

Parameter	Type	Description
data	http://anzograph.com/matrices#feature_result	PCA or SVD data.
d	int	The number of eigenvectors to consider from the end.
x1–xn	double	Feature column datasets.

Returns

Type	Description
double	Sample data transformed into the tuple of lower dimensional space.

Mathematical Operations

- [sigmoid](#): Returns the logistic sigmoid calculation of a vector.
- [vdiff](#): Returns the difference between two vectors.
- [vsum](#): Returns the sum of two vectors.

sigmoid

This function returns the logistic sigmoid calculation of a vector.

Syntax

```
matrices:sigmoid(b)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The vector to evaluate.

Returns

Type	Description
<code>http://anzograph.com/matrices#tensor</code>	The logistic sigmoid of the vector.

vdiff

This function returns the difference between two vectors.

Syntax

```
matrices:vdiff(b, c)
```

Parameter	Type	Description
b	<code>http://anzograph.com/matrices#tensor</code>	The first vector in the calculation.
c	<code>http://anzograph.com/matrices#tensor</code>	The vector to subtract from vector b .

Returns

Type	Description
<code>http://anzograph.com/matrices#tensor</code>	Tensor representation of the difference between the input vectors.

vsum

This function returns the sum of two vectors.

Syntax

```
matrices:vsum(b, c)
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The first vector in the calculation.
c	http://anzograph.com/matrices#tensor	The vector to add to vector b .

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of the sum of the input vectors.

Relational Condition Evaluation

- **mat_all**: Evaluates whether all elements in a matrix are non-zero or satisfy the specified relational condition.
- **mat_any**: Evaluates whether any elements in a matrix are non-zero or satisfy the specified relational condition.
- **vec_all**: Evaluates whether all elements in a row or column vector are non-zero or satisfy the specified relational condition.
- **vec_any**: Evaluates whether any elements in a row or column vector are non-zero or satisfy the specified relational condition.

mat_all

This function evaluates whether all elements in a matrix are non-zero or satisfy the specified relational condition.

Syntax

```
matrices:mat_all(b [, d ] [, c ] [, val ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.

Parameter	Type	Description
d	boolean	Optional argument that indicates whether to check rows or columns. Set to <code>true</code> for rows or <code>false</code> for columns. Default is <code>false</code> .
c	int	Optional argument that specifies the relational condition to test: • 0 (default): not equal • 1: greater than • 2: less than • 3: equal • 4: greater than or equal to • 5: less than or equal to
val	double	Optional argument that specifies the value to apply the condition (<code>c</code>) to. Default is <code>0</code> .

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a row vector with each element as 0 or 1, indicating whether the corresponding row or column has all non-zero elements.

mat_any

This function evaluates whether any elements in a matrix are non-zero or satisfy the specified relational condition.

Syntax

```
matrices:mat_any(b [, d ] [, c ] [, val ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The matrix to evaluate.
d	boolean	Optional argument that indicates whether to check rows or columns. Set to <code>true</code> for rows or <code>false</code> for columns. Default is <code>false</code> .
c	int	Optional argument that specifies the relational condition to test: • 0 (default): not equal • 1: greater than • 2: less than • 3: equal • 4: greater than or equal to • 5: less than or equal to
val	double	Optional argument that specifies the value to apply the condition (<code>c</code>) to. Default is <code>0</code> .

Returns

Type	Description
http://anzograph.com/matrices#tensor	Tensor representation of a row vector with each element as 0 or 1, indicating whether the corresponding row or column has any non-zero elements.

vec_all

This function evaluates whether all elements in a row or column vector are non-zero or satisfy the specified relational condition.

Syntax

```
matrices:vec_all(b [, c ] [, val ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The vector to evaluate.
c	int	Optional argument that specifies the relational condition to test: • 0 (default): not equal • 1: greater than • 2: less than • 3: equal • 4: greater than or equal to • 5: less than or equal to
val	double	Optional argument that specifies the

Parameter	Type	Description
		value to apply the condition (<code>c</code>) to. Default is <code>0</code> .

Returns

Type	Description
boolean	Returns <code>true</code> if all elements are non-zero or satisfy the condition and <code>false</code> if not.

vec_any

This function evaluates whether any elements in a row or column vector are non-zero or satisfy the specified relational condition.

Syntax

```
matrices:vec_any(b [, c ] [, val ])
```

Parameter	Type	Description
b	http://anzograph.com/matrices#tensor	The vector to evaluate.
c	int	Optional argument that specifies the relational condition to test: 0 (default): not equal 1: greater than 2: less than 3: equal 4: greater than or equal to 5: less than or equal to

Parameter	Type	Description
val	double	Optional argument that specifies the value to apply the condition (<i>c</i>) to. Default is 0.

Returns

Type	Description
boolean	Returns <code>true</code> if any elements are non-zero or satisfy the condition and <code>false</code> if not.

Sketch Library

The sketch library provides extremely efficient streaming algorithms that approximate calculations, such as count distinct, quantiles, most frequent items, joins, and matrix computations, and return data sketches. This topic describes each of the sketch functions.

Note

The URI for the sketch functions is

`<http://cambridgesemantics.com/anzograph/statistics/sketch#>`. For readability, the syntax for each function below includes the prefix `sketch:`, defined as

`PREFIX sketch:`

`<http://cambridgesemantics.com/anzograph/statistics/sketch#>`.

- **Cardinality Metric (HLL)**: Uses Apache DataSketches HyperLogLog (HLL) to calculate cardinality estimates for a dataset.
- **Frequent Items (FI)**: Collection of functions used to create frequency sketches and obtain information about frequent items.
- **Quantile/Rank Sketch (KLL)**: Collection of functions that use the KLL sketch computation model to approximate minimum and maximum items in a dataset, the quantile and rank of items, the Probability Mass Function (PMF), and the Cumulative Distribution Function (CDF).

- **Theta Sketch (THETA)**: Collection of functions that use the Theta Sketch framework to compute estimates of the cardinality, union, intersection, and difference set operations and return a Theta Sketch.

Cardinality Metric (HLL)

This aggregate calculates cardinality estimates for a dataset using Apache DataSketches HyperLogLog (HLL).

Reference: [Cardinality Prominence Metric](#)

Syntax

```
sketch:hll(data [, log_base_2_K ] [, hll_target_type ])
```

Parameter	Type	Description
data	byte, short, int, long, float, double, string, URI	The dataset.
log_base_2_K	int	Optional argument that specifies the log base 2 of K, where K is the number of buckets or slots for the sketch. Must be between 4 and 21 (inclusive). Default value is 12.
hll_target_type	int	Optional argument that specifies the target type for the HLL sketch. Supported values are 4 (HLL_4), 6 (HLL_6), or 8 (HLL_8). Default value is 4.

Returns

Type	Description
double	The cardinality metric value.

Frequent Items (FI)

The FI aggregate is used to estimate the frequency of items in a dataset, the upper and lower bounds of the items, the number of active items, and the total stream weight. FI returns a binary stream (Frequent Items Sketch) containing all of the computed values. Values can be retrieved from the sketch using the [Frequent Items Sketch Retrieval Functions](#): `get_estimates`, `get_active_items_total_weights`, `get_top_items`, and `get_top_strings`.

Tip

For more information about frequency sketches, see [Frequency Sketches Overview](#).

FI Syntax

```
sketch:fi(values [, weight ])
```

Parameter	Type	Description
values	short, int, long, float, double, string	The dataset.
weight	long	Optional argument that specifies the weight of <code>val</code> . The default value is 1.

Returns

Type	Description
http://anzograph.com/statistics#fi_sketch	Binary Frequent Items Sketch.

Frequent Items Sketch Retrieval Functions

The following functions are available for retrieving values from a Frequent Items Sketch:

- [fi::get_estimates](#)
- [fi::get_active_items_total_weights](#)

- [fi::get_top_items](#)
- [fi::get_top_strings](#)

fi::get_estimates

Returns the estimates for the frequency and lower and upper bound of the given item in a sketch.

Syntax

```
sketch:fi::get_estimates(fi_sketch, item)
```

Parameter	Type	Description
fi_sketch	http://anzograph.com/statistics#fi_sketch	Frequent Items Sketch.
item	Object	Item for which to get estimates.

Returns

Type	Description
long	Frequency estimate for the item.
long	Lower bound estimate for the item.
long	Upper bound estimate for the item.

fi::get_active_items_total_weights

Returns the number of active items and the estimated total stream weight from a sketch.

Syntax

```
sketch:fi::get_active_items_total_weights(fi_sketch)
```

Parameter	Type	Description
fi_sketch	http://anzograph.com/statistics#fi_sketch	Frequent Items Sketch.

Returns

Type	Description
long	The estimated number of active items.
long	The estimated total stream weight.

fi::get_top_items

Returns the most frequent items and their corresponding frequency.

Syntax

```
sketch:fi::get_top_items(fi_sketch)
```

Parameter	Type	Description
fi_sketch	http://anzograph.com/statistics#fi_sketch	Frequent Items Sketch.

Returns

Type	Description
double	The item with the highest frequency.
long	Frequency estimate of the first item.
double	The item with the second highest frequency.
long	Frequency estimate of the second item.

Type	Description
double	The item with the n th highest frequency.
long	Frequency estimate of the n th item.

fi::get_top_strings

Gets top frequent strings and their corresponding frequency.

Syntax

```
sketch:fi::get_top_strings(fi_sketch)
```

Parameter	Type	Description
fi_sketch	http://anzograph.com/statistics#fi_sketch	Frequent Items Sketch.

Returns

Type	Description
string	The string with the highest frequency.
long	Frequency estimate of the first string.
string	The string with the second highest frequency.
long	Frequency estimate of the second string.
string	The string with the n th highest frequency.
long	Frequency estimate of the n th string.

Quantile/Rank Sketch (KLL)

The KLL aggregate uses the KLL Sketch computation model to calculate the approximate minimum and maximum items in a dataset, the quantile and rank of items, the Probability Mass Function (PMF), and the Cumulative Distribution Function (CDF). KLL returns a binary stream (KLL Sketch) containing all of the computed values. Values can be retrieved from the sketch using various [KLL Sketch Retrieval Functions](#).

Tip

For more information about KLL sketches, see [KLL Sketch](#).

KLL Syntax

```
sketch:kll(values [, k ])
```

Parameter	Type	Description
values	short, int, long, float, double, string	The dataset.
k	int	Optional argument that configures the size of the sketch and its estimation error. Can be any value between 8 and 65535 (inclusive). The default value is 200, which results in a normalized rank error of about 1.65%. Higher values will have a smaller error but the sketch will be larger (and slower).

Returns

Type	Description
http://anzograph.com/statistics#kll_sketch	Binary KLL sketch.

KLL Sketch Retrieval Functions

The following functions are available for retrieving values from a KLL sketch:

- `kll::get_min_value`
- `kll::get_max_value`
- `kll::get_n`
- `kll::get_num_retained`
- `kll::get_rank`
- `kll::get_quantile`
- `kll::get_quantiles`
- `kll::get_quantiles_str`
- `kll::get_pmf`
- `kll::get_cdf`

`kll::get_min_value`

Returns the minimum value in a KLL sketch.

Syntax

```
sketch:kll::get_min_value(kll_sketch)
```

Parameter	Type	Description
<code>kll_sketch</code>	http://anzograph.com/statistics#kll_sketch	KLL sketch.

Returns

Type	Description
double	The minimum value in the sketch.

Type	Description
string	If the input is a string, the minimum string is returned.

kl1::get_max_value

Returns the maximum value in a KLL sketch.

Syntax

```
sketch:kl1::get_max_value(kl1_sketch)
```

Parameter	Type	Description
kl1_sketch	http://anzograph.com/statistics#kl1_sketch	KLL sketch.

Returns

Type	Description
double	The maximum value in the sketch.
string	If the input is a string, the minimum string is returned.

kl1::get_n

Returns the length of a KLL sketch.

Syntax

```
sketch:kl1::get_n(kl1_sketch)
```

Parameter	Type	Description
kl1_sketch	http://anzograph.com/statistics#kl1_sketch	KLL sketch.

Returns

Type	Description
long	The length of the sketch.

kl1::get_num_retained

Returns the number of retained items (samples) in a sketch.

Syntax

```
sketch:get_num_retained(kl1_sketch)
```

Parameter	Type	Description
kl1_sketch	http://anzograph.com/statistics#kl1_sketch	KLL sketch.

Returns

Type	Description
long	The number of retained items (samples) in the sketch.

kl1::get_rank

Returns an approximation of the normalized (fractional) rank of the given item.

Syntax

```
sketch:kl1::get_rank(kl1_sketch, v)
```

Parameter	Type	Description
kl1_sketch	http://anzograph.com/statistics#kl1_sketch	KLL sketch.
v	double	The item to retrieve the rank for.

Returns

Type	Description
double	The approximate rank of the item from 0 - 1 (inclusive).

kll::get_quantile

Returns an approximation of the value for an item from the rank.

Syntax

```
sketch:kll::get_quantile(kll_sketch, fraction)
```

Parameter	Type	Description
kll_sketch	http://anzograph.com/statistics#kll_sketch	KLL sketch.
fraction	double	The fractional position in the hypothetical sorted stream.

Returns

Type	Description
double	An approximation of the value of the item that would be preceded by the given fraction of a hypothetical sorted version of the sketch.
string	An approximation of the string when the input is a string.

kll::get_quantiles

Provides a more efficient, multiple-query version of [kll::get_quantile](#) that enables you to specify a number of evenly spaced fractional ranks.

Syntax

```
sketch:kll::get_quantiles(kll_sketch, f1, f2, ..., f10)
```

Parameter	Type	Description
kll_sketch	http://anzograph.com/statistics#kll_sketch	KLL sketch.
f1–f10	double	Normalized or fractional ranks in the hypothetical sorted stream. The ranks must be in the interval 0.0 - 1.0 (inclusive).

Returns

Type	Description
double	An approximation of the values in the same order as the given fractional positions.

kll::get_quantiles_str

Provides an approximation to the strings when the input is a string type.

Syntax

```
sketch:kll::get_quantiles_str(kll_sketch, f1, f2, ..., f10)
```

Parameter	Type	Description
kll_sketch	http://anzograph.com/statistics#kll_sketch	KLL sketch.
f1–f10	double	Normalized or fractional ranks in the hypothetical sorted stream. The ranks

Parameter	Type	Description
		must be in the interval 0.0 - 1.0 (inclusive).

Returns

Type	Description
string	An approximation of the strings.

kl1::get_pmf

Provides an approximation to the Probability Mass Function (PMF) of the input stream.

Syntax

```
sketch:kl1::get_pmf(kl1_sketch, v1, v2, ..., v10)
```

Parameter	Type	Description
kl1_sketch	http://anzograph.com/statistics#kl1_sketch	KLL sketch.
v1–v10	Object	Input values between the minimum and maximum values of the input stream. Values must be unique and monotonically increasing.

Returns

Type	Description
double	PMF values corresponding to the input.

kl::get_cdf

Provides an approximation to the Cumulative Distribution Function (CDF), which is the cumulative analog of the PMF of the input stream.

Syntax

```
sketch:kl::get_cdf(kll_sketch, v1, v2, ..., v10)
```

Parameter	Type	Description
kll_sketch	http://anzograph.com/statistics#kll_sketch	KLL sketch.
v1–v10	Object	Input values between the minimum and maximum values of the input stream. Values must be unique and monotonically increasing.

Returns

Type	Description
double	CDF values corresponding to the input.

Theta Sketch (THETA)

The THETA aggregate uses the Theta Sketch framework to compute estimates of the cardinality, union, intersection, and difference set operations and return a binary stream (Theta Sketch) containing the computed values. Values can be retrieved from the sketch using the : cardinality, union, intersection, and difference.

Tip

Theta Sketches are a generalization of the well-known K^{th} Minimum Value (KMV) sketches. For more information about the framework, you may find the following references helpful:

- [The Theta Sketch Framework](#)
- [Estimating Counts of Distinct Values with KMV](#)

THETA Syntax

```
sketch:theta (values)
```

Parameter	Type	Description
values	short, int, long, float, double, string	The dataset to operate on.

Returns

Type	Description
http://anzograph.com/statistics#theta_sketch	Binary Theta Sketch

Theta Sketch Retrieval Functions

The following functions are available for retrieving values from a Theta Sketch:

- [theta::cardinality](#)
- [theta::union](#)
- [theta::intersection](#)
- [theta::difference](#)

theta::cardinality

Retrieves the estimated count of values in a Theta Sketch.

Syntax

```
sketch:theta::cardinality(theta_sketch)
```

Parameter	Type	Description
theta_sketch	http://anzograph.com/statistics#theta_sketch	Binary Theta Sketch

Returns

Type	Description
double	The count of items in the sketch.

theta::union

Retrieves the estimate of the number of items that are in the union of two or more Theta Sketches.

Syntax

```
sketch:theta::union(theta_sketch1, theta_sketch2 [, theta_sketchN ])
```

Parameter	Type	Description
theta_sketch1– N	http://anzograph.com/statistics#theta_sketch	Any number of Theta Sketches.

Returns

Type	Description
double	The estimated number of items in the union.

theta::intersection

Retrieves the estimate of the number of items that are in the intersection between two or more Theta Sketches.

Syntax

```
sketch:theta::intersection(theta_sketch1, theta_sketch2 [, theta_sketchN ])
```

Parameter	Type	Description
theta_sketch1–N	http://anzograph.com/statistics#theta_sketch	Any number of Theta Sketches.

Returns

Type	Description
double	The estimated number of items that intersect in the sketches.

theta::difference

Retrieves the estimate of the number of items that are in the difference between two Theta Sketches, i.e., the number of items that are in the first sketch but not in the second sketch.

Syntax

```
sketch:theta::difference(a, b)
```

Parameter	Type	Description
a	http://anzograph.com/statistics#theta_sketch	The first Theta Sketch.
b	http://anzograph.com/statistics#theta_sketch	The Theta Sketch to compare to sketch a.

Returns

Type	Description
double	The estimated number of items in the difference between the sketches.

Utilities Library

The utilities library contains several miscellaneous functions. This topic describes each of the functions.

- [LCASE](#): Converts the letters in a string literal to lower case based on the given locale.
- [UCASE](#): Converts the letters in a string literal to upper case based on the given locale.
- [bitap_fuzzy](#): Performs fuzzy string matching using the Bitap algorithm.
- [cpp::fuzzy_match](#): Compares the given string to the specified pattern and returns a score.
- [cpp::levenshtein_dist](#): Calculates the Levenshtein distance between two strings.
- [damerauLevenshteinDistance](#): Calculates the Damerau-Levenshtein distance between two strings.
- [maskFirstNChars](#): Masks the beginning N characters with asterisks (*).
- [maskLastNChars](#): Masks the last N characters with asterisks (*).
- [regex](#): Creates a JSON string with all of the matches for the specified regular expression.

Note

The URI for the utilities is

`<http://cambridgesemantics.com/anzograph/utilities#>`. For readability, the syntax for each function below includes the prefix `util:`, defined as `PREFIX util:`
`<http://cambridgesemantics.com/anzograph/utilities#>`.

LCASE

This function converts the letters in a string literal to lower case according to the rules of the specified locale.

Syntax

```
util:LCASE(text, locale)
```

Argument	Type	Description
text	string	The string literal to convert to lower case.
locale	string	The locale to use for the conversion.

Returns

Type	Description
string	The string with lower case characters.

UCASE

This function converts all letters in a string to upper case according to the rules of the specified locale.

Syntax

```
util:UPPER(text, locale)
```

Argument	Type	Description
text	string	The string value to convert to upper case.
locale	string	The locale to use for the conversion.

Returns

Type	Description
string	The string with upper case characters.

bitap_fuzzy

This function performs fuzzy string matching using the [Bitap](#) algorithm. The function evaluates whether the specified text contains a string that is approximately equal to the given pattern, where approximate equality is determined in terms of Hamming distance.

Syntax

```
util:bitap_fuzzy(pattern, text, k)
```

Argument	Type	Description
pattern	string	The pattern to match the <code>text</code> against.
text	string	The string to match the <code>pattern</code> against.
k	int	The number of errors that are allowed (the Hamming distance of k).

Returns

Type	Description
int	The first match's starting index in the text. 0 means starting position, and -1 means no match.

cpp::fuzzy_match

This function is modeled after [Sublime Text's](#) fuzzy matching and compares the given string to the specified pattern and returns a score.

Syntax

```
util:cpp::fuzzy_match(pattern, string)
```

Argument	Type	Description
pattern	string	The pattern to match the <code>string</code> against.
string	string	The string to match the <code>pattern</code> against.

Returns

Type	Description
int	The matched score. The score is returned only for matching strings. If there is no match, the score is <code>-9999</code> .

Example

The following example queries the Tickit data set to find the number of city names that are a fuzzy match to the specified VALUES.

```
PREFIX util: <http://cambridgesemantics.com/anzograph/utilities#>
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT (count(*) as ?totalMatches)
FROM <http://anzograph.com/tickit>
WHERE {
    ?venueid tickit:venuecity ?city .
    VALUES (?to_match) {
        ("Denver") ("Seattle") ("East") ("Toronto")
    }
    BIND(util:cpp::fuzzy_match(?city, ?to_match) as ?matched)
    FILTER(?matched > -9999)
}
```

```
totalMatches
-----
10
1 rows
```

cpp::levenshtein_dist

This function calculates the Levenshtein distance or measure of similarity between two strings. The distance is the smallest number of insertions, deletions, and/or substitutions required to transform the first string into the second string.

Syntax

```
util:cpp::levenshtein_dist(string1, string2)
```

Argument	Type	Description
string1	string	The string that would be transformed into <code>string2</code> .
string2	string	The string to measure <code>string1</code> against.

Returns

Type	Description
int	The Levenshtein distance between the strings.

Example

The following example queries the Tickit data set to find cities whose names have a levenshtein distance that is not equal to 0 and is less than or equal to 5 when compared with the values "Denver," "Seattle," or "East."

```
PREFIX util: <http://cambridgesemantics.com/anzograph/utilities#>
PREFIX tickit: <http://anzograph.com/tickit/>
SELECT DISTINCT ?city ?dist
FROM <http://anzograph.com/tickit>
WHERE {
    ?venueid tickit:venuecity ?city .
    VALUES (?to_match) {
        ("Denver") ("Seattle") ("East")
    }
    BIND(util:cpp::levenshtein_dist(?city, ?to_match) as ?dist)
    FILTER(?dist != 0 && ?dist <= 5)
```



```
}
ORDER BY ?city
```

```
city      | dist
-----+-----
Atlanta   |    5
Boston     |    4
Carson     |    4
Dallas     |    5
Dayton     |    4
Dayton     |    5
Detroit    |    5
Frisco     |    5
Glendale   |    5
Hershey    |    5
Houston    |    5
Landrover  |    4
Miami      |    4
Newark     |    5
Ottawa     |    5
Saratoga   |    5
Seattle    |    5
Sunrise    |    5
Tampa      |    4
Vancouver  |    5
20 rows
```

damerauLevenshteinDistance

This function calculates the Damerau-Levenshtein distance or measure of similarity between two strings. The distance is the smallest number of insertions, deletions, character transpositions, and/or substitutions required to transform the first string into the second string.

Syntax

```
util:damerauLevenshteinDistance(string1, string2)
```

Argument	Type	Description
string1	string	The string that would be transformed into <code>string2</code> .

Argument	Type	Description
string2	string	The string to measure <code>string1</code> against.

Returns

Type	Description
int	The Damerau-Levenshtein distance between the strings.

maskFirstNChars

This function masks the beginning N characters with an asterisk (*).

Syntax

```
util:maskFirstNChars(string, number_of_chars)
```

Argument	Type	Description
string	string	The string to mask.
number_of_chars	int	The number of characters to mask from the beginning of the string.

Returns

Type	Description
string	The string with the masked characters.

maskLastNChars

This function masks the last N characters with an asterisk (*).

Syntax

```
util:maskLastNChars(string, number_of_chars)
```

Argument	Type	Description
string	string	The string to mask.
number_of_chars	int	The number of characters to mask from the end of the string.

Returns

Type	Description
string	The string with the masked characters.

regex

This function creates a JSON string that includes all of the matches for the specified regular expression.

Syntax

```
util:regex(string, expression)
```

Argument	Type	Description
string	string	The string to match against the regular expression.
expression	string	The regular expression in ECMAScript grammar.

Returns

Type	Description
JSON string	The JSON string with all of the regular expression matches with index "0" as the whole targeted string.

Cypher Query Language Reference

In addition to SPARQL, Graph Lakehouse also provides Cypher language support, patterned after compatibility with the openCypher community group's language specification for query and update of graph databases. The openCypher community group is an open, multi-vendor initiative aimed at making the Cypher language available as a industry-standard query language for graph databases. Cypher® is a registered trademark of Neo4j, Inc.

This documentation describes compatibility of the Graph Lakehouse Cypher implementation compared to the Cypher language as described in the openCypher community group's Cypher Query Language Reference. See the openCypher Resource page available at <https://www.opencypher.org/resources/>. A PDF copy of the Version 9 Cypher Query Language Reference is available at <https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf>.

Portions of the original Cypher Query Language reference have also been included here for purposes of comparison to note any limitations, restrictions, or exceptions in the Graph Lakehouse Cypher implementation. Typically, comparisons to the Cypher Query language specification are described as **Supported**, **Partially Supported**, and **Not Supported**. Examples included in this documentation also reference the graph data from the original Neo4j Movie dataset. The [Working with Cypher and the Movie Data](#) topic provides an executable script you can use to replicate the data in Graph Lakehouse.

In this section:

Cypher Language Overview	965
Cypher Patterns	971
Cypher Types, Lists, and Maps	976
Comparability, Equality, Orderability, and Equivalence	983
Cypher Expressions, Variables, and Parameters	990
Cypher Operators	993
Cypher Clauses	997
Cypher Functions	1013

Cypher Language Overview

The implementation of Cypher in Graph Lakehouse closely follows the openCypher community group's version 9 specification of the Cypher language. (See the openCypher Resource page available at <https://www.opencypher.org/resources/>. An Acrobat PDF copy of the Cypher Query Language Reference is available at <https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf>.

Important

Some Cypher features, based on the version 9 openCypher specification of the Cypher language, are not yet supported in Graph Lakehouse:

- Uniqueness
- Cypher statements with interleaving of read and update clauses
- The **Merge** clause
- Relational and concatenation operators on list and map types
- The **Unwind** clause with list variables
- Path variable and variable length pattern matching
- Graph Lakehouse UDX and graph algorithms support
- Parameters

For users of previous Graph Lakehouse releases, support for the following new features was added for Graph Lakehouse Version 2.4 and later releases:

- List and map type support
- The **Collect()** aggregate
- Functions operating on list and map types
- List comprehension

- Property chaining and dynamic property access with the '[' operator
- Improved node/relationship variables handling in complex expressions

To use Cypher in Graph Lakehouse, queries and other statements can be sent over the Bolt client-server network protocol to Graph Lakehouse servers. Graph Lakehouse supports Bolt protocol, version 1.0. Port 7088 is the default port used for the Bolt end-point. The `azgbolt` CLI provides a simple way to send Cypher queries to Graph Lakehouse over the Bolt protocol. For example:

```
azgbolt -f query.cql
azgbolt -c "match (m:Movie) return m"
```

Graph data can be queried with SPARQL as well as Cypher language statements; both query modes can co-exist. The Graph Lakehouse Bolt protocol can also be used to execute SPARQL queries, however, in that case, support for various data types in results returned from queries is limited.

Tip

For a brief introduction to using Cypher in Graph Lakehouse, see [Working with Cypher and the Movie Data](#).

In Cypher, `CREATE` statements can be used to load graph data into Graph Lakehouse, which is convenient for loading smaller data sets. For bulk loading of larger RDF/RDF* data sets or non-RDF data, see [Load & Manage Data](#) for more information.

Cypher node and relationship data is represented in RDF* triples format as illustrated in the following example.

Cypher `CREATE` statements:

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Keanu)-[:ACTED_IN {role:'Neo'}]->(TheMatrix)
```

RDF* triple format of data stored in Graph Lakehouse:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
<TheMatrix> rdf:type      <Movie> .
<TheMatrix> <title>       'The Matrix' .
<TheMatrix> <released>    1999 .
<TheMatrix> <tagline>     'Welcome to the Real World' .
<Keanu>      rdf:type      <Person> .
<Keanu>      <name>        'Keanu Reeves' .
<Keanu>      <born>        1964 .
<< <Keanu> <ACTED_IN> <TheMatrix> >> <role> 'Neo' .
```

To query data using Cypher, the **auto_predicate** configuration setting should be enabled before loading data into Graph Lakehouse. Saving information about Cypher nodes requires that the node labels be registered as predicates, so enabling the `auto_predicate` setting ensures that node labels are registered as predicates during any subsequent data load operations.

Cypher Syntax Conventions

The specification of Cypher language syntax provides some difference from that used with SPARQL. The following list describes some specific conventions and styling used in specifying Cypher command syntax and other elements:

- Node labels are case-sensitive, typically specified in CamelCase format, for example, (:NetworkAddress).
- Relationship types are styled in all upper-case, using the underscore character between words, for example: [:ACTED_IN].
- Property keys, variables, parameters, aliases, and functions are case-sensitive and typically styled in CamelCase where the first letter of one of these elements begins with a lower-case letter. Capitalization must match either what is in the database (properties), what is already defined in the query (variables, parameters, aliases), or Cypher definitions (functions).
- Clauses are not case-sensitive, but are typically styled in all capital letters.
- Keywords, such as AND, DISTINCT, IN, CONTAINS, NOT, and others are not case-sensitive but are typically styled in all capital letters.

- Single quotes are typically used to specify literal string values, except when single quotes are part of the string.
- Escaping special characters and spaces in property and label names is done by enclosing the identifier with special characters between single back quote characters (```), for example ``special character``.
- ***Italics*** are used in this document to identify placeholder values that you replace in a Cypher statement.

The Cypher Property Graph Model

The Cypher graph query language operates on property graphs. A property graph is defined as a directed, vertex-labeled, edge-labeled multigraph with self edges, where edges have their own identity. In a property graph, the term ***node*** is used to denote a vertex, and ***relationship*** is used to denote an edge.

The following elements may exist in a property graph:

- Entity
 - Node
 - Relationship
- Path
- Token
 - Label
 - Relationship type
 - Property key
- Property

Entities

An entity has a unique, comparable identity which is assigned a set of properties, each of which are uniquely identified.

Nodes

A node is the basic entity of the graph. In addition:

- A node may be assigned a set of unique labels.
- A node may have zero or more outgoing relationships.
- A node may have zero or more incoming relationships.

Relationships

A relationship is an entity that specifies a directed connection between exactly two nodes, the source node and the target node. In addition:

- An outgoing relationship is a directed relationship from the point of view of its source node.
- An incoming relationship is a directed relationship from the point of view of its target node. A relationship is assigned exactly one relationship type.

In Graph Lakehouse, no two relationships can have the same set of start and end nodes connected by the same relationship type. That is, relationships are uniquely identified by the start node, end node, and relationship type. Also, unique integer identifiers can be associated with nodes, however no integer identifier can be designated for relationships.

Paths

A path represents a walk-through of a property graph consisting of a sequence of alternating nodes and relationships. In addition:

- A path always starts and ends at a node.
- The shortest possible path contains a single node; also called an empty path.
- A path has a length, which is an integer greater than or equal to zero; the length is equal to the number of relationships in the path.

Tokens

A token is a nonempty string of Unicode characters.

Labels

A label is a token that is assigned only to nodes.

Relationship types

A relationship type is an attribute which is only assigned to relationships.

Property keys

A property key is a token which uniquely identifies an entity's property.

Properties

A property is a pair consisting of a property key and a property value. A property value is an instance of one of Cypher's concrete, scalar types, or a list of a concrete, scalar type.

Reserved Keywords

You can escape any Cypher reserved words by enclosing the reserved word between single back quote characters (```), for example ``reserved word``.

Cypher Patterns

Using patterns in Cypher, you can describe the shape of the data you're looking for. Patterns appear in multiple places in Cypher language syntax, such as in MATCH, CREATE, MERGE, and WHERE clauses. This section describes Graph Lakehouse compatibility with Cypher pattern features based on the Cypher Query Language Reference specification:

- [Uniqueness \(Not Supported\)](#)
- [Patterns for Nodes \(Supported\)](#)
- [Patterns for Related Nodes \(Supported\)](#)
- [Patterns for Labels \(Supported\)](#)
- [Specifying Properties \(Supported\)](#)
- [Patterns for Relationships \(Supported\)](#)
- [Variable-length Pattern Matching \(Partially Supported\)](#)
- [Assigning to Path Variables \(Not Supported\)](#)

Uniqueness (Not Supported)

While pattern matching, Cypher makes sure that it does not include matches where the same graph relationship is found multiple times in a single pattern. Cypher specifies the relationship *isomorphism*, that is, the relationship is not repeated in a single path matching the pattern.

Important

Graph Lakehouse's implementation of Cypher currently does not support relationship uniqueness. Thus, it may allow the same relationship to appear multiple times within the matched path. For example, take the following CREATE statement and the subsequent query:

```
CREATE (adam:User {name: 'Adam'}), (pernilla:User {name: 'Pernilla'}),  
(david:User {name: 'David'}),  
(adam)-[:FRIEND]->(pernilla), (pernilla)-[:FRIEND]->(david);
```

```
MATCH (user:User {name: 'Adam'})-[r1:FRIEND]-()-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName;
```

With uniqueness enforced, the expected result might be the following:

```
+-----+
| fofName |
+-----+
| "David" |
+-----+
1 row
```

In Graph Lakehouse, the same query could return the following result:

```
fofName
"Adam"
"David"
2 rows
```

Patterns for Nodes (Supported)

A node in a pattern is simply described using a pair of parentheses, and they are typically given a name. For example:

```
(a)
```

In this example, the pattern describes a single node, and the node is named using the variable **a**.

Patterns for Related Nodes (Supported)

Cypher patterns describe relationships by specifying an arrow between two nodes. For example:

```
(a) - [] -> (b)
```

This pattern describes a very simple data shape, that is, two nodes and a single relationship from one to the other. In this example, the two nodes are named **a** and **b** and the relationship is "directed", as it goes from **a** to **b**.

Patterns for Labels (Supported)

In addition to describing the shape of a node in the pattern, you can also describe its attributes. The simplest attribute that can be described in the pattern is a label that the node must have.

For example:

```
(a:User) - [] -> (b)
```

You can also specify a node that has multiple labels. For example:

```
(a:User:Admin) - [] -> (b)
```

Specifying Properties (Supported)

Nodes and relationships are the fundamental structures in a graph. Cypher allows the use of properties on both of these to let users express far richer models. Properties can be expressed in patterns using a map construct, that is, curly brackets ({ }) surrounding a number of key-expression pairs, separated by commas. For example, a node with two properties would look like:

```
(a {name: 'Andres', sport: 'Brazilian Ju-Jitsu'})
```

When properties appear in patterns, they add an additional constraint to the shape of the data. In the case of a CREATE clause, the properties will be set in the newly-created nodes and relationships.

Patterns for Relationships (Supported)

As described earlier, the simplest way to describe a relationship is by specifying an arrow between two nodes. That way, you can describe that the relationship should exist and also specify its direction. If you do not care about the direction of the relationship, you can omit the arrow head. For example:

```
(a) - [] - (b)
```

As with nodes, relationships may also be given names. In that case, you can insert a pair of square brackets to break up the arrow and specify the variable label within the square brackets. For example:

```
(a) - [r] -> (b)
```

Similar to labels on nodes, relationships can also have types. To describe a relationship with a specific type, you can specify the type following the variable name. For example:

```
(a) - [r:REL_TYPE] -> (b)
```

Unlike labels, relationships can only have one type. However, if a relationship could be one of a set of different types, you can list them all in the pattern, separated by the pipe symbol (|), that is:

```
(a) - [r:TYPE1 | TYPE2] -> (b)
```

Note

Keep in mind that these forms of patterns can only be used to describe existing data (that is, when using a pattern with the MATCH clause, or as an expression).

Also note that, as with nodes, the name of the relationship can always be omitted. For example:

```
(a) - [:REL_TYPE] -> (b)
```

Variable-length Pattern Matching (Partially Supported)

Rather than describing a long path using a sequence of many nodes and relationships in a pattern, the openCypher standard specifies that many relationships (and intermediate nodes) can be described by specifying a length in the relationship description of a pattern. For example:

```
(a) - [*2] -> (b)
```

A range of lengths can also be specified. Such relationship patterns are called "variable-length relationships". For example:

```
(a) - [*3..5] -> (b)
```

The openCypher specification allows several variations in its syntax to specify the length of a relationship path. For example, you can omit either the minimum or maximum relationship path length, or even omit both.

Important

Currently, Graph Lakehouse supports only a few variations of Cypher variable length pattern matching. Graph Lakehouse restrictions are the following:

1. Variable length patterns must include the relationship type. For example:

```
(a) - [:KNOWS*] -> (b)
```

2. Only Zero Or More and One Or More path patterns are supported. For example:

```
(a) - [:KNOWS*] -> (b) , (a) - [:KNOWS*1] -> (b)
```

3. Edge variable projection is not supported, since the list type is not yet supported. For example:

```
(a) - [r:KNOWS*] -> (b)
```

Assigning to Path Variables (Not Supported)

A series of connected nodes and relationships is called a "path". The Cypher specification allows paths to be named using an identifier. For example:

```
p = (a) - [*3..5] -> (b)
```

Important

The current Graph Lakehouse release does not support naming paths.

Cypher Types, Lists, and Maps

This section describes Graph Lakehouse compatibility with the Cypher Language specification for Cypher types, lists, and maps.

- [Types \(Partially Supported\)](#)
- [Type Coercions \(Partially Supported\)](#)
- [Lists \(Supported\)](#)
- [Maps \(Supported\)](#)
- [Working with Null \(Supported\)](#)

Types (Partially Supported)

The Cypher standard specifies data type support in three different categories:

- Property types
- Structural types
- Composite types

Property Types

Property types include the following:

- **NUMBER** – Abstract type, which has INTEGER or FLOAT as subtypes.
- **STRING** – Unicode string type.
- **BOOLEAN** – true and false values. (Cypher uses ternary logic in the WHERE clause; in addition to true and false values, a third state is a null ternary value indicating an indeterminate state.)

Each property type can be returned from Cypher queries, used as parameters, stored as properties, or constructed with Cypher literals.

Structural Types

Structural types include the following:

- **NODE** – comprised of ID, label(s), or a map (of properties).
- **RELATIONSHIP** – comprised of an ID, type, map (of properties), or the ID of the start and end nodes.
- **PATH** – An alternating sequence of nodes and relationships.

Important

The PATH type is not supported in the current Graph Lakehouse release.

Each structural type can be returned from Cypher queries. Structural types cannot be used as parameters, stored as properties, or constructed with Cypher literals.

Note

Nodes, relationships, and paths are returned as a result of pattern matching. Labels are not values but are a form of pattern syntax.

Composite Types

Composite types include:

- **LIST OF T** — is a heterogeneous, ordered collections of values, each of which has any property, structural or composite type T.
- **MAP** is a heterogeneous, unordered collections of key-value pairs, where the key is a string and the value has any property, structural, or composite type.

Composite types can be returned from Cypher queries, used as parameters. or constructed with Cypher literals.

Note

Composite values can also contain null. Composite types cannot be stored as properties.

Type Coercions (Partially Supported)

There are two type coercions described in the Cypher language specification:

- LIST OF NUMBER to LIST OF FLOAT (Not Supported)
- INTEGER to FLOAT

Important

Only the INTEGER to FLOAT coercion is supported in the current Graph Lakehouse release.

Lists (Supported)

The Cypher specification describes support for creating a literal list. You can create a list by using brackets and separating all elements in the list with commas. For example:

```
RETURN [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] AS list
```

This returns the following result:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

To access elements in the list, you can use the square brackets again. For example, with a list of numbers you could use the **range** function, which will extract all numbers between and including a starting and ending number:

```
RETURN range(0, 10) [3]
```

With the range function, you can also specify a negative number in square brackets, to start from the end of a list, rather than from the beginning. For example:

```
RETURN range(0, 10) [-3]
```

Finally, you can use ranges within the square brackets to return a range of values from the list:

```
RETURN range(0, 10) [0..3]
```

List and Pattern Comprehension (Partially Supported)

The Cypher language specification also describes two syntactic constructs for lists, **List Comprehension** and **Pattern Comprehension** (not yet supported).

- List comprehension is a syntactic construct available in Cypher for creating a list based on existing lists. It follows the form of the mathematical set-builder notation (set comprehension) instead of the use of map and filter functions.
- Pattern comprehension (not yet supported) is a syntactic construct available in Cypher for creating a list based on matching a pattern. A pattern comprehension will match the specified pattern just like a normal MATCH clause, with predicates (just like in a normal WHERE clause), but it will yield a custom projection.

List Comprehension (Supported)

List comprehension provides a query construct to create another list based on other existing lists, for example:

```
RETURN [x IN range(0,10) WHERE x % 2 = 0 | x^3] AS result
```

This returns the following result:

```
[0.0,8.0,64.0,216.0,512.0,1000.0]
```

In the previous query, either the WHERE part or the expression can be omitted, if you only want to filter or map results. For example:

```
RETURN [x IN range(0,10) WHERE x % 2 = 0] AS result
```

This query, omitting the expression, returns the following result:

```
[0,2,4,6,8,10]
```

The following query omits the WHERE clause:

```
RETURN [x IN range(0,10) | x^3] AS result
```

This query returns the following result:

```
[0.0,1.0,8.0,27.0,64.0,125.0,216.0,343.0,512.0,729.0,1000.0]
```

Maps (Supported)

The Cypher specification describes how to construct maps using Cypher and construct map projections from nodes, relationships, and other map values.

Literal Maps

The following query example shows how you can create a map based on the Neo4j Movies graph data:

```
RETURN {key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}
```

The result from this query is the following:

```
{key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}
```

```
{listKey -> [{inner -> "Map1"}, {inner -> "Map2"}], key -> "Value"}
```

Map Projections (Partially Supported)

Cypher supports a concept called *map projection* that you can easily construct from nodes, relationships and other map values. A map projection begins with the variable bound to the graph entity to be projected from, and contains a body of comma-separated map elements, enclosed by { and }.

```
map_variable {map_element, [, ...n]}
```

A map element projects one or more key-value pairs to the map projection. There are four different types of map projection elements:

- **Property selector** — Projects the property name as the key, and the value from the map_variable as the value for the projection.
- **Literal entry** — This is a key-value pair, with the value being arbitrary expression key: <expression>.
- **Variable selector** — Not yet supported in current Graph Lakehouse release. Projects a variable, with the variable name as the key, and the value the variable is pointing to as the value of the projection.
- **All-properties selector** — Not yet supported in current Graph Lakehouse release. Projects all key-value pairs from the map_variable value.

Note

If the `map_variable` points to a null value, the whole map projection will evaluate to null.

The following example creates a map projection with a literal entry, which in turn also uses map projection inside the aggregating `collect()`.

```
Query MATCH (actor:Person {name: 'Charlie Sheen'})-[:ACTED_IN]->(movie:Movie) RETURN
actor { .name, .realName, movies: collect(movie { .title, .year })}
```

This query locates 'Charlie Sheen' and return data about him and the movies he has acted in:

```
actor
{name -> "Charlie Sheen", movies -> [{title -> "Apocalypse Now", year -> 1979},
  {title -> "Red Dawn", year -> 1984},{title -> "Wall Street", year -> 1987}],
realName -> "Carlos Irwin Est évez"
```

Note

Two variations of map projections are not yet supported:

- Variable selector
- All properties selector

For example:

```
MATCH (actor:Person {name: 'Charlie Sheen'})[:ACTED_IN]>(movie:Movie)
RETURN actor { .name, .realName, movies: collect(movie { .title, .year })}
```

```
MATCH (actor:Person)[:ACTED_IN]>(movie:Movie)WITH actor, count(movie)
AS nrOfMoviesRETURN actor { .name, nrOfMovies }
```

Working with Null (Supported)

In Cypher, **null** is used to represent missing or undefined values. Conceptually, null represents a missing or unknown value and it is treated somewhat differently from other values. For example, obtaining a property value from a node that does not have that property value defined produces a null. Most expressions that take null as input will also produce a null result. This includes boolean expressions that are used as predicates in the WHERE clause.

Logical Operations with Null

The logical operators (AND, OR, XOR, NOT) treat null as the unknown value of three-valued logic (true, false, and unknown). In this case, null values are interpreted as being false.

The IN Operator and Null

If Cypher determines that a value or element exists in a list, the result returned will be true. Any list that contains a null and doesn't have an element that matches will return null. Otherwise, the result returned will be false.

Expressions That Return Null

The following expressions will return null values:

- Accessing a property that does not exist on a node or relationship, that is, `n.missingProperty`
- Comparisons where either side of the expression is null, for example: `1 < null`
- Arithmetic expressions containing null, for example: `1 + null`
- Function calls where any arguments are null, for example: `sin(null)`

Comparability, Equality, Orderability, and Equivalence

This section describes Graph Lakehouse compatibility with the Cypher Language specification for equality, comparability, and orderability operations.

- [Comparability and Equality \(Partially Supported\)](#)
- [Orderability and Equivalence \(Partially Supported\)](#)
- [Aggregation \(Supported\)](#)

Cypher provides operations around four distinct concepts related to equality and ordering:

- **Comparability:** Comparability is specified by the inequality operators ($>$, $<$, $>=$, $<=$), and determines how to compare two values.
- **Equality :** Equality is specified by the equality operators ($=$, $<>$), and the list membership operator (`IN`) to determine if two values are the same. Equality is also used implicitly by literal maps in node and relationship patterns, since such literal maps provide a shorthand notation for equality predicates.
- **Orderability:** Orderability is specified by the `ORDER BY` clause and determines how to order values.
- **Equivalence:** Equivalence is specified by the `DISTINCT` modifier and by grouping in projection clauses (`WITH`, `RETURN`) to determine if two values are the same.

Besides providing semantics for equality within the primitive types (boolean, string, integer, and float) and maps, Cypher also provides semantics for comparability and orderability for integer, float, and string values within each of the types.

Comparability and Equality (Partially Supported)

Comparability and equality are consistently aligned, that is,

`expr1 = expr2` if and only if `expr1 >= expr2 && expr1 <= expr2`.

If comparison or equality tests involve specific values that evaluate to null, the values are said to be incomparable.

Comparability

Important

List, map, and path type comparisons are not supported in the current Graph Lakehouse release.

Comparability is defined between any pair of values, as specified below.

- General rules
 - Values are only comparable within their most specific type (except for numbers).
 - Equal values are grouped together.
- Numbers
 - Integers are compared numerically in ascending order.
 - Floats (excluding NaN and infinity values) are compared numerically in ascending order.
 - Numbers of different types (excluding NaN and infinity values) are compared to each other as if both numbers would have been coerced to larger precision decimal values before comparing them numerically in ascending order.
 - Positive infinity is of type FLOAT, equal to itself, and greater than any other number (excluding NaN values).
 - Negative infinity is of type FLOAT, equal to itself, and less than any other number (excluding NaN values).
 - NaN values are incomparable.
 - Numbers are not comparable with any value that is not also a number.
- Booleans
 - Booleans are compared such that false is less than true.
 - Booleans are not comparable to any value that is not also a boolean.

- Strings
 - Strings are compared in dictionary order, that is, characters are compared pair-wise, in ascending order, from the start of the string to the end. Characters missing in a shorter string are considered to be less than any other character. For example, 'a' < 'aa'.
 - Strings are not comparable to any value that is not also a string.
- Implementation-specific types
 - Implementations may choose to define suitable comparability rules for values of additional, non-canonical types.
 - Values of an additional, non-canonical type are expected to be incomparable to values of a canonical type.
- Null is not comparable with any other value (including other null values).

Equality

To align equality with comparability, the equality of lists and maps that contain null values is treated in the same way as if they would have been compared outside of those lists and maps, that is, as individual, simple values.

Important

List and map comparisons are not supported in the current Graph Lakehouse release.

Orderability and Equivalence (Partially Supported)

Based on the Cypher language specification, orderability and equivalence are aligned such that **expr1** is equivalent to **expr2** if and only if they have the same position under orderability. As a result, **expr1** and **expr2** are sorted before or after any other non-equivalent value in the same way.

Important

List and map comparisons are not supported in the current Graph Lakehouse release.

Orderability

Orderability is defined between any pair of values such that the result is always true or false. To accomplish this, Cypher defines a pre-determined order of types and ensures that each value falls under exactly one disjoint type in this order.

Important

List and map comparisons are not supported in the current Graph Lakehouse release.

The Cypher language specification prescribes using the following ascending global sort order of disjoint types:

- MAP types
 - Regular map
 - NODE
 - RELATIONSHIP
- LIST OF
- PATH
- STRING
- BOOLEAN
- NUMBER

Note

NaN values are treated as the largest numbers in orderability, placed after any positive infinity values.

- VOID (the type of null)

Using this global sort order, all nodes come before all strings.

The corresponding descending global sort order is the same order, in reverse. That is, the order runs from VOID to MAP. Between values of the same type in the global sort order, orderability defers to comparability, except that equality is overridden by equivalence.

Important

The current release of Graph Lakehouse uses a different type order:

- Void
- Node/Relationship
- Number
- Boolean
- String

Equivalence (Partially Supported)

Equivalence can be defined as being identical to equality except for the following:

- Any two null values are equivalent (both directly or inside nested structures) and, similarly, any two NaN values are also equivalent (both directly or inside nested structures). However, null and NaN values are not equivalent (both directly or inside nested structures).
- Equivalence of lists is identical to equality of lists, but it uses equivalence for comparing the contained list elements.
- Equivalence of regular maps is identical to equality of regular maps, but it uses equivalence for comparing the contained map entries.
- Equivalence is reflexive for all values.

Important

List and map comparisons are not supported in the current Graph Lakehouse release.

Aggregation (Supported)

An aggregation (**aggr(expr)**) processes all matching rows for each aggregation key found in an incoming record (where keys are compared using equivalence). For a fixed aggregation key and each matching record, **expr** is evaluated to a value. This yields a list of candidate values. Generally, the order of candidate values is unspecified. However, if the aggregation happens in a projection with an associated ORDER BY subclause, the list of candidate values is ordered in the same way as the underlying records and as specified by the associated ORDER BY subclause.

In a regular aggregation (that is, of the form **aggr(expr)**), the list of aggregated values is the list of candidate values with all null values removed from it. In a distinct aggregation (that is, **aggr(DISTINCT expr)**), the list of aggregated values is the list of candidate values with all null values removed from it. Furthermore, in a distinct aggregation, only one of all equivalent candidate values is included in the list of aggregated values, that is, duplicates under equivalence are removed. However, if the distinct aggregation happens in a projection with an associated ORDER BY subclause, only one element from each set of equivalent candidate values is included in the list of aggregated values.

Finally, the remaining aggregated values are processed by the actual aggregation function. If the list of aggregated values is empty, the aggregation function returns a default value (null unless otherwise specified; Graph Lakehouse currently returns null). Aggregating values of different types, like summing a number and a string, may lead to runtime errors.

Important

Currently, the SUM of a number and a string will return null in Graph Lakehouse.

The semantics of a few actual aggregation functions depends on the determination of sameness and sorting:

- **count(expr)** returns the number of aggregated values; it returns zero if the list of aggregated values is empty.

- **min/max(expr)** returns the smallest and largest of the aggregated values under orderability. Note that null values will never be returned as a maximum, as they are never included in the list of aggregated values.
- **sum(expr)** returns the sum of aggregated values; it returns zero if the list of aggregated values is empty.
- **avg(expr)** returns the arithmetic mean of aggregated values; it returns zero if the list of aggregated values is empty.
- **collect(expr)** returns the list of aggregated values.
- **stdev(expr)** returns the standard deviation of the aggregated values (assuming they represent a random sample); it returns zero if the list of aggregated values is empty.
- **stdevp(expr)** returns the standard deviation of the aggregated values (assuming they form a complete population); it returns zero if the list of aggregated values is empty.
- **percentile_disc(expr)** computes the inverse distribution function (assuming a discrete distribution model); it returns zero if the list of aggregated values is empty.
- **percentile_cont(expr)** computes the inverse distribution function (assuming a continuous distribution model); it returns zero if the list of aggregated values is empty.

Cypher Expressions, Variables, and Parameters

This section describes Graph Lakehouse compatibility with Cypher expression, variables, and parameter features based on the Cypher Query Language Reference:

- [Expressions \(Supported\)](#)
- [CASE expressions \(Supported\)](#)
- [Variables \(Supported\)](#)
- [Parameters \(Not Supported\)](#)

Expressions (Supported)

Valid expressions in Ciper may include or be specified as any of the following:

- A decimal (integer or double) literal. For example: **13**, **-40000**, **3.14**, **6.022E23**.
- A hexadecimal integer literal (starting with 0x). For example: **0x13zf**, **0xFC3A9**, **-0x66eff**.
- An octal integer literal (starting with zero). For example: **01372**, **02127**, **-05671**.
- A string literal. For example: **'Hello'**, **"World"**.
- A boolean literal. For example: **true**, **false**, **TRUE**, **FALSE**.
- A variable. For example: **n**, **x**, **rel**, **myFancyVariable**, **.**
- A property. For example: **n.prop**, **x.prop**, **rel.thisProperty**, **myFancyVariable**.
- A dynamic property. For example: **n["prop"]**, **rel[n.city + n.zip]**, **map[coll[0]]**.
- A parameter. For example: **\$param**, **\$0**
- A list of expressions. For example: **['a', 'b']**, **[1, 2, 3]**, **['a', 2, n.property, \$param]**, **[]**.
- A function call. For example: **length(p)**, **nodes(p)**.
- An aggregate function. For example: **avg(x.prop)**, **count(*)**.
- A path-pattern. For example: **(a)-[]->()-<[]-(b)**.
- An operator application. For example: **1 + 2** and **3 < 4**.

- A predicate expression that returns true or false. For example: **a.prop = 'Hello', length(p) > 36 10, exists(a.name).**
- A case-sensitive string matching expression. For example: **a.surname STARTS WITH 'Sven', a.surname ENDS WITH 'son' or a.surname CONTAINS 'son'**
- A CASE expression.

Escape Characters

String literals can contain the following escape sequences:

Character	Description
\t	Tab
\b	Backspace
\n	Newline
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\	Backslash
\uxxxx	Unicode UTF-16 code point (4 hex digits must follow the \u)
\Uxxxxxxxx	Unicode UTF-32 code point (8 hex digits must follow the \U)

CASE expressions (Supported)

Generic conditional expressions may be expressed using the well-known CASE construct. For example:

```
CASE test
WHEN value THEN result
  [WHEN ...]
  [ELSE default]
END
```

Two variants of CASE exist within Cypher: the simple form, which allows an expression to be compared against multiple values, and the generic form, which allows multiple conditional statements to be expressed.

Variables (Supported)

When you reference parts of a pattern or a query, you do so by naming them. The names you give the different parts are called variables. For example:

```
MATCH (n) -[]->(b) RETURN b
```

In this example, the variables are **n** and **b**.

Variable names are case-sensitive, and can contain underscores and alphanumeric characters (a-z, 0-9), but must always start with a letter. To include other characters are needed, you can escape them with the single back quote (`) character. The same rule applies to property names.

Parameters (Not Supported)

The Cypher language specification supports querying with parameters. However, the current Graph Lakehouse release does not support them.

Cypher Operators

This section describes Graph Lakehouse compatibility with Cypher operators based on the Cypher Query Language Reference specification:

- [General Operators \(Supported\)](#)
- [Mathematical Operators \(Supported\)](#)
- [Comparison Operators \(Supported\)](#)
- [Boolean Operators \(Supported\)](#)
- [String Operators \(Partially Supported\)](#)
- [List Operators \(Partially Supported\)](#)
- [Equality and Comparison of Values \(Partially Supported\)](#)
- [Ordering and Comparison of Values \(Supported\)](#)
- [Chaining Comparison Operations \(Supported\)](#)

General Operators (Supported)

General operators include:

- **DISTINCT** – removes duplicates values.
- **Dot operator** – access the property of a node, relationship or literal map.
- **Subscript operator ([])** – provides dynamic property access.

Mathematical Operators (Supported)

The mathematical operators supported in Cypher are the following:

- **addition (+)**
- **subtraction or unary minus (-)**
- **multiplication (*)**
- **division (/)**

- modulo division (%)
- exponentiation (^)

Comparison Operators (Supported)

Cypher comparison operators include the following:

- equality (=)
- inequality (<>)
- less than (<)
- greater than (>)
- less than or equal to (<=)
- greater than or equal to (>=)
- IS NULL
- IS NOT NULL

String-specific comparison operators in Cypher include the following:

- STARTS WITH – provides case-sensitive prefix searching on strings.
- ENDS WITH – provides case-sensitive suffix searching on strings.
- CONTAINS – provides case-sensitive inclusion searching in strings.

Boolean Operators (Supported)

Cypher Boolean operators, also referred to as logical operators, include the following:

- conjunction – AND
- disjunction – OR
- exclusive disjunction – XOR
- negation – NOT

String Operators (Partially Supported)

The sole string operator that Cypher supports is the plus sign (+) concatenation operator.

Important

The plus sign (+) string concatenation operator is not supported in the current Graph Lakehouse release. (The CONCAT() function is supported to perform the same function.)

List Operators (Partially Supported)

Cypher list operators include the concatenation plus sign (+) operator and the IN operator that checks if an element exists in a list.

Important

List concatenation with the plus sign (+) is not supported in the current Graph Lakehouse release.

Equality and Comparison of Values (Partially Supported)

Cypher supports comparing values for equality using the equals (=) and less-than-greater-than, not equals (<>) operators. Values of the same type are only equal if they have the same identical value, for example, 3 = 3.

Important

List and map comparisons are not supported in the current Graph Lakehouse release.

Values of different types are considered equal based on the following rules:

- Paths are treated as lists of alternating nodes and relationships; they are considered equal to all lists that contain that very same sequence of nodes and relationships.
- Testing any value against null with either the equals (=) or the less-than-great-than, not equal (<>) operators always returns null. This includes null = null and null <> null. The only

way to reliably test if a value *v* is null is by using the special **v IS NULL** or **v IS NOT NULL** equality operators.

- Maps are only equal if they map exactly the same keys to equal values; lists are only equal if they contain the same sequence of equal values, for example: `[3, 4] = [1+2, 8/2]`.

All other combinations of value types cannot be compared with each other. Nodes, relationships, and literal maps also cannot be compared with each other. Attempting to specify comparisons of values that cannot be compared will return an error.

Ordering and Comparison of Values (Supported)

The following comparison operators are used to compare values for ordering:

- `<=`
- `<` (for ascending)
- `>=`, and `>` (for descending)

The following details describe how the comparisons are performed:

- Numerical values are compared for ordering using numerical order. For example, **3 < 4** is true.
- String values are compared for ordering using lexicographic order. For example, **"x" < "xy"**.
- Boolean values are compared for ordering, that is **false < true**.
- Comparing for ordering when one argument is null returns null. For example, **null < 3** is null.

Specifying comparisons for ordering of other types of values will return an error.

Chaining Comparison Operations (Supported)

Comparisons can be chained together arbitrarily. For example, **x < y <= z** is equivalent to **x < y AND y <= z**. As a general practice, if **a, b, c, ..., y, z** are expressions and **op1, op2, ..., opN** are comparison operators, then **a op1 b op2 c ... y opN z** is equivalent to **a op1 b** and **b op2 c** and ... **y opN z**.

Cypher Clauses

This section describes Graph Lakehouse compatibility with Cypher commands based on the Cypher Query Language Reference:

- [MATCH \(Supported\)](#)
- [OPTIONAL MATCH \(Supported\)](#)
- [MANDATORY MATCH \(Not Supported\)](#)
- [RETURN \(Supported\)](#)
- [WITH \(Supported\)](#)
- [UNWIND \(Partially Supported\)](#)
- [WHERE \(Supported\)](#)
- [ORDER BY \(Supported\)](#)
- [SKIP \(Supported\)](#)
- [LIMIT \(Supported\)](#)
- [CREATE \(Partially Supported\)](#)
- [DELETE \(Supported\)](#)
- [SET \(Partially Supported\)](#)
- [REMOVE \(Supported\)](#)
- [MERGE \(Not supported\)](#)
- [CALL \[...YIELD\] \(Not Supported\)](#)
- [UNION and UNION ALL \(Supported\)](#)
- [State Visibility and Behavior between Clauses \(Partially Supported\)](#)

MATCH (Supported)

The Cypher MATCH clause allows you to specify the patterns that Cypher will search for in data. The MATCH clause is often used with a WHERE clause that adds restrictions or predicates to the MATCH pattern. In that case, the predicates are part of the pattern description, and should not be considered just a filter applied only after the matching is done.

The MATCH clause can be included at the beginning of a query or later, for example, as part of a WITH clause. If it is the first clause in a statement, no data will have been bound yet to the result, and Cypher will search to find the results matching the pattern in the MATCH clause and any associated predicates specified in any WHERE clause. This could involve a scan of the database, a search for nodes of a certain label, or a search of an index to find starting points for the pattern matching. Nodes and relationships found by this search are available as bound pattern elements, and can be used for pattern matching of sub-graphs. They can also be used in any further MATCH clauses, where Cypher will use the known elements, and find further unknown elements from there. Predicates in a WHERE clause can be evaluated before pattern matching, during pattern matching, or after finding matches.

Finding All Nodes (Supported)

By specifying a MATCH pattern with just a single node and no labels, all nodes in the graph will be returned. For example:

```
MATCH (n)
RETURN n
```

This example returns all nodes in the database.

Finding All Nodes with a Label (Supported)

To return all nodes with a label, you can specify a single node pattern where the node has a label on it. For example:

```
MATCH (movie:Movie)
RETURN movie.title
```

This example returns all the movies in the database.

Finding Related Nodes (Supported)

You can use the notation (`-[]-`) to find related nodes, without regard to the type or direction of their relationship. For example:

```
MATCH (director {name: 'Oliver Stone'})-[]-(movie)
RETURN movie.title
```

This example returns all the movies directed by 'Oliver Stone'.

Finding Matches with Labels (Supported)

To specify a pattern to return only nodes with labels, you can add the label syntax to your node match pattern. For example:

```
MATCH (:Person {name: 'Oliver Stone'})-[]-(movie:Movie)
RETURN movie.title
```

This example returns only those nodes connected with the Person 'Oliver' that are labeled Movie.

Finding Matches Based on the Direction of Relationships (Supported)

When you want to specify the direction of a relationship in a pattern match, you can use the directional notation, (`->`) or (`<-`). For example:

```
MATCH (:Person {name: 'Oliver Stone'})-[]->(movie)
RETURN movie.title
```

This example returns any nodes connected with the Person 'Oliver' by an outgoing relationship.

Directed Relationships and Variables (Supported)

Cypher allows you to use variables in MATCH queries, either for filtering on properties of a relationship, or to return the relationship. For example:

```
MATCH (:Person {name: 'Oliver Stone'})-[r]->(movie)
RETURN type(r)
```

This example returns the type of each outgoing relationship from 'Oliver'.

Specifying Matches Based on Relationship Type (Supported)

You can specify matches based on the relationship type by entering a colon followed by the relationship type. For example:

```
MATCH (wallstreet:Movie {title: 'Wall Street'})<-[:ACTED_IN]-(actor)
RETURN actor.name
```

This example returns all actors that ACTED_IN 'Wall Street'.

Specifying Matches Based on Multiple Relationship Types (Supported)

To specify a match based on multiple relationship types, you can combine the different relationship types with the pipe (|) symbol. For example:

```
MATCH (wallstreet {title: 'Wall Street'})<-[:ACTED_IN|:DIRECTED]-(person)
RETURN person.name
```

This example returns nodes with an ACTED_IN or DIRECTED relationship to 'Wall Street'.

Specifying Matches on the Relationship Types Using Variables (Supported)

Cypher also allows you to use a variable in pattern matches based on the relationship type and then return the relationship in the variable. For example:

```
MATCH (wallstreet {title: 'Wall Street'})<-[r:ACTED_IN]-(actor)
RETURN r.role
```

This example returns the ACTED_IN roles for the movie 'Wall Street'.

Specifying Matches for Relationship Types That Contain Non-letter Characters (Supported)

If your database contains relationship types that include non-letter characters or spaces, you can use the single back quote (`) character to escape the type. For example, to demonstrate this, you could add an additional relationship between 'Charlie Sheen' and 'Rob Reiner':

```
MATCH (charlie:Person {name: 'Charlie Sheen'}),
      (rob:Person {name: 'Rob Reiner'})
CREATE (rob)-[:`TYPE WITH SPACE`]->(charlie)
```

This example returns a relationship type with a space in it:


```
type(r)
"TYPE WITH SPACE"

1 row
```

Finding Matches with Multiple Relationships (Supported)

To find matches for multiple relationships, you can specify the relationship match pattern using the form:

```
(- [ ] -)
```

You could then string them together in a single MATCH statement. For example:

```
MATCH (charlie {name: 'Charlie Sheen'})-[:ACTED_IN]->(movie)<-[:DIRECTED]-(director)
RETURN movie.title, director.name
```

This example returns the movie that 'Charlie Sheen' acted in and also returns its director's name.

Variable-length relationships (Partially Supported)

Based on the Cypher Language specification, nodes that are a variable number of **relationship** -> **node** hops away can be found using the following syntax:

```
-[:TYPE*minHops..maxHops]->.
```

In this case, minHops and maxHops are optional and default to 1 and infinity. When no bounds are given, the dots may be omitted. The dots may also be omitted when setting only one bound and this implies a fixed-length pattern.

Important

Currently, Graph Lakehouse supports only a few variations of variable length pattern matching:

- Variable length patterns must include the relationship type. For example:

```
(a) - [:KNOWS*] -> (b)
```

- Only ZeroOrMore and OneOrMore path patterns are supported. For example:

```
(a) - [:KNOWS*] -> (b) , (a) - [:KNOWS*1] -> (b)
```

Edge variable projection is not supported, since the list type is not currently supported in Graph Lakehouse.

Using Relationship Variable in Variable-length Relationships (Supported)

When the connection between two nodes is of variable length, the list of relationships comprising the connection can be returned using the following syntax:

```
MATCH p = (actor {name: 'Charlie Sheen'})-[:ACTED_IN*2]-(co_actor)
RETURN relationships(p)
```

This example returns a list of relationships.

Match with Properties on a Variable-length Path (Supported)

A variable-length relationship with properties defined on it means that all relationships in the path must have the property set to the given value.

Zero-length Paths (Supported)

Using variable-length paths that have the lower bound set to zero means that two variables can point to

the same node. If the path length between two nodes is zero, they are, by definition, the same node.

Note that when matching zero-length paths, the result may contain a match even when matching on a relationship type that is not in use. For example:

```
MATCH (wallstreet:Movie {title: 'Wall Street'})-[*0..1]-(x)
RETURN x
```

This example returns the movie itself as well as actors and directors one relationship away.

Named Paths (Supported)

If you want to return or specify a filter on a path in your pattern graph, you can introduce a named path. For example:

```
MATCH p = (michael {name: 'Michael Douglas'})-[]->()
RETURN p
```

This example returns the two paths starting from 'Michael Douglas'.

Matching on a Bound Relationship (Supported)

When a pattern contains a bound relationship, and that relationship pattern does not specify direction, Cypher will attempt to match the relationship in both directions. For example:

```
MATCH (a)-[r]-(b)
WHERE id(r) = 0
RETURN a,b
```

This example returns the two connected nodes, the start node, and the end node.

Finding the Single Shortest Path (Not Supported)

You can use the **shortestPath()** function to find a single shortest path between two nodes. For example:

```
MATCH (martin:Person {name: 'Martin Sheen'}), (oliver:Person {name: 'Oliver Stone'}),
p = shortestPath((martin)-[*..15]-(oliver))
RETURN p
```

Important

Finding the single shortest path operation is not supported in the current Graph Lakehouse release.

Finding All Shortest Paths (Not Supported)

You can use the **allShortestPaths()** function to find all the shortest paths between two nodes. For example:

```
MATCH (martin:Person {name: 'Martin Sheen'}), (michael:Person {name: 'Michael Douglas'}), p = allShortestPaths((martin)-[*]-(michael))
RETURN p
```

Important

Operations to find all shortest paths are not supported in the current Graph Lakehouse release.

Finding Nodes by ID (Supported)

You can use the `id()` function in a predicate to search for nodes. For example

```
MATCH (n)
WHERE id(n) = 0
RETURN n
```

This example returns the corresponding node.

Finding a Relationship by ID (Not Supported)

Based on the Cypher language specification, you can use the `id()` function in a predicate to search for relationships. For example:

```
MATCH ()-[r]->()
WHERE id(r) = 0
RETURN r
```

Important

Finding a relationship by ID is not supported in the current Graph Lakehouse release.

Finding Multiple Nodes by ID (Supported)

You can use the `id()` function with the `IN` clause in a predicate to find multiple nodes by ID. For example:

```
MATCH (n)
WHERE id(n) IN [0, 3, 5]
RETURN n
```

This example returns the nodes listed in the `IN` expression.

OPTIONAL MATCH (Supported)

This clause is used to specify the patterns to search for, while using nulls for missing parts of the pattern.

Optional Relationships (Supported)

If a relationship is optional, you can use the OPTIONAL MATCH clause to find relationships, similar to how an outer join works in SQL. Cypher returns the relationship if it is found; otherwise a null is returned.

