



# The Neo4j Graph Data Science Library Manual v1.8

[[graph-data-science]]

# Table of Contents

1. Introduction	2
1.1. Algorithms	2
1.2. Graph Catalog	3
1.3. Editions	3
2. Installation	4
2.1. Supported Neo4j versions.	4
2.2. Neo4j Desktop	5
2.3. Neo4j Server	5
2.4. Enterprise Edition Configuration	6
2.5. Neo4j Docker	7
2.6. Neo4j Causal Cluster	7
2.7. Additional configuration options	8
2.8. System Requirements	8
3. Common usage	11
3.1. Memory Estimation	12
3.2. Creating graphs	16
3.3. Running algorithms	16
3.4. Logging	19
3.5. Monitoring system	20
4. Graph management	23
4.1. Graph Catalog	23
5. Export a named graph to CSV	83
5.1. Syntax	83
5.2. Estimation	84
5.3. Export format	85
5.4. Example	86
5.5. Example with additional node properties	86
5.6. Anonymous graphs	87
5.7. Node Properties	88
5.8. Utility functions	89
5.9. Cypher on GDS graph <a href="#">Enterprise edition</a>	92
5.10. Administration	94
6. Model catalog	97
6.1. Listing models	97
6.2. Checking if a model exists	99
6.3. Removing models	99
6.4. Storing models on disk	101
6.5. Publishing models	103

7. Algorithms	105
7.1. Syntax overview	105
7.2. Centrality	106
7.3. Community detection	220
7.4. Similarity	345
7.5. Path finding	429
7.6. Topological link prediction	507
7.7. Node embeddings	517
7.8. Machine learning models	566
7.9. Auxiliary procedures	643
7.10. Pregel API	664
8. End-to-end examples	674
8.1. FastRP and kNN example	674
9. Production deployment	679
9.1. Transaction Handling	679
10. Transaction termination	683
10.1. Using GDS and Fabric	683
10.2. GDS Feature Toggles	684
Appendix A: Operations reference	685
Appendix B: Migration from Graph Algorithms v3.5	696

The manual covers the following areas:

- [Introduction](#) — An introduction to the Neo4j Graph Data Science library.
- [Installation](#) — Instructions for how to install and use the Neo4j Graph Data Science library.
- [Common usage](#) — General usage patterns and recommendations for getting the most out of the Neo4j Graph Data Science library.
- [Graph management](#) — A detailed guide to the graph catalog and utility procedures included in the Neo4j Graph Data Science library.
- [Model catalog](#) — A detailed guide to the model catalog and utility procedures included in the Neo4j Graph Data Science library.
- [Algorithms](#) — A detailed guide to each of the algorithms in their respective categories, including use-cases and examples.
- [Production deployment](#) — This chapter explains advanced details with regards to common Neo4j components.
- [Operations reference](#) — Reference of all procedures contained in the Neo4j Graph Data Science library.
- [Migration from Graph Algorithms v3.5](#) — Additional resources - migration guide, books, etc - to help using the Neo4j Graph Data Science library.

The source code of the library is available at [GitHub](#). If you have a suggestion on how we can improve the library or want to report a problem, you can create a [new issue](#).



# Chapter 1. Introduction

This library provides efficiently implemented, parallel versions of common graph algorithms for Neo4j, exposed as Cypher procedures.

## 1.1. Algorithms

Graph algorithms are used to compute metrics for graphs, nodes, or relationships.

They can provide insights on relevant entities in the graph (centralities, ranking), or inherent structures like communities (community-detection, graph-partitioning, clustering).

Many graph algorithms are iterative approaches that frequently traverse the graph for the computation using random walks, breadth-first or depth-first searches, or pattern matching.

Due to the exponential growth of possible paths with increasing distance, many of the approaches also have high algorithmic complexity.

Fortunately, optimized algorithms exist that utilize certain structures of the graph, memoize already explored parts, and parallelize operations. Whenever possible, we've applied these optimizations.

The Neo4j Graph Data Science library contains a large number of algorithms, which are detailed in the [Algorithms](#) chapter.

### 1.1.1. Algorithm traits

Algorithms in GDS have specific ways to make use of various aspects of its input graph(s). We call these *algorithm traits*. When an algorithm supports an algorithm trait this indicates that the algorithm has been implemented to produce well-defined results in accordance with the trait. The following algorithm traits exist:

#### *Directed*

The algorithm is well-defined on a directed graph.

#### *Undirected*

The algorithm is well-defined on an undirected graph.

#### *Homogeneous*

The algorithm will treat all nodes and relationships in its input graph(s) similarly, as if they were all of the same type. If multiple types of nodes or relationships exist in the graph, this must be taken into account when analysing the results of the algorithm.

#### *Heterogeneous*

The algorithm has the ability to distinguish between nodes and/or relationships of different types.

#### *Weighted*

The algorithm supports configuration to set node and/or relationship properties to use as weights. These values can represent cost, time, capacity or some other domain-specific property, specified via

the [nodeWeightProperty](#) and [relationshipWeightProperty](#) configuration parameters. The algorithm will by default consider each node and/or relationship as equally important.

## 1.2. Graph Catalog

In order to run the algorithms as efficiently as possible, the Neo4j Graph Data Science library uses a specialized in-memory graph format to represent the graph data. It is therefore necessary to load the graph data from the Neo4j database into an in memory graph catalog. The amount of data loaded can be controlled by so called graph projections, which also allow, for example, filtering on node labels and relationship types, among other options.

For more information see [Graph Management](#).

## 1.3. Editions

The Neo4j Graph Data Science library is available in two editions.

- The open source Community Edition includes all algorithms and features, but is limited to four CPU cores.
- The Neo4j Graph Data Science library Enterprise Edition:
  - Can run on an unlimited amount of CPU cores.
  - Supports the role-based access control system (RBAC) from Neo4j Enterprise Edition.
  - Supports various additional model catalog features
    - Storing unlimited amounts of models in the model catalog
    - [Publishing a stored model](#)
    - [Persisting a stored model to disk](#)
  - Supports an [optimized in-memory graph implementation](#)

For more information see [System Requirements - CPU](#).

# Chapter 2. Installation

The Neo4j Graph Data Science (GDS) library is delivered as a plugin to the Neo4j Graph Database. The plugin needs to be installed into the database and added to the allowlist in the Neo4j configuration. There are two main ways of achieving this, which we will detail in this chapter.

This chapter is divided into the following sections:

1. [Supported Neo4j versions](#)
2. [Neo4j Desktop](#)
3. [Neo4j Server](#)
4. [Enterprise Edition Configuration](#)
5. [Neo4j Docker](#)
6. [Neo4j Causal Cluster](#)
7. [Additional configuration options](#)
8. [System Requirements](#)

## 2.1. Supported Neo4j versions

Below is the compatibility matrix for The GDS library vs Neo4j. In general, you can count on the latest version of GDS supporting the latest version of Neo4j and vice versa, and we recommend you always upgrade to that combination.

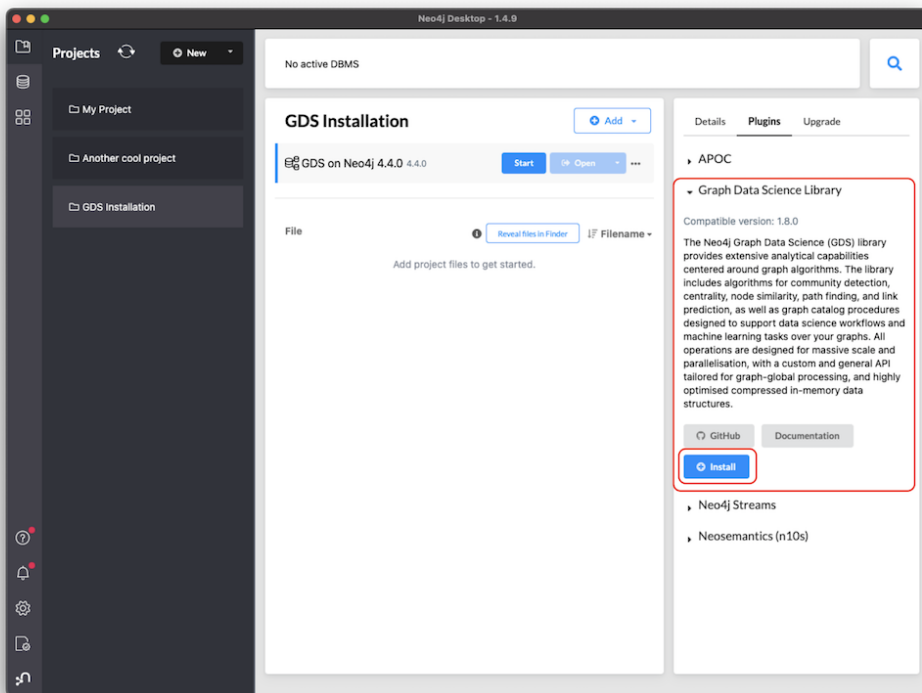
We list software with major and minor version only, e.g. GDS library 1.5. You should read that as any patch version of that major+minor version, but again, do upgrade to the latest patch always, to ensure you get all bug fixes included.

Not finding your version of GDS or Neo4j listed? Time to upgrade!

Neo4j Graph Data Science	Neo4j version
1.8.8 <sup>[1]</sup>	4.4
	4.3
	4.2
	4.1 <sup>[2]</sup>
1.7 <sup>[1]</sup>	4.3
	4.2
	4.1 <sup>[3]</sup>
1.1 <sup>[1]</sup>	3.5

## 2.2. Neo4j Desktop

The most convenient way of installing the GDS library is through the [Neo4j Desktop](#) plugin called Neo4j Graph Data Science. The plugin can be found in the 'Plugins' tab of a database.



The installer will download the GDS library and install it in the 'plugins' directory of the database. It will also add the following entry to the settings file:

```
dbms.security.procedures.unrestricted=gds.*
```

This configuration entry is necessary because the GDS library accesses low-level components of Neo4j to maximise performance.

If the procedure allowlist is configured, make sure to also include procedures from the GDS library:

```
dbms.security.procedures.allowlist=gds.*
```



Before **Neo4j 4.2**, the configuration setting is called `dbms.security.procedures.whitelist`

## 2.3. Neo4j Server

The GDS library is intended to be used on a standalone Neo4j server.



Running the GDS library in a Neo4j Causal Cluster is not supported. Read more about how to use GDS in conjunction with Neo4j Causal Cluster deployment [below](#).

On a standalone Neo4j Server, the library will need to be installed and configured manually.

1. Download `neo4j-graph-data-science-[version].jar` from the [Neo4j Download Center](#) and copy it into the `$NEO4J_HOME/plugins` directory.
2. Add the following to your `$NEO4J_HOME/conf/neo4j.conf` file:

```
dbms.security.procedures.unrestricted=gds.*
```

This configuration entry is necessary because the GDS library accesses low-level components of Neo4j to maximise performance.

3. Check if the procedure allowlist is enabled in the `$NEO4J_HOME/conf/neo4j.conf` file and add the GDS library if necessary:

```
dbms.security.procedures.allowlist=gds.*
```



Before Neo4j 4.2, the configuration setting is called `dbms.security.procedures.whitelist`

4. Restart Neo4j

### 2.3.1. Verifying installation

To verify your installation, the library version can be printed by entering into the browser in Neo4j Desktop and calling the `gds.version()` function:

```
RETURN gds.version()
```

To list all installed algorithms, run the `gds.list()` procedure:

```
CALL gds.list()
```

## 2.4. Enterprise Edition Configuration

Unlocking the Enterprise Edition of the Neo4j Graph Data Science library requires a valid license key. To register for a license, please contact Neo4j at <https://neo4j.com/contact-us/?ref=graph-analytics>.

The license is issued in the form of a license key file, which needs to be placed in a directory accessible by the Neo4j server. You can configure the location of the license key file by setting the `gds.enterprise.license_file` option in the `neo4j.conf` configuration file of your Neo4j installation. The location must be specified using an absolute path. It is necessary to restart the database when configuring the license key for the first time and every time the license key is changed, e.g., when a new license key is added or the location of the key file changes.

Example configuration for the license key file:

```
gds.enterprise.license_file=/path/to/my/license/keyfile
```

If the `gds.enterprise.license_file` setting is set to a non-empty value, the Neo4j Graph Data Science library will verify that the license key file is accessible and contains a valid license key. When a valid license key is configured, all Enterprise Edition features are unlocked. In case of a problem, e.g, when the license key file is inaccessible, the license has expired or is invalid for any other reason, all calls to the Neo4j Graph Data Science Library will result in an error, stating the problem with the license key.

## 2.5. Neo4j Docker

The Neo4j Graph Data Science library is available as a plugin for Neo4j on Docker. The plugins guide for Docker is found at the [operations manual](#).

To run a Neo4j Container with GDS available, you can run

```
docker run -it --rm \
  --publish=7474:7474 --publish=7687:7687 \
  --user="$(id -u):$(id -g)" \
  -e NEO4J_AUTH=none \
  --env NEO4JLABS_PLUGINS='["graph-data-science"]' \
  neo4j:4.2
```

## 2.6. Neo4j Causal Cluster

A Neo4j Causal Cluster consists of multiple machines that together support a highly available database management system. The GDS library uses main memory on a single machine for hosting graphs in the graph catalog and computing algorithms over these. These two architectures are not compatible and should not be used in conjunction. A GDS workload will attempt to consume most of the system resources of the machine during runtime, which may make the machine unresponsive for extended periods of time. For these reasons, we strongly advise against running GDS in a cluster as this potentially leads to data corruption or cluster outage.

To make use of GDS on graphs hosted by a Neo4j Causal Cluster deployment, these graphs should be detached from the running cluster. This can be accomplished in several ways, including:

1. Dumping a snapshot of the Neo4j store and importing it in a separate standalone Neo4j server.
2. Adding a Read Replica to the Neo4j Causal Cluster and then detaching it to safely operate GDS on a snapshot in separation from the Neo4j Causal Cluster.
3. Adding a Read Replica to the Neo4j Causal Cluster and configuring it for GDS workloads. Be aware that the in-memory graph and the underlying database will eventually become out of sync due to updates to the Read Replica. Since GDS can consume all available resources, responsiveness of the Read Replica might decrease and its state might fall behind the cluster. Using GDS in this scenario requires:
  - installing GDS on the Read Replica
  - using mutate or stream invocation modes
  - consuming results from GDS workloads directly via Cypher (see [Utility functions](#))
  - **not using GDS write-back features** (writing triggers many large transactions and will potentially terminate the cluster)

After the GDS workload has finished on a detached machine (for cases 1. and 2.) it now contains out-of-sync results written to its copied version of the graph from the Neo4j Causal Cluster. To integrate these results back to the cluster, custom programs are necessary.

## 2.7. Additional configuration options

In order to make use of certain features of the GDS library, additional configuration is necessary. Configuration is done in the `neo4j.conf` configuration file before starting the DBMS. The following features require such additional configuration:

### 2.7.1. Graph export

Exporting [graphs to CSV](#) files requires the configuration parameter `gds.export.location` to be set to the absolute path to the folder in which exported graphs will be stored. This directory has to be writable by the Neo4j process.

### 2.7.2. Model persistence

The [model persistence feature](#) requires the configuration parameter `gds.model.store_location` to be set to the absolute path to the folder in which the models will be stored. This directory has to be writable by the Neo4j process.

## 2.8. System Requirements

### 2.8.1. Main Memory

The GDS library runs within a Neo4j instance and is therefore subject to the general [Neo4j memory configuration](#).

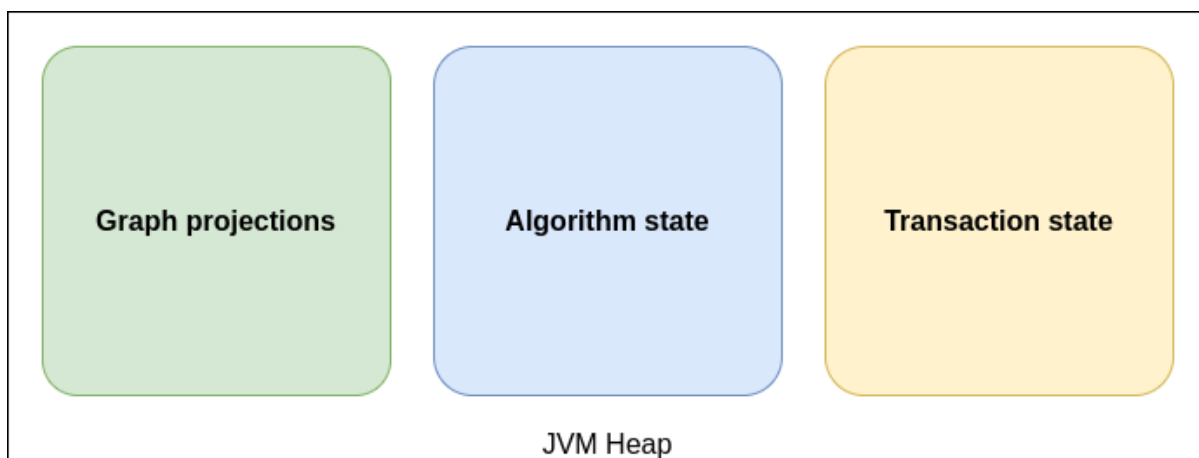


Figure 1. GDS heap memory usage

### Heap size

The heap space is used for storing graph projections in the graph catalog and algorithm state. When writing algorithm results back to Neo4j, heap space is also used for handling transaction state (see [dbms.tx\\_state.memory\\_allocation](#)). For purely analytical workloads, a general recommendation is to set the

heap space to about 90% of the available main memory. This can be done via [dbms.memory.heap.initial\\_size](#) and [dbms.memory.heap.max\\_size](#).

To better estimate the heap space required to create in-memory graphs and run algorithms, consider the [Memory Estimation](#) feature. The feature estimates the memory consumption of all involved data structures using information about number of nodes and relationships from the Neo4j count store.

## Page cache

The page cache is used to cache the Neo4j data and will help to avoid costly disk access.

For purely analytical workloads including [native projections](#), it is recommended to decrease [dbms.memory.pagecache.size](#) in favor of an increased heap size. However, setting a minimum page cache size is still important while creating in-memory graphs:

- For [native projections](#), the minimum page cache size for creating the in-memory graph can be roughly estimated by  $8KB * 100 * readConcurrency$ .
- For [Cypher projections](#), a higher page cache is required depending on the query complexity.

However, if it is required to write algorithm results back to Neo4j, the write performance is highly depended on store fragmentation as well as the number of properties and relationships to write. We recommend starting with a page cache size of roughly  $250MB * writeConcurrency$  and evaluate write performance and adapt accordingly. Ideally, if the [memory estimation](#) feature has been used to find a good heap size, the remaining memory can be used for page cache and OS.



Decreasing the page cache size in favor of heap size is **not** recommended if the Neo4j instance runs both, operational and analytical workloads at the same time. See [Neo4j memory configuration](#) for general information about page cache sizing.

## 2.8.2. CPU

The library uses multiple CPU cores for graph projections, algorithm computation, and results writing. Configuring the workloads to make best use of the available CPU cores in your system is important to achieve maximum performance. The concurrency used for the stages of projection, computation and writing is configured per algorithm execution, see [Common Configuration parameters](#)

The default concurrency used for most operations in the Graph Data Science library is 4.

The maximum concurrency that can be used is limited depending on the license under which the library is being used:

- Neo4j Graph Data Science Library - Community Edition (GDS CE)
  - The maximum concurrency in the library is limited to 4.
- Neo4j Graph Data Science Library - Enterprise Edition (GDS EE)
  - The maximum concurrency in the library is unlimited. To register for a license, please contact Neo4j at <https://neo4j.com/contact-us/?ref=graph-data-science>.





Concurrency limits are determined based on whether you have a GDS EE license, or if you are using GDS CE. The maximum concurrency limit in the graph data science library is not set based on your edition of the Neo4j database.

[1] This version series is end-of-life and will not receive further patches. Please use a later version.

[2] There is a bug in Neo4j 4.1.1 that can lead to an exception when using Cypher projection. If possible, use the latest patch version.

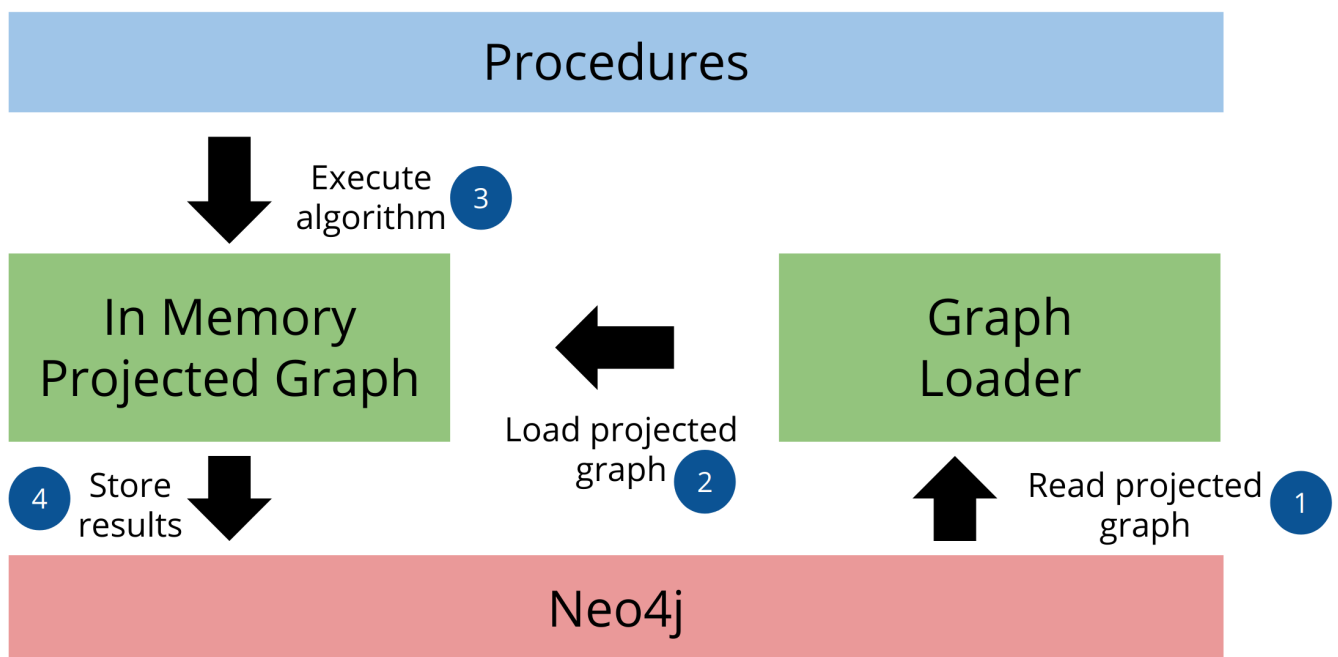
[3] There is a bug in Neo4j 4.1.1 that can lead to an exception when using Cypher projection. If possible, use the latest patch version.

# Chapter 3. Common usage

The GDS library usage pattern is typically split in two phases: development and production. In the development phase the goal is to establish a workflow of useful algorithms. In order to do this, the system must be configured, graph projections must be defined, and algorithms must be selected. It is typical to make use of the memory estimation features of the library. This enables you to successfully configure your system to handle the amount of data to be processed. There are two kinds of resources to keep in mind: the in-memory graph and the algorithm data structures.

In the production phase, the system would be configured appropriately to successfully run the desired algorithms. The sequence of operations would normally be to create a graph, run one or more algorithms on it, and consume results.

The below image illustrates an overview of standard operation of the GDS library:



	<p>The GDS library runs its procedures greedily in terms of system resources. That means that each procedure will try to use:</p> <ul style="list-style-type: none"><li>• as much memory as it needs (see <a href="#">Memory estimation</a>)</li><li>• as many CPU cores as it needs (not exceeding the limits of the <b>concurrency</b> it's configured to run with)</li></ul> <p>Concurrently running procedures share the resources of the system hosting the DBMS and as such may affect each other's performance. To get an overview of the status of the system you can use the <a href="#">System monitor procedure</a>.</p>
--	---

The more detail on each individual operation, see the corresponding section:

1. [Graph Catalog](#)
2. [Creating graphs](#)

### 3. Running algorithms

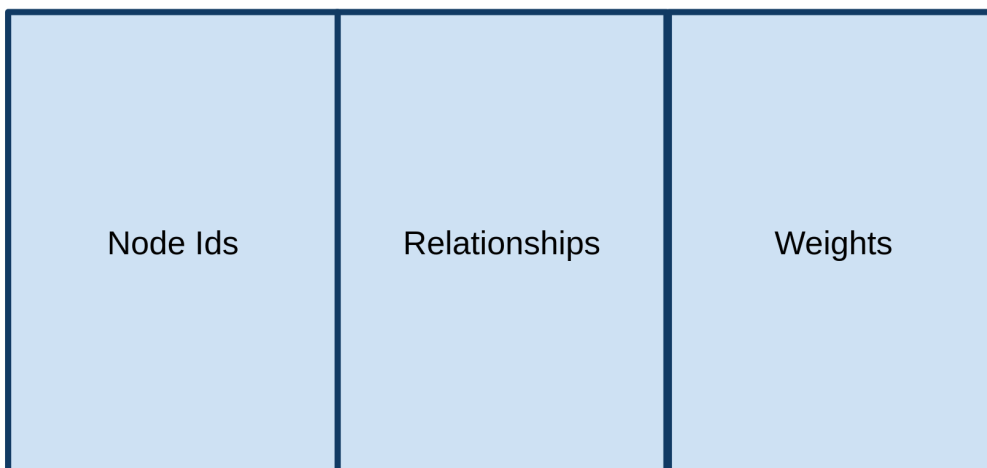
In this chapter, we will go through these aspects and guide you towards the most useful operations.

This chapter is divided into the following sections:

- [Memory Estimation](#)
- [Creating graphs](#)
- [Running algorithms](#)
- [Logging](#)
- [Monitoring system](#)

## 3.1. Memory Estimation

The graph algorithms library operates completely on the heap, which means we'll need to configure our Neo4j Server with a much larger heap size than we would for transactional workloads. The diagram below shows how memory is used by the projected graph model:



In Memory Graph Model

The model contains three types of data:

- Node ids - up to  $2^{45}$  ("35 trillion")
- Relationships - pairs of node ids. Relationships are stored twice if `orientation: "UNDIRECTED"` is used.
- Weights - stored as doubles (8 bytes per node) in an array-like data structure next to the relationships

Memory configuration depends on the graph projection that we're using.

### 3.1.1. Estimating memory requirements for algorithms

In many use cases it will be useful to estimate the required memory of projecting a graph and running an algorithm before running it in order to make sure that the workload can run on the available free memory. To do this the `.estimate` mode can be used, which returns an estimate of the amount of memory required to run graph algorithms. Note that only algorithms in the production-ready tier are guaranteed to have an `.estimate` mode. For more details please refer to [Syntax overview](#).

