HELPING THE WORLD MAKE SENSE OF DATA

neo4j

NEO4J GRAPH DATA SCIENCE CONFIGURATION GUIDE

Joe Depeau,
Dr. Alicia Frame, and
Luke Gannon

Table of Contents

Introduction

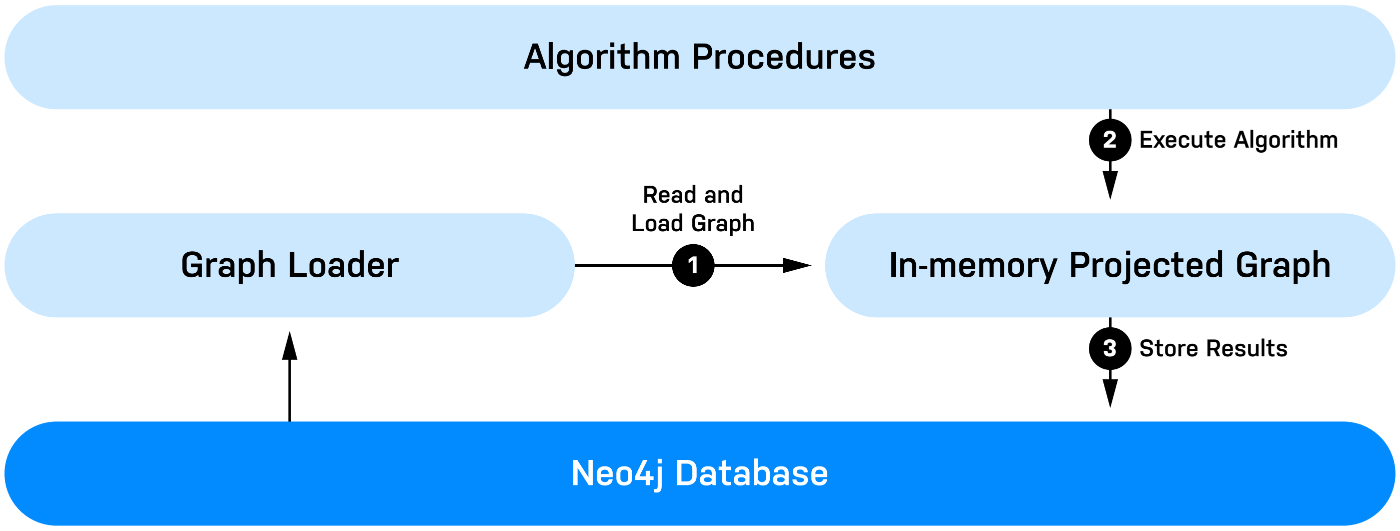
The Neo4j Graph Data Science (GDS) Library is part of Neo4j's framework for Graph Data Science, which also includes the Neo4j Graph Database for graph persistence and Neo4j Bloom for graph exploration. The GDS Library is integrated with Neo4j's native graph database for working with complex and highly connected data at enterprise scale. This framework enables data scientists in a wide range of industries to harness the natural power of relationships and network structures to infer behavior. Implementing the GDS Library, however, is quite different from running a transactional or operational Neo4j database.

The GDS Library provides both an enterprise-ready analytics workspace and a collection of powerful and efficient algorithms for graph data science and machine learning. The categories of algorithms include community detection, centrality and importance, similarity, link prediction, pathfinding, graph embeddings, and graph-native machine learning.

Data science workloads are memory intensive and highly parallelized, and GDS requires properly configured hardware to show its full capabilities. This guide will help you align hardware resourcing and architectural design with the particular demands of GDS workloads to avoid poor performance and instability.

Using the GDS Library

The GDS Library contains over 60 graph algorithms that operate in an analytics workspace and a model catalog for trained ML models. Most of your tasks will involve executing algorithms as part of a three-step process:



Step 1: Read and load graph

Before running an algorithm in GDS, a graph projection must be created. To do this, the GDS graph loader function must first read the data from the Neo4j database – either from page cache or from disk if the data is not already in the cache.

The GDS graph loader function loads this data into the graph projection, which resides in the heap space portion of memory. This graph is typically an aggregated, transformed, and optimized version of the actual data from your Neo4j database, designed to enable the algorithms to run efficiently on the data.

