



Neo4j Aura overview

Table of Contents

Neo4j AuraDB	2
Neo4j AuraDS	3
=Neo4j Aura=	4
Creating an account	5
Aura with cloud provider marketplaces	6
AuraDB Virtual Dedicated Cloud, AuraDS Enterprise, and Aura Business Critical	6
Aura Professional	6
Security	9
Secure connections	9
Single Sign-On (SSO)	19
Encryption	21
User management	26
Projects	26
Users	26
APOC support	29
apoc	29
apoc.agg	29
apoc.algo	30
apoc.any	31
apoc.atomic	31
apoc.bitwise	32
apoc.coll	32
apoc.convert	36
apoc.create	38
apoc.cypher	39
apoc.data	40
apoc.date	40
apoc.diff	41
apoc.do	42
apoc.example	42
apoc.export	42
apoc.graph	44
apoc.hashing	44
apoc.import	45
apoc.json	45
apoc.label	45
apoc.load	45
apoc.lock	46

apoc.map	46
apoc.math	48
apoc.merge	49
apoc.meta	50
apoc.neighbors	50
apoc.node	51
apoc.nodes	52
apoc.number	53
apoc.path	54
apoc.periodic	55
apoc.refactor	56
apoc.rel	57
apoc.schema	58
apoc.scoring	58
apoc.search	59
apoc.spatial	59
apoc.stats	60
apoc.temporal	60
apoc.text	60
apoc.util	64
apoc.warmup	65
apoc.xml	65
Customer Metrics Integration (CMI)	66
Process overview	66
Detailed steps	66
Security	68
Metric scrape interval	70
Example using Prometheus	70
Example using Datadog	71
Programmatic support	72
Metrics granularity	73
Metric definitions	74
Logging	81
Request and download logs	81
Security log forwarding	84
Query log analyzer	85
Neo4j connectors	89
Neo4j Connector for Apache Spark	89
Neo4j Connector for Apache Kafka	89
Neo4j Connector for BI	89
Aura API	91

Overview	91
Authentication.....	92
API Specification	96
Consumption report.....	97
Monitor consumption in real-time	97
Filters	97
=Neo4j AuraDB=.....	99
Neo4j AuraDB overview.....	100
Plans	100
Updates and upgrades.....	100
Support	100
Getting Started	101
Creating an instance	101
Connecting to an instance	105
Querying an instance	109
Importing	110
Importing data	110
Importing an existing database.....	111
Managing instances.....	113
Instance actions	113
Backup, export and restore	117
Secondaries.....	119
Monitoring.....	120
Advanced metrics	121
Connecting applications.....	124
Change Data Capture.....	124
=Neo4j AuraDS=.....	125
Neo4j AuraDS overview.....	126
Plans	126
Updates and upgrades.....	126
Support	126
Architecture.....	127
Neo4j Graph Data Science concepts.....	127
Graph data flow.....	127
Creating an AuraDS instance	129
Connecting to AuraDS	130
Connecting with Neo4j applications	130
Connecting with Python	133
Usage examples	138
Projecting graphs and using the graph catalog.....	138
Executing the different algorithm modes	148

Estimating memory usage and resizing an instance	159
Monitoring the progress of a running algorithm	167
Persisting and sharing machine learning models	176
Loading and streaming back data with Apache Arrow	191
Importing data	194
Importing an existing database	194
Using Neo4j Data Importer	195
Loading CSV files	196
Managing instances	205
Monitoring	205
Advanced metrics	205
Backup, export, and restore	207
Instance actions	208
=Tutorials=	212
Upgrade and migration	213
Upgrade to Neo4j 5 within Aura	213
Migrate from self-managed Neo4j to Aura	216
Integrating with Neo4j Connectors	221
Using the Neo4j Connector for Apache Spark	221
Using the Neo4j BI Connector	222
Improving Cypher performance	226
Cypher statements with literal values	226
Review queries and model	226
Index specification	227
Review metrics and instance size	227
Consider concurrency	227
Runtime engine and Cypher version	227
Network and the cost of the round-trip	228
Troubleshooting	229
Query performance	229
Neo4j Admin database upload errors	230
Driver integration	232
Create an AuraDB instance in the terminal	233
Preparation	233
Obtain a bearer token	233
Obtain the project ID	234
Configure an AuraDB instance	234

Neo4j Aura is a fast, scalable, always-on, fully automated graph platform offered as a cloud service.

Aura includes AuraDB, the graph database as a service for developers building intelligent applications, and AuraDS, the graph data science as a service for data scientists building predictive models and analytics workflows.

Neo4j AuraDB

Neo4j AuraDB is the fully managed graph database as a service that helps build intelligent, context-driven applications faster with lightning-fast queries, real-time insights, built-in developer tools, data visualization, and integrations supported by the largest graph developer community.

For more information on AuraDB, see the [Neo4j AuraDB overview](#).

Neo4j AuraDS

Neo4j AuraDS is the fully managed data science as a service solution for data scientists that unifies the machine learning (ML) surface and graph database into a single workspace, making it easy to uncover the connections in big data and answer business-critical questions.

For more information on AuraDS, see the [Neo4j AuraDS overview](#).

© 2024 License: [Creative Commons 4.0](#)

<https://raw.githubusercontent.com/neo4j-graphacademy/courses/main/asciidoc/courses/neo4j-fundamentals/promo.adoc>

=Neo4j Aura=

Creating an account

To access Neo4j Aura, you need to have an Aura account.

To create an Aura account:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Enter an email address and password and select **Register**, or select **Continue with Google** to use a Google account. If entering an email address and password, follow these additional steps:
 - a. Verify your email address.
 - b. Select **Go to the dashboard** from the Aura Console.
3. Select **I agree** once you have read the **Terms of Service** and **Privacy Policy**.

Aura with cloud provider marketplaces

AuraDB Virtual Dedicated Cloud, AuraDS Enterprise, and Aura Business Critical

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#) [AuraDB Business Critical](#)

You can purchase AuraDB Virtual Dedicated Cloud, AuraDS Enterprise, and AuraDB Business Critical via private offer through the following cloud provider marketplaces:

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

[Contact us](#) to discuss private offers.

Aura Professional

[AuraDB Professional](#) [AuraDS Professional](#)

You can purchase Neo4j Aura Professional on a pay-as-you-go basis through the following cloud provider marketplaces:

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

Purchasing Neo4j Aura Professional through a cloud provider marketplace gives you access to integrated billing and usage reporting in your chosen cloud provider's console.

AWS

1. Purchase the service

To get started, visit the [Neo4j Aura Professional AWS Marketplace page](#) and select **View Purchase options**.

From here you will need to select the **Neo4j Aura Professional Contract** option, decide if you would like to auto-renew your contract when it ends, and then select **Create contract and Pay now**.



While you are shown a \$0 yearly contract option, pricing is pay-as-you-go based on usage and not a fixed subscription service.

2. Set up your account

To start using Neo4j Aura, select [Click here to set up your account](#) to be directed to the Aura Console.

If you are not already logged in to the Aura Console, you will be taken to the Neo4j Aura login/sign-up page. From here you can either log in with an existing Neo4j Aura account or create a new one.



You do not need to use the same email address for your Neo4j Aura account as your AWS account.

If you are creating a Neo4j Aura account for the first time, you will need to confirm your email address and accept the Neo4j Aura Terms of Service before you can access the Aura Console.

Once logged in to the Aura Console, the AWS Marketplace order will be confirmed, and your account will automatically default to the AWS Marketplace tenant.

GCP

1. Purchase the service

To get started, visit the [Neo4j Aura GCP Marketplace page](#) and click **Purchase**.

Note that pricing is pay-as-you-go based on usage and you won't be billed until you create an instance.



Purchasing from the GCP Marketplace requires the **Billing Account Administrator** (`roles/billing.admin`) role as highlighted in the [Overview of Cloud Billing access control](#) Google Cloud documentation.

2. Choose a project

If you purchase the service at the top level of your GCP account, you'll need to choose a target project. You will only need to purchase Neo4j Aura for GCP once, as you can then enable it on a project-by-project basis. However, you still need to choose a target project when you first purchase the service.

3. Enable the service

Once you have purchased the service and switched to a target project, select **Enable** to activate Neo4j Aura for GCP on your project.



Enabling a service through GCP requires the `roles/serviceusage.serviceUsageAdmin` role as highlighted in the [Access Control with IAM](#) Google Cloud documentation.

Once you have enabled the service, you will be directed to the **Neo4j Aura for GCP API** page. This page displays your service details and billing information. When this is first set up, you should have no billing history.

4. Complete the set up

<!-- vale Vale.Terms = NO --> <!-- vale Neo4j.ParaNewLine = NO -->

To start using Neo4j Aura, select **MANAGE VIA NEO4J, INC.** to be directed to the Aura Console.



When you click "MANAGE VIA NEO4J, INC.", you will be alerted that "You're leaving Google". When you click **Confirm**, if the Aura Console fails to open you may need to address any popup blockers in your browser and try again.

<!-- vale Vale.Terms = YES --> <!-- vale Neo4j.ParaNewLine = YES -->

For security purposes, Neo4j and GCP do not share your login credentials. You will need to log in to the Neo4j Console with the same Google account you have used on GCP.

Once logged in, you'll be asked to accept the Neo4j Aura Terms of Service, before being directed to the Neo4j Console and placed within your GCP Marketplace tenant.

From here, you are ready to create a Neo4j Aura instance hosted on GCP and usage will be billed directly to your GCP billing account.

Azure

1. Purchase the service

To get started, visit the [Neo4j Aura Professional Azure Marketplace page](#) and select **Get It Now**.

2. Sign in to your Microsoft Azure Marketplace account

3. Subscribe to Neo4j Aura Professional

- Select the resource group that the Aura Professional subscription will apply to. Then, create a name for the SaaS subscription so you can easily identify it later.
- Your billing term will be a 1-month subscription at \$0 cost. Aura Professional has a consumption based pricing model, so you will only be charged for the amount you consume in Gigabyte hours (Gb/h)
- Set recurring billing to **On**
- Click **Review + subscribe**



- Ensure your Azure account is upgraded before continuing.
- Enable **marketplace purchases** in Azure. See more info on the [Azure website](#)

Security

Secure connections

VPC isolation

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

AuraDB Virtual Dedicated Cloud and AuraDS Enterprise run in a dedicated cloud Account (AWS), Subscription (Azure) or Project (GCP) to achieve complete isolation for your deployment.

Additional VPC boundaries enable you to operate within an isolated section of the service, where your processing, networking, and storage are further protected.

The Aura Console runs in a separate VPC, separate from the rest of Aura.

Network access

An Aura instance can be publicly available, completely private, or both. To configure this, you need to be authorized to access the part of the infrastructure that runs and handles these instances as well as the networking used to establish secure connections between the database and the application's VPC. This includes the ability to connect over the cloud provider's private link and private endpoint.

If your Aura instances are public, traffic to them is allowed to traverse the public internet and they are accessible with the correct username and password.

For your instance to be completely private, turn public traffic off, use the cloud provider's network, and create a private endpoint inside your VPC, which gives you a private connection to Aura. The only way to connect to your database is from inside your network (your VPC in your AWS/Azure/GCP account) using an internal IP address you choose and DNS records you create.

To select network access settings go to [Aura Console > Security > Network Access](#).

Private endpoints

Private endpoints are network interfaces inside your own VPC, which can only be accessed within your private network. The cloud provider connects them over their network to Neo4j Aura. By design they are not exposed to the public internet, ensuring that critical services are accessible only through private, secure networks.

A single private link connection applies to all instances in a region. So if you've set one up for `us-east-1` then those network connections will apply to all instances in that region. You can set up a second private link connection to applications that are hosted in a second region (for example `us-west-1`) but still housed inside the same Aura project.

AWS private endpoints

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

AuraDB Virtual Dedicated Cloud and AuraDS Enterprise support private endpoints on AWS using [AWS PrivateLink](#).

Once activated, you can create an endpoint in your VPC that connects to Aura.

For a step-by-step guide, see the [How to Configure Neo4j Aura With AWS PrivateLink](#) blog article.

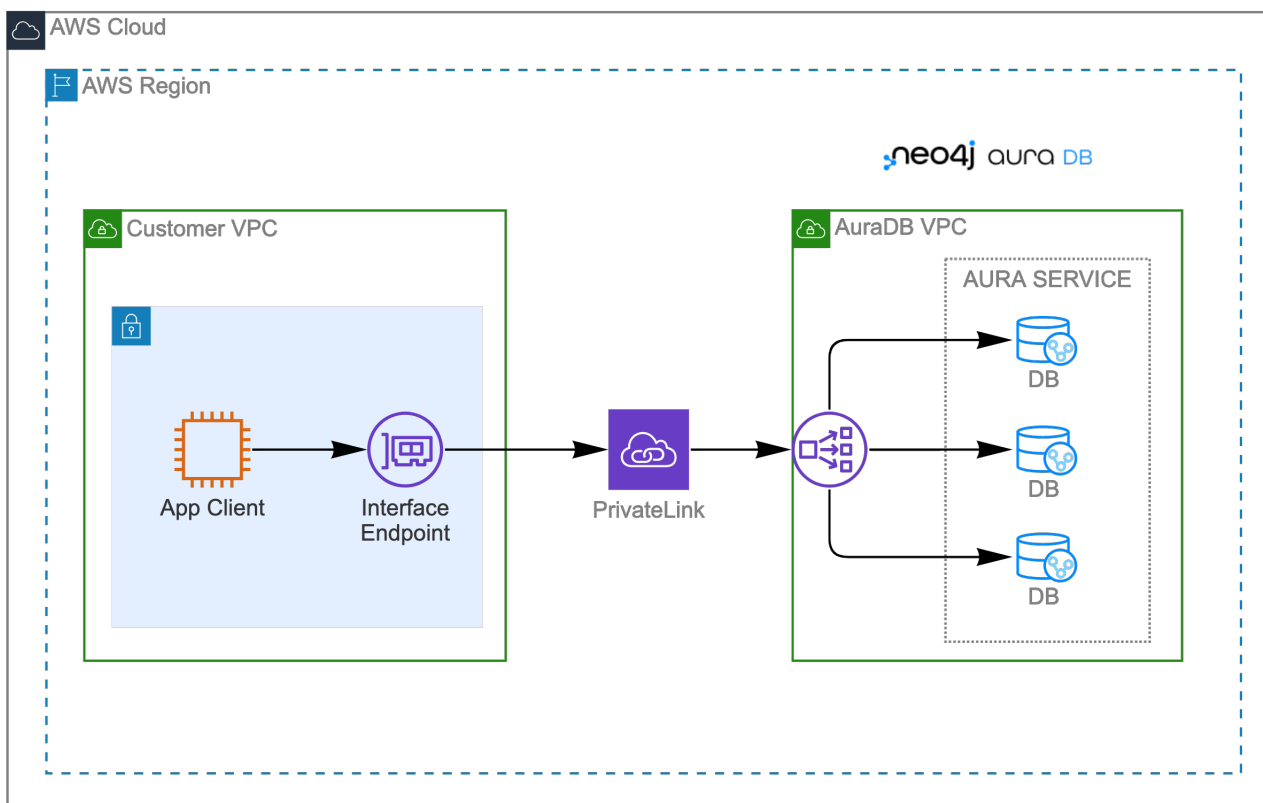


Figure 1. VPC connectivity with AWS PrivateLink

All applications running Neo4j workloads inside the VPC are routed directly to your isolated environment in Aura without traversing the public internet. You can then disable public traffic, ensuring all traffic to the instance remains private to your VPC.



- PrivateLink applies to all instances in the region.
- When activated, a **Private Connection** label, shield icon, and dedicated **Private URI** will appear on any instance tile using PrivateLink in the Aura Console.
- If you disable public traffic, you must use a dedicated VPN to connect to your instance via Browser or Bloom.
- Connections using private endpoints are one-way. Aura VPCs can't initiate connections back to your VPCs.
- In AWS region us-east-1, we do not support the Availability Zone with ID use1-az3 for private endpoints.

Browser and Bloom access over private endpoints

To connect to your instance via Browser or Bloom, you must use a dedicated VPN. This is because when you disable public access to your instance, this applies to all connections, including those from your computer when using Browser or Bloom.

Without private endpoints, you access Browser and Bloom over the internet:

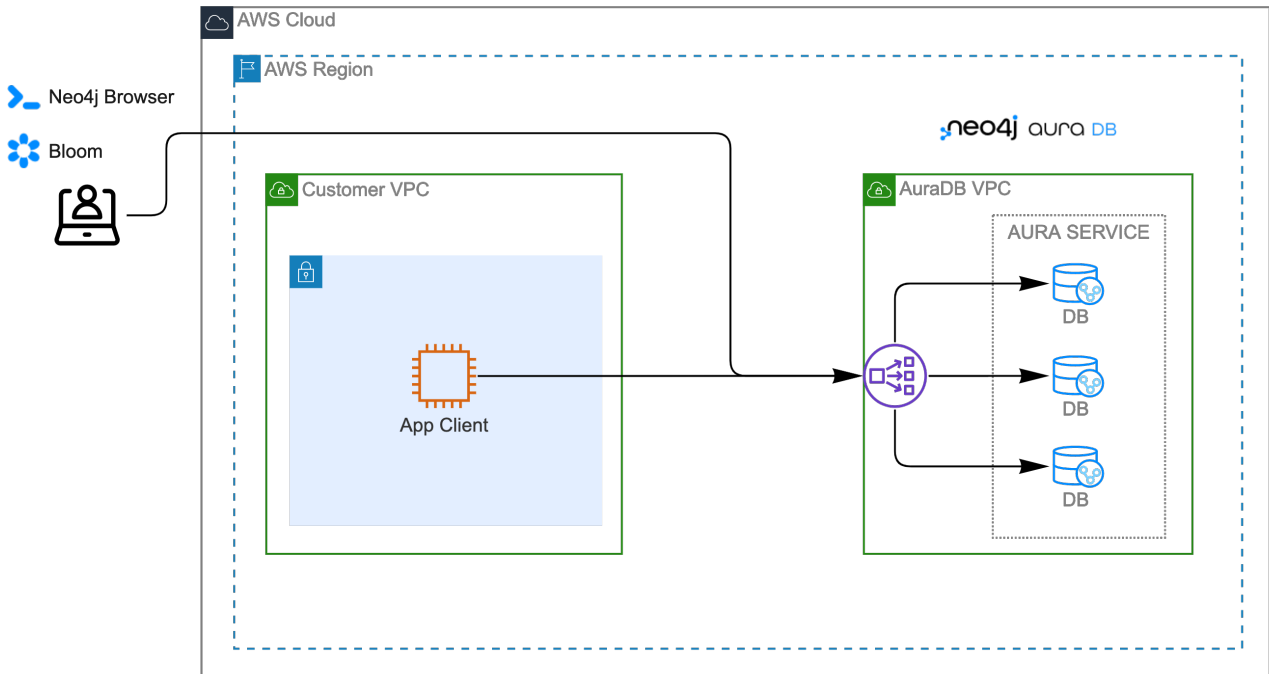


Figure 2. Architecture overview before enabling private endpoints

When you have enabled private endpoints and disabled public internet access, you can no longer connect Browser or Bloom to your instances over the internet:

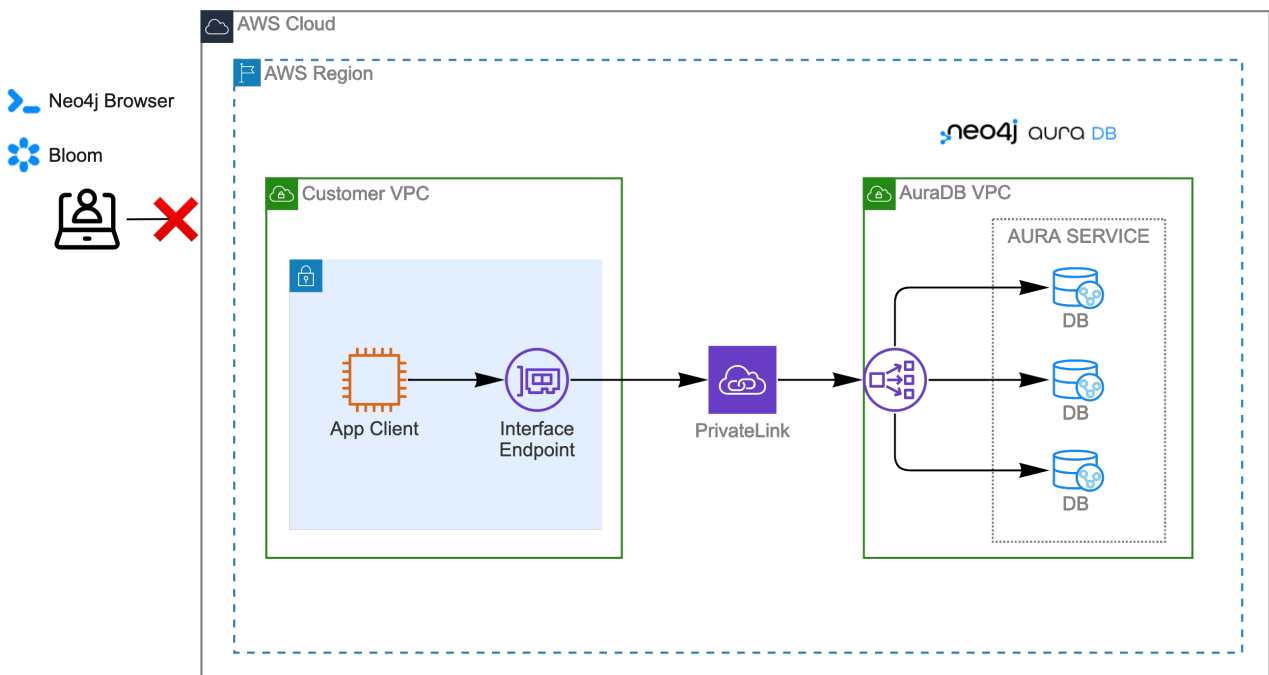



Figure 3. Architecture overview with private endpoints enabled and public traffic disabled

To continue accessing Browser and Bloom, you can configure a VPN (Virtual Private Network) in your VPC and connect to Browser and Bloom over the VPN.

 To access Bloom and Browser over a VPN, you must ensure that:

- The VPN server uses the [VPC's DNS servers](#).
- You use the **Private URI** shown on the instance tile and in the instance details. It will be different from the **Connection URI** you used before.

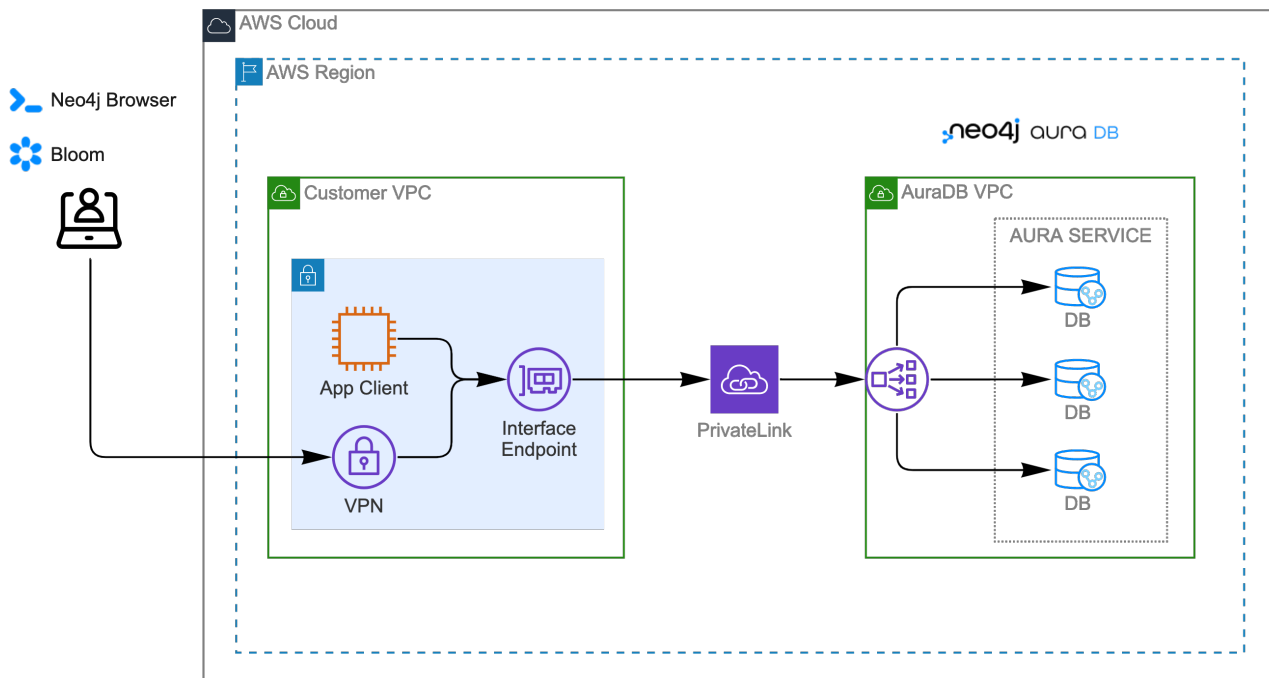


Figure 4. Accessing Browser and Bloom over a VPN

Enabling private endpoints

To enable private endpoints using AWS PrivateLink:

1. Select **Network Access** from the sidebar menu of the Console.
2. Select **New network access configuration** and follow the setup instructions.

You will need an AWS account with permissions to create, modify, describe and delete endpoints. Please see the [AWS Documentation](#) for more information.

GCP private endpoints

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

AuraDB Virtual Dedicated Cloud and AuraDS Enterprise support private endpoints on GCP using [GCP Private Service Connect](#).

Once activated, you can create an endpoint in your VPC that connects to Aura.

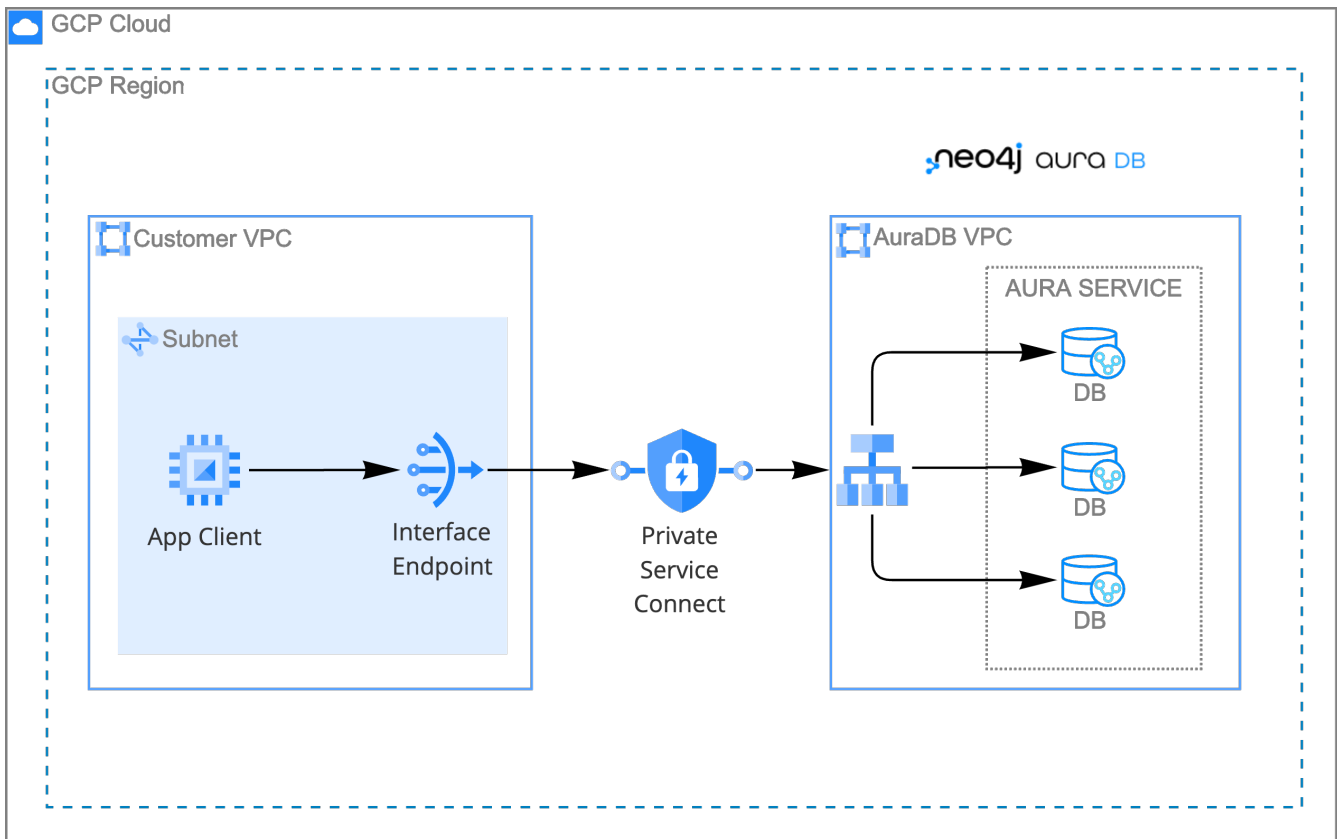


Figure 5. VPC connectivity with GCP Private Service Connect

All applications running Neo4j workloads inside the VPC are routed directly to your isolated environment in Aura without traversing the public internet. You can then disable public traffic, ensuring all traffic to the instance remains private to your VPC.



- Private Service Connect applies to all instances in the region.
- When activated, a **Private Connection** label, shield icon, and dedicated **Private URI** will appear on any instance tile using Private Service Connect in the Aura Console.
- If you disable public traffic, you must use a dedicated VPN to connect to your instance via Browser or Bloom.
- Connections using private endpoints are one-way. Aura VPCs can't initiate connections back to your VPCs.

Browser and Bloom access over private endpoints

To connect to your instance via Browser or Bloom, you must use a dedicated VPN. This is because when you disable public access to your instance, this applies to all connections, including those from your computer when using Browser or Bloom.

Without private endpoints, you access Browser and Bloom over the internet:

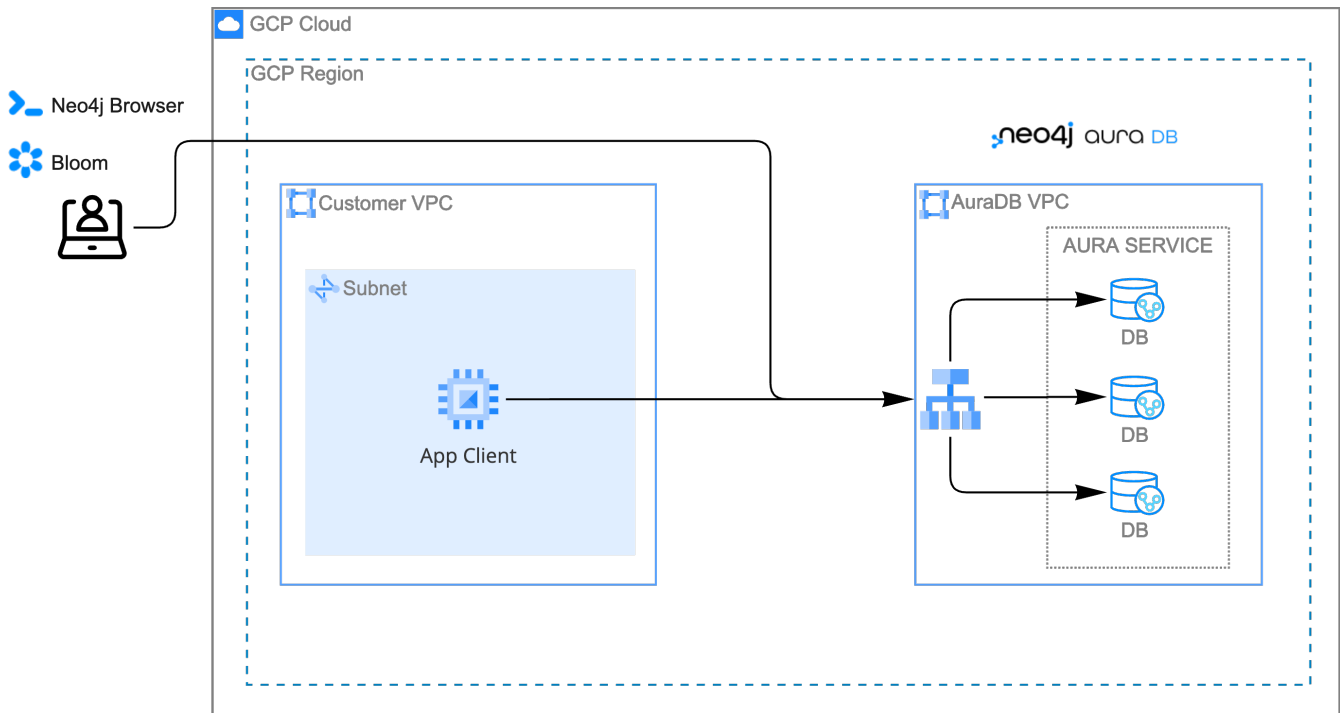


Figure 6. Architecture overview before enabling private endpoints

When you have enabled private endpoints and disabled public internet access, you can no longer connect Browser or Bloom to your instances over the internet:

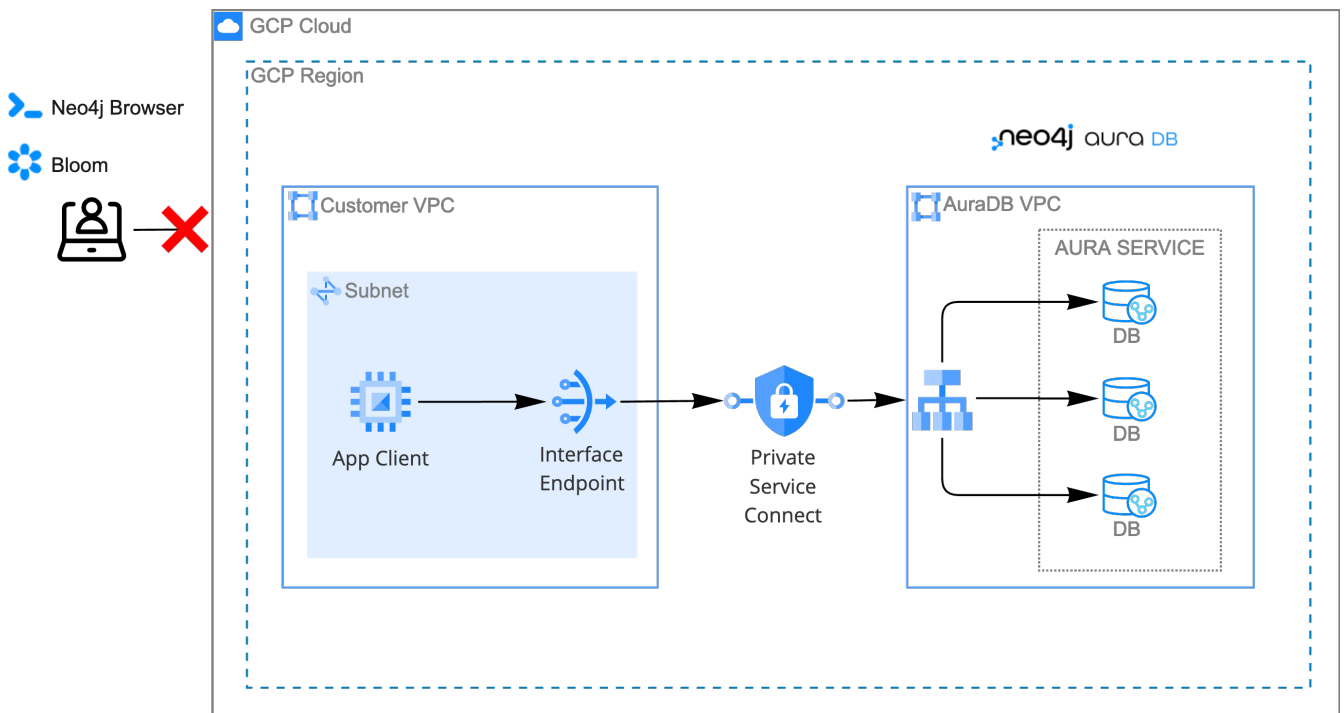


Figure 7. Architecture overview with private endpoints enabled and public traffic disabled

To continue accessing Browser and Bloom, you can configure a [GCP Cloud VPN](#) (Virtual Private Network) in your VPC and connect to Browser and Bloom over the VPN.



To access Bloom and Browser over a VPN, you must ensure that:

- You have set up [GCP Response Policy Zone](#), or an equivalent DNS service, inside of the VPC.
- You use the **Private URI** shown on the instance tile and in the instance details. It will be different from the **Connection URI** you used before.

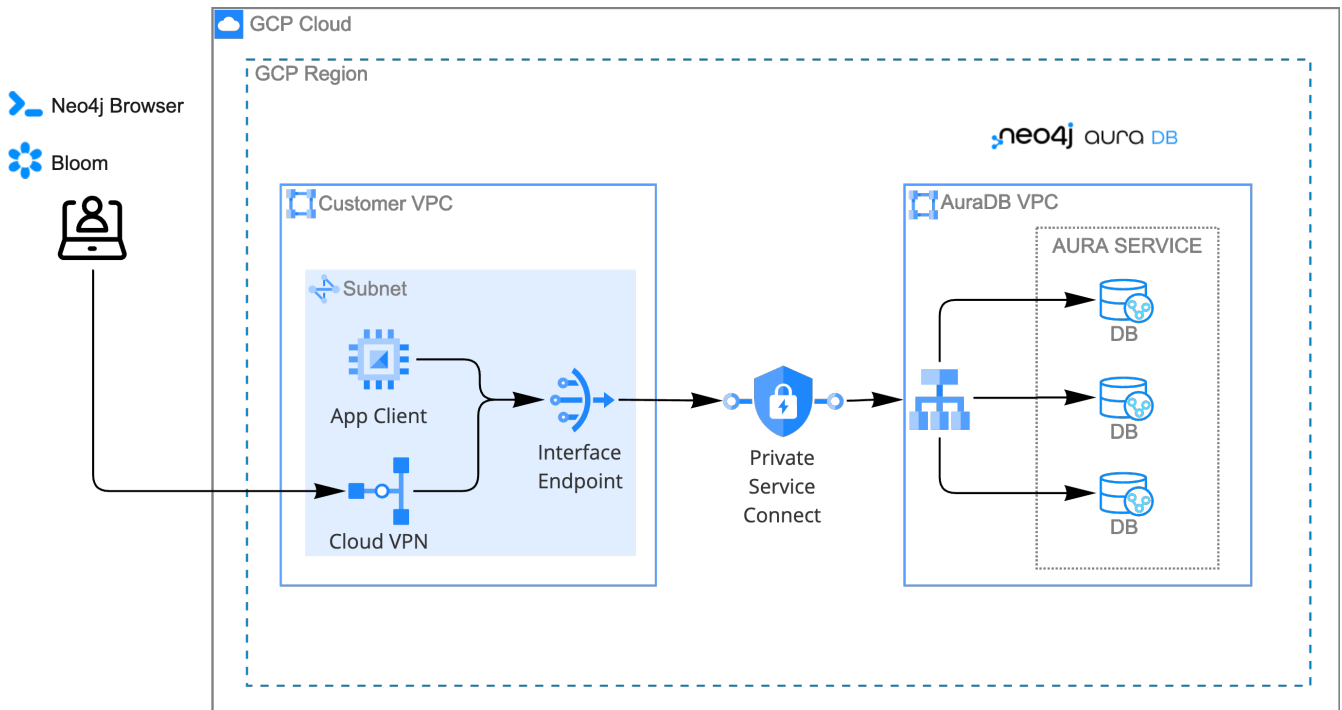


Figure 8. Accessing Browser and Bloom over a VPN

Enabling private endpoints

To enable private endpoints using GCP Private Service Connect:

1. Select **Network Access** from the sidebar menu of the Console.
2. Select **New network access configuration** and follow the setup instructions.

Please see the [GCP Documentation](#) for required roles and permissions.

Azure private endpoints

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

AuraDB Virtual Dedicated Cloud and AuraDS Enterprise support private endpoints on Azure using [Azure Private Link](#).