```
MATCH (a:Movie {title: 'Wall Street'})
OPTIONAL MATCH (a)-[]->(x)
RETURN x
```

This example returns null, since the node has no outgoing relationships.

Returning Null for Null Properties on Optional Elements (Supported)

Returning a property from an optional element that is null will also return null. For example:

```
MATCH (a:Movie {title: 'Wall Street'})
OPTIONAL MATCH (a)-[]->(x)
RETURN x, x.name
```

This example will return the x element (null in this query), and null as its name.

Optional Typed and Named Relationships (Supported)

Just as with a normal relationship, you can decide which variable a relationship goes into, and what relationship type you want to return. For example:

```
MATCH (a:Movie {title: 'Wall Street'})
OPTIONAL MATCH (a)-[r:ACTS_IN]->()
RETURN a.title, r
```

This example returns the title of the node, that is, 'Wall Street'. If the node has no outgoing ACTS_IN relationships, null is returned for the relationship denoted by r.

MANDATORY MATCH (Not Supported)

The Cypher MANDATORY MATCH clause lets you specify the patterns to search for.

Important Not supported in the current Graph Lakehouse release.

RETURN (Supported)

The RETURN clause specifies what to include in a query result set. Based on the Cypher language specification, any expression, literals, predicates, properties, and functions, can be used as a return item. For example:

```
MATCH (a {name: 'A'})  
RETURN a.age > 30, "I'm a literal", (a)-[]->()
```

Important

In the current Graph Lakehouse release, the pattern expression “**RETURN (a)-[]->()**” is not supported.

WITH (Supported)

The WITH clause allows queries to be chained together, piping the results from one query to be used as the starting point or search criteria for the next query.

UNWIND (Partially Supported)

The UNWIND clause expands a list into a sequence of records.

Important

In the current Graph Lakehouse release, UNWIND is currently only supported for operation on a list of literals. For example:

```
UNWIND [1, 2, 3] AS xRETURN x
```

WHERE (Supported)

The WHERE clause adds constraints to the patterns in a MATCH or OPTIONAL MATCH clause or used to filter the results of a WITH clause.

Filter on Dynamically-computed Node Property (Supported)

Based on the Cypher language specification, you can use square bracket syntax to filter on a property using a dynamically-computed name. For example:

```
WITH 'AGE' AS propname
MATCH (n)
WHERE n[toLower(propname)] < 30
RETURN n.name, n.age
```

Checking for the Existence of a Property (Supported)

Based on the Cypher language specification, you can use the **exists()** function to only include in results the nodes or relationships in which a property exists.. For example:

```
MATCH (n)
WHERE exists(n.belt)
RETURN n.name, n.belt
```

Filter on Patterns with Properties (Supported)

Based on the Cypher language specification, you can add properties to filter patterns. For example:

```
MATCH (n)
WHERE (n)-[:KNOWS]-({name: 'Tobias'})
RETURN n.name, n.age
```

ORDER BY (Supported)

An ORDER BY clause following RETURN or WITH specifies that the output should be sorted in either ascending (the default) or descending order.

Ordering Null (Not Supported)

Based on the Cypher language specification, when sorting the result set, null values will always be placed at the end of the result set with ascending sorting, and first in the result when doing descending sort.

```
Query
MATCH (n)
RETURN n.length, n.name, n.age ORDER BY n.length
```

Important

Orderability across types and null values is not supported in the current Graph Lakehouse release.

SKIP (Supported)

The SKIP clause specifies the record to start including in output records.

Using an Expression with SKIP to Return a Subset of the Rows (Not Supported)

Based on the Cypher language specification, SKIP accepts any expression that evaluates to a positive integer, as long as it is not referring to any external variables. For example:

```
MATCH (n)
RETURN n.name ORDER BY n.name
SKIP toInteger(3*rand())+ 1
```

Important

Specifying a constant expression in the SKIP clause is not supported in the current Graph Lakehouse release.

LIMIT (Supported)

The LIMIT clause specifies the maximum number of records to include in output results.

Using an Expression with LIMIT to Return a Subset of the Rows (Partially Supported)

Based on the Cypher language specification, LIMIT accepts any expression that evaluates to a positive integer, as long as it is not referring to any external variables:

```
MATCH (n)
RETURN n.name
ORDER BY n.name
LIMIT toInteger(3 * rand())+ 1
```

Important

Specifying a constant expression in the LIMIT clause is not supported in the current Graph Lakehouse release.

CREATE (Partially Supported)

The CREATE clause is used to create nodes and relationships.

Return Created Node (Not Supported)

Based on the Cypher language specification, you can use RETURN to return the name and details about newly created nodes. For example:

```
CREATE (a {name: 'Andres'})
RETURN a
```

Important

An update statement (CREATE, DELETE, SET, or REMOVE) followed by RETURN is not supported in the current Graph Lakehouse release.

Create Node with a Parameter for the Properties (Not Supported)

Based on the Cypher language specification, you can also create a graph entity from a map.

All the key-value pairs in the map will be set as properties on the created relationship or node.

Important Use of parameters is not currently supported in Graph Lakehouse.

Create Multiple Nodes with a Parameter for Their Properties (Not Supported)

Based on the Cypher language specification, if you provide Cypher with an array of maps, it will create a node for each map.

Important

The current Graph Lakehouse release does not allow you to use multiple update clauses (CREATE/DELETE/SET/REMOVE) in a statement. See [State Visibility and Behavior between Clauses \(Partially Supported\)](#).

DELETE (Supported)

The DELETE clause lets you specify nodes, relationships or paths to delete. Any node to be deleted must also have all associated relationships explicitly deleted.

The DETACH DELETE clause lets you delete a node or set of nodes. All associated relationships will automatically be deleted.

SET (Partially Supported)

The SET clause can be used to update labels on nodes and properties on nodes and relationships.

Copying Properties between Nodes and Relationships (Not Supported)

Based on the Cypher language specification, you can also use SET to copy all properties from one graph element to another. Doing this also removes all other properties on the receiving graph element.

Important

Copying properties between nodes and relationships is currently not supported in Graph Lakehouse.

Set a Property Using a Parameter (Not Supported)

Based on the Cypher language specification, you can use a parameter to specify the value of a property.

Important

The use of parameters is currently not supported in Graph Lakehouse.

Set All Properties Using a Parameter (Not Supported)

Based on the Cypher language specification, you can replace all existing properties on a node with a new set of properties provided by the parameter.

Important

The use of parameters is currently not supported in Graph Lakehouse.

REMOVE (Supported)

The REMOVE clause lets you remove properties and labels from nodes and relationships.

MERGE (Not supported)

The MERGE clause ensures that a pattern exists in the graph. Either the pattern already exists, or if it does not already exist, it will be created.

Important

MERGE operations are not currently supported in Graph Lakehouse.

CALL [...YIELD] (Not Supported)

The CALL [...YIELD] clause lets you invoke a procedure and return any results.

Important

Cypher CALL [...YIELD] operations are not currently supported in Graph Lakehouse.

UNION and UNION ALL (Supported)

The UNION and UNION ALL clauses are used to combine the result of multiple queries into a single result set. UNION combines the results of two or more queries into a single result set that includes all the records that belong to all queries in the union. The number and the names of the fields must be identical in all queries combined by using UNION.

When using the UNION clause, it will combine and remove duplicates from the result set. To keep all the result records, you can use UNION ALL.

Combining Two Queries and Removing Duplicates (Supported)

By using the UNION clause without the ALL keyword, duplicates are removed from the combined result set. For example:

```
MATCH (n:Actor)
RETURN n.name AS name
UNION
MATCH (n:Movie)
RETURN n.title AS name
```

The combined result is returned, without duplicates.

Combining Two Queries and Retaining Duplicates (Supported)

You can combine the results from two queries, and keep duplicate records in the result, by using `UNION ALL`. For example:

```
MATCH (n:Actor)
RETURN n.name AS name
UNION ALL MATCH (n:Movie)
RETURN n.title AS name
```

The combined result is returned, including duplicates.

State Visibility and Behavior between Clauses (Partially Supported)

Based on the Cypher Language specification, Cypher allows clauses that read data from a graph to be interleaved with clauses that write to the graph. Some Cypher clauses can both read from a graph and write to it at the same time. Explicit state change visibility makes it possible to understand queries without having to worry about ordering of updates and reads.

Important

There is a restriction on state visibility and statement interleaving in the current Graph Lakehouse release: `READ`, `UPDATE`, or `READ` statements may be followed by `UPDATE`. However, a `READ` clause should not follow the `UPDATE` clause.

Cypher Functions

This section describes Graph Lakehouse compatibility with the Cypher Language specification for Cypher functions.

- [Predicate Functions \(Supported\)](#)
- [Scalar Functions \(Partially Supported\)](#)
- [Aggregating Functions \(Supported\)](#)
- [List Functions \(Supported\)](#)
- [Mathematical Numeric Functions \(Partially Supported\)](#)
- [Mathematical Logarithmic Functions \(Partially Supported\)](#)
- [Mathematical Trigonometric Functions \(Partially Supported\)](#)
- [String Functions \(Partially Supported\)](#)
- [User-defined Functions \(Not Supported\)](#)
- [Comments \(Supported\)](#)
- [Compatibility and Versioning \(Not Supported\)](#)

Predicate Functions (Supported)

These functions return either true or false for the given arguments:

- **exists()**: Returns `true` if the specified property exists in the node, relationship, or map.

Scalar Functions (Partially Supported)

These functions return a single value:

- **coalesce()**: Returns the first non-null value in a list of expressions.
- **endNode()**: Returns the end node of a relationship.
- **head()**: Returns the first element in a list.

- **id()**: Returns the id of a relationship or node.

Important

In Graph Lakehouse, nodes have a unique integer identifier, however, relationships do not; so **id()** on relationships is not available. Relationships can, however, be uniquely identified by their start and end node IDs and relationship type.

- **last()**: Returns the last element in a list.
- **length()**: Returns the length of a path.

Important

Paths are not supported in the current Graph Lakehouse release, so functions on path arguments are also not supported.

- **properties()**: Returns a map containing all the properties of a node or relationship.
- **size()**: Returns the number of items in a list. When applied to a pattern expression, the function returns the number of sub-graphs matching the pattern expression. When applied to a string, the function returns the size of a string.
- **startNode()**: Returns the start node of a relationship.
- **timestamp()**: Returns the difference, measured in milliseconds, between the current time and midnight January 1 1970 UTC.

Important

Date/time functions are not supported in the current Graph Lakehouse release.

- **toBoolean()**: Converts a string value to a boolean value.
- **toFloat()**: Converts an integer or string value to a floating point number.
- **toInteger()**: Converts a floating point or string value to an integer value.
- **type()**: Returns the string representation of the relationship type.

Aggregating Functions (Supported)

Aggregating functions accept multiple values as arguments and calculate and return an aggregated result value.

- **avg()**: Returns the average of a set of numeric values.
- **collect()**: Returns a list containing the values returned by an expression.
- **count()**: Returns the number of values or records.
- **max()**: Returns the maximum value in a set of values.

Important

The `max()` function is not supported on List types in the current release.

- **min()**: Returns the minimum value in a set of values.

Important

In the current Graph Lakehouse release, there is a result mismatch for `min(val)` due to different orderability behavior. The `min()` function on List types is also not supported in the current release.

- **percentileCont()**: Returns the percentile of a value over a group using linear interpolation.

Important

The `percentileCont()` function is partially supported in the current Graph Lakehouse release, with the `percentileCont()` function supported with the GROUP BY clause.

- **percentileDisc()**: Returns the nearest value to a specified percentile over a group using a rounding method.

Important

The `percentileDisc()` function is partially supported in the current Graph Lakehouse release, with the `percentileDisc()` function supported with the `GROUP BY` clause.

- **stDev()**: Returns the standard deviation for a given value over a group for a sample of a population.
- **stDevP()**: Returns the standard deviation for a given value over a group for an entire population.
- **sum()**: Returns the sum of a set of numeric values.

List Functions (Supported)

Cypher List functions include the following:

- **keys()**: Returns a list containing the string representations for all the property names of a node relationship or map.
- **labels()**: Returns a list containing the string representations for all the labels of a node.
- **nodes()**: Returns a list containing all the nodes in a path.

Important

Paths are not supported in the current Graph Lakehouse release, so functions on path arguments are also not supported.

- **range()**: Returns a list comprising all integer values within a specified range.
- **relationships()**: Returns a list containing all the relationships in a path.

Important

Paths are not supported in the current Graph Lakehouse release, so functions on path arguments are also not supported.

- **reverse()**: Returns a list in which the order of all elements in the original list have been reversed.
- **tail()**: Returns all but the first element in a list.

Mathematical Numeric Functions (Partially Supported)

Cypher mathematical numeric functions all operate only on numeric expressions. They will return an error if used with any other values.

- **abs()**: Returns the absolute value of a number.
- **ceil()**: Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.
- **floor()**: Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.
- **rand()**: Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); that is [0 ; 1).
- **round()**: Returns the value of a number rounded to the nearest integer.
- **sign()**: Returns the sign of a number: returns 0 if the number is 0; returns -1 for any negative number; and returns 1 for any positive number.

Important The `sign()` function is not supported in the current release.

Mathematical Logarithmic Functions (Partially Supported)

Cypher mathematical logarithmic functions all operate only on numeric expressions. They will return an error if used with any other values.

- **e()**: Returns the base of the natural logarithm.

Important The `e()` function is not supported in the current release.

- **exp()**: Returns e^n where e is the base of the natural logarithm and n is the value of the argument expression.
- **log()**: Returns the natural logarithm of a number.
- **log10()**: Returns the common logarithm (base 10) of a number.
- **sqrt()**: Returns the square root of a number.

Mathematical Trigonometric Functions (Partially Supported)

Cypher mathematical trigonometric functions all operate only on numeric expressions. They will return an error if used with any other values.

Important

The **acos**, **asin**, **atan**, **atan2**, **cot**, and **degrees** functions are not supported in the current Graph Lakehouse release.

- **cos()**: Returns the cosine of a number.
- **pi()**: Returns the mathematical constant pi.
- **radians()**: Converts degrees to radians.
- **sin()**: Returns the sine of a number.
- **tan()**: Returns the tangent of a number.

String Functions (Partially Supported)

Cypher string functions all operate only on string expressions. They will return an error if used with any other values. The exception to this rule is `toString()`, which also accepts numbers and boolean values as arguments.

- **left()**: Returns a string containing the specified number of left-most characters of the original string.
- **lTrim()**: Returns the original string with leading whitespace removed.

- **replace()**: Returns a string in which all occurrences of a specified string in the original string have been replaced by another specified string.
- **reverse()**: Returns a string in which the order of all characters in the original string have been reversed.
- **right()**: Returns a string containing the specified number of rightmost characters of the original string.
- **rTrim()**: Returns the original string with trailing white space removed.
- **split()**: Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.
- **substring()**: Returns a substring of the original string, beginning with a zero-based index start and length.
- **toLowerCase()**: Returns the original string in lowercase.
- **toString()**: Converts an integer, float, or boolean value to a string.
- **toUpperCase()**: Returns the original string in uppercase.
- **trim()**: Returns the original string with leading and trailing white space removed.

User-defined Functions (Not Supported)

The use of user-defined functions is not supported in the current release.

Comments (Supported)

Comments may be added to queries. Single line or inline comments begin with `//`, and multi-line comments are delimited by `/*` and `*/`. For example:

```
MATCH (n) RETURN n // This is an end of line comment.
```

Compatibility and Versioning (Not Supported)

The use of previous compatible version selection is not supported in the current release.

Admin

This section provides information about managing the configuration and administration of Graph Lakehouse.

In this section:

- [Start and Stop Graph Lakehouse](#) 1021
- [Deploy the Frontend Container](#) 1023
- [Authentication and Access Control](#) 1031
- [Manage the Server Configuration](#) 1064

Start and Stop Graph Lakehouse

The Query & Admin Console provides options to stop and start Graph Lakehouse. The following information provides instructions for starting and stopping Graph Lakehouse from the file system when the Query & Admin Console is not included in the deployment or is unavailable.

Note

The system management daemon, **azgmgrd**, should remain running at all times. When you restart the database, do not stop and start the daemon. There are three circumstances that require you to restart azgmgrd:

1. When changing a system configuration setting value that requires a restart of the system management daemon, such as `sysmgr_port` or `auto_restart_max_attempts`.
2. When upgrading or re-installing Graph Lakehouse in a RHEL/Rocky environment.
3. When making changes to the `<install_path>/config/ip_addrs.conf` file to add or remove servers from a cluster in a RHEL/Rocky environment.

Follow the appropriate instructions below, depending on the current state of Graph Lakehouse and your use case:

- [Stop the Database and Leave the System Management Daemon Running](#)
- [Start the Database \(the System Management Daemon is Running\)](#)
- [Stop the Database and the System Management Daemon](#)
- [Start the System Management Daemon and the Database](#)
- [Reinitialize the Database](#)

Stop the Database and Leave the System Management Daemon Running

To stop the database, run the following command from the **leader server**:

```
sudo systemctl stop anzograph
```

If queries are running, the system manager waits the number of seconds in `stop_timeout` (the default value is 30 seconds) for any outstanding queries to complete and then stops the database.

Start the Database (the System Management Daemon is Running)

To start the database, run the following command from the **leader server**:

```
sudo systemctl start anzograph
```

Stop the Database and the System Management Daemon

To stop the database and system management daemon, run the following commands from the **leader server**:

```
sudo systemctl stop anzograph
```

```
sudo systemctl stop azgmgrd
```

Start the System Management Daemon and the Database

To start the system management daemon, run the following command. On clusters, run the command on **each server in the cluster**:

```
sudo systemctl start azgmgrd
```

To start the database after the system management daemon is running, run the following command on the **leader node**:

```
sudo systemctl start anzograph
```

Reinitialize the Database

If you need to reinitialize the database to remove the generated code and any persisted data, run the following command. The system management daemon (azgmgrd) should be running.

```
/<install_path>/bin/azgctl -start -init
```

Deploy the Frontend Container

This topic provides instructions for deploying the frontend container with Docker for Linux and then connecting the frontend to your existing cluster. For information on installing the frontend using the RHEL/Rocky installer, see [Enterprise Linux 9 Deployments](#).

Follow the instructions below to deploy the Graph Lakehouse frontend console on Docker for Linux.

Tip

You might want to follow the steps in [Post-installation steps for Linux](#) to make sure that a non-root user can run Docker commands and you do not need to include "sudo" in the commands below.

1. If necessary, start Docker with `sudo systemctl start docker`.

Note

Docker caches images on the Docker host. If you have deployed a front end container previously, that image is cached on the host and can be used to redeploy the front end console. If you want to deploy the latest release, first pull the latest image. To do so, run the following command, and then proceed to the next step.

```
docker pull cambridgesemantics/anzograph-frontend:latest
```

You can deploy alternate front end console versions by replacing the "latest" tag with any of the tags that are available on the [Graph Lakehouse Frontend Docker Hub](#) site.

2. Run the following Docker command to deploy the Graph Lakehouse front end container image. The command instructs Docker to start the container and configure HTTP and HTTPS access to the application by mapping the container ports to the HTTP and HTTPS ports on the local host:

```
docker run -d -p host_http_port:8080 -p host_https_port:8443 --name=container_name cambridgesemantics/anzograph-frontend:tag
```

The list below describes each of the parameters:

- **host_http_port** is the port on the local host to use for HTTP access to the Graph Lakehouse console user interface. In the container, the user interface binds to port 8080 for HTTP access. Altair recommends that you specify **80** to map the container's HTTP port to port 80 on the local host. If port 80 is in use, specify an alternate port for **host_http_port**.
- **host_https_port** is the port on the local host to use for HTTPS access to the Graph Lakehouse console user interface. In the container, the user interface binds to port 8443 for HTTPS access. Altair recommends that you specify **443** to map the container's HTTPS port to port 443 on the local host. If port 443 is in use, specify an alternate port for **host_https_port**.
- **container_name** is the short name to use to identify the Graph Lakehouse front end container. For example, **anzograph-frontend**.
- **tag** is the tag from the [Graph Lakehouse Frontend Docker Hub](#) site that identifies the version of the front end console to deploy. If you pulled an image in the first step, this tag should match the tag from the pull command. Usually the **latest** tag is specified so the most recent release is deployed.

For example:

```
docker run -d -p 80:8080 -p 443:8443 --name=anzograph-frontend
cambridgesemantics/anzograph-frontend:latest
```

When the prompt returns the container ID, the container is running. For example:

```
7ad7a2c2b60c0b15e75af9f05d41edc665497c58939da561249c9067f04b59fc
```

3. The front end console user interface is now installed and ready to use. To open the console, open a browser and go to the following URL:

```
https://IP_address
```

Where **IP_address** is the IP address of the host server where Docker for Linux is installed. If you mapped the container's HTTPS port to port 443 on the host, you do not need to specify a port. If you specified a port other than 443, include the port in the URL. For example:

https://10.100.0.1:8888

Note

If you are using Docker for Linux locally on the same server as the Graph Lakehouse leader node and need to know the IP address of the front end console container, you can run the following command:

```
docker inspect container_name | grep '"IPAddress"' | head -n 1
```

For example:

```
docker inspect anzograph-frontend | grep '"IPAddress"' | head -n 1
```

```
"IPAddress": "172.17.0.2"
```

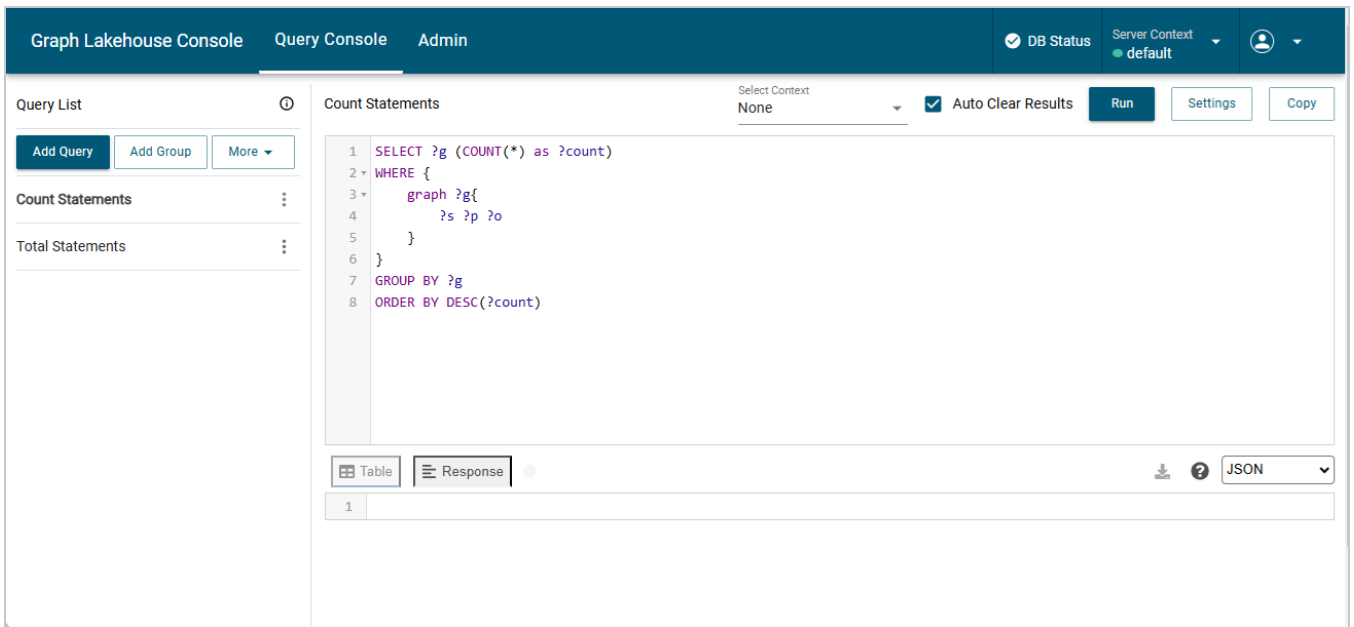
4. The browser displays the Graph Lakehouse console login screen. On the login screen, specify the following credentials and then click **Sign In**.

Username: **admin**

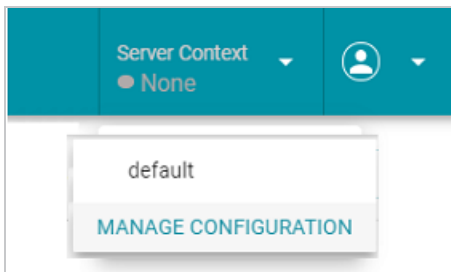
Password: **Passw0rd1**

The End User License Agreement (EULA) is displayed.

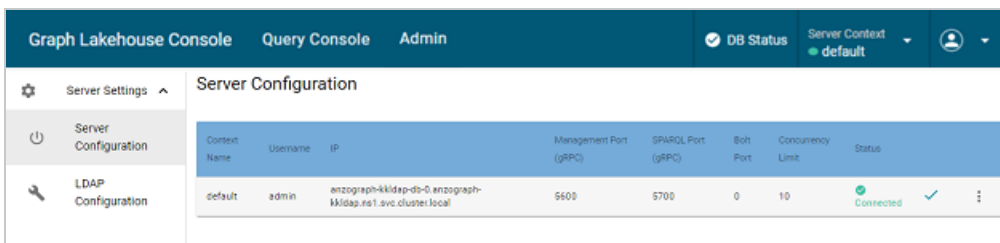
5. Review the EULA and click **Accept** to proceed. The front end console (also referred to as the Query and Admin Console) is displayed.



- The top right of the screen displays the **Server Context**. Because the user interface was deployed separately and is not associated with an Graph Lakehouse deployment, the Server Context is initially "None." Click the **Server Context** drop-down list and select **MANAGE CONFIGURATION**.



The Server Setting or Contexts screen is displayed:



By default, the **Server Configuration** context is used both for authenticating front end console users and providing access to an Graph Lakehouse deployment to execute SPARQL queries. Depending on settings in the Graph Lakehouse settings.conf configuration file, the **LDAP Configuration** option provides the capability to do the following:

1. Authenticate users to access the front end console, while still using the Server Configuration context to access Graph Lakehouse.
2. Configure Graph Lakehouse to both authenticate users and use LDAP group membership to authorize Graph Lakehouse to execute specific SPARQL queries and other statements.

Note

See [Configure Graph Lakehouse for LDAP Authentication](#) and [Create and Manage Roles from the Console](#) for information on setting up Graph Lakehouse to use LDAP directory services for Console and Graph Lakehouse authentication and authorization.

Updating the Server Context

To configure or update settings for the Graph Lakehouse server configuration context:

1. Select the **Server Configuration** option and then click the menu icon (⋮) to the right of the default context and select **Edit**. The Edit Context screen is displayed:

Context Name *

default

IP *

localhost:5432

Management Port (gRPC) *

5600

SPARQL Port (gRPC) *

5700

Bolt Port *

0

Concurrency Limit

10

Username *

admin

Password *

Test Connection

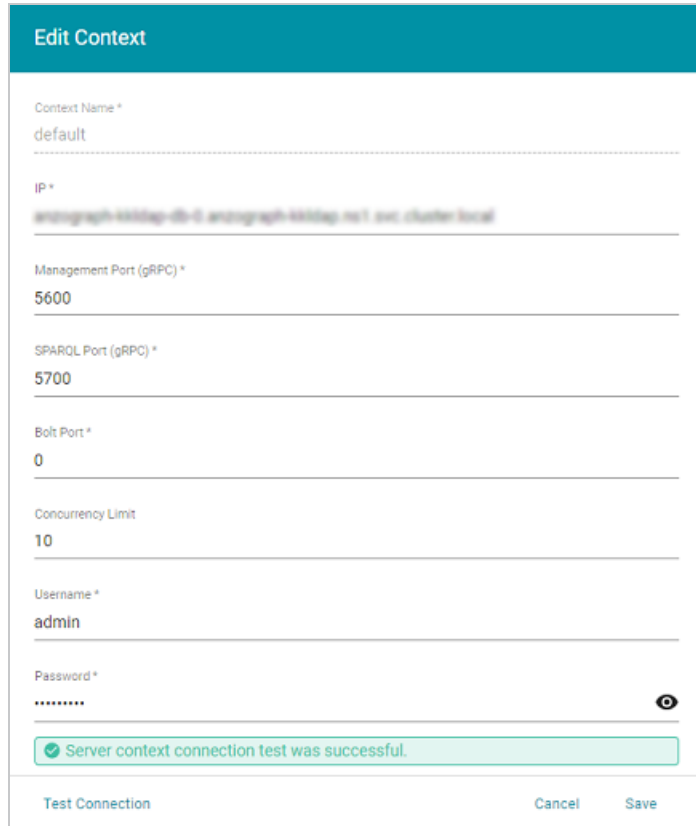
Cancel

Save

- On the Edit Context screen, configure the connection to your Graph Lakehouse site deployment by supplying the values for the following required fields:
 - IP:** Specify the IP address for the Graph Lakehouse leader server. Even if this Docker container is installed on the Graph Lakehouse leader node, you must enter the routable IP address for the server.
 - Management Port (gRPC):** This port is the system management communications port. It is set to **5600** by default and is rarely changed. Accept the default value of **5600** unless you have changed the Graph Lakehouse **sysmgr_port** setting in `<install_path>/azg/config/settings.conf`.
 - SPARQL Port (gRPC):** This port is the secure, encrypted, gRPC-based protocol port used to send SPARQL queries and receiving results. It is set to **5700** by default and is rarely changed. Accept the default value of **5700** unless you have changed the **anzo_protocol_port** setting in `<install_path>/azg/config/settings.conf`.
 - Bolt Port:** This field is for future use. Accept the default value of **0**.

- **Concurrency Limit:** This parameter specifies the maximum number of concurrent threads allowed per database context for the gRPC connection and queries executed from the Graph Lakehouse front end console user interface. The default value is **10**.
- **Username:** This is the gRPC admin user name that you created for the connection.
- **Password:** This is the password that you created for the gRPC user.

1. When you have supplied all of the connection details, click **Test Connection** at the bottom of the screen to ensure that the connection is valid. For example:



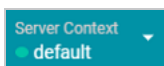
The screenshot shows the 'Edit Context' form with the following fields and values:

- Context Name *: default
- IP *: anisograph-4434ap-05-0 anisograph-4434ap-ns1.ec2.cluster.local
- Management Port (gRPC) *: 5600
- SPARQL Port (gRPC) *: 5700
- Bolt Port *: 0
- Concurrency Limit: 10
- Username *: admin
- Password *: [masked]

A green message bar at the bottom states: "Server context connection test was successful." Below the message bar are three buttons: "Test Connection", "Cancel", and "Save".

If the test fails, adjust the values as needed and test the connection again.

3. Click **Save** to save the connection. At the top of the screen, click the **Server Context** drop-down list again and select **default**. The context is now green to show that the console user interface is now connected:



You can now use the front end console to query and manage your Graph Lakehouse deployment. For more information on using the console, see [Use the Query & Admin Console](#).

Authentication and Access Control

The topics in this section provide details about how Graph Lakehouse authentication and access control works, instructions for setting it up, and configuring or changing roles and privileges to access data.

In this section:

- [Access Control Basics and Terminology](#) 1032
- [Configure Graph Lakehouse for LDAP Authentication](#) 1046
- [Create and Manage Roles from the Console](#) 1050
- [Monitor Access Control Activity](#) 1061

Access Control Basics and Terminology

Graph Lakehouse supports two basic modes of authentication and access control for users submitting requests to access data.

1. The first mode is one in which both user authentication and authorization of privileges are performed entirely by Graph Lakehouse.
2. In the second mode, a trusted external LDAP or directory service system provides authentication of users to validate their identity before submitting a request to Graph Lakehouse.

Note

In many access control systems, *privileges* are often referred to as *permissions*.

Regardless of the authentication mode selected, Graph Lakehouse supports Access Control Lists (ACLs) to manage and control which users have privileges to read or write (update) database objects such as graphs and views. The ACLs are "role-based" which means, for example, that someone in a "manager" role may be permitted to read and update certain graphs, but someone in an "admin" role, with additional privileges, may be permitted to create and drop graphs. When remote authentication is used, an LDAP or directory server returns the list of groups/roles that a validated user is a member of and Graph Lakehouse checks if the user is authorized to perform the requested operation.

When you configure Graph Lakehouse to perform its own authentication of users (without using LDAP or another external directory service), Graph Lakehouse stores both user credentials as well as the privileges that a user has been given. An administrator, or another authorized user given the same authority, can grant users additional privileges or add users to other roles where they will inherit the privileges associated with those roles. When a user submits a request to perform some operation, Graph Lakehouse first validates the user's credentials and then checks the user's privileges before processing their request.

With the integration of LDAP or another directory service, users are authenticated remotely. Before a user's request to access data is sent to Graph Lakehouse, a user's identity is first validated based on the credentials they present for the directory server. If the user's credentials are validated, the service passes to Graph Lakehouse the list of groups/roles that the user is a member of. Graph Lakehouse uses that information to check the user's privileges against the defined ACLs. Graph Lakehouse then processes the user's requested operation if the user's privileges allow it.

Note

In LDAP and other external directory services, roles are commonly referred to as groups. When LDAP or another directory service is used to authenticate users, administrators of those systems are responsible to make sure the group (role) names that are maintained in the directory service match and are kept in sync with the role names defined in Graph Lakehouse. If a group/role name specified by a directory service does not yet exist as a role defined in Graph Lakehouse, an authenticated user is granted access only to the database objects authorized for whatever other roles the user is a member of. The definition of Graph Lakehouse roles can be specified in the **db.ini** file used to configure roles and privileges when Graph Lakehouse is restarted. Roles can also be modified later by a user assigned SUPERUSER role privileges.

ACL Configuration Settings

Two configuration settings, **enable_acl** and **enable_external_auth**, in the `<install_path>/config/settings.conf` file, control the methods of authentication and authorization are used. The **enable_acl** setting controls whether any type of access control, local or remote, is enabled. If access control is first enabled, the **enable_external_auth** setting specifies whether users are authenticated by an external LDAP or directory service or whether Graph Lakehouse is used to authenticate users.

Graph Lakehouse Roles

Access to Graph Lakehouse database objects (graphs, views, and queries) and the ability to perform other operations is controlled by defining "user" roles or "group" roles to which users can be assigned. Each role specifies a set of privileges that will be granted to members assigned to the

role. Role and privilege information is saved in a system graph, which can be queried just like any other graph data by users with sufficient privileges.

A role can be either a "user" role (when using local authentication), a "group" role, or both. Roles can own database objects or entities and members of those roles can assign privileges on those same objects to other roles. Additionally, it is possible to grant membership of a role to a group, which allows all members of that role all the privileges assigned to that group.

After a user has been authenticated and is logged in, any subsequent queries and other operations run in a current session are attributed to that user (which is the user role when using local Graph Lakehouse authentication). For example, if the current logged-in user runs the CREATE GRAPH command, the newly created graph will have the current user designated as the owner of the graph. Similarly, if a SELECT query is run, the privileges to perform the query are verified against the privileges granted to the current user through their membership in specific group roles.

Note

When configured to have Graph Lakehouse provide user authentication, only user roles that have the `LOGIN` attribute enabled can connect to Graph Lakehouse.

Role Attributes

Role attributes define privileges or permissions provided to members of a specific role. You assign attributes to a role using the CREATE ROLE command, or update later with the ALTER ROLE command. You can also delete roles with the DROP ROLE command.

Attributes you can assign to a role are the following:

LOGIN | NOLOGIN

Only roles that have the LOGIN attribute set can be used to connect to Graph Lakehouse (when using Graph Lakehouse "local" authentication). The default role setting is NOLOGIN.

INHERIT | NOINHERIT

If a role is created with this attribute, and when it is added to another group role, then this role will inherit all the privileges of the group role. The default role setting is INHERIT.

SUPERUSER | NOSUPERUSER

The SUPERUSER attribute designates a role with superuser privileges. A database superuser bypasses all available privileges and checks to access Graph Lakehouse data. So, it is recommended that the SUPERUSER attribute only be used very rarely, and that most database operations be done within a role that is not assigned superuser privileges. The default setting for roles is NOSUPERUSER.

PASSWORD = “*password*” | NOPASSWORD

The PASSWORD | NOPASSWORD attribute specifies whether a password is required for user login when Graph Lakehouse is used to provide authentication. If a PASSWORD is supplied at the time of role creation (or added later), then login must supply this same password to connect to the database for that role.

CREATEROLE | NOCREATEROLE

This attribute allows a role to create other roles. Any user logged in as a member of this role with the CREATEROLE attribute set can create, alter, or drop roles as well as grant or revoke membership of roles. A SUPERUSER privilege is required to change the membership of a superuser status.

SYSCATALOG | NOSYSCATALOG

This attribute determines if a role has the ability to SELECT from system graphs or views. The default is NOSYSCATALOG. Only users with SUPERUSER privilege can create a role having the SYSCATALOG privilege.

CREATEGRAPH | NOCREATEGRAPH

This attribute allows a role to create graphs. The default role setting is NOCREATEGRAPH.

CREATEVIEW | NOCREATEVIEW

This attribute allows a role to create views. The default role setting is NOCREATEVIEW.

CREATEQUERY | NOCREATEQUERY

This attribute allows a role to create queries. The default role setting is NOCREATEQUERY.

ROWLIMIT = <num_rows> | NOROWLIMIT

This attribute specifies a positive integer value that determines the maximum number of output rows this role is allowed to retrieve from a SELECT query. The default role setting is NOROWLIMIT.

PRIORITY = <priority_value> | NOPRIORITY

The priority value of a role determines its priority for Graph Lakehouse job scheduling. The default priority value is 50. The range of values allowed is between 1 to 100, with 100 being the highest priority.

Creating, Altering, or Dropping Roles

Two commands are available to create or delete (drop) a role. The syntax for these commands is the following:

```
CREATE ROLE <rolename> [privilege attributes list]
```

When you create a role, Graph Lakehouse inserts the following triples into the **<sbxroles>** system graph based on the attributes specified with the CREATE ROLE command:

```
<role>
  <attrs> attributes;           # combined list of attributes
  <password> "password"^^xsd:string; # optional triple added if PASSWORD set
  <rowlimit> NNN;               # optional triple added if ROWLIMIT set
  <priority> NNN;               # optional triple added if PRIORITY set
  <grants> "member1,member2,..."; # list of all the members to which privilege is
granted
.
```

Tip

For more information on system graphs used to store Graph Lakehouse roles and privileges, see [Access Control System Graphs](#).

The syntax of the command to delete or drop a role is the following:

```
DROP ROLE <rolename>
```

In both the CREATE ROLE and DROP ROLE commands, *<rolename>* is a URI, which specifies a user role (when using only local authentication), or a group role that define privileges granted to group members for specific operations.

Once a role is created, its attributes can be altered with the ALTER ROLE command. The syntax for this command is the following:

```
ALTER ROLE <rolename> [LOGIN | NOLOGIN | INHERIT | NOINHERIT |  
                        SUPERUSER | NOSUPERUSER | CREATEGRAPH | NOCREATEGRAPH |  
                        CREATEVIEW | NOCREATEVIEW | CREATEQUERY | NOCREATEQUERY |  
                        CREATEROLE | NOCREATEROLE | PASSWORD "password" | NOPASSWORD |  
                        ...]
```

The ALTER ROLE command can specify multiple attributes in the same statement. For example:

```
ALTER ROLE <manager> LOGIN NOINHERIT CREATEGRAPH CREATEROLE
```

After roles are defined, you can use the GRANT command to add role membership to other roles. Similarly, you can use the REVOKE command to remove a role's membership from another role. In addition, a couple of additional commands, SET ROLE and RESET ROLE, allow you to clear and reset roles to their original definitions.

Note

The following section describes inheritance of roles when one role is added as a member of another role, and use of the GRANT, REVOKE, SET ROLE, and RESET ROLE commands to change role membership and associated privileges.

Role Membership and Inheritance

There are few differences between "user" roles and "group" roles. If you assign (or grant) a user role to another role, that group simply becomes a group role. To grant privileges of a group to another role, you can run the following command:

```
GRANT <group_rolename> TO <rolename>
```

To remove privileges of a group from a role, you can run the following command:

```
REVOKE <group_role> FROM <rolename>
```

For example:

```
CREATE ROLE <msmith> LOGIN INHERIT;;
CREATE ROLE <pjones> LOGIN INHERIT;;
CREATE ROLE <engineers> NOLOGIN INHERIT;;
CREATE ROLE <managers> NOLOGIN INHERIT;;
GRANT <managers> TO <msmith> ;;           # privileges of <managers> added to <msmith>
GRANT <engineers> TO <pjones> ;;          # privileges of <engineers> added to <pjones>
GRANT <engineers> TO <managers> ;;        # <engineers> privileges added to <managers>
GRANT <dba> TO <msmith> ;;                # <dba> privileges added to <msmith>
```

After the statements above are executed, the <msmith> role will have all the privileges it was originally assigned, but will also inherit all the privileges defined for the <managers> role. In addition, the <managers> role will inherit all the privileges of the <engineers> role with the GRANT command. Any role can be granted multiple other roles (that is, it may be added to more than one group). The structure of roles can be thought of as a hierarchical inheritance structure, with a single user having the combined privileges of all the groups they are a member of.

To remove membership in a role, you can use the REVOKE command. For example:

```
REVOKE <dba> FROM <msmith>
```

If a user is logged into Graph Lakehouse as <msmith>, you can use the SET ROLE command to remove all of <msmith>'s assigned privileges, so the <msmith> role will just be assigned all the privileges from the user or group role it is set to. For example:

```
SET ROLE <msmith> TO <engineers>
```

To reset a role to its original definition, you can execute the following commands:

```
SET ROLE <smith> TO NONE;;
RESET ROLE <smith>
```

Assigning Ownership of Database Objects

By default, Graph Lakehouse assigns original ownership of database objects such as graphs, views, and queries to the currently logged-in user or user role that created an object. You can change the ownership of an object with the ALTER <object> OWNER command. For example:

```
ALTER GRAPH <tpch> OWNER TO <msmith>
```

To remove a role and assign ownership of all the database objects currently owned by that role to a new role, you can use the REASSIGN OWNED BY command. For example:

```
REASSIGN OWNED BY <obsolete-role> TO <new-successor-role>;  
DROP OWNED BY <obsolete-role> ;;  
DROP ROLE <obsolete-role>
```

Database Object Permissions

The roles that a user is a member of determine the Graph Lakehouse data privileges that a user is granted (or can be revoked) after they log in and are authenticated. When an Graph Lakehouse object is created, for example, a graph, query, or view, the creator of that object is designated as the owner of that object. To allow other roles to access the same object, the owner must grant specific privileges on that object to other roles.

The different privileges that can be granted or revoked for database objects are the following:

SELECT

READ privilege for a named GRAPH, VIEW, or QUERY.

UPDATE

Permission for SELECT, INSERT, DELETE, COPY, MOVE, ADD, or CLEAR operations on a named GRAPH.

DROP

Permission to drop a named GRAPH, VIEW, or QUERY.

Granting and Revoking Object Permissions

Unless ownership has been reassigned, the <owner> of a database object has full privileges to perform operations on that object. Owners can use the GRANT and REVOKE commands to grant or revoke privileges to other roles to perform operations on those same database objects they own. For example, to grant UPDATE privileges on an existing <tpch> graph to the <msmith> role, the owner or other authorized users could run the following command:

```
GRANT UPDATE on <tpch> TO <msmith>
```

A special PUBLIC keyword is available to represent all roles, so you could execute the following command to grant privileges to all roles on a database object:

```
GRANT UPDATE on <tpch> TO PUBLIC
```

Note

PUBLIC is a special role, which encompasses all roles currently created, as well as any future roles that may be defined in Graph Lakehouse. If PUBLIC has been granted a privilege, then that privilege is available for all current roles, and the same privilege will be extended to future roles that have not been created yet. If PUBLIC has been granted, then revoking that privilege from roles will not have any effect. The PUBLIC role cannot be created, dropped, or altered.

Another special keyword, ALL, is available to specify granting or revoking "all privileges". For example, to grant all privileges on the <tpch> graph to the <msmith> user, you could run the following command:

```
GRANT ALL on <tpch> TO <msmith>
```

To revoke privileges from a database object, you can use the REVOKE command. For example, to revoke access to <tpch> from everyone (except for the OWNER and SUPERUSER roles), you could run the following command:

```
REVOKE ALL on <tpch> FROM PUBLIC
```

As with the GRANT command, you can also specify privileges with the REVOKE command. For example, to revoke only the UPDATE privilege from the <msmith> role, you could run the following command:

```
REVOKE UPDATE on <tpch> FROM <msmith>
```

Access Control Initialization and Updates

When Graph Lakehouse is first started or when you reinitialize Graph Lakehouse, the **db.ini** initialization file, located in the **<installdir>/config** directory (by default), is executed. The db.ini file contains various DDL statements, for example, CREATE ROLE, DROP ROLE, ALTER ROLE, and

GRANT commands. The db.ini file is treated as "trusted", so statements contained within the file can create any number of roles with any allowed attributes. However, the db.ini file cannot contain any DML statements such as SELECT, CONSTRUCT, ASK, DESCRIBE, etc.

When Graph Lakehouse first starts up, it creates two system roles, @system and @sysadmin. Both are considered superusers, however, by default, you cannot log into Graph Lakehouse with these system roles. In the case of Graph Lakehouse local authentication, you should create other roles that you can login with, by creating those new login roles in the db.ini file. (At least one of the roles should typically be given SUPERUSER privileges.) New user roles may be altered, given passwords, or dropped during the bootstrap initialization process using statements included in the db.ini file.

Important

You need to create the db.ini file before updating **enable_acl** and **enable_external_auth** settings in the **settings.conf** file to enable either local or remote authentication and access control. To reinitialize Graph Lakehouse with new ACL configuration settings, you can run the following command:

```
/install_path/bin/azgctl -start -init
```

Alternatively, you could also run the following command:

```
/install_path/bin/azgctl -start -init data
```

For more information on starting and stopping, restarting and initializing Graph Lakehouse, see [Start and Stop Graph Lakehouse](#).

If Graph Lakehouse fails to successfully execute the db.ini file due to errors, Graph Lakehouse will not start up. To prevent this situation from occurring, an administrator should test and make sure the db.ini file can be executed without errors.

Here is a sample db.ini file that shows some of the statements the file might include:

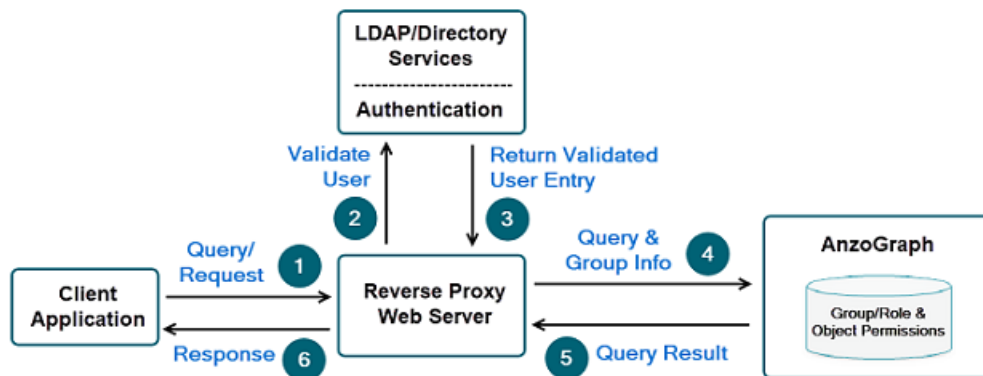
```
# group roles
CREATE OR REPLACE ROLE <superadmin> SUPERUSER LOGIN PASSWORD = 'superadmin' ;;
CREATE OR REPLACE ROLE <manager> NOLOGIN NOINHERIT CREATEROLE CREATEGRAPH CREATEVIEW
CREATEQUERY ;;
CREATE OR REPLACE ROLE <dev> NOLOGIN NOINHERIT CREATEGRAPH CREATEVIEW CREATEQUERY ;;
```

```
# login users (local authentication mode only)
CREATE OR REPLACE ROLE <julio> LOGIN ;;
CREATE OR REPLACE ROLE <john> LOGIN ;;
CREATE OR REPLACE ROLE <mary> LOGIN ;;
# grant privileges to local user roles
GRANT <dev> TO <julio> <john> <mary>;
GRANT <manager> TO <julio>
```

LDAP/Directory Services Integration

There are various ways to configure integration of LDAP or another directory service system with Graph Lakehouse, depending on the security policy needs and the current IT infrastructure in place within an organization. First, customers can choose from any number of open source or commercially available authenticator applications, for example, from companies such as Apple, Facebook, and Google, or other authentication solutions published on web sites such as GitHub. These applications can provide an interface between client applications and directory servers and can also be configured to provide authentication and delivery of user profile and group/role information to Graph Lakehouse.

For additional security, many customers may also choose to use a reverse proxy web server to isolate their authentication systems and Graph Lakehouse from direct connection to their client applications and exposure to other potential threats. The following diagram shows a configuration in which a reverse proxy web server processes incoming Graph Lakehouse queries and other requests from a client application. A client application directs the user's credentials and the requested operation through the reverse proxy server to an LDAP/Directory Services system for authentication. If the user is validated, the reverse proxy then forwards the user profile, the names of the groups in which the user is a member, and their original operation request, to Graph Lakehouse.



The basic operation common to all these configurations is that a client application first constructs and outputs a formatted request that includes a user's credentials along with the Graph Lakehouse operation the user wants to perform. For example:

```
azgi -u username:password -c "create graph <abc>"
```

Note

Many authenticator programs allow client applications to omit password entries from the request, and then prompt users later to enter a password. In addition, authenticator programs will often cache an authentication token once a user's credentials have been validated so that a user's credentials do not need to be re-validated for every request unless the token has expired.

After validating the user, the LDAP or directory service systems looks up whatever group/role membership a user has been assigned. That information is then forwarded to Graph Lakehouse as a base 64 encoded JSON string in the `User-Entry` header of the HTTP request. For example, a JSON string might look like the following:

```
{"name":"jsmith","groups":[{"name":"manager"}]}
```

The JSON blob returned for validated users can also specify multiple assigned groups/roles. It may also include nested groups/roles if your privilege hierarchy is set up to use them. For example:

```
{
  "name": "jsmith",
  "member_of": [{
    "name": "engineering",
```

```
"member_of": [{ "name": "manager" }]
},
{ "name": "support" }]
}
```

Note

LDAP and other directory services use the term *group*, synonymous to *roles* in Graph Lakehouse, to reflect a common set of privileges or privileges granted to any members assigned to the same group.

Once Graph Lakehouse receives the User-Entry JSON string, it compares the specified group names with roles of the same name defined in Graph Lakehouse. Graph Lakehouse then checks the Access Control privileges (ACLs) on database objects to verify the user's privileges to perform the requested operation. If the user has the appropriate privileges, Graph Lakehouse processes the request and returns results back to the client application; otherwise it returns a message indicating the request is not allowed.

Note

The names of all groups specified in the User-Entry JSON string must exactly match role names already defined in Graph Lakehouse for the current user to be given the associated role privileges. Graph Lakehouse roles may be defined at startup with entries in the db.ini file or by issuing role creation commands after Graph Lakehouse has started.

Sample LDAP/Directory Services Configuration

Sample files and instructions for configuring an example integration of Graph Lakehouse with an LDAP server is provided on the Cambridge Semantics GitHub web site:

<https://github.com/cambridgesemantics/csi-anzograph-ldap-demo>

The README.md file on the site describes the series of steps you can perform to install and configure this example setup. The sample configuration of the LDAP/Directory Services integration with Graph Lakehouse includes the following components:

- **NGINX:** Free, open source HTTP web server to which client application operation requests are sent.
- **LDAP authentication proxy:** Example Python program that processes requests to validate user credentials and pass client operation requests to an LDAP server.
- **OpenLDAP:** Open source LDAP implementation that maintains user profile and group/role membership information for Graph Lakehouse privileges.
- **Graph Lakehouse** database engine pre-configured for remote authentication and access control.

After completing the setup steps in the instructions, you can use any HTTP client application to send Graph Lakehouse operation requests to the reverse proxy server. For demonstration purposes, you can also use the AZGI CLI to submit requests.

In addition to the basic setup, you can install and use other third party tools such as Apache Directory Studio to administer changes to OpenLDAP user profiles and group/role membership. You can also use tools such as Postman to submit queries and other requests to Graph Lakehouse.

Configure Graph Lakehouse for LDAP Authentication

By default, the Graph Lakehouse front end console is configured to use authentication and authorization credentials maintained by Graph Lakehouse itself. You can also configure the Graph Lakehouse front end console to use a specified LDAP or directory service to authenticate users and authorize Graph Lakehouse operations based on user membership in LDAP groups. See [Create and Manage Roles from the Console](#) for information on creating user roles and granting or revoking permissions to access specific Graph Lakehouse database objects (graphs, views, and queries), whether you are using local Graph Lakehouse or LDAP service authentication of users.

Define an LDAP Configuration

To configure the Graph Lakehouse front end console and Graph Lakehouse to use LDAP authentication:

1. Select **LDAP Configuration** from the Server Settings list to display the LDAP Configuration screen.
2. On the LDAP Configuration screen, configure the connection to your Graph Lakehouse deployment by selecting the **Enable LDAP Authentication** checkbox and then choosing among the various radio button options and supplying values for the required fields. Selecting the **Enable LDAP Authentication** checkbox enables front end authentication using the the LDAP configuration.

LDAP Configuration

☒ Enable LDAP Authentication

Host *
mygraphlakehouse-openldap.my1.svc.cluster.local

Port *
389

HTTPS
☒ None ☐ SSL Connection ☐ Start TLS

User Base DN *
dc=example,dc=org

User Filter Prefix *
cn

Groups Search Filter *
(objectClass=groupOfUniqueNames)

Groups Member Filter Prefix *
uniqueMember

☒ Search Subtree ☐ Anonymous Bind

User DN *
cn=admin,dc=example,dc=org

Password *
.....

Confirm Password *
.....

Test Connection Cancel Save

Field entries for the LDAP Configuration are the following:

- **Enable LDAP Authentication** checkbox: Selection that allows you to enable front end authentication using the LDAP configuration.
- **Host**: Host name or IP address of the LDAP directory server.
- **Port**: The port used to connect to the LDAP directory server.
- **HTTPS** radio buttons: Specifies whether the directory server uses an **SSL** (LDAPS) or a **StartTLS** protocol connection.
- **User Base DN**: LDAP distinguished name that contains users than can be authenticated, for example: `dc=example,dc=org`.
- **User Filter Prefix**: Property name that a user name is mapped to, for example: `cn`.
- **Groups Search Filter**: Filter used to search for LDAP group names, for example: `(objectClass=groupOfUniqueNames)`.

- **Groups Member Filter Prefix:** Property name prefix used for searching if user is part of group, for example: `uniqueMember`.
 - **Search Subtree** checkbox: Option to specify whether to search LDAP subtrees.
 - **Anonymous Bind** checkbox: Option to specify whether the Graph Lakehouse front end console connects to the directory server anonymously.
 - **User DN:** Full distinguished name of the account that the Graph Lakehouse front end console will bind against to perform authentication on the directory server, for example: `cn=admin,dc=example,dc=org`.
 - **Password:** Password specified for the User DN.
3. When you have supplied all of the necessary connection details, click **Test Connection** at the bottom of the screen to ensure that the connection with your LDAP directory service can be made.
- If the test fails, adjust the values as needed and test the connection again.
4. Click **Save** to save the connection.

Enable LDAP Authentication for the Console

To use an LDAP configuration to authenticate Console login and authorize Graph Lakehouse operations users are able to perform, you need to update settings in the Graph Lakehouse **settings.conf** configuration file. That is, to enable external LDAP authentication to the console, you need to configure the following options in the Graph Lakehouse **settings.conf** file (located in the `InstallDir/anzograph/config` directory) :

```
enable_acl=true  
enable_external_auth=true
```

Important

After updating the settings in the Graph Lakehouse configuration file, you need to restart Graph Lakehouse for the new settings to take effect. For example:

```
<install_path>/anzograph/bin/azgctl -restart
```


With these new ACL settings, Graph Lakehouse front end console users will be authenticated against an externally- configured LDAP directory service. A user's LDAP group membership information will be passed to Graph Lakehouse along with any submitted SPARQL query request or statement they submit to help in authorizing requests. Where Graph Lakehouse roles are already defined that match the names of LDAP groups a user is a member of, the Graph Lakehouse assigned role permissions will determine a user's authorization or permission to execute any submitted SPARQL request.

Note

See [Access Control Basics and Terminology](#) for more information on Graph Lakehouse ACL operations and additional methods of integrating LDAP directory services with Graph Lakehouse.

You can now use the front end console using LDAP directory service authentication of users. For more information on using the front end console, see [Use the Query & Admin Console](#). Also refer to [Create and Manage Roles from the Console](#) for information on defining or updating roles that control Console user access and permissions.

Create and Manage Roles from the Console

Graph Lakehouse supports two basic modes of authentication and access control for users submitting requests to access or perform operations on data. The first mode is one in which both user authentication and authorization of privileges are performed entirely by Graph Lakehouse. In the second mode, a trusted external LDAP or directory service system provides authentication of users to validate their identity before submitting an operation request to Graph Lakehouse, where a user's Graph Lakehouse role permissions determine the operations a user is able to perform.

The particular way in which Graph Lakehouse operates depends on two switch settings in the `settings.conf` file, **enable_acl** and **enable_external_auth**. (See [Authentication and Access Control](#) for more information on these two Graph Lakehouse configuration settings.)

When using the Query & Admin Console to access and perform Graph Lakehouse operations, the operations that a user can perform are controlled by the specific attributes of a user's role and the permissions granted to a user or role on specific Graph Lakehouse database objects, for example, graphs, views, and queries. A user with special SUPERUSER administration permissions can login into the Console, create roles that will define permissions granted to new users in those roles, and specify permissions to access specific Graph Lakehouse database objects.

The following sections describe the process to configure Graph Lakehouse for authentication and authorization (also referred to as ACL) and provides more information on how to define new roles for users (or those corresponding to LDAP groups) from the Console, and specify attributes or permissions granted to those roles.

- [Configure Graph Lakehouse for User Role Management](#)
- [Create and Configure User Roles](#)
- [Grant Permissions to Database Objects](#)
- [Add Roles Mapped to LDAP Groups](#)
- [Enable External LDAP Authentication](#)

Configure Graph Lakehouse for User Role Management

To use the Query & Admin Console to create and manage roles for Graph Lakehouse authentication and authorization there are a few basic configuration steps you first need to perform:

1. Create a **db.ini** file in the **InstallDir/anzograph/config** directory in which you define a local user role with LOGIN and SUPERUSER attributes. For example:

```
CREATE OR REPLACE ROLE <superadmin> SUPERUSER LOGIN PASSWORD = 'superadmin' ;;
```

Note

Graph Lakehouse includes a sample file, **db.ini-example**, that you can rename and use as a starting point for your own custom access control initialization. The sample file defines a default SUPERUSER LOGIN user role named **<superadmin>** that provides initial administrator access to the Console when Graph Lakehouse access control is first enabled. See [Access Control Initialization and Updates](#) for more information on the db.ini file and Graph Lakehouse access control initialization and updates.

2. In the **InstallDir/jetty/frontend** directory, include the following line in the **anzograph-frontend.properties** file to provide administrator access to the Console that matches the SUPERUSER user role defined in the Graph Lakehouse db.ini file:

```
database.super.user= superadmin
```

Note

Updating the **database.super.user** setting and restarting the Jetty application, as described in these steps, is not required if you keep the **<superadmin>** user role name as defined, by default, in the **db.ini-example** sample file. The Eclipse Jetty application is a Java web server and Java Servlet container that provides the web user interface for the Graph Lakehouse Query & Admin Console.

3. Restart the Jetty web server.