Step 2: Execute algorithm

GDS algorithms are executed against the graph projection. The GDS Library includes functions to estimate how much memory is needed for the projection as well as for algorithm execution. Before running any algorithm, you can use the estimate mode to calculate the expected memory requirements. The GDS Library does include memory guardrails that prevent algorithms from running when insufficient memory is available, as well as procedures to understand how much memory is utilized by other users and workflows, but it's best to size your database appropriately rather than risking an out of memory error.

Your database must have enough heap available for the peak memory consumption to support loading your projected graphs and to execute your algorithms on the graph projections. When you have multiple users executing data science workloads concurrently, you'll need to ensure you have enough memory to support all of those users.

Step 3: Return results

The results of the GDS Library algorithms may be handled in a number of ways.

- **Mutation:** The mutate mode of graph algorithms can feed new data into a projected graph so you can chain workflows of algorithms together.
- **Streaming:** Results may be streamed to Neo4j Browser, Cypher Shell, or a driver, without storing them, or
- **Writing:** Results may be stored permanently by writing them back to a Neo4j database, enriching it with features such as scores, community identifiers, or new relationships.

The GDS Library also provides graph export functions to export your analytics graph to another Neo4j database or to a CSV file.

Neo4j offers a Community Edition and an Enterprise Edition of the GDS Library. All core features are available in both editions: 60+ graph algorithms, graph embeddings, and machine learning, as well as the Pregel API to write your own graph algorithms. The table on the right illustrates how these two editions compare.

	Community Edition	Enterprise Edition
Algorithms & Embedding	✓	✓
Pregel API	✓	✓
Parallelization	≤ 4 CPUs	unlimited
Low-Memory Analytics Graph Format	✗	✓
Model Catalog	3 models	unlimited
Model Persistence & Publication	✗	✓
Fine-Grained Security	✗	✓

Our algorithms are highly parallelized for maximum scalability. The Enterprise Edition enables unlimited utilization of the available machine CPU cores to handle massive workloads. Likewise, we created a low-memory graph projection format that compresses the in-memory graph by up to 75% so enterprises can more easily run data science tasks over extremely large graphs.

Using the Enterprise Edition, teams of data scientists can compare model results and work more collaboratively with an unlimited number of machine learning models that can be stored, published, and shared between database users. Finally, Enterprise Edition users can also take advantage of the Neo4j database role-based access control (RBAC) to fine-tune security.

Configuring Neo4j for Graph Data Science

The GDS Library requires a different hardware resourcing and architectural approach compared to running a transactional Neo4j database. Key considerations for your operational architecture include:

- **Physical architecture:** The GDS Library runs on a dedicated single instance of Neo4j, or on a read replica as part of a cluster.
- **Memory:** GDS is memory-intensive, and specifically allocating sufficient heap space for the algorithms and for the number of projected graphs is key, while having sufficient page cache enables you to quickly load your graph from disk and execute Cypher queries.
- **CPU:** Allocating more CPU cores allows individual algorithms to run with greater parallelization – and finish faster. It also allows more users to run algorithms at the same time without running out of threads.
- **Algorithm tuning:** Each algorithm may be tuned to run faster or consume fewer resources. It is important to review and select the optimum algorithm tuning options.
- **Moving data in and out of the GDS Library instance:** There are many ways of getting data in and out of a GDS Library instance and into applications or other databases. The appropriate option will depend on existing or planned architecture, as well as individual use cases.

Integrating Neo4j with Your Data Ecosystem

There's always a need to integrate Neo4j workflows with your data ecosystem. From flat files like JSON or CSV to consuming data from external APIs or from another database, you can retrieve data from nearly any source and use it to populate your database.

Importing data: Although loading from a CSV is one of the most common approaches, you can also use the Admin Import tool for fast offline bulk ingestion. Alternatively integrate with tools like Apache Spark using the Neo4j Connector for Apache Spark to get data from data lakes and DataFrames into Neo4j. The developer guides and product documentation contain more information.

Exporting data: Once your data is in Neo4j and the GDS Library is generating new features and enhancing your graph, there may be a need to export new data out of the Graph Data Science Neo4j instance into another system. Our procedure library, APOC, adds support for exporting data into various data formats, including JSON, CSV, GraphML, and Cypher Script. Alternatively, you can use the Neo4j Connector for Apache Spark to get data into DataFrames to integrate with the rest of your ML workflow.