## Syntax outline:

```
CALL gds[.<tier>].<algorithm>.<execution-mode>.estimate(  
  graphNameOrConfig: String or Map, configuration: Map  
) YIELD  
  nodeCount: Integer,  
  relationshipCount: Integer,  
  requiredMemory: String,  
  treeView: String,  
  mapView: Map,  
  bytesMin: Integer,  
  bytesMax: Integer,  
  heapPercentageMin: Float,  
  heapPercentageMax: Float
```

Table 1. Parameters

Name	Type	Default	Optional	Description
graphNameOr Config	String or Map	-	no	The name of the projected graph or the algorithm configuration in the case of an anonymous graph.
configuration	Map	{}	yes	If the first parameter is the name of a projected graph, this parameter is the algorithm config, otherwise it needs to be null or an empty map.

The configuration map accepts the same configuration parameters as the estimated algorithm. See the specific algorithm documentation for more information.

Table 2. Results

Name	Type	Description
nodeCount	Integer	The number of nodes in the graph.
relationship Count	Integer	The number of relationships in the graph.
requiredMemo ry	String	An estimation of the required memory in a human readable format.
treeView	String	A more detailed representation of the required memory, including estimates of the different components in human readable format.
mapView	Map	A more detailed representation of the required memory, including estimates of the different components in structured format.
bytesMin	Integer	The minimum number of bytes required.
bytesMax	Integer	The maximum number of bytes required.
heapPercenta geMin	Float	The minimum percentage of the configured maximum heap required.
heapPercenta geMax	Float	The maximum percentage of the configured maximum heap required.

### 3.1.2. Estimating memory requirements for graphs

The `gds.graph.create` procedures also support `.estimate` to estimate memory usage for just the graph. Those procedures don't accept the graph name as the first argument, as they don't actually create the graph.

## Syntax

```
CALL gds.graph.create.estimate(nodeProjection: String|List|Map, relationshipProjection: String|List|Map,
configuration: Map)
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, heapPercentageMin, heapPercentageMax,
nodeCount, relationshipCount
```

The `nodeProjection` and `relationshipProjection` parameters follow the same syntax as in `gds.graph.create`.

Table 3. Parameters

Name	Type	Default	Optional	Description
nodeProjection	String or List or Map	-	no	The node projection to estimate for.
relationshipProjection	String or List or Map	-	no	The relationship projection to estimate for.
configuration	Map	{}	yes	Additional configuration, such as concurrency.

The result of running `gds.graph.create.estimate` has the same form as the algorithm memory estimation results above.

It is also possible to estimate the memory of a fictive graph, by explicitly specifying its node and relationship count. Using this feature, one can estimate the memory consumption of an arbitrarily sized graph.

To achieve this, use the following configuration options:

Table 4. Configuration

Name	Type	Default	Optional	Description
nodeCount	Integer	0	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	0	yes	The number of relationships in a fictive graph.

When estimating a fictive graph, syntactically valid `nodeProjection` and `relationshipProjection` must be specified. However, it is recommended to specify `'*'` for both in the fictive graph case as this does not interfere with the specified values above.

The query below is an example of estimating a fictive graph with 100 nodes and 1000 relationships.

### Example

```
CALL gds.graph.create.estimate('*', '*', {
  nodeCount: 100,
  relationshipCount: 1000,
  nodeProperties: 'foo',
  relationshipProperties: 'bar'
})
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, nodeCount, relationshipCount
```

Table 5. Results

requiredMemory	bytesMin	bytesMax	nodeCount	relationshipCount
"593 KiB"	607576	607576	100	1000

The `gds.graph.create.cypher` procedure has to execute both, the `nodeQuery` and `relationshipQuery`, in order to count the number of nodes and relationships of the graph.

### Syntax

```
CALL gds.graph.create.cypher.estimate(nodeQuery: String, relationshipQuery: String, configuration: Map)
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, heapPercentageMin, heapPercentageMax,
nodeCount, relationshipCount
```

Table 6. Parameters

Name	Type	Default	Optional	Description
nodeQuery	String	-	no	The node query to estimate for.
relationshipQuery	String	-	no	The relationship query to estimate for.
configuration	Map	{}	yes	Additional configuration, such as concurrency.

### 3.1.3. Automatic estimation and execution blocking

All procedures in the GDS library that support estimation, including graph creation, will do an estimation check at the beginning of their execution. This includes all execution modes, but not the `estimate` procedures themselves.

If the estimation check can determine that the current amount of free memory is insufficient to carry through the operation, the operation will be aborted and an error will be reported. The error will contain details of the estimation and the free memory at the time of estimation.

This heap control logic is restrictive in the sense that it only blocks executions that are certain to not fit into memory. It does not guarantee that an execution that passed the heap control will succeed without depleting memory. Thus, it is still useful to first run the estimation mode before running an algorithm or graph creation on a large data set, in order to view all details of the estimation.

The free memory taken into consideration is based on the Java runtime system information. The amount of free memory can be increased by either [dropping](#) unused graphs from the catalog, or by [increasing the maximum heap size](#) prior to starting the Neo4j instance.

### Bypassing heap control

Occasionally you will want the ability to bypass heap control if it is too restrictive. You might have insights into how your particular procedure call will behave, memory-wise; or you might just want to take a chance e.g. because the memory estimate you received is very close to system limits.

For that use case we have `sudo mode` which allows you to manually skip heap control and run your procedure regardless. Sudo mode is off by default to protect users - we fail fast if we can see your potentially long-running procedure would not be able to complete successfully.

To enable sudo mode, add the sudo parameter when calling a procedure. Here is an example of calling the popular Louvain community detection algorithm in sudo mode:

Run Louvain in sudo mode:

```
CALL gds.louvain.write('myGraph', { writeProperty: 'community', sudo: true })
YIELD communityCount, modularity, modularities
```

Accidentally enabling sudo mode when calling a procedure, causing it to run out of memory, will not significantly damage your installation, but it will waste your time.

## 3.2. Creating graphs

In order for any algorithm in the GDS library to run, we must first create a graph to run on. The graph is created as either an *anonymous graph* or a *named graph*. An anonymous graph is created for just a single algorithm and will be lost after its execution has finished. A named graph is given a name and stored in the graph catalog. For a detailed guide on all graph catalog operations, see [Graph Catalog](#).

Creating a named graph has several advantages:

- it can be used by multiple algorithms
- the creation is cleanly separated from the algorithm execution
- the algorithm runtime can be measured in isolation
- the configuration for creating the graph may be retrieved from the graph catalog

Using an anonymous graph has the advantage that a single query may be used for an entire algorithm computation. This can be especially useful in the development phase when the workflow is being set up and the graph projections are experimented with.

## 3.3. Running algorithms

All algorithms are exposed as Neo4j procedures. They can be called directly from Cypher using Neo4j Browser, `cypher-shell`, or from your client code using a Neo4j Driver in the language of your choice.

For a detailed guide on the syntax to run algorithms, please see the [Syntax overview](#) section. In short, algorithms are run using one of the execution modes `stream`, `stats`, `mutate` or `write`, which we cover in this chapter.

The execution of any algorithm can be canceled by terminating the Cypher transaction that is executing the procedure call. For more on how transactions are used, see [Transaction Handling](#).

### 3.3.1. Stream

The `stream` mode will return the results of the algorithm computation as Cypher result rows. This is similar to how standard Cypher reading queries operate.

The returned data can be a node ID and a computed value for the node (such as a Page Rank score, or WCC componentId), or two node IDs and a computed value for the node pair (such as a Node Similarity

similarity score).

If the graph is very large, the result of a `stream` mode computation will also be very large. Using the `ORDER BY` and `LIMIT` subclauses in the Cypher query could be useful to support 'top N'-style use cases.

### 3.3.2. Stats

The `stats` mode returns statistical results for the algorithm computation like counts or percentile distributions. A statistical summary of the computation is returned as a single Cypher result row. The direct results of the algorithm are not available when using the `stats` mode. This mode forms the basis of the `mutate` and `write` execution modes but does not attempt to make any modifications or updates anywhere.

### 3.3.3. Mutate

The `mutate` mode will write the results of the algorithm computation back to the in-memory graph. Note that the specified `mutateProperty` value must not exist in the in-memory graph beforehand. This enables running multiple algorithms on the same in-memory graph without writing results to Neo4j in-between algorithm executions.

This execution mode is especially useful in three scenarios:

- Algorithms can depend on the results of previous algorithms without the need to write to Neo4j.
- Algorithm results can be written altogether (see [write node properties](#) and [write relationships](#)).
- Algorithm results can be queried via Cypher without the need to write to Neo4j at all (see [gds.util.nodeProperty](#)).

A statistical summary of the computation is returned similar to the `stats` mode. Mutated data can be node properties (such as Page Rank scores), new relationships (such as Node Similarity similarities), or relationship properties.

### 3.3.4. Write

The `write` mode will write the results of the algorithm computation back to the Neo4j database. This is similar to how standard Cypher writing queries operate. A statistical summary of the computation is returned similar to the `stats` mode. This is the only execution mode that will attempt to make modifications to the Neo4j database.

The written data can be node properties (such as Page Rank scores), new relationships (such as Node Similarity similarities), or relationship properties. The `write` mode can be very useful for use cases where the algorithm results would be inspected multiple times by separate queries since the computational results are handled entirely by the library.

In order for the results from a `write` mode computation to be used by another algorithm, a new graph must be created from the Neo4j database with the updated graph.



### 3.3.5. Common Configuration parameters

All algorithms allow adjustment of their runtime characteristics through a set of configuration parameters. Although some of the parameters are algorithm-specific, many are shared between algorithms and execution modes.



To learn more about algorithm specific parameters and to find out if an algorithm supports a certain parameter, please consult the algorithm-specific documentation page.

*List of the most commonly accepted configuration parameters*

*concurrency - Integer*

Controls the parallelism with which the algorithm is executed. By default this value is set to 4. For more details on the concurrency settings and limitations please see [the CPU section](#) of the System Requirements.

*nodeLabels - List of String*

If the graph, on which the algorithm is run, was created with multiple node label projections, this parameter can be used to select only a subset of the projected labels. The algorithm will only consider nodes with the selected labels.

*relationshipTypes - List of String*

If the graph, on which the algorithm is run, was created with multiple relationship type projections, this parameter can be used to select only a subset of the projected types. The algorithm will only consider relationships with the selected types.

*nodeWeightProperty - String*

In algorithms that support node weights this parameter defines the node property that contains the weights.

*relationshipWeightProperty - String*

In algorithms that support relationship weights this parameter defines the relationship property that contains the weights. The specified property is required to exist in the specified graph on all specified [relationship types](#). The values must be numeric, and some algorithms may have additional value restrictions, such as requiring only positive weights.

*maxIterations - Integer*

For iterative algorithms this parameter controls the maximum number of iterations.

*tolerance - Float*

Many iterative algorithms accept the tolerance parameter. It controls the minimum delta between two iterations. If the delta is less than the tolerance value, the algorithm is considered converged and stops.

*seedProperty - String*

Some algorithms can be calculated incrementally. This means that results from a previous execution can be taken into account, even though the graph has changed. The [seedProperty](#) parameter defines the node property that contains the seed value. Seeding can speed up computation and write times.

`writeProperty` - String

In `write` mode this parameter sets the name of the node or relationship property to which results are written. If the property already exists, existing values will be overwritten.

`writeConcurrency` - Integer

In `write` mode this parameter controls the parallelism of write operations. The Default is `concurrency`

## 3.4. Logging

In the GDS library there are two types of logging: debug logging and progress logging.

**Debug logging** provides information about events in the system. For example, when an algorithm computation completes, the amount of memory used and the total runtime may be logged. Exceptional events, when an operation fails to complete normally, are also logged. The debug log information is useful for understanding events in the system, especially when troubleshooting a problem.

**Progress logging** is performed to track the progress of operations that are expected to take a long time. This includes graph projections, algorithm computation, and result writing.

All log entries are written to the log files configured for the Neo4j database. For more information on configuring Neo4j logs, please refer to the [Neo4j Operations Manual](#).

### 3.4.1. Progress-logging procedure Beta

Progress is also tracked by the GDS library itself. This makes it possible to inspect progress via Cypher, in addition to looking in the log files. To access progress information for currently running tasks (also referred to as jobs), we can make use of the list progress procedure: `gds.beta.listProgress`. A task in the GDS library is defined as a running procedure, such as an algorithm or a graph load procedure.

The list progress procedure has two modes, depending on whether a `jobId` parameter was set: First, if `jobId` is not set, the procedure will produce a single row for each task currently running. This can be seen as the summary of those tasks, displaying the overall progress of a particular task for example. Second, if the `jobId` parameter is set it will show a detailed view for the given running job. The detailed view will produce a row for each step or task that job will perform during execution. It will also show how tasks are structured as a tree and print progress for each individual task.

## Syntax

Getting the progress of tasks:

```
CALL gds.beta.listProgress(jobId: String)
YIELD
  jobId,
  taskName,
  progress,
  progressBar,
  status,
  timeStarted,
  elapsedTime
```

Table 7. Parameters

Name	Type	Default	Optional	Description
jobId	String	""	yes	The jobId of a running task. This will trigger a detailed overview for that particular task.

Table 8. Results

Name	Type	Description
jobId	String	A generated identifier of the running task.
taskName	String	The name of the running task, i.e. <code>Node2Vec</code> .
progress	String	The progress of the job shown as a percentage value.
progressBar	String	The progress of the job shown as an ASCII progress bar.
status	String	The current status of the job, i.e. <code>RUNNING</code> or <code>CANCELED</code> .
timeStarted	LocalTime	The local wall clock time when the task has been started.
elapsedTime	Duration	The duration from <code>timeStarted</code> to now.

## Examples

Assuming we just started `gds.beta.node2vec.stream` procedure.

```
CALL gds.beta.listProgress()
YIELD
  jobId,
  taskName,
  progress
```

Table 9. Results

jobId	taskName	progress
"d21bb4ca-e1e9-4a31-a487-42ac8c9c1a0d"	"Node2Vec"	"42%"

## 3.5. Monitoring system

GDS supports multiple users concurrently working on the same system. Typically, GDS procedures are resource heavy in the sense that they may use a lot of memory and/or many CPU cores to do their computation. To know whether it is a reasonable time for a user to run a GDS procedure it is useful to know the current capacity of the system hosting Neo4j and GDS, as well as the current GDS workload on the system. Graphs and models are not shared between non-admin users by default, however GDS users on the same system will share its capacity.

### 3.5.1. System monitor procedure Alpha

To be able to get an overview of the system's current capacity and its analytics workload one can use the procedure `gds.alpha.systemMonitor`. It will give you information on the capacity of the DBMS's JVM instance in terms of memory and CPU cores, and an overview of the resources consumed by the GDS

procedures currently being run on the system.

## Syntax

Monitor the system capacity and analytics workload:

```
CALL gds.alpha.systemMonitor()  
YIELD  
  freeHeap,  
  totalHeap,  
  maxHeap,  
  jvmAvailableCpuCores,  
  availableCpuCoresNotRequested,  
  jvmHeapStatus,  
  ongoingGdsProcedures
```

Table 10. Results

Name	Type	Description
freeHeap	Integer	The amount of currently free memory in bytes in the Java Virtual Machine hosting the Neo4j instance.
totalHeap	Integer	The total amount of memory in bytes in the Java virtual machine hosting the Neo4j instance. This value may vary over time, depending on the host environment.
maxHeap	Integer	The maximum amount of memory in bytes that the Java virtual machine hosting the Neo4j instance will attempt to use.
jvmAvailableCpuCores	Integer	The number of logical CPU cores currently available to the Java virtual machine. This value may change vary over the lifetime of the DBMS.
availableCpuCoresNotRequested	Integer	The number of logical CPU cores currently available to the Java virtual machine that are not requested for use by currently running GDS procedures. Note that this number may be negative in case there are fewer available cores to the JVM than there are cores being requested by ongoing GDS procedures.
jvmHeapStatus	Map	The above-mentioned heap metrics in human-readable form.
ongoingGdsProcedures	List of Map	A list of maps containing resource usage and progress information for all GDS procedures (of all users) currently running on the Neo4j instance. Each map contains the name of the procedure, how far it has progressed, its estimated memory usage as well as how many CPU cores it will try to use at most.



**freeHeap** is influenced by ongoing GDS procedures, graphs stored the [Graph catalog](#) and the underlying Neo4j DBMS. Stored graphs can take up a significant amount of heap memory. To inspect the graphs in the graph catalog you can use the [Graph list](#) procedure.

## Example

First let us assume that we just started `gds.beta.node2vec.stream` procedure with some arbitrary parameters.

We can have a look at the status of the JVM heap.

### Monitor JVM heap status:

```
CALL gds.alpha.systemMonitor()  
YIELD  
  freeHeap,  
  totalHeap,  
  maxHeap
```

Table 11. Results

freeHeap	totalHeap	maxHeap
1234567	2345678	3456789

We can see that there currently is around **1.23 MB** free heap memory in the JVM instance running our Neo4j DBMS. This may increase independently of any procedures finishing their execution as **totalHeap** is currently smaller than **maxHeap**. We can also inspect CPU core usage as well as the status of currently running GDS procedures on the system.

### Monitor CPU core usage and ongoing GDS procedures:

```
CALL gds.alpha.systemMonitor()  
YIELD  
  availableCpuCoresNotRequested,  
  jvmAvailableCpuCores,  
  ongoingGdsProcedures
```

Table 12. Results

jvmAvailableCpuCores	availableCpuCoresNotRequested	ongoingGdsProcedures
100	84	[[ procedure: "Node2Vec", progress: "33.33%", estimatedMemoryRange: "[123 kB ... 234 kB]", requestedNumberOfCpuCores: "16" ]]

Here we can note that there is only one GDS procedure currently running, namely the **Node2Vec** procedure we just started. It has finished around **33.33%** of its execution already. We also see that it may use up to an estimated **234 kB** of memory. Note that it may not currently be using that much memory and so it may require more memory later in its execution, thus possible lowering our current **freeHeap**. Apparently it wants to use up to **16** CPU cores, leaving us with a total of **84** currently available cores in the system not requested by any GDS procedures.

# Chapter 4. Graph management

A central concept in the GDS library is the management of in-memory graphs.

This chapter is divided into the following sections:

- [Graph Catalog](#)
- [Anonymous graphs](#)
- [Node Properties](#)
- [Utility functions](#)
- [Cypher on GDS graph](#)
- [Administration](#)

## 4.1. Graph Catalog

Graph algorithms run on a graph data model which is a *projection* of the Neo4j property graph data model. A graph projection can be seen as a materialized view over the stored graph, containing only analytically relevant, potentially aggregated, topological and property information. Graph projections are stored entirely in-memory using compressed data structures optimized for topology and property lookup operations.

The graph catalog is a concept within the GDS library that allows managing multiple graph projections by name. Using its name, a created graph can be used many times in the analytical workflow. Named graphs can be created using either a [Native projection](#) or a [Cypher projection](#). After usage, named graphs can be removed from the catalog to free up main memory.

Graphs can also be created when running an algorithm without placing them in the catalog. We refer to such graphs as [anonymous graphs](#).



The graph catalog exists as long as the Neo4j instance is running. When Neo4j is restarted, graphs stored in the catalog are lost and need to be re-created.

This chapter explains the available graph catalog operations.

Name	Description
<a href="#">gds.graph.create</a>	Creates a graph in the catalog using Native projection.
<a href="#">gds.graph.create.cypher</a>	Creates a graph in the catalog using Cypher projection.
<a href="#">gds.beta.graph.create.subgraph</a>	Creates a graph in the catalog by filtering an existing graph using node and relationship predicates.
<a href="#">gds.graph.list</a>	Prints information about graphs that are currently stored in the catalog.
<a href="#">gds.graph.exists</a>	Checks if a named graph is stored in the catalog.
<a href="#">gds.graph.removeNodeProperties</a>	Removes node properties from a named graph.

Name	Description
<code>gds.graph.deleteRelationships</code>	Deletes relationships of a given relationship type from a named graph.
<code>gds.graph.drop</code>	Drops a named graph from the catalog.
<code>gds.graph.streamNodeProperty</code>	Streams a single node property stored in a named graph.
<code>gds.graph.streamNodeProperties</code>	Streams node properties stored in a named graph.
<code>gds.graph.streamRelationshipProperty</code>	Streams a single relationship property stored in a named graph.
<code>gds.graph.streamRelationshipProperties</code>	Streams relationship properties stored in a named graph.
<code>gds.graph.writeNodeProperties</code>	Writes node properties stored in a named graph to Neo4j.
<code>gds.graph.writeRelationship</code>	Writes relationships stored in a named graph to Neo4j.
<code>gds.graph.export</code>	Exports a named graph into a new offline Neo4j database.
<code>gds.beta.graph.export.csv</code>	Exports a named graph into CSV files.



Creating, using, listing, and dropping named graphs are management operations bound to a Neo4j user. Graphs created by a different Neo4j user are not accessible at any time.

### 4.1.1. Creating graphs

A projected graph can be stored in the [catalog](#) under a user-defined name. Using that name, the graph can be referred to by any algorithm in the library. This allows multiple algorithms to use the same graph without having to re-create it on each algorithm run.

Native projections provide the best performance by reading from the Neo4j store files. Recommended for both the development, and the production phase.



There is also a way to generate a random graph, see [Graph Generation](#) documentation for more details.



The projected graphs will reside in the catalog until:

- the graph is dropped using `gds.graph.drop`
- the Neo4j database from which to graph was projected is stopped or dropped
- the Neo4j database management system is stopped.

### Syntax

A native projection takes three mandatory arguments: `graphName`, `nodeProjection` and `relationshipProjection`. In addition, the optional `configuration` parameter allows us to further configure the graph creation.

```
CALL gds.graph.create(
  graphName: String,
  nodeProjection: String or List or Map,
  relationshipProjection: String or List or Map,
  configuration: Map
)
YIELD
  graphName: String,
  nodeProjection: Map,
  nodeCount: Integer,
  relationshipProjection: Map,
  relationshipCount: Integer,
  createMillis: Integer
```



To get information about a stored graph, such as its schema, one can use [gds.graph.list](#).

Table 13. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
nodeProjection	String, List or Map	no	One or more <a href="#">node projections</a> .
relationshipProjection	String, List or Map	no	One or more <a href="#">relationship projections</a> .
configuration	Map	yes	Additional parameters to configure the native projection.

Table 14. Configuration

Name	Type	Default	Description
readConcurrency	Integer	4	The number of concurrent threads used for creating the graph.
nodeProperties	String, List or Map	{}	The node properties to load for <i>all</i> node projections.
relationshipProperties	String, List or Map	{}	The relationship properties to load for <i>all</i> relationship projections.
validateRelationships	Boolean	false	Whether to throw an error if the <code>relationshipProjection</code> includes relationships between nodes not part of the <code>nodeProjection</code> .

Table 15. Results

Name	Type	Description
graphName	String	The name under which the graph is stored in the catalog.
nodeProjection	Map	The <a href="#">node projections</a> used to project the graph.
nodeCount	Integer	The number of nodes stored in the projected graph.
relationshipProjection	Map	The <a href="#">relationship projections</a> used to project the graph.
relationshipCount	Integer	The number of relationships stored in the projected graph.
createMillis	Integer	Milliseconds for creating the graph.



## Node Projection

Short-hand String-syntax for `nodeProjection`. The projected graph will contain the given `neo4j-label`.

```
<neo4j-label>
```

Short-hand List-syntax for `nodeProjection`. The projected graph will contain the given `neo4j-label`'s.

```
[<neo4j-label>, ..., <neo4j-label>]
```

Extended Map-syntax for `nodeProjection`.

```
{
  <projected-label>: {
    label: <neo4j-label>,
    properties: <neo4j-property-key>
  },
  <projected-label>: {
    label: <neo4j-label>,
    properties: [<neo4j-property-key>, <neo4j-property-key>, ...]
  },
  ...
  <projected-label>: {
    label: <neo4j-label>,
    properties: {
      <projected-property-key>: {
        property: <neo4j-property-key>,
        defaultValue: <fallback-value>
      },
      ...
      <projected-property-key>: {
        property: <neo4j-property-key>,
        defaultValue: <fallback-value>
      }
    }
  }
}
```

Table 16. Node Projection fields

Name	Type	Optional	Default	Description
<code>&lt;projected-label&gt;</code>	String	no	n/a	The node label in the projected graph.
<code>label</code>	String	yes	<code>projected-label</code>	The node label in the Neo4j graph. If not set, uses the <code>projected-label</code> .
<code>properties</code>	Map, List or String	yes	<code>{}</code>	The projected node properties for the specified <code>projected-label</code> .
<code>&lt;projected-property-key&gt;</code>	String	no	n/a	The key for the node property in the projected graph.
<code>property</code>	String	yes	<code>projected-property-key</code>	The node property key in the Neo4j graph. If not set, uses the <code>projected-property-key</code> .

Name	Type	Optional	Default	Description
defaultValue	Float	yes	Double.NaN	The default value if the property is not defined for a node.
	Float[]		null	
	Integer		Integer.MIN_VALUE	
	Integer[]		null	

## Relationship Projection

Short-hand String-syntax for `relationshipProjection`. The projected graph will contain the given `neo4j-type`.

```
<neo4j-type>
```

Short-hand List-syntax for `relationshipProjection`. The projected graph will contain the given `neo4j-type`s`.

```
[<neo4j-type>, ..., <neo4j-type>]
```

Extended Map-syntax for `relationshipProjection`.

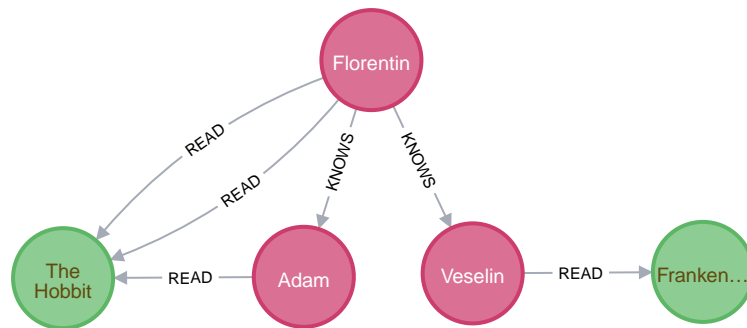
```
{
  <projected-type>: {
    type: <neo4j-type>,
    orientation: <orientation>,
    aggregation: <aggregation-type>,
    properties: <neo4j-property-key>
  },
  <projected-type>: {
    type: <neo4j-type>,
    orientation: <orientation>,
    aggregation: <aggregation-type>,
    properties: [<neo4j-property-key>, <neo4j-property-key>]
  },
  ...
  <projected-type>: {
    type: <neo4j-type>,
    orientation: <orientation>,
    aggregation: <aggregation-type>,
    properties: {
      <projected-property-key>: {
        property: <neo4j-property-key>,
        defaultValue: <fallback-value>,
        aggregation: <aggregation-type>
      },
      ...
      <projected-property-key>: {
        property: <neo4j-property-key>,
        defaultValue: <fallback-value>,
        aggregation: <aggregation-type>
      }
    }
  }
}
```

Table 17. Relationship Projection fields

Name	Type	Optional	Default	Description
<projected-type>	String	no	n/a	The name of the relationship type in the projected graph.
type	String	yes	projected-type	The relationship type in the Neo4j graph.
orientation	String	yes	NATURAL	Denotes how Neo4j relationships are represented in the projected graph. Allowed values are NATURAL, UNDIRECTED, REVERSE.
aggregation	String	no	NONE	Handling of parallel relationships. Allowed values are NONE, MIN, MAX, SUM, SINGLE, COUNT.
properties	Map, List or String	yes	{}	The projected relationship properties for the specified projected-type.
<projected-property-key>	String	no	n/a	The key for the relationship property in the projected graph.
property	String	yes	projected-property-key	The node property key in the Neo4j graph. If not set, uses the projected-property-key.
defaultValue	Float or Integer	yes	Double.NaN	The default value if the property is not defined for a node.