Once activated, you can create an endpoint in your Virtual Network (VNet) that connects to Aura.

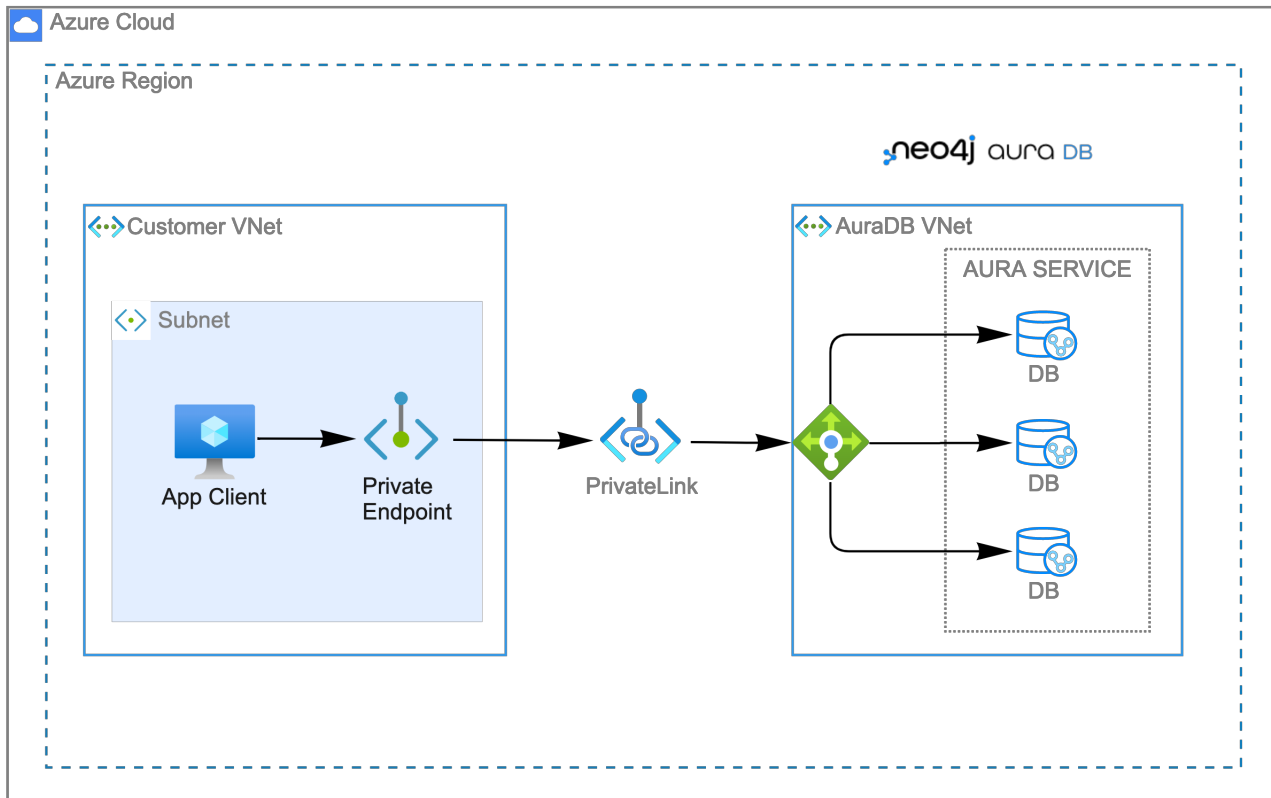


Figure 9. VNet connectivity with Azure Private Link

All applications running Neo4j workloads inside the VNet are routed directly to your isolated environment in Aura without traversing the public internet. You can then disable public traffic, ensuring all traffic to the instance remains private to your VNet.



- Private Link applies to all instances in the region.
- When activated, a **Private Connection** label, shield icon, and dedicated **Private URI** will appear on any instance tile using Private Link in the Aura Console.
- If you disable public traffic, you must use a dedicated VPN to connect to your instance via Browser or Bloom.
- Connections using private endpoints are one-way. Aura VNets can't initiate connections back to your VNets.

Browser and Bloom access over private endpoints

To connect to your instance via Browser or Bloom, you must use a dedicated VPN. This is because when you disable public access to your instance, this applies to all connections, including those from your computer when using Browser or Bloom.

Without private endpoints, you access Browser and Bloom over the internet:

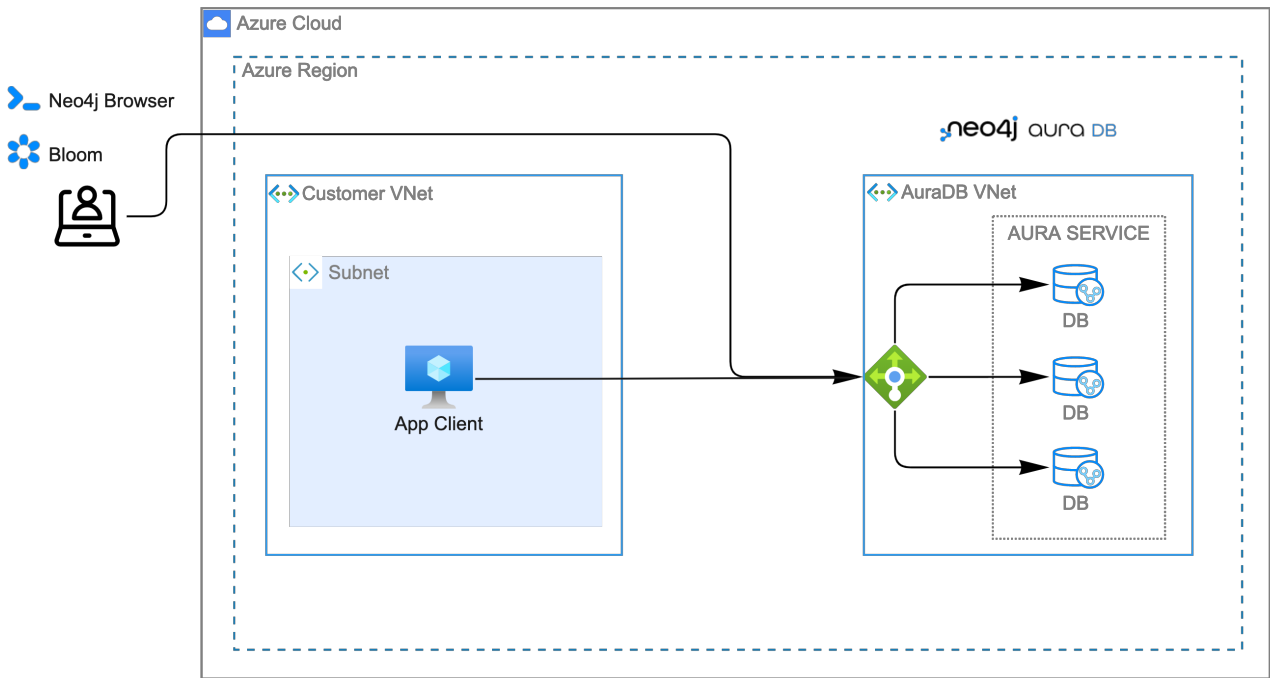


Figure 10. Architecture overview before enabling private endpoints

When you have enabled private endpoints and disabled public internet access, you can no longer connect Browser or Bloom to your instances over the internet:

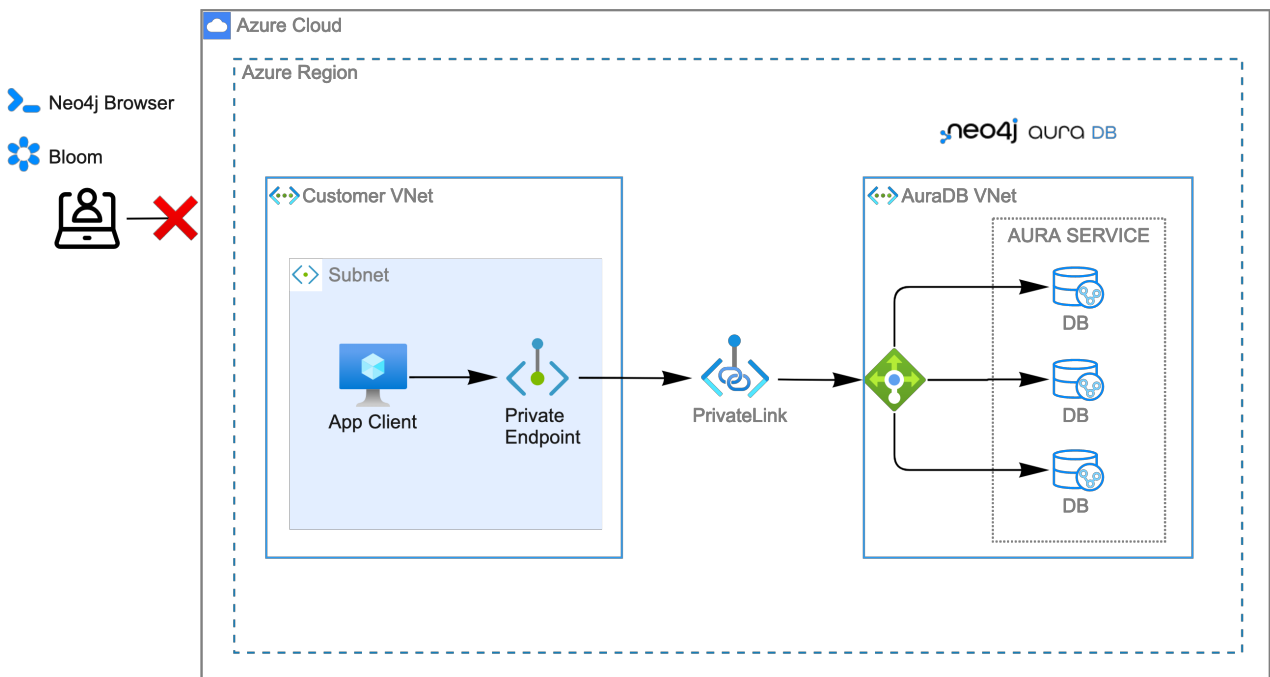


Figure 11. Architecture overview with private endpoints enabled and public traffic disabled

To continue accessing Browser and Bloom, you can configure a VPN (Virtual Private Network) in your VNet and connect to Browser and Bloom over the VPN.



To access Bloom and Browser over a VPN, you must ensure that:

- You have setup [Azure Private DNS](#), or an equivalent DNS service, inside of the VNet.
- You use the **Private URI** shown on the instance tile and in the instance details. It will be different from the **Connection URI** you used before.

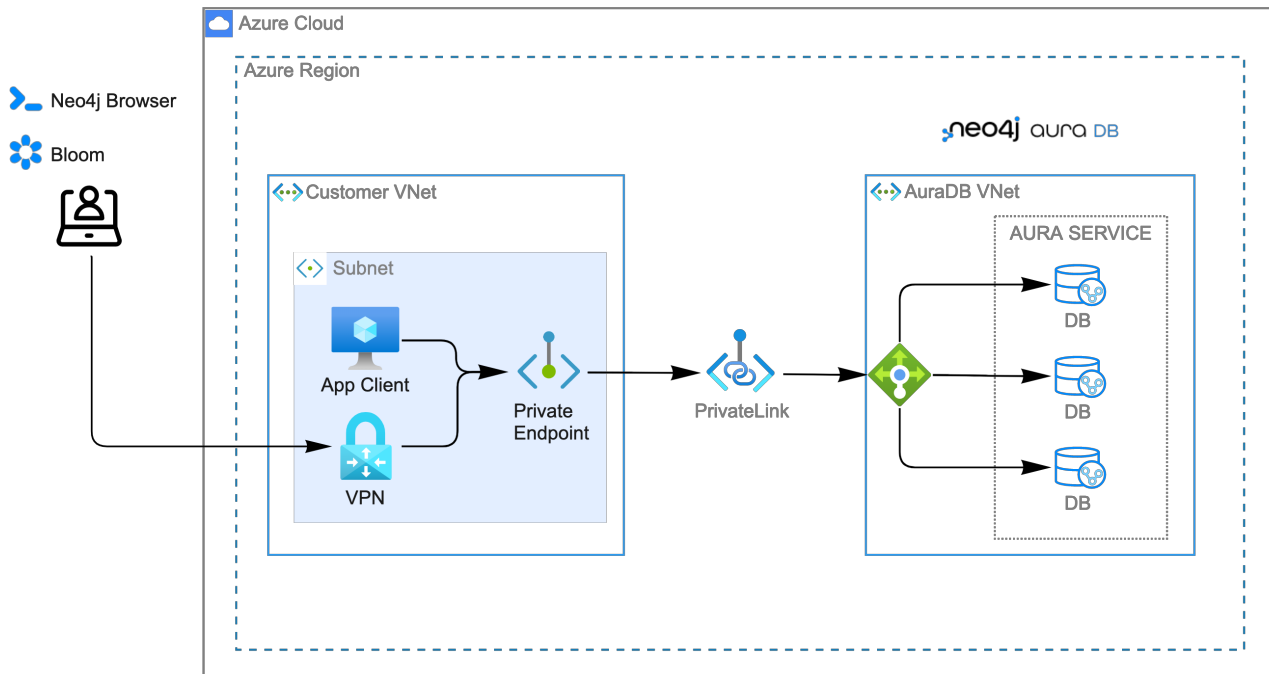


Figure 12. Accessing Browser and Bloom over a VPN

Enabling private endpoints

To enable private endpoints using Azure Private Link:

1. Select **Network Access** from the sidebar menu of the Console.
2. Select **New network access configuration** and follow the setup instructions.

Please see the [Azure Documentation](#) for required roles and permissions.

Supported TLS cipher suites

For additional security, client communications are carried via TLS v1.2 and TLS v1.3.

AuraDB has a restricted list of cipher suites accepted during the TLS handshake, and does not accept all of the available cipher suites. The following list conforms to safety recommendations from IANA, the OpenSSL, and GnuTLS library.

TLS v1.3:

- **TLS_CHACHA20_POLY1305_SHA256 (RFC8446)**
- **TLS_AES_128_GCM_SHA256 (RFC8446)**

- [TLS_AES_256_GCM_SHA384 \(RFC8446\)](#)

TLS v1.2:

- [TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 \(RFC5288\)](#)
- [TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 \(RFC5289\)](#)
- [TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 \(RFC5289\)](#)
- [TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 \(RFC7905\)](#)
- [TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 \(RFC5288\)](#)

Single Sign-On (SSO)

[AuraDB Virtual Dedicated Cloud](#)

[AuraDS Enterprise](#)

[AuraDB Business Critical](#)

SSO levels

Organization admins can configure organization level SSO (org SSO) and project level SSO (project SSO).

SSO is a log-in method and access, roles, and permissions are dictated by role-based access control (RBAC).

- **Org SSO:** Allows org admins to restrict how users log in when they are trying to access the org. Access beyond log-in is managed via RBAC.
- **Project-level SSO:** Impacts new database instances created within that project. It ensures users logging in with SSO have access to the database instances within the project. It depends on RBAC if the user can access and view or modify data within the database instances themselves. For this level, the role mapping may be used to grant users different levels of access based on a role in their identity provider (IdP). It **does not** give access to edit the project settings, for example to edit the project name, network access, or to edit the instance settings such as to rename an instance, or pause and resume.

SSO Org level roles

The following roles are available at the org level and these are assigned via invitation:

- Owner
- Admin
- Member

Table 1. Roles

Capability	Owner	Admin	Member
List org	✓	✓	✓
List org projects	✓	✓	✓
Update org	✓	✓	

Capability	Owner	Admin	Member
Add projects	✓	✓	
List existing SSO configs	✓	✓	
Add SSO configs	✓	✓	
List SSO configs on project-level	✓	✓	
Update SSO configs on project-level	✓	✓	
Delete SSO configs on project-level	✓	✓	
Invite non-owner users to org	✓	✓	
List users	✓	✓	
List roles	✓	✓	
List members of a project	✓	✓ ^[1]	
Invite owners to org	✓		
Add owner	✓		
Delete owners	✓		
Transfer projects to and from the org	✓ ^[2]		

Log-in methods

Log-in methods are different for each SSO level. Administrators can configure a combination of one or more of the log-in methods.

Org SSO supports:

- Email/password
- Okta
- Microsoft Entra ID
- Google SSO (not Google Workspace SSO)

An organization's administrator can add Aura as a log-in from a tile in an organization's Apps Dashboard.

Project SSO supports:

- User/password
- Okta
- Microsoft Entra ID

However, at the project level you cannot disable user/password, but at the org level you can disable email/password and Google SSO as long as you have at least one other custom SSO provider configured.

Setup requirements

Accessing Aura with SSO requires:

- Authorization Code Flow
- A publicly accessible IdP server

To configure SSO, go to [Aura Console](#) > [Settings](#) > [SSO Configuration](#).

To create an SSO Configuration either a Discovery URI or a combination of Issuer, Authorization Endpoint, Token Endpoint and JWKS URI is required.

Individual instance level SSO configurations available from Support

Support can assist with:

- Role mapping specific to a database instance
- Custom groups claim besides [groups](#)
- Updating SSO on already running instances

If you require support assistance, visit [Customer Support](#) and raise a support ticket including the following information:

1. The *Project ID* of the projects you want to use SSO for. See [Projects](#) for more information on how to find your *Project ID*.
2. The name of your IdP

Encryption

All data stored in Neo4j Aura is encrypted using intra-cluster encryption between the various nodes comprising your instance and encrypted at rest using the underlying cloud provider's encryption mechanism.

Aura always requires encrypted connections and ensures that clients validate server certificates when establishing a connection. This means that network traffic flowing to and from Neo4j Aura is always encrypted.

By default, each cloud provider encrypts all backup buckets (including the objects stored inside) using either [Google-managed encryption](#), [AWS SSE-S3 encryption](#), or [Azure Storage encryption](#).

To protect data at rest, Aura uses encrypted data storage capabilities offered by the cloud providers. Whether customers choose to host in AWS, Azure, or GCP, each object store provides server-side encrypted buckets for data at rest encryption. By default, AWS, Azure, and GCP encrypt all backup buckets (including the objects stored inside) with AWS SSE-S3 encryption, Azure Storage Encryption (SSE), or Google-managed encryption. This ensures all your data stored in any one of these cloud providers uses 256-bit Advanced Encryption Standard (AES).

In addition to Aura's default encryption for data at rest, Customer Managed Keys enable security-

conscious enterprises to manage encryption keys through their Cloud Service Provider's Key Management Services (KMS) on Aura, granting control over data protection and access management, including the ability to revoke access from Neo4j. This allows adherence to strict security policies alongside Aura's default enterprise-grade security measures.

Customer Managed Keys

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

A Customer Managed Key (CMK) gives you more control over key operations than the standard Neo4j encryption. These are created and managed using a supported cloud key management service (KMS). Externally, Customer Managed Keys are also known as Customer Managed Encryption Keys (CMEK).

When using a Customer Managed Key, all data at rest is encrypted with the key. Customer Managed Keys are supported for v4.x and v5.x instances.

When using Customer Managed Keys, you give Aura permission to encrypt and decrypt using the key, but Aura has no access to the key's material. Aura has no control over the availability of your externally managed key in the KMS.



The loss of a Customer Managed Key makes all data encrypted with that key inaccessible. Neo4j is unable to manage database instances if the key is disabled, deleted, expired, or if permissions are revoked.

Key rotation

In your KMS platform, you can either configure automatic rotation for the Customer Managed Key, or you can perform a manual rotation.

Although automatic rotation is not enforced by Aura, it is best practice to rotate keys regularly. Manual key rotation is **not** recommended.

Import an existing database

You can upload a database to instances encrypted with Customer Managed Keys in Neo4j 5 directly from the console or by using `neo4j-admin database upload`. If the database is larger than 4 GB, you have to use `neo4j-admin database upload`. Note that the `neo4j-admin push-to-cloud` command in Neo4j v4.4 and earlier is **not** supported for instances encrypted with Customer Managed Keys. For more information see the [Neo4j Admin database upload](#) documentation.

Clone an instance protected by CMK

To clone an instance protected by a Customer Managed Key, the key must be valid and available to Aura. The cloned instance, by default, uses the available Customer Managed Key for that region and product.

It is best practice to use the same CMK key as the instance it's being cloned from. You can override this to use another CMK key—but you can not use the Neo4j Managed Key.

Remove a CMK from Aura

When using a Customer Managed Key within Aura to encrypt one or more Aura database instances, it cannot be removed from Aura. If you no longer need to use this Customer Managed Key to encrypt Aura databases, first delete the Aura database instances that are encrypted with the key, then you can remove the key from Aura. Keep in mind that this process only breaks the link between the key and Aura — it does not delete the actual key from the Cloud KMS.

AWS keys

Create an AWS key

1. Create a key in the AWS KMS making sure the region matches your Aura database instance. Copy the generated ARN. You need it in the next step.
2. Go to **security settings** in the Aura Console, add a **Customer Managed Key** and copy the JSON code that is generated in the Aura Console when you add a key.
3. In the AWS KMS, edit the key policy to include the JSON code.

Edit the AWS key policy

After you have initially created a key in the AWS KMS, you can edit the key policy. In the AWS key policy, "Statement" is an array that consists of one or more objects. Each object in the array describes a security identifier (SID). The objects in the AWS code array are comma-separated, for example `[[{'a'}, {'b'}, {'c'}]]`.

Add a comma after the curly brace in the final SID, and then paste the JSON code that was generated in the Aura Console (for example `[[{'a'}, {'b'}, {'c'}, add code here]]`).

AWS regions

Aura supports AWS Customer Managed Keys that reside in the same region as the instance. When creating a Customer Managed Key in the AWS KMS, you can create a single-region key, or create a multi-region key.

Single-region keys reside in only one AWS region, which must be the same region as your Aura instance.

Multi-region keys have a primary region, however these can be replicated to other regions that match the region of your Aura instance. The replicas share the same key ID and different Amazon Resource Names (ARNs) with the primary key.

AWS automatic key rotation

Aura supports automatic key rotation via the AWS KMS. To enable automatic key rotation in the AWS KMS, tick the **Key rotation** checkbox after initially creating a key, to automatically rotate the key once a year.

Azure keys

Create an Azure key vault

Create a Key Vault in the Azure portal ensuring the region matches your Aura database instance region. Move through the tabs to enable to following:

- Purge protection
- Azure role-based access control
- Azure Disk Encryption for volume encryption
- Allow access from all networks

Create a key

1. When preparing to create a key, if needed grant a role assignment:
 - a. Inside the key vault, go to **Access Control (IAM)** and add role assignment.
 - b. In the **Role** tab, select **Key Vault Administrator**.
 - c. In the **Member** tab, select **User, group, or service principal**.
 - d. From **Select members**, add yourself or the relevant person, then **Review + Assign**.
2. Create a key in the Azure Key Vault.
3. After the key is created, click into key version and copy the **Key Identifier**, you need it in the next step.
4. Go to **security settings** in the Aura Console and add a **Customer Managed Key**.
5. Follow the instructions in the Aura Console for the next sections.

Create a service principal

In the Azure Entra ID tenant where your key is located, create a service principal linked to the Neo4j CMK Application with the **Neo4j CMK Application ID** displayed in the Aura Console.

One way to do this is by clicking the terminal icon at the top of the Azure portal, to open the Azure Cloud Shell.

Using Azure CLI, the command is:

```
az ad sp create --id Neo4jCMKApplicationID
```

For more information about the Azure CLI, see [az ad sp documentation](#).

Grant key permissions

1. To add role assignment to the Azure key, inside the key, go to **Access control (IAM)** and add role assignment.
2. In the **Role** tab, select **Key Vault Crypto Officer**.

3. In the Member tab, select **User, group, or service principal**.
4. In **Select members**, paste the **Neo4j CMK Application name** that is displayed in the Aura Console.
5. The **Neo4j CMK Application** should appear, select this application then **Review + Assign**.

GCP keys

Create a key ring

1. Go to **Key Management** in the Google Cloud console.
2. Create a **key ring**.
3. The key ring **Location type** should be set to **Region**.
4. Make sure the region matches your Aura database instance region.
5. Select **Create** and you are automatically taken to the key creation page.

Create a key

1. Create a key in the Google Console. You can use default settings for the options, but setting a key rotation period is recommended.
2. Select **Create** and you are brought to the key ring, with your key listed.
3. Click **More** (three dots) and **Copy resource name**, you need it in the next step. For more information, see [Google Cloud docs](#)
4. Go to **security settings** in the Aura Console and add a **Customer Managed Key**. Paste the **resource name** into the **Encryption Key Resource Name** field.
5. After you select **Add Key** in the Aura Console, three **service accounts** are displayed in the Aura Console. You will need these in the next steps.

Grant key permissions

1. Go to the Google Cloud console, click into the key and go to **Permissions** then **Grant Access**.
2. In **Add principals** paste the three service accounts from the Aura Console.
3. In **Assign roles** assign both **Cloud KMS CryptoKey Encrypter/Decrypter** and **Cloud KMS Viewer** roles to all three service accounts.

User management

User management is a feature within Aura that allows you to invite users and set their roles within an isolated environment.

Projects

Projects are the primary mechanism for granting users access to an Aura environment.

The project you're currently viewing is displayed in the header of the Console. You can select the project name to open the project dropdown menu, allowing you to view all the projects that you have access to and switch between them.

Additionally, you can perform the following actions from the project dropdown menu:

- Copy the *Project ID* of any project in the list by selecting the clipboard icon that appears when you hover over the project.
- Edit the name of the project you are currently viewing by selecting the pencil icon next to the project. This action requires you to be an *Admin* of the project.

Users

Each project can have multiple users with individual accounts allowing access to the same environment.

The users with access to a project can be viewed and managed from the **User Management** page. You can access the **User Management** page by selecting **User Management** from the sidebar menu of the Console.

Roles

Users within a project can be assigned one of the following roles:

- *Admin*
- *Member*
- *Viewer*

Table 2. Roles

Capability	Admin	Member	Viewer
View users and their roles	✓	✓	✓
View and open instances	✓	✓	✓
Access the Neo4j Customer Support Portal	✓	✓	✓
Perform all actions on instances ^[3]	✓	✓	
Clone data to new and existing instances	✓	✓	
Take on-demand snapshots	✓	✓	

Capability	Admin	Member	Viewer
Restore from snapshots	✓	✓	
Edit the project name	✓		
Invite new users to the project	✓		
Edit existing users' roles	✓		
Delete existing users from the project	✓		
View and edit billing information	✓		



Each project must have at least one *Admin*, but it is also possible for projects to have multiple *Admins*.

Inviting users

As an *Admin*, to invite a new user:

1. Select **Invite user** from the **User Management** page.
2. Enter the **Email** address of the person you want to invite.
3. Select the user's **Role**.
4. Select **Invite**.

The new user will appear within the list of users on the **User Management** page with the *Pending invite Status* until they accept the invite.

An email will be sent to the user with a link to accept the invite.

Editing users

As an *Admin*, to edit an existing user's role:

1. Select the pencil icon next to the user's name from the **User Management** page.
2. Select the user's new **Role**.
3. Select **Save changes**.

Deleting users

As an *Admin*, to delete an existing user:

1. Select the trash can icon next to the user's name from the **User Management** page.
2. Select **Delete**.



It is also possible to delete a user whose **Status** is *Pending invite*.

Select the trash can icon next to the user's name, and then select **Revoke**.

Accepting an invite

When invited to a project, you receive an email with a link to accept the invite. This link directs you to the Aura Console, where a **Project invitation** modal will appear. You can select the projects you have been invited to and accept or decline the invites.

You can also close the **Project invitation** modal without accepting or declining the invites and later manually re-open the modal by selecting the **Pending invites** envelope icon in the Console header.



User management within the Aura Console does not replace built-in roles or fine-grained RBAC at the database level.

[1] An admin can only list members of projects the admin is also a member of.

[2] An owner needs to permission for both the source and destination orgs.





[3] Actions include creating, deleting, pausing, resuming, and editing instances.

APOC support




APOC (Awesome Procedures on Cypher) is a Neo4j library that provides access to additional procedures and functions, extending the use of the Cypher query language. For more information on APOC, see [the APOC documentation](#).









A subset of the APOC Core functions and procedures are pre-installed and available in Aura, as shown below:

apoc


Qualified Name	Type
apoc.case  For each pair of conditional and read-only queries in the given <code>LIST<ANY></code> , this procedure will run the first query for which the conditional is evaluated to true. If none of the conditionals are true, the <code>ELSE</code> query will run instead.	Procedure
apoc.help  Returns descriptions of the available APOC procedures and functions. If a keyword is provided, it will return only those procedures and functions that have the keyword in their name.	Procedure
apoc.version  Returns the APOC version currently installed.	Function
apoc.when  This procedure will run the read-only <code>ifQuery</code> if the conditional has evaluated to true, otherwise the <code>elseQuery</code> will run.	Procedure





apoc.agg

Qualified Name	Type
apoc.agg.first  Returns the first value from the given collection.	Function
apoc.agg.graph  Returns all distinct <code>NODE</code> and <code>RELATIONSHIP</code> values collected into a <code>MAP</code> with the keys <code>nodes</code> and <code>relationships</code> .	Function
apoc.agg.last  Returns the last value from the given collection.	Function



Qualified Name	Type
apoc.agg.maxItems  Returns a MAP {items: LIST<ANY>, value: ANY} where the value key is the maximum value present, and items represent all items with the same value. The size of the list of items can be limited to a given max size.	Function
apoc.agg.median  Returns the mathematical median for all non-null INTEGER and FLOAT values.	Function
apoc.agg.minItems  Returns a MAP {items: LIST<ANY>, value: ANY} where the value key is the minimum value present, and items represent all items with the same value. The size of the list of items can be limited to a given max size.	Function
apoc.agg.nth  Returns the nth value in the given collection (to fetch the last item of an unknown length collection, -1 can be used).	Function
apoc.agg.percentiles  Returns the given percentiles over the range of numerical values in the given collection.	Function
apoc.agg.product  Returns the product of all non-null INTEGER and FLOAT values in the collection.	Function
apoc.agg.slice  Returns a subset of non-null values from the given collection (the collection is considered to be zero-indexed). To specify the range from start until the end of the collection, the length should be set to -1.	Function
apoc.agg.statistics  Returns the following statistics on the INTEGER and FLOAT values in the given collection: percentiles, min, minNonZero, max, total, mean, stdev.	Function

apoc.algo



Qualified Name	Type
apoc.algo.aStar  Runs the A* search algorithm to find the optimal path between two NODE values, using the given RELATIONSHIP property name for the cost function.	Procedure





Qualified Name	Type
apoc.algo.aStarConfig  Runs the A* search algorithm to find the optimal path between two NODE values, using the given RELATIONSHIP property name for the cost function. This procedure looks for weight, latitude and longitude properties in the config.	Procedure
apoc.algo.allSimplePaths  Runs a search algorithm to find all of the simple paths between the given RELATIONSHIP values, up to a max depth described by maxNodes . The returned paths will not contain loops.	Procedure
apoc.algo.cover  Returns all RELATIONSHIP values connecting the given set of NODE values.	Procedure
apoc.algo.dijkstra  Runs Dijkstra's algorithm using the given RELATIONSHIP property as the cost function.	Procedure

apoc.any


Qualified Name	Type
apoc.any.properties  Returns all properties of the given object. The object can be a virtual NODE , a real NODE , a virtual RELATIONSHIP , a real RELATIONSHIP , or a MAP .	Function
apoc.any.property  Returns the property for the given key from an object. The object can be a virtual NODE , a real NODE , a virtual RELATIONSHIP , a real RELATIONSHIP , or a MAP .	Function

apoc.atomic





Qualified Name	Type
apoc.atomic.add  Sets the given property to the sum of itself and the given INTEGER or FLOAT value. The procedure then sets the property to the returned sum.	Procedure
apoc.atomic.concat  Sets the given property to the concatenation of itself and the STRING value. The procedure then sets the property to the returned STRING .	Procedure













Qualified Name	Type
apoc.atomic.insert  Inserts a value at position into the <code>LIST<ANY></code> value of a property. The procedure then sets the result back on the property.	Procedure
apoc.atomic.remove  Removes the element at position from the <code>LIST<ANY></code> value of a property. The procedure then sets the property to the resulting <code>LIST<ANY></code> value.	Procedure
apoc.atomic.subtract  Sets the property of a value to itself minus the given <code>INTEGER</code> or <code>FLOAT</code> value. The procedure then sets the property to the returned sum.	Procedure
apoc.atomic.update  Updates the value of a property with a Cypher operation.	Procedure













apoc.bitwise

Qualified Name	Type
apoc.bitwise.op  Returns the result of the bitwise operation	Function












apoc.coll

Qualified Name	Type
apoc.coll.avg  Returns the average of the numbers in the <code>LIST<INTEGER FLOAT></code> .	Function
apoc.coll.combinations  Returns a collection of all combinations of <code>LIST<ANY></code> elements between the selection size <code>minSelect</code> and <code>maxSelect</code> (default: <code>minSelect</code>).	Function
apoc.coll.contains  Returns whether or not the given value exists in the given collection (using a HashSet).	Function
apoc.coll.containsAll  Returns whether or not all of the given values exist in the given collection (using a HashSet).	Function














Qualified Name	Type
apoc.coll.containsAllSorted  Returns whether or not all of the given values in the second <code>LIST<ANY></code> exist in an already sorted collection (using a binary search).	Function
apoc.coll.containsDuplicates  Returns true if a collection contains duplicate elements.	Function
apoc.coll.containsSorted  Returns whether or not the given value exists in an already sorted collection (using a binary search).	Function
apoc.coll.different  Returns true if all the values in the given <code>LIST<ANY></code> are unique.	Function
apoc.coll.disjunction  Returns the disjunct set from two <code>LIST<ANY></code> values.	Function
apoc.coll.dropDuplicateNeighbors  Removes duplicate consecutive objects in the <code>LIST<ANY></code> .	Function
apoc.coll.duplicates  Returns a <code>LIST<ANY></code> of duplicate items in the collection.	Function
apoc.coll.duplicatesWithCount  Returns a <code>LIST<ANY></code> of duplicate items in the collection and their count, keyed by <code>item</code> and <code>count</code> .	Function
apoc.coll.elements  Deconstructs a <code>LIST<ANY></code> into identifiers indicating their specific type.	Procedure
apoc.coll.fill  Returns a <code>LIST<ANY></code> with the given count of items.	Function
apoc.coll.flatten  Flattens the given <code>LIST<ANY></code> (to flatten nested <code>LIST<ANY></code> values, set recursive to true).	Function
apoc.coll.frequencies  Returns a <code>LIST<ANY></code> of frequencies of the items in the collection, keyed by <code>item</code> and <code>count</code> .	Function



Qualified Name	Type
apoc.coll.frequenciesAsMap  Returns a MAP of frequencies of the items in the collection, keyed by item and count .	Function
apoc.coll.indexOf  Returns the index for the first occurrence of the specified value in the LIST<ANY> .	Function
apoc.coll.insert  Inserts a value into the specified index in the LIST<ANY> .	Function
apoc.coll.insertAll  Inserts all of the values into the LIST<ANY> , starting at the specified index.	Function
apoc.coll.intersection  Returns the distinct intersection of two LIST<ANY> values.	Function
apoc.coll.isEqualCollection  Returns true if the two collections contain the same elements with the same cardinality in any order (using a HashMap).	Function
apoc.coll.max  Returns the maximum of all values in the given LIST<ANY> .	Function
apoc.coll.min  Returns the minimum of all values in the given LIST<ANY> .	Function
apoc.coll.occurrences  Returns the count of the given item in the collection.	Function
apoc.coll.pairs  Returns a LIST<ANY> of adjacent elements in the LIST<ANY> ([1,2],[2,3],[3,null]).	Function
apoc.coll.pairsMin  Returns LIST<ANY> values of adjacent elements in the LIST<ANY> ([1,2],[2,3]), skipping the final element.	Function
apoc.coll.partition  Partitions the original LIST<ANY> into a new LIST<ANY> of the given batch size. The final LIST<ANY> may be smaller than the given batch size.	Function

Qualified Name	Type
apoc.coll.partition <p>Partitions the original <code>LIST<ANY></code> into a new <code>LIST<ANY></code> of the given batch size. The final <code>LIST<ANY></code> may be smaller than the given batch size.</p>	Procedure
apoc.coll.randomItem <p>Returns a random item from the <code>LIST<ANY></code>, or null on <code>LIST<NOTHING></code> or <code>LIST<NULL></code>.</p>	Function
apoc.coll.randomItems <p>Returns a <code>LIST<ANY></code> of <code>itemCount</code> random items from the original <code>LIST<ANY></code> (optionally allowing elements in the original <code>LIST<ANY></code> to be selected more than once).</p>	Function
apoc.coll.remove <p>Removes a range of values from the <code>LIST<ANY></code>, beginning at position <code>index</code> for the given length of values.</p>	Function
apoc.coll.removeAll <p>Returns the first <code>LIST<ANY></code> with all elements also present in the second <code>LIST<ANY></code> removed.</p>	Function
apoc.coll.runningTotal <p>Returns an accumulative <code>LIST<INTEGER FLOAT></code>.</p>	Function
apoc.coll.set <p>Sets the element at the given index to the new value.</p>	Function
apoc.coll.shuffle <p>Returns the <code>LIST<ANY></code> shuffled.</p>	Function
apoc.coll.sort <p>Sorts the given <code>LIST<ANY></code> into ascending order.</p>	Function
apoc.coll.sortMaps <p>Sorts the given <code>LIST<MAP<STRING, ANY>></code> into descending order, based on the <code>MAP</code> property indicated by <code>prop</code>.</p>	Function
apoc.coll.sortMulti <p>Sorts the given <code>LIST<MAP<STRING, ANY>></code> by the given fields. To indicate that a field should be sorted according to ascending values, prefix it with a caret (^). It is also possible to add limits to the <code>LIST<MAP<STRING, ANY>></code> and to skip values.</p>	Function










Qualified Name	Type
apoc.coll.sortNodes  Sorts the given <code>LIST<NODE></code> by the property of the nodes into descending order.	Function
apoc.coll.sortText  Sorts the given <code>LIST<STRING></code> into ascending order.	Function
apoc.coll.split  Splits a collection by the given value. The value itself will not be part of the resulting <code>LIST<ANY></code> values.	Procedure
apoc.coll.subtract  Returns the first <code>LIST<ANY></code> as a set with all the elements of the second <code>LIST<ANY></code> removed.	Function
apoc.coll.sum  Returns the sum of all the <code>INTEGER FLOAT</code> in the <code>LIST<INTEGER FLOAT></code> .	Function
apoc.coll.sumLongs  Returns the sum of all the <code>INTEGER FLOAT</code> in the <code>LIST<INTEGER FLOAT></code> .	Function
apoc.coll.toSet  Returns a unique <code>LIST<ANY></code> from the given <code>LIST<ANY></code> .	Function
apoc.coll.union  Returns the distinct union of the two given <code>LIST<ANY></code> values.	Function
apoc.coll.unionAll  Returns the full union of the two given <code>LIST<ANY></code> values (duplicates included).	Function
apoc.coll.zip  Returns the two given <code>LIST<ANY></code> values zipped together as a <code>LIST<LIST<ANY>></code> .	Function
apoc.coll.zipToRows  Returns the two <code>LIST<ANY></code> values zipped together, with one row per zipped pair.	Procedure












apoc.convert

Qualified Name	Type
apoc.convert.fromJsonList  Converts the given JSON list into a Cypher <code>LIST<STRING></code> .	Function
apoc.convert.fromJsonMap  Converts the given JSON map into a Cypher <code>MAP</code> .	Function
apoc.convert.getJsonProperty  Converts a serialized JSON object from the property of the given <code>NODE</code> into the equivalent Cypher structure (e.g. <code>MAP</code> , <code>LIST<ANY></code>).	Function
apoc.convert.getJsonPropertyMap  Converts a serialized JSON object from the property of the given <code>NODE</code> into a Cypher <code>MAP</code> .	Function
apoc.convert.setJsonProperty  Serializes the given JSON object and sets it as a property on the given <code>NODE</code> .	Procedure
apoc.convert.toJson  Serializes the given JSON value.	Function
apoc.convert.toList  Converts the given value into a <code>LIST<ANY></code> .	Function
apoc.convert.toMap  Converts the given value into a <code>MAP</code> .	Function
apoc.convert.toNode  Converts the given value into a <code>NODE</code> .	Function
apoc.convert.toNodeList  Converts the given value into a <code>LIST<NODE></code> .	Function
apoc.convert.toRelationship  Converts the given value into a <code>RELATIONSHIP</code> .	Function
apoc.convert.toRelationshipList  Converts the given value into a <code>LIST<RELATIONSHIP></code> .	Function
apoc.convert.toSet  Converts the given value into a set represented in Cypher as a <code>LIST<ANY></code> .	Function