```
/usr/sbin/su-exec jetty jetty.sh restart
```

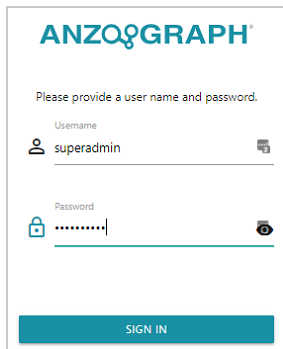
4. Update the **settings.conf** file located in the **InstallDir/anzograph/config** directory to include the following two settings:

```
enable_acl=true  
enable_external_auth=false
```

5. Run the following commands to stop and reinitialize Graph Lakehouse with the new **settings.conf** configuration file settings and **db.ini** entries:

```
/InstallDir/anzograph/bin/azgctl -stop  
/InstallDir/anzograph/bin/azgctl -start -init
```

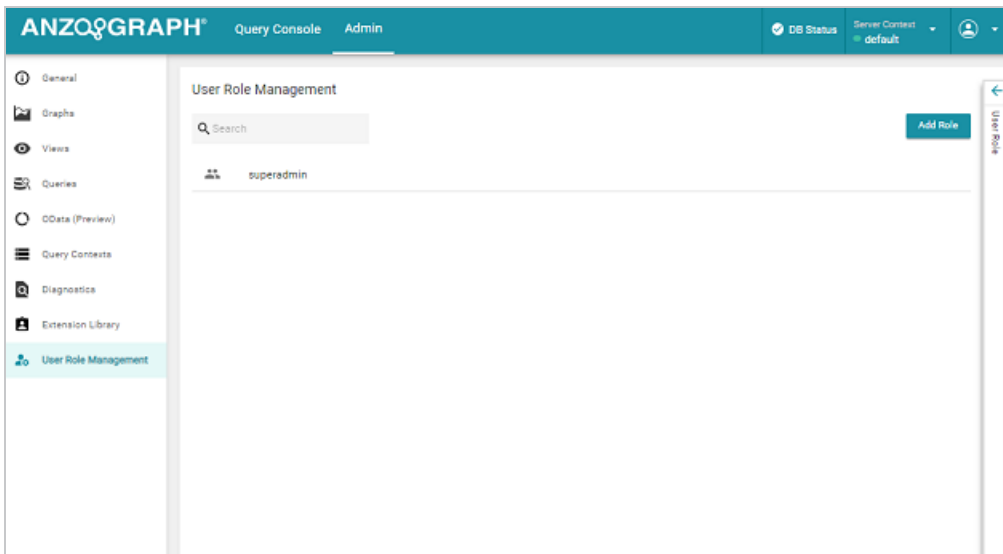
6. Log in to the Query & Admin Console using the new superadmin user role credentials defined in the **db.ini** file.



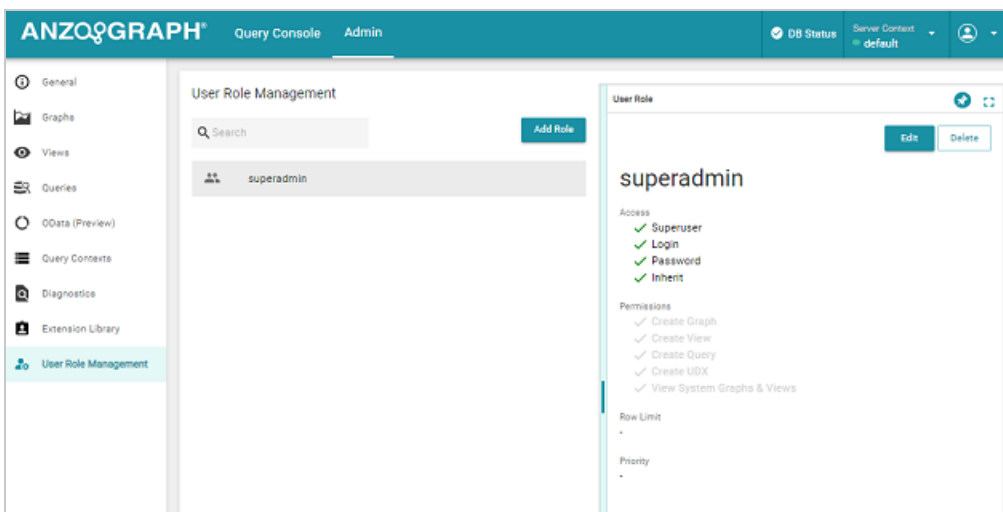
After successful user login, the console displays the main Admin tab web page.

Create and Configure User Roles

When you log in to the Query & Admin Console with the Graph Lakehouse **superadmin** user role credentials, you have full access to Graph Lakehouse operations and data, including permission and access to all options available in the Console, and the User Role Management option in the Admin navigation panel. When you first select the **User Role Management** option, the Console displays a single role, that of the superadmin user role you defined in the **db.ini** file.



To view the access and permissions of the superadmin role or any other defined role, you can simply click on the role's name to display the information in a panel display on the right side of the Console.



Access and permission settings correspond to ACL attributes and permissions that may be specified for Graph Lakehouse roles as described in [Role Attributes](#) and [Database Object Permissions](#).

Adding a New Role

To create a new role:

1. Click the **Add Role** button.

The Console displays a popup dialog in which you can choose a name for the new role and specify other properties of the role.

Add Role

Role Name

jmith

Whitespaces are not allowed.

☐

Do you want to make this role a superuser?

☒

Do you want this role to inherit from other roles?

☐

Do you want to allow this role to login?

Password

Confirm Password

Select permissions for this role.

☒ Create Graph

☒ Create View

☒ Create Query

☐ Create UDX

☒ View System Graphs & Views

Select roles who will inherit the permissions of this role.

Select...

Query Row Limit

Priority

Only positive integers are allowed.

Only integers from 1 to 100 are allowed.

Cancel

Add

Note

New role names may not start with either an @ or \$ character. Role names may also not contain spaces or any of the following special characters:

- < > ? / \ [] { } + = () * & ^ % ~ ` " '

More generally, role names may not contain any ASCII code control characters ranging from 0 through 32 with ASCII code 32 being the space character.

Options in the Add Role dialog allow you to specify the following:

- Make this role a superuser — if enabled, allows user role to access the Admin tab of the Console, versus just the Query Console tab.
- Inherit permissions from other roles — if enabled, this role may be selected to inherit permissions as a member of other roles.

- Allow this role to login — when enabled and using local Graph Lakehouse authentication (`enable_external_auth=false`), allows this user role to login to Graph Lakehouse.
- Select permissions for this role — specifies operations that a user role is allowed to perform in Graph Lakehouse.
- Select other roles to inherit permissions from this role — allows you to specify other existing user roles that will inherit permissions from this role.
- Query Row Limit and Priority — Specifies limits to rows that a user role's query may return; also lets you specify the execution priority of this user role's queries.

Note

See [Role Attributes](#) for more information on attributes or permissions that can be assigned to specific roles.

2. When you've finished selecting options for the new user role, click **Add**.

The new user role now appears in the Admin Console's list of created roles.

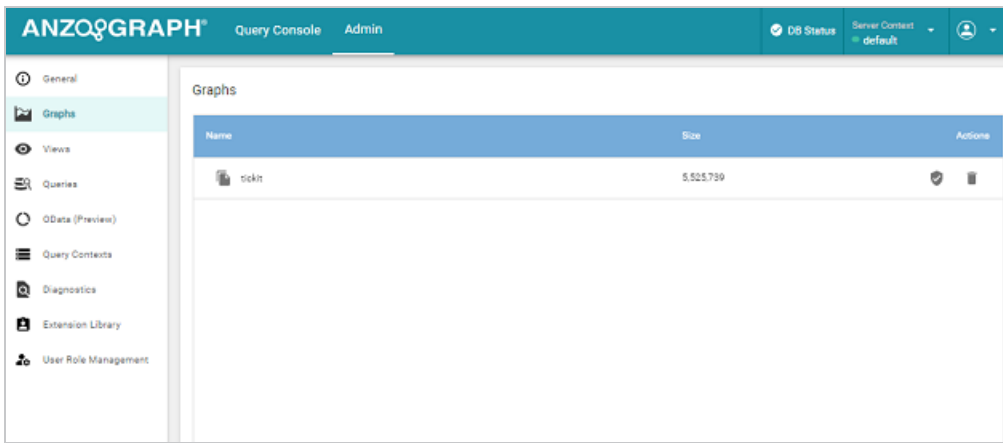
Grant Permissions to Database Objects

When an Graph Lakehouse database object is created, for example, a graph, query, or view, the creator of that object is designated as the owner of that object, by default. To allow other roles to access the same object, or to change ownership, the owner may grant specific privileges on that object to other roles.

Besides being able to change ownership, SELECT, UPDATE, and DROP permissions privileges on database objects can be granted or revoked from specific roles.

To assign or change object permissions:

1. From the Admin tab, click on a database object (Graphs, Views, or Queries) in the left-side navigation panel. For example, selecting the Graphs option displays a list of graphs currently loaded into Graph Lakehouse.



- On the far-right side of a listed object (in this example, the tickit graph), click the ♥ button. The Console displays a popup dialog in which the current owner and permissions of other existing roles are displayed.

Manage Permissions (tickit)

Select an owner.

superadmin

Owner of this graph will have all privileges.

Grant or revoke permissions.

Search

	AZG-manager	☐ Select	☐ Update	☐ Drop
	jsmith	☐ Select	☐ Update	☐ Drop
	mjones	☐ Select	☐ Update	☐ Drop
	superadmin	☒ Select	☒ Update	☒ Drop

Cancel
Save

Besides being able to change ownership, privileges that can be granted or revoked on database objects for specific roles are the following:

- SELECT - grant or revoke read privilege on a named graph, view, or query.
- UPDATE - grant or revoke permission for SELECT, INSERT, DELETE, COPY, MOVE,

ADD, or CLEAR operations on a named graph.

- DROP - grant or revoke permission to drop a named GRAPH, VIEW, or QUERY.
3. When you've finished selecting permissions granted or revoked for specific roles, click **Save**.

The Console confirms that permissions for the object have been updated.

Add Roles Mapped to LDAP Groups

In addition to defining roles for use when local Graph Lakehouse authentication is enabled, you can also define roles that are mapped to LDAP groups for use when external LDAP authentication is used. That way, when users log into the Console using LDAP authentication, the data they can access and the operations they can perform are controlled by the permissions defined in roles corresponding to their LDAP group membership.

To define Console roles based on LDAP groups, you need to first specify and enable the LDAP configuration you want to define Graph Lakehouse roles for. Once the LDAP configuration is enabled, you can define roles based on LDAP directory groups the same way as you defined roles using local Graph Lakehouse authentication.

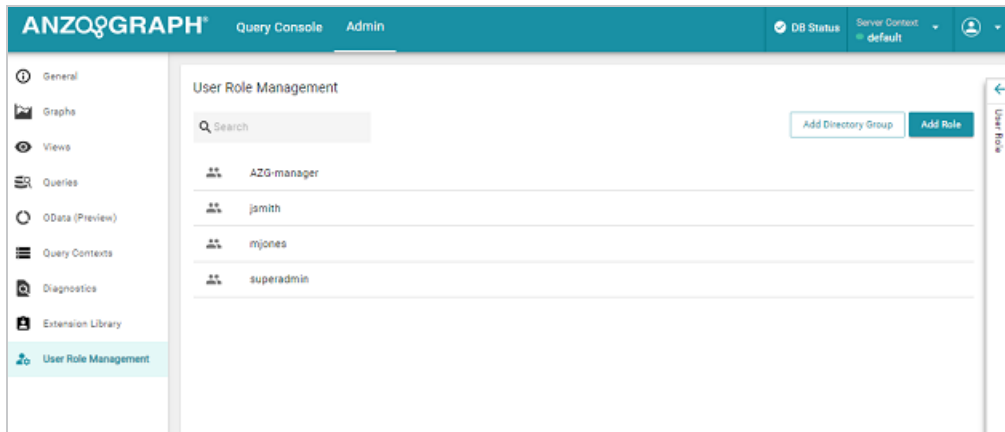
Note

When using external LDAP authentication, you should define at least one LDAP group role that has SUPERUSER privileges configured, so that member users of that group have access to the Admin tab of the Console.

To add an LDAP group role:

1. Define the LDAP Configuration that you want to define Graph Lakehouse roles for. For specific instructions on doing that, see [Configure Graph Lakehouse for LDAP Authentication](#).
2. From the Console, select the **Settings** menu option, or select the **LDAP Configuration** option from the **Server Context/Server Setting** page and make sure the **Enable LDAP Configuration** checkbox option is selected.

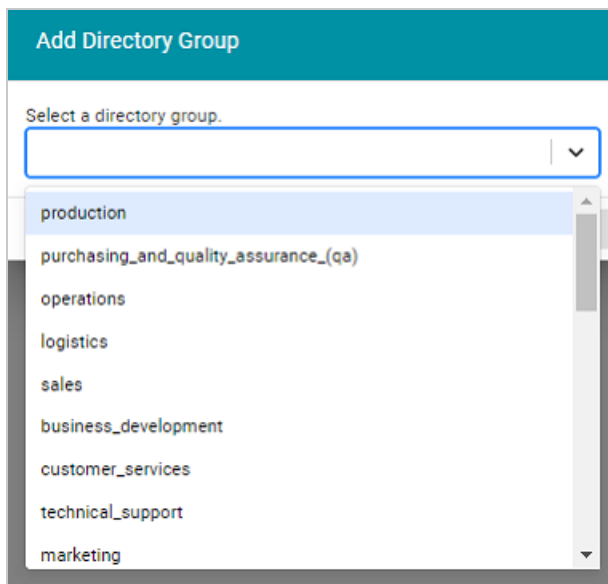
When you return to the User Role Management display on the Admin tab page, the Console now shows an **Add Directory Group** button on the far right, next to the Add Role button.



3. Click the **Add Directory Group** button.

The Console now retrieves a list of groups defined in the LDAP directory.

4. Click the ▼ down arrow icon to display a drop-down list of LDAP groups. Note that the Console replaces any spaces in group names with an underscore character.



Note

Graph Lakehouse role names may not start with either an @ or \$ character. Also, role names may not contain spaces or any of the following special characters:

- < > ? / \ [] { } + = () * & ^ % ~ ` ' " ,

More generally, role names may not contain any ASCII code control characters ranging from 0 through 32 with ASCII code 32 being the space character.

5. Select an LDAP group from the list that you want to define an Graph Lakehouse role for.

The Console now displays a popup dialog, the same as for local roles, to specify group attributes and permissions for the new Graph Lakehouse role.

Add Directory Group

Select a directory group.

administrator

☒ Do you want to make this role a superuser?

☐ Do you want this role to inherit from other roles?

☐ Do you want to allow this role to login?

Select permissions for this role.

☒ Create Graph ☒ Create View ☒ Create Query

☒ Create UDX ☒ View System Graphs & Views

Select roles who will inherit the permissions of this role.

Select...

Query Row Limit Only positive integers are allowed.

Priority Only integers from 1 to 100 are allowed.

Cancel Add

6. When you've finished selecting options for the new role based on an existing LDAP group, click **Add**.

The new LDAP group role now appears in the Admin Console's list of created roles. Besides the role's basic attributes and permissions, you can also grant or revoke specific object permissions for the new LDAP group role.

Enable External LDAP Authentication

To change the Graph Lakehouse mode of authentication, from authentication of user credentials stored within Graph Lakehouse itself, to authentication of credentials validated by an external LDAP directory service, you need to update ACL settings specified in the Graph Lakehouse `settings.conf` configuration file.

1. Update the **settings.conf** file located in the `<install_path>/anzograph/config` directory to specify the following two settings:

```
enable_acl=true
enable_external_auth=true
```

2. Next, run the following command to stop and restart Graph Lakehouse to use the new configuration settings in the `settings.conf` file:

```
/InstallDir/anzograph/bin/azgctl -restart
```

Graph Lakehouse authentication requests, including login to the Query & Admin Console, will now be authenticated using the LDAP Configuration you specified.

3. Login to the Query & Admin Console using the user credentials provided by your LDAP directory service administrator.

With LDAP authentication enabled, users will enter their LDAP user login credentials to be authenticated. However, following authentication, each user's permissions to perform Graph Lakehouse operations will be defined by their membership in LDAP groups for which Graph Lakehouse roles are defined.

Monitor Access Control Activity

All Graph Lakehouse data access and access control modifications are logged in a system table named **sth_acl**. System administrators can monitor the sth_acl system table for various types of access control entries and activities. Those events include:

- "Init file execution"
- "Authorization Success"
- "Authorization Failure"
- "Create Role"
- "Drop Role"
- "Alter Role"
- "Alter Graph" (changes of ownership)
- "Alter Owned By"
- "Grant Membership"
- "Grant Privilege"
- "Revoke Membership"
- "Revoke Privilege"

In addition to monitoring access, an administrator can diagnose failures by examining the entries in the "sth_acl" table. You can query the Graph Lakehouse system table using regular SPARQL queries just like that of any other database source. For example:

```
azgi -c "select * where {table 'sth_acl'}"
```

The following provides a sample query of sth_acl table entries following execution of a GRANT statement:

xrowid	query	time	user	action_type	detail
1219813	12453	2020-11-20...	test	Grant Privilege	Granted privileges 1

```

on <testGraph> to <jsmith>
1220751 | 0 | 2020-11-20... | | Authorization Success | test
1219294 | 0 | 2020-11-20... | | Authorization Success | test
1212465 | 12370 | 2020-11-20... | test | Grant Membership | Granted membership of
<engineers> to <jsmith>

```

Access Control System Graphs

All Graph Lakehouse role and object attributes and privileges are stored in one of two system graphs, **<sbxroles>** or **<sbxaclobj>**.

Regardless of whether users are authenticated locally or by a remote directory service, the privileges granted to specific groups or roles is stored locally within the system graph named **<sbxroles>**.

Note

A second system graph named **<sbxaclobj>** stores privileges to perform operations such as SELECT, UPDATE, DROP, and EXECUTE on objects such as graphs, views, and queries. When an object is created, its creator is designated as the owner of that object and that information is recorded with the entry of a triple in the **<sbxaclobj>** system graph. To allow other users to access the same object, the owner can grant privileges on that object to other Graph Lakehouse roles.

The **<sbxaclobj>** system graph is updated with new triples whenever an object is either created or dropped or if its privileges are altered. Only system administrators assigned the superuser role or belonging to a group with administration privileges have the ability to modify the **<sbxaclobj>** system graph to update privileges.

The <sbxroles> System Graph

The Graph Lakehouse **<sbxroles>** system graph is updated whenever a role is created, altered, or dropped. The following triples get inserted, updated, or removed from the **<sbxroles>** graph, whenever one of these operations is performed.

```

INSERT DATA {
GRAPH <sbxroles> {
<role1> a <Role>;

```

```

<attrs> attrib_list;                # combined list of attributes
<password> "passwd"^^xsd:string;    # optional triple for local authentication
<rowlimit> NNN;                     # optional triple
<priority> NNN;                     # optional triple
<grants> "member1,member2";         # list of all the roles granted privileges
.
}
}

```

The <sbxaclobj> System Graph

A second system graph, named **<sbxaclobj>**, is updated with new triples whenever an object is created or dropped or if its privileges are altered. The following triples get inserted, updated, or removed from the **<sbxroles>** system graph whenever one of these operations is performed.

```

INSERT DATA {
GRAPH <sbxaclobj> {
    <aclobj_uri>                                # <based on objname,objtype,objcontainer
    <objname> name;
    <objtype> <graph | view | query | udx | subject | predicate | triple> ;
    <privileges> acl;                            # mandatory: serialized json string for Privileges
    <objid> (graphid | viewid | queryid);        # optional: for graph
    <container> "container" ; # optional: for udx; name of the library
# for subject - name of the graph
# for predicate - name of the graph
# for triple - name of the graph
.
}
}

```

Note

Again, only an Graph Lakehouse system administrator, assigned the superuser role or belonging to a group with administration privileges, has the ability to directly modify the **<sbxaclobj>** system graph.

Manage the Server Configuration

This section provides reference information for the Graph Lakehouse server settings and includes instructions for changing the configuration file.

In this section:

System Settings Reference	1065
Change System Settings	1079

System Settings Reference

This topic provides reference information for each of the Graph Lakehouse system configuration settings. The configuration file, `<install_path>/config/settings.conf`, categorizes the settings as either **Basic** or **Advanced**. The advanced-level settings should only be configured by system administrators or users with an advanced level of knowledge about Graph Lakehouse or databases in general. For instructions on changing settings, see [Change System Settings](#).

- [Basic Settings](#)
- [Advanced Settings](#)

Basic Settings

This section describes the settings in the Basic section of settings.conf.

- [anzo_protocol_port](#)
- [auto_restart_directory](#)
- [enable_persistence](#)
- [enable_sparql_protocol](#)
- [enable_ssl_protocol](#)
- [internal_directory](#)
- [license_file](#)
- [load_errors_graph](#)
- [load_normalize_datetime](#)
- [log_directory](#)
- [log_rotate_counter](#)
- [log_size_limit](#)
- [max_memory](#)
- [output_format](#)

- `persistence_directory`
- `service_graph_rowset_limit`
- `sparql_protocol_port`
- `sparql_spec_default_graph`
- `spill_directory`
- `ssl_protocol_port`
- `startup_info`
- `stop_timeout`
- `truncate_clob`
- `use_custom_ssl_files`
- `user_queues`
- `xray_sth_portion`
- `xray_sth_spool_duration`
- `xray_sth_spool_maxgb`

Setting	Default Value (type)	Description
anzo_protocol_port	5700 (int)	The Anzo protocol (gRPC) port for secure communication between Graph Lakehouse and other Altair applications.
auto_restart_directory	Not set (char)	Specifies the base location of the auto_restart directory, which contains the <code>denied_list</code> , <code>warned_list</code> , and <code>unanalyzed_list</code> directories. When not set, the default is <code><install_path>/internal</code> . For more information about the auto-restart feature, see Manage Automatic Database Restart Options .

Setting	Default Value (type)	Description
enable_persistence	true (boolean)	Controls Graph Lakehouse's save data to disk option. By default, persistence is enabled and the data in memory is saved to disk (in the persistence_directory) after every transaction. Each time the database is restarted, the persisted data is automatically loaded back into memory. When persistence is disabled, data must be reloaded back into memory when the database is restarted.
enable_sparql_protocol	true (boolean)	This setting controls whether to enable the standard SPARQL-compliant HTTP endpoint. The sparql_protocol_port setting controls the port to use to access the endpoint. The SPARQL HTTP endpoint is not secured.
enable_ssl_protocol	true (boolean)	This setting controls whether to enable the secure HTTPS SPARQL endpoint. The ssl_protocol_port setting controls the port to use. The SPARQL HTTPS endpoint is encrypted but not authenticated.
internal_directory	Not set (char)	The directory where Graph Lakehouse should save internal database-related files such as generated code, logs, and query plans. When not set, the default is <code><install_path>/internal</code> . For more information, see Relocate Graph Lakehouse Directories .
license_file	license.pem (char)	This setting specifies the name of the license file. Do not change this value unless instructed to do so by Altair Support.
load_errors_graph	load_errors (char)	This setting specifies the name of the graph to load error messages to if LOAD SILENT is specified with a SPARQL LOAD query and errors are encountered during the load.

Setting	Default Value (type)	Description
load_normalize_datetime	Not set (char)	This setting specifies a dateTime value to use in place of any invalid dateTime values that are found when loading files with a SPARQL LOAD query. If Graph Lakehouse returns a "Datum is not a datetime, use setting 'load_normalize_datetime' to patch bad data" error, you can specify the value to substitute for any invalid dateTimes. For example, "0001-01-01T00:00:00Z".
log_directory	Not set (char)	Specifies where to write system management daemon (azgmgrd) log files. These types of logs (azgmgrd.log, azgctl-<user>.log, azgpidsmgr.log, and azgpids.log) are created before the system is initialized and may be written before the <install_path>/internal/log directory exists. Therefore, they are located outside of the Graph Lakehouse file system, /tmp by default. If you change the log_directory value, Altair recommends that you choose another location that is outside the internal Graph Lakehouse directories. When not set, the default location is /tmp.
log_rotate_counter	5 (int)	This setting controls the number of azgmgrd.log files to retain when the logs are rotated and a new one is created. Logs are rotated once a file reaches the size limit specified in log_size_limit . By default, 5 files are kept plus the current one. When this value is 0, the log files are not rotated and a single file will contain all of the azgmgrd logging information.
log_size_limit	1790000 bytes (int)	This setting sets the limit (in bytes) for the size of the azgmgrd.log file. When the limit is reached, the current file is saved and a new one is started.

Setting	Default Value (type)	Description
max_memory	0=System-based (int)	Specifies the amount of memory (in MB) that is available for Graph Lakehouse. The default is system-based; at startup, Graph Lakehouse determines the amount of RAM that is available and sets max_memory. In test environments where Graph Lakehouse may be co-located with other programs, you can set the max_memory value to put a limit on the amount of memory Graph Lakehouse can use. However, Altair recommends that you do not set max_memory unless instructed by Support.
output_format	xml (char)	Specifies the default output format for Graph Lakehouse responses. Valid values are <code>xml</code> , <code>json</code> , or <code>csv</code> .
persistence_directory	Not set (char)	The directory where Graph Lakehouse should save data when enable_persistence is <code>true</code> and data is persisted to disk. When not set, the default is <code><install_path>/persistence</code> . For more information, see Relocate Graph Lakehouse Directories .
service_graph_rowset_limit	1000 (int)	This setting places a limit on the number of rows that can be returned per request against the SPARQL endpoint.
sparql_protocol_port	7070 (int)	This setting specifies the port to use to access the SPARQL HTTP endpoint when enable_sparql_protocol is <code>true</code> .
sparql_spec_default_graph	false (boolean)	Controls the default scope of SPARQL queries when FROM clauses are excluded from a query. When false , queries without FROM clauses target the default graph (DEFAULTSET) only. Triples in named graphs will not be

Setting	Default Value (type)	Description
		included in the scope of the query. When true , Graph Lakehouse conforms to the SPARQL specification and includes the default graph and all named graphs in the scope of a query that omits the FROM clause. For more information, see Change the Default FROM Clause Behavior .
spill_directory	Not set (char)	The directory where Graph Lakehouse should save temporary query files that spill to disk. When not set, the default is <install_path>/spill. For more information, see Relocate Graph Lakehouse Directories. Important Graph Lakehouse uses O_DIRECT to read the spill files into the database. If you relocate the spill directory, make sure to place it on an ext4 file system that supports O_DIRECT.
ssl_protocol_port	8256 (int)	This setting specifies the port to use to access the SPARQL HTTPS endpoint when enable_ssl_protocol is true.
startup_info	1 (int)	Controls how verbose the database startup message is: - 0 -quiet, 1 -ready, 2 -ports, 3 -more.
stop_timeout	30 (int)	When the database stop command is issued, this setting specifies the number of seconds to wait for queries to finish before stopping the database.
truncate_clob	false (boolean)	Controls whether to automatically truncate large strings to

Setting	Default Value (type)	Description
		the maximum string size (2 MB).
use_custom_ssl_files	false (boolean)	Indicates whether you are replacing Graph Lakehouse's self-signed certificates with your own custom certificates. To configure Graph Lakehouse to use your certificates, place the certificate files in the <code><install_path>/config</code> directory and set <code>use_custom_ssl_files</code> to <code>true</code> . Restart the system management daemon (azgmgrd) as well as the database to apply the configuration change.
user_queues	64 (int)	This setting configures the limit on the number of queries that can run concurrently.
xray_sth_portion	0.001 (float)	This setting configures the percentage of total memory to use for storing historical system table information in memory before spilling to disk. The default value <code>0.001</code> = 0.1% of memory.
xray_sth_spool_duration	7days (char)	This setting controls the length of time to accumulate historical system table information on disk for xrays.
xray_sth_spool_maxgb	20 (int)	This setting controls the maximum size (in GB) per node of historical system table information to keep on disk for xrays. When the limit is reached, Graph Lakehouse deletes the oldest N records, where N depends on the server workload, but is typically about 5 to 6 minutes worth of system table data.

Advanced Settings

This section describes the settings in the Advanced section of settings.conf.

- `auto_restart_max_attempts`
- `auto_restart_time`
- `comm_enable_ssl`
- `comm_port_base`
- `compile_concurrent`
- `compile_max_memory`
- `compile_max_seconds`
- `compile_optimized`
- `copy_file_size`
- `cron_graphs`
- `cron_graphs_recheck`
- `enable_acl`
- `enable_external_auth`
- `enable_ocx`
- `enable_owlstats`
- `enable_refresh_stats_on_update`
- `enable_root_user`
- `enable_unbound_variables`
- `float_decimals`
- `float_format`
- `ignore_deniedlist_queries`
- `jvm_max_memory`
- `jvm_options`

- [policy_file_enabled](#)

Setting	Default Value (type)	Description
auto_restart_max_attempts	5 (int)	Specifies the number of times the system manager should attempt to start the database after a crash. The default value is 5 , which means the system manager will attempt to restart the database a maximum of 5 times. Changing <code>auto_restart_max_attempts</code> to 0 disables the auto-restart feature. For more information about the auto-restart feature, see Manage Automatic Database Restart Options .
auto_restart_time	600 (int)	Specifies the number of seconds to spend attempting to restart the database. If all attempts fail and this time limit is reached, the system manager stops trying to restart the database. The default value is 600 , which means that the system manager will attempt to restart the database for a maximum of 600 seconds (10 minutes). For more information about the auto-restart feature, see Manage Automatic Database Restart Options .
comm_enable_ssl	false (boolean)	This setting controls whether SSL rather than gRPC is used for communication between the nodes in a cluster.
comm_port_base	9100 (int)	This setting specifies the port to use for internal cluster communication.
compile_concurrent	8 (int)	This setting specifies the maximum number of generated code compilations to perform concurrently.
compile_max_memory	500 (int)	Sets the limit on the amount of memory (in MB) that Graph Lakehouse can allocate for compiling generated code before switching from optimized compile to non-optimized

Setting	Default Value (type)	Description
		compile.
compile_max_seconds	30 (int)	Sets the limit on the number of seconds to spend compiling generated code before switching from optimized compile to non-optimized compile.
compile_optimized	background (char)	Specifies the type of optimized compile to perform.
copy_file_size	5 (int)	This setting controls the size (in MB) of the Turtle files that are generated when graphs are exported to files. For more information, see Copy Graphs to Files .
cron_graphs	Not set (char)	This setting configures the comma-separated list of the cron graph names to enable. For information about cron graphs, see Schedule Automated Data Updates .
cron_graphs_recheck	10 (int)	This setting specifies the interval (in seconds) to wait before re-checking the cron_graphs value to determine if there are changes.
enable_acl	false (boolean)	This setting controls whether Graph Lakehouse's internal access control mechanism is enabled.
enable_external_auth	false (boolean)	This setting controls whether external access control is enabled. For information about access control, see Authentication and Access Control .
enable_ocx	true (boolean)	This setting controls whether support for OpenCypher and BOLT protocol is enabled. For information about OpenCypher support, see Cypher Query Language

Setting	Default Value (type)	Description
		Reference .
enable_owlstats	true (boolean)	In order to generate query execution plans, Graph Lakehouse needs to gather statistics about the data, such as the number of triples per graph and number of distinct subjects and predicates. This setting controls whether advanced statistics gathering, called OWL stats, is enabled. OWL stats use the metadata from data models to generate statistics. Altair recommends that you leave enable_owlstats enabled unless otherwise instructed.
enable_refresh_stats_on_update	true (boolean)	Controls whether the statistics in Graph Lakehouse are flagged as outdated when a graph is updated.
enable_root_user	false (boolean)	This setting controls whether to allow a user running with root privileges to start Graph Lakehouse.
enable_unbound_variables	false (boolean)	Controls whether Graph Lakehouse returns an empty result or an error if a query references a missing graph or includes unbound variables. This value is set to false by default, which means Graph Lakehouse returns an error. For more information, see Ignore Missing Graphs and Unbound Variables in Queries .
float_decimals	6 (int)	This setting does not apply to results that are returned from Graph Lakehouse to other Altair gRPC protocol applications. This setting would only affect results that are returned directly from Graph Lakehouse to another application over HTTP/S protocol.

Setting	Default Value (type)	Description
		Graph Lakehouse formats floating point types using the printf format string %.precision format, where precision is the value of the float_decimals, and format is the value of float_format. Note The interpretation of <code>float_decimals</code> differs depending on the value in float_format. For fixed point formats (f and F), <code>float_decimals</code> specifies the number of digits to include after the decimal point, padded with zeros if necessary. For floating point formats (e, E, g, and G), <code>float_decimals</code> specifies the number of significant digits to round the result to.
float_format	g (char)	This setting does not apply to results that are returned from Graph Lakehouse to other Altair gRPC protocol applications. This setting would only affect results that are returned directly from Graph Lakehouse to another application over HTTP/S protocol. Graph Lakehouse formats floating point types using the printf format string %.precision format, where format is the value of the float_format, and precision is the value of float_decimals. Valid values for float_format are e, E, f, F, g, or G. In the default configuration, a value of 10000000000.123 is returned as 1e+10.
ignore_deniedlist_	true (boolean)	Controls whether denied list queries are blocked from

Setting	Default Value (type)	Description
queries		running or are allowed to be run when the database is returned to normal operation. The default value is true , which means denied list queries are ignored. Incoming queries are not compared with the denied list and are permitted to run. If <code>ignore_deniedlist_queries</code> is false , denied list queries are not ignored and are therefore blocked from running until they are removed from the denied list. For more information about the auto-restart feature, see Manage Automatic Database Restart Options .
jvm_max_memory	Not set (char)	Specifies the maximum size of the heap that can be used by the embedded Java virtual machine (JVM). Use k , m , or g (case insensitive) for KiB, MiB, or GiB. You can also specify % to indicate a percentage of the total memory that is available to Graph Lakehouse. By default, this value is not set, which means <code>jvm_max_memory</code> defaults to either 5% of the total memory or 4g , whichever value is smaller. When not set, the default is 5% or 4g, depending on which value is smaller.
jvm_options	Not set (char)	Lists any optional parameters to use for configuring the embedded JVM. Use a semicolon-delimited (;) list to specify multiple parameters. For information about JVM options, see Options in the Java Documentation.
policy_file_enabled	false (boolean)	Enables or disables file system access control policies. When <code>policy_file_enabled</code> is false (the default value), Graph Lakehouse does not perform file path access checks when a query reads or writes files or directories on the file system. When <code>policy_file_enabled</code> is true and a query attempts to access a file or directory on the file

Setting	Default Value (type)	Description
		system, Graph Lakehouse performs the file path access checks that are configured in the <code>file_policy_*</code> settings and returns an access denied error message if the path is not accessible. For instructions on configuring file access policies and the <code>file_policy_read</code>, <code>write</code>, <code>delete</code>, and <code>deny</code> settings, see Manage File Access Policies.

Change System Settings

The default system configuration is optimized for most Graph Lakehouse installations. If want to change settings due to your own preferences, or if Altair Support recommends that you change the configuration, follow the instructions below to change setting values.

Note

Each time you start the database, Graph Lakehouse reads settings from the configuration file and stores the values in memory. After you save configuration changes, you must restart Graph Lakehouse for the new settings to take effect. **On a cluster, change `settings.conf` on the leader server only.** See [System Settings Reference](#) for information about the units of measurement for the settings as well as any special instructions.

1. Open the `<install_path>/config/settings.conf` file in a text editor.

The `settings.conf` file groups settings into two categories: **Basic** and **Advanced**. The Basic settings are intended for users with basic knowledge about Graph Lakehouse or databases in general, and the Advanced settings are intended for system administrators or users with an advanced level of knowledge about Graph Lakehouse or databases.

2. Locate the setting whose value you want to change.

Each setting has two lines: the first line describes the setting, and the second line lists the setting name and its current value (`setting=value`). Settings that are set to the default value are commented out. For example, in the following two lines, the first line describes the purpose of the `sparql_protocol_port` setting, and the second line lists the actual setting and its current value. Since the setting/value pair is commented out, it shows that `sparql_protocol_port` is set to the default value and has not been changed:

```
# sparql_protocol_port (overall) - The port to open for HTTP (SPARQL end-  
point) clients  
# sparql_protocol_port=7070
```

Note

If Altair Support has recommended that you change a setting that does not appear in the file, add a new line to the end of the file and specify the setting name and value. Graph Lakehouse applies settings from the top to the bottom of the file. If the same setting appears more than once, Graph Lakehouse applies the value for the last instance of the setting. The last instance overrides any previous instances.

3. If necessary, uncomment the line and then change the value for the setting. To add settings to `settings.conf`, add the new setting and value in the format below. Type each setting and value pair on a new line.