## Examples

In order to demonstrate the GDS Graph Create capabilities we are going to create a small social network graph in Neo4j. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
(florentin:Person { name: 'Florentin', age: 16 }),
(adam:Person { name: 'Adam', age: 18 }),
(veselin:Person { name: 'Veselin', age: 20, ratings: [5.0] }),
(hobbit:Book { name: 'The Hobbit', isbn: 1234, numberOfPages: 310, ratings: [1.0, 2.0, 3.0, 4.5] }),
(frankenstein:Book { name: 'Frankenstein', isbn: 4242, price: 19.99 }),

(florentin)-[:KNOWS { since: 2010 }]->(adam),
(florentin)-[:KNOWS { since: 2018 }]->(veselin),
(florentin)-[:READ { numberOfPages: 4 }]->(hobbit),
(florentin)-[:READ { numberOfPages: 42 }]->(hobbit),
(adam)-[:READ { numberOfPages: 30 }]->(hobbit),
(veselin)-[:READ]->(frankenstein)

```

## Simple graph

A simple graph is a graph with only one node label and relationship type, i.e., a monopartite graph. We are going to start with demonstrating how to load a simple graph by projecting only the **Person** node label and **KNOWS** relationship type.

Project **Person** nodes and **KNOWS** relationships:

```
CALL gds.graph.create(  
  'persons',           ①  
  'Person',           ②  
  'KNOWS'             ③  
)  
YIELD  
  graphName AS graph, nodeProjection, nodeCount AS nodes, relationshipProjection, relationshipCount AS  
  rels
```

- ① The name of the graph. Afterwards, **persons** can be used to run algorithms or manage the graph.
- ② The nodes to be projected. In this example, the nodes with the **Person** label.
- ③ The relationships to be projected. In this example, the relationships of type **KNOWS**.

Table 18. Results

graph	nodeProjection	nodes	relationshipProjection	rels
"persons"	{Person={label=Person, properties={}}}	3	{KNOWS={orientation=NATURAL, aggregation=DEFAULT, type=KNOWS, properties={}}}	2

In the example above, we used a short-hand syntax for the node and relationship projection. The used projections are internally expanded to the full **Map** syntax as shown in the **Results** table. In addition, we can see the projected in-memory graph contains three **Person** nodes, and the two **KNOWS** relationships.

## Multi-graph

A multi-graph is a graph with multiple node labels and relationship types.

To project multiple node labels and relationship types, we can adjust the projections as follows:

Project **Person** and **Book** nodes and **KNOWS** and **READ** relationships:

```
CALL gds.graph.create(  
  'personsAndBooks',  ①  
  ['Person', 'Book'], ②  
  ['KNOWS', 'READ']  ③  
)  
YIELD  
  graphName AS graph, nodeProjection, nodeCount AS nodes, relationshipCount AS rels
```

- ① Projects a graph under the name **personsAndBooks**.
- ② The nodes to be projected. In this example, the nodes with a **Person** or **Book** label.
- ③ The relationships to be projected. In this example, the relationships of type **KNOWS** or **READ**.

Table 19. Results

graph	nodeProjection	nodes	rels
"personsAndBooks"	{Book={label=Book, properties={}}, Person={label=Person, properties={}}}	5	6

In the example above, we used a short-hand syntax for the node and relationship projection. The used projections are internally expanded to the full `Map` syntax as shown for the `nodeProjection` in the Results table. In addition, we can see the projected in-memory graph contains five nodes, and the two relationships.

## Relationship orientation

By default, relationships are loaded in the same orientation as stored in the Neo4j db. In GDS, we call this the `NATURAL` orientation. Additionally, we provide the functionality to load the relationships in the `REVERSE` or even `UNDIRECTED` orientation.

Project `Person` nodes and undirected `KNOWS` relationships:

```
CALL gds.graph.create(
  'undirectedKnows',           ①
  'Person',                   ②
  {KNOWS: {orientation: 'UNDIRECTED'}} ③
)
YIELD
  graphName AS graph,
  relationshipProjection AS knowsProjection,
  nodeCount AS nodes,
  relationshipCount AS rels
```

- ① Projects a graph under the name `undirectedKnows`.
- ② The nodes to be projected. In this example, the nodes with the `Person` label.
- ③ Projects relationships with type `KNOWS` and specifies that they should be `UNDIRECTED` by using the `orientation` parameter.

Table 20. Results

graph	knowsProjection	nodes	rels
"undirectedKnows"	{KNOWS={orientation=UNDIRECTED, aggregation=DEFAULT, type=KNOWS, properties={}}}	3	4

To specify the orientation, we need to write the `relationshipProjection` with the extended `Map`-syntax. Projecting the `KNOWS` relationships `UNDIRECTED`, loads each relationship in both directions. Thus, the `undirectedKnows` graph contains four relationships, twice as many as the `persons` graph in [Simple graph](#).

## Node properties

To project node properties, we can either use the `nodeProperties` configuration parameter for shared properties, or extend an individual `nodeProjection` for a specific label.

Project **Person** and **Book** nodes and **KNOWS** and **READ** relationships:

```
CALL gds.graph.create(
  'graphWithProperties',           ①
  {                               ②
    Person: {properties: 'age'},  ③
    Book: {properties: {price: {defaultValue: 5.0}}} ④
  },
  ['KNOWS', 'READ'],           ⑤
  {nodeProperties: 'ratings'}  ⑥
)
YIELD
  graphName, nodeProjection, nodeCount AS nodes, relationshipCount AS rels
RETURN graphName, nodeProjection.Book AS bookProjection, nodes, rels
```

- ① Projects a graph under the name `graphWithProperties`.
- ② Use the expanded node projection syntax.
- ③ Projects nodes with the `Person` label and their `age` property.
- ④ Projects nodes with the `Book` label and their `price` property. Each `Book` that doesn't have the `price` property will get the `defaultValue` of `5.0`.
- ⑤ The relationships to be projected. In this example, the relationships of type `KNOWS` or `READ`.
- ⑥ The global configuration, projects node property `rating` on each of the specified labels.

Table 21. Results

graphName	bookProjection	nodes	rels
"graphWithProperties"	{label=Book, properties={price={defaultValue=5.0, property=price}, ratings={defaultValue=null, property=ratings}}}	5	6

The projected `graphWithProperties` graph contains five nodes and six relationships. In the returned `bookProjection` we can observe, the node properties `price` and `ratings` are loaded for `Books`.



GDS currently only supports loading numeric properties.

Further, the `price` property has a default value of `5.0`. Not every book has a price specified in the example graph. In the following we check if the price was correctly projected:

Verify the ratings property of Adam in the projected graph:

```
MATCH (n:Book)
RETURN n.name AS name, gds.util.nodeProperty('graphWithProperties', id(n), 'price') AS price
ORDER BY price
```

Table 22. Results

name	price
"The Hobbit"	5.0
"Frankenstein"	19.99

We can see, that the price was projected with the Hobbit having the default price of `5.0`.

## Relationship properties

Analogous to node properties, we can either use the `relationshipProperties` configuration parameter or extend an individual `relationshipProjection` for a specific type.

Project `Person` and `Book` nodes and `READ` relationships with `numberOfPages` property:

```
CALL gds.graph.create(  
  'readWithProperties',           ①  
  ['Person', 'Book'],           ②  
  {                               ③  
    READ: { properties: "numberOfPages" } ④  
  }  
)  
YIELD  
  graphName AS graph,  
  relationshipProjection AS readProjection,  
  nodeCount AS nodes,  
  relationshipCount AS rels
```

- ① Projects a graph under the name `readWithProperties`.
- ② The nodes to be projected. In this example, the nodes with a `Person` or `Book` label.
- ③ Use the expanded relationship projection syntax.
- ④ Project relationships of type `READ` and their `numberOfPages` property.

Table 23. Results

graph	readProjection	nodes	rels
"readWithProperties"	{READ={orientation=NATURAL, aggregation=DEFAULT, type=READ, properties={numberOfPages={defaultValue=null, property=numberOfPages, aggregation=DEFAULT}}}}	5	4

Next, we will verify that the relationship property `numberOfPages` were correctly loaded.

Stream the relationship property `numberOfPages` of the projected graph:

```
CALL gds.graph.streamRelationshipProperty('readWithProperties', 'numberOfPages')  
YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages  
RETURN  
  gds.util.asNode(sourceNodeId).name AS person,  
  gds.util.asNode(targetNodeId).name AS book,  
  numberOfPages  
ORDER BY person ASC, numberOfPages DESC
```

Table 24. Results

person	book	numberOfPages
"Adam"	"The Hobbit"	30.0
"Florentin"	"The Hobbit"	42.0
"Florentin"	"The Hobbit"	4.0
"Veselin"	"Frankenstein"	NaN

We can see, that the `numberOfPages` property is loaded. The default property value is `Double.NaN` and could be changed using the Map-Syntax the same as for node properties in [Node properties](#).

## Parallel relationships

Neo4j supports parallel relationships, i.e., multiple relationships between two nodes. By default, GDS preserves parallel relationships. For some algorithms, we want the projected graph to contain at most one relationship between two nodes.

We can specify how parallel relationships should be aggregated into a single relationship via the `aggregation` parameter in a relationship projection.

For graphs without relationship properties, we can use the `COUNT` aggregation. If we do not need the count, we could use the `SINGLE` aggregation.

Project `Person` and `Book` nodes and `COUNT` aggregated `READ` relationships:

```
CALL gds.graph.create(  
  'readCount',           ①  
  ['Person', 'Book'],   ②  
  {  
    READ: {              ③  
      properties: {  
        numberOfReads: {  ④  
          property: '*',   ⑤  
          aggregation: 'COUNT' ⑥  
        }  
      }  
    }  
  }  
)  
YIELD  
  graphName AS graph,  
  relationshipProjection AS readProjection,  
  nodeCount AS nodes,  
  relationshipCount AS rels
```

- ① Projects a graph under the name `readCount`.
- ② The nodes to be projected. In this example, the nodes with a `Person` or `Book` label.
- ③ Project relationships of type `READ`.
- ④ Project relationship property `numberOfReads`.
- ⑤ A placeholder, signaling that the value of the relationship property is derived and not based on Neo4j property.
- ⑥ The aggregation type. In this example, `COUNT` results in the value of the property being the number of parallel relationships.

Table 25. Results

graph	readProjection	nodes	rels
"readCount"	{READ={orientation=NATURAL, aggregation=DEFAULT, type=READ, properties={numberOfReads={defaultValue=null, property=*, aggregation=COUNT}}}}	5	3

Next, we will verify that the `READ` relationships were correctly aggregated.



Stream the relationship property `numberOfReads` of the projected graph:

```
CALL gds.graph.streamRelationshipProperty('readCount', 'numberOfReads')
YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfReads
RETURN
  gds.util.asNode(sourceNodeId).name AS person,
  gds.util.asNode(targetNodeId).name AS book,
  numberOfReads
ORDER BY numberOfReads DESC, person
```

Table 26. Results

person	book	numberOfReads
"Florentin"	"The Hobbit"	2.0
"Adam"	"The Hobbit"	1.0
"Veselin"	"Frankenstein"	1.0

We can see, that the two READ relationships between Florentin, and the Hobbit result in 2 `numberOfReads`.

### Parallel relationships with properties

For graphs with relationship properties we can also use other aggregations.

Project `Person` and `Book` nodes and aggregated `READ` relationships by summing the `numberOfPages`:

```
CALL gds.graph.create(
  'readSums',                                     ①
  ['Person', 'Book'],                             ②
  {READ: {properties: {numberOfPages: {aggregation: 'SUM'}}}} ③
)
YIELD
  graphName AS graph,
  relationshipProjection AS readProjection,
  nodeCount AS nodes,
  relationshipCount AS rels
```

- ① Projects a graph under the name `readSums`.
- ② The nodes to be projected. In this example, the nodes with a `Person` or `Book` label.
- ③ Project relationships of type `READ`. Aggregation type `SUM` results in a projected `numberOfPages` property with its value being the sum of the `numberOfPages` properties of the parallel relationships.

Table 27. Results

graph	readProjection	nodes	rels
"readSums"	{READ={orientation=NATURAL, aggregation=DEFAULT, type=READ, properties={numberOfPages={defaultValue=null, property=numberOfPages, aggregation=SUM}}}}	5	3

Next, we will verify that the relationship property `numberOfPages` was correctly aggregated.

Stream the relationship property `numberOfPages` of the projected graph:

```
CALL gds.graph.streamRelationshipProperty('readSums', 'numberOfPages')
YIELD
  sourceNodeId, targetNodeId, propertyValue AS numberOfPages
RETURN
  gds.util.asNode(sourceNodeId).name AS person,
  gds.util.asNode(targetNodeId).name AS book,
  numberOfPages
ORDER BY numberOfPages DESC, person
```

Table 28. Results

person	book	numberOfPages
"Florentin"	"The Hobbit"	46.0
"Adam"	"The Hobbit"	30.0
"Veselin"	"Frankenstein"	0.0

We can see, that the two READ relationships between Florentin and the Hobbit sum up to `46` `numberOfReads`.

Validate relationships flag

As mentioned in the [syntax section](#), the `validateRelationships` flag controls whether an error will be raised when attempting to create a relationship where either the source or target node is not present in the [node projection](#). Note that even if the flag is set to `false` such a relationship will still not be created but the loading process will not be aborted.

We can simulate such a case with the [graph present in the Neo4j database](#):

Project `READ` and `KNOWS` relationships but only `Person` nodes, with `validateRelationships` set to `true`:

```
CALL gds.graph.create(
  'danglingRelationships',
  'Person',
  ['READ', 'KNOWS'],
  {
    validateRelationships: true
  }
)
YIELD
  graphName AS graph,
  relationshipProjection AS readProjection,
  nodeCount AS nodes,
  relationshipCount AS rels
```

## Results

```
org.neo4j.graphdb.QueryExecutionException: Failed to invoke procedure `gds.graph.create`: Caused by:
java.lang.IllegalArgumentException: Failed to load a relationship because its target-node with id 3 is not
part of the node query or projection. To ignore the relationship, set the configuration parameter
`validateRelationships` to false.
```

We can see that the above query resulted in an exception being thrown. The exception message will provide information about the specific node id that was missing, which will help debugging underlying problems.

## 4.1.2. Creating graphs using Cypher

A projected graph can be stored in the catalog under a user-defined name. Using that name, the graph can be referred to by any algorithm in the library. This allows multiple algorithms to use the same graph without having to re-create it on each algorithm run.

Using Cypher projections is a more flexible and expressive approach with diminished focus on performance compared to the [native projections](#). Cypher projections are primarily recommended for the development phase (see [Common usage](#)).



There is also a way to generate a random graph, see [Graph Generation](#) documentation for more details.



The projected graph will reside in the catalog until:

- the graph is dropped using [gds.graph.drop](#)
- the Neo4j database from which the graph was projected is stopped or dropped
- the Neo4j database management system is stopped.

### Syntax

A Cypher projection takes three mandatory arguments: `graphName`, `nodeQuery` and `relationshipQuery`. In addition, the optional `configuration` parameter allows us to further configure graph creation.

```
CALL gds.graph.create.cypher(  
  graphName: String,  
  nodeQuery: String,  
  relationshipQuery: String,  
  configuration: Map  
) YIELD  
  graphName: String,  
  nodeQuery: String,  
  nodeCount: Integer,  
  relationshipQuery: String,  
  relationshipCount: Integer,  
  createMillis: Integer
```

Table 29. Parameters

Name	Optional	Description
graphName	no	The name under which the graph is stored in the catalog.
nodeQuery	no	Cypher query to project nodes. The query result must contain an <code>id</code> column. Optionally, a <code>labels</code> column can be specified to represent node labels. Additional columns are interpreted as properties.
relationshipQuery	no	Cypher query to project relationships. The query result must contain <code>source</code> and <code>target</code> columns. Optionally, a <code>type</code> column can be specified to represent relationship type. Additional columns are interpreted as properties.
configuration	yes	Additional parameters to configure the Cypher projection.

Table 30. Configuration

Name	Type	Default	Description
readConcurrenty	Integer	4	The number of concurrent threads used for creating the graph.
validateRelationships	Boolean	true	Whether to throw an error if the <code>relationshipQuery</code> returns relationships between nodes not returned by the <code>nodeQuery</code> .
parameters	Map	{}	A map of user-defined query parameters that are passed into the node and relationship queries.

Table 31. Results

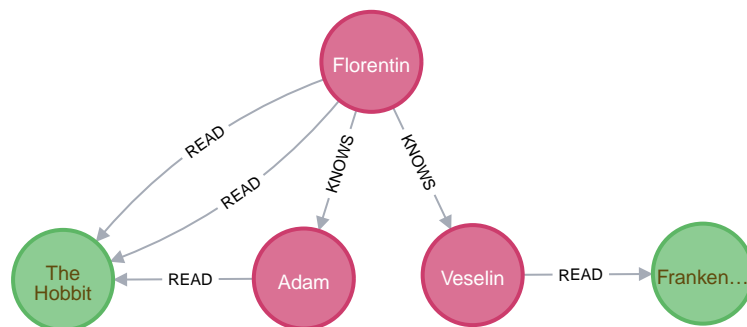
Name	Type	Description
graphName	String	The name under which the graph is stored in the catalog.
nodeQuery	String	The Cypher query used to project the nodes in the graph.
nodeCount	Integer	The number of nodes stored in the projected graph.
relationshipQuery	String	The Cypher query used to project the relationships in the graph.
relationshipCount	Integer	The number of relationships stored in the projected graph.
createMillis	Integer	Milliseconds for creating the graph.



To get information about a stored graph, such as its schema, one can use [gds.graph.list](#).

## Examples

In order to demonstrate the GDS Graph Create capabilities we are going to create a small social network graph in Neo4j. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(florentin:Person { name: 'Florentin', age: 16 }),
(adam:Person { name: 'Adam', age: 18 }),
(veselin:Person { name: 'Veselin', age: 20, ratings: [5.0] }),
(hobbit:Book { name: 'The Hobbit', isbn: 1234, numberOfPages: 310, ratings: [1.0, 2.0, 3.0, 4.5] }),
(frankenstein:Book { name: 'Frankenstein', isbn: 4242, price: 19.99 }),

(florentin)-[:KNOWS { since: 2010 }]->(adam),
(florentin)-[:KNOWS { since: 2018 }]->(veselin),
(florentin)-[:READ { numberOfPages: 4 }]->(hobbit),
(florentin)-[:READ { numberOfPages: 42 }]->(hobbit),
(adam)-[:READ { numberOfPages: 30 }]->(hobbit),
(veselin)-[:READ]->(frankenstein)
```

## Simple graph

A simple graph is a graph with only one node label and relationship type, i.e., a monopartite graph. We are going to start with demonstrating how to load a simple graph by projecting only the **Person** node label and **KNOWS** relationship type.

Project **Person** nodes and **KNOWS** relationships:

```
CALL gds.graph.create.cypher(
'persons',
'MATCH (n:Person) RETURN id(n) AS id',
'MATCH (n:Person)-[r:KNOWS]->(m:Person) RETURN id(n) AS source, id(m) AS target')
YIELD
graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipQuery, relationshipCount AS rels
```

Table 32. Results

graph	nodeQuery	nodes	relationshipQuery	rels
"persons"	"MATCH (n:Person) RETURN id(n) AS id"	3	"MATCH (n:Person)-[r:KNOWS] →(m:Person) RETURN id(n) AS source, id(m) AS target"	2

## Multi-graph

A multi-graph is a graph with multiple node labels and relationship types.

To retain the label and type information when we load multiple node labels and relationship types, we can add a **labels** column to the node query and a **type** column to the relationship query.

Project **Person** and **Book** nodes and **KNOWS** and **READ** relationships:

```
CALL gds.graph.create.cypher(
'personsAndBooks',
'MATCH (n) WHERE n:Person OR n:Book RETURN id(n) AS id, labels(n) AS labels',
'MATCH (n)-[r:KNOWS|READ]->(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type')
YIELD
graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipCount AS rels
```

Table 33. Results

graph	nodeQuery	nodes	rels
"personsAndBooks"	"MATCH (n) WHERE n:Person OR n:Book RETURN id(n) AS id, labels(n) AS labels"	5	6

### Relationship orientation

The native projection supports specifying an orientation per relationship type. The Cypher projection will treat every relationship returned by the relationship query as if it was in **NATURAL** orientation. It is thus not possible to project graphs in **UNDIRECTED** or **REVERSE** orientation when Cypher projections are used.



Some algorithms require that the graph was loaded with **UNDIRECTED** orientation. These algorithms can not be used with a graph created by a Cypher projection.

### Node properties

To load node properties, we add a column to the result of the node query for each property. Thereby, we use the Cypher function `coalesce()` function to specify the default value, if the node does not have the property.

Project **Person** and **Book** nodes and **KNOWS** and **READ** relationships:

```
CALL gds.graph.create.cypher(
  'graphWithProperties',
  'MATCH (n)
  WHERE n:Book OR n:Person
  RETURN
    id(n) AS id,
    labels(n) AS labels,
    coalesce(n.age, 18) AS age,
    coalesce(n.price, 5.0) AS price,
    n.ratings AS ratings',
  'MATCH (n)-[r:KNOWS|READ]->(m) RETURN id(n) AS source, id(m) AS target, type(r) AS type'
)
YIELD
  graphName, nodeCount AS nodes, relationshipCount AS rels
RETURN graphName, nodes, rels
```

Table 34. Results

graphName	nodes	rels
"graphWithProperties"	5	6

The projected **graphWithProperties** graph contains five nodes and six relationships. In a Cypher projection every node from the **nodeQuery** gets the same node properties, which means you can't have label-specific properties. For instance in the example above the **Person** nodes will also get **ratings** and **price** properties, while **Book** nodes get the **age** property.

Further, the **price** property has a default value of **5.0**. Not every book has a price specified in the example graph. In the following we check if the price was correctly projected:

Verify the ratings property of Adam in the projected graph:

```
MATCH (n:Book)
RETURN n.name AS name, gds.util.nodeProperty('graphWithProperties', id(n), 'price') AS price
ORDER BY price
```

Table 35. Results

name	price
"The Hobbit"	5.0
"Frankenstein"	19.99

We can see, that the price was projected with the Hobbit having the default price of 5.0.

## Relationship properties

Analogous to node properties, we can project relationship properties using the `relationshipQuery`.

Project `Person` and `Book` nodes and `READ` relationships with `numberOfPages` property:

```
CALL gds.graph.create.cypher(
  'readWithProperties',
  'MATCH (n) RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:READ]->(m)
  RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages'
)
YIELD
  graphName AS graph, nodeCount AS nodes, relationshipCount AS rels
```

Table 36. Results

graph	nodes	rels
"readWithProperties"	5	4

Next, we will verify that the relationship property `numberOfPages` was correctly loaded.

Stream the relationship property `numberOfPages` from the projected graph:

```
CALL gds.graph.streamRelationshipProperty('readWithProperties', 'numberOfPages')
YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages
RETURN
  gds.util.asNode(sourceNodeId).name AS person,
  gds.util.asNode(targetNodeId).name AS book,
  numberOfPages
ORDER BY person ASC, numberOfPages DESC
```

Table 37. Results

person	book	numberOfPages
"Adam"	"The Hobbit"	30.0
"Florentin"	"The Hobbit"	42.0
"Florentin"	"The Hobbit"	4.0

person	book	numberOfPages
"Veselin"	"Frankenstein"	NaN

We can see, that the `numberOfPages` are loaded. The default property value is `Double.NaN` and can be changed as in the previous example [Node properties](#) by using the Cypher function `coalesce()`.

## Parallel relationships

The Property Graph Model in Neo4j supports parallel relationships, i.e., multiple relationships between two nodes. By default, GDS preserves the parallel relationships. For some algorithms, we want the projected graph to contain at most one relationship between two nodes.

The simplest way to achieve relationship deduplication is to use the `DISTINCT` operator in the relationship query. Alternatively, we can aggregate the parallel relationship by using the `count()` function and store the count as a relationship property.

Project `Person` and `Book` nodes and `COUNT` aggregated `READ` relationships:

```
CALL gds.graph.create.cypher(
  'readCount',
  'MATCH (n) RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:READ]->(m)
   RETURN id(n) AS source, id(m) AS target, type(r) AS type, count(r) AS numberOfReads'
)
YIELD
  graphName AS graph, nodeCount AS nodes, relationshipCount AS rels
```

Table 38. Results

graph	nodes	rels
"readCount"	5	3

Next, we will verify that the `READ` relationships were correctly aggregated.

Stream the relationship property `numberOfReads` of the projected graph:

```
CALL gds.graph.streamRelationshipProperty('readCount', 'numberOfReads')
YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfReads
RETURN
  gds.util.asNode(sourceNodeId).name AS person,
  gds.util.asNode(targetNodeId).name AS book,
  numberOfReads
ORDER BY numberOfReads DESC, person
```

Table 39. Results

person	book	numberOfReads
"Florentin"	"The Hobbit"	2.0
"Adam"	"The Hobbit"	1.0
"Veselin"	"Frankenstein"	1.0

We can see, that the two `READ` relationships between Florentin and the Hobbit result in `2`



numberOfReads.

Parallel relationships with properties

For graphs with relationship properties we can also use other aggregations documented in the [Cypher Manual](#).

Project **Person** and **Book** nodes and aggregated **READ** relationships by summing the **numberOfPages**:

```
CALL gds.graph.create.cypher(
  'readSums',
  'MATCH (n) RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:READ]->(m)
  RETURN id(n) AS source, id(m) AS target, type(r) AS type, sum(r.numberOfPages) AS numberOfPages'
)
YIELD
  graphName AS graph, nodeCount AS nodes, relationshipCount AS rels
```

Table 40. Results

graph	nodes	rels
"readSums"	5	3

Next, we will verify that the relationship property **numberOfPages** were correctly aggregated.

Stream the relationship property **numberOfPages** of the projected graph:

```
CALL gds.graph.streamRelationshipProperty('readSums', 'numberOfPages')
YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages
RETURN
  gds.util.asNode(sourceNodeId).name AS person,
  gds.util.asNode(targetNodeId).name AS book,
  numberOfPages
ORDER BY numberOfPages DESC, person
```

Table 41. Results

person	book	numberOfPages
"Florentin"	"The Hobbit"	46.0
"Adam"	"The Hobbit"	30.0
"Veselin"	"Frankenstein"	0.0

We can see, that the two **READ** relationships between Florentin and the Hobbit sum up to **46** **numberOfPages**.

Projecting filtered Neo4j graphs

Cypher-projections allow us to specify the graph to project in a more fine-grained way. The following examples will demonstrate how we to filter out **READ** relationship if they do not have a **numberOfPages** property.