Qualified Name	Type
apoc.convert.toSortedJsonMap  Converts a serialized JSON object from the property of a given NODE into a Cypher MAP .	Function
apoc.convert.toTree  Returns a stream of MAP values, representing the given PATH values as a tree with at least one root.	Procedure

apoc.create

Qualified Name	Type
apoc.create.addLabels  Adds the given labels to the given NODE values.	Procedure
apoc.create.node  Creates a NODE with the given dynamic labels.	Procedure
apoc.create.nodes  Creates NODE values with the given dynamic labels.	Procedure
apoc.create.relationship  Creates a RELATIONSHIP with the given dynamic relationship type.	Procedure
apoc.create.removeLabels  Removes the given labels from the given NODE values.	Procedure
apoc.create.removeProperties  Removes the given properties from the given NODE values.	Procedure
apoc.create.removeRelProperties  Removes the given properties from the given RELATIONSHIP values.	Procedure
apoc.create.setLabels  Sets the given labels to the given NODE values. Non-matching labels are removed from the nodes.	Procedure
apoc.create.setProperties  Sets the given properties to the given NODE values.	Procedure

Qualified Name	Type
apoc.create.setProperty  Sets the given property to the given NODE values.	Procedure
apoc.create.setRelProperties  Sets the given properties on the RELATIONSHIP values.	Procedure
apoc.create.setRelProperty  Sets the given property on the RELATIONSHIP values.	Procedure
apoc.create.uuid  Returns a UUID.	Function Deprecated
apoc.create.uuids  Returns a stream of UUIDs.	Procedure Deprecated
apoc.create.vNode  Returns a virtual NODE .	Procedure
apoc.create.vNode  Returns a virtual NODE .	Function
apoc.create.vNodes  Returns virtual NODE values.	Procedure
apoc.create.vRelationship  Returns a virtual RELATIONSHIP .	Procedure
apoc.create.vRelationship  Returns a virtual RELATIONSHIP .	Function
apoc.create.virtual.fromNode  Returns a virtual NODE from the given existing NODE . The virtual NODE only contains the requested properties.	Function

apoc.cypher











Qualified Name	Type
apoc.cypher.dolt <p>Runs a dynamically constructed statement with the given parameters. This procedure allows for both read and write statements.</p>	Procedure
apoc.cypher.run <p>Runs a dynamically constructed read-only statement with the given parameters.</p>	Procedure
apoc.cypher.runFirstColumnMany <p>Runs the given statement with the given parameters and returns the first column collected into a <code>LIST<ANY></code>.</p>	Function
apoc.cypher.runFirstColumnSingle <p>Runs the given statement with the given parameters and returns the first element of the first column.</p>	Function
apoc.cypher.runMany <p>Runs each semicolon separated statement and returns a summary of the statement outcomes.</p>	Procedure
apoc.cypher.runTimeboxed <p>Terminates a Cypher statement if it has not finished before the set timeout (ms).</p>	Procedure

apoc.data


Qualified Name	Type
apoc.data.url <p>Turns a URL into a <code>MAP</code>.</p>	Function

apoc.date

Qualified Name	Type
apoc.date.add <p>Adds a unit of specified time to the given timestamp.</p>	Function
apoc.date.convert <p>Converts the given timestamp from one time unit into a timestamp of a different time unit.</p>	Function

Qualified Name	Type
apoc.date.convertFormat  Converts a STRING of one type of date format into a STRING of another type of date format.	Function
apoc.date.currentTimestamp  Returns the current Unix epoch timestamp in milliseconds.	Function
apoc.date.field  Returns the value of one field from the given date time.	Function
apoc.date.fields  Splits the given date into fields returning a MAP containing the values of each field.	Function
apoc.date.format  Returns a STRING representation of the time value. The time unit (default: ms), date format (default: ISO), and time zone (default: current time zone) can all be changed.	Function
apoc.date.fromISO8601  Converts the given date STRING (ISO8601) to an INTEGER representing the time value in milliseconds.	Function
apoc.date.parse  Parses the given date STRING from a specified format into the specified time unit.	Function
apoc.date.systemTimezone  Returns the display name of the system time zone (e.g. Europe/London).	Function
apoc.date.toISO8601  Returns a STRING representation of a specified time value in the ISO8601 format.	Function
apoc.date.toYears  Converts the given timestamp or the given date into a FLOAT representing years.	Function

apoc.diff

Qualified Name	Type
apoc.diff.nodes  Returns a MAP detailing the differences between the two given NODE values.	Function

apoc.do













Qualified Name	Type
apoc.do.case ⓘ For each pair of conditional queries in the given <code>LIST<ANY></code> , this procedure will run the first query for which the conditional is evaluated to true. If none of the conditionals are true, the <code>ELSE</code> query will run instead.	Procedure
apoc.do.when ⓘ Runs the given read/write <code>ifQuery</code> if the conditional has evaluated to true, otherwise the <code>elseQuery</code> will run.	Procedure

apoc.example









Qualified Name	Type
apoc.example.movies ⓘ Seeds the database with the Neo4j movie dataset.	Procedure

apoc.export


Qualified Name	Type
apoc.export.csv.all ⓘ Exports the full database to the provided CSV file.	Procedure
apoc.export.csv.data ⓘ Exports the given <code>NODE</code> and <code>RELATIONSHIP</code> values to the provided CSV file.	Procedure
apoc.export.csv.graph ⓘ Exports the given graph to the provided CSV file.	Procedure
apoc.export.csv.query ⓘ Exports the results from running the given Cypher query to the provided CSV file.	Procedure
apoc.export.cypher.all ⓘ Exports the full database (incl. indexes) as Cypher statements to the provided file (default: Cypher Shell).	Procedure



Qualified Name	Type
apoc.export.cypher.data  Exports the given NODE and RELATIONSHIP values (incl. indexes) as Cypher statements to the provided file (default: Cypher Shell).	Procedure
apoc.export.cypher.graph  Exports the given graph (incl. indexes) as Cypher statements to the provided file (default: Cypher Shell).	Procedure
apoc.export.cypher.query  Exports the NODE and RELATIONSHIP values from the given Cypher query (incl. indexes) as Cypher statements to the provided file (default: Cypher Shell).	Procedure
apoc.export.cypher.schema  Exports all schema indexes and constraints to Cypher statements.	Procedure
apoc.export.graphml.all  Exports the full database to the provided GraphML file.	Procedure
apoc.export.graphml.data  Exports the given NODE and RELATIONSHIP values to the provided GraphML file.	Procedure
apoc.export.graphml.graph  Exports the given graph to the provided GraphML file.	Procedure
apoc.export.graphml.query  Exports the given NODE and RELATIONSHIP values from the Cypher statement to the provided GraphML file.	Procedure
apoc.export.json.all  Exports the full database to the provided JSON file.	Procedure
apoc.export.json.data  Exports the given NODE and RELATIONSHIP values to the provided JSON file.	Procedure
apoc.export.json.graph  Exports the given graph to the provided JSON file.	Procedure
apoc.export.json.query  Exports the results from the Cypher statement to the provided JSON file.	Procedure

apoc.graph



Qualified Name	Type
apoc.graph.from  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the given data.	Procedure
apoc.graph.fromCypher  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the data returned by the given Cypher statement.	Procedure
apoc.graph.fromDB  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the data returned by the given database.	Procedure
apoc.graph.fromData  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the given data.	Procedure
apoc.graph.fromDocument  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the data returned by the given JSON file.	Procedure
apoc.graph.fromPath  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the data returned by the given PATH .	Procedure
apoc.graph.fromPaths  Generates a virtual sub-graph by extracting all of the NODE and RELATIONSHIP values from the data returned by the given PATH values.	Procedure
apoc.graph.validateDocument  Validates the JSON file and returns the result of the validation.	Procedure

apoc.hashing


Qualified Name	Type
apoc.hashing.fingerprint  Calculates a MD5 checksum over a NODE or RELATIONSHIP (identical entities share the same checksum). Unsuitable for cryptographic use-cases.	Function

Qualified Name	Type
apoc.hashing.fingerprintGraph  Calculates a MD5 checksum over the full graph. This function uses in-memory data structures. Unsuitable for cryptographic use-cases.	Function
apoc.hashing.fingerprinting  Calculates a MD5 checksum over a NODE or RELATIONSHIP (identical entities share the same checksum). Unlike <code>apoc.hashing.fingerprint()</code> , this function supports a number of config parameters. Unsuitable for cryptographic use-cases.	Function


apoc.import

Qualified Name	Type
apoc.import.csv  Imports NODE and RELATIONSHIP values with the given labels and types from the provided CSV file.	Procedure
apoc.import.graphml  Imports a graph from the provided GraphML file.	Procedure




apoc.json

Qualified Name	Type
apoc.json.path  Returns the given JSON path.	Function






apoc.label

Qualified Name	Type
apoc.label.exists  Returns true or false depending on whether or not the given label exists.	Function



apoc.load














Qualified Name	Type
apoc.load.json  Imports JSON file as a stream of values if the given JSON file is a LIST<ANY> . If the given JSON file is a MAP , this procedure imports a single value instead.	Procedure
apoc.load.jsonArray  Loads array from a JSON URL (e.g. web-API) to then import the given JSON file as a stream of values.	Procedure
apoc.load.xml  Loads a single nested MAP from an XML URL (e.g. web-API).	Procedure










apoc.lock

Qualified Name	Type
apoc.lock.all  Acquires a write lock on the given NODE and RELATIONSHIP values.	Procedure
apoc.lock.nodes  Acquires a write lock on the given NODE values.	Procedure
apoc.lock.read.nodes  Acquires a read lock on the given NODE values.	Procedure
apoc.lock.read.rels  Acquires a read lock on the given RELATIONSHIP values.	Procedure
apoc.lock.rels  Acquires a write lock on the given RELATIONSHIP values.	Procedure


apoc.map









Qualified Name	Type
apoc.map.clean  Filters the keys and values contained in the given LIST<ANY> values.	Function
apoc.map.flatten  Flattens nested items in the given MAP . This function is the reverse of the apoc.map.unflatten function.	Function

Qualified Name	Type
apoc.map.fromLists  Creates a MAP from the keys and values in the given LIST<ANY> values.	Function
apoc.map.fromNodes  Returns a MAP of the given prop to the node of the given label.	Function
apoc.map.fromPairs  Creates a MAP from the given LIST<LIST<ANY>> of key-value pairs.	Function
apoc.map.fromValues  Creates a MAP from the alternating keys and values in the given LIST<ANY> .	Function
apoc.map.get  Returns a value for the given key. If the given key does not exist, or lacks a default value, this function will throw an exception.	Function
apoc.map.groupBy  Creates a MAP of the LIST<ANY> keyed by the given property, with single values.	Function
apoc.map.groupByMulti  Creates a MAP of the LIST<ANY> values keyed by the given property, with the LIST<ANY> values.	Function
apoc.map.merge  Merges the two given MAP values into one MAP .	Function
apoc.map.mergeList  Merges all MAP values in the given LIST<MAP<STRING, ANY>> into one MAP .	Function
apoc.map.mget  Returns a LIST<ANY> for the given keys. If one of the keys does not exist, or lacks a default value, this function will throw an exception.	Function
apoc.map.removeKey  Removes the given key from the MAP (recursively if recursive is true).	Function
apoc.map.removeKeys  Removes the given keys from the MAP (recursively if recursive is true).	Function
apoc.map.setEntry  Adds or updates the given entry in the MAP .	Function





Qualified Name	Type
apoc.map.setKey  Adds or updates the given entry in the MAP .	Function
apoc.map.setLists  Adds or updates the given keys/value pairs provided in LIST<ANY> format (e.g. [key1, key2],[value1, value2]) in a MAP .	Function
apoc.map.setPairs  Adds or updates the given key/value pairs (e.g. [key1,value1],[key2,value2]) in a MAP .	Function
apoc.map.setValues  Adds or updates the alternating key/value pairs (e.g. [key1,value1,key2,value2]) in a MAP .	Function
apoc.map.sortedProperties  Returns a LIST<ANY> of key/value pairs. The pairs are sorted by alphabetically by key, with optional case sensitivity.	Function
apoc.map.submap  Returns a sub-map for the given keys. If one of the keys does not exist, or lacks a default value, this function will throw an exception.	Function
apoc.map.unflatten  Unflattens items in the given MAP to nested items. This function is the reverse of the apoc.map.flatten function.	Function
apoc.map.updateTree  Adds the data MAP on each level of the nested tree, where the key-value pairs match.	Function
apoc.map.values  Returns a LIST<ANY> indicated by the given keys (returns a null value if a given key is missing).	Function

apoc.math












Qualified Name	Type
apoc.math.maxByte  Returns the maximum value of a byte.	Function

Qualified Name	Type
apoc.math.maxDouble  Returns the largest positive finite value of type double.	Function
apoc.math.maxInt  Returns the maximum value of an integer.	Function
apoc.math.maxLong  Returns the maximum value of a long.	Function
apoc.math.minByte  Returns the minimum value of a byte.	Function
apoc.math.minDouble  Returns the smallest positive non-zero value of type double.	Function
apoc.math.minInt  Returns the minimum value of an integer.	Function
apoc.math.minLong  Returns the minimum value of a long.	Function
apoc.math.regr  Returns the coefficient of determination (R-squared) for the values of propertyY and propertyX in the given label.	Procedure







apoc.merge

Qualified Name	Type
apoc.merge.node  Merges the given NODE values with the given dynamic labels.	Procedure
apoc.merge.node.eager  Merges the given NODE values with the given dynamic labels eagerly.	Procedure
apoc.merge.relationship  Merges the given RELATIONSHIP values with the given dynamic types/properties.	Procedure
apoc.merge.relationship.eager  Merges the given RELATIONSHIP values with the given dynamic types/properties eagerly.	Procedure





apoc.meta





Qualified Name	Type
apoc.meta.cypher.isType  Returns true if the given value matches the given type.	Function
apoc.meta.cypher.type  Returns the type name of the given value.	Function
apoc.meta.cypher.types  Returns a MAP containing the type names of the given values.	Function
apoc.meta.data  Examines the full graph and returns a table of metadata.	Procedure
apoc.meta.graph  Examines the full graph and returns a meta-graph.	Procedure
apoc.meta.graphSample  Examines the full graph and returns a meta-graph. Unlike <code>apoc.meta.graph</code> , this procedure does not filter away non-existing paths.	Procedure
apoc.meta.nodeTypeProperties  Examines the full graph and returns a table of metadata with information about the NODE values therein.	Procedure
apoc.meta.relTypeProperties  Examines the full graph and returns a table of metadata with information about the RELATIONSHIP values therein.	Procedure
apoc.meta.schema  Examines the given sub-graph and returns metadata as a MAP .	Procedure
apoc.meta.stats  Returns the metadata stored in the transactional database statistics.	Procedure
apoc.meta.subGraph  Examines the given sub-graph and returns a meta-graph.	Procedure

apoc.neighbors







Qualified Name	Type
apoc.neighbors.athop  Returns all NODE values connected by the given RELATIONSHIP types at the specified distance.	Procedure
apoc.neighbors.athop.count  Returns the count of all NODE values connected by the given RELATIONSHIP types at the specified distance.	Procedure
apoc.neighbors.byhop  Returns all NODE values connected by the given RELATIONSHIP types within the specified distance. Returns LIST<NODE> values, where each PATH of NODE values represents one row of the LIST<NODE> values.	Procedure
apoc.neighbors.byhop.count  Returns the count of all NODE values connected by the given RELATIONSHIP types within the specified distance.	Procedure
apoc.neighbors.tohop  Returns all NODE values connected by the given RELATIONSHIP types within the specified distance. NODE values are returned individually for each row.	Procedure
apoc.neighbors.tohop.count  Returns the count of all NODE values connected by the given RELATIONSHIP values in the pattern within the specified distance.	Procedure





apoc.node

Qualified Name	Type
apoc.node.degree  Returns the total degrees of the given NODE .	Function
apoc.node.degree.in  Returns the total number of incoming RELATIONSHIP values connected to the given NODE .	Function
apoc.node.degree.out  Returns the total number of outgoing RELATIONSHIP values from the given NODE .	Function
apoc.node.id  Returns the id for the given virtual NODE .	Function








Qualified Name	Type
apoc.node.labels  Returns the labels for the given virtual NODE .	Function
apoc.node.relationship.exists  Returns a BOOLEAN based on whether the given NODE has a connecting RELATIONSHIP (or whether the given NODE has a connecting RELATIONSHIP of the given type and direction).	Function
apoc.node.relationship.types  Returns a LIST<STRING> of distinct RELATIONSHIP types for the given NODE .	Function
apoc.node.relationships.exist  Returns a BOOLEAN based on whether the given NODE has connecting RELATIONSHIP values (or whether the given NODE has connecting RELATIONSHIP values of the given type and direction).	Function

apoc.nodes

Qualified Name	Type
apoc.nodes.collapse  Merges NODE values together in the given LIST<NODE> . The NODE values are then combined to become one NODE , with all labels of the previous NODE values attached to it, and all RELATIONSHIP values pointing to it.	Procedure
apoc.nodes.connected  Returns true when a given NODE is directly connected to another given NODE . This function is optimized for dense nodes.	Function
apoc.nodes.delete  Deletes all NODE values with the given ids.	Procedure
apoc.nodes.get  Returns all NODE values with the given ids.	Procedure
apoc.nodes.group  Allows for the aggregation of NODE values based on the given properties. This procedure returns virtual NODE values.	Procedure
apoc.nodes.isDense  Returns true if the given NODE is a dense node.	Function

Qualified Name	Type
apoc.nodes.link  Creates a linked list of the given NODE values connected by the given RELATIONSHIP type.	Procedure
apoc.nodes.relationship.types  Returns a LIST<STRING> of distinct RELATIONSHIP types from the given LIST<NODE> values.	Function
apoc.nodes.relationships.exist  Returns a BOOLEAN based on whether or not the given NODE values have the given RELATIONSHIP values.	Function
apoc.nodes.rels  Returns all RELATIONSHIP values with the given ids.	Procedure




apoc.number

Qualified Name	Type
apoc.number.arabicToRoman  Converts the given Arabic numbers to Roman numbers.	Function
apoc.number.exact.add  Returns the result of adding the two given large numbers (using Java BigDecimal).	Function
apoc.number.exact.div  Returns the result of dividing a given large number with another given large number (using Java BigDecimal).	Function
apoc.number.exact.mul  Returns the result of multiplying two given large numbers (using Java BigDecimal).	Function
apoc.number.exact.sub  Returns the result of subtracting a given large number from another given large number (using Java BigDecimal).	Function
apoc.number.exact.toExact  Returns the exact value of the given number (using Java BigDecimal).	Function
apoc.number.exact.toFloat  Returns the FLOAT of the given large number (using Java BigDecimal).	Function








Qualified Name	Type
apoc.number.exact.toInteger Returns the INTEGER of the given large number (using Java BigDecimal).	Function
apoc.number.format Formats the given INTEGER or FLOAT using the given pattern and language to produce a STRING .	Function
apoc.number.parseFloat Parses the given STRING using the given pattern and language to produce a FLOAT .	Function
apoc.number.parseInt Parses the given STRING using the given pattern and language to produce a INTEGER .	Function
apoc.number.romanToArabic Converts the given Roman numbers to Arabic numbers.	Function

apoc.path










Qualified Name	Type
apoc.path.combine Combines the two given PATH values into one PATH .	Function
apoc.path.create Returns a PATH from the given start NODE and LIST<RELATIONSHIP> .	Function
apoc.path.elements Converts the given PATH into a LIST<NODE RELATIONSHIP> .	Function
apoc.path.expand Returns PATH values expanded from the start NODE following the given RELATIONSHIP types from min-depth to max-depth.	Procedure
apoc.path.expandConfig Returns PATH values expanded from the start NODE with the given RELATIONSHIP types from min-depth to max-depth.	Procedure
apoc.path.slice Returns a new PATH of the given length, taken from the given PATH at the given offset.	Function









Qualified Name	Type
apoc.path.spanningTree  Returns spanning tree PATH values expanded from the start NODE following the given RELATIONSHIP types to max-depth.	Procedure
apoc.path.subgraphAll  Returns the sub-graph reachable from the start NODE following the given RELATIONSHIP types to max-depth.	Procedure
apoc.path.subgraphNodes  Returns the NODE values in the sub-graph reachable from the start NODE following the given RELATIONSHIP types to max-depth.	Procedure

apoc.periodic



Qualified Name	Type
apoc.periodic.cancel  Cancels the given background job.	Procedure
apoc.periodic.commit  Runs the given statement in separate batched transactions.	Procedure
apoc.periodic.countdown  Runs a repeatedly called background statement until it returns 0.	Procedure
apoc.periodic.iterate  Runs the second statement for each item returned by the first statement. This procedure returns the number of batches and the total number of processed rows.	Procedure
apoc.periodic.list  Returns a LIST<ANY> of all background jobs.	Procedure
apoc.periodic.repeat  Runs a repeatedly called background job. To stop this procedure, use apoc.periodic.cancel .	Procedure
apoc.periodic.submit  Creates a background job which runs the given Cypher statement once.	Procedure

apoc.refactor









Qualified Name	Type
apoc.refactor.categorize  Creates new category NODE values from NODE values in the graph with the specified sourceKey as one of its property keys. The new category NODE values are then connected to the original NODE values with a RELATIONSHIP of the given type.	Procedure
apoc.refactor.cloneNodes  Clones the given NODE values with their labels and properties. It is possible to skip any NODE properties using skipProperties (note: this only skips properties on NODE values and not their RELATIONSHIP values).	Procedure
apoc.refactor.cloneSubgraph  Clones the given NODE values with their labels and properties (optionally skipping any properties in the skipProperties LIST<STRING> via the config MAP), and clones the given RELATIONSHIP values. If no RELATIONSHIP values are provided, all existing RELATIONSHIP values between the given NODE values will be cloned.	Procedure
apoc.refactor.cloneSubgraphFromPaths  Clones a sub-graph defined by the given LIST<PATH> values. It is possible to skip any NODE properties using the skipProperties LIST<STRING> via the config MAP .	Procedure
apoc.refactor.collapseNode  Collapses the given NODE and replaces it with a RELATIONSHIP of the given type.	Procedure
apoc.refactor.extractNode  Expands the given RELATIONSHIP VALUES into intermediate NODE VALUES. The intermediate NODE values are connected by the given outType and inType .	Procedure
apoc.refactor.from  Redirects the given RELATIONSHIP to the given start NODE .	Procedure
apoc.refactor.invert  Inverts the direction of the given RELATIONSHIP .	Procedure
apoc.refactor.mergeNodes  Merges the given LIST<NODE> onto the first NODE in the LIST<NODE> . All RELATIONSHIP values are merged onto that NODE as well.	Procedure

Qualified Name	Type
apoc.refactor.mergeRelationships  Merges the given <code>LIST<RELATIONSHIP></code> onto the first <code>RELATIONSHIP</code> in the <code>LIST<RELATIONSHIP></code> .	Procedure
apoc.refactor.normalizeAsBoolean  Refactors the given property to a <code>BOOLEAN</code> .	Procedure
apoc.refactor.rename.label  Renames the given label from <code>oldLabel</code> to <code>newLabel</code> for all <code>NODE</code> values. If a <code>LIST<NODE></code> is provided, the renaming is applied to the <code>NODE</code> values within this <code>LIST<NODE></code> only.	Procedure
apoc.refactor.rename.nodeProperty  Renames the given property from <code>oldName</code> to <code>newName</code> for all <code>NODE</code> values. If a <code>LIST<NODE></code> is provided, the renaming is applied to the <code>NODE</code> values within this <code>LIST<NODE></code> only.	Procedure
apoc.refactor.rename.type  Renames all <code>RELATIONSHIP</code> values with type <code>oldType</code> to <code>newType</code> . If a <code>LIST<RELATIONSHIP></code> is provided, the renaming is applied to the <code>RELATIONSHIP</code> values within this <code>LIST<RELATIONSHIP></code> only.	Procedure
apoc.refactor.rename.typeProperty  Renames the given property from <code>oldName</code> to <code>newName</code> for all <code>RELATIONSHIP</code> values. If a <code>LIST<RELATIONSHIP></code> is provided, the renaming is applied to the <code>RELATIONSHIP</code> values within this <code>LIST<RELATIONSHIP></code> only.	Procedure
apoc.refactor.setType  Changes the type of the given <code>RELATIONSHIP</code> .	Procedure
apoc.refactor.to  Redirects the given <code>RELATIONSHIP</code> to the given end <code>NODE</code> .	Procedure


apoc.rel

Qualified Name	Type
apoc.rel.id  Returns the id for the given virtual <code>RELATIONSHIP</code> .	Function
apoc.rel.type  Returns the type for the given virtual <code>RELATIONSHIP</code> .	Function

apoc.schema






Qualified Name	Type
apoc.schema.assert  Drops all other existing indexes and constraints when <code>dropExisting</code> is <code>true</code> (default is <code>true</code>). Asserts at the end of the operation that the given indexes and unique constraints are there.	Procedure
apoc.schema.node.constraintExists  Returns a <code>BOOLEAN</code> depending on whether or not a constraint exists for the given <code>NODE</code> label with the given property names.	Function
apoc.schema.node.indexExists  Returns a <code>BOOLEAN</code> depending on whether or not an index exists for the given <code>NODE</code> label with the given property names.	Function
apoc.schema.nodes  Returns all indexes and constraints information for all <code>NODE</code> labels in the database. It is possible to define a set of labels to include or exclude in the config parameters.	Procedure
apoc.schema.properties.distinct  Returns all distinct <code>NODE</code> property values for the given key.	Procedure
apoc.schema.properties.distinctCount  Returns all distinct property values and counts for the given key.	Procedure
apoc.schema.relationship.constraintExists  Returns a <code>BOOLEAN</code> depending on whether or not a constraint exists for the given <code>RELATIONSHIP</code> type with the given property names.	Function
apoc.schema.relationships  Returns the indexes and constraints information for all the relationship types in the database. It is possible to define a set of relationship types to include or exclude in the config parameters.	Procedure

apoc.scoring



Qualified Name	Type
apoc.scoring.existence  Returns the given score if true, 0 if false.	Function



Qualified Name	Type
apoc.scoring.pareto  Applies a Pareto scoring function over the given INTEGER values.	Function

apoc.search


Qualified Name	Type
apoc.search.multiSearchReduced  Returns a reduced representation of the NODE values found after a parallel search over multiple indexes. The reduced NODE values representation includes: node id, node labels, and the searched properties.	Procedure
apoc.search.node  Returns all the distinct NODE values found after a parallel search over multiple indexes.	Procedure
apoc.search.nodeAll  Returns all the NODE values found after a parallel search over multiple indexes.	Procedure
apoc.search.nodeAllReduced  Returns a reduced representation of the NODE values found after a parallel search over multiple indexes. The reduced NODE values representation includes: node id, node labels, and the searched properties.	Procedure
apoc.search.nodeReduced  Returns a reduced representation of the distinct NODE values found after a parallel search over multiple indexes. The reduced NODE values representation includes: node id, node labels, and the searched properties.	Procedure

apoc.spatial




Qualified Name	Type
apoc.spatial.geocode  Returns the geographic location (latitude, longitude, and description) of the given address using a geocoding service (default: OpenStreetMap).	Procedure
apoc.spatial.geocodeOnce  Returns the geographic location (latitude, longitude, and description) of the given address using a geocoding service (default: OpenStreetMap). This procedure returns at most one result.	Procedure

Qualified Name	Type
apoc.spatial.reverseGeocode  Returns a textual address from the given geographic location (latitude, longitude) using a geocoding service (default: OpenStreetMap). This procedure returns at most one result.	Procedure
apoc.spatial.sortByDistance  Sorts the given collection of PATH values by the sum of their distance based on the latitude/longitude values in the NODE values.	Procedure



apoc.stats














Qualified Name	Type
apoc.stats.degrees  Returns the percentile groupings of the degrees on the NODE values connected by the given RELATIONSHIP types.	Procedure

apoc.temporal













Qualified Name	Type
apoc.temporal.format  Formats the given temporal value into the given time format.	Function
apoc.temporal.formatDuration  Formats the given duration into the given time format.	Function
apoc.temporal.toZonedTemporal  Parses the given date STRING using the specified format into the given time zone.	Function








apoc.text

Qualified Name	Type
apoc.text.base64Decode  Decodes the given Base64 encoded STRING .	Function
apoc.text.base64Encode  Encodes the given STRING with Base64.	Function





Qualified Name	Type
apoc.text.base64UrlDecode  Decodes the given Base64 encoded URL.	Function
apoc.text.base64UrlEncode  Encodes the given URL with Base64.	Function
apoc.text.byteCount  Returns the size of the given STRING in bytes.	Function
apoc.text.bytes  Returns the given STRING as bytes.	Function
apoc.text.camelCase  Converts the given STRING to camel case.	Function
apoc.text.capitalize  Capitalizes the first letter of the given STRING .	Function
apoc.text.capitalizeAll  Capitalizes the first letter of every word in the given STRING .	Function
apoc.text.charAt  Returns the INTEGER value of the character at the given index.	Function
apoc.text.clean  Strips the given STRING of everything except alpha numeric characters and converts it to lower case.	Function
apoc.text.code  Converts the INTEGER value into a STRING .	Function
apoc.text.compareCleaned  Compares two given STRING values stripped of everything except alpha numeric characters converted to lower case.	Function
apoc.text.decapitalize  Turns the first letter of the given STRING from upper case to lower case.	Function
apoc.text.decapitalizeAll  Turns the first letter of every word in the given STRING to lower case.	Function





Qualified Name	Type
apoc.text.distance Compares the two given STRING values using the Levenshtein distance algorithm.	Function
apoc.text.doubleMetaphone Returns the double metaphone phonetic encoding of all words in the given STRING value.	Function
apoc.text.format Formats the given STRING with the given parameters.	Function
apoc.text.fuzzyMatch Performs a fuzzy match search of the two given STRING values.	Function
apoc.text.hammingDistance Compares the two given STRING values using the Hamming distance algorithm.	Function
apoc.text.hexCharAt Returns the hexadecimal value of the given STRING at the given index.	Function
apoc.text.hexValue Returns the hexadecimal value of the given value.	Function
apoc.text.indexOf Returns the first occurrence of the lookup STRING in the given STRING , or -1 if not found.	Function
apoc.text.indexesOf Returns all occurrences of the lookup STRING in the given STRING , or an empty list if not found.	Function
apoc.text.jaroWinklerDistance Compares the two given STRING values using the Jaro-Winkler distance algorithm.	Function
apoc.text.join Joins the given STRING values using the given delimiter.	Function
apoc.text.levenshteinDistance Compares the given STRING values using the Levenshtein distance algorithm.	Function
apoc.text.levenshteinSimilarity Returns the similarity (a value within 0 and 1) between the two given STRING values based on the Levenshtein distance algorithm.	Function

Qualified Name	Type
apoc.text.lpad  Left pads the given STRING by the given width.	Function
apoc.text.phonetic  Returns the US_ENGLISH phonetic soundex encoding of all words of the STRING .	Function
apoc.text.phoneticDelta  Returns the US_ENGLISH soundex character difference between the two given STRING values.	Procedure
apoc.text.random  Generates a random STRING to the given length using a length parameter and an optional STRING of valid characters. Unsuitable for cryptographic use-cases.	Function
apoc.text.regexGroups  Returns all groups matching the given regular expression in the given text.	Function
apoc.text.regreplace  Finds and replaces all matches found by the given regular expression with the given replacement.	Function
apoc.text.repeat  Returns the result of the given item multiplied by the given count.	Function
apoc.text.replace  Finds and replaces all matches found by the given regular expression with the given replacement.	Function
apoc.text.rpad  Right pads the given STRING by the given width.	Function
apoc.text.slug  Replaces the whitespace in the given STRING with the given delimiter.	Function
apoc.text.snakeCase  Converts the given STRING to snake case.	Function
apoc.text.sorensenDiceSimilarity  Compares the two given STRING values using the Sørensen–Dice coefficient formula, with the provided IETF language tag.	Function


Qualified Name	Type
apoc.text.split  Splits the given STRING using a given regular expression as a separator.	Function
apoc.text.swapCase  Swaps the cases in the given STRING .	Function
apoc.text.toCypher  Converts the given value to a Cypher property STRING .	Function
apoc.text.toUpperCase  Converts the given STRING to upper case.	Function
apoc.text.upperCamelCase  Converts the given STRING to upper camel case.	Function
apoc.text.urldecode  Decodes the given URL encoded STRING .	Function
apoc.text.urlencode  Encodes the given URL STRING .	Function

apoc.util


Qualified Name	Type
apoc.util.md5  Returns the MD5 checksum of the concatenation of all STRING values in the given LIST<ANY> . MD5 is a weak hashing algorithm which is unsuitable for cryptographic use-cases.	Function
apoc.util.sha1  Returns the SHA1 of the concatenation of all STRING values in the given LIST<ANY> . SHA1 is a weak hashing algorithm which is unsuitable for cryptographic use-cases.	Function
apoc.util.sha256  Returns the SHA256 of the concatenation of all STRING values in the given LIST<ANY> .	Function
apoc.util.sha384  Returns the SHA384 of the concatenation of all STRING values in the given LIST<ANY> .	Function

Qualified Name	Type
apoc.util.sha512  Returns the SHA512 of the concatenation of all STRING values in the LIST<ANY> .	Function
apoc.util.sleep  Causes the currently running Cypher to sleep for the given duration of milliseconds (the transaction termination is honored).	Procedure
apoc.util.validate  If the given predicate is true an exception is thrown.	Procedure
apoc.util.validatePredicate  If the given predicate is true an exception is thrown, otherwise it returns true (for use inside WHERE subclauses).	Function

apoc.warmup

Qualified Name	Type
apoc.warmup.run  Loads all NODE and RELATIONSHIP values in the database into memory.	Procedure Deprecated

apoc.xml

Qualified Name	Type
apoc.xml.parse  Parses the given XML STRING as a MAP .	Function

Customer Metrics Integration (CMI)

AuraDB Virtual Dedicated Cloud AuraDS Enterprise AuraDB Business Critical

An application performance monitoring system can be configured to fetch metrics of AuraDB instances of types:

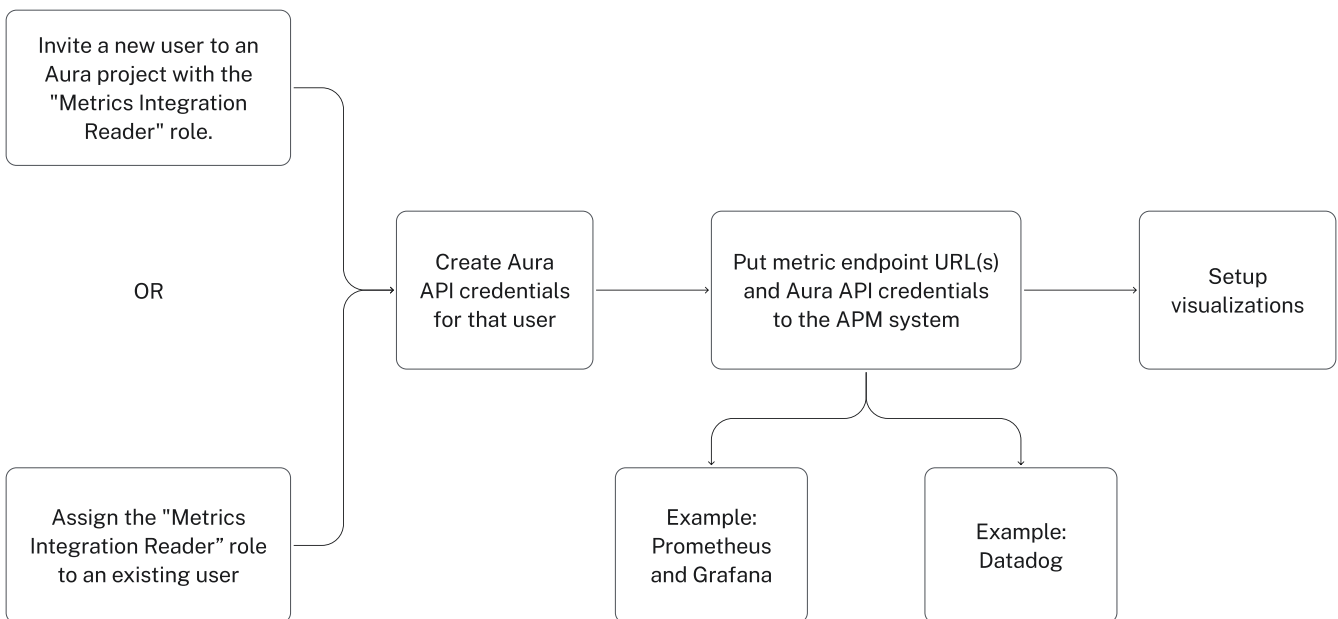
- AuraDB Virtual Dedicated Cloud
- AuraDS Enterprise
- AuraDB Business Critical

This gives users access to their Neo4j Aura instance metric data for monitoring purposes.

Analyzing the metrics data allows users to:

- Optimize their Neo4j load
- Adjust Aura instance sizing
- Set up notifications

Process overview



Detailed steps

1. Log in to Aura as project admin.
2. Make sure there is a dedicated Aura user to use for fetching metrics. You can either:
 - Create a new user:
 - i. In "[User Management](#)" of Neo4j Aura, invite a new user, selecting "Metrics Integration Reader" as a role.

Invite User



Email

Role



Cancel

Invite

- ii. Follow the invitation link and log in to Neo4j Aura.
- iii. Confirm the project membership.
- ° Or you can find an existing user in "[User Management](#)" and change its role to "Metrics Integration Reader"




Capabilities of users with the role "Metrics Integration Reader" are limited to fetching the metrics and getting a read-only view of the project.

3. Ensure you are logged in to Aura as the user selected in the previous step. In "[Account Details](#)", create new Aura API credentials. Save client secret.

Create Aura API Credentials



Client ID

KoIX3jAmanRzrI6QdAdluuvvgB6k5B4Dh 

Client Secret

MH1PmdOUUnY2eB1VwqegpYzOFGpHEWYmliUugpHKZfa1Ctm8GHyxZYfmxfm 



After closing this window, you will no longer be able to access your client secret. We recommend either securely storing it or downloading it for safekeeping.

Download

Done

4. Configure the APM system to fetch metrics from the URL(s) or configuration templates shown in "[Metrics Integration](#)" of Neo4j Aura. Use `oauth2` type of authentication specifying the Client ID and Client Secret created in the previous step. See examples for [Prometheus and Grafana](#) and [Datadog](#) below.
5. Use the APM system to create visualizations, dashboards, and alarms based on Neo4j metrics.

Security

Metrics for a Neo4j Aura instance are only returned if all the following are true:

- `Authorization` header of the metrics request contains a valid token.
- The token was issued for an Aura user with "Metrics Integration Reader" role.
- Project has instances of types `Enterprise (Virtual Dedicated Cloud)` or `Business Critical`.
- The specified instance belongs to the specified project.

```
<!-- vale Neo4j.ProductDeprecations = NO -->
```



The legacy term `Enterprise` is still used within the codebase and API. However, in the Aura console and documentation, the AuraDB Enterprise project type is now known as AuraDB Virtual Dedicated Cloud.

```
<!-- vale Neo4j.ProductDeprecations = YES -->
```

Revoke access to metrics

To revoke a user's access to metrics of a specific project, remove the user from that project in "[User](#)"

Management". After that, the user still exists but its connection to the project is removed.