```
setting_name=value
```

4. When you finish changing settings, save and close the `settings.conf` file.
5. Restart Graph Lakehouse for the new settings to take effect.

In this section:

Manage File Access Policies	1080
Ignore Missing Graphs and Unbound Variables in Queries	1084
Change the Default FROM Clause Behavior	1085
Relocate Graph Lakehouse Directories	1086
Manage Automatic Database Restart Options	1087

Manage File Access Policies

In Graph Lakehouse and AnzoGraph DB Version 2.5.6 and later, you can configure file system access control policies to ensure that only certain files or directories are accessible to Graph Lakehouse during the execution of a query. This topic describes the configuration settings that define the file access policies and provides instructions for setting up policies.

- [File Access Policy Settings Reference](#)
- [File Access Control Behavior](#)

- [Setting Up File Access Policies](#)

File Access Policy Settings Reference

`policy_file_enabled`

The `policy_file_enabled` setting is the parent setting that controls whether or not file system access policies are enabled and followed. When `policy_file_enabled` is **false** (the default value), Graph Lakehouse does not perform file path access checks when a query references files or directories on the file system. When `policy_file_enabled` is **true** and a query attempts to access a file or directory on the file system, Graph Lakehouse performs the file path access checks that are configured in the [policy_file_read](#), [write](#), [delete](#), and [deny](#) settings described below.

`policy_file_read`, `write`, `delete`, and `deny`

The `policy_file_read`, `write`, `delete`, and `deny` settings specify the paths to directories and/or files on the file system that Graph Lakehouse requests are allowed to read from, write to, or delete from. For each of the "allowed" read, write, and delete settings, there is a corresponding **deny** setting that configures the paths for which requests are denied read, write, and delete access. This enables you to allow broad access to parent directories, if desired, and then use the deny settings to restrict access to certain subdirectories under them if needed.

The values for the settings are wildcard patterns that Graph Lakehouse uses to match directories and/or file names. Patterns are specified using basic file globbing syntax as described in the [glob \(7\) Linux manual page](#). Each `policy_file_*` setting accepts one or more patterns. Separate multiple patterns with a semicolon (;). For readability, you can also include spaces between patterns.

Important

Prior to matching paths in an incoming request to the configured access policy patterns, Graph Lakehouse resolves the paths in the request to canonical paths (using the `std::filesystem::weakly_canonical` function described [here](#) at cppreference.com). That means segments such as `./` or `../` are fully expanded prior to being compared to patterns. If a segment in the request path is a symlink, that segment is

also expanded prior to checking for a match. **Make sure that all access policy patterns match absolute paths.** Otherwise, expanded relative path or symlink segments in a request will not match any patterns. For example, if users normally include a path like `/source-files/` in a request but `/source-files/` is a symlink to `/mnt/anzoshare/data/source-files/`, include the path to `/mnt/anzoshare/data/source-files/` in the pattern.

The following list describes the settings and provides sample pattern values. The [File Access Control Behavior](#) section below includes specifics about pattern matching and access checks.

- **policy_file_read:** Specifies the pattern(s) to match for paths that queries have permission to read from. For example, a value such as the following gives Graph Lakehouse requests read-only access to all files and directories under the `/opt/share` and `/mnt/data` directories:

```
policy_file_read=/opt/share/* ; /mnt/data/*
```

- **policy_file_read_deny:** Specifies the pattern(s) to match for paths that queries should not be allowed to read. For example, the following value means requests will not be allowed to read any files or directories under `/etc` or `/root`:

```
policy_file_read_deny=/etc/* ; /root/*
```

- **policy_file_write:** Specifies the pattern(s) to match for paths that queries have permission to write to. For example, the following value gives requests write access to the `/tmp` and `/home` directories in addition to the `/opt/share` and `/mnt/data` directories.

```
policy_file_write=/tmp/* ; /home/* ; /opt/share/* ; /mnt/data/*
```

- **policy_file_write_deny:** Specifies the pattern(s) to match for paths that queries are denied write access to.
- **policy_file_delete:** Specifies the pattern(s) to match for paths that queries have permission to delete.
- **policy_file_delete_deny:** Specifies the pattern(s) to match for paths that queries are denied delete access to.

Note

The Graph Lakehouse installation path (`<install_path>/*`) is automatically added to each of the `*_deny` policies.

File Access Control Behavior

When a query that includes a path to a file or directory is run (such as in a GDI query with `s:url "/opt/share/data/csv"` or in a `LOAD <dir:/mnt/data/rdf.ttl.gz>` statement), Graph Lakehouse resolves that path (for example, if the path includes `./` or `../` segments) to a canonical path prior to checking whether it matches a `policy_file` pattern. If any segment of the path is a symlink, that segment is also expanded prior to being matched to a pattern. If the specified file or directory matches one of the allowed access patterns and it is not matched to a deny pattern, the query is executed. If the specified path is matched to a denied pattern or is not matched to any of the allowed patterns, the query is aborted and Graph Lakehouse returns an access denied error message.

Setting Up File Access Policies

1. Stop the database. See [Stop the Database and Leave the System Management Daemon Running](#) for instructions.
2. **On the leader node**, open the Graph Lakehouse settings file, **settings.conf**, in a text editor. The file is in the `<install_path>/config` directory.
3. In **settings.conf**, uncomment the `policy_file_enabled=false` line and change the value to true:

```
policy_file_enabled=true
```

4. Locate the additional `policy_file_*` settings:

```
# File system paths that may be deleted (';' delimited) ()
# policy_file_delete=

# File system paths that may not be deleted (';' delimited) ()
# policy_file_delete_deny=
```

```
# File system paths that may be read from (';' delimited) ()
# policy_file_read=

# File system paths that may not be read from (';' delimited) ()
# policy_file_read_deny=

# File system paths that may be written to (';' delimited) ()
# policy_file_write=

# File system paths that may not be written to (';' delimited) ()
# policy_file_write_deny=
```

5. Uncomment each of the `policy_file_*=` lines that you want to set, and add the wildcard pattern or patterns that you want to match for each of the policies.
6. Save and close `settings.conf`.
7. Restart the database to apply the configuration change. See [Start the Database \(the System Management Daemon is Running\)](#) for instructions.

Ignore Missing Graphs and Unbound Variables in Queries

By default, Graph Lakehouse returns a "No such graph or view" error and aborts the query if a query references a graph that does not exist. You can configure Graph Lakehouse to conform to the SPARQL specification and return an empty result instead of an error, however, if a query references a missing graph. Follow the instructions below to configure the system to return empty results instead of an error when a referenced graph does not exist.

1. Stop the database. See [Stop the Database and Leave the System Management Daemon Running](#) for instructions.
2. **On the leader node**, open the Graph Lakehouse settings file, **settings.conf**, in a text editor. The file is in the `<install_path>/config` directory.
3. In `settings.conf`, uncomment the `enable_unbound_variables=false` line and change the value to true:

```
enable_unbound_variables=true
```

4. Save and close `settings.conf`.

5. Restart the database to apply the configuration change. See [Start the Database \(the System Management Daemon is Running\)](#) for instructions.

Note

In addition to allowing queries that reference non-existent graphs to succeed, setting `enable_unbound_variables` to `true` also configures Graph Lakehouse to ignore unbound variables elsewhere in queries. For example, by default (when `enable_unbound_variables=false`), if a query includes a variable in the `SELECT` list that is not referenced in a `WHERE` clause pattern, Graph Lakehouse aborts the query and returns a "Named variable not in contained `WHERE` clause" error. When `enable_unbound_variables=true`, the user is not warned about unbound variables. Instead, the results are empty for the unbound variable. For example:

```
SELECT ?unbound ?person ?name
FROM <http://anzograph.com/people>
WHERE {?person <http://anzograph.com/people#firstname> ?name}
LIMIT 5
```

unbound	person	name
	person35632	Ross
	person20216	Quin
	person35859	Kellie
	person2551	Maris
	person24963	Madonna

5 rows

Change the Default FROM Clause Behavior

By default, if a query omits `FROM` clauses, the scope of the query is limited to the default graph (`DEFAULTSET`). Triples in named graphs will not be included in the scope of the query. The default behavior is controlled by the [sparql_spec_default_graph](#) configuration setting. To configure Graph Lakehouse to conform to the SPARQL specification and include the default graph and all named graphs in the scope of a query that omits the `FROM` clause, follow the instructions below.

1. Stop the database. See [Stop the Database and Leave the System Management Daemon Running](#) for instructions.
2. **On the leader node**, open the Graph Lakehouse settings file, **settings.conf**, in a text editor. The file is in the `install_path/config` directory.
3. In **settings.conf**, uncomment the `sparql_spec_default_graph=false` line and change the value to true:

```
sparql_spec_default_graph=true
```
4. Save and close **settings.conf**.
5. Restart the database to apply the configuration change. See [Start the Database \(the System Management Daemon is Running\)](#) for instructions.

Relocate Graph Lakehouse Directories

Follow the instructions in this section to designate alternate locations for certain directories included in the Graph Lakehouse installation. You have the option to relocate the **persistence** directory where the system saves the data in memory to the file system, the **internal** directory where the system saves database-related files such as logs and generated code, and the **spill** directory where the system saves any temporary query files that spill to disk.

You can change the settings described in this section at any time. Once you restart the database, Graph Lakehouse starts saving any new files in the directory locations that you specify. **The system does not relocate any existing directories or files. You can move the existing files manually if needed.**

1. Stop the database. See [Stop the Database and Leave the System Management Daemon Running](#) for instructions.
2. **On the leader node**, open the Graph Lakehouse settings file, **settings.conf**, in a text editor. The file is in the `<install_path>/config` directory.
3. In **settings.conf**, uncomment the lines for any of the following settings. Then edit the value portion of *setting=value* to specify the desired directory.

- [internal_directory](#): The directory where you want Graph Lakehouse to save internal database-related files such as generated code, logs, and query plans.
- [persistence_directory](#): The directory where you want Graph Lakehouse to save data when writing data to disk.
- [spill_directory](#): The directory where you want the Graph Lakehouse to save any temporary query files that spill to disk.

Important

Graph Lakehouse uses O_DIRECT to read the spill files into the database. If you relocate the spill directory, make sure to place it on an ext4 file system that supports O_DIRECT.

4. Save and close settings.conf.
5. Restart the database to apply the configuration change. See [Start the Database \(the System Management Daemon is Running\)](#) for instructions.

Manage Automatic Database Restart Options

If Graph Lakehouse shuts down unexpectedly, the system manager automatically restarts the database and evaluates the queries that were running at the time of the shutdown. This topic describes the process that occurs when Graph Lakehouse automatically restarts and provides information about the configuration settings that control the functionality as well as administrative information for managing the evaluated queries.

- [Automated Restart Procedure](#)
- [Automated Restart System Settings](#)
- [Removing a Query from the Block List](#)

Automated Restart Procedure

The steps below describe what occurs during the automatic restart process after Graph Lakehouse has crashed:

1. The system manager restarts the database in **safe mode**. In safe mode, Graph Lakehouse is locked to users and returns the following message if a user runs a query: "Graph Lakehouse is running in safe-mode. Cannot execute query." In addition, running `azgctl -status` to check the status of the database returns the message "Graph Lakehouse is running in safe-mode." If persistence is enabled, the data that was in memory at the time of the crash is reloaded into memory.
2. While in safe mode, Graph Lakehouse runs any queries that were inflight at the time of the crash. By executing the queries that were running, Graph Lakehouse tries to determine if the crash was directly caused by one of the inflight queries.
3. Depending on the outcome of running the inflight queries, Graph Lakehouse does the following:
 - If all inflight queries run to completion in safe mode, they are all added to the **warned_list**. In addition, each query is copied to a file named `<query_ID>.txt` in the `<install_path>/internal/auto_restart/<timestamp>/warned_list` directory.

Note

When all inflight queries complete successfully, that means it is unlikely that any one of the queries on its own is the culprit for the crash. However, all of the queries are added to the warned list because it is possible that the combination of queries run concurrently could have caused the crash.

- If any of the inflight queries fail or crash the database in safe mode, those queries are added to the **denied_list**. In addition, each query is copied to a file named `<query_ID>.txt` in the `<install_path>/internal/auto_restart/<timestamp>/denied_list` directory.

Note

If an inflight query fails, none of the inflight queries are added to the warned list. Instead, the failed queries are added to the denied list.

- If Graph Lakehouse runs a query in safe mode and cannot determine if it should be added to the denied or warned list, those queries are copied to a file named `<query_ID>.txt` in the `<install_path>/internal/auto_restart/<timestamp>/unanalyzed_list` directory.
- Metadata about the `warned_list`, `denied_list`, and `unanalyzed_list` queries is captured in the `stc_blocklist` system table.

Note

The `auto_restart_directory` setting in the system configuration file, `<install_path>/config/settings.conf`, controls the location of the `auto_restart` directories listed above. For more information about the setting, see the [Automated Restart System Settings](#) section below.

4. After the inflight queries have been run, Graph Lakehouse restarts the database, loads the persisted data back into memory, and returns the system to normal operation.

To help prevent the circumstance that caused the database to crash, any queries that were added to the **denied** list are blocked from being executed when the system returns to normal operation. When a user runs a query, Graph Lakehouse compares that query with the denied list. If the query is on the list, the query is terminated and Graph Lakehouse returns an "Attempting to execute a denied-listed query" error message. Queries on the warned list are not blocked. A denied list query cannot be run unless it is removed from the denied list. This behavior is controlled by the `ignore_deniedlist_queries` setting. For more information about the setting, see the [Automated Restart System Settings](#) section below. For information about removing queries from the denied list, see [Removing a Query from the Block List](#) below.

Automated Restart System Settings

The automatic restart feature is controlled by the following four settings in `<install_path>/config/settings.conf`:

- **auto_restart_max_attempts**: This setting specifies the number of times the system manager should attempt to start the database after a crash. The default value is **5**, which means the system manager will attempt to restart the database a maximum of 5 times. Changing `auto_restart_max_attempts` to **0** disables the auto-restart feature.
- **auto_restart_time**: This setting specifies the number of seconds to spend attempting to restart the database. If all attempts fail and this time limit is reached, the system manager stops trying to restart the database. The default value is **600**, which means that the system manager will attempt to restart the database for a maximum of 600 seconds (10 minutes).
- **auto_restart_directory**: This setting specifies the base location of the **auto_restart** directory, which contains the `denied_list`, `warned_list`, and `unanalyzed_list` directories. The default value is `<install_path>/internal`.
- **ignore_deniedlist_queries**: This setting controls whether denied list queries are blocked from running or are allowed to be run when the database is returned to normal operation. The default value is **false**, which means denied list queries are not ignored and are therefore blocked from running. If `ignore_deniedlist_queries` is **true**, incoming queries are not compared with the denied list and are run.

Important

Changing the `auto_restart_max_attempts`, `auto_restart_time`, or `auto_restart_directory` values requires a restart of the system management daemon, **azgmgrd**, as well as the database. See [Start and Stop Graph Lakehouse](#) for instructions.

Removing a Query from the Block List

Graph Lakehouse stores metadata about the denied and warned list queries in the **stc_blocklist** system table. To remove a query from either list, you remove the entry from the `stc_blocklist` table by running the `REMOVE_FROM_BLOCKLIST` command.

```
REMOVE_FROM_BLOCKLIST '<list_name>' <query_ID>
```

Where <list_name> is the name of the list that the query is on and <query_ID> is the ID number for the query. To retrieve the list name and query ID values, run the following query to return the stc_blocklist contents:

```
SELECT * WHERE { TABLE 'stc_blocklist' } ORDER BY ?blocklist
```

For example:

```
/opt/anzograph/bin/azgi -c "select * where {table 'stc_blocklist'} order by ?blocklist"
```

query	blocklist	updated	query_text	part
3587	denied_list	2020-08-25 14:29:27	select * from <http://an..	0
3592	denied_list	2020-08-25 14:29:32	select * where {?s ?p ?o}	0
3612	warned_list	2020-08-25 14:32:15	select * from <http://an..	0

In the results, the <list_name> is the value in the **blocklist** column, and <query_id> is the value in the **query** column. Running the following command removes the first entry from the stc_blocklist table, which removes that query from the denied list.

```
REMOVE_FROM_BLOCKLIST 'denied_list' 3587
```

Develop

Graph Lakehouse exposes a number of extension points that developers can use to customize and extend their system's analytic capabilities. The extension point interfaces and the user code that implements them are called user-defined extensions (UDX). Currently, Graph Lakehouse offers C++ and JVM APIs that developers can use to implement user-defined functions and other extensions. JVM extensions, for example, include those developed in languages such as Java or Scala.

Note

Developing C++ UDXs requires a compiler that is compatible with C++ version 11 or later. You may also use any number of vendor IDEs that are compatible with these requirements, such as Eclipse or Microsoft Visual Studio Code. Developing JVM UDXs requires that you install OpenJDK 11. Apache Maven 3.6.2 is also useful in JVM environments for compiling and packaging JVM source files into JAR files.

The topics in this section introduce you to the fundamental concepts of developing user-defined extensions and provide instructions for using either the Graph Lakehouse C++ or JVM-based API to create these user-defined extensions.

In this section:

UDX Terminology and Concepts	1093
Developing User-Defined Extensions	1102
Loading a UDX to the Database	1129
Using Extensions in SPARQL Queries	1130
UDX Examples	1131

UDX Terminology and Concepts

This topic introduces the Graph Lakehouse user-defined extensions (UDX) interface and describes fundamental terminology and concepts associated with developing custom Graph Lakehouse extensions that implement the UDX interface. Subjects covered here are the following:

- [Extension Types](#)
- [Extension Libraries](#)
- [Extension Metadata](#)
- [Extension Data Types](#)
- [Data Type Handling](#)

Extension Types

Graph Lakehouse extensions are programs that implement the UDX interface and can be registered and loaded into the Graph Lakehouse system where they can be used within queries or other command statements. Graph Lakehouse currently supports three different kinds of extensions. Each extension has similar but distinct requirements:

- **User-Defined Functions (UDF):** A UDF extension maps or processes a single row of input values to return a single row of output values. For example, a developer can design a UDF extension to create an analytic function, such as those that concatenate values or convert integers to alternate currencies.
- **User-Defined Aggregates (UDA):** A UDA extension maps or processes multiple rows of input values to return a single row of output values. For example, a developer can design a UDA extension, such as those that compute an arithmetic mean, or perform operations like SUM, STDDEV, or MAX. Unlike a UDF, which returns a distinct value each time it is applied, a UDA aggregates the collection of values to which it is applied into a single summary value.
- **User-Defined Services (UDS):** A UDS extension maps or processes multiple rows of input values to return multiple rows of output values. For example, a developer can design and register a UDS extension that defines a SPARQL endpoint.

Extension Libraries

Extension libraries are executable code modules that define and organize a collection of extensions. Libraries can be implemented in either C++ or any JVM-based language such as Java or Scala. Developers can create and register any number of extension libraries.

Extension Metadata

Extension libraries are self-describing; that is, they include the necessary metadata that describe the number, name, type, and calling signature of the various extensions they implement. When a new UDX is implemented, the developer adds the metadata to an extension library that describes each new UDX. When the extension library is loaded into Graph Lakehouse, the system adds the extension library metadata to an internal Graph Lakehouse registry so that the new UDX can be invoked from within subsequent SPARQL queries.

Extension Data Types

The following table describes the types of values that can be passed into and returned from a user-defined extension. For each type, we can specify:

- Enum Type: A unique number that identifies the data type.
- RDF Type: The name by which the type is known within the SPARQL query language.
- C++ Type: The type by which it is known within the C++ language.
- JVM Type: The type by which it is known within the JVM language.
- UDX Registry Data Type: The language-independent name by which it is known within the Graph Lakehouse registry.

UDX Data Types

The following table describes mapping for the various data types that can be specified in an Graph Lakehouse user-defined extension.

Note

The data types listed in the table describe values that can be passed into and out of a user-defined extension. In C++, we do this by placing the values into the elements of a row. In JVM languages, the values are passed on the stack as explicit parameters to the relevant UDX.

Enum Type	RDF Type	Description	C++ Type	JVM Type	
t_boolean	xsd:boolean	A non-nullable 8-bit boolean value	bool	boolean	boolean
t_byte	xsd:byte	A non-nullable 8-bit signed integer	byte/uint8_t	byte	byte
t_short	xsd:short	A non-nullable 16-bit signed integer	short/int16_t	short	short
t_int	xsd:int	A non-nullable 32-bit signed integer	int/int32_t	int	int
t_long	xsd:long	A non-nullable	long/int64_t	long	long

Enum Type	RDF Type	Description	C++ Type	JVM Type	
		64-bit signed integer			
t_float	xsd:float	A non-nullable 32-bit IEEE single precision float	float	float	float
t_double	xsd:double	A non-nullable 64-bit IEEE double precision float	double	double	double
t_Object	N/A	A direct sum of all possible nullable types	--	java/lang/Object	Object
t_Boolean	xsd:boolean	A nullable 8-bit boolean value	bool	java/lang/Boolean	Boolean
t_Byte	xsd:byte	A nullable 8-bit	byte/uint8_t	java/lang/Byte	Byte

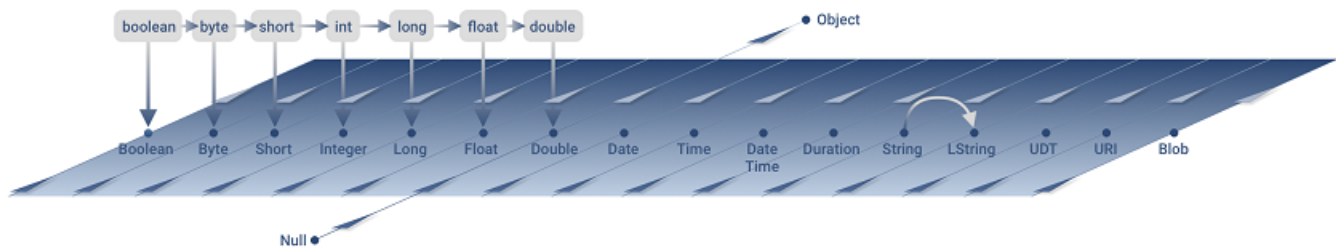
Enum Type	RDF Type	Description	C++ Type	JVM Type	
		signed integer			
t_Short	xsd:short	A nullable 16-bit boolean integer	short/int16_t	java/lang/Short	Short
t_Integer	xsd:int	A nullable 32-bit signed integer	int/int32_t	java/lang/Integer	Int
t_Long	xsd:long	A nullable 64 bit signed integer	long/int64_t	java/lang/Long	Long
t_Float	xsd:float	A nullable 32-bit IEE single precision float	float	java/lang/Float	Float
t_Double	xsd:double	A nullable 64-bit IEE double precision float	double	java/lang/Double	Double
t_Date	xsd:date	A nullable	udx2::Date	java/time/LocalDate	Date

Enum Type	RDF Type	Description	C++ Type	JVM Type	
		32-bit signed number of days since 1/1/2000			
t_Time	xsd:time	A nullable 64-bit signed number of microseconds since 1/1/2000	udx2::Time	java/time/OffsetTime	Time
t_DateTime	xsd:dateTime	A nullable <us, time zone> pair - since 1/1/2000	udx2::DateTime	java/time/ZonedDateTime	DateTime
t_Duration	xsd:duration	A nullable <months, us> pair - since 1/1/2000	udx2::Duration	java/time/Duration	Duration
t_String	xsd:string	A nullable	udx2::String	java/lang/String	String

Enum Type	RDF Type	Description	C++ Type	JVM Type	
		view into a string of UTF8 characters			
t_LString	xsd:string	A nullable pair of string views	udx2::LString	com/cambridgesemantics/anzograph/udx/LString	LString
t_UDT	N/A	A nullable pair of string views	udx2::UDT	com/cambridgesemantics/anzograph/udx/UDT	UDT
t_URI	IRI	A nullable view into a string of UTF8 characters	udx2::String	com/cambridgesemantics/anzograph/udx/URI	URI
t_Blob	N/A	A nullable block of raw binary bytes	udx2::Blob	com/cambridgesemantics/anzograph/udx/Blob	N/A

Data Type Handling

The illustration below provides a diagram of Graph Lakehouse's UDX data type handling. The top row in the diagram shows the built-in primitive types, and the bottom plane shows the corresponding reference types. The arrows pointing from primitive types to corresponding reference types represent automatic coercions. Details about data type processing and automatic type coercion follow the diagram.



Primitive Types

The top row in the diagram depicts non-nullable types that are native to both the C++ and JVM languages.

If a UDX registers itself as requiring a primitive type as one of its arguments, but it receives a null value at run time, the system generates an exception and the query is aborted. Similarly, if a UDX registers itself as returning a primitive type as one of its results, but it actually returns a null value, the system also generates an exception and the query is aborted.

Note

Passing and returning values of primitive types is generally faster than using the corresponding reference types, and thus, is preferred whenever possible for best performance.

Reference Types

The reference types shown in the bottom plane of the diagram represent data values that are passed by reference. These types are ultimately derived from "Object," have methods, are instances of classes, and are interrogated at run-time for their type. Reference types are also

nullable. Each primitive type (boolean, byte, short, int, long, float, double) has a corresponding reference type that it is mapped to (Boolean, Byte, Short, Integer, Long, Float, Double).

Note

Passing and returning values as reference types is generally slower than using their primitive counterparts, but using reference types often provide more flexibility.

Data Type Coercion

Graph Lakehouse supports automatic type coercion of certain data types. These data types are represented by the downward-pointing arrows in the previous diagram showing Graph Lakehouse data type mapping. Where automatic conversion is supported, a value of one type can be supplied to a UDX where a value of another type is generally prescribed, and Graph Lakehouse will convert the data type without a loss of information or precision.

For example, if a UDX expects a **Double** value as an input argument and the value supplied is an **int**, Graph Lakehouse coerces the value as follows:

```
int→long→float→double→Double
```

If a UDX requires a **long** value, but an **int** is supplied, Graph Lakehouse converts the **int** from a 32-bit signed integer to a 64-bit signed integer **3L** type and clears out the high 32 bits.

Developing User-Defined Extensions

The topics in this section provide guidance on developing extensions.

In this section:

- UDX Development Process Overview 1103
- Reviewing UDX Interface Files 1105
- Creating New UDX Library Source Files 1116
- Registering a UDX in an Extension Library 1121
- Compiling UDX Source Files 1126

UDX Development Process Overview

This topic provides a summary of the procedure for developing and deploying new user-defined extensions (UDX) created in either C++ or JVM environments.

1. Familiarize yourself with the UDX interface, the extension point classes, input and output parameters, data members and methods defined in the `udx_api.hpp` include file (for C++ UDX development), or imported from the `com.cambridgesemantics.anzograph.udx` package (for JVM UDX development). For information about the UDX interface, see [Reviewing UDX Interface Files](#).
2. Create the C++ `.cpp` or JVM source files for a user-defined extension library. (Each source library file may contain one or more UDX functions, aggregates, or service extensions.)

Tip

For C++ environments, the `udx_api.hpp` include file provides a description of classes, data members, and methods available for C++ UDX development. For JVM environments, Graph Lakehouse provides two JAR files in the `<install_path>/lib/jar` directory:

- `anzograph-udx-api-2.1.0` – Graph Lakehouse UDX Interface library.
- `anzograph-udx-api-2.1.0-javadoc` – Full HTML API documentation describing Graph Lakehouse JVM UDX interface library packages, classes, annotations, data members and methods.

3. Add the required metadata statements to your UDX library source file so you can register the new UDXs in your source file as an extension library available in Graph Lakehouse. You may create one or more separate UDX or extension library files to register and use in Graph Lakehouse.
4. Compile the UDX library source file into a shared object file for C++ based extension libraries or a JAR file for JVM-based extension libraries.
5. Place the C++ shared object (`.so`) or JAR files, and any files or libraries the extensions are dependent on, in a pre-configured location on the Graph Lakehouse leader node of a server

cluster. Each time Graph Lakehouse is started, the system loads the C++ shared object or JAR files and adds any registered extensions to the Graph Lakehouse registry.

Reviewing UDX Interface Files

This topic provides more detailed information on the content of UDX API interface files available for UDX development in C++.

Note

Documentation on UDX development in JVM environments is in progress and not available at this time.

Reviewing the C++ UDX Include File

The following sections describe each of the sections within the `udx_api.hpp` file. This file is located in the `<install_path>/include` directory.

- [Overview](#)
- [Data Types](#)
- [Extension Points](#)
- [System Information](#)
- [Memory Allocation](#)
- [Exceptions](#)
- [Implementation Details](#)
- [Reviewing the C++ UDX Include File](#)

Overview

The beginning portion of the `udx_api.hpp` file simply provides some versioning information and description of major changes among different versions of the include file. The top portion of the include file also provides additional directives to the C++ compiler regarding additional system library includes, a name space declaration of various function and variable scope, and the metadata structure that provides various version compatibility information.

Data Types

This section of the C++ include header (`.hpp`) file provides several different things:

- Enumeration of available UDX data types, for example, `t_Null = 0`, `t_boolean = 1`, `t_byte = 2`, and so on.
- Listing of various aliases mapped to corresponding UDX data types with the `using` keyword directive or declaration, for example, `using blob = std::int64_t;`, `using Date = std::int32_t`, and so on.
- Declarations of a public `DateTime()` class along with data members and several different function call signatures to store and return various date, time, and timezone information.
- Various public helper functions:
 - `Duration()` – calculate months or time (in microseconds) since January 1, 2000.
 - `LString()` –
 - `UDT()` –
 - `Blob()` –
- An `Allocated` structure that developers can use to allocate space for objects with memory managed by Graph Lakehouse
- A `Row` class that stores and returns information on UDX input arguments and result shapes or schema. Graph Lakehouse uses rows to marshal values in and out of user code.
- A namespace that provides operations to insert data values and other information, for example, schemas, rows, datetime values, and blobs, into memory,

Extension Points

This section defines different extension point interfaces that developers can implement to create different types of extensions in an UDX library file:

- User Defined Function (UDF) looks to the user like a normal function that maps one row of input values to one row of output values.

- User Defined Aggregate (UDA) looks to the user like a normal function that maps many rows of input values to one row of output values.
- User Defined Service (UDS) looks to the user like a *service* that maps many rows of input values to many output values, or like a *table* that maps zero rows of input values to many output values.

An `Extension` struct provides a common base from which all user-defined extensions are derived.

Extension Library Meta-Data

C++ extensions are compiled and linked into *extension libraries*, which are modules of executable code that export a meta-data description of their contents. Formally, a C++ extension library is any Linux shared library that exports an entry point of the form:

```

/**
/**  extern "C" void register_extensions(MetaData &md) {
/**      md.json_metadata = R"({
/**          "name"          : <name>,
/**          "language"      : "c++",
/**          "version"       : <version>,
/**          "description"   : <description>,
/**          "author"        : <author>,
/**          "copyright"     : <copyright>,
/**          "contents"      : [ <extension> * ]
/**      })";
/**  }
/**

```

The values of each attribute in the JSON metadata have the following meaning:

Field	Description
name	The name of the library; a string that distinguishes it from all other such libraries currently installed within the system.
language	Hard-coded to "C++" for UDX libraries created with C++.
version	The version of the library; specified as a string of the form

Field	Description
	"MAJOR.MINOR.PATCH".
description	A brief description of the library and the kind of functionality that it provides. This attribute is optional.
author	The author of the library. This attribute is optional.
copyright	Any copyright that may pertain to the library. This attribute is optional.
contents [<extension> *]	Specifies a meta-data description of each extension defined with the same source file and exported to Graph Lakehouse from the library.

Extension Meta-Data

The `contents` attribute of the Graph Lakehouse extension library meta-data fields enumerates JSON meta-data descriptions of each extension defined within the same extension library source file. Each type of extension point share a certain number of attributes in common:

Field	Description
name	The name-space qualified name of the extension as it appears to a user of the query language.
type	The type of the extension point implemented: "function", "aggregate", or "service". , or "table". (Use of the "table" type is deprecated, since the same functionality can be achieved using the service extension type.)
signature	The name of the exported entry point that returns an "ExtensionFactory" for the given extension.
description	A brief description of the extension itself. This attribute is optional.

Note

See [Registering a UDX in an Extension Library](#) for more information on specifying entries in a C++ source file for the extension library and all the UDXs defined within the same library source file. For examples on creating C++ UDXs of each extension type, see [UDX Examples](#).

UDX Type Structures and Meta-Data

The remaining portions of the Extension Point section of the `udx_api.hpp` include file provide template class structures for the construction of every supported UDX type: **Function**, **Aggregate**, **Service**, and **Table** (deprecated). Each of the extension type classes also provide methods for processing input arguments passed to extensions or values returned from those same extensions.

Class Structure	Description
<code>struct Function : Extension</code>	The Function class represents, perhaps, the simplest of the extension points, and allows developers to extend the set of analytic functions already built into the query language seen by the end user. A function extension applies some operation to the given arguments and returns an output row result based on the function's implementation.
<code>struct Aggregate : Extension</code>	The Aggregate class enables developers to extend the set of aggregate functions that are already built into Graph Lakehouse for use in queries. Aggregate functions process a given row of values across ...
<code>struct Service : Extension</code>	The Service class represents perhaps the most powerful of the extension points and ...
<code>struct Table : Extension</code>	Deprecated.

Each of the different extension type classes has additional attributes specified as part of UDX library meta-data.

Additional Attributes for the Function Extension

In addition to those attributes common to every extension, the following attributes are specified for extension functions.

Attribute	Description
arguments	An array of zero or more types that specifies the number and type of arguments required by any application of the UDF. Determines the shape of the input row passed as the "a" parameter of the <code>apply</code> method.
results	An array of zero or more types that specifies the number and type of results that are returned by any application of the UDF. Determines the shape of the output row passed as the "r" parameter of the <code>apply</code> method.
variadic	A boolean value which indicates, if true, that the final type listed in the arguments array may be repeated one or more times in an application of the UDF. This attribute is optional.

When compiling a query including a UDF, for example, the leader verifies that the number and type of the arguments passed to the UDF are consistent with its domain, as specified by the **arguments** and **variadic** attributes of the meta-data description. When executing a query, each slice creates its own distinct instance of the UDF for every occurrence in the query by invoking the `create` method of the associated `ExtensionFactory`.

As each row of values streams through a slice, it is passed to the instance by calling its `apply` method, and the results are then passed up to the consumer of the stream. The factory, and the instances that it creates, are destroyed only when the query has eventually finished executing.

Additional Attributes for the Aggregate Extension

In addition to those attributes common to every extension, the Aggregate extension has the following additional meta-data attributes.

Attribute	Description
arguments	An array of zero or more types that specifies the number and type of arguments required by any application of the UDF. Determines the shape of the input row passed as the 'a' parameter of the <code>accumulate</code> method.
results	An array of zero or more types that specifies the number and type of results that are returned by any application of the UDF. Determines the shape of the output row passed as the 'r' parameter of the <code>result</code> method.
variadic	A boolean value which, if true, indicates that the final type listed in the arguments array may be repeated one or more times in an application of the UDA. This attribute is optional.
states	An array of zero or more types that specifies the number and type of states that are marshaled across the cluster when merging those intermediate results that accumulated on the slices during the accumulation phase. Determines the shape of the output row passed as the "s" parameter of the <code>save</code> method and determines the shape of the input row passed as the "s" parameter of the <code>merge</code> method. This attribute is optional.

When compiling a query, the leader verifies that the number and type of the arguments passed to the UDA are consistent with its domain, as specified by the **arguments** and **variadic** attributes of the meta-data description. A slice of the cluster is designated as the receiver of the final aggregate result.

When executing a query, each slice creates its own distinct instance of the UDA for every occurrence in the query by invoking the `create` method of the associated `ExtensionFactory`.

- As each row of values streams through a slice, it is passed to the instance by calling its `accumulate` method, which responds by updating its internal state as necessary to record having processed the row in some appropriate way.

- When all rows on the slice have been accumulated, the instance is now given a mutable row (of shape `states`) into which it serializes any intermediate results it has accumulated, and instance is then destroyed.
- The slice receiving the result now creates an instance of the UDA and a row (of shape `states`), and the system arranges for all intermediate states to be transmitted across the cluster and 'merged' into the instance by passing each in turn to its `merge` method.

The factory, and the instances that it creates, are destroyed only when the query has eventually finished executing.

System Information

Graph Lakehouse also provides a number of utility functions that can be invoked at any time by any UDX. These functions let you query basic system information useful in providing more exact control of a UDX's execution.

The following table provides a brief description of these functions.

Function	Description
<code>amPlanning()</code>	Boolean; returns true if the caller is being invoked during the execution of a query on the nodes of the cluster.
<code>amExecuting()</code>	Returns the identifier of the worker node on which the caller is executing as an integer in the range 0 to <code>getNodes()</code> .
<code>getNode()</code>	Returns the total number of worker nodes in the cluster.
<code>getNodes()</code>	Returns the identifier of the slice on which the caller is executing; returns an integer in the range 0 to <code>getSlices()</code> .
<code>getSlice()</code>	Returns the total number of slices in the cluster.
<code>getSlices()</code>	Returns a string of the form "<MAJOR>.<MINOR>,<PATCH>" that describes the Graph Lakehouse API version supported by this server.

Function	Description
<code>getHostVersion()</code>	Returns the source text of the currently executing query.
<code>getQueryText()</code>	Returns any additional data that may have been supplied by the client along with the text of the query that is currently executing. The client context string that is returned may take any form whatsoever; the server does not parse it in any way, but merely makes it available to extensions unmodified via this access function.
<code>getClientContext()</code>	Logs a null-terminated string and an associated integer to the <code>sth_udx</code> system table. The string includes the namespace-qualified name of the extension that is logging a given message, an integer indicating the severity level of the event, and the raw text of the message to be logged. The 'level' ostensibly describes the 'severity' of the given logging event, but can in fact represent anything; the system does not interpret its value in any particular way.
<code>logText(...)</code>	Logs a null-terminated string and an associated integer to the <code>sth_udx</code> system table. This variant of the log function formats any additional details into the given message template printf style before forwarding the arguments on to the <code>logText</code> function.
<code>vlog(...)</code>	Provides another variant of the logging functions.
<code>log(...)</code>	

Note

Refer to the `udx_api.hpp` include file for additional comments and descriptions of parameters for each of the listed functions.

Memory Allocation

Developers are encouraged to call the following functions to acquire memory for extensions from the system's free memory, thus allowing the server to monitor an extension's usage of memory, warn of possible leaks, and generally ensure the smooth running of the system as a whole.