Project **Person** and **Book** nodes and **READ** relationships where **numberOfPages** is present:

```
CALL gds.graph.create.cypher(
  'existingNumberOfPages',
  'MATCH (n) RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:READ]->(m)
   WHERE r.numberOfPages IS NOT NULL
   RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages'
)
YIELD
  graphName AS graph, nodeCount AS nodes, relationshipCount AS rels
```

Table 42. Results

graph	nodes	rels
"existingNumberOfPages"	5	3

Next, we will verify that the relationship property **numberOfPages** was correctly loaded.

Stream the relationship property **numberOfPages** from the projected graph:

```
CALL gds.graph.streamRelationshipProperty('existingNumberOfPages', 'numberOfPages')
YIELD sourceNodeId, targetNodeId, propertyValue AS numberOfPages
RETURN
  gds.util.asNode(sourceNodeId).name AS person,
  gds.util.asNode(targetNodeId).name AS book,
  numberOfPages
ORDER BY person ASC, numberOfPages DESC
```

Table 43. Results

person	book	numberOfPages
"Adam"	"The Hobbit"	30.0
"Florentin"	"The Hobbit"	42.0
"Florentin"	"The Hobbit"	4.0

If we compare the results to the ones from [Relationship properties](#), we can see that using **IS NOT NULL** is filtering out the relationship from Veselin to the book *Frankenstein*. This functionality is only expressible with [native projections](#) by creating a [subgraph](#).

## Using query parameters

Similar to [Cypher](#), it is also possible to set query parameters. In the following example we supply a list of strings to limit the cities we want to project.

Project **Person** and **Book** nodes and **READ** relationships where **numberOfPages** is greater than 9:

```
CALL gds.graph.create.cypher(
  'existingNumberOfPages',
  'MATCH (n) RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:READ]->(m)
  WHERE r.numberOfPages > $minNumberOfPages
  RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages',
  { parameters: { minNumberOfPages: 9} }
)
YIELD
  graphName AS graph, nodeCount AS nodes, relationshipCount AS rels
```

Table 44. Results

graph	nodes	rels
"existingNumberOfPages"	5	2

### Further usage of parameters

The parameters can also be used to directly pass in a list of nodes or a list of relationships. For example, pre-computing the list of nodes can be useful if the node filter is expensive.

Project **Person** nodes younger than 17 and their name not beginning with V, and **KNOWS** relationships:

```
CALL gds.graph.create.cypher(
  'personSubset',
  'MATCH (n)
  WHERE n.age < 20 AND NOT n.name STARTS WITH "V"
  RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:KNOWS]->(m)
  WHERE (n.age < 20 AND NOT n.name STARTS WITH "V") AND
  (m.age < 20 AND NOT m.name STARTS WITH "V")
  RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages'
)
YIELD
  graphName, nodeCount AS nodes, relationshipCount AS rels
```

Table 45. Results

graphName	nodes	rels
"personSubset"	2	1

By passing the relevant Persons as a parameter, the above query can be transformed into the following:

Project **Person** nodes younger than 20 and their name not beginning with V, and **KNOWS** relationships by using parameters:

```

MATCH (n)
WHERE n.age < 20 AND NOT n.name STARTS WITH "V"
WITH collect(n) AS olderPersons
CALL gds.graph.create.cypher(
  'personSubsetViaParameters',
  'UNWIND $nodes AS n RETURN id(n) AS id, labels(n) AS labels',
  'MATCH (n)-[r:KNOWS]->(m)
   WHERE (n IN $nodes) AND (m IN $nodes)
   RETURN id(n) AS source, id(m) AS target, type(r) AS type, r.numberOfPages AS numberOfPages',
  { parameters: { nodes: olderPersons } }
)
YIELD
  graphName, nodeCount AS nodes, relationshipCount AS rels
RETURN graphName, nodes, rels

```

Table 46. Results

graphName	nodes	rels
"personSubsetViaParameters"	2	1

### 4.1.3. Listing graphs

Information about graphs in the catalog can be retrieved using the `gds.graph.list()` procedure.

#### Syntax

List information about graphs in the catalog:

```

CALL gds.graph.list(
  graphName: String
) YIELD
  graphName: String,
  database: String,
  nodeProjection: Map,
  relationshipProjection: Map,
  nodeQuery: String,
  relationshipQuery: String,
  nodeFilter: String,
  relationshipFilter: String,
  nodeCount: Integer,
  relationshipCount: Integer,
  schema: Map,
  degreeDistribution: Map,
  density: Float,
  creationTime: Datetime,
  modificationTime: Datetime,
  sizeInBytes: Integer,
  memoryUsage: String

```

Table 47. Parameters

Name	Type	Optional	Description
graphName	String	yes	The name under which the graph is stored in the catalog. If no graph name is given, information about all graphs will be listed. If a graph name is given but not found in the catalog, an empty list will be returned.

Table 48. Results

Name	Type	Description
<code>graphName</code>	String	Name of the graph.
<code>database</code>	String	Name of the database in which the graph has been created.
<code>nodeProjection</code>	Map	Node projection used to create the graph. If a Cypher projection was used, this will be a derived node projection.
<code>relationshipProjection</code>	Map	Relationship projection used to create the graph. If a Cypher projection was used, this will be a derived relationship projection.
<code>nodeQuery</code>	String	Node query used to create the graph. If a native projection was used, this will be <code>null</code> .
<code>relationshipQuery</code>	String	Relationship query used to create the graph. If a native projection was used, this will be <code>null</code> .
<code>nodeFilter</code>	String	The node filter used when creating this subgraph from another in-memory graph. If the graph has been created from Neo4j, this will be <code>null</code> .
<code>relationshipFilter</code>	String	The relationship filter used when creating this subgraph from another in-memory graph. If the graph has been created from Neo4j, this will be <code>null</code> .
<code>nodeCount</code>	Integer	Number of nodes in the graph.
<code>relationshipCount</code>	Integer	Number of relationships in the graph.
<code>schema</code>	Map	Node labels, Relationship types and properties contained in the in-memory graph.
<code>degreeDistribution</code>	Map	Histogram of degrees in the graph.
<code>density</code>	Float	Density of the graph.
<code>creationTime</code>	Datetime	Time when the graph was created.
<code>modificationTime</code>	Datetime	Time when the graph was last modified.
<code>sizeInBytes</code>	Integer	Number of bytes used in the Java heap to store the graph.
<code>memoryUsage</code>	String	Human readable description of <code>sizeInBytes</code> .

The information contains basic statistics about the graph, e.g., the node and relationship count. The result field `creationTime` indicates when the graph was created in memory. The result field `modificationTime` indicates when the graph was updated by an algorithm running in `mutate` mode.

The `database` column refers to the name of the database the corresponding graph has been created on. Referring to a named graph in a procedure is only allowed on the database it has been created on.

The `schema` consists of information about the nodes and relationships stored in the graph. For each node label, the schema maps the label to its property keys and their corresponding property types. Similarly, the schema maps the relationship types to their property keys and property types. The property type is either `Integer`, `Float`, `List of Integer` or `List of Float`.

The `degreeDistribution` field can be fairly time-consuming to compute for larger graphs. Its computation is cached per graph, so subsequent listing for the same graph will be fast. To avoid computing the degree

distribution, specify a **YIELD** clause that omits it. Note that not specifying a **YIELD** clause is the same as requesting all possible return fields to be returned.

The **density** is the result of **relationshipCount** divided by the maximal number of relationships for a simple graph with the given **nodeCount**.

## Examples

In order to demonstrate the GDS Graph List capabilities we are going to create a small social network graph in Neo4j.

The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(florentin:Person { name: 'Florentin', age: 16 }),
(adam:Person { name: 'Adam', age: 18 }),
(veselin:Person { name: 'Veselin', age: 20 }),
(florentin)-[:KNOWS { since: 2010 }]->(adam),
(florentin)-[:KNOWS { since: 2018 }]->(veselin)
```

Additionally we will project a few graphs to the graph catalog, for more details see [native projections](#) and [Cypher projections](#).

Project **Person** nodes and **KNOWS** relationships using native projections:

```
CALL gds.graph.create('personsNative', 'Person', 'KNOWS')
```

Project **Person** nodes and **KNOWS** relationships using Cypher projections:

```
CALL gds.graph.create.cypher(
'personsCypher',
'MATCH (n:Person) RETURN id(n) AS id, labels(n) as labels',
'MATCH (n:Person)-[r:KNOWS]->(m:Person) RETURN id(n) AS source, id(m) AS target, type(r) as type')
```

Project **Person** nodes with property **age** and **KNOWS** relationships using Native projections:

```
CALL gds.graph.create(
'personsWithAgeNative',
{
  Person: {properties: 'age'}
},
'KNOWS'
)
```

List basic information about all graphs in the catalog

List basic information about all graphs in the catalog:

```
CALL gds.graph.list()
YIELD graphName, nodeCount, relationshipCount
RETURN graphName, nodeCount, relationshipCount
ORDER BY graphName ASC
```

Table 49. Results

graphName	nodeCount	relationshipCount
"personsCypher"	3	2
"personsNative"	3	2
"personsWithAgeNative"	3	2

List extended information about a specific named graph in the catalog

List extended information about a specific Cypher named graph in the catalog:

```
CALL gds.graph.list('personsCypher')
YIELD graphName, nodeProjection, nodeQuery
```

Table 50. Results

graphName	nodeProjection	nodeQuery
"personsCypher"	null	"MATCH (n:Person) RETURN id(n) AS id, labels(n) as labels"

List extended information about a specific native named graph in the catalog:

```
CALL gds.graph.list('personsNative')
YIELD graphName, nodeProjection, nodeQuery
```

Table 51. Results

graphName	nodeProjection	nodeQuery
"personsNative"	{Person={label=Person, properties={}}}	null

The above examples demonstrate that `nodeQuery` only has value when the graph is projected using Cypher projection while `nodeProjection` is present when we have a native graph. This is also true for `relationshipQuery` and `relationshipProjection` respectively.

Despite different result columns being present for the different projections that we can use the Graph Schemas are the same, which is demonstrated in the example below.

Cypher graph schema:

```
CALL gds.graph.list('personsCypher')
YIELD graphName, schema
```

Table 52. Results

graphName	schema
"personsCypher"	{relationships={KNOWS=[]}, nodes={Person={}}}

Native graph schema:

```
CALL gds.graph.list('personsNative')
YIELD graphName, schema
```

Table 53. Results

graphName	schema
"personsNative"	{relationships={KNOWS=[]}, nodes={Person={}}}

Degree distribution of a specific graph

List information about the degree distribution of a specific graph:

```
CALL gds.graph.list('personsNative')
YIELD graphName, degreeDistribution;
```

Table 54. Results

graphName	degreeDistribution
"personsNative"	{p99=2, min=0, max=2, mean=0.6666666666666666, p90=2, p50=0, p999=2, p95=2, p75=0}

#### 4.1.4. Check if a graph exists

We can check if a graph is stored in the catalog by looking up its name.

#### Syntax

Check if a graph exists in the catalog:

```
CALL gds.graph.exists(graphName: String) YIELD
  graphName: String,
  exists: Boolean
```

Table 55. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.

Table 56. Results

Name	Type	Description
graphName	String	Name of the removed graph.
exists	Boolean	If the graph exists in the graph catalog.

Additionally, to the procedure, we provide a function which directly returns the exists field from the procedure.

Check if a graph exists in the catalog:

```
RETURN gds.graph.exists(graphName: String): Boolean
```



## Examples

In order to demonstrate the GDS Graph Exists capabilities we are going to create a small social network graph in Neo4j and project it into our graph catalog.

The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(florentin:Person { name: 'Florentin', age: 16 }),
(adam:Person { name: 'Adam', age: 18 }),
(veselin:Person { name: 'Veselin', age: 20 }),
(florentin)-[:KNOWS { since: 2010 }]->(adam),
(florentin)-[:KNOWS { since: 2018 }]->(veselin)
```

Project **Person** nodes and **KNOWS** relationships:

```
CALL gds.graph.create('persons', 'Person', 'KNOWS')
```

## Procedure

Check if graphs exist in the catalog:

```
UNWIND ['persons', 'books'] AS graph
CALL gds.graph.exists(graph)
YIELD graphName, exists
RETURN graphName, exists
```

Table 57. Results

graphName	exists
"persons"	true
"books"	false

We can verify the projected **persons** graph exists while a **books** graph does not.

## Function

As an alternative to the procedure, we can also use the corresponding function. Unlike procedures, functions can be inlined in other cypher-statements such as **RETURN** or **WHERE**.

Check if graphs exists in the catalog:

```
RETURN gds.graph.exists('persons') AS personsExists, gds.graph.exists('books') AS booksExists
```

Table 58. Results

personsExists	booksExists
true	false

As before, we can verify the projected **persons** graph exists while a **books** graph does not.

## 4.1.5. Removing graphs

To free up memory, we can remove unused graphs. In order to do so, the `gds.graph.drop` procedure comes in handy.

### Syntax

Remove a graph from the catalog:

```
CALL gds.graph.drop(  
  graphName: String,  
  failIfMissing: Boolean,  
  dbName: String,  
  username: String  
) YIELD  
  graphName: String,  
  database: String,  
  nodeProjection: Map,  
  relationshipProjection: Map,  
  nodeQuery: String,  
  relationshipQuery: String,  
  nodeFilter: String,  
  relationshipFilter: String,  
  nodeCount: Integer,  
  relationshipCount: Integer,  
  schema: Map,  
  density: Float,  
  creationTime: Datetime,  
  modificationTime: Datetime,  
  sizeInBytes: Integer,  
  memoryUsage: String
```

Table 59. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
failIfMissing	Boolean	true	By default, the library will raise an error when trying to remove a non-existing graph. When set to <code>false</code> , the procedure returns empty result.
dbName	String	active database name	Then name of the database that was used to project the graph. When empty, the current database is used.
username	String	active user	The name of the user who projected the graph. Can only be used by GDS administrator.

Table 60. Results

Name	Type	Description
graphName	String	Name of the removed graph.
database	String	Name of the database in which the graph has been created.
nodeProjection	Map	Node projection used to create the graph. If a Cypher projection was used, this will be a derived node projection.
relationshipProjection	Map	Relationship projection used to create the graph. If a Cypher projection was used, this will be a derived relationship projection.
nodeQuery	String	Node query used to create the graph. If a native projection was used, this will be <code>null</code> .

Name	Type	Description
<code>relationshipQuery</code>	String	Relationship query used to create the graph. If a native projection was used, this will be <code>null</code> .
<code>nodeFilter</code>	String	The node filter used when creating this subgraph from another in-memory graph.
<code>relationshipFilter</code>	String	The relationship filter used when creating this subgraph from another in-memory graph.
<code>nodeCount</code>	Integer	Number of nodes in the graph.
<code>relationshipCount</code>	Integer	Number of relationships in the graph.
<code>schema</code>	Map	Node labels, Relationship types and properties contained in the in-memory graph.
<code>density</code>	Float	Density of the graph.
<code>creationTime</code>	Datetime	Time when the graph was created.
<code>modificationTime</code>	Datetime	Time when the graph was last modified.
<code>sizeInBytes</code>	Integer	Number of bytes used in the Java heap to store the graph.
<code>memoryUsage</code>	String	Human readable description of <code>sizeInBytes</code> .

## Examples

In this section we are going to demonstrate the usage of `gds.graph.drop`. All the graph names used in these examples are fictive and should be replaced with real values.

### Basic usage

Remove a graph from the catalog:

```
CALL gds.graph.drop('my-store-graph') YIELD graphName;
```

If we run the example above twice, the second time it will raise an error. If we want the procedure to fail silently on non-existing graphs, we can set a boolean flag as the second parameter to `false`. This will yield an empty result for non-existing graphs.

Try removing a graph from the catalog:

```
CALL gds.graph.drop('my-fictive-graph', false) YIELD graphName;
```

Multi-database support [Enterprise edition](#)

If we want to drop a graph created on another database, we can set the database name as the third parameter.

Try removing a graph from the catalog:

```
CALL gds.graph.drop('my-fictive-graph', true, 'my-other-db') YIELD graphName;
```

## Multi-user support

If we are a GDS administrator and want to drop a graph that belongs to another user we can set the username as the fourth parameter to the procedure. This is useful if there are multiple users with graphs of the same name.

Remove a graph from a specific user's graph catalog:

```
CALL gds.graph.drop('my-fictive-graph', true, '', 'another-user') YIELD graphName;
```

See [Administration](#) for more details on this.

## 4.1.6. Creating a subgraph Beta

In GDS, algorithms can be executed on a named graph that has been filtered based on its [node labels](#) and [relationship types](#). However, that filtered graph only exists during the execution of the algorithm and it is not possible to filter on property values. If a filtered graph needs to be used multiple times, one can use the subgraph catalog procedure to create a new graph in the graph catalog.

The filter predicates in the subgraph procedure can take labels, relationship types as well as node and relationship properties into account. The new graph can be used in the same way as any other in-memory graph in the catalog. Creating subgraphs of subgraphs is also possible.

## Syntax

A new graph can be created by using the `gds.beta.graph.create.subgraph()` procedure:

```
CALL gds.beta.graph.create.subgraph(  
  graphName: String,  
  fromGraphName: String,  
  nodeFilter: String,  
  relationshipFilter: String,  
  configuration: Map  
) YIELD  
  graphName: String,  
  fromGraphName: String,  
  nodeFilter: String,  
  relationshipFilter: String,  
  nodeCount: Integer,  
  relationshipCount: Integer,  
  createMillis: Integer
```

Table 61. Parameters

Name	Type	Description
graphName	String	The name of the new graph that is stored in the graph catalog.
fromGraphName	String	The name of the original graph in the graph catalog.

Name	Type	Description
nodeFilter	String	A Cypher predicate for filtering nodes in the input graph. * can be used to allow all nodes.
relationshipFilter	String	A Cypher predicate for filtering relationships in the input graph. * can be used to allow all relationships.
configuration	Map	Additional parameters to configure subgraph creation.

Table 62. Subgraph specific configuration

Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for filtering the graph.

Table 63. Results

Name	Type	Description
graphName	String	The name of the new graph that is stored in the graph catalog.
fromGraphName	String	The name of the original graph in the graph catalog.
nodeFilter	String	Filter predicate for nodes.
relationshipFilter	String	Filter predicate for relationships.
nodeCount	Integer	Number of nodes in the subgraph.
relationshipCount	Integer	Number of relationships in the subgraph.
createMillis	Integer	Milliseconds for creating the subgraph.

The `nodeFilter` and `relationshipFilter` configuration keys can be used to express filter predicates. Filter predicates are `Cypher` predicates bound to a single entity. An entity is either a node or a relationship. The filter predicate always needs to evaluate to `true` or `false`. A node is contained in the subgraph if the node filter evaluates to `true`. A relationship is contained in the subgraph if the relationship filter evaluates to `true` and its source and target nodes are contained in the subgraph.

A predicate is a combination of expressions. The simplest form of expression is a literal. GDS currently supports the following literals:

- float literals, e.g., `13.37`
- integer literals, e.g., `42`
- boolean literals, i.e., `TRUE` and `FALSE`

Property, label and relationship type expressions are bound to an entity. The node entity is always identified by the variable `n`, the relationship entity is identified by `r`. Using the variable, we can refer to:

- node label expression, e.g., `n:Person`
- relationship type expression, e.g., `r:KNOWS`
- node property expression, e.g., `n.age`
- relationship property expression, e.g., `r.since`

Boolean predicates combine two expressions and return either `true` or `false`. GDS supports the following boolean predicates:

- greater/lower than, such as `n.age > 42` or `r.since < 1984`
- greater/lower than or equal, such as `n.age >= 42` or `r.since <= 1984`
- equality, such as `n.age = 23` or `r.since = 2020`
- logical operators, such as
  - `n.age > 23 AND n.age < 42`
  - `n.age = 23 OR n.age = 42`
  - `n.age = 23 XOR n.age = 42`
  - `n.age IS NOT 23`

Variable names that can be used within predicates are not arbitrary. A node predicate must refer to variable `n`. A relationship predicate must refer to variable `r`.

## Examples

In order to demonstrate the GDS create subgraph capabilities we are going to create a small social graph in Neo4j.

The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(p0:Person { age: 16 }),
(p1:Person { age: 18 }),
(p2:Person { age: 20 }),
(b0:Book { isbn: 1234 }),
(b1:Book { isbn: 4242 }),
(p0)-[:KNOWS { since: 2010 }]->(p1),
(p0)-[:KNOWS { since: 2018 }]->(p2),
(p0)-[:READS]->(b0),
(p1)-[:READS]->(b0),
(p2)-[:READS]->(b1)
```

Project the social network graph:

```
CALL gds.graph.create(
  'social-graph',
  {
    Person: { properties: 'age' },    ①
    Book: {}                          ②
  },
  {
    KNOWS: { properties: 'since' },  ③
    READS: {}                          ④
  }
)
YIELD graphName, nodeCount, relationshipCount, createMillis
```

- ① Project `Person` nodes with their `age` property.
- ② Project `Book` nodes without any of their properties.
- ③ Project `KNOWS` relationships with their `since` property.
- ④ Project `READS` relationships without any of their properties.

## Node filtering

Create a new graph containing only users of a certain age group:

```
CALL gds.beta.graph.create.subgraph(  
  'teenagers',  
  'social-graph',  
  'n.age > 13 AND n.age <= 18',  
  '*'  
)  
YIELD graphName, fromGraphName, nodeCount, relationshipCount
```

Table 64. Results

graphName	fromGraphName	nodeCount	relationshipCount
"teenagers"	"social-graph"	2	1

## Node and relationship filtering

Create a new graph containing only users of a certain age group that know each other since a given point a time:

```
CALL gds.beta.graph.create.subgraph(  
  'teenagers',  
  'social-graph',  
  'n.age > 13 AND n.age <= 18',  
  'r.since >= 2012.0'  
)  
YIELD graphName, fromGraphName, nodeCount, relationshipCount
```

Table 65. Results

graphName	fromGraphName	nodeCount	relationshipCount
"teenagers"	"social-graph"	2	0

## Bipartite subgraph

Create a new bipartite graph between books and users connected by the **READS** relationship type:

```
CALL gds.beta.graph.create.subgraph(  
  'teenagers-books',  
  'social-graph',  
  'n:Book OR n:Person',  
  'r:READS'  
)  
YIELD graphName, fromGraphName, nodeCount, relationshipCount
```

Table 66. Results

graphName	fromGraphName	nodeCount	relationshipCount
"teenagers-books"	"social-graph"	5	3

## Bipartite graph node filtering

The previous example can be extended with an additional filter applied only to persons:

```
CALL gds.beta.graph.create.subgraph(  
  'teenagers-books',  
  'social-graph',  
  'n:Book OR (n:Person AND n.age > 18)',  
  'r:READS'  
)  
YIELD graphName, fromGraphName, nodeCount, relationshipCount
```

Table 67. Results

graphName	fromGraphName	nodeCount	relationshipCount
"teenagers-books"	"social-graph"	3	1

### 4.1.7. Node operations

The graphs in the Neo4j Graph Data Science Library support properties for nodes. We provide multiple operations to work with the stored node-properties in projected graphs. Node properties are either created during the graph creation or when using the `mutate` mode of our graph algorithms.

To inspect stored values, the `gds.graph.streamNodeProperties` procedure can be used. This is useful if we ran multiple algorithms in `mutate` mode and want to retrieve some or all of the results.

To persist the values in a Neo4j database, we can use `gds.graph.writeNodeProperties`. Similar to streaming node properties, it is also possible to write those back to Neo4j. This is similar to what an algorithm `write` execution mode does, but allows more fine-grained control over the operations.

We can also remove node properties from a named graph in the catalog. This is useful to free up main memory or to remove accidentally created node properties.

#### Syntax



```
CALL gds.graph.streamNodeProperty(
  graphName: String,
  nodeProperties: String,
  nodeLabels: String or List of Strings,
  configuration: Map
)
YIELD
  nodeId: Integer,
  propertyValue: Integer or Float or List of Integer or List of Float
```

Table 68. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
nodeProperties	String	no	The node property in the graph to stream.
nodeLabels	String or List of Strings	yes	The node labels to stream the node properties for graph.
configuration	Map	yes	Additional parameters to configure streamNodeProperties.

Table 69. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads. Note, this procedure is always running single-threaded.

Table 70. Results

Name	Type	Description
nodeId	Integer	The id of the node.
propertyValue	<ul style="list-style-type: none"> <li>• Integer</li> <li>• Float</li> <li>• List of Integer</li> <li>• List of Float</li> </ul>	The stored property value.

```

CALL gds.graph.streamNodeProperties(
  graphName: String,
  nodeProperties: String or List of Strings,
  nodeLabels: String or List of Strings,
  configuration: Map
)
YIELD
  nodeId: Integer,
  nodeProperty: String,
  propertyValue: Integer or Float or List of Integer or List of Float

```

Table 71. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
nodeProperties	String or List of Strings	no	The node properties in the graph to stream.
nodeLabels	String or List of Strings	yes	The node labels to stream the node properties for graph.
configuration	Map	yes	Additional parameters to configure streamNodeProperties.

Table 72. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads. Note, this procedure is always running single-threaded.

Table 73. Results

Name	Type	Description
nodeId	Integer	The id of the node.
nodeProperty	String	The name of the node property.
propertyValue	<ul style="list-style-type: none"> <li>• Integer</li> <li>• Float</li> <li>• List of Integer</li> <li>• List of Float</li> </ul>	The stored property value.

```

CALL gds.graph.writeNodeProperties(
  graphName: String,
  nodeProperties: String or List of Strings,
  nodeLabels: String or List of Strings,
  configuration: Map
)
YIELD
  writeMillis: Integer,
  propertiesWritten: Integer,
  graphName: String,
  nodeProperties: String or List of String

```

Table 74. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
nodeProperties	String or List of Strings	no	The node properties in the graph to write back.
nodeLabels	String or List of Strings	yes	The node labels to write back their node properties.
configuration	Map	yes	Additional parameters to configure writeNodeProperties.

Table 75. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads used for running the procedure. Also provides the default value for <code>writeConcurrency</code>
writeConcurrency	Integer	'concurrency'	The number of concurrent threads used for writing the node properties.

Table 76. Results

Name	Type	Description
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
propertiesWritten	Integer	Number of properties written.
graphName	String	The name of a graph stored in the catalog.
nodeProperties	String or List of String	The written node properties.

```
CALL gds.graph.removeNodeProperties(
  graphName: String,
  nodeProperties: String or List of Strings,
  nodeLabels: String or List of Strings,
  configuration: Map
)
YIELD
  propertiesRemoved: Integer,
  graphName: String,
  nodeProperties: String or List of String
```

Table 77. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
nodeProperties	String or List of Strings	no	The node properties in the graph to remove.
nodeLabels	String or List of Strings	yes	The node labels to remove the node properties from.
configuration	Map	yes	Additional parameters to configure removeNodeProperties.

Table 78. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads. Note, this procedure is always running single-threaded.

Table 79. Results

Name	Type	Description
propertiesRemoved	Integer	Number of properties removed.
graphName	String	The name of a graph stored in the catalog.
nodeProperties	String or List of String	The removed node properties.

## Examples

In order to demonstrate the GDS capabilities over node properties, we are going to create a small social network graph in Neo4j and project it into our graph catalog.

The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(florentin:Person { name: 'Florentin', age: 16 }),
(adam:Person { name: 'Adam', age: 18 }),
(veselin:Person { name: 'Veselin', age: 20 }),
(hobbit:Book { name: 'The Hobbit', numberOfPages: 310 }),
(florentin)-[:KNOWS { since: 2010 }]->(adam),
(florentin)-[:KNOWS { since: 2018 }]->(veselin),
(adam)-[:READ]->(hobbit)
```

Project the small social network graph:

```
CALL gds.graph.create(
  'socialGraph',
  {
    Person: {properties: "age"},
    Book: {}
  },
  ['KNOWS', 'READ']
)
```

Compute the Degree Centrality in our social graph:

```
CALL gds.degree.mutate('socialGraph', {mutateProperty: 'score'})
```

## Stream

We can stream node properties stored in a named in-memory graph back to the user. This is useful if we ran multiple algorithms in `mutate` mode and want to retrieve some or all of the results. This is similar to what an algorithm `stream` execution mode does, but allows more fine-grained control over the operations.