The revocation described takes effect after the authorization caches expire, which takes approximately 5 minutes. It results in HTTP 401 being returned, along with the message **User doesn't have access to Metrics resources**. However, if you remove only the Aura API credentials used to retrieve metrics, the revocation will take effect only after the tokens issued with these credentials expire, as no new token can be issued anymore. Currently used token expiration time is 1 hour.

Metric labels

Depending on the metric, the following labels are applied:

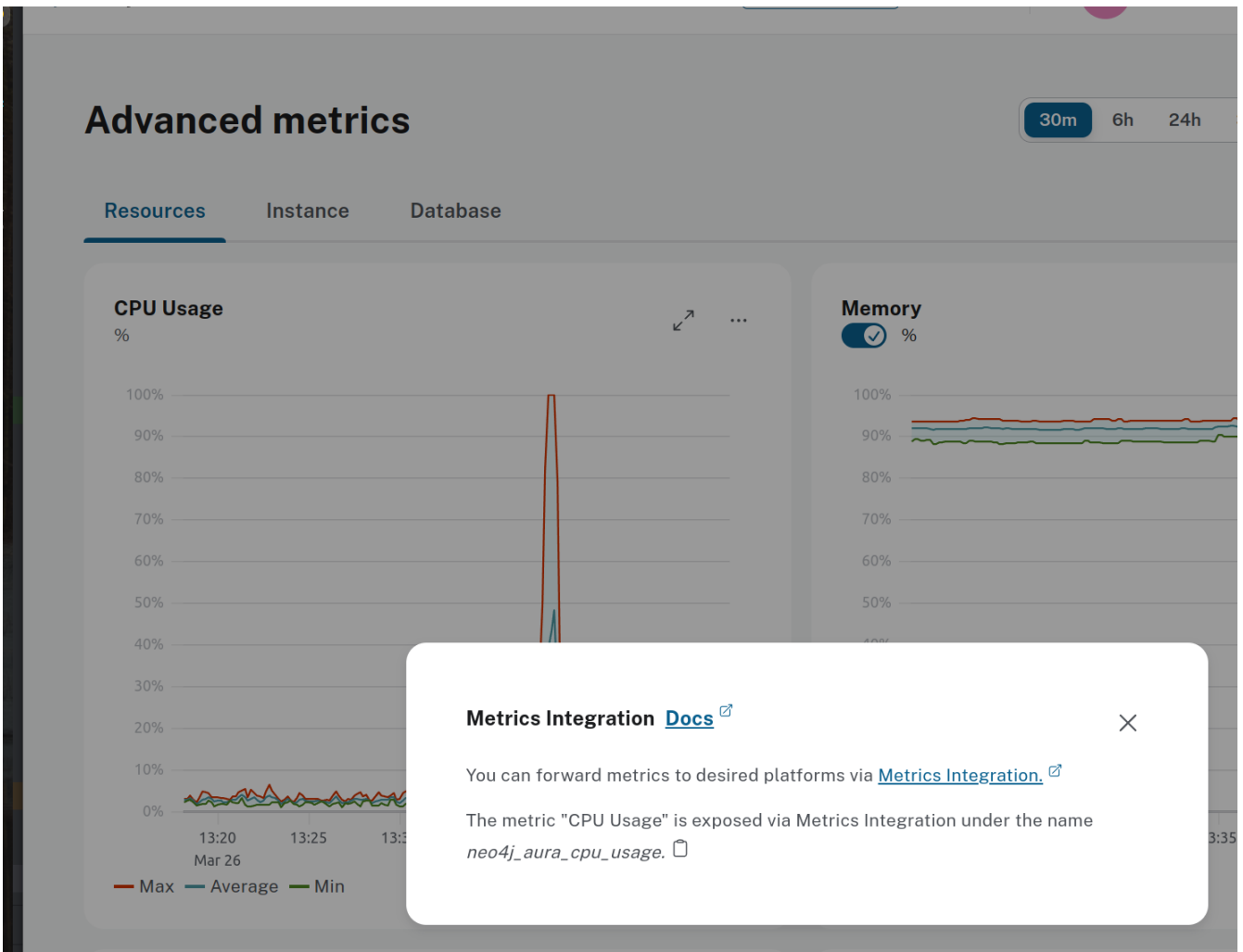
- **aggregation**: the aggregation used to calculate the metric value, set on every metric. Since the Neo4j instance is deployed as a Neo4j cluster, aggregations are performed to combine values from all relevant cluster nodes. The following aggregations are used: **MIN**, **MAX**, **AVG** and **SUM**.
- **instance_id**: the Aura instance ID the metric is reported for, set on every metric.
- **database**: the name of the Neo4j database the metric is reported for. Set to **neo4j** by default.

Example

```
# HELP neo4j_database_count_node The total number of nodes in the database.
# TYPE neo4j_database_count_node gauge
neo4j_database_count_node{aggregation="MAX",database="neo4j",instance_id="78e7c3e0"} 778114.000000
1711462853000
```

Looking up metric name in Neo4j Aura Advanced Metrics

In Neo4j Aura Advanced Metrics, it is possible to find out the metric name that corresponds to the chart, by using the chart menu item "Metrics Integration" as shown.



Metric scrape interval

Recommended scrape interval for metrics is in the range of 30 seconds up to 2 minutes, depending on requirements. The metrics endpoint caches metrics for 30 seconds.

Example using Prometheus

Install Prometheus

One way is to get a tarball from <https://prometheus.io/docs/prometheus/latest/installation/>

Configure Prometheus

To monitor one or more instances, add a section to the Prometheus configuration file `prometheus.yml`.

Copy the configuration section proposed in [Metrics Integration](#), as shown.



Replace the placeholders `<AURA_CLIENT_ID>` and `<AURA_CLIENT_SECRET>` with corresponding values created in the previous step.

Metrics integration [Docs](#)

Endpoint URL for tenant:

`https://customer-metrics-api-devcmi.neo4j-dev.io/api/46f476d2-eea2-5008-b236-6e3e78020d8a/metrics`

Endpoint URL for instances:

Cmitest1 `https://customer-metrics-api-devcmi.neo4j-dev.io/api/46f476d2-eea2-5008-b236-6e3e78020d8a/565eda86/metrics`

Endpoint authentication:

Accessing metric endpoints requires Aura API credentials. You can find and create them [here](#).

Prometheus job configuration

```
global:
  scrape_interval: '1m'
  scrape_timeout: '10s'
  evaluation_interval: '5s'
scrape_configs:
- job_name: 'aura-metrics'
  metrics_path: '/api/46f476d2-eea2-5008-b236-6e3e78020d8a/metrics'
  scheme: 'https'
  static_configs:
  - targets: ['customer-metrics-api-devcmi.neo4j-dev.io']
  oauth2:
    client_id: '<AURA_CLIENT_ID>'
    client_secret: '<AURA_CLIENT_SECRET>'
    token_url: 'https://api-devcmi.neo4j-dev.io/oauth/token'
```

For details, see [Prometheus configuration reference](#).

Start Prometheus

```
./prometheus --config.file=prometheus.yml
```

Test that metrics are fetched

Open <http://localhost:9090> and enter a metric name or expression in the search field (ex. `neo4j_aura_cpu_usage`).

Use Grafana

Install and configure Grafana, adding the endpoint of the Prometheus instance configured in the previous step as a data source. You can create visualizations, dashboards, and alarms based on Neo4j metrics.

Example using Datadog

Configure an endpoint with token authentication

Edit `/etc/datadog-agent/conf.d/openmetrics.d/conf.yaml` as follows:



Replace the placeholders `<ENDPOINT_URL>`, `<AURA_CLIENT_ID>` and `<AURA_CLIENT_SECRET>` with corresponding values from the previous steps.

`/etc/datadog-agent/conf.d/openmetrics.d/conf.yaml`

```
init_config:
instances:
- openmetrics_endpoint: <ENDPOINT_URL>
  metrics:
  - neo4j_.*
  auth_token:
  reader:
    type: oauth
    url: https://api.neo4j.io/oauth/token
    client_id: <AURA_CLIENT_ID>
    client_secret: <AURA_CLIENT_SECRET>
  writer:
    type: header
    name: Authorization
    value: "Bearer <TOKEN>"
```

For details, see [Datadog Agent documentation](#) and [configuration reference](#).

Test that metrics are fetched

- `sudo systemctl restart datadog-agent`
- Watch `/var/log/datadog/*` to see if fetching metrics happens or if there are warnings regarding parsing the configuration.
- Check in Datadog metric explorer to see if metrics appear (after a couple of minutes).

Programmatic support

Aura API for Metrics Integration

- Aura API supports fetching metrics integration endpoints using:
 - endpoint `/tenants/{tenantId}/metrics-integration` (for project metrics)
 - JSON property `metrics_integration_url` as part of `/instances/{instanceId}` response (for instance metrics)
- Reference: [Aura API Specification](#)



Project replaces Tenant in the console UI and documentation. However, in the API, `tenant` remains the nomenclature.

Aura CLI for Metrics Integration

- Aura CLI has a subcommand for `tenants` command to fetch project metrics endpoint:

```
aura projects get-metrics-integration --tenant-id <YOUR_PROJECT_ID>

# example output
{
  endpoint: "https://customer-metrics-api.neo4j.io/api/v1/<YOUR_PROJECT_ID>/metrics"
}

# extract endpoint
aura projects get-metrics-integration --project-id <YOUR_PROJECT_ID> | jq '.endpoint'
```

- For instance metrics endpoint, Aura CLI `instances get` command JSON output includes a new property `metrics_integration_url`:

```
aura instances get --instance-id <YOUR_INSTANCE_ID>

# example output
{
  "id": "id",
  "name": "Production",
  "status": "running",
  "tenant_id": "YOUR_PROJECT_ID",
  "cloud_provider": "gcp",
  "connection_url": "YOUR_CONNECTION_URL",
  "metrics_integration_url": "https://customer-metrics-
api.neo4j.io/api/v1/<YOUR_PROJECT_ID>/<YOUR_INSTANCE_ID>/metrics",
  "region": "europe-west1",
  "type": "enterprise-db",
  "memory": "8GB",
  "storage": "16GB"
}

# extract endpoint
aura instances get --instance-id <YOUR_INSTANCE_ID> | jq '.metrics_integration_url'
```

- Reference: [Aura CLI cheatsheet](#)

Metrics granularity

The metrics returned by the integration endpoint are grouped based on the labels provided: `aggregation`, `instance_id`, and `database`.

An Aura instance typically runs on multiple servers to achieve availability and workload scalability. These servers are deployed across different Cloud Provider availability zones in the user-selected region.

Metrics Integration supports a more granular view of the Aura instance metrics with additional data points for availability zone & instance mode combinations. This view can be enabled on demand.



Contact [Customer Support](#) to enable more granular metrics of instances for your project.



There may be a delay in more granular metrics being available when a new Aura instance is created. This is because of the way 'availability zone' data is collected.

Example metric data points

```
neo4j_aura_cpu_usage{aggregation="MAX",instance_id="a59d71ae",availability_zone="eu-west-1a",instance_mode="PRIMARY"} 0.025457 1724245310000
neo4j_aura_cpu_usage{aggregation="MAX",instance_id="a59d71ae",availability_zone="eu-west-1b",instance_mode="PRIMARY"} 0.047088 1724245310000
neo4j_aura_cpu_usage{aggregation="MAX",instance_id="a59d71ae",availability_zone="eu-west-1c",instance_mode="PRIMARY"} 0.021874 1724245310000
```

Additional metric labels

- `availability_zone` - User selected Cloud provider zone.
- `instance_mode` - `PRIMARY` based on user selected workload requirement of reads and writes. (Minimum 3 primaries per instance)

Usage

The following is an example of gaining more insights into your Aura instance CPU usage for capacity planning:

Example PromQL query to plot

```
max by(availability_zone) (neo4j_aura_cpu_usage{instance_mode="PRIMARY"}) / sum by(availability_zone) (neo4j_aura_cpu_limit{instance_mode="PRIMARY"})
```

CPU Usage by Primaries in AZ

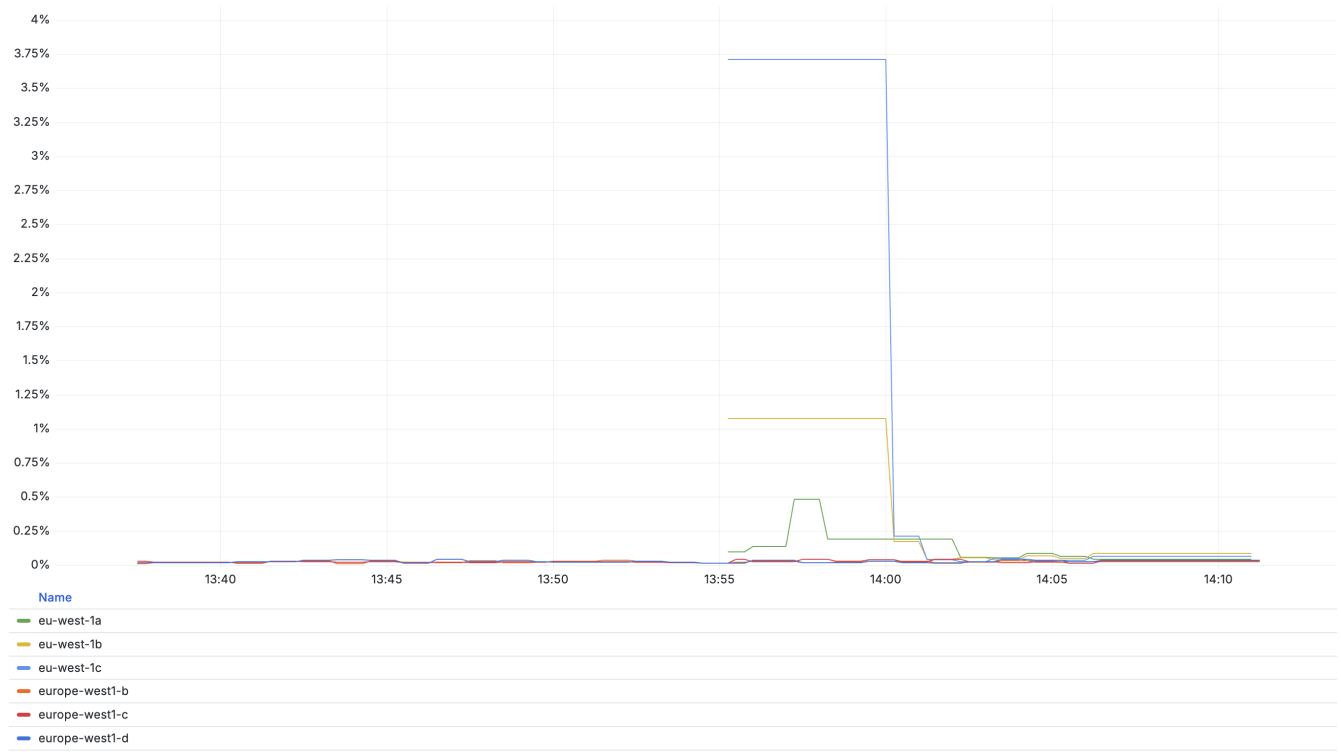


Figure 13. Chart shows CPU usage of primaries by availability zone

Metric definitions

Out of Memory Errors

Metric name	neo4j_aura_out_of_memory_errors_total
Description	The total number of Out of Memory errors for the instance. Consider increasing the size of the instance if any OOM errors.
Metric type	Counter
Default aggregation	SUM

CPU Available

Metric name	neo4j_aura_cpu_limit
Description	The total CPU cores assigned to the instance nodes.
Metric type	Gauge
Default aggregation	MAX

CPU Usage

Metric name	neo4j_aura_cpu_usage
Description	CPU usage (cores). CPU is used for planning and serving queries. If this metric is constantly spiking or at its limits, consider increasing the size of your instance.
Metric type	Gauge
Default aggregation	MAX

Storage Total

Metric name	neo4j_aura_storage_limit
Description	The total disk storage assigned to the instance.
Metric type	Gauge
Default aggregation	MAX

Heap Used

Metric name	neo4j_dbms_vm_heap_used_ratio
Description	The percentage of configured heap memory in use. The heap space is used for query execution, transaction state, management of the graph etc. The size needed for the heap is very dependent on the nature of the usage of Neo4j. For example, long-running queries, or very complicated queries, are likely to require a larger heap than simpler queries. To improve performance, the heap should be large enough to sustain concurrent operations. This value should not exceed 80% for long periods, short spikes can be normal. In case of performance issues, you may have to tune your queries and monitor their memory usage, to determine whether the heap needs to be increased. If the workload of Neo4j and performance of queries indicates that more heap space is required, consider increasing the size of your instance. This helps avoid unwanted pauses for garbage collection.
Metric type	Gauge
Default aggregation	MAX

Page Cache Usage Ratio

Metric name	neo4j_dbms_page_cache_usage_ratio
Description	The percentage of the allocated page cache in use. If this is close to or at 100%, then it is likely that the hit ratio will start dropping, and you should consider increasing the size of your instance so that more memory is available for the page cache.
Metric type	Gauge
Default aggregation	MIN

Bolt Connections Running

Metric name	neo4j_dbms_bolt_connections_running
Description	The total number of Bolt connections that are currently executing Cypher transactions and returning results. This is a set of snapshots over time and may appear to spike if workloads are all completed quickly.

Metric type	Gauge
Default aggregation	MAX

Bolt Connections Idle

Metric name	neo4j_dbms_bolt_connections_idle
Description	The total number of Bolt connections that are connected to the Aura database but not currently executing Cypher or returning results.
Metric type	Gauge
Default aggregation	MAX

Bolt Connections Closed

Metric name	neo4j_dbms_bolt_connections_closed_total
Description	The total number of Bolt connections closed since startup. This includes both properly and abnormally ended connections. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Bolt Connections Opened

Metric name	neo4j_dbms_bolt_connections_opened_total
Description	The total number of Bolt connections opened since startup. This includes both successful and failed connections. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Garbage Collection Young Generation

Metric name	neo4j_dbms_vm_gc_time_g1_young_generation_total
Description	Shows the total time since startup spent clearing up heap space for short lived objects. Young garbage collections typically complete quickly, and the Aura instance waits while the garbage collector is run. High values indicate that the instance is running low on memory for the workload and you should consider increasing the size of your instance.
Metric type	Counter
Default aggregation	MAX

Garbage Collection Old Generation

Metric name	neo4j_dbms_vm_gc_time_g1_old_generation_total
Description	Shows the total time since startup spent clearing up heap space for long-lived objects. Old garbage collections can take time to complete, and the Aura instance waits while the garbage collector is run. High values indicate that there are long-running processes or queries that could be optimized, or that your instance is running low on CPU or memory for the workload and you should consider reviewing these metrics and possibly increasing the size of your instance.

Metric type	Counter
Default aggregation	MAX

Replan Events

Metric name	neo4j_database_cypher_replan_events_total
Description	The total number of times Cypher has replanned a query since the server started. If this spikes or is increasing, check that the queries executed are using parameters correctly. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Active Read Transactions

Metric name	neo4j_database_transaction_active_read
Description	The number of currently active read transactions.
Metric type	Gauge
Default aggregation	MAX

Active Write Transactions

Metric name	neo4j_database_transaction_active_write
Description	The number of active write transactions.
Metric type	Gauge
Default aggregation	MAX

Committed Transactions

Metric name	neo4j_database_transaction_committed_total
Description	The total number of committed transactions since the server was started. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Peak Concurrent Transactions

Metric name	neo4j_database_transaction_peak_concurrent_total
Description	The highest number of concurrent transactions detected since the server started. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Transaction Rollbacks

Metric name	<code>neo4j_database_transaction_rollbacks_total</code>
Description	The total number of rolled-back transactions. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Checkpoint Events

Metric name	<code>neo4j_database_check_point_events_total</code>
Description	The total number of checkpoint events executed since the server started. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Checkpoint Events Cumulative Time

Metric name	<code>neo4j_database_check_point_total_time_total</code>
Description	The total time in milliseconds spent in checkpointing since the server started. This value may drop if background maintenance is performed by Aura.
Metric type	Counter
Default aggregation	MAX

Last Checkpoint Duration

Metric name	<code>neo4j_database_check_point_duration</code>
Description	The duration of the last checkpoint event. Checkpoints should typically take several seconds to several minutes. Values over 30 minutes warrant investigation.
Metric type	Gauge
Default aggregation	MAX

Relationships

Metric name	<code>neo4j_database_count_relationship</code>
Description	The total number of relationships in the database.
Metric type	Gauge
Default aggregation	MAX

Nodes

Metric name	<code>neo4j_database_count_node</code>
Description	The total number of nodes in the database.
Metric type	Gauge
Default aggregation	MAX

Store Size Database

Metric name	<code>neo4j_database_store_size_database</code>
Description	Amount of disk space reserved to store user database data, in bytes. Ideally, the database should all fit into memory (page cache) for the best performance. Keep an eye on this metric to make sure you have enough storage for today and for future growth. Check this metric with page cache usage to see if the data is too large for the memory and consider increasing the size of your instance in this case.
Metric type	Gauge
Default aggregation	MAX

Page Cache Evictions

Metric name	<code>neo4j_dbms_page_cache_evictions_total</code>
Description	The number of times data in memory is being replaced in total. A spike can mean your workload is exceeding the instance's available memory, and you may notice a degradation in performance or query execution errors. Consider increasing the size of your instance to improve performance if this metric remains high.
Metric type	Counter
Default aggregation	MAX

Successful Query Executions

Metric name	<code>neo4j_db_query_execution_success_total</code>
Description	The total number of successful queries executed on this database.
Metric type	Counter
Default aggregation	SUM

Query Execution Failures

Metric name	<code>neo4j_db_query_execution_failure_total</code>
Description	The total number of failed queries executed on this database.
Metric type	Counter
Default aggregation	SUM

Query Latency 99th Percentile

Metric name	<code>neo4j_db_query_execution_internal_latency_q99</code>
Description	The query execution time in milliseconds where 99% of queries executed faster than the reported time.
Metric type	Gauge
Default aggregation	MAX

Query Latency 75th Percentile

Metric name	neo4j_db_query_execution_internal_latency_q75
Description	The query execution time in milliseconds where 75% of queries executed faster than the reported time.
Metric type	Gauge
Default aggregation	MAX

Query Latency 50th Percentile

Metric name	neo4j_db_query_execution_internal_latency_q50
Description	The query execution time in milliseconds where 50% of queries executed faster than the reported time. This also corresponds to the median of the query execution time.
Metric type	Gauge
Default aggregation	MAX

Last Committed Transaction ID

Metric name	neo4j_database_transaction_last_committed_tx_id_total
Description	The id of the last committed transaction. Track this for primary cluster members of your Aura instance. It should show overlapping, ever-increasing lines and if one of the lines levels off or falls behind, it is clear that this cluster member is no longer replicating data, and action is needed to rectify the situation.
Metric type	Counter
Default aggregation	MAX

Logging

Request and download logs

Aura allows you to request and download security and query logs.

You can access logs from an Aura instance via the **Logs** tab.

To access the **Logs** tab:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to export the logs from.
3. Select the **Logs** tab.

Query logs

A query log provides a log of queries executed on an instance within a specified time range.



Queries that complete successfully within 50 ms are not logged.

Requesting query logs

To request a query log from the **Logs** tab:

1. Click **Request log**.
2. Select the **Query Log** option under **Type**.
3. Select a **Range** option.
4. Click **Request**.



Requested logs will appear for up to 7 days, at which point they will expire and be removed.

You can select from the following time ranges when requesting a query log:

- Last 15 minutes
- Last hour
- Custom range - Any range up to one hour from the previous 30 days



We recommend shorter time ranges for busy, read/write heavy instances to reduce request time.

Aura will generate a query log for your selected time range, available to download once the **Status** shows **Completed**.

Downloading query logs

You can download query logs by selecting the download icon on the right-hand side of the log entry.

Downloaded query logs take the form of a zipped JSON file that, when extracted, contains the following information:

Query log entries

Name	Description
<code>allocatedBytes</code>	The number of bytes allocated by the query.
<code>annotationData</code>	The metadata attached to the transaction.
<code>authenticatedUser</code>	The name of the user who executed the query (whose credentials were used to log in).
<code>database</code>	The name of the database the query was executed on.
<code>dbid</code>	The ID of the instance the query was executed on.
<code>elapsedTimeMs</code>	The time the query took to complete in milliseconds.
<code>event</code>	The query event: <code>start</code> - The query was successfully parsed, awaiting execution. <code>fail</code> - The query failed to either parse or execute. <code>success</code> - The query executed successfully or was canceled.
<code>executingUser</code>	The name of the user who executed the query either through authentication (<code>authenticatedUser</code>) or through <code>impersonation</code> .
<code>id</code>	The ID of the query.
<code>message</code>	The log message: a truncated version of <code>query</code> .
<code>pageFaults</code>	The number of page faults resulting from the query.
<code>pageHits</code>	The number of page hits resulting from the query.
<code>query</code>	The full query text.
<code>runtime</code>	The <code>Cypher runtime</code> used to execute the query.
<code>type</code>	The type of log message.
<code>time</code>	The timestamp of the log message.

Security logs

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

A security log provides a log of all the security events that have occurred on an instance within a specified time range.

Security events include:

- Login attempts: both successful and unsuccessful.

- Authorization failures from role-based access control.
- [Administration commands](#) run against the `system` database.

Requesting security logs

To request a security log from the **Logs** tab:

1. Click **Request log**.
2. Select the **Security Log** option under **Type**.
3. Select a **Range** option.
4. Click **Request**.



Requested logs will appear for up to 7 days, at which point they will expire and be removed.

You can select from the following time ranges when requesting a security log:

- Last 6 hours
- Last 12 hours
- Custom range - Any range up to 12 hours from the previous 30 days

Aura will generate a security log for your selected time range, available to download once the **Status** shows **Completed**.

Downloading security logs

You can download security logs by selecting the download icon on the right-hand side of the log entry.

Downloaded security logs take the form of a zipped JSON file that, when extracted, contains the following information:

Security log entries

Name	Description
<code>authenticatedUser</code>	The name of the user who executed the security event (whose credentials were used to log in).
<code>dbid</code>	The ID of the instance the security event occurred on.
<code>executingUser</code>	The name of the user who executed the security event either through authentication (<code>authenticatedUser</code>) or through <code>impersonation</code> .
<code>message</code>	The log message.
<code>type</code>	The type of log message.
<code>time</code>	The timestamp of the log message.

Security log forwarding

[AuraDB Virtual Dedicated Cloud](#) [AuraDS Enterprise](#)

With security log forwarding, you can stream security logs directly to a cloud project owned by your organization, in real time.

To access **Log forwarding**:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select **Log forwarding** from the sidebar menu.

This will display a list of currently configured log forwarding processes for the active project.

If no log forwarding process is set up, a button to do so is displayed in the center of the page.



A log forwarding process is scoped to a specific product and region combination, and limited to one for each.

Set up log forwarding



Aura Database and Analytics services are business critical for our users. We have requests to introduce more capabilities enabling access to logs and metrics to derive actionable insights using your choice of monitoring platform.

We have a strong roadmap of observability sharing features including security logs, query logs and other capabilities. Many of these logs can be of significant size hence we will introduce in the future a new consumption based billing model including cloud egress costs.

We believe security is of paramount importance hence we have decided to make security logs available for you initially at no extra charge.

The complete steps for setting up log forwarding depends on the chosen cloud provider.

Exhaustive instructions are provided in the wizard which appears by following the steps below.

1. Navigate to the **Log forwarding** page as described above.
2. Click **Create new log forwarding process**.
3. Follow the instructions specific to your cloud provider.

Output destination

Log forwarding can forward logs to the log service of the same cloud provider as the monitored instance is located in.

Cross-region log forwarding is supported.

If your instance is in:

- **Google Cloud Platform** - Forward logs to Google Cloud Logging in your own GCP project.
- **Amazon Web Services** - Forward logs to CloudWatch in your own AWS account.
- **Azure** - Forward logs to a Log Analytics workspace in your own Azure subscription.

Logs can be further forwarded into third party systems using the log routing capabilities provided by your cloud provider.

Query log analyzer

[AuraDB Professional](#) [AuraDB Business Critical](#) [AuraDB Virtual Dedicated Cloud](#)

Query log analyzer is a feature that provides a UI to review the queries executed on an Aura instance.

To access Query log analyzer:

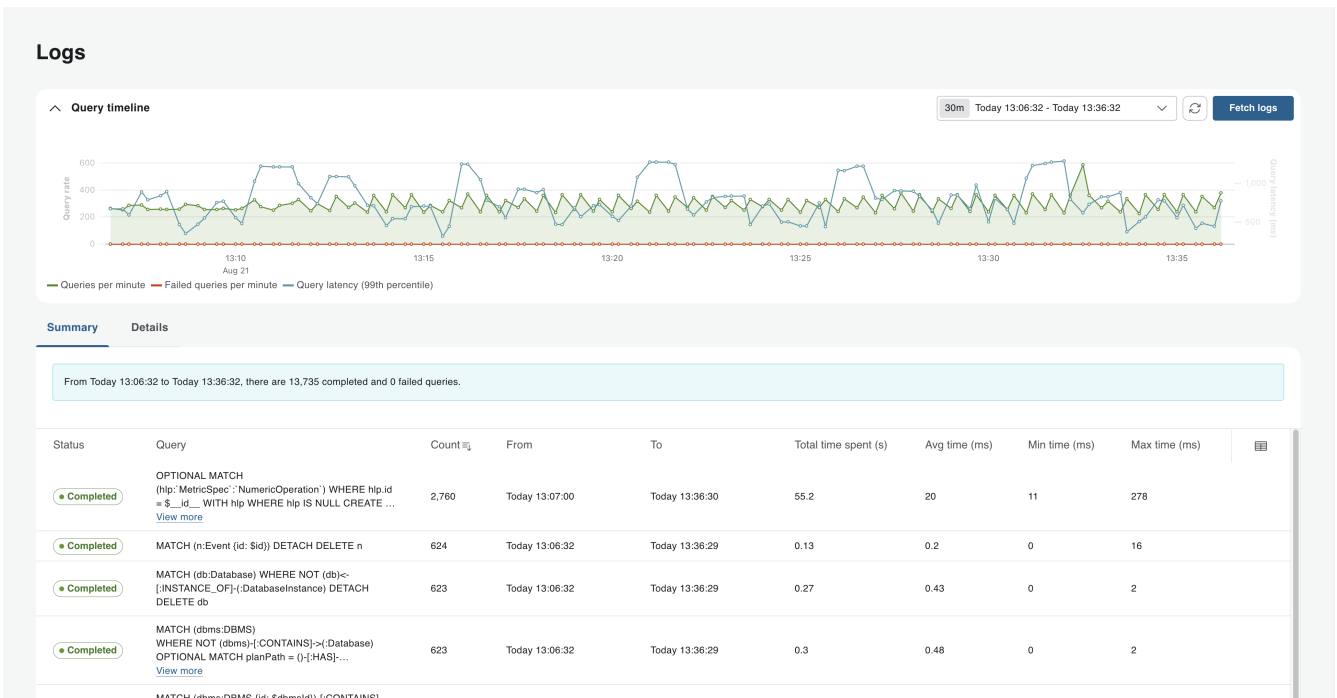
1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to access.
3. Select the Logs tab.
4. Select the Query log analyzer button.

It is also possible to enter Query log analyzer from the Query rate or Query latency charts in the Database tab of Advanced Metrics. To do so, click the ellipsis button (...) on the chart and select Explore query logs.

Overview

Query log analyzer is split up in three parts:

- **Query timeline** - Timeline showing metrics for number of queries, failed queries and query latency.
- **Summary table** - An aggregated view of query logs, giving a high level overview over the selected time period.
- **Details table** - A detailed view showing individual query executions in the selected time period.



To fetch logs, first choose a time range in the Query timeline. With a time selection done, press the Fetch logs button. You may optionally choose any filters or search text if required, then press Go.

A summary of query executions is returned, showing aggregations per query. To see the individual query executions, click the right arrow at the end of the line to show details for that query. The details pane shows individual executions.

Query timeline

When viewing the query timeline, you can select from the following time intervals:

- 30 minutes
- Last hour
- Last 2 hours
- Last 6 hours
- Last 24 hours
- Last 3 days
- Last week

The query timeline can be collapsed by clicking on the header.

i

The query timeline may show activity from internal meta queries, which are filtered in the table.

Zoom

To zoom in to a narrower time interval, select and drag inside the timeline to select your desired time interval. The data in the timeline will automatically update to match the increased resolution. To update the

table, click the **Fetch logs** button.

To reset zoom, double-click anywhere inside the timeline.

Toggle data series

To hide or show individual data series, select the corresponding data series in the legend below the timeline.

Fetch logs



Query logs are available for a period of 7 days, and each request can be for up to 24 hours of data.

The **Fetch logs** button will open up a dialog where you can add filters and search before fetching the logs. The Query timeline determines the current time selection, which can be changed by closing the dialog and modifying the timeline. To fetch the logs after selection of filters and search is done, click the **Go** button.

Filters

To filter, click the filter button. This will load the available filters over the selected time period. Filters are available for the following fields:

- Status
- User
- Driver
- Application
- Initiation type

Search

To search, click the search button. Search can be specified for the **Query text** and the **Error text**. The fields are case-insensitive and allows you to find specific queries or error that are interesting.

Table interactions

Sort table

By default, the table will be sorted on **Count** for **Summary** and **Status** for **Details**. To sort by a column (such as **Max Time ms**) click on the column heading.

Modify columns

The columns in the table can be modified by clicking the button to the right of the column row. Columns can be enabled or disabled, or the order changed using the grid icon at the top right of the table.

Expand query

In the table three rows of query text will be shown. To see the whole query if the query is longer, press the **View more** button under the query text.

Neo4j connectors

Neo4j Connector for Apache Spark



Tutorial: [Using the Apache Spark Connector with Aura](#)

The Neo4j Connector for Apache Spark is intended to make integrating graphs with Spark easy. There are two ways to use the connector:

- As a [data source](#): read any set of nodes or relationships as a `DataFrame` in Spark
- As a [sink](#): write any `DataFrame` to Neo4j as a collection of nodes or relationships, or use a Cypher statement to process records contained in a `DataFrame` into the graph pattern of your choice

Connecting to Aura only requires to make a few changes to the [Neo4j driver configuration](#):

1. Replace the `bolt` URI (the value of the `neo4j.server.uri` configuration parameter) with the `neo4j+s://` connection URI from the Aura instance detail page
2. Update the username and password configuration parameters as appropriate

For more information check the [Neo4j Apache Spark Connector](#) page.

Neo4j Connector for Apache Kafka

Many users and customers want to integrate Kafka and other streaming solutions with Neo4j, either to ingest data into the graph from other sources or to send update events to the event log for later consumption. Aura supports the use of the [Kafka Connect Neo4j Connector](#), which allows you to ingest data into Neo4j from Kafka topics or send change events from Neo4j into Kafka topics.

Connecting to Aura only requires to make a few changes to the [source](#) and [sink](#) configuration examples:

- Replace the `bolt` URI in the examples (the value of the `neo4j.server.uri` configuration parameter) with the `neo4j+s://` connection URI from the Aura instance detail page
- Update the username and password configuration parameters as appropriate

For more information check the [Kafka Connect Neo4j Connector](#) user guide.

Neo4j Connector for BI



Tutorial: [Using the BI Connector with Aura](#)

The Neo4j Connector for Business Intelligence (BI) delivers access to Neo4j graph data from BI tools such as Tableau, Power BI, Looker, TIBCO, Spotfire Server, MicroStrategy, and more. It can be used to run SQL queries on a Neo4j graph and retrieve data in tabular format.

The connection to Aura requires the usage of the `SSL` parameter in the connection string. For example, if

the connection URI of the Aura instance is `neo4j+s://xxxxxxx.databases.neo4j.io`, the following connection strings must be used:

- With JDBC: `jdbc:neo4j://xxxxxxx.databases.neo4j.io?SSL=true` (note the usage of the `neo4j` protocol instead of `neo4j+s`)
- With ODBC: `Host=xxxxxxx.databases.neo4j.io;SSL=1`

The Neo4j Connector for BI can be downloaded from the [Download Center](#).

Aura API

Overview

The Aura API allows you to programmatically perform actions on your Aura instances without the need to log in to the Console.

A complete list of the available endpoints can be seen and tested in the [API Specification](#).



Before using the API, you must follow the steps outlined in [Authentication](#) to create your credentials and authenticate your requests.

API URL

Base URL

The base URL for the Aura API is <https://api.neo4j.io>.

Versioning

The current version of the Aura API is [v1](#)

As and when we need to introduce breaking changes to the API, we will release a new version to ensure we do not break existing integrations.

In the future, as we deprecate legacy API versions, we will provide notice. Once the expiry date for a deprecated version has passed, that version will no longer be available.

Example request

The following example shows how to use the base URL and versioning to make a request to the API:

```
GET https://api.neo4j.io/v1/instances
```

Retries

In the event of [5xx](#) server error responses, you may consider retrying the request after a delay if it is safe to do so. The response may include a [Retry-After](#) header with a suggestion of a suitable minimum delay before attempting to retry.

Rate limiting is set to 125 requests per minute.

You should consider your use of the Rate Limit before attempting to retry, and we recommend using an exponential backoff delay with a limited number of retries before giving up.

A request is only guaranteed to be safe to retry if it uses an idempotent HTTP method, such as [GET](#). If, for

example, you retry a request for creating an instance, you may end up with duplicate instances and end up being charged extra as a result.

In the case of **429 Too Many Requests**, we would recommend slowing down the rate of all requests sent from your client application and consider retrying with a suitable minimum delay and backoff strategy.

For other **4xx** client error responses, you should not resend such requests without first correcting them.

Request tracing and troubleshooting

An **X-Request-Id** response header is returned with each request and can be used for troubleshooting.

The value of this header contains a unique ID that can be used to track the journey of a request.

If you run into any issues with a particular request, you can [raise a support ticket](#) and provide the **X-Request-Id**.

Authentication

The Aura API uses OAuth 2.0 for API authentication.

Creating credentials



AuraDB Virtual Dedicated Cloud users, and AuraDS Enterprise users have unrestricted access to creating API credentials. However, users with Free and Professional instances must have entered billing information or be a member of a marketplace project before they can create API credentials.

1. Navigate to the [Neo4j Aura Console Account Details page](#) in your browser.
2. Select the **Create** button in the **Aura API Credentials** section.
3. Enter a **Client name**, and select **Create**.
4. Securely save the **Client ID** and **Client Secret** you are given in the resulting modal; you will need these for the next step.



You cannot retrieve your secret after you close the modal, so save it securely.

Authenticating API requests

To authenticate API requests to the [Aura API](#), you must provide a Bearer Token in the **Authorization** header of each request as shown below:

Authorization: `Bearer <access_token>`



You can use the **Authorize** button on the [Aura API](#) page to authenticate and test endpoints directly using your client ID and client secret. The UI sets the **Authorization** header for you.

Token endpoint

You can use the following `/oauth/token` endpoint to obtain a Bearer Token for authenticating API requests.

Authentication to the token endpoint uses HTTP Basic Authentication, where the client ID and client secret are provided as the username and password, respectively.

Request parameters

- Method: `POST`
- Token URL: `https://api.neo4j.io/oauth/token`

Request header

Parameter	Value
Authorization ^[4]	Basic <credentials> ^[5]
Content-Type	application/x-www-form-urlencoded

Request body

Parameter	Value
<code>grant_type</code>	<code>client_credentials</code>



Both the request and response contain sensitive information and must be kept secure.

You are responsible for keeping the client credentials and access tokens confidential, whether in transit (by specifying HTTPS), if stored at rest, in log files, etc.

Request examples

```
curl --request POST 'https://api.neo4j.io/oauth/token' \
--user '<client_id>:<client_secret>' \ ①
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=client_credentials'
```

- ① The `--user` option sets the `Authorization` header for you, handling the base64 encoding of the client ID and client secret.

```
import requests
from requests.auth import HTTPBasicAuth

# Make the request to the Aura API Auth endpoint
response = requests.request(
    "POST",
    "https://api.neo4j.io/oauth/token",
    headers={"Content-Type": "application/x-www-form-urlencoded"},
    data={"grant_type": "client_credentials"},
    auth=HTTPBasicAuth(client_id, client_secret) ①
)

print(response.json())
```

- ① `client_id` and `client_secret` must be set to the values obtained from the Aura Console.