Neo4j offers drivers for several languages including Java, JavaScript, Python, .NET, and Go. There are community-developed drivers for a number of other languages such as R and Julia.

Physical Architecture

There are many options for where and how to run a database designed for the GDS Library. The database can be deployed on-premise, in the cloud, or using Docker.

The Graph Data Science Library is intended to run on a large dedicated single instance machine configured with sufficient RAM and CPUs. GDS runs on heap, and algorithms are highly parallelized, so the amount of memory available will determine the size of the workload that you can execute while CPUs determine the speed.

Starting with Neo4j 4.3, it's possible to configure read replicas as GDS instances, supporting read-only workloads (such as analytics queries), or backing applications like Bloom. Alternatively, single instance architectures can support disaster recovery (warm backups).

In contrast, transactional Neo4j use cases typically deploy a causal cluster of database servers. The GDS Library is not compatible with core members of these clusters because of the long-running and computationally intensive nature of executing graph algorithms. Graph algorithms could interfere with the core cluster's transactional workloads by consuming resources required for quorum, replication, and servicing queries. This can lead to poor cluster performance. It will interfere with intra-cluster communication and operations, and might even result in instability within the core cluster.

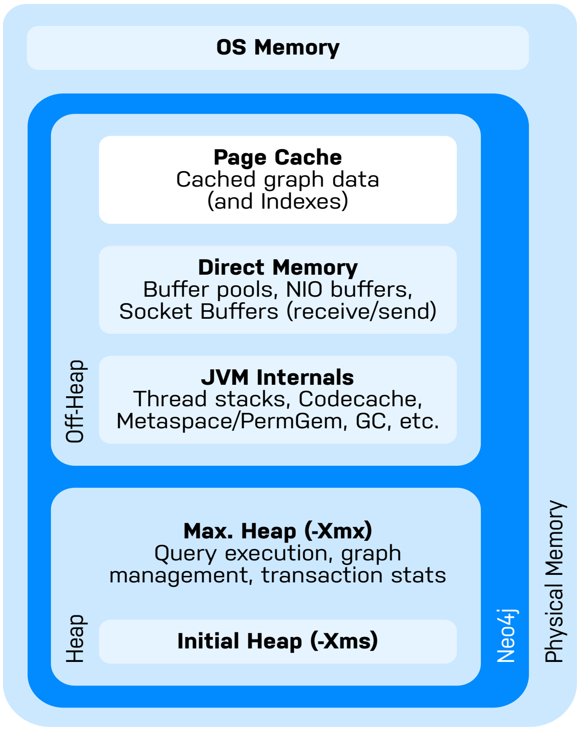
In hybrid use cases, when using GDS in conjunction with a causal cluster, it is important to understand how you intend to consume the results of your algorithm runs. We do not recommend running GDS in a clustered environment unless absolutely necessary. In these cases, GDS should only run on a read replica. In this scenario, it is common to use Kafka and the Neo4j Streams Integration to feed the GDS Library output on the read replica back to the rest of the cluster. Alternatively, in some scenarios, customers can create and detach read replicas and run them as single instances, or export results to a CSV and write data back to the leader by way of a script.

Whether the optimum design is a dedicated single Neo4j instance or a read replica as part of a cluster, it's important to configure the system with appropriate memory and CPU resources to deliver the performance needed to handle the dataset and algorithms.

Memory

For any Neo4j installation, sizing the memory for the database is an important consideration. When configuring a Neo4j database server, two main types of memory can be allocated:

- **Heap Space:** Used for query execution, graph management, maintaining transaction state, storing the in-memory graph, and executing algorithms.
- **Page Cache:** Used to cache the Neo4j data stored on disk. The caching of graph data and indexes into memory helps avoid costly disk access and results in optimal performance when querying the database and projecting graphs for graph data science.


OS Memory
Page Cache
Cached graph data (and Indexes)
Direct Memory
Buffer pools, NIO buffers, Socket Buffers (receive/send)
JVM Internals
Thread stacks, Codecache, Metaspace/PermGem, GC, etc.
Off-Heap
Max. Heap (-Xmx)
Query execution, graph management, transaction stats
Initial Heap (-Xms)
Heap
Neo4j
Physical Memory

The GDS Library uses the same types of memory and configurations as other Neo4j installations, but it takes a slightly different approach to using memory. We recommend allocating as much heap as possible while still providing sufficient page cache to load your data and support Cypher queries. In situations where you are memory constrained, allocating memory to heap is more important than page cache.