### Single property

In the following, we stream the previously computed scores `score`.

Stream the `score` node property:

```
CALL gds.graph.streamNodeProperty('socialGraph', 'score')
YIELD nodeId, propertyValue
RETURN gds.util.asNode(nodeId).name AS name, propertyValue AS score
ORDER BY score DESC
```

Table 80. Results

name	score
"Florentin"	2.0
"Adam"	1.0
"Veselin"	0.0
"The Hobbit"	0.0



The above example requires all given properties to be present on at least one node projection, and the properties will be streamed for all such projections.

### NodeLabels

The procedure can be configured to stream just the properties for specific node labels.

Stream the **score** property for **Person** nodes:

```
CALL gds.graph.streamNodeProperty('socialGraph', 'score', ['Person'])
YIELD nodeId, propertyValue
RETURN gds.util.asNode(nodeId).name AS name, propertyValue AS score
ORDER BY score DESC
```

Table 81. Results

name	score
"Florentin"	2.0
"Adam"	1.0
"Veselin"	0.0

It is required, that all specified node labels have the node property.

### Multiple Properties

We can also stream several properties at once.

Stream multiple node properties:

```
CALL gds.graph.streamNodeProperties('socialGraph', ['score', 'age'])
YIELD nodeId, nodeProperty, propertyValue
RETURN gds.util.asNode(nodeId).name AS name, nodeProperty, propertyValue
ORDER BY name, nodeProperty
```

Table 82. Results

name	nodeProperty	propertyValue
"Adam"	"age"	18
"Adam"	"score"	1.0
"Florentin"	"age"	16
"Florentin"	"score"	2.0
"Veselin"	"age"	20
"Veselin"	"score"	0.0



When streaming multiple node properties, the name of each property is included in the result. This adds with some overhead, as each property name must be repeated for each node in the result, but is necessary in order to distinguish properties.

## Write

To write the 'score' property for all node labels in the social graph, we use the following query:

Write the **score** property back to Neo4j:

```
CALL gds.graph.writeNodeProperties('socialGraph', ['score'])
YIELD propertiesWritten
```

Table 83. Results

propertiesWritten
4

The above example requires the **score** property to be present on at least one projected node label, and the properties will be written for all such labels.

## NodeLabels

The procedure can be configured to write just the properties for some specific node labels. In the following example, we will only write back the scores of the **Person** nodes.

Write node properties of a specific projected node label to Neo4j:

```
CALL gds.graph.writeNodeProperties('socialGraph', ['score'], ['Person'])
YIELD propertiesWritten
```

Table 84. Results

propertiesWritten
3



If the **nodeLabels** parameter is specified, it is required that *all* given node labels have *all* of the given properties.

## Remove

Remove the **score** property from all projected nodes in the **socialGraph**:

```
CALL gds.graph.removeNodeProperties('socialGraph', ['score'])
YIELD propertiesRemoved
```

Table 85. Results

propertiesRemoved

4



The above example requires all given properties to be present on at least one projected node label.

## NodeLabels

Consider we compute the Degree Centrality only for a subset of the graph.

Compute the Degree Centrality for only the **Book** nodes in our social graph:

```
CALL gds.degree.mutate('socialGraph', {nodeLabels: ['Book'], mutateProperty: 'degree'})
```

The procedure can be configured to remove just the properties for s. In the following example, we will only remove the scores from the **Book** nodes.

Remove the **degree** property from the projected **Book** nodes:

```
CALL gds.graph.removeNodeProperties('socialGraph', ['degree'], ['Book'])  
YIELD propertiesRemoved
```

Table 86. Results

propertiesRemoved

1



If the **nodeLabels** parameter is specified, it is required that *all* given node labels have *all* of the given properties.

## Utility functions

Utility functions allow accessing specific nodes of in-memory graphs directly from a Cypher query.

Table 87. Catalog Functions

Name	Description
<code>gds.util.nodeProperty</code>	Allows accessing a node property stored in a named graph.

## Syntax

Name	Description
<code>gds.util.nodeProperty(graphName: STRING, nodeId: INTEGER, propertyKey: STRING, nodeLabel: STRING?)</code>	Named graph in the catalog, Neo4j node id, node property key and optional node label present in the named-graph.

If a node label is given, the property value for the corresponding projection and the given node is returned.



If no label or '\*' is given, the property value is retrieved and returned from an arbitrary projection that contains the given propertyKey. If the property value is missing for the given node, null is returned.

## Examples

We use the `socialGraph` with the property `score` introduced above.

Access a property node property for Florentin:

```
MATCH (florentin:Person {name: 'Florentin'})
RETURN
  florentin.name AS name,
  gds.util.nodeProperty('socialGraph', id(florentin), 'score') AS score
```

Table 88. Results

name	score
"Florentin"	2.0

We can also specifically return the `score` property from the `Person` projection in case other projections also have a `score` property as follows.

Access a property node property from Person for Florentin:

```
MATCH (florentin:Person {name: 'Florentin'})
RETURN
  florentin.name AS name,
  gds.util.nodeProperty('socialGraph', id(florentin), 'score', 'Person') AS score
```

Table 89. Results

name	score
"Florentin"	2.0

## 4.1.8. Relationship operations

The graphs in the Neo4j Graph Data Science Library support properties for relationships. We provide multiple operations to work with the stored relationship-properties in projected graphs. Relationship properties are either created during the graph creation or when using the `mutate` mode of our graph algorithms.

To inspect stored relationship property values, the `streamRelationshipProperties` procedure can be used. This is useful if we ran multiple algorithms in `mutate` mode and want to retrieve some or all of the results.

To persist relationship types in a Neo4j database, we can use `gds.graph.writeRelationship`. Similar to streaming relationship properties, it is also possible to write back to Neo4j. This is similar to what an algorithm `write` execution mode does, but allows more fine-grained control over the operations. By default, no relationship properties will be written. To write relationship properties, these have to be explicitly specified.

We can also remove relationships from a named graph in the catalog. This is useful to free up main

memory or to remove accidentally created relationship types.

Syntax

```
CALL gds.graph.streamRelationshipProperty(
  graphName: String,
  relationshipProperties: List of String,
  relationshipTypes: List of Strings,
  configuration: Map
)
YIELD
  sourceNodeId: Integer,
  targetNodeId: Integer,
  relationshipType: String,
  propertyValue: Integer or Float
```

Table 90. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
relationshipProperties	List of String	no	The relationship properties in the graph to stream.
relationshipTypes	List of Strings	yes	The relationship types to stream the relationship properties for graph.
configuration	Map	yes	Additional parameters to configure streamNodeProperties.

Table 91. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads. Note, this procedure is always running single-threaded.

Table 92. Results

Name	Type	Description
sourceNodeId	Integer	The id of the source node for the relationship.
targetNodeId	Integer	The id of the target node for the relationship.
relationshipType	Integer	The type of the relationship.
propertyValue	<ul style="list-style-type: none"> <li>• Integer</li> <li>• Float</li> </ul>	The stored property value.

```

CALL gds.graph.streamRelationshipProperties(
  graphName: String,
  relationshipProperties: List of String,
  relationshipTypes: List of Strings,
  configuration: Map
)
YIELD
  sourceNodeId: Integer,
  targetNodeId: Integer,
  relationshipType: String,
  relationshipProperty: String,
  propertyValue: Integer or Float

```

Table 93. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
relationshipProperties	List of String	no	The relationship properties in the graph to stream.
relationshipTypes	List of Strings	yes	The relationship types to stream the relationship properties for graph.
configuration	Map	yes	Additional parameters to configure streamNodeProperties.

Table 94. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads. Note, this procedure is always running single-threaded.

Table 95. Results

Name	Type	Description
sourceNodeId	Integer	The id of the source node for the relationship.
targetNodeId	Integer	The id of the target node for the relationship.
relationshipType	Integer	The type of the relationship.
relationshipProperty	Integer	The name of the relationship property.
propertyValue	<ul style="list-style-type: none"> <li>• Integer</li> <li>• Float</li> </ul>	The stored property value.

```
CALL gds.graph.writeRelationship(
  graphName: String,
  relationshipType: String,
  relationshipProperty: String,
  configuration: Map
)
YIELD
  writeMillis: Integer,
  graphName: String,
  relationshipType: String,
  relationshipsWritten: Integer,
  relationshipProperty: String,
  propertiesWritten: Integer
```

Table 96. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
relationshipType	String	no	The relationship type in the graph to write back.
relationshipProperty	String	yes	The relationship property to write back.
configuration	Map	yes	Additional parameters to configure writeRelationship.

Table 97. Configuration

Name	Type	Default	Description
concurrency	Integer	4	The number of concurrent threads used for running the procedure. Also provides the default value for <code>writeConcurrency</code> . Note, this procedure is always running single-threaded.
writeConcurrency	Integer	'concurrency'	The number of concurrent threads used for writing the relationship properties. Note, this procedure is always running single-threaded.

Table 98. Results

Name	Type	Description
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
graphName	String	The name of a graph stored in the catalog.
relationshipType	String	The type of the relationship that was written.
relationshipsWritten	Integer	Number relationships written.
relationshipProperty	String	The name of the relationship property that was written.
propertiesWritten	Integer	Number relationships properties written.

```

CALL gds.graph.deleteRelationships(
  graphName: String,
  relationshipType: String
)
YIELD
  graphName: String,
  relationshipType: String,
  deletedRelationships: Integer,
  deletedProperties: Map

```

Table 99. Parameters

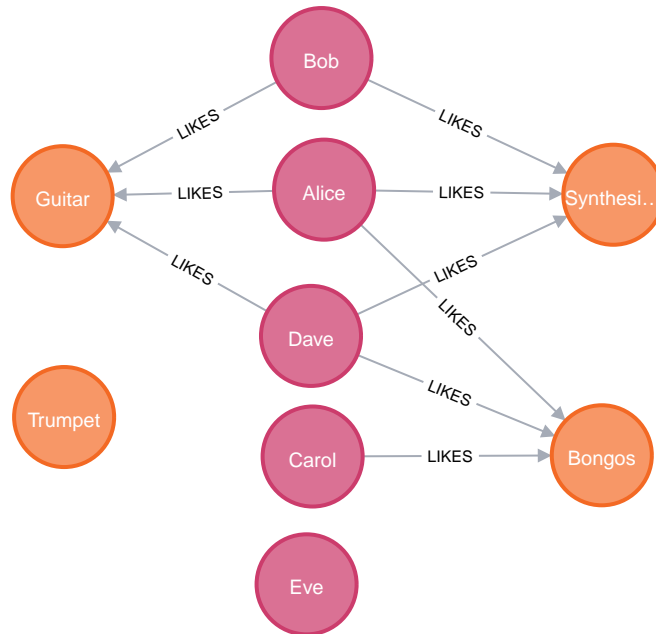
Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
relationshipType	String	no	The relationship type in the graph to remove.

Table 100. Results

Name	Type	Description
graphName	String	The name of a graph stored in the catalog.
relationshipType	String	The type of the removed relationships.
deletedRelationships	Integer	Number of removed relationships from the in-memory graph.
deletedProperties	Integer	Map where the key is the name of the relationship property, and the value is the number of removed properties under that name.

## Examples

In order to demonstrate the GDS capabilities over node properties, we are going to create a small graph in Neo4j and project it into our graph catalog.



The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
  (alice:Person {name: 'Alice'}),
  (bob:Person {name: 'Bob'}),
  (carol:Person {name: 'Carol'}),
  (dave:Person {name: 'Dave'}),
  (eve:Person {name: 'Eve'}),
  (guitar:Instrument {name: 'Guitar'}),
  (synth:Instrument {name: 'Synthesizer'}),
  (bongos:Instrument {name: 'Bongos'}),
  (trumpet:Instrument {name: 'Trumpet'}),

  (alice)-[:LIKES { score: 5 }]->(guitar),
  (alice)-[:LIKES { score: 4 }]->(synth),
  (alice)-[:LIKES { score: 3, strength: 0.5 }]->(bongos),
  (bob)-[:LIKES { score: 4 }]->(guitar),
  (bob)-[:LIKES { score: 5 }]->(synth),
  (carol)-[:LIKES { score: 2 }]->(bongos),
  (dave)-[:LIKES { score: 3 }]->(guitar),
  (dave)-[:LIKES { score: 1 }]->(synth),
  (dave)-[:LIKES { score: 5 }]->(bongos)

```

Project the graph:

```

CALL gds.graph.create(
  'personsAndInstruments',
  ['Person', 'Instrument'],           ①
  {
    LIKES: {
      type: 'LIKES',                  ②
      properties: {
        strength: {                   ③
          property: 'strength',
          defaultValue: 1.0
        },
        score: {                       ④
          property: 'score'
        }
      }
    }
  }
)

```

- ① Project node labels **Person** and **Instrument**.
- ② Project relationship type **LIKES**.

- ③ Project property `strength` of relationship type `LIKES` setting a default value of `1.0` because not all relationships have that property.
- ④ Project property `score` of relationship type `LIKES`.

Compute the Node Similarity in our graph:

```
CALL gds.nodeSimilarity.mutate('personsAndInstruments', {
  mutateRelationshipType: 'SIMILAR',
  mutateProperty: 'score'
})
```

- ① Run NodeSimilarity in `mutate` mode on `personsAndInstruments` projected graph.
- ② The algorithm will create relationships of type `SIMILAR` in the projected graph.
- ③ The algorithm will create relationship property `score` for each created relationship.

## Stream

### Single property

The most basic case for streaming relationship information from a named graph is a single property. In the example below we stream the relationship property `score`.

Stream a single relationship property:

```
CALL gds.graph.streamRelationshipProperty(
  'personsAndInstruments',
  'score'
)
YIELD
  sourceNodeId, targetNodeId, relationshipType, propertyValue
RETURN
  gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target,
  relationshipType, propertyValue
ORDER BY source ASC, target ASC
```

- ① The name of the projected graph.
- ② The property we want to stream out.

Table 101. Results

source	target	relationshipType	propertyValue
"Alice"	"Bob"	"SIMILAR"	0.6666666666666666
"Alice"	"Bongos"	"LIKES"	3.0
"Alice"	"Carol"	"SIMILAR"	0.3333333333333333
"Alice"	"Dave"	"SIMILAR"	1.0
"Alice"	"Guitar"	"LIKES"	5.0
"Alice"	"Synthesizer"	"LIKES"	4.0
"Bob"	"Alice"	"SIMILAR"	0.6666666666666666



source	target	relationshipType	propertyValue
"Bob"	"Dave"	"SIMILAR"	0.6666666666666666
"Bob"	"Guitar"	"LIKES"	4.0
"Bob"	"Synthesizer"	"LIKES"	5.0
"Carol"	"Alice"	"SIMILAR"	0.3333333333333333
"Carol"	"Bongos"	"LIKES"	2.0
"Carol"	"Dave"	"SIMILAR"	0.3333333333333333
"Dave"	"Alice"	"SIMILAR"	1.0
"Dave"	"Bob"	"SIMILAR"	0.6666666666666666
"Dave"	"Bongos"	"LIKES"	5.0
"Dave"	"Carol"	"SIMILAR"	0.3333333333333333
"Dave"	"Guitar"	"LIKES"	3.0
"Dave"	"Synthesizer"	"LIKES"	1.0

As we can see from the results, we get two relationship types (**SIMILAR** and **LIKES**) that have the **score** relationship property. We can further on filter the relationship types we want to stream, this is demonstrated in the next example.

Stream a single relationship property for specific relationship type:

```
CALL gds.graph.streamRelationshipProperty(
  'personsAndInstruments',      ①
  'score',                      ②
  ['SIMILAR']                  ③
)
YIELD
  sourceNodeId, targetNodeId, relationshipType, propertyValue
RETURN
  gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target,
  relationshipType, propertyValue
ORDER BY source ASC, target ASC
```

- ① The name of the projected graph.
- ② The property we want to stream out.
- ③ List of relationship types we want to stream the property from, only use the ones we need.

Table 102. Results

source	target	relationshipType	propertyValue
"Alice"	"Bob"	"SIMILAR"	0.6666666666666666
"Alice"	"Carol"	"SIMILAR"	0.3333333333333333
"Alice"	"Dave"	"SIMILAR"	1.0
"Bob"	"Alice"	"SIMILAR"	0.6666666666666666
"Bob"	"Dave"	"SIMILAR"	0.6666666666666666

source	target	relationshipType	propertyValue
"Carol"	"Alice"	"SIMILAR"	0.3333333333333333
"Carol"	"Dave"	"SIMILAR"	0.3333333333333333
"Dave"	"Alice"	"SIMILAR"	1.0
"Dave"	"Bob"	"SIMILAR"	0.6666666666666666
"Dave"	"Carol"	"SIMILAR"	0.3333333333333333

### Multiple properties

It is also possible to stream multiple relationship properties.

Stream multiple relationship properties:

```
CALL gds.graph.streamRelationshipProperties(
  'personsAndInstruments',      ①
  ['score', 'strength'],        ②
  ['LIKES']                      ③
)
YIELD
  sourceNodeId, targetNodeId, relationshipType, relationshipProperty, propertyValue
RETURN
  gds.util.asNode(sourceNodeId).name as source, gds.util.asNode(targetNodeId).name as target,
  relationshipType, relationshipProperty, propertyValue
ORDER BY source ASC, target ASC
```

- ① The name of the projected graph.
- ② List of properties we want to stream out, allows us to stream more than one property.
- ③ List of relationship types we want to stream the property from, only use the ones we need.

Table 103. Results

source	target	relationshipType	relationshipProperty	propertyValue
"Alice"	"Bongos"	"LIKES"	"score"	3.0
"Alice"	"Bongos"	"LIKES"	"strength"	0.5
"Alice"	"Guitar"	"LIKES"	"score"	5.0
"Alice"	"Guitar"	"LIKES"	"strength"	1.0
"Alice"	"Synthesizer"	"LIKES"	"score"	4.0
"Alice"	"Synthesizer"	"LIKES"	"strength"	1.0
"Bob"	"Guitar"	"LIKES"	"score"	4.0
"Bob"	"Guitar"	"LIKES"	"strength"	1.0
"Bob"	"Synthesizer"	"LIKES"	"score"	5.0
"Bob"	"Synthesizer"	"LIKES"	"strength"	1.0
"Carol"	"Bongos"	"LIKES"	"score"	2.0
"Carol"	"Bongos"	"LIKES"	"strength"	1.0

source	target	relationshipType	relationshipProperty	propertyValue
"Dave"	"Bongos"	"LIKES"	"score"	5.0
"Dave"	"Bongos"	"LIKES"	"strength"	1.0
"Dave"	"Guitar"	"LIKES"	"score"	3.0
"Dave"	"Guitar"	"LIKES"	"strength"	1.0
"Dave"	"Synthesizer"	"LIKES"	"score"	1.0
"Dave"	"Synthesizer"	"LIKES"	"strength"	1.0

### Multiple relationship types

Similar to the multiple relationship properties we can stream properties for multiple relationship types.

Stream relationship properties of a multiple relationship projections:

```
CALL gds.graph.streamRelationshipProperties(
  'personsAndInstruments',           ①
  ['score'],                          ②
  ['LIKES', 'SIMILAR']              ③
)
YIELD
  sourceNodeId, targetNodeId, relationshipType, relationshipProperty, propertyValue
RETURN
  gds.util.asNode(sourceNodeId).name as source,    ④
  gds.util.asNode(targetNodeId).name as target,    ⑤
  relationshipType,
  relationshipProperty,
  propertyValue
ORDER BY source ASC, target ASC
```

- ① The name of the projected graph.
- ② List of properties we want to stream out, allows us to stream more than one property.
- ③ List of relationship types we want to stream the property from, only use the ones we need.
- ④ Return the **name** of the source node.
- ⑤ Return the **name** of the target node.

Table 104. Results

source	target	relationshipType	relationshipProperty	propertyValue
"Alice"	"Bob"	"SIMILAR"	"score"	0.6666666666666666
"Alice"	"Bongos"	"LIKES"	"score"	3.0
"Alice"	"Carol"	"SIMILAR"	"score"	0.3333333333333333
"Alice"	"Dave"	"SIMILAR"	"score"	1.0
"Alice"	"Guitar"	"LIKES"	"score"	5.0
"Alice"	"Synthesizer"	"LIKES"	"score"	4.0
"Bob"	"Alice"	"SIMILAR"	"score"	0.6666666666666666

source	target	relationshipType	relationshipProperty	propertyValue
"Bob"	"Dave"	"SIMILAR"	"score"	0.6666666666666666
"Bob"	"Guitar"	"LIKES"	"score"	4.0
"Bob"	"Synthesizer"	"LIKES"	"score"	5.0
"Carol"	"Alice"	"SIMILAR"	"score"	0.3333333333333333
"Carol"	"Bongos"	"LIKES"	"score"	2.0
"Carol"	"Dave"	"SIMILAR"	"score"	0.3333333333333333
"Dave"	"Alice"	"SIMILAR"	"score"	1.0
"Dave"	"Bob"	"SIMILAR"	"score"	0.6666666666666666
"Dave"	"Bongos"	"LIKES"	"score"	5.0
"Dave"	"Carol"	"SIMILAR"	"score"	0.3333333333333333
"Dave"	"Guitar"	"LIKES"	"score"	3.0
"Dave"	"Synthesizer"	"LIKES"	"score"	1.0



The properties we want to stream must exist for each specified relationship type.

Write

We can write relationships stored in a named in-memory graph back to Neo4j. This can be used to write algorithm results (for example from [Node Similarity](#)) or relationships that have been aggregated during graph creation.

The relationships to write are specified by a relationship type.



Relationships are always written using a single thread.

### Relationship type

Write relationships to Neo4j:

```
CALL gds.graph.writeRelationship(
  'personsAndInstruments', ①
  'SIMILAR' ②
)
YIELD
  graphName, relationshipType, relationshipProperty, relationshipsWritten, propertiesWritten
```

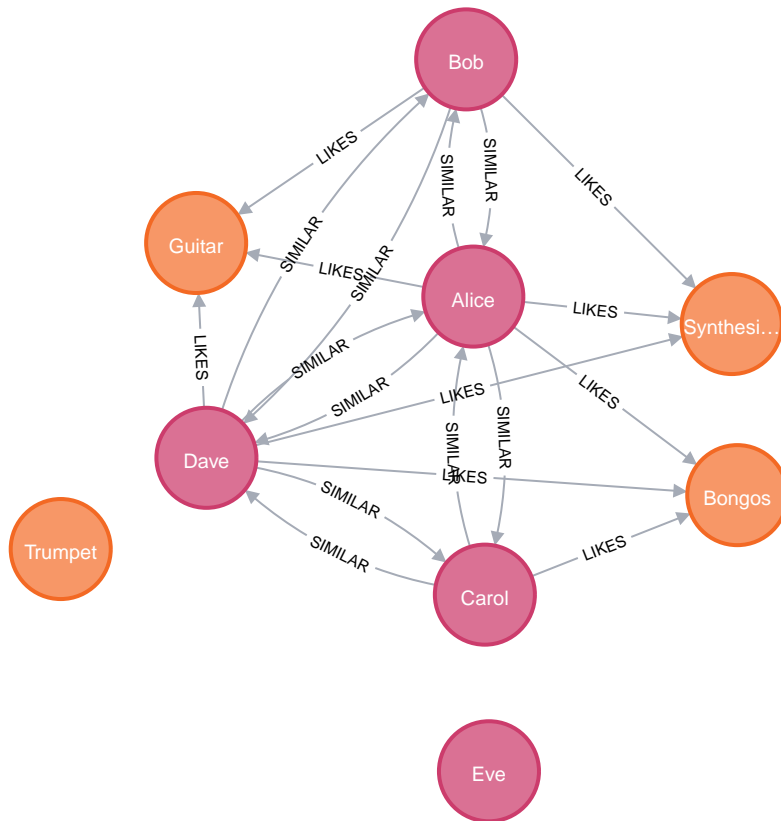
- ① The name of the projected graph.
- ② The relationship type we want to write back to the Neo4j database.

Table 105. Results

graphName	relationshipType	relationshipProperty	relationshipsWritten	propertiesWritten
"personsAndInstruments"	"SIMILAR"	null	10	0

By default, no relationship properties will be written, as it can be seen from the results, the `relationshipProperty` value is `null` and `propertiesWritten` are `0`.

Here is an illustration of how the example graph looks in Neo4j after executing the example above.



The `SIMILAR` relationships have been added to the underlying database and can be used in Cypher queries or for projecting to in-memory graph for running algorithms. The relationships in this example are undirected because we used `Node Similarity` to mutate the in-memory graph and this algorithm creates undirected relationships, this may not be the case if we use different algorithms.

### Relationship type with property

To write relationship properties, these have to be explicitly specified.

Write relationships and their properties to Neo4j:

```
CALL gds.graph.writeRelationship(
  'personsAndInstruments', ①
  'SIMILAR',                ②
  'score'                   ③
)
YIELD
  graphName, relationshipType, relationshipProperty, relationshipsWritten, propertiesWritten
```

- ① The name of the projected graph.

- ② The relationship type we want to write back to the Neo4j database.
- ③ The property name of the relationship we want to write back to the Neo4j database.

Table 106. Results

graphName	relationshipType	relationshipProperty	relationshipsWritten	propertiesWritten
"personsAndInstruments"	"SIMILAR"	"score"	10	10

## Delete

We can delete all relationships of a given type from a named graph in the catalog. This is useful to free up main memory or to remove accidentally created relationship types.



Deleting relationships of a given type is only possible if it is not the last relationship type present in the graph. If we still want to delete these relationships we need to [drop the graph](#) instead.

Delete all relationships of type **SIMILAR** from a named graph:

```
CALL gds.graph.deleteRelationships(
  'personsAndInstruments',      ①
  'SIMILAR'                     ②
)
YIELD
  graphName, relationshipType, deletedRelationships, deletedProperties
```

- ① The name of the projected graph.
- ② The relationship type we want to delete from the projected graph.

Table 107. Results

graphName	relationshipType	deletedRelationships	deletedProperties
"personsAndInstruments"	"SIMILAR"	10	{score=10}

## 4.1.9. Export operations

### Create Neo4j databases from named graphs

We can create new Neo4j databases from named in-memory graphs stored in the graph catalog. All nodes, relationships and properties present in an in-memory graph are written to a new Neo4j database. This includes data that has been projected in `gds.graph.create` and data that has been added by running algorithms in `mutate` mode. The newly created database will be stored in the Neo4j `databases` directory using a given database name.

The feature is useful in the following, exemplary scenarios:

- Avoid heavy write load on the operational system by exporting the data instead of writing back.
- Create an analytical view of the operational system that can be used as a basis for running algorithms.

- Produce snapshots of analytical results and persistent them for archiving and inspection.
- Share analytical results within the organization.

## Syntax

Export an in-memory graph to a new database in the Neo4j databases directory:

```
CALL gds.graph.export(graphName: String, configuration: Map)
YIELD
  dbName: String,
  graphName: String,
  nodeCount: Integer,
  nodePropertyCount: Integer,
  relationshipCount: Integer,
  relationshipTypeCount: Integer,
  relationshipPropertyCount: Integer,
  writeMillis: Integer
```

Table 108. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
configuration	Map	no	Additional parameters to configure the database export.

Table 109. Graph export configuration

Name	Type	Default	Optional	Description
dbName	String	none	No	The name of the exported Neo4j database.
writeConcurrency	Boolean	4	yes	The number of concurrent threads used for writing the database.
enableDebugLog	Boolean	false	yes	Prints debug information to Neo4j log files.
batchSize	Integer	10000	yes	Number of entities processed by one single thread at a time.
defaultRelationshipType	String	__ALL__	yes	Relationship type used for * relationship projections.
additionalNodeProperties	String, List or Map	{}	yes	Allows for exporting additional node properties from the original graph backing the in-memory graph.

Table 110. Results

Name	Type	Description
dbName	String	The name of the exported Neo4j database.
graphName	String	The name under which the graph is stored in the catalog.
nodeCount	Integer	The number of nodes exported.
nodePropertyCount	Integer	The number of node properties exported.
relationshipCount	Integer	The number of relationships exported.

Name	Type	Description
relationshipTypeCount	Integer	The number of relationship types exported.
relationshipPropertyCount	Integer	The number of relationship properties exported.
writeMillis	Integer	Milliseconds for writing the graph into the new database.

## Example

Export the `my-graph` from GDS into a Neo4j database called `mydatabase`:

```
CALL gds.graph.export('my-graph', { dbName: 'mydatabase' })
```

The new database can be started using [databases management commands](#).



The database must not exist when using the export procedure. It needs to be created manually using the following commands.

After running exporting the graph, we can start a new database and query the exported graph:

```
:use system
CREATE DATABASE mydatabase;
:use mydatabase
MATCH (n) RETURN n;
```

## Example with additional node properties

Suppose we have a graph `my-db-graph` in the Neo4j database that has a string node property `myproperty`, and that we have a corresponding in-memory graph called `my-in-memory-graph` which does not have the `myproperty` node property. If we want to export `my-in-memory-graph` but additionally add the `myproperty` properties from `my-db-graph` we can use the `additionalProperties` configuration parameter.

Export the `my-in-memory-graph` from GDS with `myproperty` from `my-db-graph` into a Neo4j database called `mydatabase`:

```
CALL gds.graph.export('my-graph', { dbName: 'mydatabase', additionalNodeProperties: ['myproperty'] })
```

The new database can be started using [databases management commands](#).



The original database (`my-db-graph`) must not have changed since loading the in-memory representation (`my-in-memory-graph`) that we export in order for the export to work correctly.

The `additionalNodeProperties` parameter uses the same syntax as `nodeProperties` of the [graph create procedure](#). So we could for instance define a default value for our `myproperty`.



Export the `my-in-memory-graph` from GDS with `myproperty` from `my-db-graph` with default value into a Neo4j database called `mydatabase`:

```
CALL gds.graph.export('my-graph', { dbName: 'mydatabase', additionalNodeProperties: [{ myproperty: {defaultValue: 'my-default-value'}}] })
```

# Chapter 5. Export a named graph to CSV

We can export named in-memory graphs stored in the graph catalog to a set of CSV files. All nodes, relationships and properties present in an in-memory graph are exported. This includes data that has been projected with `gds.graph.create` and data that has been added by running algorithms in `mutate` mode. The location of the exported CSV files can be configured via the configuration parameter `gds.export.location` in the `neo4j.conf`. All files will be stored in a subfolder using the specified export name. The export will fail if a folder with the given export name already exists.



The `gds.export.location` parameter must be configured for this feature.

## 5.1. Syntax

Export a named graph to a set of CSV files:

```
CALL gds.beta.graph.export.csv(graphName: String, configuration: Map)
YIELD
  graphName: String,
  exportName: String,
  nodeCount: Integer,
  nodePropertyCount: Integer,
  relationshipCount: Integer,
  relationshipTypeCount: Integer,
  relationshipPropertyCount: Integer,
  writeMillis: Integer
```

Table 111. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
configuration	Map	no	Additional parameters to configure the database export.

Table 112. Graph export configuration

Name	Type	Default	Optional	Description
exportName	String	none	No	The name of the directory where the graph is exported to. The absolute path of the exported CSV files depends on the configuration parameter <code>gds.export.location</code> in the <code>neo4j.conf</code> .
writeConcurrency	Boolean	4	yes	The number of concurrent threads used for writing the database.
defaultRelationshipType	String	__ALL__	yes	Relationship type used for * relationship projections.
additionalNodeProperties	String, List or Map	{}	yes	Allows for exporting additional node properties from the original graph backing the in-memory graph.

Table 113. Results

Name	Type	Description
graphName	String	The name under which the graph is stored in the catalog.
exportName	String	The name of the directory where the graph is exported to.
nodeCount	Integer	The number of nodes exported.
nodePropertyCount	Integer	The number of node properties exported.
relationshipCount	Integer	The number of relationships exported.
relationshipTypeCount	Integer	The number of relationship types exported.
relationshipPropertyCount	Integer	The number of relationship properties exported.
writeMillis	Integer	Milliseconds for writing the graph into the new database.

## 5.2. Estimation

As many other procedures in GDS, export to csv has an estimation mode. For more details see [Memory Estimation](#). Using the `gds.beta.graph.export.csv.estimate` procedure, it is possible to estimate the required disk space of the exported CSV files. The estimation uses sampling to generate a more accurate estimate.

Estimate the required disk space for exporting a named graph to CSV files.:

```
CALL gds.beta.graph.export.csv.estimate(graphName:String, configuration: Map)
YIELD
  nodeCount: Integer,
  relationshipCount: Integer,
  requiredMemory: String,
  treeView: String,
  mapView: Map,
  bytesMin: Integer,
  bytesMax: Integer,
  heapPercentageMin: Float,
  heapPercentageMax: Float;
```

Table 114. Parameters

Name	Type	Optional	Description
graphName	String	no	The name under which the graph is stored in the catalog.
configuration	Map	no	Additional parameters to configure the database export.

Table 115. Graph export estimate configuration

Name	Type	Default	Optional	Description
exportName	String	none	no	Name of the folder the exported CSV files are saved at.
samplingFactor or	Double	0.001	yes	The fraction of nodes and relationships to sample for the estimation.
writeConcurrency	Boolean	4	yes	The number of concurrent threads used for writing the database.

Name	Type	Default	Optional	Description
defaultRelationshipType	String	__ALL__	yes	Relationship type used for * relationship projections.

Table 116. Results

Name	Type	Description
nodeCount	Integer	The number of nodes in the graph.
relationshipCount	Integer	The number of relationships in the graph.
requiredMemory	String	An estimation of the required memory in a human readable format.
treeView	String	A more detailed representation of the required memory, including estimates of the different components in human readable format.
mapView	Map	A more detailed representation of the required memory, including estimates of the different components in structured format.
bytesMin	Integer	The minimum number of bytes required.
bytesMax	Integer	The maximum number of bytes required.
heapPercentageMin	Float	The minimum percentage of the configured maximum heap required.
heapPercentageMax	Float	The maximum percentage of the configured maximum heap required.

## 5.3. Export format

The format of the exported CSV files is based on the format that is supported by the Neo4j Admin [import](#) command.

### 5.3.1. Nodes

Nodes are exported into files grouped by the nodes labels, i.e., for every label combination that exists in the graph a set of export files is created. The naming schema of the exported files is:

`nodes_LABELS_INDEX.csv`, where:

- `LABELS` is the ordered list of labels joined by `_`.
- `INDEX` is a number between 0 and concurrency.

For each label combination one or more data files are created, as each exporter thread exports into a separate file.

Additionally, each label combination produces a single header file, which contains a single line describing the columns in the data files. More information about the header files can be found here: [CSV header format](#).

For example a Graph with the node combinations `:A`, `:B` and `:A:B` might create the following files

```
nodes_A_header.csv
nodes_A_0.csv
nodes_B_header.csv
nodes_B_0.csv
nodes_B_2.csv
nodes_A_B_header.csv
nodes_A_B_0.csv
nodes_A_B_1.csv
nodes_A_B_2.csv
```

## 5.3.2. Relationships

The format of the relationship files is similar to those of the nodes. Relationships are exported into files grouped by the relationship type. The naming schema of the exported files is:

`relationships_TYPE_INDEX.csv`, where:

- `TYPE` is the relationship type
- `INDEX` is a number between 0 and concurrency.

For each relationship type one or more data files are created, as each exporter thread exports into a separate file.

Additionally, each relationship type produces a single header file, which contains a single line describing the columns in the data files.

For example a Graph with the relationship types `:KNOWS`, `:LIVES_IN` might create the following files

```
relationships_KNOWS_header.csv
relationships_KNOWS_0.csv
relationships_LIVES_IN_header.csv
relationships_LIVES_IN_0.csv
relationships_LIVES_IN_2.csv
```

## 5.4. Example

Export the `my-graph` from GDS into a directory `my-export`:

```
CALL gds.beta.graph.export.csv('my-graph', { exportName: 'my-export' })
```

## 5.5. Example with additional node properties

Suppose we have a graph `my-db-graph` in the Neo4j database that has a string node property `myproperty`, and that we have a corresponding in-memory graph called `my-in-memory-graph` which does not have the `myproperty` node property. If we want to export `my-in-memory-graph` but additionally add the `myproperty` properties from `my-db-graph` we can use the `additionalProperties` configuration parameter.

Export the `my-in-memory-graph` from GDS with the `myproperty` from `my-db-graph` into a directory `my-export`:

```
CALL gds.beta.graph.export.csv('my-graph', { exportName: 'my-export', additionalNodeProperties: ['myproperty'] })
```



The original database (`my-db-graph`) must not have changed since loading the in-memory representation (`my-in-memory-graph`) that we export in order for the export to work correctly.

The `additionalNodeProperties` parameter uses the same syntax as `nodeProperties` of the `graph create` procedure. So we could for instance define a default value for our `myproperty`.

Export the `my-in-memory-graph` from GDS with `myproperty` from `my-db-graph` with default value into a directory called `my-export`:

```
CALL gds.beta.graph.export.csv('my-graph', { exportName: 'my-export', additionalNodeProperties: [{ myproperty: {defaultValue: 'my-default-value'}}] })
```

## 5.6. Anonymous graphs

The typical workflow when using the GDS library is to `create a graph` and store it in the catalog. This is useful to minimize reads from Neo4j and to run an algorithm with various settings or several algorithms on the same graph projection.

However, if you want to quickly run a single algorithm, it can be convenient to use an anonymous projection. The syntax is similar to the ordinary syntax for `gds.graph.create`, described [here](#). It differs however in that relationship projections cannot have more than one property. Moreover, the `nodeProjection` and `relationshipProjection` arguments are named and placed in the configuration map of the algorithm:

Anonymous native projection syntax

```
CALL gds.<algo>.<mode>(  
  {  
    nodeProjection: String, List or Map,  
    relationshipProjection: String, List or Map,  
    nodeProperties: String, List or Map,  
    relationshipProperties: String, List or Map,  
    // algorithm and other create configuration  
  }  
)
```

The following examples demonstrates creating an anonymous graph from `Person` nodes and `KNOWS` relationships.

```
CALL gds.<algo>.<mode>(  
  {  
    nodeProjection: 'Person',  
    relationshipProjection: 'KNOWS',  
    nodeProperties: 'age',  
    relationshipProperties: 'weight',  
    // algorithm and other create configuration  
  }  
)
```

The above example can be an alternative to the calls below:

```

CALL gds.graph.create(
  {
    'new-graph-name',
    'Person',
    'KNOWS',
    {
      nodeProperties: 'age',
      relationshipProperties: 'weight'
      // other create configuration
    }
  }
);
CALL gds.<algo>.<mode>(
  'new-graph-name',
  {
    // algorithm configuration
  }
);
CALL gds.graph.drop('new-graph-name');

```

Similarly for [Cypher projection](#), the explicit creation with `gds.graph.create.cypher` can be inlined in an algorithm call using the `nodeQuery` and `relationshipQuery` configuration keys.

Anonymous cypher projection syntax

```

CALL gds.<algo>.<mode>(
  {
    nodeQuery: String,
    relationshipQuery: String,
    // algorithm and other create configuration
  }
)

```

## 5.7. Node Properties

The Neo4j Graph Data Science Library is capable of augmenting nodes with additional properties. These properties can be loaded from the database when the graph projection is created. Many algorithms can also persist their result as one or more node properties when they are run using the `mutate` mode.

### 5.7.1. Supported types

The Neo4j Graph Data Science library does not support all property types that are supported by the Neo4j database. Every supported type also defines a fallback value, which is used to indicate that the value of this property is not set.

The following table lists the supported property types, as well as, their corresponding fallback values.

- `Long` - `Long.MIN_VALUE`
- `Double` - `NaN`
- `Long Array` - `null`
- `Float Array` - `null`
- `Double Array` - `null`

## 5.7.2. Defining the type of a node property

When creating a graph projection that specifies a set of node properties, the type of these properties is automatically determined using the first property value that is read by the loader for any specified property. All integral numerical types are interpreted as **Long** values, all floating point values are interpreted as **Double** values. Array values are explicitly defined by the type of the values that the array contains, i.e. a conversion of, for example, an **Integer Array** into a **Long Array** is not supported. Arrays with mixed content types are not supported.

## 5.7.3. Automatic type conversion

Most algorithms that are capable of using node properties require a specific property type. In cases of a mismatch between the type of the provided property and the required type, the library will try to convert the property value into the required type. This automatic conversion only happens when the following conditions are satisfied:

- Neither the given, nor the expected type are an **Array** type.
- The conversion is loss-less
  - **Long to Double**: The Long values does not exceed the supported range of the Double type.
  - **Double to Long**: The Double value does not have any decimal places.

The algorithm computation will fail if any of these conditions are not satisfied for any node property value.



The automatic conversion is computationally more expensive and should therefore be avoided in performance critical applications.

## 5.8. Utility functions

### 5.8.1. System Functions

Name	Description
<code>gds.version</code>	Return the version of the installed Neo4j Graph Data Science library.

Usage:

```
RETURN gds.version() AS version
```

Table 117. Results

version
"1.8.9"



## 5.8.2. Numeric Functions

Table 118. Numeric Functions

Name	Description
<code>gds.util.NaN</code>	Returns NaN as a Cypher value.
<code>gds.util.infinity</code>	Return infinity as a Cypher value.
<code>gds.util.isFinite</code>	Return false if the given argument is $\pm$ Infinity, NaN, or null.
<code>gds.util.isInfinite</code>	Return true if the given argument is $\pm$ Infinity, NaN, or null.

### Syntax

Name	Parameter
<code>gds.util.NaN()</code>	-
<code>gds.util.infinity()</code>	-
<code>gds.util.isFinite(value: NUMBER)</code>	value to be checked if it is finite.
<code>gds.util.isInfinite(value: NUMBER)</code>	value to be checked if it is infinite.

### Examples

Example for `gds.util.isFinite`:

```
UNWIND [1.0, gds.util.NaN(), gds.util.infinity()] AS value
RETURN gds.util.isFinite(value) AS isFinite
```

Table 119. Results

isFinite
true
false
false

Example for `gds.util.isInfinite()`:

```
UNWIND [1.0, gds.util.NaN(), gds.util.infinity()] AS value
RETURN gds.util.isInfinite(value) AS isInfinite
```

Table 120. Results

isInfinite
false
true
true

The utility function `gds.util.NaN` can be used as a default value for input parameters, as shown in the examples of [cosine similarity](#). A common usage of `gds.util.IsFinite` and `gds.util.IsInfinite` is for filtering streamed results, as for instance seen in the examples of [gds.alpha.allShortestPaths](#).

### 5.8.3. Node id functions

Table 121. Node id functions

Name	Description
<code>gds.util.asNode</code>	Return the node object for the given node id or null if none exists.
<code>gds.util.asNodes</code>	Return the node objects for the given node ids or an empty list if none exists.

#### Syntax

Name	Parameters
<code>gds.util.asNode(nodeId: NUMBER)</code>	nodeId of a node in the neo4j-graph
<code>gds.util.asNodes(nodeIds: List of NUMBER)</code>	list of nodeIds of nodes in the neo4j-graph

#### Examples

Consider the graph created by the following Cypher statement:

Example graph:

```
CREATE (nAlice:User {name: 'Alice'})
CREATE (nBridget:User {name: 'Bridget'})
CREATE (nCharles:User {name: 'Charles'})
CREATE (nAlice)-[:LINK]->(nBridget)
CREATE (nBridget)-[:LINK]->(nCharles)
```

Example for `gds.util.asNode`:

```
MATCH (u:User{name: 'Alice'})
WITH id(u) AS nodeId
RETURN gds.util.asNode(nodeId).name AS node
```

Table 122. Results

node
"Alice"

Example for `gds.util.asNodes`:

```
MATCH (u:User)
WHERE NOT u.name = 'Charles'
WITH collect(id(u)) AS nodeIds
RETURN [x in gds.util.asNodes(nodeIds) | x.name] AS nodes
```

Table 123. Results

```
nodes
```

```
[Alice, Bridget]
```

As many algorithms streaming mode only return the node id, `gds.util.asNode` and `gds.util.asNodes` can be used to retrieve the whole node from the neo4j database.

## 5.9. Cypher on GDS graph Enterprise edition

This feature is in the alpha tier.

Exploring in-memory graphs after loading them and potentially executing algorithms in mutate mode can be tricky in the Neo4j Graph Data Science library. A natural way to achieve this in the Neo4j database is to use Cypher queries. Cypher queries allow for example to get a hold of which properties are present on a node among many other things. Executing Cypher queries on an in-memory graph can be achieved by leveraging the `gds.alpha.create.cypherdb` procedure. This procedure will create a new impermanent database which you can switch to. That database will then use data from the in-memory graph as compared to the store files for usual Neo4j databases.

### 5.9.1. Limitations

Although it is possible to execute arbitrary Cypher queries on the database created by the `gds.alpha.create.cypherdb` procedure, not every aspect of Cypher is implemented yet. Some known limitations are listed below:

- Dropping the newly created database
  - Restarting the DBMS will remove the database instead
- Writes
  - All queries that attempt to write things, such as nodes, properties or labels, will fail

### 5.9.2. Syntax

```
CALL gds.alpha.create.cypherdb(  
  dbName: String  
  graphName: String  
)  
YIELD  
  dbName: String,  
  graphName: String,  
  createMillis: Integer
```

Table 124. Parameters

Name	Type	Optional	Description
dbName	String	no	The name under which the new database is stored.
graphName	String	no	The name under which the graph is stored in the catalog.

Table 125. Results

Name	Type	Description
dbName	String	The name under which the new database is stored.
graphName	String	The name under which the graph is stored in the catalog.
createMillis	Integer	Milliseconds for creating the database.

### 5.9.3. Example

To demonstrate how to execute cypher statements on in-memory graphs we are going to create a simple social network graph. We will use this graph to create a new database which we will execute our statements on.

```
CREATE
  (alice:Person { name: 'Alice', age: 23 }),
  (bob:Person { name: 'Bob', age: 42 }),
  (carl:Person { name: 'Carl', age: 31 }),

  (alice)-[:KNOWS]->(bob),
  (bob)-[:KNOWS]->(alice),
  (alice)-[:KNOWS]->(carl)
```

We will now load a graph projection of the created graph via the [graph create](#) procedure:

Project **Person** nodes and **KNOWS** relationships:

```
CALL gds.graph.create(
  'social_network',
  'Person',
  'KNOWS'
)
YIELD
  graphName, nodeCount, relationshipCount
```

Table 126. Results

graph	nodeCount	relationshipCount
"social_network"	3	3

With a named graph loaded into the Neo4j Graph Data Science library, we can proceed to create the new database using the loaded graph as underlying data.

Create a new database **gdsDb** using our **social\_network** graph:

```
CALL gds.alpha.create.cypherdb(
  'gdsDb',
  'social_network'
)
```

In order to verify that the new database was created successfully we can use the Neo4j database administration commands.

```
SHOW DATABASES
```

Table 127. Results

name	address	role	requestedStatus	currentStatus	error	default	home
"neo4j"	"localhost:7687"	"standalone"	"online"	"online"	""	true	true
"system"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false
"gdsDb"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false

We can now switch to the newly created database.

```
:use gdsDb
```

Finally, we are set up to execute cypher queries on our in-memory graph.

```
MATCH (n:Person)-[:KNOWS]->(m:Person) RETURN n.age AS age1, m.age AS age2
```

Table 128. Results

age1	age2
23	42
42	23
23	31

We can see that the returned ages correspond to the structure of the original graph.

## 5.10. Administration

The GDS catalog offers elevated access to administrator users. Any user granted a role with the name `admin` is considered an administrator by GDS.

A GDS administrator has access to graphs created by any other user. This includes the ability to list, drop and run algorithms over these graphs.

### 5.10.1. Disambiguating identically named graphs

Sometimes, several users (including the admin user themselves) could have a graph with the same name. To disambiguate between these graphs, the `username` configuration parameter can be used.

### 5.10.2. Examples

We will illustrate the administrator capabilities using a small example. In this example we have three users where one is an administrator. We create the users and set up the roles using the following Cypher commands:

```

CREATE USER alice SET PASSWORD $alice_pw CHANGE NOT REQUIRED;
CREATE USER bob SET PASSWORD $bob_pw CHANGE NOT REQUIRED;
CREATE USER carol SET PASSWORD $carol_pw CHANGE NOT REQUIRED;

GRANT ROLE reader TO alice;
GRANT ROLE reader TO bob;
GRANT ROLE admin TO carol;

```

As we can see, **alice** and **bob** are standard users with read access to the database. **carol** is an administrator by virtue of being granted the **admin** role (for more information about this role see the [Cypher manual](#)).

Now **alice** and **bob** each create a few graphs. They both create a graph called **graphA** and **bob** also creates a graph called **graphB**.

## Listing

To list all graphs from all users, **carol** simply uses the graph list procedure.

*Listing all graphs as administrator user:*

```

CALL gds.graph.list()
YIELD graphName

```

Table 129. Results

graphName
"graphA"
"graphA"
"graphB"

Notice that all graphs from all users are visible to **carol** since they are considered a GDS admin.

## Running algorithms with other users' graphs

**carol** may use **graphB** by simply naming it.

**carol** can run WCC on the **graphB** graph owned by **bob**:

```

CALL gds.wcc.stats('graphB')
YIELD componentCount

```

To use the **graphA** owned by **alice**, **carol** must use the **username** override.

**carol** can run WCC on **graphA** owned by **alice**:

```

CALL gds.wcc.stats('graphA', { username: 'alice' })
YIELD componentCount

```

## Dropping other users' graphs

Unlike for listing, the full procedure signature must be used when using the `username` override to disambiguate. In the query below we have used the default values for the second and third parameter for the drop procedure. `username` is the fourth parameter. For more details see [Dropping graphs](#).

To drop `graphA` owned by `bob`, `carol` can run the following:

```
CALL gds.graph.drop('graphA', true, '', 'bob')
YIELD graphName
```

# Chapter 6. Model catalog

Some graph algorithms use trained models in their computation. A model is generally a mathematical formula representing a real-world or fictitious entities. Each algorithm requiring a trained model provides the formulation and means to compute this model (see [GraphSage train syntax](#)).

The model catalog is a concept within the GDS library that allows storing and managing multiple trained models by name.

This chapter explains the available model catalog operations.

Name	Description
<a href="#">gds.beta.model.list</a>	Prints information about models that are currently available in the catalog.
<a href="#">gds.beta.model.exists</a>	Checks if a named model is available in the catalog.
<a href="#">gds.beta.model.drop</a>	Drops a named model from the catalog.
<a href="#">gds.alpha.model.store</a>	Stores a names model from the catalog on disk.
<a href="#">gds.alpha.model.load</a>	Loads a named and stored model from disk.
<a href="#">gds.alpha.model.delete</a>	Removes a named and stored model from disk.
<a href="#">gds.alpha.model.publish</a>	Makes a model accessible to all users.



Training models is a responsibility of the corresponding algorithm and is provided by a procedure mode - `train`. Training, using, listing, and dropping named models are management operations bound to a Neo4j user. Models trained by a different Neo4j user are not accessible at any time.

## 6.1. Listing models

Information about models in the catalog can be retrieved using the `gds.beta.model.list()` procedure.

### 6.1.1. Syntax

List models from the catalog:

```
CALL gds.beta.model.list(modelName: String)
YIELD
  modelInfo: Map,
  trainConfig: Map,
  graphSchema: Map,
  loaded: Boolean,
  stored: Boolean,
  creationTime: DateTime,
  shared: Boolean
```

Table 130. Parameters



Name	Type	Default	Optional	Description
modelName	String	n/a	yes	The name of a model. If not specified, all models in the catalog are listed.

Table 131. Results

Name	Type	Description
modelInfo	Map	Detailed information about the trained model. Always includes the <code>modelName</code> and <code>modelType</code> , e.g., <code>GraphSAGE</code> . Dependent on the model type, there are additional fields.
trainConfig	Map	The configuration used for training the model.
graphSchema	Map	The schema of the graph on which the model was trained.
loaded	Boolean	True, if the model is <code>loaded</code> in the in-memory model catalog.
stored	Boolean	True, if the model is <code>stored</code> on disk.
creationTime	Datetime	Time when the model was created.
shared	Boolean	True, if the model is <code>shared</code> between users.

## 6.1.2. Examples

Once we have trained models in the catalog we can see information about either all of them or a single model using its name

### Listing all models

Listing detailed information about all models:

```
CALL gds.beta.model.list()
YIELD modelInfo, loaded, shared, stored
RETURN modelInfo.modelName AS modelName, loaded, shared, stored
```

Table 132. Results

modelName	loaded	shared	stored
"my-model"	true	false	false

### Listing a specific model

Listing detailed information about specific model:

```
CALL gds.beta.model.list('my-model')
YIELD modelInfo, loaded, shared, stored
RETURN modelInfo.modelName AS modelName, loaded, shared, stored
```

Table 133. Results

modelName	loaded	shared	stored
"my-model"	true	false	false

## 6.2. Checking if a model exists

We can check if a model is available in the catalog by looking up its name.

### 6.2.1. Syntax

Check if a model exists in the catalog:

```
CALL gds.beta.model.exists(modelName: String)
YIELD
  modelName: String,
  modelType: String,
  exists: Boolean
```

Table 134. Parameters

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a model.

Table 135. Results

Name	Type	Description
modelName	String	The name of a model.
modelType	String	The type of the model.
exists	Boolean	True, if the model exists in the model catalog.

### 6.2.2. Example

In this section we are going to demonstrate the usage of `gds.beta.model.exists`. Assume we trained a model by running `train` on one of our [Machine learning algorithms](#).

Check if a model exists in the catalog:

```
CALL gds.beta.model.exists('my-model');
```

Table 136. Results

modelName	modelType	exists
"my-model"	"graphSage"	true

## 6.3. Removing models

If we no longer need a trained model and want to free up memory, we can remove the model from the catalog.

### 6.3.1. Syntax

Remove a model from the catalog:

```
CALL gds.beta.model.drop(modelName: String)
YIELD
  modelInfo: Map,
  trainConfig: Map,
  graphSchema: Map,
  loaded: Boolean,
  stored: Boolean,
  creationTime: DateTime,
  shared: Boolean
```

Table 137. Parameters

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a model stored in the catalog.

Table 138. Results

Name	Type	Description
modelInfo	Map	Detailed information about the trained model. Always includes the <code>modelName</code> and <code>modelType</code> , e.g., <code>GraphSAGE</code> . Dependent on the model type, there are additional fields.
trainConfig	Map	The configuration used for training the model.
graphSchema	Map	The schema of the graph on which the model was trained.
loaded	Boolean	True, if the model is <code>loaded</code> in the in-memory model catalog.
stored	Boolean	True, if the model is <code>stored</code> on disk.
creationTime	Datetime	Time when the model was created.
shared	Boolean	True, if the model is <code>shared</code> between users.

## 6.3.2. Example

In this section we are going to demonstrate the usage of `gds.beta.model.drop`. Assume we trained a model by running `train` on one of our [Machine learning algorithms](#).

Remove a model from the catalog:

```
CALL gds.beta.model.drop('my-model')
YIELD modelInfo, loaded, shared, stored
RETURN modelInfo.modelName AS modelName, loaded, shared, stored
```

Table 139. Results

modelName	loaded	shared	stored
"my-model"	true	false	false

In this example, the removed `my-model` was of the imaginary type `some-model-type`. The model was loaded in-memory, but neither stored on disk nor published.



If the model name does not exist, an error will be raised.

## 6.4. Storing models on disk

The model store feature is in the alpha tier.

The model catalog exists as long as the Neo4j instance is running. When Neo4j is restarted, models are no longer available in the catalog and need to be trained again. This can be prevented by storing a model on disk.

The location of the stored models can be configured via the configuration parameter `gds.model.store_location` in the `neo4j.conf`. The location must be a directory and writable by the Neo4j process.



The `gds.model.store_location` parameter must be configured for this feature.

### 6.4.1. Storing models from the catalog on disk Alpha

#### Models that can be stored

- [GraphSAGE model](#)
- [Node Classification model](#)
- [Link Prediction model](#)

#### Models that cannot be stored

- [Link prediction training pipeline](#)
- [Link prediction pipeline](#)

## Syntax

Remove a model from the catalog:

```
CALL gds.alpha.model.store(  
  modelName: String,  
  failIfUnsupportedType: Boolean  
)  
YIELD  
  modelName: String,  
  storeMillis: Integer
```

Table 140. Parameters

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a model.
failIfUnsupportedType	Boolean	true	yes	By default, the library will raise an error when trying to store a non-supported model. When set to <code>false</code> , the procedure returns an empty result.

Table 141. Results

Name	Type	Description
modelName	String	The name of the stored model.
storeMillis	Integer	The number of milliseconds it took to store the model.

## Example

Store a model on disk:

```
CALL gds.alpha.model.store('my-model')
YIELD
  modelName,
  storeMillis
```

## 6.4.2. Loading models from disk Alpha

GDS will discover available models from the configured store location upon database startup. During discovery, only model metadata is loaded, not the actual model data. In order to use a stored model, it has to be explicitly loaded.

## Syntax

Remove a model from the catalog:

```
CALL gds.alpha.model.load(modelName: String)
YIELD
  modelName: String,
  loadMillis: Integer
```

Table 142. Parameters

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a model.

Table 143. Results

Name	Type	Description
modelName	String	The name of the loaded model.
loadMillis	Integer	The number of milliseconds it took to load the model.

## Example

Store a model on disk:

```
CALL gds.alpha.model.load('my-model')
YIELD
  modelName,
  loadMillis
```

To verify if a model is loaded, we can use the `gds.beta.model.list` procedure. The procedure returns flags to indicate if the model is stored and if the model is loaded into memory. The operation is idempotent, and

skips loading if the model is already loaded.

### 6.4.3. Deleting models from disk Alpha

To remove a stored model from disk, it has to be deleted. This is different from dropping a model.

Dropping a model will remove it from the in-memory model catalog, but not from disk. Deleting a model will remove it from disk, but keep it in the in-memory model catalog if it was already loaded.

#### Syntax

Remove a model from the catalog:

```
CALL gds.alpha.model.delete(modelName: String)
YIELD
  modelName: String,
  deleteMillis: Integer
```

Table 144. Parameters

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a model.

Table 145. Results

Name	Type	Description
modelName	String	The name of the loaded model.
deleteMillis	Integer	The number of milliseconds it took to delete the model.

#### Example

Store a model on disk:

```
CALL gds.alpha.model.delete('my-model')
YIELD
  modelName,
  deleteMillis
```

## 6.5. Publishing models

Publishing models is an alpha tier feature.

By default, a trained model is visible to the user that created it. Making a model accessible to other users can be achieved by publishing it.

### 6.5.1. Syntax

Publish a model from the catalog:

```
CALL gds.alpha.model.publish(modelName: String)
YIELD
  modelInfo: Map,
  trainConfig: Map,
  graphSchema: Map,
  loaded: Boolean,
  stored: Boolean,
  creationTime: DateTime,
  shared: Boolean
```

Table 146. Parameters

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a model stored in the catalog.

Table 147. Results

Name	Type	Description
modelInfo	Map	Detailed information about the trained model. Always includes the <code>modelName</code> and <code>modelType</code> , e.g., <code>GraphSAGE</code> . Dependent on the model type, there are additional fields.
trainConfig	Map	The configuration used for training the model.
graphSchema	Map	The schema of the graph on which the model was trained.
loaded	Boolean	True, if the model is <code>loaded</code> in the in-memory model catalog.
stored	Boolean	True, if the model is <code>stored</code> on disk.
creationTime	Datetime	Time when the model was created.
shared	Boolean	True, if the model is <code>shared</code> between users.

## 6.5.2. Examples

Publishing trained model:

```
CALL gds.alpha.model.publish('my-model')
YIELD modelInfo, loaded, shared, stored
RETURN modelInfo.modelName AS modelName, shared
```

Table 148. Results

modelName	shared
"my-model_public"	true

We can see that the model is now shared. The shared model has the `_public` suffix.

# Chapter 7. Algorithms

The Neo4j Graph Data Science (GDS) library contains many graph algorithms. The algorithms are divided into categories which represent different problem classes. The categories are listed in this chapter.

Algorithms exist in one of three tiers of maturity:

- Production-quality
  - Indicates that the algorithm has been tested with regards to stability and scalability.
  - Algorithms in this tier are prefixed with `gds.<algorithm>`.
- Beta
  - Indicates that the algorithm is a candidate for the production-quality tier.
  - Algorithms in this tier are prefixed with `gds.beta.<algorithm>`.
- Alpha
  - Indicates that the algorithm is experimental and might be changed or removed at any time.
  - Algorithms in this tier are prefixed with `gds.alpha.<algorithm>`.

This chapter is divided into the following sections:

- [Syntax overview](#)
- [Centrality](#)
- [Community detection](#)
- [Similarity](#)
- [Path finding](#)
- [Topological link prediction](#)
- [Node embeddings](#)
- [Machine learning models](#)
- [Auxiliary procedures](#)
- [Pregel API](#)

## 7.1. Syntax overview

The general algorithm syntax comes in two variants:

- Named graph variant
  - The graph to operate over will be read from the graph catalog.
- Anonymous graph variant
  - The graph to operate over will be created and deleted as part of the algorithm execution.

Each syntax variant additionally provides different execution modes. These are the supported execution



modes:

- **stream**
  - Returns the result of the algorithm as a stream of records.
- **stats**
  - Returns a single record of summary statistics, but does not write to the Neo4j database.
- **mutate**
  - Writes the results of the algorithm to the in-memory graph and returns a single record of summary statistics. This mode is designed for the named graph variant, as its effects will be invisible on an anonymous graph.
- **write**
  - Writes the results of the algorithm to the Neo4j database and returns a single record of summary statistics.

Finally, an execution mode may be **estimated** by appending the command with **estimate**.



Only the production-quality tier guarantees availability of all execution modes and estimation procedures.

Including all of the above mentioned elements leads to the following syntax outlines:

*Syntax composition for the named graph variant:*

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
  graphName: String,
  configuration: Map
)
```

*Syntax composition for the anonymous graph variant:*

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
  configuration: Map
)
```

The detailed sections in this chapter include concrete syntax overviews and examples.

## 7.2. Centrality

Centrality algorithms are used to determine the importance of distinct nodes in a network. The Neo4j GDS library includes the following centrality algorithms, grouped by quality tier:

- Production-quality
  - [Page Rank](#)
  - [Article Rank](#)
  - [Eigenvector Centrality](#)
  - [Betweenness Centrality](#)

- [Degree Centrality](#)
- Alpha
  - [Closeness Centrality](#)
  - [Harmonic Centrality](#)
  - [HITS](#)
  - [Influence Maximization](#)

## 7.2.1. PageRank

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

The PageRank algorithm measures the importance of each node within the graph, based on the number incoming relationships and the importance of the corresponding source nodes. The underlying assumption roughly speaking is that a page is only as important as the pages that link to it.

PageRank is introduced in the original Google paper as a function that solves the following equation:

$$PR(A) = (1 - d) + d\left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_n)}{C(T_n)}\right)$$

where,

- we assume that a page A has pages  $T_1$  to  $T_n$  which point to it.
- d is a damping factor which can be set between 0 (inclusive) and 1 (exclusive). It is usually set to 0.85.
- $C(A)$  is defined as the number of links going out of page A.

This equation is used to iteratively update a candidate solution and arrive at an approximate solution to the same equation.

For more information on this algorithm, see:

- [The original google paper](#)
- [An Efficient Partition-Based Parallel PageRank Algorithm](#)
- [PageRank beyond the web](#) for use cases



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

## Considerations

There are some things to be aware of when using the PageRank algorithm:

- If there are no relationships from within a group of pages to outside the group, then the group is considered a spider trap.
- Rank sink can occur when a network of pages is forming an infinite cycle.
- Dead-ends occur when pages have no outgoing relationship.

Changing the damping factor can help with all the considerations above. It can be interpreted as a probability of a web surfer to sometimes jump to a random page and therefore not getting stuck in sinks.

## Syntax

This section covers the syntax used to execute the PageRank algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run PageRank in stream mode on a named graph.

```
CALL gds.pageRank.stream(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  nodeId: Integer,  
  score: Float
```

Table 149. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 150. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 151. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Page Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 152. Results

Name	Type	Description
nodeId	Integer	Node ID.
score	Float	PageRank score.

Run PageRank in stats mode on a named graph.

```
CALL gds.pageRank.stats(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 153. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 154. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 155. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Page Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 156. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <code>centralityDistribution</code> .
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.



Run PageRank in mutate mode on a named graph.

```
CALL gds.pageRank.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  nodePropertiesWritten: Integer,  
  ranIterations: Integer,  
  didConverge: Boolean,  
  createMillis: Integer,  
  computeMillis: Integer,  
  postProcessingMillis: Integer,  
  mutateMillis: Integer,  
  centralityDistribution: Map,  
  configuration: Map
```

Table 157. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 158. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 159. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Page Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.

Name	Type	Default	Optional	Description
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <b>None</b> , <b>MinMax</b> , <b>Max</b> , <b>Mean</b> , <b>Log</b> , <b>L1Norm</b> , <b>L2Norm</b> and <b>StdScore</b> .

Table 160. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <b>centralityDistribution</b> .
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	The number of properties that were written to the in-memory graph.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

Run PageRank in write mode on a named graph.

```
CALL gds.pageRank.write(
  graphName: String,
  configuration: Map
)
YIELD
  nodePropertiesWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 161. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 162. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 163. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Page Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Name	Type	Default	Optional	Description
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <code>None</code> , <code>MinMax</code> , <code>Max</code> , <code>Mean</code> , <code>Log</code> , <code>L1Norm</code> , <code>L2Norm</code> and <code>StdScore</code> .

Table 164. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <code>centralityDistribution</code> .
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run PageRank in write mode on an anonymous graph:

```
CALL gds.pageRank.write(
  configuration: Map
)
YIELD
  nodePropertiesWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 165. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 166. Algorithm specific configuration

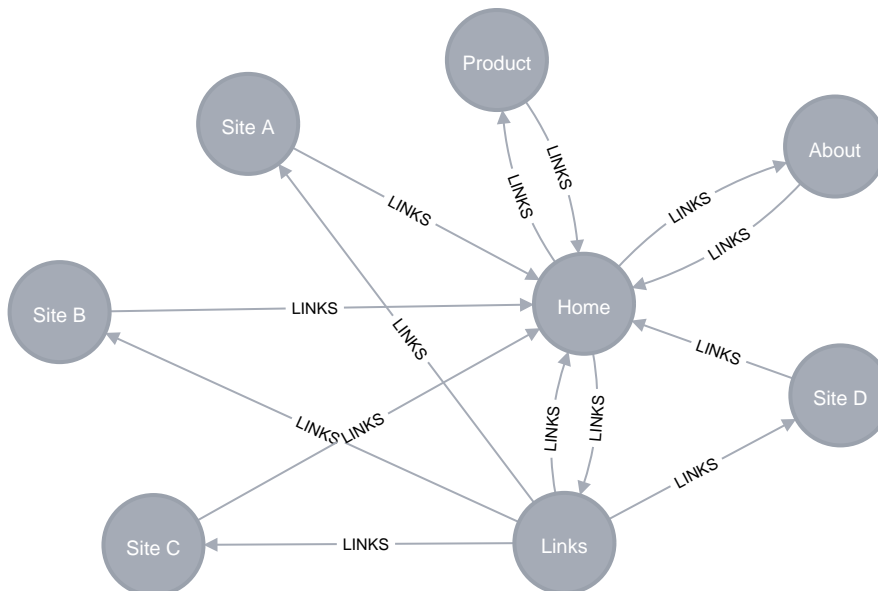
Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Page Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Name	Type	Default	Optional	Description
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <b>None</b> , <b>MinMax</b> , <b>Max</b> , <b>Mean</b> , <b>Log</b> , <b>L1Norm</b> , <b>L2Norm</b> and <b>StdScore</b> .

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the PageRank algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small web network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
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);
```

This graph represents eight pages, linking to one another. Each relationship has a property called `weight`, which describes the importance of the relationship.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Page',
  'LINKS',
  {
    relationshipProperties: 'weight'
  }
)
```

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.pageRank.write.estimate('myGraph', {
  writeProperty: 'pageRank',
  maxIterations: 20,
  dampingFactor: 0.85
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 167. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
8	14	696	696	"696 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the score for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode:

```
CALL gds.pageRank.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 168. Results

name	score
"Home"	3.215681999884452
"About"	1.0542700552146722
"Links"	1.0542700552146722
"Product"	1.0542700552146722
"Site A"	0.3278578964488539
"Site B"	0.3278578964488539
"Site C"	0.3278578964488539
"Site D"	0.3278578964488539

The above query is running the algorithm in `stream` mode as `unweighted` and the returned scores are not normalized. Below, one can find an example for [weighted graphs](#). Another [example](#) shows the application of a scaler to normalize the final scores.



While we are using the `stream` mode to illustrate running the algorithm as `weighted` or `unweighted`, all the algorithm modes support this configuration parameter.



## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm and returns the result in form of statistical and measurement values

```
CALL gds.pageRank.stats('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85
})
YIELD centralityDistribution
RETURN centralityDistribution.max AS max
```

Table 169. Results

max
3.2156810760498047

The centrality histogram can be useful for inspecting the computed scores or perform normalizations.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the score for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.pageRank.mutate('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  mutateProperty: 'pagerank'
})
YIELD nodePropertiesWritten, ranIterations
```

Table 170. Results

nodePropertiesWritten	ranIterations
8	20

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the score for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.pageRank.write('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  writeProperty: 'pagerank'
})
YIELD nodePropertiesWritten, ranIterations
```

Table 171. Results

nodePropertiesWritten	ranIterations
8	20

## Weighted

By default, the algorithm is considering the relationships of the graph to be `unweighted`, to change this behaviour we can use configuration parameter called `relationshipWeightProperty`. In the `weighted` case, the previous score of a node send to its neighbors, is multiplied by the relationship weight and then divided by the sum of the weights of its outgoing relationships. If the value of the relationship property is negative it will be ignored during computation. Below is an example of running the algorithm using the relationship property.

The following will run the algorithm in `stream` mode using relationship weights:

```
CALL gds.pageRank.stream('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  relationshipWeightProperty: 'weight'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 172. Results

name	score
"Home"	3.53751028396339
"Product"	1.9357838291651097
"About"	0.7452612763883698
"Links"	0.7452612763883698
"Site A"	0.18152677135466103

name	score
"Site B"	0.18152677135466103
"Site C"	0.18152677135466103
"Site D"	0.18152677135466103



We are using `stream` mode to illustrate running the algorithm as `weighted` or `unweighted`, all the algorithm modes support this configuration parameter.

## Tolerance

The `tolerance` configuration parameter denotes the minimum change in scores between iterations. If all scores change less than the configured `tolerance` value the result stabilises, and the algorithm returns.

The following will run the algorithm in `stream` mode using bigger `tolerance` value:

```
CALL gds.pageRank.stream('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  tolerance: 0.1
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 173. Results

name	score
"Home"	1.5812450669583336
"About"	0.5980194356381945
"Links"	0.5980194356381945
"Product"	0.5980194356381945
"Site A"	0.23374955154166668
"Site B"	0.23374955154166668
"Site C"	0.23374955154166668
"Site D"	0.23374955154166668

In this example we are using `tolerance: 0.1`, so the results are a bit different compared to the ones from [stream example](#) which is using the default value of `tolerance`. Note that the nodes 'About', 'Link' and 'Product' now have the same score, while with the default value of `tolerance` the node 'Product' has higher score than the other two.

## Damping Factor

The damping factor configuration parameter accepts values between 0 (inclusive) and 1 (exclusive). If its

value is too high then problems of sinks and spider traps may occur, and the values may oscillate so that the algorithm does not converge. If it's too low then all scores are pushed towards 1, and the result will not sufficiently reflect the structure of the graph.

The following will run the algorithm in `stream` mode using smaller `dampingFactor` value:

```
CALL gds.pageRank.stream('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.05
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 174. Results

name	score
"Home"	1.2487309425844906
"About"	0.9708121818724536
"Links"	0.9708121818724536
"Product"	0.9708121818724536
"Site A"	0.9597081216238426
"Site B"	0.9597081216238426
"Site C"	0.9597081216238426
"Site D"	0.9597081216238426

Compared to the results from the [stream example](#) which is using the default value of `dampingFactor` the score values are closer to each other when using `dampingFactor: 0.05`. Also, note that the nodes 'About', 'Link' and 'Product' now have the same score, while with the default value of `dampingFactor` the node 'Product' has higher score than the other two.

## Personalised PageRank

Personalized PageRank is a variation of PageRank which is biased towards a set of `sourceNodes`. This variant of PageRank is often used as part of [recommender systems](#).

The following examples show how to run PageRank centered around 'Site A'.

The following will run the algorithm and stream results:

```
MATCH (siteA:Page {name: 'Site A'})
CALL gds.pageRank.stream('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  sourceNodes: [siteA]
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 175. Results

name	score
"Home"	0.39902290442518784
"Site A"	0.16890325301726694
"About"	0.11220151747374331
"Links"	0.11220151747374331
"Product"	0.11220151747374331
"Site B"	0.01890325301726691
"Site C"	0.01890325301726691
"Site D"	0.01890325301726691

Comparing these results to the ones from the [stream example](#) (which is not using `sourceNodes` configuration parameter) shows that the 'Site A' node that we used in the `sourceNodes` list now scores second instead of fourth.

### Scaling centrality scores

To normalize the final scores as part of the algorithm execution, one can use the `scaler` configuration parameter. A common scaler is the `L1Norm`, which normalizes each score to a value between 0 and 1. A description of all available scalers can be found in the documentation for the `scaleProperties` procedure.

The following will run the algorithm in `stream` mode and returns normalized results:

```
CALL gds.pageRank.stream('myGraph', {
  scaler: "L1Norm"
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 176. Results

name	score
"Home"	0.4181682554824872
"About"	0.1370975954128506
"Links"	0.1370975954128506
"Product"	0.1370975954128506
"Site A"	0.04263473956974027
"Site B"	0.04263473956974027
"Site C"	0.04263473956974027
"Site D"	0.04263473956974027

Comparing the results with the [stream example](#), we can see that the relative order of scores is the same.

## 7.2.2. Article Rank

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

ArticleRank is a variant of the [Page Rank algorithm](#), which measures the transitive influence of nodes.

Page Rank follows the assumption that relationships originating from low-degree nodes have a higher influence than relationships from high-degree nodes. Article Rank lowers the influence of low-degree nodes by lowering the scores being sent to their neighbors in each iteration.

The Article Rank of a node  $v$  at iteration  $i$  is defined as:

$$ArticleRank_i(v) = (1 - d) + d \sum_{w \in N_{in}(v)} \frac{ArticleRank_{i-1}(w)}{|N_{out}(w)| + \overline{N_{out}}}$$

where,

- $N_{in}(v)$  denotes incoming neighbors and  $N_{out}(v)$  denotes outgoing neighbors of node  $v$ .
- $d$  is a damping factor in  $[0, 1]$ .
- $\overline{N_{out}}$  is the average out-degree

For more information, see [ArticleRank: a PageRank-based alternative to numbers of citations for analysing citation networks](#).

### Considerations

There are some things to be aware of when using the Article Rank algorithm:

- If there are no relationships from within a group of pages to outside the group, then the group is considered a spider trap.
- Rank sink can occur when a network of pages is forming an infinite cycle.
- Dead-ends occur when pages have no outgoing relationship.

Changing the damping factor can help with all the considerations above. It can be interpreted as a probability of a web surfer to sometimes jump to a random page and therefore not getting stuck in sinks.

## Syntax

This section covers the syntax used to execute the Article Rank algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).





Run Article Rank in stream mode on a named graph.

```
CALL gds.articleRank.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  score: Float
```

Table 177. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 178. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 179. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Article Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 180. Results

Name	Type	Description
nodeId	Integer	Node ID.
score	Float	Eigenvector score.

Run Article Rank in stats mode on a named graph.

```
CALL gds.articleRank.stats(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 181. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 182. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 183. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Article Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 184. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <b>centralityDistribution</b> .
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

Run Article Rank in mutate mode on a named graph.

```
CALL gds.articleRank.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  nodePropertiesWritten: Integer,  
  ranIterations: Integer,  
  didConverge: Boolean,  
  createMillis: Integer,  
  computeMillis: Integer,  
  postProcessingMillis: Integer,  
  mutateMillis: Integer,  
  centralityDistribution: Map,  
  configuration: Map
```

Table 185. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 186. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 187. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Article Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.

Name	Type	Default	Optional	Description
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <b>None</b> , <b>MinMax</b> , <b>Max</b> , <b>Mean</b> , <b>Log</b> , <b>L1Norm</b> , <b>L2Norm</b> and <b>StdScore</b> .

Table 188. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <b>centralityDistribution</b> .
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	The number of properties that were written to the in-memory graph.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

Run Article Rank in write mode on a named graph.

```
CALL gds.articleRank.write(
  graphName: String,
  configuration: Map
)
YIELD
  nodePropertiesWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 189. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 190. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 191. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Article Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Name	Type	Default	Optional	Description
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <code>None</code> , <code>MinMax</code> , <code>Max</code> , <code>Mean</code> , <code>Log</code> , <code>L1Norm</code> , <code>L2Norm</code> and <code>StdScore</code> .

Table 192. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <code>centralityDistribution</code> .
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).



Run Article Rank in write mode on an anonymous graph:

```
CALL gds.articleRank.write(
  configuration: Map
)
YIELD
  nodePropertiesWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 193. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 194. Algorithm specific configuration

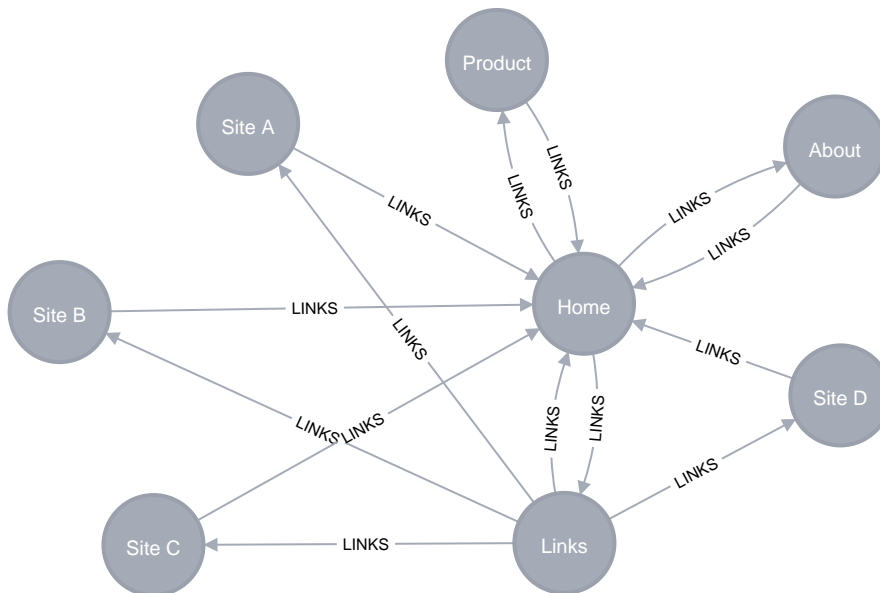
Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation. Must be in [0, 1).
maxIterations	Integer	20	yes	The maximum number of iterations of Article Rank to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable, and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Name	Type	Default	Optional	Description
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <b>None</b> , <b>MinMax</b> , <b>Max</b> , <b>Mean</b> , <b>Log</b> , <b>L1Norm</b> , <b>L2Norm</b> and <b>StdScore</b> .

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Article Rank algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small web network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
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);
```

This graph represents eight pages, linking to one another. Each relationship has a property called `weight`, which describes the importance of the relationship.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Page',
  'LINKS',
  {
    relationshipProperties: 'weight'
  }
)
```

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.articleRank.write.estimate('myGraph', {
  writeProperty: 'centrality',
  maxIterations: 20
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 195. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
8	14	696	696	"696 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the score for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode:

```
CALL gds.articleRank.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 196. Results

name	score
"Home"	0.5607071761939444
"About"	0.250337073634706
"Links"	0.250337073634706
"Product"	0.250337073634706
"Site A"	0.18152391630760797
"Site B"	0.18152391630760797
"Site C"	0.18152391630760797
"Site D"	0.18152391630760797

The above query is running the algorithm in `stream` mode as `unweighted`. Below, one can find an example for [weighted graphs](#).