Response body example

```
{
  "access_token": "<token>", ①
  "expires_in": 3600,
  "token_type": "bearer"
}
```

- ① The `access_token` returned here is what you will provide as the Bearer Token in the `Authorization` header of Aura API requests.

HTTP response codes

Code	Message	Description
200	Success	Access token requested successfully.
400	Bad Request	Request is invalid.
401	Unauthorized	The provided credentials are invalid, expired, or revoked.
403	Forbidden	The request body is invalid.
404	Not Found	The request body is missing.

Token expiration

If you send a request to the Aura API while authenticated with an expired access token, you will receive a 403 Forbidden response. You will need to obtain a new token to continue using the API.

API Specification

[4] This header is set for you when providing your client ID as the username and client secret as the password.

[5] Where `<credentials>` is the base64-encoded string of your client ID and client secret, joined by a colon (:).

Consumption report

AuraDB Virtual Dedicated Cloud

Virtual Dedicated Cloud services are offered through prepaid consumption plans. Billing is based on usage, with credits deducted from the available balance each month.

The consumption report, accessible in the Aura console's Billing section, provides real-time insights into resource usage for the current project, including both running and paused states. It displays RAM usage in GB-hours and the equivalent cost in prepaid credits.

Available to Admins, the report helps you track usage patterns over time and make informed resource allocation decisions. Note that it includes primary database usage but **not** secondary database usage.

Monitor consumption in real-time

Billing category

The consumption report shows the billing status, which can be **running** meaning customers are charged the full price, or **paused** meaning customers are charged 20% of the hourly rate.

Billing status

Billing status can be **ongoing** or **ended**.

Usage (GB-hours)

Charges are based on the time databases run and the memory consumed, measured in GB-hours. GB-hours usage is calculated by multiplying the number of hours a database is running (whether actively used or not) by the memory size in gigabytes (GB).

The total usage for the selected period is displayed in GB-hours, along with the equivalent credit.

Filters

- Filter the usage data by predefined and custom date intervals.
- Look back for a period of up to 3 months.
- Filter by **Last 24 hours**, **Last 7 days**, **Last 30 days**, **Last 90 days** or a **Custom range**.

neo4j Aura New Organization / Project 1 Send feedback Learn Jane Doe

Data services

Instances

Import

Tools

Explore

Query

Operations

Metrics

Logs

Project

Users

Billing

Settings

Billing

Usage

Credits spent: **421.80**

Instance	ID	Billing category	Billing status	Usage (Hours)	Credits consumed
Instance1	05a0bc04	running	Ended (2024-09-09 10:07 U...	138	34.50
Instance2	42deb68c	running	Ongoing	728	182.00
Instance3	b861d888	paused	Ongoing	636	31.80
Instance4	b861d888	running	Ended (2024-09-04 09:10 U...	12	3.00
Instance5	f4123993	running	Ongoing	682	170.50

Showing 1-5 of 5 results Show

*Usage for secondaries is not included in this view.

Figure 14. Consumption report visual

=Neo4j AuraDB=

Neo4j AuraDB overview

Neo4j AuraDB is a fully managed cloud graph database service.

Built to leverage relationships in data, AuraDB enables lightning-fast queries for real-time analytics and insights. AuraDB is reliable, secure, and fully automated, enabling you to focus on building graph applications without worrying about database administration.

Plans

AuraDB offers the following subscription plans: AuraDB Free, AuraDB Professional, AuraDB Business Critical, and AuraDB Virtual Dedicated Cloud. The full list of features available in each plan is available on the [Neo4j Pricing page](#).

Updates and upgrades

AuraDB does not have any scheduled maintenance windows. It is designed to be always on and available, with all corrections, fixes, and upgrades automatically applied in the background.

Releases for the Neo4j database are also deployed when they become available. Operations are non-disruptive, and you shouldn't experience any downtime as a result.

Support

For a breakdown of the support offered across plan types as well as the support holiday schedule, see the [Aura Support page](#).

Additionally, you can access the [Aura Status page](#) to check the current operational status of Aura and subscribe to updates.

Getting Started

Creating an instance

The process of creating an instance differs depending on the type.

You can select from the options below to display the relevant process.

To create an AuraDB Free instance in Neo4j AuraDB:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select **New Instance**.
3. Select **Create Free instance**.
4. Copy and store the instance's **Username** and **Generated password** or download the credentials as a **.txt** file.
5. Tick the confirmation checkbox, and select **Continue**.

You can only create **one** AuraDB Free instance per account.

A Free instance is limited to 200,000 nodes and 400,000 relationships.

If you don't perform any write queries for three days, your instance is **paused**. You can resume your paused instance from the console.

A paused instance is **deleted after 30 days** and after that, you **cannot restore it or recover your data**.

Additionally, Free instances are **not automatically backed up**. Snapshots are taken on-demand and only the latest snapshot is available for download. For more information about snapshots, see [Backup, export and restore](#) for more information.

To create an AuraDB Professional instance in Neo4j AuraDB:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select **New Instance** to open the **Create an instance** page. (Additionally, you will need to select **Select Professional instance** if you have yet to create an AuraDB Free instance.)
3. Select your preferred **Cloud provider** and **Region**. The region is the physical location of the instance; set this as close to your location as possible. The closer the region is to your location, the faster the response time for any network interactions with the instance.
4. Set your **Instance size**, the memory, CPU, and storage allocated to the instance. The larger the instance size, the more it costs to run. Once selected, you can see the running cost at the bottom of the page.
5. Set your **Instance details**:
 - **Instance Name** - The name to give the instance. This name can be whatever you like.
 - **Neo4j Version** - The version of the Neo4j instance.
6. Tick the **I understand** checkbox next to the running cost confirmation.
7. Select **Create** when happy with your instance details and size.
8. Copy and store the instance's **Username** and **Generated password** or download the credentials as a **.txt** file.
9. Tick the confirmation checkbox, and select **Continue**.



Aura retains some of your provisioned resources for managing your instance.



Pay-as-you-go (PAYG) is available on all instance sizes up to 128 GB. Prepaid is available from 16 GB+.

To create an AuraDB Business Critical instance in Neo4j AuraDB:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select **New Instance** to open the **Create an instance** page. (Additionally, you need to select **Select Business Critical instance** if you have yet to create an AuraDB Professional instance.)
3. Select your preferred **Cloud provider** and **Region**. The region is the physical location of the instance. Set this as close to your location as possible. The closer the region is to your location, the faster the response time for any network interactions with the instance.
4. Set your **Instance size**, the memory, CPU, and storage allocated to the instance. Once selected, you can see the running cost at the bottom of the page.
5. Set your **Instance details**:
 - **Instance Name** - The name of the instance. This name can be whatever you like.
 - **Neo4j Version** - The version of the Neo4j instance.
6. Tick the **I understand** checkbox next to the running cost confirmation.
7. Select **Create** when happy with your instance details and size.
8. Copy and store the instance's **Username** and **Generated password** or download the credentials as a **.txt** file.
9. Tick the confirmation checkbox, and select **Continue**.

To create an AuraDB Virtual Dedicated Cloud instance in Neo4j AuraDB:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select **New Instance** to open the **Create an instance** page.
3. Set your **Instance size**, the memory, CPU, and storage allocated to the instance. Please refer to your contract for pricing.
4. Set your **Instance details**:
 - **Instance Name** - The name to give the instance. This name can be whatever you like.
 - **Neo4j Version** - The version of the Neo4j instance.
 - **Region** - The physical location of the instance. Set this as close to your location as possible. The closer the region to your location, the faster the response time for any network interactions with the instance.
5. Tick the **I understand** checkbox.
6. Select **Create Instance** when happy with your instance details and size.
7. Copy and store the instance's **Username** and **Generated password** or download the credentials as a `.txt` file.
8. Tick the confirmation checkbox, and select **Continue**.



Aura retains some of your provisioned resources for managing your instance.



Multi-database is not currently supported within Neo4j AuraDB.

Connecting to an instance

There are several different methods of connecting to an instance in Neo4j AuraDB:

- [Neo4j Browser](#) - A browser-based interface for querying and viewing data in an instance.
- [Neo4j Bloom](#) - A graph exploration application for visually interacting with graph data.
- [Neo4j Workspace](#) - A browser-based interface used to import, visualize, and query graph data.
- [Neo4j Desktop](#) - An installable desktop application used to manage local and cloud instances.
- [Neo4j Cypher Shell](#) - A command-line tool used to run Cypher queries against a Neo4j instance.

Neo4j Browser

You can query an instance using Neo4j Browser.

To open an instance with Browser:

1. Navigate to the [Neo4j Aura Console](#) in your browser.

2. Select the **Query** button on the instance you want to open.
3. Enter the **Username** and **Password** credentials in the window that opens. These are the same credentials you stored when [creating the instance](#).
4. Select **Connect**.

Once you have successfully connected, there are built-in guides you can complete to familiarize yourself with Neo4j Browser.

For more information on using Neo4j Browser, please see the [Browser manual](#).

Neo4j Bloom

You can explore an instance using Neo4j Bloom.

To open an instance with Bloom:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the **Explore** button on the instance you want to open.
3. Select **Neo4j Bloom** from the dropdown menu.
4. Enter the **Username** and **Password** credentials in the window that opens. These are the same credentials you stored when [creating the instance](#).
5. Select **Connect**.

For more details on using Neo4j Bloom, please see the [Neo4j Bloom documentation](#).

Perspectives in AuraDB

Due to the nature of AuraDB's infrastructure, it is not currently possible to share Perspectives in Bloom, as the data for a given Perspective is stored in local storage in the user's web browser.

An alternative is to export your Perspective as a JSON file and import it into another Bloom session.

To export a Perspective:

1. Open the Bloom interface for your Neo4j AuraDB instance.
2. Navigate to the *Perspectives Gallery*.
3. Click on the vertical ellipsis (...) and select **Export**.
4. Save the file to your local disk.

You can import perspectives by clicking the blue "Import Perspective" button in the Perspective gallery. Please note that the Perspective exposes details about your graph's schema but not the actual data within.

For more information, see [Bloom Perspectives](#).

Deep links

As data for a given Perspective is stored in local storage in the user's web browser, if you want to access a deep link referencing perspectives, you will first need to import the perspectives into your local instance of Bloom.



Neo4j Workspace

Neo4j Workspace combines the functionality of Neo4j Browser, Neo4j Bloom, and Neo4j Data Importer into a single interface.

To open an instance with Workspace:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the **Open** button on the instance you want to open.
3. Enter the **Database user** and **Password** credentials in the window that opens. These are the same credentials you stored when [creating the instance](#).
4. Select **Connect**.

For more information on using Neo4j Workspace, see the [Product page](#).



Workspace is enabled by default on AuraDB Free and AuraDB Professional instances but needs to be enabled for AuraDB Virtual Dedicated Cloud instances.

If you do not see the **Open** button on your instance, you can enable it by selecting the **Settings** cog in the top menu bar and toggling **Enable workspace**.

Neo4j Desktop

You can connect AuraDB instances to the Neo4j Desktop application, allowing the ability to have a single portal for interacting with all instances of Neo4j, whether local or located in the cloud.

To connect to an instance using Neo4j Desktop:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Copy the **Connection URI** of the instance you want to connect to. The URI is below the instance status indicator.
3. In Neo4j Desktop, select the **Projects** tab and select an existing project or create a new one.
4. Select the **Add** dropdown and choose **Remote connection**.
5. Enter a name for the instance and enter the URL from the Neo4j Aura console from the second step. Once complete, select **Next**.
6. With **Username/Password** selected, enter your credentials and select **Next**. These are the same credentials you stored when [creating the instance](#).
7. When available, activate the connection by clicking the **Connect** button.



- Neo4j Desktop only allows 1 connection at a time to an instance (local or remote).
- Deactivating an instance in Neo4j Desktop won't shut it down or stop a remote instance - it will only temporarily close the connection to it in Neo4j Desktop.

As with other instances in Neo4j Desktop, you can install [Graph Apps](#) for monitoring and other functionality.

To do this, follow the same process to install the graph application you need, and open it from Neo4j Desktop or a web browser with the running and activated Neo4j AuraDB instance.

Neo4j Cypher Shell

You can connect to an AuraDB instance using the Neo4j Cypher Shell command-line interface (CLI) and run Cypher commands against your instance from the command-line.

To connect to an instance using Neo4j Cypher Shell:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Copy the **Connection URI** of the instance you want to connect to. The URI is below the instance status indicator.
3. Open a terminal and navigate to the folder where you have installed Cypher Shell.
4. Run the following `cypher-shell` command replacing:
 - `<connection_uri>` with the URI you copied in step 2.
 - `<username>` with the username for your instance.
 - `<password>` with the password for your instance.

```
./cypher-shell -a <connection_uri> -u <username> -p <password>
```

Once connected, you can run `:help` for a list of available commands.

```
Available commands:
:begin      Open a transaction
:commit     Commit the currently open transaction
:exit       Exit the logger
:help       Show this help message
:history    Print a list of the last commands executed
:param      Set the value of a query parameter
:params     Print all currently set query parameters and their values
:rollback   Rollback the currently open transaction
:source     Interactively executes cypher statements from a file
:use        Set the active instance
```

```
For help on a specific command type:
:help command
```

For more information on Cypher Shell, including how to install it, see the [Cypher Shell documentation](#).

Querying an instance

You can query data in an AuraDB instance using Cypher.

Cypher is the declarative graph query language created by Neo4j and can be used to query, update, and administer your AuraDB instance.

You can run Cypher statements through Neo4j Browser and Neo4j Cypher Shell. For more information on how to open an AuraDB instance in Browser and Cypher Shell, see [Connecting to an instance](#).

For more information on Cypher and Aura, see [the Neo4j Cypher Manual](#).

Importing

Importing data



The process of importing or loading data requires you to [create an AuraDB instance](#) beforehand.

There are two ways you can import data from a CSV file into an AuraDB instance:

- [Load CSV](#) - A Cypher statement that you run from Neo4j Browser or Neo4j Cypher Shell.
- [Neo4j Data Importer](#) - A visual application that you launch from the Console.

Load CSV

The `LOAD CSV` Cypher statement can be used from within Neo4j Browser and Cypher Shell. For instructions on how to open an AuraDB instance with Browser or Cypher Shell, see [Connecting to an instance](#).

There are some limitations to consider when using this method to load a CSV file into an AuraDB instance:

- For security reasons, you must host your CSV file on a publicly accessible HTTP or HTTPS server. Examples of such servers include AWS signed URLs, GitHub, Google Drive, and Dropbox.
- The `LOAD CSV` command is built to handle small to medium-sized data sets, such as anything up to 10 million nodes and relationships. You should avoid using this command for any data sets exceeding this limit.

Neo4j Data Importer

Neo4j Data Importer is a UI-based tool for importing data that lets you:

1. Load data from flat files (`.csv` and `.tsv`).
2. Define a graph model and map data to it.
3. Import the data into an AuraDB instance.

To load data with Neo4j Data Importer:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the **Import** button on the instance you want to open.

Alternatively, you can access Data Importer from the **Import** tab of [Neo4j Workspace](#).

For more information on Neo4j Data Importer, see the [Neo4j Data Importer documentation](#).



You must provide your AuraDB instance password before importing from the Neo4j Data Importer.

Importing an existing database



The process of importing or loading data requires you to [create an AuraDB instance](#) beforehand.

There are two ways you can import data from an existing Neo4j database into an Aura instance.

You can use the `import database` process to import either a `.backup` file or a `.dump` file. This process, however, only works for `.backup` and `.dump` files under 4GB.

If the size of the `.backup` or `.dump` file exported from a database is greater than 4GB, you must use the [Neo4j Admin database upload](#) method.

For more information about backups, see [Backup, export and restore](#).

Import database

To import a `.backup` file under 4GB:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to import the data.
3. Select the `Import Database` tab.
4. Drag and drop your `.backup` or `.dump` file into the provided window or browse for your `.backup/.dump` file.
5. Select `Upload`.

When the upload is complete, the instance goes into a `Loading` state as the backup is applied. Once this has finished, the instance returns to its `Running` state; and the data is ready.

Neo4j Admin `database upload`



This command does not work if you have a network access configuration setup that prevents public traffic to the region your instance is hosted in. See [Public traffic](#) below for more information.

`database upload` is a `neo4j-admin` command that you can run to upload the contents of a Neo4j database into an Aura instance, regardless of the database's size. Keep in mind that the database you want to upload may run a different version of Neo4j than your Aura instance. Additionally, your Neo4j Aura instance must be accessible from the machine running `neo4j-admin`. Otherwise, the upload will fail with SSL errors.

For details on how to use the `neo4j-admin database upload` command, along with a full list of options and version compatibility, see [Operations Manual → Upload to Neo4j Aura](#).



The `database upload` command, introduced in Neo4j 5, replaces the `push-to-cloud` command in Neo4j 4.4 and 4.3. If the database you want to upload is running an earlier version of Neo4j, please see the [Neo4j Admin push-to-cloud documentation](#).



The `neo4j-admin push-to-cloud` command in Neo4j 4.4 and earlier is not compatible with instances encrypted with [Customer Managed Keys](#). Use `neo4j-admin database upload` in Neo4j 5 to upload data to instances encrypted with Customer Managed Keys.

For Neo4j 4.x instances in Azure encrypted with Customer Managed Keys, use Neo4j Data Importer to load data, as `neo4j-admin database upload` is not supported. See the [Data Importer documentation](#) for more information.

Public traffic

If you have created a network access configuration from the [Network Access](#) page, accessed through the sidebar menu of the Console, **Public traffic** must be enabled for the region your instance is hosted in before you can use the `database upload` command on that instance.

To enable **Public traffic** on a network access configuration:

1. Select **Configure** next to the region that has **Public traffic** disabled.
2. Select **Next** until you reach step 4 of 4 in the resulting **Edit network access configuration** modal.
3. Clear the **Disable public traffic** checkbox and select **Save**.

You can now use the `database upload` command on the instances within that region. Once the command has completed, you can disable **Public traffic** again by following the same steps and re-selecting the **Disable public traffic** checkbox.

Managing instances

Instance actions

You can perform several instance actions from an AuraDB instance card on the [Neo4j Aura Console](#) homepage.

Rename an instance

You can change the name of an existing instance using the **Rename** action.

To rename an instance:

1. Select the ellipsis (...) button on the instance you want to rename.
2. Select **Rename** from the resulting menu.
3. Enter a new name for the instance.
4. Select **Rename**.

Reset an instance

[AuraDB Free](#) [AuraDB Professional](#)

You can clear all data in an instance using the **Reset to blank** action.

To reset an instance:

1. Select the ellipsis (...) button on the instance you want to reset.
2. Select **Reset to blank** from the resulting menu.
3. Select **Reset**.

Upgrade an instance

Upgrade AuraDB Free to AuraDB Professional

You can upgrade an AuraDB Free instance to an AuraDB Professional instance using the **Upgrade to Professional** action.

Upgrading your instance clones your Free instance data to a new Professional instance, leaving your existing Free instance untouched.

To upgrade a Free instance:

1. Select the ellipsis (...) button on the free instance you want to upgrade.
2. Select **Upgrade to Professional** from the resulting menu.

3. Set your desired settings for the new instance. For more information on AuraDB instance creation settings, see [Creating an instance](#).
4. Tick the **I understand** checkbox and select **Upgrade Instance**.

Upgrade AuraDB Professional to AuraDB Business Critical

You can upgrade an AuraDB Professional instance to an AuraDB Business Critical instance using the **Upgrade to Business Critical** action.

Upgrading your instance clones your Professional instance data to a new Business Critical instance, leaving your existing Professional instance untouched.

To upgrade a Business Critical instance:

1. Select the ellipsis (...) button on the free instance you want to upgrade.
2. Select **Upgrade to Business Critical**.
3. Set your desired settings for the new instance. For more information on AuraDB instance creation settings, see [Creating an instance](#).
4. Tick the **I understand** checkbox and select **Upgrade Instance**.

Resize an instance

[AuraDB Professional](#) [AuraDB Virtual Dedicated Cloud](#) [AuraDB Business Critical](#)

You can change the size of an existing instance using the **Resize** action.

To resize an instance:

1. Select the ellipsis (...) button on the instance you want to resize.
2. Select **Resize** from the resulting menu.
3. Select the new size you want your instance to be.
4. Tick the **I understand** checkbox and select **Upgrade instance**.

An instance remains available during the resize operation.

Pause an instance

[AuraDB Professional](#) [AuraDB Virtual Dedicated Cloud](#) [AuraDB Business Critical](#)



You cannot manually pause an AuraDB Free instance; they are paused automatically after 72 hours of inactivity. ^[6]

You can pause an instance when not needed and resume it at any time.

To pause an instance:

1. Select the pause button on the instance you want to pause.

2. Tick the **I understand** checkbox and select **Pause** to confirm.

After confirming, the instance begins pausing, and a play button replaces the pause button.



Paused instances run at a discounted rate compared to standard consumption, as outlined in the confirmation window. You can pause an instance for up to 30 days, after which point AuraDB automatically resumes the instance.

Resume a paused instance

To resume an instance:

1. Select the play button on the instance you want to pause.
2. Tick the **I understand** checkbox and select **Resume** to confirm.

After confirming, the instance begins resuming, which may take a few minutes.



AuraDB Free instances do not automatically resume after 30 days. If an AuraDB Free instance remains paused for more than 30 days, Aura deletes the instance, and all information is lost.

Clone an instance

You can clone an existing instance to create a new instance with the same data. You can clone across regions, from AuraDB to AuraDS and vice versa, and from Neo4j version 4 to Neo4j version 5.

There are four options to clone an instance:

- Clone to a new AuraDB instance
- Clone to an existing AuraDB instance
- Clone to a new AuraDS database
- Clone to an existing AuraDS database

You can access all the cloning options from the ellipsis (...) button on the AuraDB instance.



You cannot clone from a Neo4j version 5 instance to a Neo4j version 4 instance.

Clone to a new AuraDB instance

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To New** and then **AuraDB Professional/Business Critical/Virtual Dedicated Cloud** from the contextual menu.
3. Set your desired settings for the new database. For more information on AuraDB database creation, see [Creating an instance](#).
4. Check the **I understand** box and select **Clone Database**.



Make sure that the username and password are stored safely before continuing. Credentials cannot be recovered afterwards.

Clone to an existing AuraDB instance

When you clone an instance to an existing instance, the database connection URI stays the same, but the data is replaced with the data from the cloned instance.



Cloning into an existing instance will replace all existing data. If you want to keep the current data, take a snapshot and export it.

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To Existing** and then **AuraDB** from the contextual menu.
3. If necessary, change the database name.
4. Select the existing AuraDB database to clone to from the dropdown menu.



Existing instances that are not large enough to clone into will not be available for selection. In the dropdown menu, they will be grayed out and have the string **(Instance is not large enough to clone into)** appended to their name.

5. Check the **I understand** box and select **Clone**.

Clone to a new AuraDS instance

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To New** and then **AuraDS** from the contextual menu.
3. Set the desired name for the new instance.
4. Check the **I understand** box and select **Clone Instance**.



Make sure that the username and password are stored safely before continuing. Credentials cannot be recovered afterwards.

Clone to an existing AuraDS instance

When you clone an instance to an existing instance, the database connection URI stays the same, but the data is replaced with the data from the cloned instance.



Cloning into an existing instance will replace all existing data. If you want to keep the current data, take a snapshot and export it.

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To Existing** and then **AuraDS** from the contextual menu.
3. If necessary, change the instance name.

4. Select the existing AuraDS instance to clone to from the dropdown menu.



Existing instances that are not large enough to clone into will not be available for selection. In the dropdown menu, they are grayed out and have the string (Instance is not large enough to clone into) appended to their name.

5. Tick the I understand checkbox and select Clone.

Delete an instance

You can delete an instance if you no longer want to be billed for it.

To delete an instance:

1. Select the red trashcan icon on the instance you want to delete.
2. Type the exact name of the instance (as instructed) to confirm your decision, and select Destroy.



There is no way to recover data from a deleted AuraDB instance.

Backup, export and restore

The data in your AuraDB instance can be backed up, exported, and restored using snapshots.

A snapshot is a copy of the data in an instance at a specific point in time.

The Snapshots tab within an AuraDB instance shows a list of available snapshots.

To access the Snapshots tab:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to access.
3. Select the Snapshots tab.



Only the latest snapshot is available for Free instances. Snapshots are available for 7 days for Professional instances, 30 days for Business Critical instances, and 60 days for AuraDB Virtual Dedicated Cloud instances.

Snapshot types

Scheduled

[AuraDB Professional](#) [AuraDB Business Critical](#) [AuraDB Virtual Dedicated Cloud](#)

A Scheduled snapshot is a snapshot that is automatically triggered when you first create an instance, when changes to the underlying system occur (for example, a new patch release), and at a cadence depending on your plan type.

Scheduled snapshots are run automatically once a day for Professional instances and once an hour for Business Critical and Enterprise instances.



For AuraDB Virtual Dedicated Cloud database instances running Neo4j v4.x, from day 0 to 7 scheduled snapshots run automatically once every 6 hours. From day 8 to 60, snapshots run once a day.

On demand

An On Demand snapshot is a snapshot that you manually trigger by selecting **Take snapshot** from the **Snapshots** tab of an instance.

Snapshot actions

Restore



Restoring a snapshot overwrites the data in your instance, replacing it with the data contained in the snapshot.

You can restore data in your instance to a previous snapshot by selecting **Restore** next to the snapshot you want to restore.

Restoring a snapshot requires you to confirm the action by typing **RESTORE** and selecting **Restore**.

Export and create

The ellipses (...) button next to an existing snapshot, allows you to:

- **Export** - Download the instance as **.backup** file, allowing you to store a local copy and work on your data offline. (This applies to AuraDB v5 databases, for v4, the instances can be downloaded as **.dump** files.)
- **Create instance from snapshot** - Create a new AuraDB instance using the data from the snapshot.



The ability to Export or Create an instance from a Scheduled Virtual Dedicated Cloud snapshot is limited to 14 days.

Additionally, for Virtual Dedicated Cloud instances running Neo4j version 5, the ability to export or create an instance from a Scheduled snapshot is limited to the first full snapshot, taken once per day.

Use the toggle **Show exportable only** on top of the list of snapshots to filter by whether a snapshot is exportable or not.

Security of backups and exported data

Neo4j Aura Enterprise automatically creates backups of each database at regular intervals. Aura stores the data securely in encrypted and dedicated cloud storage buckets. Users access backups through the Aura

console. In the console, it's possible to:

- See a list of previous backups
- Choose to restore a backup
- Download a backup (which serves as the export mechanism)

Retention periods

The current Neo4j Aura Snapshot retention periods by tier

Tier	Aura version	Scheduled Snapshots per day	Scheduled Snapshot restorable days (exportable days)	On-Demand Snapshot restorable days (exportable days)
AuraDB Free	4, 5	N/A	N/A	30 days (30 days)
AuraDB Professional	4, 5	Full retention: 1	30 days (7 days)	30 days (30 days)
AuraDS Professional	4, 5	Full retention: 1	9 days (7 days)	180 days (180 days)
AuraDB Business Critical	5	Full: 1, Differential: 23 ^[1]	30 days (7 days)	30 days (30 days)
AuraDB Virtual Dedicated Cloud	4	Full retention: 1, Short retention: 3 ^[6]	Full retention: 60 (14), Short retention: 7 (7) ^[6]	90 days (90 days)
	5	Full: 1, Diferential: 23 ^[7]	Full: 60 (14), Differential: 60 (N/A)	90 days (90 days)
AuraDS Enterprise	4	Full retention: 1	Full retention: 16 days (7 day)	180 days (180 days)
	5	Full: 1	Full: 16 days (7 days)	180 days (180 days)

Secondaries

AuraDB Virtual Dedicated Cloud

A secondary is a read-only copy of your Aura database. Secondaries help you scale the read query workload your AuraDB instance is able to serve, by spreading the load evenly across multiple copies of the data. This increases the maximum read query throughput of a database while preventing bottlenecks.

To ensure high availability, secondaries are distributed across availability zones. They are however, only available within the same cloud region as the primary Aura instance.

Up to 15 secondaries can be added per AuraDB instance, which increases the read capacity to handle read-heavy workloads significantly. Secondaries can be added, managed, and removed through the Aura console or the Aura API. Currently, they are static and do not support elastic or auto-scaling behavior.

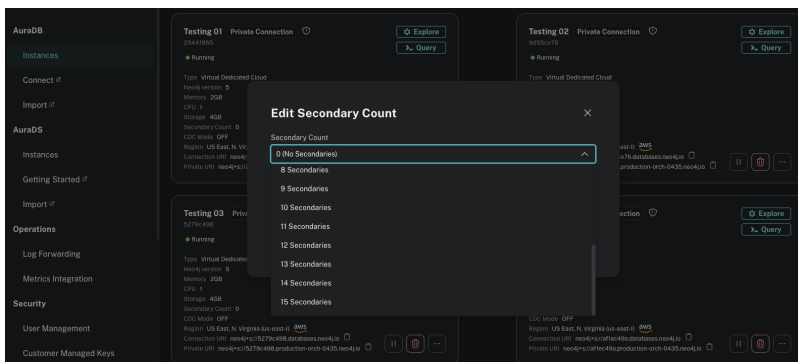
The secondary count is retained when the database is paused and resumed. For example, if your database has three secondaries and you pause it, it will resume with three secondaries.



Secondaries can take some time to become operational after they are created, and there may be delays when the system is busy. Causal consistency is maintained among your secondaries with the use of bookmarks and these also ensure that returned data is correct and up-to-date. However, if the database is under heavy load, queries using bookmarks may also experience delays in adding secondaries. See [Operations Manual](#) → [Causal consistency](#) for more information.

Edit secondary count using the console

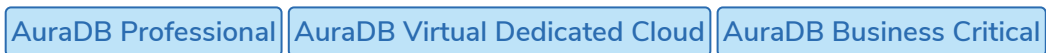
Once the feature is enabled for your project, you can see the secondary count set to zero on an instance card. To edit the number of secondaries, use the **More** menu (three dots) on the card.



Edit secondary count using the Aura API

Use the `/instances/{instanceId}` endpoint to edit the number of secondaries.

Monitoring



You can monitor the following metrics of an AuraDB instance from the **Metrics** tab:

- **CPU Usage (%)** - The amount of CPU used by the instance as a percentage.
- **Storage Used (%)** - The amount of disk storage space used by the instance as a percentage.
- **AuraDB Virtual Dedicated Cloud Heap Usage (%)** - The amount of Java Virtual Machine (JVM) memory used by the instance as a percentage.
- **Out of Memory Errors** - The number of Out of Memory (OOM) errors encountered by the instance.
- **Garbage Collection Time (%)** - The amount of time the instance spends reclaiming heap space as a percentage.
- **Page Cache Evictions** - The number of times the instance has replaced data in memory.



More information on each metric, as well as suggestions for managing them, can be found within the **Metrics** tab itself.

When viewing metrics, you can select from the following time intervals:

- 6 hours
- 24 hours
- 3 days
- 7 days
- 30 days

To access the Metrics tab:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to access.
3. Select the Metrics tab.

Advanced metrics

[AuraDB Professional](#) [AuraDB Business Critical](#) [AuraDB Virtual Dedicated Cloud](#)

Advanced metrics is a feature that enables access to a broad range of different instance and database metrics.

To access Advanced metrics:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to access.
3. Select the Metrics tab.
4. Select the Advanced metrics button.

The presented metrics will be laid out across three tabs according to their category:

- **Resources** - Overall system resources, such as CPU, RAM and disk usage.
- **Instance** - Information about the Neo4j instances running the database.
- **Database** - Metrics concerning the database itself, such as usage statistics and entity counts.

When viewing metrics, you can select from the following time intervals:

- 30 minutes
- 6 hours
- 24 hours
- 3 days
- 7 days
- 14 days
- 30 days

Chart interactions



Memory and storage charts can be toggled between absolute and relative values using the % toggle.

Toggle data series

[AuraDB Business Critical](#) [AuraDB Virtual Dedicated Cloud](#)

To hide or show individual data series, select the corresponding data series in the legend below the chart.

Zoom

To zoom in to a narrower time interval, select and drag inside any chart to select your desired time interval. The data will automatically update to match the increased resolution.

To reset zoom, double-click anywhere inside the chart or use the option in the context menu.

Expand

Any chart can be expanded to take up all the available screen estate by clicking the **expand** button (shown as two opposing arrows). To exit this mode, click the **x** button on the expanded view.

Context menu

To access the chart context menu, select the ... button on any chart.

- **More info** - Selecting **More info** brings up an explanation of the particular metric. For some metrics it also provides hints about possible actions to take if that metric falls outside the expected range.
- **Reset zoom** - If the zoom level has been altered by selecting and dragging across a chart, **Reset zoom** resets the zoom back to the selected interval.

Aggregations

[AuraDB Business Critical](#) [AuraDB Virtual Dedicated Cloud](#)

Most metrics will have several values for a given timestamp because of the following reasons:

- Multiple database replicas
- Compressing several data points into one, depending on zoom level

Aggregating functions are used to reconcile metrics having multiple data points and make the most sense of that particular metric. To convey an even more detailed picture of the state of the system, several aggregations can be shown.

The possible aggregations are:

- **Min** - The minimum value of the metric across all cluster members.

- **Max** - The maximum value of the metric across all cluster members.
- **Average** - The average value of the metric across all cluster members.
- **Sum** - The sum of the metric across all cluster members.

Detail view

AuraDB Business Critical AuraDB Virtual Dedicated Cloud

An Aura instance can run on multiple servers to achieve availability and workload scalability. These servers are deployed across different Cloud Provider availability zones in the user-selected region.

Detail view shows distinct data series for availability zone & instance mode combinations. This is presented as an alternative to the aggregations described above.

Detail view can be enabled with the toggle under the time interval selector.



Metrics in the Detail view for a new Aura instance may take time to appear because of the way 'availability zone' data is collected.

Store size metrics

Resources tab

The chart on the *Resources* tab shows the allocated store size metric for the selected database either as a percentage of the available storage assigned for the database or as absolute values.

Database tab

The *Database* tab provides a chart that shows the store size and the portion of the allocated space that the database is actively utilizing. Both metrics are represented as percentages of the available storage assigned to the database.

These metrics may differ due to the way Neo4j allocates and reuses space. Once allocated, space is never automatically de-allocated. Thus, reducing the data (nodes, relationships, properties) stored in the database does not reduce the top-line store size metric. However, Neo4j will reuse this 'available' space before allocating more from the system. The amount of allocated space that is 'available' is reported by the database, and Advanced metrics uses this metric to derive the used space by subtracting it from the allocated store size. This information can help you understand how close your database is to exceeding the assigned storage size.

Connecting applications

You can use the official [drivers and libraries](#) provided by Neo4j to connect your application to AuraDB using a variety of programming languages.

Regardless of what language you use, you will need to provide the following information to connect to an AuraDB instance:

- **uri** - The **Connection URI** for your AuraDB instance. You can copy this from the instance card or details page in the Console.
- **username** and **password** - The **Username** and **Password** for your AuraDB instance. You can copy or download these during the instance creation process.

Change Data Capture

Change Data Capture (CDC) allows you to capture and track changes to your database in real-time, enabling you to keep your other data sources up to date with Neo4j. With CDC, you can identify and respond to changes (create, update, and delete) on nodes and relationships as they happen, and integrate these changes into other systems and applications.

See [CDC on Neo4j Aura](#) for more information about setting up CDC, configuring it to capture the changes, and querying those changes for further processing, such as replicating to another system.

[6] Inactivity is when you perform no queries on the instance.

[7] Differential backups only contain the new data since the last backup, and are therefore not exportable, but they are restorable for the full duration.

[8] Short retentions backups are the same as full backups, only with a shorter lifespan.

=Neo4j AuraDS=

Neo4j AuraDS overview

AuraDS is the fully managed version of Neo4j Graph Data Science.

AuraDS instances:

- are automatically upgraded and patched;
- can be seamlessly scaled up or down;
- can be paused to reduce costs.

Plans

AuraDS offers the **AuraDS Professional** and **AuraDS Enterprise** subscription plans. The full list of features for each plan is available on the [Neo4j Pricing page](#).

Updates and upgrades

AuraDS updates and upgrades are handled by the platform, and as such do not require user intervention. Security patches and new versions of GDS and Neo4j are installed within short time windows during which the affected instances are unavailable.

The operations are non-destructive, so graph projections, models, and data present on an instance are not affected. No operation is applied until all the running GDS algorithms have completed.

Support

For a breakdown of the support offered across plan types as well as the support holiday schedule, see the [Aura Support page](#).

Additionally, you can access the [Aura Status page](#) to check the current operational status of Aura and subscribe to updates.

Architecture

AuraDS makes it easy to run graph algorithms on Neo4j by integrating two main components:

- **Neo4j Database**, where graph data are loaded and stored, and Cypher queries and all database operations (for example user management, query termination, etc.) are executed;
- **Graph Data Science**, a software component installed in the Neo4j Database, whose main purpose is to run graph algorithms on in-memory projections of Neo4j Database data.

Neo4j Graph Data Science concepts

The Neo4j Graph Data Science (GDS) library includes procedures to project and manage graphs, run algorithms, and train machine learning models.

Graph Catalog

The [graph catalog](#) is used to store and manage projected graphs via GDS procedures.

Algorithms

GDS contains many [graph algorithms](#), invoked as Cypher procedures and run on projected graphs.

GDS algorithms are broken down into three tiers of maturity:

- **Alpha**: experimental algorithms that may be changed or removed at any time. Algorithms in this tier are prefixed with `gds.alpha.<algorithm>`.
- **Beta**: algorithms promoted from the Alpha tier to candidates for the Production tier. Algorithms in this tier are prefixed with `gds.beta.<algorithm>`.
- **Production**: algorithms that have been rigorously tested for stability and scalability. Algorithms in this tier are prefixed with `gds.<algorithm>`.

Model Catalog

Some machine learning algorithms (for example Node Classification and GraphSage) need to use trained models in their computation. The [model catalog](#) is used to store and manage named trained models.

Pipeline Catalog

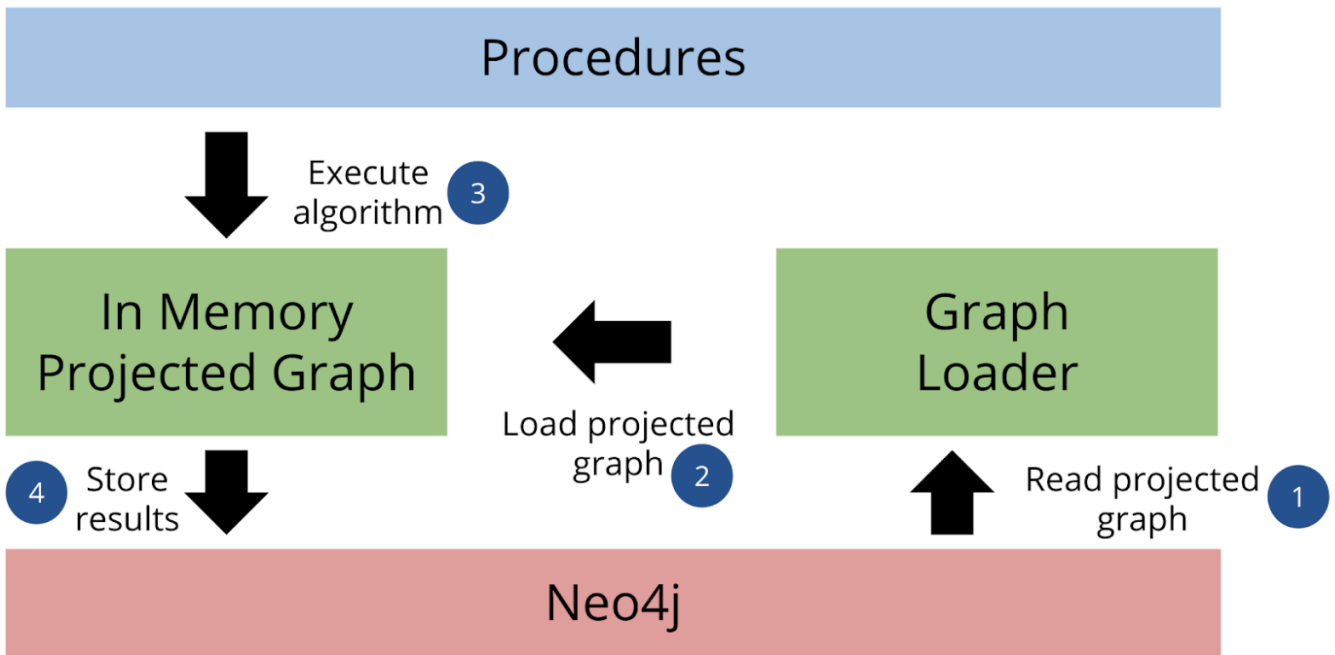
The [pipeline catalog](#) is used to manage machine learning pipelines. A pipeline groups together all the stages of a supported machine learning task (for example Node classification), from graph feature extraction to model training, in a single end-to-end workflow.