```
void *malloc(size_t);  
  
void *calloc(size_t, size_t);  
  
void *realloc(void *, size_t);  
  
void free(void *);
```

When allocating memory for large objects, or allocating memory for objects that will outlive the current stack frame (for example, assigning to a data member of an extension), it is recommended you used one of the following options:

- Use a standard library container that is parameterized on the new **udx2::allocator**:

```
struct agg : Aggregate {  
  
    std::vector<int,udx2::allocator<int>> m_vec = { ... };  
};
```

- Use a smart pointer:

```
struct agg : Aggregate {  
  
    udx2::unique_ptr<Object> = udx2::make_unique<Object>( ... );  
};
```

- call **udx2::mallo**, etc:

Objects with indeterminate lifetime should use one these three techniques, if at all possible. For other situations, you can continue to use small, short-lived objects as before, for example:

```
bool foo(const std::string& x) { ... }  
  
bool bar () { std::vector<int> = {... }; ... }
```

Exceptions

Exceptions thrown by extension code are caught by the server, which aborts the currently executing query and reports the error to the user. This section provides a collection of macros that developers can use to throw exceptions. It is recommended that developers use these macros, as other methods may trap exceptions whose error messages are not as meaningful.

Note

Exceptions thrown by extension code are caught by the server, which halts execution of the query containing an offending UDX and reports an error to the user.

- `azg_throw(extension, message, ...)` – Formats the given arguments as a user visible message and throws this as an exception.
- `azg_check(expression, extension, message, ...)` – Evaluates the given expression. If false, formats the given arguments as a user visible message and throws this as an exception.

Note

Refer to the `udx_api.hpp` include file for a description of parameters for each of the listed functions.

Implementation Details

This section provides various internal Graph Lakehouse namespace and utility operations required for UDX creation, class templates, type conversions, data streaming, and so on.

Creating New UDX Library Source Files

This topic provides information on creating new extension source files in C++ environments. You can add new extensions by creating a C++ source library file that define the operations performed by one or more UDX functions, aggregates, or services. Within the same source library file, you can create new extensions of different types (UDF, UDA, or UDS), which differ based on the number and shape of UDX input arguments and returned results.

Note

Documentation on UDX development in JVM environments is in progress and not available at this time.

Creating New Extension Source Files in C++

1. Create or edit a C++ `.cpp` source file to define new UDX function, aggregate, or service implementation details.
2. At the top of the `.cpp` file, add the following line to include the UDX header file, **udx_api.hpp**. The header file is in the `<install_path>/include` directory and defines the UDX classes, class data members and methods, utility functions, macros, and other declarations available for your use in implementing Graph Lakehouse UDX function, aggregate, or service functionality:

```
#include "udx_api.hpp"
```

3. Graph Lakehouse's API entities are defined in the **udx2** namespace. Add the following line to the file to use the `udx2` namespace:

```
using namespace udx2;
```

4. Next, for a single extension, specify the interface for the type of UDX you want to the implement: a function, aggregate, or service. Then, use methods provided for the specified extension type to process input and output parameters for the extension.

For example, to add a user-defined function (UDF) extension (which maps a single row of input values to a single row of output values), add the **Function** declaration to instantiate a named object of the Function class type. Then, add the **apply()** method statement that Graph Lakehouse calls to invoke the UDF.

```
struct function_name : Function {
    void apply(const Row& a, Row& r);
};
```

In this example, *function_name* is the short name that you want to use for the function. In the `apply()` method, argument **a** is the input to the UDF and argument **r** is the output the UDF returns. Both arguments are of type **Row**; argument **a** is a constant reference, and argument **r** is a non-constant reference. In the `apply()` function, include the appropriate **get** and **set** routines to define the Row type and read or fetch the input values for argument **a** and return the values for argument **r**.

The following table lists the get and set routines available for Graph Lakehouse extensions. Use the **Read Cell** and/or **Read Cell with Default** get routines for argument **a**. The Read Cell with Default routines are used to return a default value if the cell is not defined (empty). Use the **Write Cell** set routines for argument **r**:

Read Cell (arg a)	Read Cell with Default (arg a)	Write Cell (arg r)
defined(size_t)	N/A	undefined(size_t)
getBoolean(size_t)	getBoolean(size_t, bool)	setBoolean(size_t, bool)
getByte(size_t)	getByte(size_t, uint8_t)	setByte(size_t, byte)
getShort(size_t)	getShort(size_t, short)	setShort(size_t, short)
getInt(size_t)	getInt(size_t, int)	setInt(size_t, int)

Read Cell (arg a)	Read Cell with Default (arg a)	Write Cell (arg r)
getLong(size_t)	getLong(size_t, long)	setLong(size_t, long)
getFloat(size_t)	getFloat(size_t, float)	setFloat(size_t, float)
getDouble(size_t)	getDouble(size_t, double)	setDouble(size_t, double)
getDate(size_t)	getDate(size_t, Date)	setDate(size_t, Date)
getTime(size_t)	getTime(size_t, Time)	setTime(size_t, Time)
getDateTime(size_t)	getDateTime(size_t, DateTime)	setDateTime(size_t, DateTime)
getDuration(size_t)	getDuration(size_t, Duration)	setDuration(size_t, Duration)
getString(size_t)	getString(size_t, String)	setString(size_t, String)
getLString(size_t)	getLString(size_t, LString)	setLString(size_t, LString)
getUDT(size_t)	getUDT(size_t, UDT)	setUDT(size_t, UDT)
getURI(size_t)	getURI(size_t, URI)	setURI(size_t, URI)
getTag(size_t)	getTag(size_t, String)	N/A
getType(size_t)	N/A	N/A
N/A	N/A	clear()

Read Cell (arg a)	Read Cell with Default (arg a)	Write Cell (arg r)
getBlob(size_t)	getBlob(size_t, Blob)	setBlob(size_t, Blob)

Tip

These routines may also be used in an extension's programming logic that implements the operation of a particular extension.

- Implement the programming logic that provides the functionality of the specific extension, function, aggregate, or service. For example, as shown in the following code snippet for a **concat** UDF extension, the **apply()** function concatenates two strings and returns the concatenated string:

```
struct concat : Function {
    void apply(const Row& a, Row& r)
    {
        r.setString(0, string(a.getString(0)) + string(a.getString(1)));
    }
};
```

- If you want to implement error handling, you can include the following **azg_throw** macro, which is similar to the **printf()** function in C. Create and specify the full URI for the function as a string, and include the message to display when an error occurs.

```
azg_throw("function_URI", "error_message");
```

The *function_URI* argument is a prefix that you define, followed by the function name. The URI must be globally unique. Altair recommends that you use a format such as **http://mycompany.com/grouping/etc#function_name**. For example:

```
azg_throw("http://cambridgesemantics.com/udx/function#concat", "Error message - code %d", m_code);
```

Note

You register the function URI after you complete implementation for all extension definitions in the library file and then register the entire extension library for all the extensions in the same source file. (See [Registering a UDX in an Extension Library](#).)

7. Include statements to instantiate extensions by specifying the following **extern "C"** extension factory function for each extension defined within the same extension library source file:

```
extern "C" ExtensionFactory* udx_functionName() { return new
FactoryFor<functionName>(); }
```

Where *functionName* is the short name of the function, not the full URI. For example:

```
extern "C" ExtensionFactory* udx_concat() { return new FactoryFor<concat>(); }
```

Once you've finished implementing the operation of all extensions within an extension library file, you register the extensions. For instructions on how to do that, see [Registering a UDX in an Extension Library](#).

Tip

For more information on creating specific user-defined extension types in C++, see [UDX Examples](#).

Registering a UDX in an Extension Library

This topic provides information on registering a new user-defined extension (UDX) in C++ environments.

Note

Documentation on UDX development in JVM environments is in progress and not available at this time.

Registering C++ UDXs

The following instructions show you how to register the UDF by specifying the function's metadata in JSON format. You can supply the registration information in the same CPP file that you created to define the function, or you can create a separate CPP file for the registration metadata. You can also register multiple functions in the same CPP file.

To register a UDF, add the following `extern "C" void register_extensions (MetaData& md)` declaration to the CPP file that defines the UDF or a separate CPP file.

Note

If you register UDFs in a separate file, make sure that you also include the UDX header and namespace in that file.

 [Open register.cpp in a separate window](#)

```
extern "C" void register_extensions (MetaData& md)
{
    md.json_metadata = R" (
    {
        "name"           : "UDX_lib_name",
        "language"       : "c++",
        "version"        : "version",
        "description"     : "UDX_lib_description",
        "author"         : "author_or_company",
        "copyright"      : "copyright_statement",
        "contents"       : [
            {
```

```

    "name"          : "function_URI",
    "type"          : "function",
    "signature"     : " udx_functionName",
    "arguments"     : ["arg1_type", "arg2_type"],
    "results"       : "result_type",
    "description"   : "function_description"
  }
]
}
)";
}

```

The properties at the top of the declaration describe the overall extension library. The properties in the **contents** array describe the UDF to register. To describe multiple UDFs in a single CPP file, include multiple contents arrays, for example:

```

extern "C" void register_extensions(MetaData& md)
{
  md.json_metadata = R"(
  {
    "name"          : "UDX_lib_name",
    "language"      : "c++",
    "version"       : "version",
    "description"   : "UDX_lib_description",
    "author"        : "author_or_company",
    "copyright"     : "copyright_statement",
    "contents"      : [
      {
        "name"          : "function_URI_1",
        "type"          : "function",
        "signature"     : " udx_functionName_1",
        "arguments"     : ["arg1_type", "arg2_type"],
        "results"       : "result_type",
        "description"   : "function_description"
      },
      {
        "name"          : "function_URI_2",
        "type"          : "function",
        "signature"     : " udx_functionName_2",
        "arguments"     : ["arg1_type", "arg2_type"],
        "results"       : "result_type",
        "description"   : "function_description"
      }
    ]
  }
)";
}

```

```

    ]
  }
)";
}

```

The table below defines each property that is specified for each extension in an extension library file:

Property	Description
name	Required. Property that specifies the globally unique name to use for this extension library. The name must be unique within Graph Lakehouse. For example: "CSI C++ Extension Functions".
language	Required. Property that specifies the language for the library. The value must be "c++" or "jvm". In this case, the value "c++" is specified.
version	Required. Property that specifies the version of the library in Semantic Versioning format: <i>major.minor.patch</i> . The first digit is required and specifies the major version number. The second digit is optional and specifies the minor version number, and the third digit is optional and specifies the patch number. For example: "1.0.0".
description	Optional. Property that provides a description of the extension library. For example: "An extension library implemented in C++".
author	Optional. Property that specifies the author or company name. For example: "abc@company.com".
copyright	Optional. Property that provides a copyright statement to use for the library. For example: "Copyright © Altair Engineering Inc. All rights reserved."
contents	Required. Array that registers one or more extensions that are included in this library. When registering multiple extension in the same library, repeat the contents array for each extension.

For each extension, the contents array specifies a number of generic properties pertaining to the extension, plus some additional properties based on the UDX type (see [UDX Examples](#)). The following table lists the common properties specified in the contents array for extensions:

Property	Description
name	Required. Property that specifies the globally unique URI used to identify this extension. The URI must be unique in Graph Lakehouse and is referenced in SPARQL queries to invoke this extension. Altair recommends that you use a format such as <code>http://mycompany.com/grouping/etc#function_name</code>. For example: <pre>"http://cambridgesemantics.com/udx/function#concat"</pre>
	Note If you included the error handling azg_throw macro in the extension definition, make sure that this URI value matches the URI in azg_throw.
type	Required. Property that specifies the type of this extension. The value must be "function", "aggregate", or "service".
signature	Required. Property that specifies the globally unique C language identifier for the extension. The value must be udx_ followed by the extension name, as specified as the extern "C" factory extension name used to instantiate the extension. For example, Altair recommends names of the form: <code>"udx_mycompany_mylibraryname_myextension_name"</code>.
arguments	Required. Property that lists the JSON type for each of the extension's input arguments. Specify the JSON type that corresponds to the enum type that you chose for each of the arguments when you implemented the interface. Refer to UDX Data Types for a list of the JSON types. For example: <pre>["double", "String"].</pre>

Property	Description
results	Required. Property that specifies the JSON type for the extension's return value. For example: "double".
description	Optional. Property that provides a description of the extension. For example: "A function that concatenates string values."

Once a UDX is defined and registered, follow the instructions in [Compiling UDX Source Files](#) to create a shared object file for loading to Graph Lakehouse.

Compiling UDX Source Files

After creating and registering one or more new extensions, you need to compile your source files and registration metadata into a single shared object file, or in a JVM environment, build the extension classes into a JAR file you can load into Graph Lakehouse. This topic provides details of tools that are available and how you can compile extensions to create the files needed to run the new extensions.

Note

Documentation on UDX development in JVM environments is in progress and not available at this time.

Compiling UDX files in C++

Graph Lakehouse provides a `azg_extfn_compile` utility that you can use to compile a single CPP file into a shared object. To compile multiple CPP files into a single shared object, you can use the `CMake` utility. See the following procedures for information about configuring **CMake** to compile extensions included in one or more separate CPP files.

UDX CMake Configuration

In the parent `CMakeLists.txt` configuration file, specify the following compiler and compilation flags to use for compiling user-defined extensions:

```
cmake_minimum_required(VERSION 3.13.2)
if (NOT DEFINED ENV{AZG})
message("missing environment variable AZG:")
message("defaulting to AZG=root_anzograph_install_dir")
set(AZG root_anzograph_install_dir)
else()
set(AZG $ENV{AZG})
endif()

set(CMAKE_CXX_COMPILER ${AZG}/tools/bin/x86_64-pc-linux-gnu-g++-7.2.0)
add_compile_options(-O3 -Wall -m64 -std=c++17)
```

For example, the following `CMakeLists.txt` configures **CMake** in a Docker environment:

```

cmake_minimum_required(VERSION 3.13.2)
if (NOT DEFINED ENV{AZG})
message("missing environment variable AZG:")
message("defaulting to AZG=install_dir")
set(AZG install_dir)
else()
set(AZG $ENV{AZG})
endif()

set(CMAKE_CXX_COMPILER ${AZG}/tools/bin/x86_64-pc-linux-gnu-g++-7.2.0)
add_compile_options(-O3 -Wall -m64 -std=c++17)

```

Compiling a UDX to Create a Shared Object File

Run the following command to use the Graph Lakehouse extension compilation utility (`azg_extfn_compile`) to compile a single CPP file and generate the shared object (`.so`) file. The utility invokes the `g++` compiler to compile the C++ file.

```
<install_path>/bin/azg_extfn_compile /path/file_name
```

Note

After compiling a C++ UDX file, you need to copy the the shared object file along with any other files or libraries on which the UDX is dependent, to a common extension directory on your Graph Lakehouse Server (or leader node in a cluster). When executing the UDX specified in a query, Graph Lakehouse copies the UDX, along with any dependent files or libraries, to all the compute nodes or slices in your system for execution on their respective portions of loaded data.

For example, the following command compiles a `divReg.cpp` file and generates `divReg.so`.

```
./opt/anzograph/bin/azg_extfn_compile /home/user/cpp/divReg.cpp
```

After generating the shared object files for new extensions, see [Loading a UDX to the Database](#) for instructions on loading shared object files into Graph Lakehouse, to register new extensions and make them available from within Graph Lakehouse.

Tip

After starting up Graph Lakehouse, you can examine the boot log file to see what extensions have been scanned and registered.

Loading a UDX to the Database

To load new extensions to Graph Lakehouse, you place the UDX C++ `.so` files (or JVM JAR files) in the extension library directory, `<install_path>/lib/udx`, on the leader node and then restart Graph Lakehouse. Each time the database starts, it scans the files in the extension directory and loads the extensions in the registration database.

Using Extensions in SPARQL Queries

Once you've registered an extension library with Graph Lakehouse, you can simply include extension library UDXs within the syntax syntax of regular SPARQL or other types of query statements. For example:

```
select (<http://example/concat>( ?a, ?b) as ?concat) where ...
```

The only requirement for execution of a UDX is that the arguments for input and output parameters match values provided within the query statement. See [UDX Examples](#) for more information on the signatures and specification of input and output parameters for the different types of user-defined extensions supported by Graph Lakehouse.

When executing a query containing a user-defined extension, each slice of an Graph Lakehouse server or cluster deployment creates its own distinct instance of the extension for every occurrence in the query by invoking the "**create**" method of the associated "**ExtensionFactory**". As each row of values streams through a slice, it is passed to the instance by calling its "**apply**" method, and the results are then passed on upward to the consumer of the stream. The factory, and the instances that it creates, are destroyed only when the query has eventually finished executing.

UDX Examples

The topics in this section describe some sample extension source files that may help guide you as you develop your own extensions.

In this section:

User-Defined Function (UDF) Examples	1132
User-Defined Aggregate (UDA) Examples	1138

User-Defined Function (UDF) Examples

A user-defined function (UDF) looks to the user just like a normal function that maps one row of input values to one row of output values. The following code sample comprises a complete, minimal, working example of a UDF extension:

```
struct AND : Function {
    void apply(const Row &a, Row &r) override {
        r.setBoolean(0, a.getBoolean(0) && a.getBoolean(1));
    }
};
*
* signature:
*
extern "C" ExtensionFactory *udx_AND() { return new FactoryFor<AND>(); }
*
* meta-data:
*
{
    "name"          : "http://example/and",
    "signature"     : "udx_AND",
    "type"          : "function",
    "arguments"     : ["boolean", "boolean"],
    "results"       : "boolean",
    "variadic"      : false,
    "description"   : "Returns the logical conjunction of two booleans"
}
*
```

This example also includes meta-data information in the same file as the AND function extension definition, to register the function in Graph Lakehouse. In addition to those attributes common to every extension, the following attributes are specified for user-defined function extensions.

Attribute	Description
arguments	An array of zero or more UDX data types that specifies the number and type of arguments required by any application of the UDF. Determines the shape of the input row passed as the "a" parameter of the <code>apply</code> method.

Attribute	Description
results	An array of zero or more UDX data types that specifies the number and type of results that are returned by any application of the UDF. Determines the shape of the output row passed as the "r" parameter of the <code>apply</code> method.
variadic	A boolean value which indicates, if true, that the final type listed in the arguments array may be repeated one or more times in an application of the UDF. This attribute is optional.

Tip

See [Registering a UDX in an Extension Library](#) for more information on registration information and different methods of registering functions in Graph Lakehouse. Refer to each function's argument or result data types using their Graph Lakehouse registry or UDX data type names. (See [UDX Data Types](#).)

Implementing the Apply() Method

The previous example defines a function named "udx_AND". The **apply()** method specifies input and output row arguments "a" and "r", and then performs a boolean AND operation of two boolean values, returning a Boolean value true (1) if both row arguments are true.

The arguments to the function, "a", are passed as a row whose schema is specified by the "arguments" meta-data attribute. If the UDF is optionally marked as being 'variadic : true', then the "variadic" meta-data may also receive additional values with type "**a.shape().last()**" at the end of the given row. The output parameter, "r", specifies a row to assign the results to, passed as a row whose shape is specified by the "results" meta-data attribute.

The implementation of the function extension also uses the **setBoolean** and **getBoolean** routines to process the values of the input and output arguments and perform the operation of the function. The following table lists the **get** and **set** routines available for Graph Lakehouse extensions. Use the **Read Cell** and/or **Read Cell with Default** get routines for argument a. The **Read Cell with Default** routines are used to return a default value if the cell is not defined (empty). Use the **Write Cell** set routines for argument r:

Read Cell (arg a)	Read Cell with Default (arg a)	Write Cell (arg r)
defined(size_t)	N/A	undefined(size_t)
getBoolean(size_t)	getBoolean(size_t, bool)	setBoolean(size_t, bool)
getByte(size_t)	getByte(size_t, uint8_t)	setByte(size_t, byte)
getShort(size_t)	getShort(size_t, short)	setShort(size_t, short)
getInt(size_t)	getInt(size_t, int)	setInt(size_t, int)
getLong(size_t)	getLong(size_t, long)	setLong(size_t, long)
getFloat(size_t)	getFloat(size_t, float)	setFloat(size_t, float)
getDouble(size_t)	getDouble(size_t, double)	setDouble(size_t, double)
getDate(size_t)	getDate(size_t, Date)	setDate(size_t, Date)
getTime(size_t)	getTime(size_t, Time)	setTime(size_t, Time)
getDateTime(size_t)	getDateTime(size_t, DateTime)	setDateTime(size_t, DateTime)
getDuration(size_t)	getDuration(size_t, Duration)	setDuration(size_t, Duration)
getString(size_t)	getString(size_t, String)	setString(size_t, String)
getLString(size_t)	getLString(size_t, LString)	setLString(size_t, LString)
getUDT(size_t)	getUDT(size_t, UDT)	setUDT(size_t, UDT)

Read Cell (arg a)	Read Cell with Default (arg a)	Write Cell (arg r)
getURI(size_t)	getURI(size_t, URI)	setURI(size_t, URI)
getTag(size_t)	getTag(size_t, String)	N/A
getType(size_t)	N/A	N/A
N/A	N/A	clear()
getBlob(size_t)	getBlob(size_t, Blob)	setBlob(size_t, Blob)

Compilation and Execution of Function Extensions

When compiling a query that includes a function extension (UDF), the leader verifies that the number and type of the arguments passed to the UDF are consistent with its domain, as specified by the **"arguments"** and **"variadic"** attributes of the meta-data description. When executing a query, each slice creates its own distinct instance of the UDF for every occurrence in the query by invoking the **create** method of the associated ExtensionFactory. As each row of values streams through a slice, it is passed to the instance by calling its **apply** method, and the results are then passed on upward to the consumer of the stream. The factory, and the instances that it creates, are destroyed only when the query has eventually finished execution.

Additional UDF Examples

The following example file, **div.cpp**, defines a function named **divide**.

 [Open div.cpp in a separate window](#)

```
#include "udx_api.hpp"           // For extensions API
using namespace std;             // Everything in standard namespace
using namespace udx2;            // Everything in udx namespace

struct divide : Function
{
    void apply(const Row& a, Row& r)
    {
```

```

    if (a.defined(0) && a.defined(1))
    if (auto d = a.getLong(1))
    {
        r.setDouble(0,a.getLong(0) / d);
    }
}
};
extern "C" ExtensionFactory* udx_divide() { return new FactoryFor<divide>(); }

```

The following example file, `udf.cpp`, below defines three UDFs: a **concat** function that concatenates two strings, a **usd_to_eur** function that converts USD values to EUR, and a **sortstr** function that sorts words into a single string.

 [Open udf.cpp in a separate window](#)

```

#include "udx_api.hpp"                                // For extensions API
using namespace std;                                  // Everything in std
using namespace udx2;                                  // Everything in udx

// create a concat function that concatenates two string values

struct concat : Function
{
    void apply(const Row& a,Row& r)
    {
        r.setString(0, string(a.getString(0)) + string(a.getString(1)));
    }
};
extern "C" ExtensionFactory* udx_concat() { return new FactoryFor<concat>(); }

// create a usd_to_euro function that converts USD values to EUR

struct usd_to_euro : Function
{
    void apply(const Row& a,Row& r)
    {
        auto exchange_rate = 0.81;
        auto [udt,tag] = a.getUDT(0);
        if (tag=="$" && !udt.empty())
        {
            istringstream i{string(udt)};
            ostringstream o;o.precision(2);o.setf(ios::fixed);
            double          d;

```



```

        i >> d;
        o << d * exchange_rate ;

        r.setUDT(0,{o.str(),"€"});
    }
}

};

extern "C" ExtensionFactory* udx_usd_to_euro() { return new FactoryFor<usd_to_euro>();}

// create a sort string function that sorts the words into a string

struct sortstr : Function
{
    void apply(const Row& a,Row& r)
    {
        ostringstream          o;
        istringstream          i(string(a.getString(0)));
        istream_iterator<string> b(i);
        vector<string>          tokens(b,istream_iterator<string>());

        sort(begin(tokens),end(tokens));

        copy(begin(tokens),end(tokens),ostream_iterator<string>(o," "));

        r.setString(0,o.str().c_str());
    }
};

extern "C" ExtensionFactory* udx_sortstr() { return new FactoryFor<sortstr>(); }

```

User-Defined Aggregate (UDA) Examples

User Defined Aggregate (UDA) extensions allow developers to extend the set of aggregate functions that are already built into Graph Lakehouse. A UDA maps multiple rows of input values to a single row of output values. For example, the following code snippet comprises a complete, minimal, working example of a UDA:

```
*
* implementation:
*
struct ALL : Aggregate {
    bool all = true;
    void accumulate(const Row &a) override { all &= a.getBoolean(0); }
    void save      (      Row &s) override { s.setBoolean(0, all); }
    void merge     (const Row &s) override { all &= s.getBoolean(0); }
    void result    (      Row &r) override { r.setBoolean(0, all); }
};
*
* signature:
*
extern "C" ExtensionFactory *udx_ALL() { return new FactoryFor<ALL>(); }
*
* meta-data:
*
{
    "name"      : "http://example/all",
    "signature" : "udx_ALL",
    "type"      : "aggregate",
    "arguments" : "boolean",
    "states"    : "boolean",
    "results"   : "boolean",
    "description" : "Returns the logical conjunction of a list of booleans"
}
*
```

The implementation of this example UDA extension uses the `setBoolean` and `getBoolean` routines to process the values of the input and output arguments and perform the operation of the function.

After compiling and registering the UDA in an extension library file, you can use the extension anywhere in a SPARQL query where the syntax allows. For example:

```
select (<http://example/all>( ?a) as ?all) where ...
```

This example also includes meta-data information in the same file as the `udx_ALL` function extension, to register the function in Graph Lakehouse. In addition to the attributes common to every extension, aggregate extensions specify the following additional meta-data attributes.

Attribute	Description
arguments	An array of zero or more types that specifies the number and type of arguments required by any application of the UDA. Determines the shape of the input row passed as the "a" parameter of the accumulate method.
results	An array of zero or more types that specifies the number and type of results that are returned by any application of the UDA. Determines the shape of the output row passed as the "r" parameter of the result method.
sorted	A boolean value which, if true, indicates that the algorithm being implemented by the aggregate is sensitive to the order in which values are supplied to the <code>accumulate()</code> method. If the aggregate is computed in a subquery that has an <code>ORDER BY</code> clause, the compiler normally discards it. If you specify <code>sorted</code> as true, the <code>ORDER BY</code> clause is preserved.
variadic	A boolean value which, if true, indicates that the final type listed in the arguments array may be repeated one or more times in an application of the UDA. This attribute is optional.
states	An array of zero or more types that specifies the number and type of states that are marshaled across the cluster when merging those intermediate results that accumulated on the slices during the accumulation phase. Determines the shape of the output row passed as the "s" parameter of the save method and determines the shape of the input row passed as the "s" parameter of the merge method. This attribute is optional.

When compiling a query containing an aggregate extension, the Graph Lakehouse server or the leader node of a cluster verifies that the number and type of the arguments passed to the UDA are consistent with its domain, as specified by the **arguments** and **variadic** attributes of the meta-data description.

Aggregation Methods

Since Graph Lakehouse takes advantage of parallel processing, Graph Lakehouse performs aggregation in multiple steps. Each of the data slices operate on groups of values that are local to that particular slice and produce one result for each group. The results of those computations are stored in variables called aggregation states, and the aggregation state values are used to calculate the final result. Each UDA implements the following aggregation methods: `accumulate()`, `save()`, `merge()`, and `result()`. The UDA implements the `save()` method to transmit these partially aggregate local results or states to the leader, where they are "merged" in to a fresh instance of the UDA that will compute the final result.

Method	Description
accumulate (const Row &a)	Passes a row of values whose schema is specified by the "arguments" meta-data attribute. UDAs marked "variadic" may also receive additional values with type " <code>a.shape().last()</code> " at the end of the given row.
save(Row &s)	Assigns the accumulated intermediate results to the given row, which will then be transmitted across the cluster to the target slice and merged in to the recipient instance of the UDA..
merge(const Row &s)	Merges the intermediate results accumulated by another instance over on a remote slice with the internal state. The intermediate results of the remote slice instance are passed as a row whose shape is specified by the "states" meta-data attribute.
result(Row &r)	Assigns the results of the aggregate to the given row, whose shape is specified by the " results " meta-data attribute.

When executing a query, each slice creates its own distinct instance of the UDA for every occurrence in the query by invoking the `create` method of the associated `ExtensionFactory`.

- As each row of values streams through a slice, it is passed to the instance by calling its `accumulate` method, which responds by updating its internal state as necessary to record having processed the row in some appropriate way.
- When all rows on the slice have been accumulated, the instance is now given a mutable row (of shape `states`) into which it serializes any intermediate results it has accumulated, and the instance is then destroyed.
- The slice receiving the result now creates an instance of the UDA and a row (of shape `states`), and the system arranges for all intermediate states to be transmitted across the cluster and "merged" into the instance by passing each in turn to its `merge` method.

The factory, and the instances that it creates, are destroyed only when the query has eventually finished execution.

The Accumulate() Method

The `accumulate()` method `void accumulate(const Row& r)` accumulates values from the UDA's input arguments into the aggregation states. In the function, `r` is the input received by the UDA, and its type is `Row`. Include the appropriate **get** routines to define the [UDX Data Types](#). The following table lists the available **get** routines. The **Read Cell with Default** routines are used to return a default value if the cell is not defined (empty).

Read Cell	Read Cell with Default
<code>defined(size_t)</code>	N/A
<code>getBoolean(size_t)</code>	<code>getBoolean(size_t, bool)</code>
<code>getByte(size_t)</code>	<code>getByte(size_t, uint8_t)</code>
<code>getShort(size_t)</code>	<code>getShort(size_t, short)</code>

Read Cell	Read Cell with Default
getInt(size_t)	getInt(size_t, int)
getLong(size_t)	getLong(size_t, long)
getFloat(size_t)	getFloat(size_t, float)
getDouble(size_t)	getDouble(size_t, double)
getDate(size_t)	getDate(size_t, Date)
getTime(size_t)	getTime(size_t, Time)
getDateTime(size_t)	getDateTime(size_t, DateTime)
getDuration(size_t)	getDuration(size_t, Duration)
getString(size_t)	getString(size_t, String)
getLString(size_t)	getLString(size_t, LString)
getUDT(size_t)	getUDT(size_t, UDT)
getURI(size_t)	getURI(size_t, URI)
getTag(size_t)	getTag(size_t, String)
getType(size_t)	N/A
getBlob(size_t)	getBlob(size_t, Blob)

For example, the following `accumulate()` definition for arithmetic mean fetches the input long variable using `getLong()` and adds it into the summation. It also increments the count on each received input:

```
void accumulate(const Row& r) {
    m_sum += r.getLong(0);
    m_cnt += 1;
}
```

The Save() Method

The `save()` method (`void save(Row& r) { }`) saves each of the internal aggregation states into one of the UDX data types. In the function, `r` is the Row in which the accumulated result is saved so that it can be restored in the `merge()` function. Include the appropriate **get** or **set** routines to save the states from the `accumulate()` function. The following table lists the **get** and **set** routines available.

Save Internal State	Restore Internal State	Restore Internal State with Default Value
<code>setBoolean(size_t, bool)</code>	<code>getBoolean(size_t)</code>	<code>getBoolean(size_t, bool)</code>
<code>setShort(size_t, short)</code>	<code>getShort(size_t)</code>	<code>getShort(size_t, short)</code>
<code>setInt(size_t, int)</code>	<code>getInt(size_t)</code>	<code>getInt(size_t, int)</code>
<code>setLong(size_t, long)</code>	<code>getLong(size_t)</code>	<code>getLong(size_t, long)</code>
<code>setFloat(size_t, float)</code>	<code>getFloat(size_t)</code>	<code>getFloat(size_t, float)</code>
<code>setDouble(size_t, double)</code>	<code>getDouble(size_t)</code>	<code>getDouble(size_t, double)</code>
<code>setDate(size_t, Date)</code>	<code>getDate(size_t)</code>	<code>getDate(size_t, Date)</code>

Save Internal State	Restore Internal State	Restore Internal State with Default Value
setTime(size_t, Time)	getTime(size_t)	getTime(size_t, Time)
setDateTime(size_t, DateTime)	getDateTime(size_t)	getDateTime(size_t, DateTime)
setDuration(size_t, Duration)	getDuration(size_t)	getDuration(size_t, Duration)
setString(size_t, String)	getString(size_t)	getString(size_t, String)
setLString(size_t, LString)	getLString(size_t)	getLString(size_t, LString)
setUDT(size_t, UDT)	getUDT(size_t)	getUDT(size_t, UDT)
setURI(size_t, URI)	getURI(size_t)	getURI(size_t, URI)
setBlob(size_t, Blob)	getBlob(size_t)	getBlob(size_t, Blob)

For example, the following `save()` definition for the arithmetic mean **accumulate()** example above uses set routines to save the summation as a Long value into first cell of the row and total count as a Long value in the second cell of the row:

```
void save(Row& r) {
    r.setLong(0,m_sum)
    .setLong(1,m_cnt);
}
```


Note

If the data structure of the internal aggregation state to save is more complex (such as a map or vector) than the fixed data types in the routines above, serialize the type in the **save()** step. It can then be de-serialized in the **merge()** step to get the original data. See the [discEntropy.cpp](#) example below for a sample UDA that employs the map data structure.

The Merge() Method

The **merge()** method (`void merge(const Row& r) { }`) merges all of the internal aggregation states from all of the data slices with the leader slice. In the function, **r** is the row with the previously computed result, which can be retrieved in the same sequence they are saved in

For example, in the following **merge()** definition for the arithmetic mean example above, the leader gets the summation of other slices using the received input of the first cell of the row and total count as the second cell of the row:

```
void merge(const Row& r) {  
    m_sum += r.getLong(0);  
    m_cnt += r.getLong(1);  
}
```

The Result() Method

The **result()** method (`void result(Row& r) { }`) returns the final computation of the aggregation states. In the function, "**r**" is the row where the computed result is returned by setting the values for the cells with the appropriate set routines:

- `undefined(size_t)`
- `setBoolean(size_t, bool)`
- `setByte(size_t, byte)`
- `setShort(size_t, short)`
- `setInt(size_t, int)`
- `setLong(size_t, long)`
- `setFloat(size_t, float)`

- `setDouble(size_t, double)`
- `setDate(size_t, Date)`
- `setTime(size_t, Time)`
- `setDateTime(size_t, DateTime)`
- `setDuration(size_t, Duration)`
- `setString(size_t, String)`
- `setLString(size_t, LString)`
- `setUDT(size_t, UDT)`
- `setURI(size_t, URI)`
- `setBlob(size_t, Blob)`
- `clear()`

For example:

```
void result(Row& r) {
    if (m_cnt)
        r.setDouble(0, double(m_sum) / m_cnt);
}
```

Error Handling

In case of an error, an extension can signal an exception by including the **azg_throw** macro, which is similar to the `printf()` function in C. Create and specify as a string the full URI for the aggregate, and include the message to display when an error occurs.

```
azg_throw("aggregate_URI", "error_message");
```

Where *aggregate_URI* is a prefix that you define, followed by the aggregate name. The URI must be globally unique. Altair recommends that you use a format such as

`http://mycompany.com/grouping/etc#aggregate_name`. For example:

```
azg_throw("http://cambridgesemantics.com/udx/aggregate#mean", "Error message - code %d", m_code);
```

Instantiating Extensions

Instantiate the extension by specifying the following extern "C" factory functions:

```
extern "C" ExtensionFactory* udx_aggregateName() { return new FactoryFor<aggregateName>(); }
```

Where *aggregateName* is the short name of the function, not the full URI. For example:

```
extern "C" ExtensionFactory* udx_mean() { return new FactoryFor<mean>(); }
```

Once the aggregate is defined, see [Registering a UDX in an Extension Library](#) and [Compiling UDX Source Files](#).

Additional Aggregate Examples

mean.cpp

The following example file, mean.cpp, defines an aggregate extension named average.

 [Open mean.cpp in a separate window](#)

```
#include "udx_api.hpp"
using namespace std;
using namespace udx2;

struct average : Aggregate {
    long m_sum = 0;
    long m_cnt = 0;

    void accumulate(const Row& r) {
        m_sum += r.getLong(0);
        m_cnt += 1;
    }

    void save(Row& r) {
        r.setLong(0,m_sum)
        .setLong(1,m_cnt);
    }

    void merge(const Row& r) {
        m_sum += r.getLong(0);
        m_cnt += r.getLong(1);
    }
}
```

```

}

void result(Row& r) {
    if (m_cnt)
        r.setDouble(0, double(m_sum)/m_cnt);
    }
};

extern "C" ExtensionFactory* udx_average() { return new FactoryFor<average>(); }

```

discEntropy.cpp

The following example UDA definition, `discEntropy.cpp`, computes discrete entropy and uses `map` as an internal aggregation state.

 [Open discEntropy.cpp in a separate window](#)

```

#include <cmath>
#include "udx_api.hpp"
#include <sstream>
#include <iostream>
#include <map>

using namespace udx2;

struct DiscEntropy : Aggregate {
    long m_cnt = 0; // Total events
    std::map<std::string, long> m_map;

void accumulate(const Row& r) {
    // NOTE: Supply input as string even for numbers using SPARQL expression
    if(!r.getString(0).empty()) {
        m_map[std::string(r.getString(0).begin(), r.getString(0).end())]++;
        m_cnt += 1;
    }
}

void save(Row& r) {
    if(m_cnt != 0) {
        std::ostringstream out;
        for(const auto& [key, val] : m_map)
            out << key.size() << '~' << key << val << '~';
        r.setLong(0, m_cnt)
        .setString(1, out.str());
    }
}

```

```

    }
}

void merge(const Row& r) {
    if(r.defined(0)) {
        std::istringstream in(r.getString(1).data());
        std::string key;
        long val;
        int len;
        char delimiter;
        while(in.good()) {
            key.clear();
            val = len = 0;
            in >> len >> delimiter;
            if(in && len) {
                std::vector<char> tmp(len);
                in.read(tmp.data(), len);
                key.assign(tmp.data(), len);
                in >> val >> delimiter;
                m_map[key] += val;
            }
        }
        m_cnt += r.getLong(0);
    }
}

void result(Row& r) {
    if(m_cnt == 0)
        azg_throw("http://example/aggregates#discentropy", ": insufficient data");
    // Formula: Entropy = -Sum[p(x) * log(p(x))] for all classes of x events.
    double entropy = 0;
    double prob = 0;
    for(const auto& elem : m_map) {
        prob = double(elem.second) / m_cnt;
        entropy += prob * std::log2(prob);
    }
    r.setDouble(0, (entropy >= 0.0) ? entropy : -1 * entropy);
}

};

extern "C" ExtensionFactory* udx_csi_statistics_discentropy() { return new
FactoryFor<DiscEntropy>(); }

```

FAQ & Troubleshooting

This section includes answers to frequently asked questions, an error message reference, information on retrieving Graph Lakehouse diagnostic files, and how to get support.

In this section:

- [FAQ](#) 1151
- [Error Message Reference](#) 1160
- [Retrieving Diagnostic Files](#) 1162
- [Getting Support](#) 1168

FAQ

This topic provides answers to frequently asked questions and includes references to more detailed information. The questions are categorized by subject:

- [Container Images FAQ](#)
- [Graph Lakehouse FAQ](#)
- [SPARQL FAQ](#)

Container Images FAQ

This section includes answers for questions related to Graph Lakehouse container images.

- [Why are there three Graph Lakehouse container images?](#)

Why are there three Graph Lakehouse container images?

To offer versatility for different types of environments and deployment preferences, Altair provides three Graph Lakehouse container images. The list below describes each image and its purpose:

- **anzograph (all-in-one image):** The all-in-one image (<https://hub.docker.com/r/cambridgesemantics/anzograph>) includes the front end (user interface) as well the back end (database) in one image.
- **anzograph-frontend (user interface):** The front end image (<https://hub.docker.com/r/cambridgesemantics/anzograph-frontend>) includes the user interface only. Multiple users can deploy the front end locally and use it to access a central Graph Lakehouse database cluster.
- **anzograph-db (back end/database):** The back end image (<https://hub.docker.com/r/cambridgesemantics/anzograph-db>) includes the database only. If you have existing client applications to use with Graph Lakehouse and do not need the front end, you can deploy the database by itself.

Graph Lakehouse FAQ

This section includes answers for questions related to Graph Lakehouse usage.

- [How do I determine what size cluster to deploy?](#)
- [Are there best practices around performance benchmarking with Graph Lakehouse?](#)
- [How do I deploy Graph Lakehouse SPARQL endpoints?](#)
- [How do I use the SPARQL and RDF Graph Store endpoints?](#)
- [How do I enable SPARQL HTTP protocol?](#)
- [How do I reset the admin password?](#)
- [How do I access the Graph Lakehouse file system with Docker?](#)
- [How do I copy load files from the host to the Graph Lakehouse file system in Docker?](#)
- [How do I customize a Helm-managed Graph Lakehouse deployment?](#)
- [How do I enable database persistence?](#)
- [What RDF load file types does Graph Lakehouse support?](#)
- [How do I set up my load files to get the best load performance?](#)
- [How do I get a list of all Graph Lakehouse functions?](#)

How do I determine what size cluster to deploy?

For guidance on determining the number of instances to include in your cluster and choosing the most suitable instance type, see the best practice [Sizing Guidelines for In-Memory Storage](#).

Are there best practices around performance benchmarking with Graph Lakehouse?

For best practices and in-depth information about benchmarking with Graph Lakehouse, see the [AnzoGraph DB Benchmarking Guide](#).

How do I deploy Graph Lakehouse SPARQL endpoints?

Graph Lakehouse supports the standard W3C SPARQL 1.1 Protocol (SPARQL endpoint) and SPARQL 1.1 Graph Store HTTP Protocol (RDF Graph Store endpoint). The SPARQL endpoint and Graph Store endpoint are both enabled by default. And both endpoints can be accessed through the front end (user interface) or the back end (database).

- If you have the front end client deployed, the endpoints are enabled and can be accessed by applications that have access to the front end server. User authentication is required to access endpoints through the front end.
- Back end endpoints are also enabled by default but are controlled by the [enable_sparql_protocol](#) configuration setting. If SPARQL protocol is disabled for your deployment, the database endpoints will not be accessible. See [How do I enable SPARQL HTTP protocol?](#) for instructions on enabling SPARQL protocol. The back end endpoints do not support user authentication at this time.

For more information about the endpoints, see [Access the SPARQL and RDF Endpoints](#).

How do I use the SPARQL and RDF Graph Store endpoints?

Graph Lakehouse endpoints conform to the W3C SPARQL 1.1 standards and can be accessed like other standard SPARQL endpoints. For usage information and details about the Graph Lakehouse endpoints, see [Access the SPARQL and RDF Endpoints](#).

For W3C specifications on SPARQL endpoints, see [SPARQL 1.1 Protocol](#). For RDF graph store specifications, see [SPARQL 1.1 Graph Store HTTP Protocol](#).

How do I enable SPARQL HTTP protocol?

If HTTP protocol is disabled for your deployment and you want to enable it so that you can use the Graph Lakehouse CLI or post queries to the SPARQL HTTP port (7070 by default), follow the instructions below.

1. Stop Graph Lakehouse. You can stop the database from the Admin user interface, or see [Start and Stop Graph Lakehouse](#) for information about alternate methods.

2. Open the Graph Lakehouse settings file `/install_path/config/settings.conf` in an editor.

[How do I access the Graph Lakehouse file system with Docker?](#) For example:

```
vi /opt/anzograph/config/settings.conf
```

3. Find the **enable_sparql_protocol** setting. If the setting is listed, change the value from "false" to **true**. If the setting is not listed, add a new line to the end of the file and enter the following value:

```
enable_sparql_protocol=true
```

4. Save and close the file, and then restart Graph Lakehouse.

Once Graph Lakehouse starts, SPARQL HTTP protocol is enabled on the [sparql_protocol_port](#) (7070 by default) and you can use the CLI (AZGI) or send requests through the back end (database) SPARQL endpoints. For more information about Graph Lakehouse endpoints, see [Access the SPARQL and RDF Endpoints](#).

How do I reset the admin password?

To reset the Graph Lakehouse admin password, SSH to the Graph Lakehouse host server (the leader node if this is a cluster) and run the following command. This command runs the `azgpasswd` utility in the `anzograph/bin` directory and updates the password (**passwd**) file in the `anzograph/config` directory:

```
./<install_path>/bin/azgpasswd /<install_path>/config/passwd -u admin -p <password>
```

For example, the following command resets the password to "Passw0rd1":

```
./opt/csi/anzograph/bin/azgpasswd /opt/csi/anzograph/config/passwd -u admin -p  
Passw0rd1
```

Important

Do not use certain special characters like \$ or * in passwords. Those characters have special meaning for bash.

How do I access the Graph Lakehouse file system with Docker?

Run the following Docker command to access the Graph Lakehouse file system, the **/opt/anzograph** directory:

```
sudo docker exec -it <container_name> /bin/bash
```

Where **<container_name>** is the name of the Graph Lakehouse container whose file system you want to access. For example:

```
sudo docker exec -it anzograph /bin/bash
```

How do I copy load files from the host to the Graph Lakehouse file system in Docker?

1. Run the following Docker command to access the Graph Lakehouse file system, the **/opt/anzograph** directory:

```
sudo docker exec -it <container_name> /bin/bash
```

Where **<container_name>** is the name of the Graph Lakehouse container whose file system you want to access. For example:

```
sudo docker exec -it anzograph /bin/bash
```

2. Determine where on the file system you would like to place the load files and create a new directory if necessary. If you plan to load a directory of files, remember to include the file type in the directory name. See [RDF Load File Requirements](#) for more information. For example:

```
mkdir /opt/anzograph/load-files.ttl
```

3. Type `exit` to exit the container.
4. Run the following Docker command to copy files from the host server to a location in the Graph Lakehouse container.

```
sudo docker cp /<path>/<filename> <container_name>:/<path>/<directory>
```

For example:

```
sudo docker cp /home/user/sales.ttl anzograph:/opt/anzograph/load-files.ttl/
```

Or this command copies a directory to the container:

```
sudo docker cp -r /<path>/<directory> <container_name>:/<path>
```

For example:

```
sudo docker cp -r /home/user/load-files.ttl anzograph:/opt/anzograph/
```

How do I customize a Helm-managed Graph Lakehouse deployment?

To customize a Helm-managed deployment, modify the Graph Lakehouse Helm chart, **values.yaml**, and then deploy Graph Lakehouse using that chart.

- The **values.yaml** file is in the **HELM_HOME** directory. To view the location of **HELM_HOME**, you can run `helm home`.
- To download the latest version of the Helm chart from `csi-helm/anzograph`, you can run `helm repo update`.
- You can edit **values.yaml** directly or make a copy and edit the copy. When you run the `helm install` command to deploy Graph Lakehouse, specify the name of the Helm chart to use for that deployment.
- For details about the Helm chart options, view the readme, **Readme.md**, in the **HELM_HOME** directory.

For instructions on deploying Graph Lakehouse with Helm, see [Deploy Graph Lakehouse with Helm](#).

How do I enable database persistence?

For most installations Graph Lakehouse is configured by default to save the data in memory to disk after every transaction. Each time Graph Lakehouse is restarted, the persisted data is automatically loaded back into memory. To check whether the save to disk option is enabled, open the settings

file, `install_path/config/settings.conf`, and find the `enable_persistence` option. If `enable_persistence=true`, data persistence is enabled. If `enable_persistence=false`, persistence is disabled. For instructions on changing settings, see [Change System Settings](#).

Important Considerations

- In general, each Graph Lakehouse server needs access to about twice as much disk space as RAM on the server. By default, Graph Lakehouse saves data to the `install_path/persistence` directory on the local file system. You can also configure Graph Lakehouse to save data to a different location by changing the value of the [persistence_directory](#) setting in `settings.conf`.
- When persistence is enabled, transactional workloads that perform many concurrent write operations may experience a performance degradation due to the overhead of writing the data from each transaction to disk.

What RDF load file types does Graph Lakehouse support?

- Turtle (.ttl file type): Terse RDF Triple Language that writes an RDF graph in compact form.
- N-Triple (.n3 and .nt file types): A subset of Turtle known as simple triples.
- N-Quad (.nq and .quads file types): N-Triples with a blank node or graph designation.
- TriG (.trig file type): An extension of Turtle that supports representing a complete RDF data set.
- JSON-LD (.jsonld file type): A method of encoding linked data using JSON. JSON-LD files are supported for loading via the IO services. JSON-LD is not supported by SPARQL LOAD queries.

For more information, see [Load RDF Data from Files](#).

How do I set up my load files to get the best load performance?

When you have multiple files, Graph Lakehouse loads the files in parallel, using all available cores on all servers in the cluster. While you can load files stored on the leader node's local file system, for optimal performance, it is important to use a shared file system to ensure that all servers in the

cluster have access to the files. In a Docker or Kubernetes container environment, the storage system should also be shared with the container file system.

For more information and details about load file requirements, see [RDF Load File Requirements](#).

How do I get a list of all Graph Lakehouse functions?

You can run the following query to return a list of supported SPARQL functions. The query returns all of the function names as well as the supported argument and return types for each function:

```
SELECT ?extension_name ?extension_arguments ?extension_results
WHERE { TABLE 'stc_functions' }
ORDER BY ?extension_name
```

SPARQL FAQ

This section includes answers for questions related to the SPARQL query language.

- [What extensions to the SPARQL standard does Graph Lakehouse provide?](#)
- [Where can I find more information about SPARQL?](#)

What extensions to the SPARQL standard does Graph Lakehouse provide?

Graph Lakehouse implements the standard SPARQL forms and functions described in the W3C [SPARQL 1.1 Query Language](#) specification. In addition to supporting the standard functions, Graph Lakehouse also provides several SQL-like and Microsoft Excel-like functions as well as support for more advanced operations like window aggregates, advanced grouping sets, and graph algorithms. In addition to the built-in standard and advanced functions, Graph Lakehouse includes extension libraries that provide several data science, geospatial, Apache Arrow, and various utility functions. For information, see [SPARQL Query Language Reference](#).

Where can I find more information about SPARQL?

For basic information about SPARQL, the semantic web, or RDF, see the Altair [Semantic University](#). In addition, the [Semantic Web for the Working Ontologist](#) focuses on SPARQL and RDF usage using Internet examples.

To view the W3C formal specification and definitive reference, see the [SPARQL 1.1 Query Language](#) specification.

For additional information about best practices and tips, see [SPARQL Best Practices](#) and [SPARQL Tips and Tricks](#).

Error Message Reference

This section includes the possible causes and solutions for Graph Lakehouse error messages. Click a message in the list below to view details about that error:

- [Cannot execute as user 'root'. To override this security protection, set 'enable_root_user=true': Invalid user id](#)
- [std::exception - locale::facet::_S_create_c_locale name not valid](#)
- [stg: Cannot allocate memory - heap is exhausted](#)
- [Unsupported functionality: loading files with this extension - <URL>](#)
- [Invalid Certificate](#)
- [Fatal Error. Caught Signal 15](#)

Cannot execute as user 'root'. To override this security protection, set 'enable_root_user=true': Invalid user id

This message indicates that you tried to start Graph Lakehouse as the root user and root access is disabled. You can try the command again as a non-root user, or you can enable root access by adding `enable_root_user=true` to the Graph Lakehouse settings file. For instructions, see [Change System Settings](#).

std::exception - locale::facet::_S_create_c_locale name not valid

This message indicates that your operating system does not include the `glibc-langpack` dependency and it needs to be installed. To resolve the issue, follow the steps below:

1. Run the following command to install the dependency:

```
dnf install glibc-langpack-en
```

2. Set the locale for your server. For example, running the following command sets it to `en_US`:

```
localectl set-locale LANG=en_US.UTF-8
```


3. Restart the `azgmgrd` and `anzograph` services.

stg: Cannot allocate memory - heap is exhausted

This message indicates that all of the memory that is available to Graph Lakehouse is in use and there is not enough left to run queries. The solution is to free memory by restarting Graph Lakehouse. Then either adjust your workload to reduce the amount of data that is loaded or increase the amount of RAM on the host server(s).

Unsupported functionality: loading files with this extension - <URL>

This message indicates that the load file directory listed in the error message does not include the load file type extension in its name. For example, running a query that loads files from the URL `s3://csi-sdl-data-tickit/tickit` results in this error. Since the tickit directory includes `.ttl.gz` files, renaming `tickit` to `tickit.ttl.gz` resolves the error. For more information about load requirements, see [RDF Load File Requirements](#).

Invalid Certificate

This message indicates that you replaced the default Graph Lakehouse certificates with your own trusted certificates and the certificates are invalid. Certificates can be invalid because they expired or they were generated or signed incorrectly. For information about replacing certificates, see [Replace the Default Self-Signed Certificates with Trusted Certificates](#).

Fatal Error. Caught Signal 15

This error indicates that a process external to Graph Lakehouse stopped the Graph Lakehouse processes, such as if the host machine was shut down while Graph Lakehouse was running. Restart Graph Lakehouse to proceed with normal usage.

Retrieving Diagnostic Files

When Altair Support requests Graph Lakehouse diagnostic files for troubleshooting an issue, you can quickly retrieve the files from either the Diagnostics tab in the Admin Console or by using the system management CLI. This topic describes the diagnostic files and provides instructions for retrieving the files from the user interface and the command line.

- [Diagnostic File Overview](#)
- [Retrieving Files from the Admin Console](#)
- [Retrieving Files from the Command Line](#)

Diagnostic File Overview

There are two types of diagnostic files:

- **Xray:** Xrays are generated on-demand. If you encounter an error and the database remains running, you generate an Xray to produce the diagnostic files.
- **Crash Dump:** If you encounter an error that crashes the database, Graph Lakehouse automatically generates a crash file that contains diagnostic information about the crash.

Xrays and crash dumps harvest the diagnostic data that is stored in Graph Lakehouse's system tables. They include information such as:

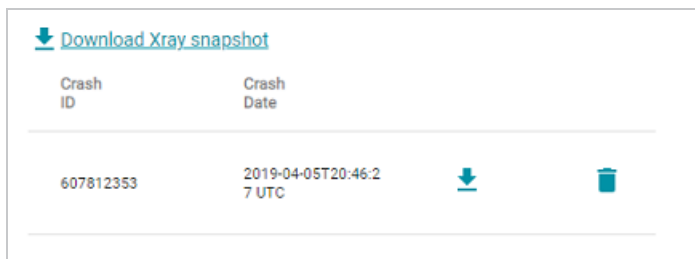
- A low level, de-identified log of the requests that were sent to the database.
- Statistics like query operation step execution times, number of rows processed, and amount of memory used.
- Detailed but de-identified trace information for errors that were encountered.
- Configuration information such as the number of nodes in the cluster and Graph Lakehouse system settings values.

Xrays and crash dumps are designed to be anonymous and can be safely shared with Altair Support. They do NOT capture user information or any of the data that is loaded into memory by a user, nor do they expose details that could be used to reveal the nature of the data being queried. They are valuable tools that enable Altair to diagnose and fix issues without access or any other visibility into a customer's data or database. They can also be used to report on overall and detailed system performance, resulting in improved query performance for future releases of Graph Lakehouse.

Retrieving Files from the Admin Console

Follow the instructions below to download an xray or crash dump from the user interface.

1. Log in to Admin Console and click the **Diagnostics** tab. The console displays the available options. For example:



Download Xray snapshot			
Crash ID	Crash Date		
607812353	2019-04-05T20:46:27 UTC		

2. If you want to retrieve an xray, click the **Download Xray snapshot** link. Graph Lakehouse creates the xray and produces a tarball with an .xray extension. The console downloads the .xray tarball to your computer.

Note

The files in the tarball are compressed. Do not compress the .xray file before sending it to Altair.

3. If you want to retrieve a crash dump, click the **Refresh** button to refresh the list of available crash dump .xray files. Click the file name that you want to download. The console downloads the .xray file to your computer.

Note

The files in the tarball are compressed. Do not compress the .xray file before sending it to Altair.

Retrieving Files from the Command Line

Follow the instructions below retrieve diagnostic files using the system management CLI, azgctl. The CLI is in the <install_path>/bin directory.

- [Taking an X-Ray](#)
- [Generating a Crash Dump](#)

Taking an X-Ray

Run the following command on the leader server to take an x-ray on a running database. The result is a tarball that includes historical system records from the specified time period. All flags for specifying a time period are optional. If you omit the options, the resulting x-ray will include the last 24 hours of historical system data.

Note

The system manager interprets time specifications using the system's local time and converts the timestamps to UTC when starting the x-ray.

```
azgctl -xray /path/name.xray [ -f <time> ] [ -t <time> ] [ -d <num_days> ]  
[ -h <num_hours> ] [ -m <num_minutes> ]
```

Option	Description
/path/name	The path on the file system where you want to save the tarball and the name of the tarball. All x-rays must be named with the .xray extension.
-f <time>	The -f <time> (or --from <time>) option can be used to specify the time to start the system data capture, i.e., omit all of the records from before the specified time. Time must be specified in the following format: YYYY-MM-DD [:HH

Option	Description
	[:MM]] . For example, <code>-f 2024-01-10:15:00</code> sets the start time to 3:00 p.m. (local system time) on January 10, 2024.
-t <time>	The <code>-t <time></code> (or <code>--to <time></code>) option can be used to specify the time to end the system data capture, i.e., omit all of the records after the specified time. Time must be specified in the following format: <code>YYYY-MM-DD [:HH [:MM]]</code> . For example, <code>-t 2024-01-09:19:30</code> sets the end time to 7:30 p.m. (local system time) on January 9, 2024.
-d <num_days>	The <code>-d <num_days></code> (or <code>--days <num_days></code>) option can be used to specify the number of days to include in the x-ray. The value must be a positive integer. - When combined with <code>-t <time></code> or <code>-f <time></code>, the number of days is relative to the <code>from</code> or <code>to</code> value. For example, <code>-f 2024-01-10 -d 2</code> means two days starting from 1/10/24 (i.e., 1/10/24 – 1/12/24). And <code>-t 2024-01-10 -d 2</code> is two days before 1/10/24 (i.e., 1/8/24 – 1/10/24). - When included without <code>-f</code> or <code>-t</code>, the number of days is relative to the current local system time. For example, <code>-d 2</code> captures the last 2 days of data starting from <code>now()</code>.
-h <num_hours>	The <code>-h <num_hours></code> (or <code>--hours <num_hours></code>) option can be used to specify the number of hours to include in the x-ray. The value must be a positive integer. - When combined with <code>-t <time></code> or <code>-f <time></code>, the number of hours is relative to the <code>from</code> or <code>to</code> value. For example, <code>-f 2024-01-10:12:00 -h 5</code> means the 5 hours after 12:00 p.m. on 1/10/24. And <code>-t 2024-01-10:12:00 -h 5</code> means the 5 hours before 12:00 p.m. on 1/10/24. - When included without <code>-f</code> or <code>-t</code>, the number of hours is relative to the current local system time. For example, <code>-h 3</code> captures the last 3 hours of data starting from <code>now()</code>.

Option	Description
-m <num_minutes>	The <code>-m <num_minutes></code> (or <code>--minutes <num_minutes></code>) option can be used to specify the number of minutes to include in the x-ray. The value must be a positive integer. - When combined with <code>-t <time></code> or <code>-f <time></code>, the number of minutes is relative to the <code>from</code> or <code>to</code> value. For example, <code>-f 2024-01-10:12:00 -m 30</code> means the 30 minutes after 12:00 p.m. on 1/10/24. And <code>-t 2024-01-10:12:00 -m 30</code> means the 30 minutes before 12:00 p.m. on 1/10/24. - When included without <code>-f</code> or <code>-t</code>, the number of minutes is relative to the current local system time. For example, <code>-m 30</code> captures the last 30 minutes of data starting from <code>now()</code>.

Examples

The following example generates an x-ray that includes the last 24 hours of system data. A tarball named `24hr_errors.xray` is written to the `/tmp` directory.

```
/opt/altair/anzograph/bin/azgctl -xray /tmp/24hr_errors.xray
```

The example below captures the last 12 hours worth of data. A tarball named `last12hours.xray` is written to the `/tmp` directory.

```
/opt/altair/anzograph/bin/azgctl -xray /tmp/last12hours.xray -h 12
```

The example below captures the last two days of data from before 5:00 p.m. on 1/18/24. A tarball named `1-16_to_1-18.xray` is written to the `/opt/shared/xrays` directory.

```
/opt/altair/anzograph/bin/azgctl -xray /opt/shared/xrays/1-16_to_1-18.xray -t 2024-01-18:17:00 -d 2
```

Generating a Crash Dump

If you encounter an issue that stops the database, Graph Lakehouse automatically generates diagnostic files. Follow the instructions below to retrieve the files after a crash.

Note

The database does not need to be running to collect the crash dump.

1. Run the following command on the leader server to view a list of the available crash dumps.

```
azgctl -crashlist
```

For example:

```
/opt/altair/anzograph/bin/azgctl -crashlist
```

The results show a list of available crash dumps by timestamp. For example:

Crash ID	Time
520460982	2023-12-28 20:30:35
520457655	2023-12-28 19:01:25

2. Run the following command to retrieve the appropriate crash files. This command creates a tarball that includes the diagnostic files:

```
azgctl -crashfetch [ crash_id ] /path/name.xray
```

Include the `crash_id` when you want to retrieve a specific crash dump that is listed in the crash list. Omit the crash ID to retrieve the latest files. All crash dump tarballs must include the `.xray` extension.

Examples

The following command captures the most recent crash files. The tarball is named `latest_crash.xray` and it is saved to the `/tmp` directory.

```
/opt/altair/anzograph/bin/azgctl -crashfetch /tmp/latest_crash.xray
```

The example below captures the crash dump for ID 520457655:

```
/opt/altair/anzograph/bin/azgctl -crashfetch 520457655 /tmp/crash_520457655.xray
```

Tip

You can run `azgctl -crashtoss` to remove all crash dumps from the server.

Getting Support

If you have comments or questions about Graph Lakehouse or run into an issue that you need help resolving, you can get help from the Graph Lakehouse user community or from [Altair Support](#). Altair provides the following options for getting Graph Lakehouse technical support:

- Send an email to **dasupport@altair.com**. Emails are automatically submitted to the Graph Lakehouse Service Desk and received by the Altair Data Analytics Support team.
- Get help from [Altair Community](#).

Tip

For answers to frequently asked questions, see the [FAQ](#). For instructions on getting diagnostic files for support, see [Retrieving Diagnostic Files](#).