CPU

More CPUs enable our highly parallel algorithms to execute faster and support more algorithms being executed concurrently.

It is important to note, however, that different graph algorithms scale across multiple CPU cores in different ways. Some scale in a linear fashion, where the more cores are available, the faster the algorithm can be run. However, certain algorithms like Louvain will not scale as well beyond 16–32 CPUs. The nature of the computations being performed, including a high degree of recursive computation, means that thread management on high numbers of CPU cores incurs more overhead cost than the benefit delivered.

When provisioning a GDS instance to support multiple concurrent users, additional cores will be needed. For each user who wants to run algorithms at the same time, add up the total number of CPU cores they want to use. For example, if you have one user who wants to run PageRank using eight cores and another user who wants to run Louvain at the same time using eight cores, you would need at least 16 cores. That does not include the CPUs needed for running the operating system and any other applications on the server.

Scaling Neo4j with GDS

The GDS Library was built for enterprise-scale datasets: we have customers running GDS on terabytes of data with tens of billions of nodes. Our algorithms are highly parallelized, and our infrastructure is built to minimize memory consumption.

Scale Up: Efficient Operations at Massive Scale

The GDS Library executes on heap, so for large datasets, more memory is always better. Our algorithm implementations are highly parallelized, so higher concurrency will mean faster calculations. Optimizing configuration parameters like sampling rate, maximum iterations, and tolerance can speed up performance as well.

Enterprise Only Memory Compression

As of GDS 1.5, GDS Enterprise Edition users have access to a new data structure that decreases the memory footprint of the in-memory graph by up to 75%. This feature is on by default but can be disabled, and is intended for use cases with very large graphs or memory constrained environments.

Memory Consumption Comparison

GDS Enterprise Edition (BitIDMap) vs. Community Edition, GDS v1.5

Graph Dataset	Nodes	Relationships	Community Edition Memory Usage	Enterprise Edition Memory Usage	Percent Change
LDBC100	317,216,062	2,154,861,632	12.5GB	7.7GB	**-38%**
LDBC100PKP	499,000	46,598,276	140MB	130MB	**-7%**

The LDBC100 is created from the LDBC Social Network Scale Factor 100 dataset, and the LDBC100PKP graph is created using only the (Person)-[KNOWS]->(Person) data.

Algorithm Tuning

Every algorithm has its own set of parameters, so it is important to review the algorithm's documentation to identify relevant configuration settings for your use case. Some algorithms, like Weakly Connected Components, can also be seeded with data – often from a previous run of the algorithm – which can help the algorithm to run more quickly.

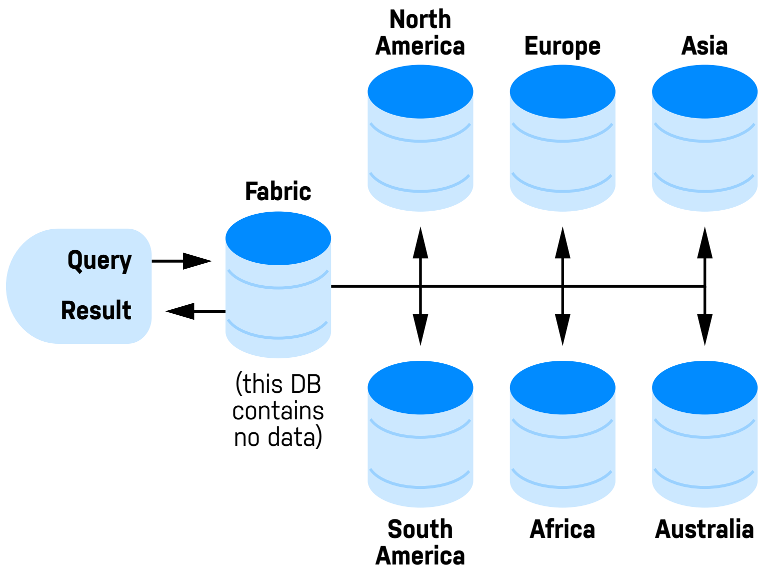
Common tuning parameters include:

- **maxIterations:** The maximum number of iterations of an algorithm to run. More iterations may mean more accurate results, but will take longer.
- **tolerance:** Minimum change in scores between iterations.
- **concurrency:** The number of threads that should be used in your calculation. More is usually faster, but Community Edition is limited to 4.
- **seed:** A seed property can be loaded from a previous run of an algorithm, or based on prior knowledge, to calculate reproducible results or speed up calculations.