Graph data flow

Since GDS algorithms can only run in memory, the typical data flow involves:

1. Reading the graph data from Neo4j Database
2. Loading (projecting) the data into an in-memory graph
3. Running an algorithm on a projected graph

4. Writing the results back to Neo4j Database (if the algorithm runs in [write](#) mode)



Creating an AuraDS instance

1. Navigate to the [Neo4j Aura Console](#).
2. Select **New instance** to open the **Create an instance** page.
3. Fill up the instance details:
 - **Instance Name** ^[9] - The name to give to the instance. A descriptive name makes it easier to find a specific instance among many.
 - **Region** - The physical location of the instance. Set this as close to your location as possible. The closer the region is to your location, the faster the response time for any network interactions with the instance.
 - **Number of nodes/relationships** - The estimated number of nodes and relationships that the instance should support.
4. Select one or more algorithm categories from the **Which algorithms are you going to use?** section (or select **I'm not sure which algorithms to use**) to help estimate the most appropriate instance size. An overview of each algorithm category can be found [here](#).
5. Select **Calculate Estimate** to get an estimate of the resources needed to run the graph (memory, CPU, storage) along with the expected price.
6. Select **Create** to proceed.
7. Copy and store the **Username** and **Generated password** credentials to access the instance just created. Alternatively, you can download the credentials as a **.txt** file.



Warning: Make sure that the username and password are stored safely before continuing. Credentials cannot be recovered afterwards.

8. Tick the confirmation checkbox and select **Continue**.



Multi-database is not supported within Neo4j AuraDS.

The process will take a few minutes to complete. Upon completion, you will be able to [connect to the instance](#).

[9] In AuraDS Professional, this field becomes available after selecting the **Calculate Estimate** button.

Connecting to AuraDS

Once you have [created](#) an AuraDS instance, you can start using it with any [Neo4j application](#) or directly [from your code](#). Keep your username and password handy, as you will need them to connect to your instance.

Connecting with Neo4j applications

There are several ways to interact with and use graph data in AuraDS.

- [Neo4j Browser](#) - A browser-based interface for querying and viewing graph data with rudimentary visualization.
- [Neo4j Bloom](#) - A graph exploration application for visually interacting with graph data.
- [Neo4j Workspace](#) - A browser-based interface used to import, visualize, and query graph data.
- [Neo4j Desktop](#) - An installable desktop application used to manage local and cloud databases.
- [Neo4j Cypher Shell](#) - A command-line tool used to run Cypher queries against a Neo4j instance.



Tip: For first-time users, we recommend using Neo4j Browser.

Neo4j Browser

To open an AuraDS instance with Neo4j Browser:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the Query button on the instance you want to open.
3. Enter the Username and Password credentials in the Neo4j Browser window that opens. These are the same credentials you stored when you [created the instance](#).
4. Select Connect.

Once you have successfully connected, there are built-in guides you can complete to familiarize yourself with Neo4j Browser. See the [Browser manual](#) for more information.

Neo4j Bloom

To open an AuraDS instance with Neo4j Bloom:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the Explore button on the instance you want to open.
3. Enter the Username and Password credentials in the Neo4j Browser window that opens. These are the same credentials you stored when you [created the instance](#).
4. Select Connect.

See the [Neo4j Bloom documentation](#) for more details.

Perspectives in AuraDS

Due to the nature of AuraDS's infrastructure, it is not currently possible to share Perspectives in Bloom, as the data for a given Perspective is stored in local storage in the user's web browser.

An alternative is to export your Perspective as a JSON file and import it into another Bloom session.

To export a Perspective:

1. Open the Bloom interface for your Neo4j AuraDS instance.
2. Navigate to the *Perspectives Gallery*.
3. Click on the vertical ellipsis (...) and select **Export**.
4. Save the file to your local disk.

You can import perspectives by clicking the blue "Import Perspective" button in the Perspective gallery. Please note that the Perspective exposes details about your graph's schema but not the actual data within.

For more information, see [Bloom Perspectives](#).

Deep links

As data for a given Perspective is stored in local storage in the user's web browser, if you want to access a deep link referencing perspectives, you will first need to import the perspectives into your local instance of Bloom.



Neo4j Workspace

Neo4j Workspace combines the functionality of Neo4j Browser, Neo4j Bloom, and Neo4j Data Importer into a single interface.

To open an instance with Workspace:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the **Open** button on the instance you want to open.
3. Enter the **Database user** and **Password** credentials in the window that opens. These are the same credentials you stored when you [created the instance](#).
4. Select **Connect**.

For more information on using Neo4j Workspace, see the [Product page](#).



Workspace is enabled by default on AuraDB Free and AuraDB Professional instances but needs to be enabled for AuraDB Virtual Dedicated Cloud instances. If you do not see the **Open** button on your instance, you can enable it by selecting the **Settings** cog in the top menu bar and toggling **Enable workspace**.

Neo4j Desktop

You can connect AuraDS instances to the Neo4j Desktop application, allowing the ability to have a single portal for interacting with all instances of Neo4j, whether local or located in the cloud.

To connect to an AuraDS instance using Neo4j Desktop:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Copy the **Connection URI** of the instance you want to connect to. The URI is in the page that opens when clicking on the instance.
3. In Neo4j Desktop, select the **Projects** tab and select an existing project or create a new one.
4. Select the **Add** dropdown and choose **Remote connection**.
5. Enter a name for the instance and enter the URL from the Neo4j Aura console from the second step. Once complete, select **Next**.
6. With **Username/Password** selected, enter your credentials and select **Next**. These are the same credentials you stored when you [created the instance](#).
7. When available, activate the connection by clicking the **Connect** button.



Notes:

- Neo4j Desktop only allows 1 connection at a time to a database (local or remote).
- Deactivating an instance in Neo4j Desktop won't shut it down or stop a remote instance - it will only temporarily close the connection to it in Neo4j Desktop.

As with other databases in Neo4j Desktop, you can install [Graph Apps](#) for monitoring and other functionality. To do this, follow the same process to install the graph application you need, and open it from Neo4j Desktop or a web browser with the running and activated Neo4j AuraDS instance.

Neo4j Cypher Shell

You can connect to an AuraDS instance using the Neo4j Cypher Shell command-line interface (CLI) and run Cypher commands against your instance from the command line. Refer to the [Operations manual](#) for instructions on how to install the Cypher Shell.

To connect to an AuraDS instance using Neo4j Cypher Shell:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Copy the **Connection URI** of the instance you want to connect to. The URI is in the page that opens when clicking on the instance.
3. Open a terminal and navigate to the folder where you have installed the Cypher Shell.
4. Run the following `cypher-shell` command replacing:
 - `<connection_uri>` with the URI you copied in step 2
 - `<username>` with the username for your instance
 - `<password>` with the password for your instance

```
./cypher-shell -a <connection_uri> -u <username> -p <password>
```

Once connected, you can run `:help` for a list of available commands.

For more information on Cypher Shell, including how to install it, see the [Cypher Shell documentation](#).

Connecting with Python



Follow along with a notebook in [Google Colab](#)

This tutorial shows how to interact with AuraDS using the [Python Driver](#). In the following sections you can switch between client and driver code clicking on the appropriate tab.

A running AuraDS instance must be available along with access credentials (generated in the [Creating an AuraDS instance](#) section) and its connection URI (found in the instance detail page, starting with `neo4j+s://`).

Installation

Both the GDS client and the Python driver can be installed using `pip`.

```
pip install graphdatascience
```

The latest stable version of the client can be found on [PyPI](#).

```
pip install neo4j
```

The latest stable version of the driver can be found on [PyPI](#).

If `pip` is not available, you can try replacing it with `python -m pip` or `python3 -m pip`.

Import and setup

Both the GDS client and the Python driver require the connection URI and the credentials as shown in the introduction.

The client is imported as the `GraphDataScience` class:

```
# Client import
from graphdatascience import GraphDataScience
```

The `aura_ds=True` constructor argument should be used to have the recommended non-default configuration settings of the Python Driver applied automatically.

```
# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = "..."

# Client instantiation
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

The driver is imported as the `GraphDatabase` class:

```
# Driver import
from neo4j import GraphDatabase
```

```
# Replace with the actual URI, username and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = "..."

# Driver instantiation
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

Running a query

Once created, the client (or the driver) can be used to run Cypher queries and call Cypher procedures. In this example the `gds.version` procedure can be used to retrieve the version of GDS running on the instance.

```
# Call a GDS method directly
print(gds.version())
```

```
# Cypher query
gds_version_query = """
    RETURN gds.version() AS version
"""

# Create a driver session
with driver.session() as session:
    # Use .data() to access the results array
    results = session.run(gds_version_query).data()
    print(results)
```

The following code retrieves all the procedures available in the library and shows the details of five of them.

```
# Assign the result of the client call to a variable
results = gds.list()

# Print the result (a Pandas DataFrame)
print(results[:5])
```

Since the result is a Pandas DataFrame, you can use methods such as `to_string` and `to_json` to pretty-print it.

```
# Print the result (a Pandas DataFrame) as a console-friendly string
print(results[:5].to_string())
```

```
# Print the result (a Pandas DataFrame) as a prettified JSON string
print(results[:5].to_json(orient="table", indent=2))
```

```
# Import the json module for pretty visualization
import json

# Cypher query
list_all_gds_procedures_query = """
    CALL gds.list()
"""

# Create a driver session
with driver.session() as session:
    # Use .data() to access the results array
    results = session.run(list_all_gds_procedures_query).data()

    # Print the prettified result
    print(json.dumps(results[:5], indent=2))
```


Serializing Neo4j `DateTime` in JSON dumps

In some cases the result of a procedure call may contain Neo4j `DateTime` objects. In order to serialize such objects into JSON, a default handler must be provided.

```
# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()

# Run the graph generation algorithm
g, _ = gds.beta.graph.generate(
    "example-graph", 10, 3, relationshipDistribution="POWER_LAW"
)

# Drop the graph keeping the result of the operation, which contains
# some DateTime fields ("creationTime" and "modificationTime")
result = gds.graph.drop(g)

# Print the result as JSON, converting the DateTime fields with
# the handler defined above
print(result.to_json(indent=2, default_handler=default))
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()

# Example query to run a graph generation algorithm
create_example_graph_query = """
    CALL gds.beta.graph.generate(
        'example-graph', 10, 3, {relationshipDistribution: 'POWER_LAW'}
    )
"""

# Example query to delete a graph
delete_example_graph_query = """
    CALL gds.graph.drop('example-graph')
"""

# Create the driver session
with driver.session() as session:
    # Run the graph generation algorithm
    session.run(create_example_graph_query).data()

    # Drop the generated graph keeping the result of the operation
    results = session.run(delete_example_graph_query).data()

# Prettify the results using the handler defined above
print(json.dumps(results, indent=2, sort_keys=True, default=default))
```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```
# Close the client connection  
gds.close()
```

```
# Close the driver connection  
driver.close()
```

References

Documentation

- [Neo4j GDS documentation](#)
- [Neo4j driver documentation](#)
- [Neo4j developer documentation](#)

Cypher

- Learn more about the [Cypher](#) syntax
- You can use the [Cypher Cheat Sheet](#) as a reference of all available Cypher features

Modelling

- [Graph modeling guidelines](#)
- [Modeling designs](#)
- [Graph model refactoring](#)

Usage examples

Projecting graphs and using the graph catalog



Follow along with a notebook in [Google Colab](#)

This example shows how to:

- load Neo4j on-disk data into in-memory projected graphs;
- use the [graph catalog](#) to manage projected graphs.

Setup

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install graphdatascience
```

```
# Import the client
from graphdatascience import GraphDataScience

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Configure the client with AuraDS-recommended settings
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

In the following code examples we use the `print` function to print Pandas `DataFrame` and `Series` objects. You can try different ways to print a Pandas object, for instance via the `to_string` and `to_json` methods; if you use a JSON representation, in some cases you may need to include a [default handler](#) to handle Neo4j `DateTime` objects. Check the [Python connection](#) section for some examples.

For more information on how to get started using the Cypher Shell, refer to the [Neo4j Cypher Shell](#) tutorial.



Run the following commands from the directory where the Cypher shell is installed.

```
export AURA_CONNECTION_URI="neo4j+s://xxxxxxx.databases.neo4j.io"
export AURA_USERNAME="neo4j"
export AURA_PASSWORD=""

./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD
```

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install neo4j
```

```
# Import the driver
from neo4j import GraphDatabase

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Instantiate the driver
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()
```

Load data from Neo4j with native projections

Native projections are used to load into memory a graph stored on disk. The `gds.graph.project` procedure allows to project a graph by selecting the node labels, relationship types and properties to be projected.

The `gds.graph.project` procedure can use a "shorthand syntax", where the nodes and relationships projections are simply passed as single values or arrays, or an "extended syntax", where each node or relationship projection has its own configuration. The extended syntax is especially useful if additional transformation of the data or the graph structure are needed. Both methods are shown in this section, using the following graph as an example.

```
# Cypher query to create an example graph on disk
gds.run_cypher("""
MERGE (a:EngineeringManagement {name: 'Alistair'})
MERGE (j:EngineeringManagement {name: 'Jennifer'})
MERGE (d:Developer {name: 'Leila'})
MERGE (a)-[:MANAGES {start_date: 987654321}]->(d)
MERGE (j)-[:MANAGES {start_date: 123456789, end_date: 987654321}]->(d)
""")
```

```
MERGE (a:EngineeringManagement {name: 'Alistair'})
MERGE (j:EngineeringManagement {name: 'Jennifer'})
MERGE (d:Developer {name: 'Leila'})
MERGE (a)-[:MANAGES {start_date: 987654321}]->(d)
MERGE (j)-[:MANAGES {start_date: 123456789, end_date: 987654321}]->(d)
```

```
# Cypher query to create an example graph on disk
write_example_graph_query = """
MERGE (a:EngineeringManagement {name: 'Alistair'})
MERGE (j:EngineeringManagement {name: 'Jennifer'})
MERGE (d:Developer {name: 'Leila'})
MERGE (a)-[:MANAGES {start_date: 987654321}]->(d)
MERGE (j)-[:MANAGES {start_date: 123456789, end_date: 987654321}]->(d)
"""

# Create the driver session
with driver.session() as session:
    session.run(write_example_graph_query)
```

Project using the shorthand syntax

In this example we use the shorthand syntax to simply project all node labels and relationship types.

```
# Project a graph using the shorthand syntax
shorthand_graph, result = gds.graph.project(
    "shorthand-example-graph",
    ["EngineeringManagement", "Developer"],
    ["MANAGES"]
)

print(result)
```

```
CALL gds.graph.project(
    'shorthand-example-graph',
    ['EngineeringManagement', 'Developer'],
    ['MANAGES']
)
YIELD graphName, nodeCount, relationshipCount
RETURN *
```

```
shorthand_graph_create_call = """
    CALL gds.graph.project(
        'shorthand-example-graph',
        ['EngineeringManagement', 'Developer'],
        ['MANAGES']
    )
    YIELD graphName, nodeCount, relationshipCount
    RETURN *
"""

# Create the driver session
with driver.session() as session:
    # Call to project a graph using the shorthand syntax
    result = session.run(shorthand_graph_create_call).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))
```

Project using the extended syntax

In this example we use the extended syntax for [relationship](#) projections to:

- transform the **EngineeringManagement** and **Developer** labels to **PersonEM** and **PersonD** respectively;
- transform the *directed* **MANAGES** relationship into the **KNOWS** *undirected* relationship;
- keep the **start_date** and **end_date** relationship properties, adding a default value of **999999999** to **end_date**.

The projected graph becomes the following:

```
(:PersonEM {first_name: 'Alistair'})-
[:KNOWS {start_date: 987654321, end_date: 999999999}]-
(:PersonD {first_name: 'Leila'})-
[:KNOWS {start_date: 123456789, end_date: 987654321}]-
(:PersonEM {first_name: 'Jennifer'})
```

```

# Project a graph using the extended syntax
extended_form_graph, result = gds.graph.project(
    "extended-form-example-graph",
    {
        "PersonEM": {
            "label": "EngineeringManagement"
        },
        "PersonD": {
            "label": "Developer"
        }
    },
    {
        "KNOWS": {
            "type": "MANAGES",
            "orientation": "UNDIRECTED",
            "properties": {
                "start_date": {
                    "property": "start_date"
                },
                "end_date": {
                    "property": "end_date",
                    "defaultValue": 999999999
                }
            }
        }
    }
)

print(result)

```

```

CALL gds.graph.project(
    'extended-form-example-graph',
    {
        PersonEM: {
            label: 'EngineeringManagement'
        },
        PersonD: {
            label: 'Developer'
        }
    },
    {
        KNOWS: {
            type: 'MANAGES',
            orientation: 'UNDIRECTED',
            properties: {
                start_date: {
                    property: 'start_date'
                },
                end_date: {
                    property: 'end_date',
                    defaultValue: 999999999
                }
            }
        }
    }
)
YIELD graphName, nodeCount, relationshipCount
RETURN *

```



```

extended_form_graph_create_call = """
CALL gds.graph.project(
  'extended-form-example-graph',
  {
    PersonEM: {
      label: 'EngineeringManagement'
    },
    PersonD: {
      label: 'Developer'
    }
  },
  {
    KNOWS: {
      type: 'MANAGES',
      orientation: 'UNDIRECTED',
      properties: {
        start_date: {
          property: 'start_date'
        },
        end_date: {
          property: 'end_date',
          defaultValue: 999999999
        }
      }
    }
  }
)
YIELD graphName, nodeCount, relationshipCount
RETURN *
"""

# Create the driver session
with driver.session() as session:
    # Call to project a graph using the extended syntax
    result = session.run(extended_form_graph_create_call).data()

    # Prettify the results
    print(json.dumps(result, indent=2, sort_keys=True))

```

Use the graph catalog

The graph catalog can be used to retrieve information on and manage the projected graphs.

List all the graphs

The `gds.graph.list` procedure can be used to list all the graphs currently stored in memory.

```
# List all in-memory graphs
all_graphs = gds.graph.list()

print(all_graphs)
```

```
CALL gds.graph.list()
```

```
show_in_memory_graphs_call = """
    CALL gds.graph.list()
"""

# Create the driver session
with driver.session() as session:
    # Run the Cypher procedure
    results = session.run(show_in_memory_graphs_call).data()

    # Prettify the results
    print(json.dumps(results, indent=2, sort_keys=True, default=default))
```

Check that a graph exists

The `gds.graph.exists` procedure can be called to check for the existence of a graph by its name.

```
# Check whether the "shorthand-example-graph" graph exists in memory
graph_exists = gds.graph.exists("shorthand-example-graph")

print(graph_exists)
```

```
CALL gds.graph.exists('example-graph')
```

```
check_graph_exists_call = """
    CALL gds.graph.exists('example-graph')
"""

# Create the driver session
with driver.session() as session:
    # Run the Cypher procedure and print the result
    print(session.run(check_graph_exists_call).data())
```

Drop a graph

When a graph is no longer needed, it can be dropped to free up memory using the `gds.graph.drop` procedure.

```
# Drop a graph object and keep the result of the call
result = gds.graph.drop(shorthand_graph)

# Print the result
print(result)

# Drop a graph object and just print the result of the call
gds.graph.drop(extended_form_graph)
```

```
CALL gds.graph.drop('shorthand-example-graph');
CALL gds.graph.drop('extended-form-example-graph');
```

```
delete_shorthand_graph_call = """
    CALL gds.graph.drop('shorthand-example-graph')
"""

delete_extended_form_graph_call = """
    CALL gds.graph.drop('extended-form-example-graph')
"""

# Create the driver session
with driver.session() as session:
    # Drop a graph and keep the result of the call
    result = session.run(delete_shorthand_graph_call).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True, default=default))

    # Drop a graph discarding the result of the call
    session.run(delete_extended_form_graph_call).data()
```

Cleanup

When the projected graphs are dropped, the underlying data on the disk are not deleted. If such data are no longer needed, they need to be deleted manually via a Cypher query.

```
# Delete on-disk data
gds.run_cypher("""
MATCH (example)
WHERE example:EngineeringManagement OR example:Developer
DETACH DELETE example
""")
```

```
MATCH (example)
WHERE example:EngineeringManagement OR example:Developer
DETACH DELETE example;
```

```
delete_example_graph_query = """
MATCH (example)
WHERE example:EngineeringManagement OR example:Developer
DETACH DELETE example
"""

# Create the driver session
with driver.session() as session:
    # Run Cypher call
    print(session.run(delete_example_graph_query).data())
```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```
# Close the client connection
gds.close()
```

```
# Close the driver connection
driver.close()
```

References

Documentation

- [Neo4j GDS documentation](#)
- [Neo4j driver documentation](#)

- [Neo4j developer documentation](#)

Cypher

- Learn more about the [Cypher](#) syntax
- You can use the [Cypher Cheat Sheet](#) as a reference of all available Cypher features

Modelling

- [Graph modeling guidelines](#)
- [Modeling designs](#)
- [Graph model refactoring](#)

Executing the different algorithm modes



Follow along with a notebook in [Google Colab](#)

This example explains [execution modes](#) for GDS algorithms and how to use each one of them.

Setup

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install graphdatascience
```

```
# Import the client
from graphdatascience import GraphDataScience

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Configure the client with AuraDS-recommended settings
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

In the following code examples we use the `print` function to print Pandas `DataFrame` and `Series` objects. You can try different ways to print a Pandas object, for instance via the `to_string` and `to_json` methods; if you use a JSON representation, in some cases you may need to include a `default handler` to handle Neo4j `DateTime` objects. Check the [Python connection](#) section for some examples.

For more information on how to get started using the Cypher Shell, refer to the [Neo4j Cypher Shell](#) tutorial.



Run the following commands from the directory where the Cypher shell is installed.

```
export AURA_CONNECTION_URI="neo4j+s://xxxxxxx.databases.neo4j.io"
export AURA_USERNAME="neo4j"
export AURA_PASSWORD=""

./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD
```

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install neo4j
```

```
# Import the driver
from neo4j import GraphDatabase

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Instantiate the driver
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()
```

Create an example graph

We start by creating some basic graph data first.

```

gds.run_cypher("""
CREATE
  (home:Page {name:'Home'}),
  (about:Page {name:'About'}),
  (product:Page {name:'Product'}),
  (links:Page {name:'Links'}),
  (a:Page {name:'Site A'}),
  (b:Page {name:'Site B'}),
  (c:Page {name:'Site C'}),
  (d:Page {name:'Site D'}),

  (home)-[:LINKS {weight: 0.2}]->(about),
  (home)-[:LINKS {weight: 0.2}]->(links),
  (home)-[:LINKS {weight: 0.6}]->(product),
  (about)-[:LINKS {weight: 1.0}]->(home),
  (product)-[:LINKS {weight: 1.0}]->(home),
  (a)-[:LINKS {weight: 1.0}]->(home),
  (b)-[:LINKS {weight: 1.0}]->(home),
  (c)-[:LINKS {weight: 1.0}]->(home),
  (d)-[:LINKS {weight: 1.0}]->(home),
  (links)-[:LINKS {weight: 0.8}]->(home),
  (links)-[:LINKS {weight: 0.05}]->(a),
  (links)-[:LINKS {weight: 0.05}]->(b),
  (links)-[:LINKS {weight: 0.05}]->(c),
  (links)-[:LINKS {weight: 0.05}]->(d)
""")

```

```

CREATE
  (home:Page {name:'Home'}),
  (about:Page {name:'About'}),
  (product:Page {name:'Product'}),
  (links:Page {name:'Links'}),
  (a:Page {name:'Site A'}),
  (b:Page {name:'Site B'}),
  (c:Page {name:'Site C'}),
  (d:Page {name:'Site D'}),

  (home)-[:LINKS {weight: 0.2}]->(about),
  (home)-[:LINKS {weight: 0.2}]->(links),
  (home)-[:LINKS {weight: 0.6}]->(product),
  (about)-[:LINKS {weight: 1.0}]->(home),
  (product)-[:LINKS {weight: 1.0}]->(home),
  (a)-[:LINKS {weight: 1.0}]->(home),
  (b)-[:LINKS {weight: 1.0}]->(home),
  (c)-[:LINKS {weight: 1.0}]->(home),
  (d)-[:LINKS {weight: 1.0}]->(home),
  (links)-[:LINKS {weight: 0.8}]->(home),
  (links)-[:LINKS {weight: 0.05}]->(a),
  (links)-[:LINKS {weight: 0.05}]->(b),
  (links)-[:LINKS {weight: 0.05}]->(c),
  (links)-[:LINKS {weight: 0.05}]->(d)

```



```

# Cypher query
create_example_graph_on_disk_query = """
CREATE
  (home:Page {name:'Home'}),
  (about:Page {name:'About'}),
  (product:Page {name:'Product'}),
  (links:Page {name:'Links'}),
  (a:Page {name:'Site A'}),
  (b:Page {name:'Site B'}),
  (c:Page {name:'Site C'}),
  (d:Page {name:'Site D'}),

  (home)-[:LINKS {weight: 0.2}]->(about),
  (home)-[:LINKS {weight: 0.2}]->(links),
  (home)-[:LINKS {weight: 0.6}]->(product),
  (about)-[:LINKS {weight: 1.0}]->(home),
  (product)-[:LINKS {weight: 1.0}]->(home),
  (a)-[:LINKS {weight: 1.0}]->(home),
  (b)-[:LINKS {weight: 1.0}]->(home),
  (c)-[:LINKS {weight: 1.0}]->(home),
  (d)-[:LINKS {weight: 1.0}]->(home),
  (links)-[:LINKS {weight: 0.8}]->(home),
  (links)-[:LINKS {weight: 0.05}]->(a),
  (links)-[:LINKS {weight: 0.05}]->(b),
  (links)-[:LINKS {weight: 0.05}]->(c),
  (links)-[:LINKS {weight: 0.05}]->(d)
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(create_example_graph_on_disk_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))

```

We then project an in-memory graph from the data just created.

```

g, result = gds.graph.project(
    "example-graph",
    "Page",
    "LINKS",
    relationshipProperties="weight"
)

print(result)

```

```

CALL gds.graph.project(
    'example-graph',
    'Page',
    'LINKS',
    {
        relationshipProperties: 'weight'
    }
)

```

```

# Cypher query
create_example_graph_in_memory_query = """
    CALL gds.graph.project(
        'example-graph',
        'Page',
        'LINKS',
        {
            relationshipProperties: 'weight'
        }
    )
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(create_example_graph_in_memory_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))

```

Execution modes

Every production-tier algorithm can be run in four different modes:

- `stats`
- `stream`
- `mutate`
- `write`

An additional `estimate` mode is explained in detail in the [Estimating memory usage and resizing an instance](#) section.

In the following we'll use the PageRank algorithm to show the usage of every execution mode.

Stats

The `stats` mode can be useful for evaluating an algorithm performance without mutating the in-memory graph. When running an algorithm in this mode, a single row containing a summary of the algorithm statistics (for example, counts or percentile distributions) is returned.

```
result = gds.pageRank.stats(  
    g,  
    maxIterations=20,  
    dampingFactor=0.85  
)  
  
print(result)
```

```
CALL gds.pageRank.stats(  
    'example-graph',  
    {maxIterations: 20, dampingFactor: 0.85}  
)  
YIELD ranIterations,  
    didConverge,  
    preProcessingMillis,  
    computeMillis,  
    postProcessingMillis,  
    centralityDistribution,  
    configuration  
RETURN *
```

```
# Cypher query  
page_rank_stats_example_graph_query = """  
    CALL gds.pageRank.stats(  
        'example-graph',  
        {maxIterations: 20, dampingFactor: 0.85}  
    )  
    YIELD ranIterations,  
        didConverge,  
        preProcessingMillis,  
        computeMillis,  
        postProcessingMillis,  
        centralityDistribution,  
        configuration  
    RETURN *  
    """  
  
# Create the driver session  
with driver.session() as session:  
    # Run query  
    result = session.run(page_rank_stats_example_graph_query).data()  
  
    # Prettyfy the result  
    print(json.dumps(result, indent=2, sort_keys=True))
```

The result contains the estimated time to run the algorithm (`computeMillis`) along with other details like the centrality distribution and the configuration parameters.

Stream

The `stream` mode returns the results of an algorithm as Cypher result rows. This is similar to how standard Cypher reading queries operate.

With the PageRank example, this mode returns a node ID and the computed PageRank score for each node. The `gds.util.asNode` procedure can then be used to find a node from its node ID.

```
results = gds.pageRank.stream(  
  g,  
  maxIterations=20,  
  dampingFactor=0.85  
)  
  
print(results)
```

```
CALL gds.pageRank.stream(  
  'example-graph',  
  {maxIterations: 20, dampingFactor: 0.85}  
)  
YIELD nodeId, score  
RETURN *
```

```
# Cypher query to just get internal node ID and score  
page_rank_stream_example_graph_query = ""  
  CALL gds.pageRank.stream(  
    'example-graph',  
    {maxIterations: 20, dampingFactor: 0.85}  
  )  
  YIELD nodeId, score  
  RETURN *  
""  
  
# Create the driver session  
with driver.session() as session:  
  # Run query  
  results = session.run(page_rank_stream_example_graph_query).data()  
  
  # Prettyfy the results  
  print(json.dumps(results, indent=2, sort_keys=True))
```

Since an algorithm can run for a long time and the connection may suddenly drop, we suggest to use the `mutate` and `write` modes instead to make sure that the computation completes and the results are saved.

Mutate

The `mutate` mode operates on the in-memory graph and updates it with a new property specified with the `mutateProperty` configuration parameter. The new property must not already exist in the in-memory graph.

This mode is useful when chaining the execution of several algorithms each of which relying on the results

on the previous.

In the case of PageRank, the result of this mode is a score for each node. In this example we add the calculated score to each node of the in-memory graph as the value of a new property called `pageRankScore`.

```
result = gds.pageRank.mutate(  
    g,  
    mutateProperty="pageRankScore",  
    maxIterations=20,  
    dampingFactor=0.85  
)  
  
print(result)
```

```
CALL gds.pageRank.mutate(  
    'example-graph',  
    {mutateProperty: 'pageRankScore', maxIterations: 20, dampingFactor: 0.85}  
)  
YIELD nodePropertiesWritten, ranIterations  
RETURN *
```

```
# Cypher query to just get mutate the graph  
page_rank_mutate_example_graph_query = """  
    CALL gds.pageRank.mutate(  
        'example-graph',  
        {mutateProperty: 'pageRankScore', maxIterations: 20, dampingFactor: 0.85}  
    )  
    YIELD nodePropertiesWritten, ranIterations  
    RETURN *  
    """  
  
# Create the driver session  
with driver.session() as session:  
    # Run query  
    result = session.run(page_rank_mutate_example_graph_query).data()  
  
    # Prettify the result  
    print(json.dumps(result, indent=2, sort_keys=True))
```

Write

The `write` mode writes the results of the algorithm computation back to the Neo4j database. The written data can be node properties (such as PageRank scores), new relationships (such as Node Similarity similarities), or relationship properties (only for newly created relationships).

Similarly to the previous example, here we add the calculated score of the PageRank algorithm to each node of the Neo4j database as the value of a new property called `pageRankScore`.



To use the result of a `write` mode computation with another algorithm, a new in-memory graph must be created from the Neo4j database.

```

result = gds.pageRank.write(
    g,
    writeProperty="pageRankScore",
    maxIterations=20,
    dampingFactor=0.85
)

print(result)

```

```

CALL gds.pageRank.write(
    'example-graph',
    {writeProperty: 'pageRankScore', maxIterations: 20, dampingFactor: 0.85}
)
YIELD nodePropertiesWritten, ranIterations
RETURN *

```

```

# Cypher query to write the graph
page_rank_write_example_graph_query = """
    CALL gds.pageRank.write(
        'example-graph',
        {writeProperty: 'pageRankScore', maxIterations: 20, dampingFactor: 0.85}
    )
    YIELD nodePropertiesWritten, ranIterations
    RETURN *
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(page_rank_write_example_graph_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))

```

Cleanup

After going through the example, both the in-memory graphs and the data in the Neo4j database can be deleted.

```

result = gds.graph.drop(g)
print(result)

gds.run_cypher("""
    MATCH (n)
    DETACH DELETE n
""")

```

```

CALL gds.graph.drop('example-graph');

MATCH (n)
DETACH DELETE n;

```

```

delete_example_in_memory_graph_query = """
    CALL gds.graph.drop('example-graph')
"""

delete_example_graph = """
    MATCH (n)
    DETACH DELETE n
"""

with driver.session() as session:
    # Delete in-memory graph
    result = session.run(delete_example_in_memory_graph_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True, default=default))

    # Delete data from Neo4j
    result = session.run(delete_example_graph).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True, default=default))

```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```
# Close the client connection  
gds.close()
```

```
# Close the driver connection  
driver.close()
```

References

Documentation

- [Neo4j GDS documentation](#)
- [Neo4j driver documentation](#)
- [Neo4j developer documentation](#)

Cypher

- Learn more about the [Cypher](#) syntax
- You can use the [Cypher Cheat Sheet](#) as a reference of all available Cypher features

Modelling

- [Graph modeling guidelines](#)
- [Modeling designs](#)
- [Graph model refactoring](#)

Estimating memory usage and resizing an instance



Follow along with a notebook in [Google Colab](#)

This example shows how to:

- use the [memory estimation](#) mode to estimate the memory requirements for an algorithm before running it
- resize an AuraDS instance to accommodate the algorithm memory requirements

Setup

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install graphdatascience
```

```
# Import the client
from graphdatascience import GraphDataScience

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Configure the client with AuraDS-recommended settings
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

In the following code examples we use the `print` function to print Pandas `DataFrame` and `Series` objects. You can try different ways to print a Pandas object, for instance via the `to_string` and `to_json` methods; if you use a JSON representation, in some cases you may need to include a [default handler](#) to handle Neo4j `DateTime` objects. Check the [Python connection](#) section for some examples.

For more information on how to get started using the Cypher Shell, refer to the [Neo4j Cypher Shell](#) tutorial.



Run the following commands from the directory where the Cypher shell is installed.

```
export AURA_CONNECTION_URI="neo4j+s://xxxxxxx.databases.neo4j.io"
export AURA_USERNAME="neo4j"
export AURA_PASSWORD=""

./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD
```

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install neo4j
```

```
# Import the driver
from neo4j import GraphDatabase

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Instantiate the driver
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()
```

Create an example graph

An easy way to create an in-memory graph is through the GDS [graph generation](#) algorithm. By specifying the number of nodes, the average number of relationships going out of each node and the relationship distribution function, the algorithm creates a graph having the following shape:

```
(:50000000_Nodes)-[:REL]->(:50000000_Nodes)
```

```

# Run the graph generation algorithm and retrieve the corresponding
# graph object and call result metadata
g, result = gds.beta.graph.generate(
    "example-graph",
    50000000,
    3,
    relationshipDistribution="POWER_LAW"
)

# Print prettified graph stats
print(result)

```

```

CALL gds.beta.graph.generate(
    'example-graph',
    50000000,
    3,
    {relationshipDistribution: 'POWER_LAW'}
)
YIELD name,
    nodes,
    relationships,
    generateMillis,
    relationshipSeed,
    averageDegree,
    relationshipDistribution,
    relationshipProperty
RETURN *

```

```

# Cypher query
create_example_graph_query = """
    CALL gds.beta.graph.generate(
        'example-graph',
        50000000,
        3,
        {relationshipDistribution: 'POWER_LAW'}
    )
    YIELD name,
        nodes,
        relationships,
        generateMillis,
        relationshipSeed,
        averageDegree,
        relationshipDistribution,
        relationshipProperty
    RETURN *
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(create_example_graph_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True, default=default))

```



The graph is fairly large, so the generation procedure will take a few minutes to complete.

Run the `estimate` mode

The estimation of the memory requirements of an algorithm on an in-memory graph can be useful to determine whether the current AuraDS instance has enough resources to run the algorithm to completion.

The Graph Data Science has guard rails built in: if an algorithm is estimated to use more RAM than is available, an exception is raised. In this case, the AuraDS instance can be resized before running the algorithm again.

In the following example we get a memory estimation for the Label Propagation algorithm to run on the generated graph. The estimated memory is between 381 MiB and 4477 MiB, which is higher than an 8 GB instance has available (4004 MiB).

```
result = gds.labelPropagation.mutate.estimate(  
    g,  
    mutateProperty="communityID"  
)  
  
print(result)
```

```
CALL gds.labelPropagation.mutate.estimate(  
    'example-graph',  
    {mutateProperty: 'communityID'}  
)  
YIELD nodeCount,  
    relationshipCount,  
    bytesMin,  
    bytesMax,  
    requiredMemory  
RETURN *
```

```
# Cypher query  
page_rank_mutate_estimate_example_graph_query = """  
    CALL gds.labelPropagation.mutate.estimate(  
        'example-graph',  
        {mutateProperty: 'communityID'}  
    )  
    YIELD nodeCount,  
        relationshipCount,  
        bytesMin,  
        bytesMax,  
        requiredMemory  
    RETURN *  
    """  
  
# Create the driver session  
with driver.session() as session:  
    # Run query  
    results = session.run(page_rank_mutate_estimate_example_graph_query).data()  
  
    # Prettify the result  
    print(json.dumps(results, indent=2, sort_keys=True))
```

The `mutate` procedure hits the guard rails on an 8 GB instance, raising an exception that suggests to resize

the AuraDS instance.

```
result = gds.labelPropagation.mutate(  
    g,  
    mutateProperty="communityID"  
)  
  
print(result)
```

```
CALL gds.labelPropagation.mutate(  
    'example-graph',  
    {mutateProperty: 'communityID'}  
)  
YIELD preProcessingMillis,  
    computeMillis,  
    mutateMillis,  
    postProcessingMillis,  
    nodePropertiesWritten,  
    communityCount,  
    ranIterations,  
    didConverge,  
    communityDistribution,  
    configuration  
RETURN *
```

```
# Cypher query  
page_rank_mutate_example_graph_query = """  
    CALL gds.labelPropagation.mutate(  
        'example-graph',  
        {mutateProperty: 'communityID'}  
    )  
    YIELD preProcessingMillis,  
        computeMillis,  
        mutateMillis,  
        postProcessingMillis,  
        nodePropertiesWritten,  
        communityCount,  
        ranIterations,  
        didConverge,  
        communityDistribution,  
        configuration  
    RETURN *  
    """  
  
# Create the driver session  
with driver.session() as session:  
    # Run query  
    results = session.run(page_rank_mutate_example_graph_query).data()  
  
    # Prettify the result  
    print(json.dumps(results, indent=2, sort_keys=True))
```

Resize the AuraDS instance

You will need to resize the instance to the next available size (16 GB) in order to continue. An AuraDS instance can be resized from the [Neo4j Aura Console](#) homepage. For more information, check the [Instance actions](#) section.



Resizing an AuraDS instance incurs a short amount of downtime.

After resizing, wait a few seconds until the projected graph is reloaded, then run the `mutate` step again. This time no exception is thrown and the step completes successfully.

Cleanup

The in-memory graph can now be deleted.

```
result = gds.graph.drop(g)
print(result)
```

```
CALL gds.graph.drop('example-graph')
```

```
delete_example_in_memory_graph_query = """
CALL gds.graph.drop('example-graph')
"""

with driver.session() as session:
    # Run query
    results = session.run(delete_example_in_memory_graph_query).data()

    # Prettify the results
    print(json.dumps(results, indent=2, sort_keys=True, default=default))
```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```
# Close the client connection
gds.close()
```

```
# Close the driver connection
driver.close()
```

References

Documentation

- [Neo4j GDS documentation](#)
- [Neo4j driver documentation](#)
- [Neo4j developer documentation](#)

Cypher

- Learn more about the [Cypher](#) syntax
- You can use the [Cypher Cheat Sheet](#) as a reference of all available Cypher features

Modelling

- [Graph modeling guidelines](#)
- [Modeling designs](#)
- [Graph model refactoring](#)

Monitoring the progress of a running algorithm



Follow along with a notebook in [Google Colab](#)

Running algorithms on large graphs can be computationally expensive. This example shows how to use the `gds.beta.listProgress` procedure to monitor the progress of an algorithm, both to get an idea of the processing speed and to determine when the computation is completed.

Setup

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install graphdatascience
```

```
# Import the client
from graphdatascience import GraphDataScience

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Configure the client with AuraDS-recommended settings
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

In the following code examples we use the `print` function to print Pandas `DataFrame` and `Series` objects. You can try different ways to print a Pandas object, for instance via the `to_string` and `to_json` methods; if you use a JSON representation, in some cases you may need to include a [default handler](#) to handle Neo4j `DateTime` objects. Check the [Python connection](#) section for some examples.

For more information on how to get started using the Cypher Shell, refer to the [Neo4j Cypher Shell](#) tutorial.



Run the following commands from the directory where the Cypher shell is installed.

```
export AURA_CONNECTION_URI="neo4j+s://xxxxxxx.databases.neo4j.io"
export AURA_USERNAME="neo4j"
export AURA_PASSWORD=""

./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD
```

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install neo4j
```

```
# Import the driver
from neo4j import GraphDatabase

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Instantiate the driver
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()
```

Create an example graph

An easy way to create an in-memory graph is through the GDS [graph generation](#) algorithm. By specifying the number of nodes, the average number of relationships going out of each node and the relationship distribution function, the algorithm creates a graph having the following shape:

```
(:1000000_Nodes)-[:REL]->(:1000000_Nodes)
```

```

# Run the graph generation algorithm and retrieve the corresponding
# graph object and call result metadata
g, result = gds.beta.graph.generate(
    "example-graph",
    1000000,
    3,
    relationshipDistribution="POWER_LAW"
)

# Print prettified graph stats
print(result)

```

```

CALL gds.beta.graph.generate(
    'example-graph',
    1000000,
    3,
    {relationshipDistribution: 'POWER_LAW'}
)
YIELD name,
    nodes,
    relationships,
    generateMillis,
    relationshipSeed,
    averageDegree,
    relationshipDistribution,
    relationshipProperty
RETURN *

```

```

# Cypher query
create_example_graph_query = """
    CALL gds.beta.graph.generate(
        'example-graph',
        1000000,
        3,
        {relationshipDistribution: 'POWER_LAW'}
    )
    YIELD name,
        nodes,
        relationships,
        generateMillis,
        relationshipSeed,
        averageDegree,
        relationshipDistribution,
        relationshipProperty
    RETURN *
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(create_example_graph_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True, default=default))

```

Run an algorithm and check the progress

We need to run an algorithm that takes some time to converge. In this example we use the Label

Propagation algorithm, which we start in a separate thread so that we can check its progress in the same Python process.

```

# Import to run the long-running algorithm in a thread
import threading
# Import to use the sleep method
import time

# Method to call the label propagation algorithm from a thread
def run_label_prop():
    print("Running label propagation")

    result = gds.labelPropagation.mutate(
        g,
        mutateProperty="communityID"
    )

    print(result)

# Method to get and pretty-print the algorithm progress
def run_list_progress():
    result = gds.beta.listProgress()

    print(result)

# Create a thread for the label propagation algorithm and start it
label_prop_query_thread = threading.Thread(target=run_label_prop)
label_prop_query_thread.start()

# Sleep for a few seconds so the label propagation query has time to get going
print('Sleeping for 5 seconds')
time.sleep(5)

# Check the algorithm progress
run_list_progress()

# Sleep for a few more seconds
print('Sleeping for 10 more seconds')
time.sleep(10)

# Check the algorithm progress again
run_list_progress()

# Block and wait for the algorithm thread to finish
label_prop_query_thread.join()

```

```
CALL gds.labelPropagation.mutate(  
  'example-graph',  
  {mutateProperty: 'communityID'}  
)  
YIELD preProcessingMillis,  
  computeMillis,  
  mutateMillis,  
  postProcessingMillis,  
  nodePropertiesWritten,  
  communityCount,  
  ranIterations,  
  didConverge,  
  communityDistribution,  
  configuration  
RETURN *  
  
// The following query has to be run in another Cypher shell, so run this command  
// in a different terminal first:  
//  
// ./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD  
  
CALL gds.beta.listProgress()  
YIELD jobId, taskName, progress, progressBar  
RETURN *
```

```

# Import to run the long-running algorithm in a thread
import threading
# Import to use the sleep method
import time

# Method to call the label propagation algorithm from a thread
def run_label_prop():
    label_prop_mutate_example_graph_query = """
        CALL gds.labelPropagation.mutate(
            'example-graph',
            {mutateProperty: 'communityID'}
        )
        YIELD preProcessingMillis,
            computeMillis,
            mutateMillis,
            postProcessingMillis,
            nodePropertiesWritten,
            communityCount,
            ranIterations,
            didConverge,
            communityDistribution,
            configuration
    """
    RETURN *

# Create the driver session
with driver.session() as session:
    # Run query
    print("Running label propagation")
    results = session.run(label_prop_mutate_example_graph_query).data()
    # Prettify the first result
    print(json.dumps(results[0], indent=2, sort_keys=True))

# Method to get and pretty-print the algorithm progress
def run_list_progress():
    gds_list_progress_query = """
        CALL gds.beta.listProgress()
        YIELD jobId, taskName, progress, progressBar
    """
    RETURN *

# Create the driver session
with driver.session() as session:
    # Run query
    print('running list progress')
    results = session.run(gds_list_progress_query).data()
    # Prettify the first result
    print('list progress results: ')
    print(json.dumps(results[0], indent=2, sort_keys=True))

# Create a thread for the label propagation algorithm and start it
label_prop_query_thread = threading.Thread(target=run_label_prop)
label_prop_query_thread.start()

# Sleep for a few seconds so the label propagation query has time to get going
print('Sleeping for 5 seconds')
time.sleep(5)

# Check the algorithm progress
run_list_progress()

# Sleep for a few more seconds
print('Sleeping for 10 more seconds')
time.sleep(10)

# Check the algorithm progress again
run_list_progress()

# Block and wait for the algorithm thread to finish
label_prop_query_thread.join()

```

Cleanup

The in-memory graph can now be deleted.

```
result = gds.graph.drop(g)
print(result)
```

```
CALL gds.graph.drop('example-graph')
```

```
delete_example_in_memory_graph_query = """
CALL gds.graph.drop('example-graph')
"""

with driver.session() as session:
    # Run query
    results = session.run(delete_example_in_memory_graph_query).data()

    # Prettify the results
    print(json.dumps(results, indent=2, sort_keys=True, default=default))
```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```
# Close the client connection
gds.close()
```

```
# Close the driver connection
driver.close()
```

References

Documentation

- [Neo4j GDS documentation](#)

- [Neo4j driver documentation](#)
- [Neo4j developer documentation](#)

Cypher

- Learn more about the [Cypher](#) syntax
- You can use the [Cypher Cheat Sheet](#) as a reference of all available Cypher features

Modelling

- [Graph modeling guidelines](#)
- [Modeling designs](#)
- [Graph model refactoring](#)

Persisting and sharing machine learning models



Follow along with a notebook in [Google Colab](#)

This example shows how to train, save, publish, and drop a machine learning model using the [Model Catalog](#).

Setup

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install graphdatascience
```

```
# Import the client
from graphdatascience import GraphDataScience

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Configure the client with AuraDS-recommended settings
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

In the following code examples we use the `print` function to print Pandas `DataFrame` and `Series` objects. You can try different ways to print a Pandas object, for instance via the `to_string` and `to_json` methods; if you use a JSON representation, in some cases you may need to include a `default handler` to handle Neo4j `DateTime` objects. Check the [Python connection](#) section for some examples.

For more information on how to get started using the Cypher Shell, refer to the [Neo4j Cypher Shell](#) tutorial.



Run the following commands from the directory where the Cypher shell is installed.

```
export AURA_CONNECTION_URI="neo4j+s://xxxxxxx.databases.neo4j.io"
export AURA_USERNAME="neo4j"
export AURA_PASSWORD=""

./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD
```

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install neo4j
```

```
# Import the driver
from neo4j import GraphDatabase

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Instantiate the driver
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()
```

Create an example graph

We start by creating some basic graph data first.

```

gds.run_cypher("""
MERGE (dan:Person:ExampleData {name: 'Dan', age: 20, heightAndWeight: [185, 75]})
MERGE (annie:Person:ExampleData {name: 'Annie', age: 12, heightAndWeight: [124, 42]})
MERGE (matt:Person:ExampleData {name: 'Matt', age: 67, heightAndWeight: [170, 80]})
MERGE (jeff:Person:ExampleData {name: 'Jeff', age: 45, heightAndWeight: [192, 85]})
MERGE (brie:Person:ExampleData {name: 'Brie', age: 27, heightAndWeight: [176, 57]})
MERGE (elsa:Person:ExampleData {name: 'Elsa', age: 32, heightAndWeight: [158, 55]})
MERGE (john:Person:ExampleData {name: 'John', age: 35, heightAndWeight: [172, 76]})

MERGE (dan)-[:KNOWS {relWeight: 1.0}]->(annie)
MERGE (dan)-[:KNOWS {relWeight: 1.6}]->(matt)
MERGE (annie)-[:KNOWS {relWeight: 0.1}]->(matt)
MERGE (annie)-[:KNOWS {relWeight: 3.0}]->(jeff)
MERGE (annie)-[:KNOWS {relWeight: 1.2}]->(brie)
MERGE (matt)-[:KNOWS {relWeight: 10.0}]->(brie)
MERGE (brie)-[:KNOWS {relWeight: 1.0}]->(elsa)
MERGE (brie)-[:KNOWS {relWeight: 2.2}]->(jeff)
MERGE (john)-[:KNOWS {relWeight: 5.0}]->(jeff)

RETURN True AS exampleDataCreated
""")

```

```

MERGE (dan:Person:ExampleData {name: 'Dan', age: 20, heightAndWeight: [185, 75]})
MERGE (annie:Person:ExampleData {name: 'Annie', age: 12, heightAndWeight: [124, 42]})
MERGE (matt:Person:ExampleData {name: 'Matt', age: 67, heightAndWeight: [170, 80]})
MERGE (jeff:Person:ExampleData {name: 'Jeff', age: 45, heightAndWeight: [192, 85]})
MERGE (brie:Person:ExampleData {name: 'Brie', age: 27, heightAndWeight: [176, 57]})
MERGE (elsa:Person:ExampleData {name: 'Elsa', age: 32, heightAndWeight: [158, 55]})
MERGE (john:Person:ExampleData {name: 'John', age: 35, heightAndWeight: [172, 76]})

MERGE (dan)-[:KNOWS {relWeight: 1.0}]->(annie)
MERGE (dan)-[:KNOWS {relWeight: 1.6}]->(matt)
MERGE (annie)-[:KNOWS {relWeight: 0.1}]->(matt)
MERGE (annie)-[:KNOWS {relWeight: 3.0}]->(jeff)
MERGE (annie)-[:KNOWS {relWeight: 1.2}]->(brie)
MERGE (matt)-[:KNOWS {relWeight: 10.0}]->(brie)
MERGE (brie)-[:KNOWS {relWeight: 1.0}]->(elsa)
MERGE (brie)-[:KNOWS {relWeight: 2.2}]->(jeff)
MERGE (john)-[:KNOWS {relWeight: 5.0}]->(jeff)

RETURN True AS exampleDataCreated

```

```

# Cypher query
create_example_graph_on_disk_query = """
MERGE (dan:Person:ExampleData {name: 'Dan', age: 20, heightAndWeight: [185, 75]})
MERGE (annie:Person:ExampleData {name: 'Annie', age: 12, heightAndWeight: [124, 42]})
MERGE (matt:Person:ExampleData {name: 'Matt', age: 67, heightAndWeight: [170, 80]})
MERGE (jeff:Person:ExampleData {name: 'Jeff', age: 45, heightAndWeight: [192, 85]})
MERGE (brie:Person:ExampleData {name: 'Brie', age: 27, heightAndWeight: [176, 57]})
MERGE (elsa:Person:ExampleData {name: 'Elsa', age: 32, heightAndWeight: [158, 55]})
MERGE (john:Person:ExampleData {name: 'John', age: 35, heightAndWeight: [172, 76]})

MERGE (dan)-[:KNOWS {relWeight: 1.0}]->(annie)
MERGE (dan)-[:KNOWS {relWeight: 1.6}]->(matt)
MERGE (annie)-[:KNOWS {relWeight: 0.1}]->(matt)
MERGE (annie)-[:KNOWS {relWeight: 3.0}]->(jeff)
MERGE (annie)-[:KNOWS {relWeight: 1.2}]->(brie)
MERGE (matt)-[:KNOWS {relWeight: 10.0}]->(brie)
MERGE (brie)-[:KNOWS {relWeight: 1.0}]->(elsa)
MERGE (brie)-[:KNOWS {relWeight: 2.2}]->(jeff)
MERGE (john)-[:KNOWS {relWeight: 5.0}]->(jeff)

RETURN True AS exampleDataCreated
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(create_example_graph_on_disk_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))

```

We then project an in-memory graph from the data just created.

```

g, result = gds.graph.project(
    "example_graph_for_graphsage",
    {
        "Person": {
            "label": "ExampleData",
            "properties": ["age", "heightAndWeight"]
        }
    },
    {
        "KNOWS": {
            "type": "KNOWS",
            "orientation": "UNDIRECTED",
            "properties": ["relWeight"]
        }
    }
)

print(result)

```

```

CALL gds.graph.project(
    'example_graph_for_graphsage',
    {
        Person: {
            label: 'ExampleData',
            properties: ['age', 'heightAndWeight']
        }
    },
    {
        KNOWS: {
            type: 'KNOWS',
            orientation: 'UNDIRECTED',
            properties: ['relWeight']
        }
    }
)

```

```

# Cypher query
create_example_graph_in_memory_query = """
    CALL gds.graph.project(
      'example_graph_for_graphsage',
      {
        Person: {
          label: 'ExampleData',
          properties: ['age', 'heightAndWeight']
        }
      },
      {
        KNOWS: {
          type: 'KNOWS',
          orientation: 'UNDIRECTED',
          properties: ['relWeight']
        }
      }
    )
"""

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(create_example_graph_in_memory_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))

```

Train a model

Machine learning algorithms that support the `train` mode produce trained models which are stored in the Model Catalog. Similarly, `predict` procedures can use such trained models to produce predictions. In this example we train a model for the [GraphSAGE algorithm](#) using the `train` mode.

```

model, result = gds.beta.graphSage.train(
    g,
    modelName="example_graph_model_for_graphsage",
    featureProperties=["age", "heightAndWeight"],
    aggregator="mean",
    activationFunction="sigmoid",
    sampleSizes=[25, 10]
)

```

```

CALL gds.beta.graphSage.train(
    'example_graph_for_graphsage',
    {
        modelName: 'example_graph_model_for_graphsage',
        featureProperties: ['age', 'heightAndWeight'],
        aggregator: 'mean',
        activationFunction: 'sigmoid',
        sampleSizes: [25, 10]
    }
)
YIELD modelInfo as info
RETURN
    info.name as modelName,
    info.metrics.didConverge as didConverge,
    info.metrics.ranEpochs as ranEpochs,
    info.metrics.epochLosses as epochLosses

```

```

# Cypher query
train_graph_sage_on_in_memory_graph_query = """
    CALL gds.beta.graphSage.train(
        'example_graph_for_graphsage',
        {
            modelName: 'example_graph_model_for_graphsage',
            featureProperties: ['age', 'heightAndWeight'],
            aggregator: 'mean',
            activationFunction: 'sigmoid',
            sampleSizes: [25, 10]
        }
    )
    YIELD modelInfo as info
    RETURN
        info.name as modelName,
        info.metrics.didConverge as didConverge,
        info.metrics.ranEpochs as ranEpochs,
        info.metrics.epochLosses as epochLosses
    """

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(train_graph_sage_on_in_memory_graph_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))

```

View the model catalog

We can use the `gds.beta.model.list` procedure to get information on all the models currently available in the catalog. Along with information on the graph schema, the model name, and the training configuration,

the result of the call contains the following fields:

- **loaded**: flag denoting if the model is in memory (**true**) or available on disk (**false**)
- **stored**: flag denoting whether the model has been persisted to disk
- **shared**: flag denoting whether the model has been published, making it accessible to all users

```
results = gds.beta.model.list()
print(results)
```

```
CALL gds.beta.model.list()
```

```
# Cypher query
list_model_catalog_query = """
    CALL gds.beta.model.list()
    """

# Create the driver session
with driver.session() as session:
    # Run query
    results = session.run(list_model_catalog_query).data()

    # Prettify the results
    print(json.dumps(results, indent=2, sort_keys=True, default=default))
```

Save a model to disk

The `gds.alpha.model.store` procedure can be used to persist a model to disk. This is useful both to keep models for later reuse and to free up memory.



Not all the models can be saved to disk. A list of the supported models can be found on the [GDS manual](#).

If a model cannot be saved to disk, it will be lost when the AuraDS instance is restarted.

```
result = gds.alpha.model.store(model)
print(result)
```

```
CALL gds.alpha.model.store("example_graph_model_for_graphsage")
```

```
# Cypher query
save_graph_sage_model_to_disk_query = """
    CALL gds.alpha.model.store("example_graph_model_for_graphsage")
    """

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(save_graph_sage_model_to_disk_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True))
```

If we list the model catalog again after persisting a model, we can see that the `stored` flag for that model has been set to `true`.

```
results = gds.beta.model.list()
print(results)
```

```
CALL gds.beta.model.list()
```

```
# Cypher query
list_model_catalog_query = """
    CALL gds.beta.model.list()
    """

# Create the driver session
with driver.session() as session:
    # Run query
    results = session.run(list_model_catalog_query).data()

    # Prettify the results
    print(json.dumps(results, indent=2, sort_keys=True, default=default))
```

Share a model with other users

After a model has been created, it can be useful to make it available to other users for different use cases.



A model can only be shared with other users of the same AuraDS instance.

Create a new user

In order to see how this works in practice on AuraDS, we first of all need to [create another user](#) to share the model with.

```
# Switch to the "system" database to run the
# "CREATE USER" admin command
gds.set_database("system")

gds.run_cypher("""
    CREATE USER testUser IF NOT EXISTS
    SET PASSWORD 'password'
    SET PASSWORD CHANGE NOT REQUIRED
    """)
```

```
:connect system

CREATE USER testUser IF NOT EXISTS
SET PASSWORD 'password'
SET PASSWORD CHANGE NOT REQUIRED
```

```
# Cypher query
create_a_new_user_query = """
    CREATE USER testUser IF NOT EXISTS
    SET PASSWORD 'password'
    SET PASSWORD CHANGE NOT REQUIRED
    """

# Create the driver session using the "system" database
with driver.session(database="system") as session:
    # Run query
    result = session.run(create_a_new_user_query).data()

# Prettify the result
print(json.dumps(result, indent=2, sort_keys=True))
```

Publish the model

A model can be *published* (made accessible to other users) using the `gds.alpha.model.publish` procedure. Upon publication, the model name is updated by appending `_public` to its original name.

```

# Switch back to the default "neo4j" database
# to publish the model
gds.set_database("neo4j")

model_public = gds.alpha.model.publish(model)

print(model_public)

```

```

:connect neo4j

CALL gds.alpha.model.publish('example_graph_model_for_graphsage')

```

```

# Cypher query
publish_graph_sage_model_to_disk_query = """
    CALL gds.alpha.model.publish('example_graph_model_for_graphsage')
    """

# Create the driver session
with driver.session() as session:
    # Run query
    result = session.run(publish_graph_sage_model_to_disk_query).data()

    # Prettify the result
    print(json.dumps(result, indent=2, sort_keys=True, default=default))

```

View the model as a different user

In order to verify that the published model is visible to the user we have just created, we need to create a new client (or driver) session. We can then use it to run the `gds.beta.model.list` procedure again under the new user and verify that the model is included in the list.

```

test_user_gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=("testUser", "password"),
    aura_ds=True
)

results = test_user_gds.beta.model.list()

print(results)

```

```

// First, open a new Cypher shell with the following command:
//
// ./cypher-shell -a $AURA_CONNECTION_URI -u testUser -p password

CALL gds.beta.model.list()

```

```

test_user_driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=("testUser", "password")
)

# Create the driver session
with test_user_driver.session() as session:
    # Run query
    results = session.run(list_model_catalog_query).data()

    # Prettyfy the results
    print(json.dumps(results, indent=2, sort_keys=True, default=default))

```

Cleanup

The in-memory graphs, the data in the Neo4j database, the models, and the test user can now all be deleted.

```

# Delete the example dataset
gds.run_cypher("""
    MATCH (example:ExampleData)
    DETACH DELETE example
""")

# Delete the projected graph from memory
gds.graph.drop(g)

# Drop the model from memory
gds.beta.model.drop(model_public)

# Delete the model from disk
gds.alpha.model.delete(model_public)

# Switch to the "system" database to delete the example user
gds.set_database("system")

gds.run_cypher("""
    DROP USER testUser
""")

```

```

// Delete the example dataset from the database
MATCH (example:ExampleData)
DETACH DELETE example;

// Delete the projected graph from memory
CALL gds.graph.drop("example_graph_for_graphsage");

// Drop the model from memory
CALL gds.beta.model.drop("example_graph_model_for_graphsage_public");

// Delete the model from disk
CALL gds.alpha.model.delete("example_graph_model_for_graphsage_public");

// Delete the example user
DROP USER testUser;

```

```

# Delete the example dataset from the database
delete_example_graph_query = """
    MATCH (example:ExampleData)
    DETACH DELETE example
    """

# Delete the projected graph from memory
drop_in_memory_graph_query = """
    CALL gds.graph.drop("example_graph_for_graphsage")
    """

# Drop the model from memory
drop_example_models_query = """
    CALL gds.beta.model.drop("example_graph_model_for_graphsage_public")
    """

# Delete the model from disk
delete_example_models_query = """
    CALL gds.alpha.model.delete("example_graph_model_for_graphsage_public")
    """

# Delete the example user
drop_example_user_query = """
    DROP USER testUser
    """

# Create the driver session
with driver.session() as session:
    # Run queries
    print(session.run(delete_example_graph_query).data())
    print(session.run(drop_in_memory_graph_query).data())
    print(session.run(drop_example_models_query).data())
    print(session.run(delete_example_models_query).data())

# Create another driver session on the system database
# to drop the test user
with driver.session(database='system') as session:
    print(session.run(drop_example_user_query).data())

driver.close()
test_user_driver.close()

```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```

# Close the client connection
gds.close()

```

```

# Close the driver connection
driver.close()

```

References

Documentation

- [Neo4j GDS documentation](#)
- [Neo4j driver documentation](#)
- [Neo4j developer documentation](#)

Cypher

- Learn more about the [Cypher](#) syntax
- You can use the [Cypher Cheat Sheet](#) as a reference of all available Cypher features

Modelling

- [Graph modeling guidelines](#)
- [Modeling designs](#)
- [Graph model refactoring](#)

Loading and streaming back data with Apache Arrow



Follow along with a notebook in [Google Colab](#)

The Enterprise Edition of GDS installed on AuraDS includes an [Arrow Flight server](#), configured and running by default. The Arrow Flight server speeds up data-intensive processes such as:

- Creating a graph directly from in-memory data.
- Streaming node and relationship properties.
- Streaming the relationship topology of a graph.

There are two ways to use the Arrow Flight server with GDS:

1. By using the GDS Python client, which includes an Arrow Flight client.
2. By implementing a custom Arrow Flight client as explained in the [GDS manual](#).

In the following examples we use the GDS client as it is the most convenient option. All the loading and streaming methods can be used without Arrow, but are more efficient if Arrow is available.

Setup


```
%pip install 'graphdatascience>=1.7'

from graphdatascience import GraphDataScience

# Replace with the actual connection URI and credentials
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# When initialized, the client tries to use Arrow if it is available on the server.
# This behaviour is controlled by the `arrow` parameter, which is set to `True` by default.
gds = GraphDataScience(AURA_CONNECTION_URI, auth=(AURA_USERNAME, AURA_PASSWORD), aura_ds=True)

# Necessary if Arrow is enabled (as is by default on Aura)
gds.set_database("neo4j")
```

You can call the `gds.debug.arrow()` method to verify that Arrow is enabled and running:

```
gds.debug.arrow()
```

Loading data

You can load data directly into a graph using the `gds.graph.construct` client method.

The data must be a Pandas `DataFrame`, so we need to install and import the `pandas` library.

```
%pip install pandas

import pandas as pd
```

We can then create a graph as in the following example. The format of each `DataFrame` with the required columns is specified in the [GDS manual](#).

```
nodes = pd.DataFrame(
    {
        "nodeId": [0, 1, 2],
        "labels": ["Article", "Article", "Article"],
        "pages": [3, 7, 12],
    }
)

relationships = pd.DataFrame(
    {
        "sourceNodeId": [0, 1],
        "targetNodeId": [1, 2],
        "relationshipType": ["CITES", "CITES"],
        "times": [2, 1]
    }
)

article_graph = gds.graph.construct(
    "article-graph",
    nodes,
    relationships
)
```

Now we can check that the graph has been created:

```
gds.graph.list()
```

Streaming node and relationship properties

After creating the graph, you can read the node and relationship properties [as streams](#).

```
# Read all the values for the node property `pages`
gds.graph.nodeProperties.stream(article_graph, "pages")
```

```
# Read all the values for the relationship property `times`
gds.graph.relationshipProperties.stream(article_graph, "times")
```

Performance

To see the difference in performance when Arrow is available, we can measure the time needed to load a dataset into a graph. In this example we use a built-in [OGBN dataset](#), so we need to install the `ogb` extra.

```
%pip install 'graphdatascience[ogb]>=1.7'

# Load and immediately drop the dataset to download and cache the data
ogbn_arxiv = gds.graph.ogbn.load("ogbn-arxiv")
ogbn_arxiv.drop()
```

We can then time the loading process. On an 8 GB AuraDS instance, this should take less than 30 s.

```
%timeit -n 1 -r 1

# This call uses the cached dataset, so only the actual loading is timed
ogbn_arxiv = gds.graph.ogbn.load("ogbn-arxiv")
```

With Arrow disabled by adding `arrow=False` to the `GraphDataScience` constructor, the same loading process would take more than 1 minute. Therefore, with this dataset, Arrow provides at least a 2x speedup.

Cleanup

```
article_graph.drop()
ogbn_arxiv.drop()

gds.close()
```

Importing data

There are several ways to import data into AuraDS:

- Importing an [existing Neo4j database](#)
- Using the [Data Importer](#)
- Using Cypher's [LOAD CSV](#) procedure
- Using the [Arrow Flight server](#)

Importing an existing database



Note: The process of importing or loading data requires you to [create an AuraDS instance](#) beforehand.

There are two ways you can import data from an existing Neo4j database into an Aura instance.

You can use the [import database](#) process to import either a `.backup` file or a `.dump` file. This process, however, only works for `.backup` and `.dump` files under 4GB.

If the size of the `.backup` or `.dump` file exported from a database is greater than 4GB, you must use the [Neo4j Admin database upload](#) method.

For more information about backups, see [Backup, export and restore](#).

Import database

To import a `.backup` file under 4GB:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to import the data.
3. Select the [Import Database](#) tab.
4. Drag and drop your `.backup` or `.dump` file into the provided window or browse for your `.backup/.dump` file.
5. Select [Upload](#).

When the upload is complete, the instance goes into a [Loading](#) state as the backup is applied. Once this has finished, the instance returns to its [Running](#) state; and the data is ready.

Neo4j Admin [database upload](#)



This command does not work if you have a network access configuration setup that prevents public traffic to the region your instance is hosted in. See [Public traffic](#) below for more information.

`database upload` is a `neo4j-admin` command that you can run to upload the contents of a Neo4j database into an Aura instance, regardless of the database's size. Keep in mind that the database you want to upload may run a different version of Neo4j than your Aura instance. Additionally, your Neo4j Aura instance must be accessible from the machine running `neo4j-admin`. Otherwise, the upload will fail with SSL errors.

For details on how to use the `neo4j-admin database upload` command, along with a full list of options and version compatibility, see [Operations Manual → Upload to Neo4j Aura](#).



The `database upload` command, introduced in Neo4j 5, replaces the `push-to-cloud` command in Neo4j 4.4 and 4.3. If the database you want to upload is running an earlier version of Neo4j, please see the [Neo4j Admin push-to-cloud documentation](#).



The `neo4j-admin push-to-cloud` command in Neo4j 4.4 and earlier is not compatible with instances encrypted with [Customer Managed Keys](#). Use `neo4j-admin database upload` in Neo4j 5 to upload data to instances encrypted with Customer Managed Keys.

For Neo4j 4.x instances in Azure encrypted with Customer Managed Keys, use Neo4j Data Importer to load data, as `neo4j-admin database upload` is not supported. See the [Data Importer documentation](#) for more information.

Public traffic

If you have created a network access configuration from the **Network Access** page, accessed through the sidebar menu of the Console, **Public traffic** must be enabled for the region your instance is hosted in before you can use the `database upload` command on that instance.

To enable **Public traffic** on a network access configuration:

1. Select **Configure** next to the region that has **Public traffic** disabled.
2. Select **Next** until you reach step 4 of 4 in the resulting **Edit network access configuration** modal.
3. Clear the **Disable public traffic** checkbox and select **Save**.

You can now use the `database upload` command on the instances within that region. Once the command has completed, you can disable **Public traffic** again by following the same steps and re-selecting the **Disable public traffic** checkbox.

Using Neo4j Data Importer



Note: The process of importing or loading data requires you to [create an AuraDS instance](#) beforehand.

Neo4j Data Importer is a UI-based tool for importing data that lets you:

1. Load data from flat files (`.csv` and `.tsv`).
2. Define a graph model and map data to it.

3. Import the data into an AuraDS instance.

To load data with Neo4j Data Importer:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the **Import** button on the instance you want to open.

Alternatively, you can access Data Importer from the **Import** tab of [Neo4j Workspace](#).

Once you have opened Neo4j Data Importer, you can follow the built-in tutorial to learn how to use the tool.

For more information on Neo4j Data Importer, see the [Neo4j Data Importer documentation](#).

Loading CSV files



Follow along with a notebook in [Google Colab](#)

A CSV file can be loaded into an AuraDS instance using the **LOAD CSV** Cypher clause. For security reasons it is not possible to load local CSV files, which must be instead publicly accessible on HTTP or HTTPS servers such as GitHub, Google Drive, and Dropbox. Another way to make CSV files available is to upload them to a cloud bucket storage (such as Google Cloud Storage or Amazon S3) and configure the bucket as a static website.

In this example we will load three CSV files:

- **movies.csv**: a list of movies with their title, release year and a short description
- **people.csv**: a list of actors with their year of birth
- **actors.csv**: a list of acting roles, where actors are matched with the movies they had a role in



The **LOAD CSV** command is built to handle small to medium-sized data sets, such as anything up to 10 million nodes and relationships. You should avoid using this command for any data sets exceeding this limit.

Setup

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install graphdatascience
```

```
# Import the client
from graphdatascience import GraphDataScience

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Configure the client with AuraDS-recommended settings
gds = GraphDataScience(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD),
    aura_ds=True
)
```

In the following code examples we use the `print` function to print Pandas `DataFrame` and `Series` objects. You can try different ways to print a Pandas object, for instance via the `to_string` and `to_json` methods; if you use a JSON representation, in some cases you may need to include a `default handler` to handle Neo4j `DateTime` objects. Check the [Python connection](#) section for some examples.

For more information on how to get started using the Cypher Shell, refer to the [Neo4j Cypher Shell](#) tutorial.



Run the following commands from the directory where the Cypher shell is installed.

```
export AURA_CONNECTION_URI="neo4j+s://xxxxxxx.databases.neo4j.io"
export AURA_USERNAME="neo4j"
export AURA_PASSWORD=""

./cypher-shell -a $AURA_CONNECTION_URI -u $AURA_USERNAME -p $AURA_PASSWORD
```

For more information on how to get started using Python, refer to the [Connecting with Python](#) tutorial.

```
pip install neo4j
```

```
# Import the driver
from neo4j import GraphDatabase

# Replace with the actual URI, username, and password
AURA_CONNECTION_URI = "neo4j+s://xxxxxxx.databases.neo4j.io"
AURA_USERNAME = "neo4j"
AURA_PASSWORD = ""

# Instantiate the driver
driver = GraphDatabase.driver(
    AURA_CONNECTION_URI,
    auth=(AURA_USERNAME, AURA_PASSWORD)
)
```

```
# Import to prettify results
import json

# Import for the JSON helper function
from neo4j.time import DateTime

# Helper function for serializing Neo4j DateTime in JSON dumps
def default(o):
    if isinstance(o, (DateTime)):
        return o.isoformat()
```

Create constraints

Adding constraints before loading any data usually improves data loading performance. In fact, besides adding an integrity check, a unique constraint adds an index on a property at the same time, so that **MATCH** and **MERGE** operations during loading are faster.



For best performance when using **MERGE** or **MATCH** with **LOAD CSV**, make sure an index or a unique constraint has been created on the property used for merging. Read the [Cypher documentation](#) for more information on constraints.

In this example we add uniqueness constraints on both movie titles and actors' names.

```

# Make movie titles unique
gds.run_cypher("""
    CREATE CONSTRAINT FOR (movie:Movie) REQUIRE movie.title IS UNIQUE
""")

# Make person names unique
gds.run_cypher("""
    CREATE CONSTRAINT FOR (person:Person) REQUIRE person.name IS UNIQUE
""")

```

```

CREATE CONSTRAINT FOR (movie:Movie) REQUIRE movie.title IS UNIQUE;
CREATE CONSTRAINT FOR (person:Person) REQUIRE person.name IS UNIQUE;

```

```

movie_title_constraint = """
    CREATE CONSTRAINT FOR (movie:Movie) REQUIRE movie.title IS UNIQUE
""

person_name_constraint = """
    CREATE CONSTRAINT FOR (person:Person) REQUIRE person.name IS UNIQUE
""

# Create the driver session
with driver.session() as session:
    # Make movie titles unique
    session.run(movie_title_constraint).data()
    # Make person names unique
    session.run(person_name_constraint).data()

```

Add nodes from CSV files

We are now ready to load the CSV files from their URIs and create nodes from the data they contain. In the following examples, `LOAD CSV` is used with `WITH HEADERS` to access `row` fields by their corresponding column name. Furthermore:

- `MERGE` is used with the indexed properties to take advantage of the constraints created in the [Create constraints](#) section.
- `ON CREATE SET` is used to set the value of a node property when a new one is created.
- `RETURN count(*)` is used to show the number of processed rows.

Note that the CSV files in this example are curated, so some assumptions are made for simplicity. In a real-world scenario, for example, a CSV file could contain multiple rows that would try to assign different property values to the same node; in this case, an `ON MATCH SET` clause must be added to ensure this case is dealt with appropriately.


```

gds.run_cypher("""
LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/movies.csv' AS row
MERGE (m:Movie {title: row.title})
ON CREATE SET m.released = toInteger(row.released), m.tagline = row.tagline
RETURN count(*)
""")

gds.run_cypher("""
LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/people.csv' AS row
MERGE (p:Person {name: row.name})
ON CREATE SET p.born = toInteger(row.born)
RETURN count(*)
""")

```

```

LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/movies.csv' AS row
MERGE (m:Movie {title: row.title})
ON CREATE SET m.released = toInteger(row.released), m.tagline = row.tagline
RETURN count(*);

LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/people.csv' AS row
MERGE (p:Person {name: row.name})
ON CREATE SET p.born = toInteger(row.born)
RETURN count(*);

```

```

load_movies_csv = """
LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/movies.csv' AS row
MERGE (m:Movie {title: row.title})
ON CREATE SET m.released = toInteger(row.released), m.tagline = row.tagline
RETURN count(*)
"""

load_people_csv = """
LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/people.csv' AS row
MERGE (p:Person {name: row.name})
ON CREATE SET p.born = toInteger(row.born)
RETURN count(*)
"""

# Create the driver session
with driver.session() as session:
    # Load the CSV files
    session.run(load_movies_csv).data()
    session.run(load_people_csv).data()

```

Add relationships from CSV files

Similarly to what we have done for nodes, we now create relationships from the `actors.csv` file. In the following example, `LOAD CSV` is used with the `WITH HEADERS` option to access the fields in each `row` by their corresponding column name.



The default field delimiter for `LOAD CSV` is the comma (`,`). Use the `FIELDTERMINATOR` option to set a different delimiter.

If the CSV file is large, use the `CALL IN TRANSACTIONS` clause to commit a number of rows per transaction instead of the whole file.

Furthermore:

- `MATCH` and `MERGE` are used to find nodes (taking advantage of the constraints created in the `Create constraints` section) and create a relationship between them.
- `ON CREATE SET` is used to set the value of a relationship property when a new one is created.
- `RETURN count(*)` is used to show the number of processed rows.

```

gds.run_cypher("""
  LOAD CSV
  WITH HEADERS
  FROM 'https://data.neo4j.com/intro/movies/actors.csv' AS row
  MATCH (p:Person {name: row.person})
  MATCH (m:Movie {title: row.movie})
  MERGE (p)-[actedIn:ACTED_IN]->(m)
  ON CREATE SET actedIn.roles = split(row.roles, ';')
  RETURN count(*)
""")

```

```

LOAD CSV
WITH HEADERS
FROM 'https://data.neo4j.com/intro/movies/actors.csv' AS row
MATCH (p:Person {name: row.person})
MATCH (m:Movie {title: row.movie})
MERGE (p)-[actedIn:ACTED_IN]->(m)
ON CREATE SET actedIn.roles = split(row.roles, ';')
RETURN count(*)

```

```

load_actors_csv = """
  LOAD CSV
  WITH HEADERS
  FROM 'https://data.neo4j.com/intro/movies/actors.csv' AS row
  MATCH (p:Person {name: row.person})
  MATCH (m:Movie {title: row.movie})
  MERGE (p)-[actedIn:ACTED_IN]->(m)
  ON CREATE SET actedIn.roles = split(row.roles, ';')
  RETURN count(*)
"""

# Create the driver session
with driver.session() as session:
  # Load the CSV file
  session.run(load_actors_csv).data()

```

Run a Cypher query

Once all the nodes and relationships have been created, we can run a query to check that the data have been inserted correctly. The following query looks for movies with **Keanu Reeves**, orders them by release date and groups their titles.

```
gds.run_cypher("""
MATCH (person:Person {name: "Keanu Reeves"})-[:ACTED_IN]->(movie)
RETURN movie.released, COLLECT(movie.title) AS movies
ORDER BY movie.released
""")
```

```
MATCH (person:Person {name: "Keanu Reeves"})-[:ACTED_IN]->(movie)
RETURN movie.released, COLLECT(movie.title) AS movies
ORDER BY movie.released
```

```
query = """
MATCH (person:Person {name: "Keanu Reeves"})-[:ACTED_IN]->(movie)
RETURN movie.released, COLLECT(movie.title) AS movies
ORDER BY movie.released
"""

# Create the driver session
with driver.session() as session:
    # Run the Cypher query
    result = session.run(query).data()

    # Print the formatted result
    print(json.dumps(result, indent=2))
```

Cleanup

When the data are no longer useful, the database can be cleaned up.

```
# Delete data
gds.run_cypher("""
    MATCH (n)
    DETACH DELETE n
""")
```

```
MATCH (n)
DETACH DELETE n
```

```
delete_data = """
    MATCH (n)
    DETACH DELETE n
"""

# Create the driver session
with driver.session() as session:
    # Delete the data
    session.run(delete_data).data()
```

Closing the connection

The connection should always be closed when no longer needed.