Scale Out: Running GDS with Neo4j Fabric (Sharding)

Neo4j Fabric is a way to store and retrieve data in multiple databases, whether they are on the same Neo4j DBMS or in multiple DBMSs, using a single Cypher query. For more information about Fabric itself, please visit the documentation.

The Neo4j Fabric architecture consists of two core concepts: one or many shards that hold the data, along with the inclusion of one or many Fabric virtual databases or proxy DBMS that coordinate the distributed queries. GDS operations should be executed on each of the shards.



The graph projections and algorithms must be executed on each shard individually, and the results can be combined via the Fabric proxy. This scenario is useful if the graph is partitioned into disjoint subgraphs across shards, i.e., there is no logical relationship between nodes on different shards. Another option is to replicate the graph's topology across multiple shards, where some shards act as operational databases and others as analytical databases.

For more information, see "Using GDS and Fabric" and "Sharding Graph Data with Neo4j Fabric."

Performance Examples

There are many factors that impact performance, from your hardware configuration to algorithm settings as referred to in previous sections.

Below are performance examples on two public benchmark datasets. For each dataset, we report the execution time and memory consumed.

Machine: AWS EC2 R5D16XLarge
CPU: Intel Xeon Platinum 8000 (Skylake-SP or Cascade Lake)
Logical Cores: 64
Memory: 512GB
Storage: 600GB NVMe-SSD

Neo4j Configuration
Neo4j GDS Library Version 1.7.0 - Enterprise Edition
Neo4j Version: 4.3.3 - Enterprise Edition
Max Heap: 400GB
Page Cache: 50GB

Graph Create Performance

The following table compares the data sizes and times to create two graph projections. The LDBC100 is created from the LDBC Social Network Scale Factor 100 dataset, and the LDBC100PKP graph is created using only the (Person)-[KNOWS]->(Person) data.

Graph Dataset	Nodes	Relationships	Seconds to Create	Memory Used
LDBC100	317,216,062	2,154,861,632	91	7.7GB
LDBC100PKP	499,000	46,598,276	9	130MB

Running Algorithms

The following table provides a few examples of algorithm run times on the LDBC100 and LDBC100PKP graphs. Algorithm run times are for the algorithm execution only, not including .stream or .write actions.

Graph Algorithm	Graph Dataset	Run Time	Memory Used
Label Propagation	LDBC100	46 seconds	65GB
Weakly Connected Components	LDBC100	36 seconds	59GB
Triangle Counting	LDBC100	24.8 minutes	70GB
Local Clustering Coefficient	LDBC100	4.76 minutes	45.9GB
FastRP	LDBC100	1.33 minutes	212GB
PageRank	LDBC100	1 minute	45.9GB
Louvain Modularity	LDBC100	14.66 minutes	45.9GB
Node Similarity	LDBC100PKP	20 minutes	716MB
Betweenness Centrality	LDBC100PKP	10.28 minutes	35GB
Node2Vec	LDBC100PKP	2.83 minutes	1.2GB

Summary

The Neo4j framework for Graph Data Science provides a high-performance, highly scalable way to execute graph algorithms and machine learning. This document provides some guidance for properly configuring and tuning your implementation to your needs. Neo4j offers a number of resources to support organizations as they develop hardware resourcing and architectural design for the GDS Library.

Resources

Start here:
https://neo4j.com/graph-data-science/

The current manual for the Neo4j Graph Data Science Library:
https://neo4j.com/docs/graph-data-science/current/

The Neo4j Download Center:
https://neo4j.com/download-center/

Free online Neo4j training, including Data Science
and Applied Graph Algorithms modules:
https://neo4j.com/graphacademy/online-training/

For more information or assistance, please contact your
account representative or reach out to the Neo4j Community:
https://community.neo4j.com/