Although the GDS client automatically closes the connection when the object is deleted, it is good practice to close it explicitly.

```
# Close the client connection
gds.close()
```

```
# Close the driver connection
driver.close()
```

Managing instances

Monitoring

To access the Metrics tab:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the name of the instance you want to access.
3. Select the Metrics tab.

You can monitor the following metrics of an AuraDS instance:

- **CPU Usage (%)** - The amount of CPU used by the instance as a percentage.
- **Storage Used (%)** - The amount of disk storage space used by the instance as a percentage.
- **Heap Usage (%)** - The amount of Java Virtual Machine (JVM) memory used by the instance as a percentage.
- **Out of Memory Errors** - The number of Out of Memory (OOM) errors encountered by the instance.
- **Garbage Collection Time (%)** - The amount of time the instance spends reclaiming heap space as a percentage.



More information on each metric, as well as suggestions for managing them, can be found within the Metrics tab itself.

When viewing metrics, you can select from the following time intervals:

- 6 hours
- 24 hours
- 3 days
- 7 days
- 30 days

Advanced metrics

Advanced metrics is a feature that enables access to a broad range of different instance and database metrics.

To access Advanced metrics:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to access.
3. Select the Metrics tab.

4. Select the **Advanced metrics** button.

The presented metrics will be laid out across three tabs according to their category:

- **Resources** - Overall system resources, such as CPU, RAM and disk usage.
- **Instance** - Information about the Neo4j instances running the database.
- **Database** - Metrics concerning the database itself, such as usage statistics and entity counts.

When viewing metrics, you can select from the following time intervals:

- 30 minutes
- 6 hours
- 24 hours
- 3 days
- 7 days
- 14 days
- 30 days

Chart interactions



Memory and storage charts can be toggled between absolute and relative values using the % toggle.

Zoom

To zoom in to a narrower time interval, select and drag inside any chart to select your desired time interval. The data will automatically update to match the increased resolution.

To reset zoom, double-click anywhere inside the chart or use the option in the context menu.

Expand

Any chart can be expanded to take up all the available screen estate by clicking the **expand** button (shown as two opposing arrows). To exit this mode, click the **x** button on the expanded view.

Context menu

To access the chart context menu, select the ... button on any chart.

- **More info** - Selecting **More info** brings up an explanation of the particular metric. For some metrics it also provides hints about possible actions to take if that metric falls outside the expected range.
- **Reset zoom** - If the zoom level has been altered by selecting and dragging across a chart, **Reset zoom** resets the zoom back to the selected interval.

Backup, export, and restore

The data in your AuraDS instance can be backed up, exported, and restored using snapshots. A snapshot is a copy of the data in an instance at a specific point in time.

The Snapshots tab within an AuraDS instance shows a list of available snapshots. To access the Snapshots tab:

1. Navigate to the [Neo4j Aura Console](#) in your browser.
2. Select the instance you want to access.
3. Select the Snapshots tab.

Snapshot types

There are two different types of snapshot:

- **Scheduled** - Runs when you first create an instance, when changes to the underlying system occur (for example, a new patch release), and automatically once a day.
- **On-demand** - Runs when you select **Take snapshot** from the Snapshots tab of an instance.



Scheduled daily snapshots are kept for 7 days for Professional instances and 14 days for Enterprise instances. On-demand snapshots are kept in the system for 180 days for all instances.

Snapshot actions

Take a snapshot

You can manually trigger an On-demand snapshot by selecting **Take snapshot** in the Snapshots tab. The snapshot status is shown as **In progress** until the snapshot has been created; then, the **Status** becomes **Completed**.

Restore

You can restore data in your instance to a previous snapshot by selecting **Restore** next to the snapshot you want to restore.

Restoring a snapshot requires you to confirm the action by typing **RESTORE** and selecting **Restore**.



Restoring a snapshot overwrites the data in your instance, replacing it with the data contained in the snapshot.

Backup and export

By selecting the ellipses (...) button next to a snapshot, you can:

- **Export** - Download the database as a compressed file, allowing you to store a local copy and work on your data offline. The compressed archive contains a `.dump` file that can be imported directly or pushed to the cloud.
- **Create instance from snapshot** - Create a new AuraDS instance using the data from the snapshot. This opens a window where you can assign a name to the instance that will be created.

Instance actions

You can perform several actions on an AuraDS instance from the [Neo4j Aura Console](#) homepage.

Renaming an instance

You can change the name of an existing instance by using the **Rename** action.

To rename an instance:

1. Select the ellipsis (...) button on the instance you want to rename.
2. Select **Rename** from the resulting menu.
3. Enter a new name for the instance.
4. Select **Rename**.

Resizing an instance

You can change the size of an existing instance by using the **Resize** action.

To resize an instance:

1. Select the ellipsis (...) on the instance you want to resize.
2. Select **Resize** from the resulting menu.
3. Select the new size you want your instance to be.
4. Tick the **I understand** checkbox and select **Submit**.

An instance becomes unavailable for a short period of time during the resize operation.

Pausing an instance

You can pause an instance during periods where you don't need it and resume at any time.

To pause an instance:

1. Select the pause icon on the instance you want to pause.
2. Select **Pause** to confirm.

After confirming, the instance will begin pausing, and a **Resume** button will replace the **Pause** button.



Paused instances run at a discounted rate compared to standard consumption, as outlined in the confirmation window. You can pause an instance for up to 30 days, after which point AuraDS automatically resumes the instance.

Resuming an instance

To resume an instance:

1. Select the play icon on the instance you want to pause.
2. Tick the **I understand** checkbox and select **Resume** to confirm.

After confirming, the instance will begin resuming, which may take a few minutes.

Cloning an instance

You can clone an existing instance to create a new instance with the same data. You can clone across regions, from AuraDB to AuraDS and vice versa, and from Neo4j version 4 to Neo4j version 5.

There are four options to clone an instance:

- Clone to a new AuraDS instance
- Clone to an existing AuraDS instance
- Clone to a new AuraDB database
- Clone to an existing AuraDB database

You can access all the cloning options from the ellipsis (...) button on the AuraDS instance.



You cannot clone from a Neo4j version 5 instance to a Neo4j version 4 instance.

Clone to a new AuraDS instance

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To New** and then **AuraDS** from the contextual menu.
3. Set the desired name for the new instance.
4. Check the **I understand** box and select **Clone Instance**.



Make sure that the username and password are stored safely before continuing. Credentials cannot be recovered afterwards.

Clone to an existing AuraDS instance

When you clone an instance to an existing instance, the database connection URI stays the same, but the data is replaced with the data from the cloned instance.



Cloning into an existing instance will replace all existing data. If you want to keep the current data, take a snapshot and export it.

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To Existing** and then **AuraDS** from the contextual menu.
3. If necessary, change the instance name.
4. Select the existing AuraDS instance to clone to from the dropdown menu.



Existing instances that are not large enough to clone into will not be available for selection. In the dropdown menu, they are grayed out and have the string **(Instance is not large enough to clone into)** appended to their name.

5. Tick the **I understand** checkbox and select **Clone**.

Clone to a new AuraDB instance



An AuraDS instance can only be cloned to an AuraDB Professional database (not Free).

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To New** and then **AuraDB** from the contextual menu.
3. Set your desired settings for the new database. For more information on AuraDB database creation, see [Creating an instance](#).
4. Check the **I understand** box and select **Clone Database**.



Make sure that the username and password are stored safely before continuing. Credentials cannot be recovered afterwards.

Clone to an existing AuraDB instance



An AuraDS instance can only be cloned to an AuraDB Professional database (not Free).



Cloning into an existing instance will replace all existing data. If you want to keep the current data, take a snapshot and export it.

1. Select the ellipsis (...) button on the instance you want to clone.
2. Select **Clone To Existing** and then **AuraDB** from the contextual menu.
3. If necessary, change the database name.
4. Select the existing AuraDB database to clone to from the dropdown menu.



Existing instances that are not large enough to clone into will not be available for selection. In the dropdown menu, they will be grayed out and have the string **(Instance is not large enough to clone into)** appended to their name.

5. Check the I understand box and select Clone.

Deleting an instance

You can delete an instance if you no longer want to be billed for it.



There is no way to recover data from a deleted AuraDS instance.

To delete an instance:

- Select the red trashcan icon on the instance you want to delete.
- Type the exact name of the instance (as instructed) to confirm your decision, and select **Destroy**.

=Tutorials=

Upgrade and migration

Upgrade to Neo4j 5 within Aura

This tutorial describes how to upgrade an Aura instance running Neo4j version 4 to Neo4j version 5.



New AuraDS and AuraDB Free instances use Neo4j 5 as standard, while all others give the option to choose between Neo4j 4 and 5 during creation.

Prepare for the upgrade

Drivers

Neo4j's official drivers have some significant and breaking changes between versions you need to be aware of. For a smooth migration:

1. Check the breaking changes for each driver you use, for example in the [Python driver](#) and in the [GDS client](#).
2. Make sure you switch to the latest version of the driver in line with the version of the Neo4j database. This can be done before upgrading the version of Neo4j that you are using with Aura, as 5.x drivers are backward compatible.

The [Update and migration guide](#) contains all information and lists all the breaking changes.

Indexes

In Neo4j 5, BTREE indexes are replaced by RANGE, POINT, and TEXT indexes. Before migrating a database, in Neo4j 4, you should create a matching RANGE, POINT, or TEXT index for each BTREE index (or index-backed constraint). You can run `SHOW INDEXES` on your Neo4j 4 database to display its indexes.

In most cases, RANGE indexes can replace BTREE. However, there might be occasions when a different index type is more suitable, such as:

- Use POINT indexes if the property value type is `point` and `distance` or `bounding box` queries are used for the property.
- Use TEXT indexes if the property value type is `text` and the values can be larger than 8Kb.
- Use TEXT indexes if the property value type is `text` and `CONTAINS` and `ENDS WITH` are used in queries for the property.

After creating the new index, the old index should be dropped. The following example shows how to create a new RANGE index and drop an existing `index_name` index:

```
CREATE RANGE INDEX range_index_name FOR (n:Label) ON (n.prop1);
DROP INDEX index_name;
```

The following example instead shows how to create a constraint backed by a RANGE index:

```
CREATE CONSTRAINT constraint_with_provider FOR (n:Label) REQUIRE (n.prop1) IS UNIQUE OPTIONS
{indexProvider: 'range-1.0'}
```

For more information about creating indexes, see [Cypher Manual → Creating indexes](#).

Cypher updates

Neo4j 5 introduces some changes to the Cypher syntax and error handling.

Cypher syntax

All changes in the Cypher language syntax are detailed in [Cypher Manual → Removals, deprecations, additions and extensions](#). Thoroughly review this section in the version you are moving to and make the necessary changes in your code.

Here is a short list of the main changes introduced in Neo4j 5:

Deprecated feature	Details
<pre>MATCH (n)-[r:REL]->(m) SET n=r</pre>	<p>Use the <code>properties()</code> function instead to get the map of properties of nodes/relationships that can then be used in a <code>SET</code> clause:</p> <pre>MATCH (n)-[r:REL]->(m) SET n=properties(r)</pre>
<pre>MATCH (a), (b), allShortestPaths((a)-[r]->(b)) RETURN b MATCH (a), (b), shortestPath((a)-[r]->(b)) RETURN b</pre>	<p><code>shortestPath</code> and <code>allShortestPaths</code> without <code>variable-length relationship</code> are deprecated. Instead, use a <code>MATCH</code> with a <code>LIMIT</code> of 1 or:</p> <pre>MATCH (a), (b), shortestPath((a)-[r*1..1]->(b)) RETURN b</pre>
<pre>CREATE DATABASE databaseName.withDot ...</pre>	<p>Creating a database with unescaped dots in the name has been deprecated, instead escape the database name:</p> <pre>CREATE DATABASE `databaseName.withDot` ...</pre>

Error handling in Cypher

Many semantic errors that Cypher finds are reported as `Neo.ClientError.Statement.SyntaxError` even though they are semantic and not syntax errors. In Neo4j 5, the metadata returned by Cypher queries is

improved.

- The severity of some of the Warning codes is moved to Info:
 - `SubqueryVariableShadowingWarning` → `SubqueryVariableShadowing`
 - `NoApplicableIndexWarning` → `NoApplicableIndex`
 - `CartesianProductWarning` → `CartesianProduct`
 - `DynamicPropertyWarning` → `DynamicProperty`
 - `EagerOperatorWarning` → `EagerOperator`
 - `ExhustiveShortestPathWarning` → `ExhaustiveShortestPath`
 - `UnboundedVariableLengthPatternWarning` → `UnboundedVariableLengthPattern`
 - `ExperimentalFeature` → `RuntimeExperimental`

APOC

All APOC procedures and functions available in Aura are listed in the [APOC Core library](#). See the [APOC documentation](#) for further details.

Procedures

Some procedures have been replaced by commands:

Procedure	Replacement
<code>db.indexes</code>	<code>SHOW INDEXES</code> command
<code>db.indexDetails</code>	<code>SHOW INDEXES YIELD *</code> command
<code>db.schemaStatements</code>	<code>SHOW INDEXES YIELD *</code> command and <code>SHOW CONSTRAINTS YIELD *</code> command
<code>db.constraints</code>	<code>SHOW CONSTRAINTS</code> command
<code>db.createIndex</code>	<code>CREATE INDEX</code> command
<code>db.createUniquePropertyConstraint</code>	<code>CREATE CONSTRAINT ... IS UNIQUE</code> command
<code>db.index.fulltext.createNodeIndex</code>	<code>CREATE FULLTEXT INDEX</code> command
<code>db.index.fulltext.createRelationshipIndex</code>	<code>CREATE FULLTEXT INDEX</code> command
<code>db.index.fulltext.drop</code>	<code>DROP INDEX</code> command
<code>dbms.procedures</code>	<code>SHOW PROCEDURES</code> command
<code>dbms.functions</code>	<code>SHOW FUNCTIONS</code> command
<code>dbms.listTransactions</code>	<code>SHOW TRANSACTIONS</code> command
<code>dbms.killTransaction</code>	<code>TERMINATE TRANSACTIONS</code> command
<code>dbms.killTransactions</code>	<code>TERMINATE TRANSACTIONS</code> command
<code>dbms.listQueries</code>	<code>SHOW TRANSACTIONS</code> command

Procedure	Replacement
<code>dbms.killQuery</code>	<code>TERMINATE TRANSACTIONS</code> command
<code>dbms.killQueries</code>	<code>TERMINATE TRANSACTIONS</code> command
<code>dbms.scheduler.profile</code>	-

Refer to the [Update and migration guide](#) for a full list of removals and deprecations.

Neo4j Connectors

If you are using a Neo4j Connector for [Apache Spark](#) or [Apache Kafka](#), make sure its version is compatible with Neo4j 5.

The Neo4j BI Connectors available on the [Download center](#) are compatible with Neo4j 5.

Perform the upgrade

Once you have prepared your Neo4j 4 Aura instance, you are ready to migrate the instance to a new or existing Neo4j 5 instance.

Clone

If you have an existing Neo4j 5 instance, you can use the [Clone To Existing](#) instance action on your Neo4j 4 [AuraDB](#) or [AuraDS](#) instance.

If you do not have an existing Neo4j 5 instance, you can use the [Clone To New](#) instance action on your Neo4j 4 [AuraDB](#) or [AuraDS](#) instance.

Export and import

Alternatively, you can [Export](#) a snapshot dump file from your Neo4j 4 [AuraDB](#) or [AuraDS](#) instance, create a new Neo4j 5 instance manually, and then import the dump file into your new Neo4j 5 [AuraDB](#) or [AuraDS](#) instance.

Migrate from self-managed Neo4j to Aura

This tutorial describes how to migrate from a self-managed Neo4j database to Aura.



If your local Neo4j version is older than 4.3, you need to upgrade to at least Neo4j 4.3 first as explained in [Upgrade and Migration Guide → Neo4j 4 upgrades and migration](#).

Preparation

Migrating to Neo4j 5

If you are migrating from self-managed Neo4j 4.3 or 4.4 to Neo4j 5 on Aura, carefully read the [Preparation section in the Upgrade tutorial](#) to ensure you are well prepared for the migration.

Aura instance size

Before starting, verify that the Aura instance you are migrating to is sized accordingly. The instance must be at least as large as your self-managed database to accommodate the data. The Aura RAM-to-storage ratio is 1:2, which means, for example, that a 32 GB Aura instance provides 64 GB of storage.

APOC compatibility

If you are using any APOC procedures and functions, make sure they are all available in Aura by checking the [APOC support page](#).

Creating and uploading a database dump

In order to move data from your self-managed database to Aura, you need to create a dump of the existing database.



This process requires a short downtime for your self-managed database.

The following admin commands must be invoked with the same user as your self-managed Neo4j database. This guarantees that Neo4j has full rights to start and work with the database files you use.

1. Stop your self-managed Neo4j database. If you are running AuraDB Virtual Dedicated Cloud or AuraDS Enterprise, you can stop only the database you want to dump using the command `STOP DATABASE neo4j` in Cypher Shell or Browser.
2. Ensure the target directory to store the database dumps (for instance `/dumps/neo4j`) exists.
3. Depending on your self-managed Neo4j version, create a dump of your database (e.g., `neo4j`) using one of the following options:

Use the `neo4j-admin dump` command.

```
bin/neo4j-admin dump --database=neo4j --to=/dumps/neo4j
```

Use the `neo4j-admin database dump` command.

```
bin/neo4j-admin database dump neo4j --to-path=/dumps/neo4j
```

4. Depending on your self-managed Neo4j version, upload the database dump (e.g., `neo4j`) to your Aura instance using one of the following options:

Use the `neo4j-admin push-to-cloud` command.

```
bin/neo4j-admin push-to-cloud --dump=/dumps/neo4j/file.dump --bolt-uri  
=neo4j+s://xxxxxxx.databases.neo4j.io --overwrite
```

Use the `neo4j-admin database upload` command.

```
bin/neo4j-admin database upload neo4j --from-path=/dumps/neo4j --to-uri  
=neo4j+s://xxxxxxx.databases.neo4j.io --overwrite-destination=true
```

= Migrating your Neo4j AuraDB Free instance to another AuraDB plan :description: This section describes migrating your Neo4j AuraDB Free Instance to another AuraDB plan

AuraDB Professional or AuraDB Virtual Dedicated Cloud

Upgrading your plan to AuraDB Professional or AuraDB Virtual Dedicated Cloud gives you access to additional resources and functionalities to support production workloads and applications with demanding storage and processing needs.

Migration options

- Upgrade to AuraDB Professional
- Clone to new (Works for AuraDB Professional and AuraDS Professional)
- Manual process

Upgrade to AuraDB Professional

You can upgrade an instance to the Professional plan directly from the console.

Click the ellipsis (...) button on an instance card > **Upgrade to Professional**

Verify that the cloud provider and region are correct and select the instance size you need. Note that the default version of Neo4j is 5. Once you are satisfied, click **Upgrade**.

Clone (Works for AuraDB Professional and AuraDS)

The other way is to clone your existing instance to the Professional plan.

- Click the ellipsis (...) button on an instance
- Select either: **Clone to new** or **Clone to existing** (the current content will be overwritten)
- Select the type: **AuraDB** or **AuraDS**

Manual process

In your existing instance:

1. (Optional but recommended) Capture existing index and constraint definitions:

a. Run the following Cypher statement:

```
SHOW CONSTRAINTS YIELD createStatement
```

Save result to a file, to use later in the process.

b. Run the following Cypher statement:

```
SHOW INDEXES YIELD createStatement
```

Save result to a file, to use later in the process.

2. (Optional but recommended) Drop the indexes and constraints.

a. Run the following Cypher statement to generate the commands to drop existing constraints:

```
SHOW CONSTRAINTS YIELD name  
RETURN 'DROP CONSTRAINT ' + name + ';' ;'
```

b. Execute the generated commands to drop existing constraints.

c. Run the following Cypher statement to generate the commands to drop existing indexes:

```
SHOW INDEX YIELD name  
RETURN 'DROP INDEX ' + name + ';' ;'
```

d. Execute the generated commands to drop existing indexes.

For more information about indexes and constraints, see [Cypher Manual → Constraints](#).

3. In the console of your existing instance (AuraDB Free), do the following:

- a. Download snapshot/Dump locally (the daily automatic snapshot)
- b. In the Aura Console select the AuraDB instance
- c. Go to the Snapshots tab
- d. Click the **three dots**, and select **Export**
- e. Save the dump file locally (preserve the .dump extension)

4. Then create a new AuraDB instance in AuraDB Professional or AuraDB Virtual Dedicated Cloud with the right resource sizing. From your new instance, do the following:

- a. Upload via Console drag and drop or push-to-cloud
 - i. From the Aura Console: drag and drop the .dump file
 - ii. Using the command line: `neo4j-admin push-to-cloud`

5. In the newly created AuraDB Professional or AuraDB Virtual Dedicated Cloud instance

(Optional) Once the AuraDB instance is loaded and started, you can recreate the indexes and constraints, using the information captured earlier in the process.

Integrating with Neo4j Connectors

Using the Neo4j Connector for Apache Spark

This tutorial shows how to use the Neo4j Connector for Apache Spark to write to and read data from an Aura instance.

Setup

1. Download [Apache Spark](#).

Example: Spark 3.4.1, pre-built for Apache Hadoop 3.3 and later with Scala 2.12.

2. Download the [Neo4j Connector JAR file](#), making sure to match both the Spark version and the Scala version.

Example: Neo4j Connector 5.1.0, built for Spark 3.x with Scala 2.12.

3. Decompress the Spark file and launch the Spark shell as in the following example:

```
$ spark-3.4.1-bin-hadoop3/bin/spark-shell --jars neo4j-connector-apache-spark_2.12-5.1.0_for_spark_3.jar
```

Running code in Apache Spark



You can copy-paste Scala code in the Spark shell by activating the `paste` mode with the `:paste` command.

Create a Spark session and set the Aura connection credentials:

```
import org.apache.spark.sql.{SaveMode, SparkSession}

val spark = SparkSession.builder().getOrCreate()

// Replace with the actual connection URI and credentials
val url = "neo4j+s://xxxxxxx.databases.neo4j.io"
val username = "neo4j"
val password = ""
```

Then, create the `Person` class and a Spark `Dataset` with some example data:

```

case class Person(name: String, surname: String, age: Int)

// Create example Dataset
val ds = Seq(
  Person("John", "Doe", 42),
  Person("Jane", "Doe", 40)
).toDS()

```

Write the data to Aura:

```

// Write to Neo4j
ds.write.format("org.neo4j.spark.DataSource")
  .mode(SaveMode.Overwrite)
  .option("url", url)
  .option("authentication.basic.username", username)
  .option("authentication.basic.password", password)
  .option("labels", ":Person")
  .option("node.keys", "name,surname")
  .save()

```

You can then query and visualize the data using the Neo4j Browser.

You can also read the data back from Aura:

```

// Read from Neo4j
val data = spark.read.format("org.neo4j.spark.DataSource")
  .option("url", url)
  .option("authentication.basic.username", username)
  .option("authentication.basic.password", password)
  .option("labels", "Person")
  .load()

// Visualize the data as a table
data.show()

```

For further information on how to use the connector, read the [Neo4j Spark Connector docs](#).

Using the Neo4j BI Connector

In this tutorial we use the Neo4j Connector for BI to read graph data from an Aura instance using some common [SQL clients](#) and [BI tools](#).



This tutorial includes instructions on the usage of third-party software, which may be subject to changes beyond our control. In case of doubt, please refer to the third-party software documentation.

Downloading the connector

Download the connector from the [Download Center](#). Depending on the SQL client or BI tool it will be used with, you will need either the JDBC or the ODBC connector; see the usage examples for further details.

Preparing example data

Before trying the connector with any of the listed tools, some data needs to be loaded on Aura. This can be achieved by running the following Cypher query in the Neo4j Browser:

CREATE

```
(john:Person {name: "John", surname: "Doe", age: 42}),  
(jane:Person {name: "Jane", surname: "Doe", age: 40}),  
(john)-[:KNOWS]->(jane)
```

Using BI tools

Commonly used BI tools include [Tableau](#) (which uses the JDBC driver) and [Power BI](#) (which uses the ODBC driver).



When connecting with a JDBC driver, the `neo4j+s` URI scheme must be changed into `neo4j` and the `SSL=true` parameter must be added to the URL.

Tableau



This example requires [Tableau Desktop](#).

Refer to the [Tableau documentation](#) for more information on how to add a JDBC database.

After downloading the JDBC Neo4j Connector for BI from the [Download Center](#):

1. Close any running instances of Tableau Desktop.
2. Copy the Neo4j driver to the appropriate Tableau drivers folder (for example `C:\Program Files\Tableau\Drivers` on Windows, or `~/Library/Tableau/Drivers` on macOS).
3. Start Tableau and search for the `Other Databases (JDBC)` option.
4. Insert the Aura URL as `jdbc:neo4j://xxxxxxx.databases.neo4j.io?SSL=true`, leave the SQL dialect as `SQL92`, and complete the relevant credentials.

If the connection fails with a `Generic JDBC connection error`, you can do one of the following:

- Download the `SSL.com` CA root certificate from [ssl.com](#) and install it as explained in the [Tableau documentation](#), then restart Tableau and repeat the previous steps (recommended option).
- Add `&sslTrustStrategy=TRUST_ALL_CERTIFICATES` to the connection string (after `SSL=true`) and try to connect again. This option requires caution and should not be used in a production environment.

After the connection is established, you can select the `neo4j` database and the `Node` schema to find the `Person` table. You can then explore the table to find the example data.

Power BI



This example requires Microsoft Windows and [Power BI Desktop](#).

Refer to the [Power BI documentation](#) for more information on how to add an ODBC database.

After downloading and installing the ODBC Neo4j Connector for BI from the [Download Center](#):

1. Open Power BI Desktop.
2. Search for **ODBC** in the **Get data from another source** panel.
3. Select **Simba Neo4j** in the **DSN dropdown** menu.
4. Insert the connection string **Host=xxxxxxxx.databases.neo4j.io;SSL=1** in the **Advanced options** section.
5. Insert your username and password.

Once connected, open sequentially **ODBC** → **neo4j** → **Node** → **Person** in the **Navigator** window to see a preview of the table.

Using command-line SQL clients

In order to run SQL queries, we need a SQL client that can use a custom driver. Common JDBC-based command-line SQL clients include **sqlline** and **jdbcsql**.



When connecting with a JDBC driver, the **neo4j+s** URI scheme must be changed into **neo4j** and the **SSL=true** parameter must be added to the URL.

sqlline

sqlline is a command-line tool for issuing SQL queries to relational databases via JDBC. To clone and build it, run the following:

```
$ git clone https://github.com/julianhyde/sqlline
$ cd sqlline
$ ./mvnw package
```

We now need to make the BI connector driver available to **sqlite**. This can be done by extracting the **Neo4jJDBC42.jar** file from the downloaded **JDBC BI connector** into the **sqlline/target** folder.

The **sqlline** client can now be run as follows:

```
$ ./bin/sqlline -d com.simba.neo4j.neo4j.jdbc42.Driver
```

From the client prompt, it is possible to connect to the Aura instance by supplying the username and password when prompted to do so:

```
sqlline> !connect jdbc:neo4j://xxxxxxxx.databases.neo4j.io?SSL=true
```

When the connection is established, a list of tables can be obtained with the **!tables** command:

```
jdbc:neo4j://xxxxxxxx.databases.neo4j.io> !tables
```

```

+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
| TABLE_CAT | TABLE_SCHEM | TABLE_NAME | TABLE_TYPE | REMARKS | TYPE_CAT | TYPE_SCHEM |
TYPE_NAME | SELF_R |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
| neo4j | Node | Person | TABLE | | | | |
|
| neo4j | Relationship | Person_KNOWS_Person | TABLE | | | | |
|
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+

```

It is also possible to run SQL queries:

```

jdbc:neo4j://xxxxxxxx.databases.neo4j.io> SELECT * FROM Person;

```

```

+-----+-----+-----+-----+
| _NodeId_ | age | name | surname |
+-----+-----+-----+-----+
| 0 | 42 | John | Doe |
| 1 | 40 | Jane | Doe |
+-----+-----+-----+-----+

```

jdbcsql

[jdbcsql](#) is a command-line tool that can be used to connect to a DBMS via a JDBC driver.

After downloading the [jdbcsql-1.0.zip](#) file from [SourceForge](#), extract it into the [jdbcsql](#) folder; then, copy the [Neo4jJDBC42.jar](#) file from the downloaded [JDBC BI Connector](#) into [jdbcsql](#) and make the following changes:

1. Add the following lines to [JDBCConfig.properties](#)

```

# neo4j settings
neo4j_driver = com.simba.neo4j.neo4j.jdbc42.Driver
neo4j_url = jdbc:neo4j://host?SSL=true

```

2. Add [Neo4jJDBC42.jar](#) to [Rsrc-Class-Path](#) line in [META-INF/MANIFEST.MF](#)

Now run the following command (replacing [xxxxxxxx.databases.neo4j.io](#) with the Aura connection URI, and [yyyyyyyy](#) with the actual password):

```

$ java org.eclipse.jdt.internal.jarinjarloader.JarRsrcLoader -m neo4j -h xxxxxxxx.databases.neo4j.io -d
neo4j -U neo4j -P yyyyyyy 'SELECT * FROM Person'

```

The result of the query is:

```

"_NodeId_" age name surname
0 42 John Doe
1 40 Jane Doe

```

Improving Cypher performance

This page covers a number of steps you can take to improve the Cypher performance of your workload.

Cypher statements with literal values

One of the main causes of poor query performance is due to running many Cypher statements with literal values. This leads to inefficient Cypher processing as there is currently no use of parameters. As a result, you don't benefit fully from the execution plan cache that would occur otherwise.

The following Cypher queries are identical in form but use different literals:

```
MATCH (tg:asset) WHERE tg.name = "ABC123"  
MERGE (tg)<-[:TAG_OF]-(z1:tag {name: "/DATA01/" + tg.name + "/Top_DOOR"})  
MERGE (tg)<-[:TAG_OF]-(z2:tag {name: "/DATA01/" + tg.name + "/Data_Vault"})
```

In cases like this, query parsing and execution plan generation happen multiple times, resulting in a loss of efficiency. One way to solve that is by rewriting the former example as follows:

```
MATCH (tg:asset) WHERE tg.name = $tgName  
WITH tg  
UNWIND $tags as tag  
MERGE (tg)<-[:TAG_OF]-(tag {name: tag.name})
```

By replacing the literal values in the queries with parameters you get a better execution plan caching reuse. Your application needs to place all the values in a parameter list and then you can issue one statement that iterates through them. Making these changes will lead to improvements in execution and memory usage.

Review queries and model

One first action that you can take is reviewing and listing all your Cypher queries. The best starting point is to have a good understanding of the sequence and frequency of the Cypher queries submitted.

Additionally, if the queries are generated by a framework, it is essential to log them in Cypher form to make reviewing easier.

You can also profile a Cypher query by prepending it with **EXPLAIN** (to see the execution plan without running the query) or **PROFILE** (to run and profile the query). Read more about [profiling a query](#).



When using **PROFILE** you may need to run it multiple times in order to get the optimal value. The first time the query runs, it gets a full cycle of evaluation, planning, and interpreting before making its way into the query cache. Once in the cache, the subsequent execution time will improve. Furthermore, always use parameters instead of literal values to benefit from the cache.

Read more about [how to capture the execution plans](#).

To best interpret the output of your execution plan, it is recommended that you get familiar with the terms used on it. See [this summary of execution plan operators](#) for more information.

Index specification

As your data volume grows, it is important to define constraints and indexes in order to achieve the best performance for your queries. For that, the runtime engine will need to evaluate the cost associated with a query and, to get the best estimations, it will rely on already existing indexes. This will likely show whether an index is missing from the execution plan and which one is it. Though in some circumstances it might look like an index is not available or possible, it may also make sense to reconsider the model and create an intermediate node or another relationship type just to leverage it.

Read more about [the use of indexes](#) for a more comprehensive explanation.



You can also fine-tune the usage of an index in your query by leveraging it with the `USING` clause.

Review metrics and instance size

With Aura, you can keep an eye on some key metrics to see which resource constraints your instance may be experiencing. Follow the steps described in [Monitoring](#) to check that information.

At this stage, if the key metrics are too high, you may want to reconsider the instance sizing. A resize operation does not cause any downtime, and you would only pay for what you use.



You should always size your instance against your workload activity peaks.

Consider concurrency

Sometimes individual queries may be optimized on their own and run fine, but the sheer volume and concurrency of operations can overwhelm your Aura instance.

To review what is running at any given time (this makes particular sense if you have a long-running query), you can use these statements and list what is running:

- `SHOW TRANSACTIONS`
- `CALL dbms.listQueries()`

Runtime engine and Cypher version

The execution plan should show you the runtime that is selected for the execution of your query. Usually, the planner makes the right decision, but it may be worth checking at times if any other runtime performs better. Read more about [query tuning](#) on Cypher runtime.

To invoke the use of a given runtime forcibly, prepend your Cypher statement with:

- `CYPHER runtime=pipelined` for `pipelined` runtime

- CYPHER runtime=slotted for slotted runtime
- CYPHER runtime=interpreted for interpreted runtime

If you have a Cypher pattern that is not performing without error, it could as well be running on a prior Cypher version. You can control the version used to interpret your queries by using these [Cypher query options](#).

Network and the cost of the round-trip

With Aura, it is essential to consider the best cloud in your region as the physical distance is a direct factor in the achievable network latency.

When some event causes any network disruption between your application and Aura, you would be affected by round-trip network latency to re-submit a query. With Aura, this is particularly important because you will need to be using transaction functions when [connecting your instance to applications](#).

Troubleshooting

This page provides possible solutions to several common issues you may encounter when using Neo4j Aura.

Regardless of the issue, viewing the [Aura query log](#) is always recommended to monitor processes and verify any problems.

Query performance

MemoryLimitExceededException

During regular operations of your Aura instance, you may at times see that some of your queries fail with the error:

MemoryLimitExceededException error

```
org.neo4j.memory.MemoryLimitExceededException: The allocation of an extra 8.3 MiB would use more than the limit 278.0 MiB.
Currently using 275.1 MiB. dbms.memory.transaction.global_max_size threshold reached
```

The `org.neo4j.memory.MemoryLimitExceededException` configuration acts as a safeguard, limiting the quantity of memory allocated to all transactions while preserving the regular operations of the Aura instance. Similarly, the property `dbms.memory.transaction.global_max_size` also aims to protect the Aura Instance from experiencing any OOM (Out of memory) exceptions and increase resiliency. It is enabled in Aura and cannot be disabled.

However, the measured heap usage of all transactions is only an estimate and may differ from the actual number. The estimation algorithm relies on a conservative approach, which can lead to overestimation of memory usage. In such cases, all contributing objects' identities are unknown and cannot be assumed to be shared.

Solution



We recommend handling this error in your application code, as it may be intermittent.

Overestimation is most likely to happen when using `UNWIND` on long lists or when expanding a variable length or shortest path pattern. The many relationships shared between the computed result paths could be the cause of a lack of precision in the estimation algorithm.

To avoid this scenario, try running the same query without using a sorting operation like `ORDER BY` or `DISTINCT`. Additionally, if possible, handle this ordering or uniqueness in your application.

If removing the `ORDER BY` or `DISTINCT` clauses does not solve the issue, the primary mitigation for this error is to perform one or more of these actions:

- Handle this exception in your code and be prepared to retry if this is an intermittent error. Keep in mind that the query can succeed regardless.
- Rework the relevant query to optimize it.

Return a list of indexes and their types

```
CALL db.constraints() YIELD description
UNWIND ["DROP", "CREATE"] AS command
RETURN command + " " + description
```

3. In Neo4j Browser, select the "Enable multi statement query editor" option under the browser settings.
4. Take the list of commands from the 2nd step and copy them in one list of multiple queries into Browser and run those queries.
5. After the indexes are recreated, try the `database upload` command again.

InconsistentData

This error message will likely trigger when Neo4j Aura cannot safely load the data provided due to inconsistencies.

Solution

If you encounter this error, please raise a ticket with our [Customer Support](#) team.

UnsupportedStoreFormat

You may get this error if the store you are uploading is in a Neo4j version that is not directly supported in Neo4j Aura.

Solution

1. [Upgrade your database](#). Make sure you are on Neo4j 4.4 or later.
2. If you encounter problems upgrading, please raise a ticket with our [Customer Support](#) team.

LogicalRestrictions

You may get this error when the store you are uploading exceeds the logical limits of your database.

Solution

1. Delete nodes and relationships to ensure the data is within the specified limits for your instance, and try the upload again.
2. If you are confident you have not exceeded these limits, please raise a ticket with our [Customer Support](#) team.

Fallback

This error can be triggered when the uploaded file is not recognized as a valid Neo4j dump file.

Solution

1. Check the file and try again.
2. If you are confident the file being uploaded is correct, please raise a ticket with our [Customer Support](#) team.

Driver integration

JavaScript routing table error

JavaScript driver version 4.4.5 and greater assumes the existence of database connectivity. When the connection fails, the two most common error messages are "Session Expired" or a routing table error:

Routing table error

```
Neo4jError: Could not perform discovery.  
No routing servers available.  
Known routing table: RoutingTable[database=default database, expirationTime=0, currentTime=1644933316983,  
routers=[], readers=[], writers=[]]
```

This error can also be encountered when no default database is defined.

Solution

Verify connectivity before creating a session object, and specify the default database in your driver definition.

Verifying connectivity

```
const session = driver.session({ database: "neo4j" })  
driver.verifyConnectivity()  
  
let session = driver.session(...)
```



Rapid session creation can exceed the database's maximum concurrent connection limit, resulting in the "Session Expired" error when creating more sessions.

Obtain the project ID

Use cURL to obtain the project ID with your token. Replace `YOUR_BEARER_TOKEN` with your token.

```
curl --location 'https://api.neo4j.io/v1/projects' --header 'Accept: application/json' --header 'Authorization: Bearer YOUR_BEARER_TOKEN'
```

This returns something similar to:

```
{"data":[{"id":"6e6bbbe2-5678-5f8a-1234-b1f62f08b98f","name":"team1"}, {"id":"ad69ee24-1234-5678-af02-ff8d3cc23611","name":"team2"}]}
```

In the example response above, two projects are returned. If you're a member of multiple projects, select the one you wish to use.



Project replaces *Tenant* in the console UI and documentation. However, in the API, `tenant` remains the nomenclature.

Configure an AuraDB instance

Configure the instance values

Use the bearer token and Project ID to create the Aura instance. Replace `YOUR_BEARER_TOKEN` with your token. Replace `YOUR_PROJECT_ID` with your project ID.

The following values are customizable `version`, `region`, `memory`, `name`, `type`, `tenant_id`, and `cloud_provider`.

```
curl --location 'https://api.neo4j.io/v1/instances' --header 'Content-Type: application/json' --header 'Accept: application/json' --header 'Authorization: Bearer YOUR_BEARER_TOKEN' --data '{ "version": "5", "region": "europe-west1", "memory": "8GB", "name": "instance01", "type": "enterprise-db", "tenant_id": "YOUR_PROJECT_ID", "cloud_provider": "gcp" }'
```

See [Aura API documentation](#) for more details.

At this point, an Aura instance is provisioned in the Aura Console. Optionally, use this code in the terminal to check the status:

```
curl --location 'https://api.neo4j.io/v1/instances/YOUR_INSTANCE_ID' --header 'Accept: application/json' --header 'Authorization: Bearer YOUR_BEARER_TOKEN'
```

Response

```
curl --location 'https://api.neo4j.io/v1/instances/YOUR_INSTANCE_ID' --header 'Accept: application/json' --header 'Authorization: Bearer YOUR_BEARER_TOKEN'
```

If the value of `status` shows `running`, you can start using the new Aura instance.

License

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

Attribution

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial

You may not use the material for commercial purposes.

ShareAlike

If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions

You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See <https://creativecommons.org/licenses/by-nc-sa/4.0/> for further details. The full license text is available at <https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>.