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning

the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm and return statistics about the centrality scores.

```
CALL gds.articleRank.stats('myGraph')
YIELD centralityDistribution
RETURN centralityDistribution.max AS max
```

Table 197. Results

max
0.5607099533081055

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the score for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.articleRank.mutate('myGraph', {
  mutateProperty: 'centrality'
})
YIELD nodePropertiesWritten, ranIterations
```

Table 198. Results

nodePropertiesWritten	ranIterations
8	19

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the score for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in **write** mode:

```
CALL gds.articleRank.write('myGraph', {
  writeProperty: 'centrality'
})
YIELD nodePropertiesWritten, ranIterations
```

Table 199. Results

nodePropertiesWritten	ranIterations
8	19

## Weighted

By default, the algorithm considers the relationships of the graph to be unweighted. To change this behaviour, we can use the **relationshipWeightProperty** configuration parameter. If the parameter is set, the associated property value is used as relationship weight. In the **weighted** case, the previous score of a node sent to its neighbors is multiplied by the normalized relationship weight. Note, that negative relationship weights are ignored during the computation.

In the following example, we use the **weight** property of the input graph as relationship weight property.

The following will run the algorithm in **stream** mode using relationship weights:

```
CALL gds.articleRank.stream('myGraph', {
  relationshipWeightProperty: 'weight'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 200. Results

name	score
"Home"	0.5160810726222141
"Product"	0.24570958074084706
"About"	0.1819031935802824
"Links"	0.1819031935802824
"Site A"	0.15281123078335393
"Site B"	0.15281123078335393
"Site C"	0.15281123078335393
"Site D"	0.15281123078335393

As in the unweighted example, the "Home" node has the highest score. In contrast, the "Product" now has the second highest instead of the fourth highest score.



We are using **stream** mode to illustrate running the algorithm as **weighted**, however, all the algorithm modes support the **relationshipWeightProperty** configuration parameter.

## Tolerance

The `tolerance` configuration parameter denotes the minimum change in scores between iterations. If all scores change less than the configured tolerance, the iteration is aborted and considered converged. Note, that setting a higher tolerance leads to earlier convergence, but also to less accurate centrality scores.

The following will run the algorithm in `stream` mode using a high `tolerance` value:

```
CALL gds.articleRank.stream('myGraph', {
  tolerance: 0.1
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 201. Results

name	score
"Home"	0.4470707070707072
"About"	0.23000212652844235
"Links"	0.23000212652844235
"Product"	0.23000212652844235
"Site A"	0.16888888888888892
"Site B"	0.16888888888888892
"Site C"	0.16888888888888892
"Site D"	0.16888888888888892

We are using `tolerance: 0.1`, which leads to slightly different results compared to the [stream example](#). However, the computation converges after four iterations, and we can already observe a trend in the resulting scores.

## Personalised Article Rank

Personalized Article Rank is a variation of Article Rank which is biased towards a set of `sourceNodes`. By default, the power iteration starts with the same value for all nodes:  $1 / |V|$ . For a given set of source nodes `S`, the initial value of each source node is set to  $1 / |S|$  and to  $0$  for all remaining nodes.

The following examples show how to run Eigenvector centrality centered around 'Site A' and 'Site B'.

The following will run the algorithm and stream results:

```
MATCH (siteA:Page {name: 'Site A'}), (siteB:Page {name: 'Site B'})
CALL gds.articleRank.stream('myGraph', {
  maxIterations: 20,
  sourceNodes: [siteA, siteB]
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 202. Results

name	score
"Site A"	0.15249052775314756
"Site B"	0.15249052775314756
"Home"	0.1105231342997017
"About"	0.019777824032578193
"Links"	0.019777824032578193
"Product"	0.019777824032578193
"Site C"	0.002490527753147571
"Site D"	0.002490527753147571

Comparing these results to the ones from the [stream example](#) (which is not using `sourceNodes` configuration parameter) shows the 'Site A' and 'Site B' nodes we used in the `sourceNodes` list now score second and third instead of fourth and fifth.

### Scaling centrality scores

To normalize the final scores as part of the algorithm execution, one can use the `scaler` configuration parameter. A common scaler is the `L1Norm`, which normalizes each score to a value between 0 and 1. A description of all available scalers can be found in the documentation for the `scaleProperties` procedure.

The following will run the algorithm in `stream` mode and returns normalized results:

```
CALL gds.articleRank.stream('myGraph', {
  scaler: "L1Norm"
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 203. Results

name	score
"Home"	0.275151294006312
"About"	0.12284588582564794
"Links"	0.12284588582564794
"Product"	0.12284588582564794
"Site A"	0.08907776212918608
"Site B"	0.08907776212918608
"Site C"	0.08907776212918608
"Site D"	0.08907776212918608



Comparing the results with the [stream example](#), we can see that the relative order of scores is the same.

## 7.2.3. Eigenvector Centrality

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

Eigenvector Centrality is an algorithm that measures the **transitive** influence of nodes. Relationships originating from high-scoring nodes contribute more to the score of a node than connections from low-scoring nodes. A high eigenvector score means that a node is connected to many nodes who themselves have high scores.

The algorithm computes the eigenvector associated with the largest absolute eigenvalue. To compute that eigenvalue, the algorithm applies the [power iteration](#) approach. Within each iteration, the centrality score for each node is derived from the scores of its incoming neighbors. In the power iteration method, the eigenvector is L2-normalized after each iteration, leading to normalized results by default.

The [PageRank](#) algorithm is a variant of Eigenvector Centrality with an additional jump probability.

### Considerations

There are some things to be aware of when using the Eigenvector centrality algorithm:

- Centrality scores for nodes with no incoming relationships will converge to  $0$ .
- Due to missing degree normalization, high-degree nodes have a very strong influence on their neighbors' score.

### Syntax

This section covers the syntax used to execute the Eigenvector Centrality algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Eigenvector Centrality in stream mode on a named graph.

```
CALL gds.eigenvector.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  score: Float
```

Table 204. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 205. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 206. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	20	yes	The maximum number of iterations of Eigenvector Centrality to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 207. Results

Name	Type	Description
nodeId	Integer	Node ID.

Name	Type	Description
score	Float	Eigenvector score.

Run Eigenvector Centrality in stats mode on a named graph.

```
CALL gds.eigenvector.stats(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 208. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 209. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 210. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	20	yes	The maximum number of iterations of Eigenvector Centrality to run.
tolerance	Float	0.000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 211. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <b>centralityDistribution</b> .
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

Run Eigenvector Centrality in mutate mode on a named graph.

```
CALL gds.eigenvector.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  nodePropertiesWritten: Integer,  
  ranIterations: Integer,  
  didConverge: Boolean,  
  createMillis: Integer,  
  computeMillis: Integer,  
  postProcessingMillis: Integer,  
  mutateMillis: Integer,  
  centralityDistribution: Map,  
  configuration: Map
```

Table 212. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 213. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 214. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	20	yes	The maximum number of iterations of Eigenvector Centrality to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

Table 215. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <b>centralityDistribution</b> .
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	The number of properties that were written to the in-memory graph.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.



Run Eigenvector Centrality in write mode on a named graph.

```
CALL gds.eigenvector.write(
  graphName: String,
  configuration: Map
)
YIELD
  nodePropertiesWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 216. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 217. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 218. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	20	yes	The maximum number of iterations of Eigenvector Centrality to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.

Name	Type	Default	Optional	Description
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are <b>None</b> , <b>MinMax</b> , <b>Max</b> , <b>Mean</b> , <b>Log</b> , <b>L1Norm</b> , <b>L2Norm</b> and <b>StdScore</b> .

Table 219. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the <b>centralityDistribution</b> .
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the **write** mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Eigenvector Centrality in write mode on an anonymous graph:

```
CALL gds.eigenvector.write(
  configuration: Map
)
YIELD
  nodePropertiesWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  centralityDistribution: Map,
  configuration: Map
```

Table 220. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 221. Algorithm specific configuration

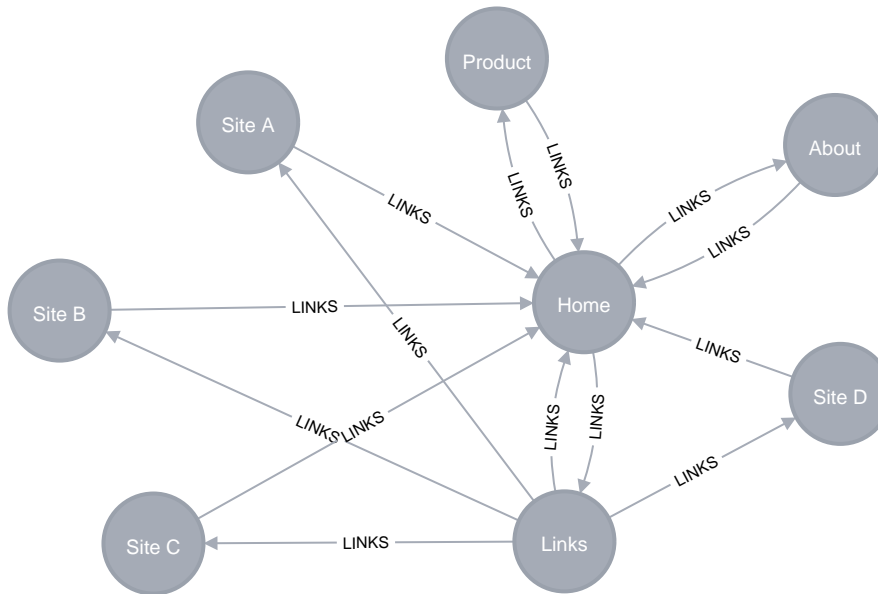
Name	Type	Default	Optional	Description
maxIterations	Integer	20	yes	The maximum number of iterations of Eigenvector Centrality to run.
tolerance	Float	0.0000001	yes	Minimum change in scores between iterations. If all scores change less than the tolerance value the result is considered stable and the algorithm returns.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
sourceNodes	List or Node or Number	[]	yes	The nodes or node ids to use for computing Personalized Page Rank.
scaler	String	None	yes	The name of the scaler applied for the final scores. Supported values are None, MinMax, Max, Mean, Log, L1Norm, L2Norm and StdScore.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Eigenvector Centrality algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the

algorithm in a real setting. We will do this on a small web network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
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);
```

This graph represents eight pages, linking to one another. Each relationship has a property called `weight`, which describes the importance of the relationship.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(  
  'myGraph',  
  'Page',  
  'LINKS',  
  {  
    relationshipProperties: 'weight'  
  }  
)
```

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.eigenvector.write.estimate('myGraph', {  
  writeProperty: 'centrality',  
  maxIterations: 20  
})  
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 222. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
8	14	696	696	"696 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the score for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode:

```
CALL gds.eigenvector.stream('myGraph')  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score  
ORDER BY score DESC, name ASC
```

Table 223. Results

name	score
"Home"	0.7465574981728249
"About"	0.33997520529777137
"Links"	0.33997520529777137
"Product"	0.33997520529777137
"Site A"	0.15484062876886298
"Site B"	0.15484062876886298
"Site C"	0.15484062876886298
"Site D"	0.15484062876886298

The above query is running the algorithm in `stream` mode as `unweighted`. Below, one can find an example for [weighted graphs](#).

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm and return statistics about the centrality scores.

```
CALL gds.eigenvector.stats('myGraph', {
  maxIterations: 20
})
YIELD centralityDistribution
RETURN centralityDistribution.max AS max
```

Table 224. Results

max
0.7465581893920898

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the score for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.eigenvector.mutate('myGraph', {
  maxIterations: 20,
  mutateProperty: 'centrality'
})
YIELD nodePropertiesWritten, ranIterations
```

Table 225. Results

nodePropertiesWritten	ranIterations
8	20

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the score for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.eigenvector.write('myGraph', {
  maxIterations: 20,
  writeProperty: 'centrality'
})
YIELD nodePropertiesWritten, ranIterations
```

Table 226. Results

nodePropertiesWritten	ranIterations
8	20

## Weighted

By default, the algorithm considers the relationships of the graph to be unweighted. To change this behaviour, we can use the `relationshipWeightProperty` configuration parameter. If the parameter is set, the associated property value is used as relationship weight. In the `weighted` case, the previous score of a node sent to its neighbors is multiplied by the normalized relationship weight. Note, that negative relationship weights are ignored during the computation.

In the following example, we use the `weight` property of the input graph as relationship weight property.

The following will run the algorithm in `stream` mode using relationship weights:

```
CALL gds.eigenvector.stream('myGraph', {
  maxIterations: 20,
  relationshipWeightProperty: 'weight'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 227. Results

name	score
"Home"	0.8328163407319487
"Product"	0.5004775834976313
"About"	0.1668258611658771
"Links"	0.1668258611658771
"Site A"	0.008327591469710233
"Site B"	0.008327591469710233
"Site C"	0.008327591469710233
"Site D"	0.008327591469710233

As in the unweighted example, the "Home" node has the highest score. In contrast, the "Product" now has the second highest instead of the fourth highest score.



We are using `stream` mode to illustrate running the algorithm as `weighted`, however, all the algorithm modes support the `relationshipWeightProperty` configuration parameter.

## Tolerance

The `tolerance` configuration parameter denotes the minimum change in scores between iterations. If all scores change less than the configured tolerance, the iteration is aborted and considered converged. Note, that setting a higher tolerance leads to earlier convergence, but also to less accurate centrality scores.

The following will run the algorithm in `stream` mode using a high `tolerance` value:

```
CALL gds.eigenvector.stream('myGraph', {
  maxIterations: 20,
  tolerance: 0.1
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 228. Results

name	score
"Home"	0.7108273818583551



name	score
"About"	0.3719400001993262
"Links"	0.3719400001993262
"Product"	0.3719400001993262
"Site A"	0.14116155811301126
"Site B"	0.14116155811301126
"Site C"	0.14116155811301126
"Site D"	0.14116155811301126

We are using `tolerance: 0.1`, which leads to slightly different results compared to the [stream example](#). However, the computation converges after three iterations, and we can already observe a trend in the resulting scores.

### Personalised Eigenvector Centrality

Personalized Eigenvector Centrality is a variation of Eigenvector Centrality which is biased towards a set of `sourceNodes`. By default, the power iteration starts with the same value for all nodes:  $1 / |V|$ . For a given set of source nodes `S`, the initial value of each source node is set to  $1 / |S|$  and to  $0$  for all remaining nodes.

The following examples show how to run Eigenvector centrality centered around 'Site A'.

The following will run the algorithm and stream results:

```
MATCH (siteA:Page {name: 'Site A'}), (siteB:Page {name: 'Site B'})
CALL gds.eigenvector.stream('myGraph', {
  maxIterations: 20,
  sourceNodes: [siteA, siteB]
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 229. Results

name	score
"Home"	0.7465645391567868
"About"	0.33997203172449453
"Links"	0.33997203172449453
"Product"	0.33997203172449453
"Site A"	0.15483736775159632
"Site B"	0.15483736775159632
"Site C"	0.15483736775159632
"Site D"	0.15483736775159632

## Scaling centrality scores

Internally, centrality scores are scaled after each iteration using L2 normalization. As a consequence, the final values are already normalized. This behavior cannot be changed as it is part of the power iteration method.

However, to normalize the final scores as part of the algorithm execution, one can use the `scaler` configuration parameter. A common scaler is the `L1Norm`, which normalizes each score to a value between 0 and 1. A description of all available scalers can be found in the documentation for the `scaleProperties` procedure.

The following will run the algorithm in `stream` mode and returns normalized results:

```
CALL gds.eigenvector.stream('myGraph', {
  scaler: "L1Norm"
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 230. Results

name	score
"Home"	0.31291106560043064
"About"	0.1424967320371402
"Links"	0.1424967320371402
"Product"	0.1424967320371402
"Site A"	0.06489968457203725
"Site B"	0.06489968457203725
"Site C"	0.06489968457203725
"Site D"	0.06489968457203725

Comparing the results with the [stream example](#), we can see that the relative order of scores is the same.

## 7.2.4. Betweenness Centrality

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

## Introduction

Betweenness centrality is a way of detecting the amount of influence a node has over the flow of information in a graph. It is often used to find nodes that serve as a bridge from one part of a graph to another.

The algorithm calculates unweighted shortest paths between all pairs of nodes in a graph. Each node receives a score, based on the number of shortest paths that pass through the node. Nodes that more frequently lie on shortest paths between other nodes will have higher betweenness centrality scores.

The GDS implementation is based on [Brandes' approximate algorithm](#) for unweighted graphs. The implementation requires  $O(n + m)$  space and runs in  $O(n * m)$  time, where  $n$  is the number of nodes and  $m$  the number of relationships in the graph.

For more information on this algorithm, see:

- [A Faster Algorithm for Betweenness Centrality](#)
- [Centrality Estimation in Large Networks](#)
- [A Set of Measures of Centrality Based on Betweenness](#)



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

## Considerations and sampling

The Betweenness Centrality algorithm can be very resource-intensive to compute. [Brandes' approximate algorithm](#) computes single-source shortest paths (SSSP) for a set of source nodes. When all nodes are selected as source nodes, the algorithm produces an exact result. However, for large graphs this can potentially lead to very long runtimes. Thus, approximating the results by computing the SSSPs for only a subset of nodes can be useful. In GDS we refer to this technique as *sampling*, where the size of the source node set is the *sampling size*.

There are two things to consider when executing the algorithm on large graphs:

- A higher parallelism leads to higher memory consumption as each thread executes SSSPs for a subset of source nodes sequentially.
  - In the worst case, a single SSSP requires the whole graph to be duplicated in memory.
- A higher sampling size leads to more accurate results, but also to a potentially much longer execution time.

Changing the values of the configuration parameters `concurrency` and `samplingSize`, respectively, can help to manage these considerations.

## Sampling strategies

Brandes defines several strategies for selecting source nodes. The GDS implementation is based on the random degree selection strategy, which selects nodes with a probability proportional to their degree. The

idea behind this strategy is that such nodes are likely to lie on many shortest paths in the graph and thus have a higher contribution to the betweenness centrality score.

## Syntax

This section covers the syntax used to execute the Betweenness Centrality algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

Run Betweenness Centrality in stream mode on a named graph.

```
CALL gds.betweenness.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  score: Float
```

Table 231. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 232. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 233. Algorithm specific configuration

Name	Type	Default	Optional	Description
samplingSize	Integer	node count	yes	The number of source nodes to consider for computing centrality scores.
samplingSeed	Integer	null	yes	The seed value for the random number generator that selects start nodes.

Table 234. Results

Name	Type	Description
nodeId	Integer	Node ID.
score	Float	Betweenness Centrality score.

Run Betweenness Centrality in stats mode on a named graph.

```
CALL gds.betweenness.stats(
  graphName: String,
  configuration: Map
)
YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  configuration: Map
```

Table 235. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 236. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 237. Algorithm specific configuration

Name	Type	Default	Optional	Description
samplingSize	Integer	node count	yes	The number of source nodes to consider for computing centrality scores.
samplingSeed	Integer	null	yes	The seed value for the random number generator that selects start nodes.

Table 238. Results

Name	Type	Description
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the statistics.

Name	Type	Description
configuration	Map	Configuration used for running the algorithm.

Run Betweenness Centrality in mutate mode on a named graph.

```
CALL gds.betweenness.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 239. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 240. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 241. Algorithm specific configuration

Name	Type	Default	Optional	Description
samplingSize	Integer	node count	yes	The number of source nodes to consider for computing centrality scores.
samplingSeed	Integer	null	yes	The seed value for the random number generator that selects start nodes.

Table 242. Results

Name	Type	Description
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the statistics.



Name	Type	Description
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	Number of properties added to the in-memory graph.
configuration	Map	Configuration used for running the algorithm.

Run Betweenness Centrality in write mode on a named graph.

```
CALL gds.betweenness.write(
  graphName: String,
  configuration: Map
)
YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 243. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 244. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 245. Algorithm specific configuration

Name	Type	Default	Optional	Description
samplingSize	Integer	node count	yes	The number of source nodes to consider for computing centrality scores.
samplingSeed	Integer	null	yes	The seed value for the random number generator that selects start nodes.

Table 246. Results

Name	Type	Description
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the statistics.
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	Number of properties written to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run *Betweenness Centrality* in `write` mode on an anonymous graph:

```
CALL gds.betweenness.write(
  configuration: Map
)
YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 247. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.

Name	Type	Default	Optional	Description
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	<code>null</code>	yes	The relationship properties to project during anonymous graph creation.
<code>concurrency</code>	Integer	<code>4</code>	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
<code>readConcurrency</code>	Integer	<code>value of 'concurrency'</code>	yes	The number of concurrent threads used for creating the graph.
<code>writeConcurrency</code>	Integer	<code>value of 'concurrency'</code>	yes	The number of concurrent threads used for writing the result to Neo4j.

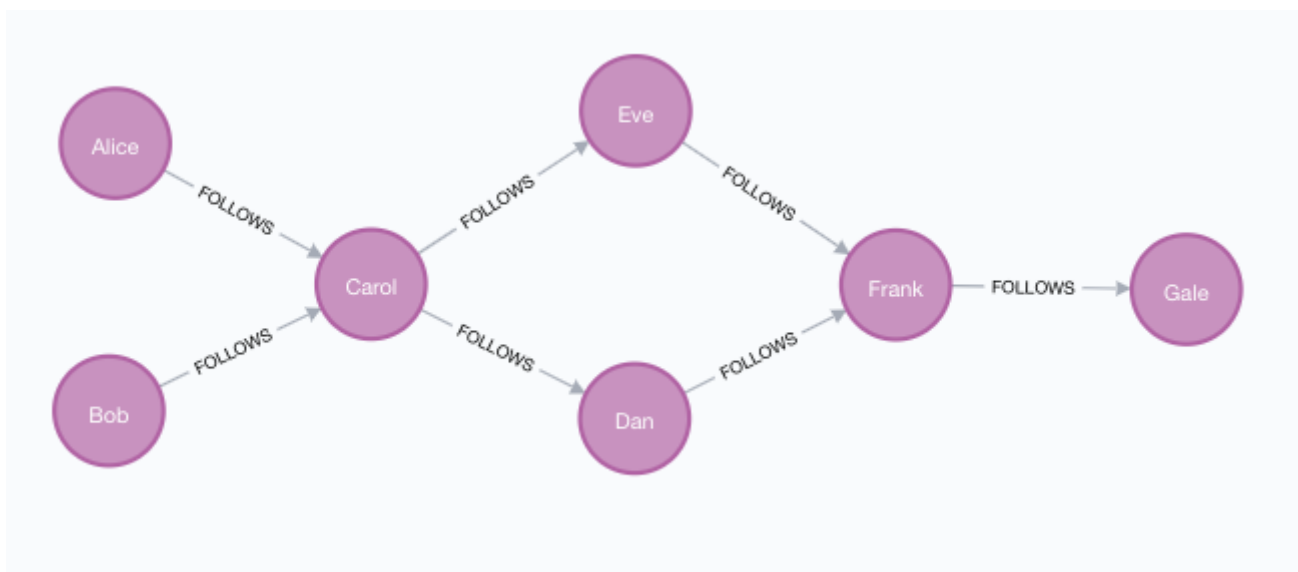
Table 248. Algorithm specific configuration

Name	Type	Default	Optional	Description
<code>samplingSize</code>	Integer	<code>node count</code>	yes	The number of source nodes to consider for computing centrality scores.
<code>samplingSeed</code>	Integer	<code>null</code>	yes	The seed value for the random number generator that selects start nodes.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Betweenness Centrality algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(alice:User {name: 'Alice'}),
(bob:User {name: 'Bob'}),
(carol:User {name: 'Carol'}),
(dan:User {name: 'Dan'}),
(eve:User {name: 'Eve'}),
(frank:User {name: 'Frank'}),
(gale:User {name: 'Gale'}),

(alice)-[:FOLLOWS]->(carol),
(bob)-[:FOLLOWS]->(carol),
(carol)-[:FOLLOWS]->(dan),
(carol)-[:FOLLOWS]->(eve),
(dan)-[:FOLLOWS]->(frank),
(eve)-[:FOLLOWS]->(frank),
(frank)-[:FOLLOWS]->(gale);
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `User` nodes and the `FOLLOWS` relationships.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create('myGraph', 'User', 'FOLLOWS')
```

In the following examples we will demonstrate using the Betweenness Centrality algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.betweenness.write.estimate('myGraph', { writeProperty: 'betweenness' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 249. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
7	7	2912	2912	"2912 Bytes"

As is discussed in [Considerations and sampling](#) we can configure the memory requirements using the

`concurrency` configuration parameter.

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.betweenness.write.estimate('myGraph', { writeProperty: 'betweenness', concurrency: 1 })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 250. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
7	7	848	848	"848 Bytes"

Here we can note that the estimated memory requirements were lower than when running with the default concurrency setting. Similarly, using a higher value will increase the estimated memory requirements.

## Stream

In the `stream` execution mode, the algorithm returns the centrality for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode:

```
CALL gds.betweenness.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY name ASC
```

Table 251. Results

name	score
"Alice"	0.0
"Bob"	0.0
"Carol"	8.0
"Dan"	3.0
"Eve"	3.0
"Frank"	5.0
"Gale"	0.0

We note that the 'Carol' node has the highest score, followed by the 'Frank' node. Studying the [example graph](#) we can see that these nodes are in bottleneck positions in the graph. The 'Carol' node connects the 'Alice' and 'Bob' nodes to all other nodes, which increases its score. In particular, the shortest path from 'Alice' or 'Bob' to any other reachable node passes through 'Carol'. Similarly, all shortest paths that lead to the 'Gale' node passes through the 'Frank' node. Since 'Gale' is reachable from each other node, this causes the score for 'Frank' to be high.

Conversely, there are no shortest paths that pass through either of the nodes 'Alice', 'Bob' or 'Gale' which causes their betweenness centrality score to be zero.

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm in `stats` mode:

```
CALL gds.betweenness.stats('myGraph')
YIELD centralityDistribution
RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore
```

Table 252. Results

minimumScore	meanScore
0.0	2.714292253766741

Comparing this to the results we saw in the [stream example](#), we can find our minimum and maximum values from the table. It is worth noting that unless the graph has a particular shape involving a directed cycle, the minimum score will almost always be zero.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the centrality for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.betweenness.mutate('myGraph', { mutateProperty: 'betweenness' })
YIELD centralityDistribution, nodePropertiesWritten
RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore,
nodePropertiesWritten
```

Table 253. Results

minimumScore	meanScore	nodePropertiesWritten
0.0	2.714292253766741	7

The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `betweenness` which stores the betweenness centrality score for each node. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs](#).

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the centrality for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.betweenness.write('myGraph', { writeProperty: 'betweenness' })
YIELD centralityDistribution, nodePropertiesWritten
RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore,
nodePropertiesWritten
```

Table 254. Results

minimumScore	meanScore	nodePropertiesWritten
0.0	2.714292253766741	7

The returned result is the same as in the `stats` example. Additionally, each of the seven nodes now has a new property `betweenness` in the Neo4j database, containing the betweenness centrality score for that node.

## Sampling

Betweenness Centrality can be very resource-intensive to compute. To help with this, it is possible to approximate the results using a sampling technique. The configuration parameters `samplingSize` and `samplingSeed` are used to control the sampling. We illustrate this on our example graph by approximating Betweenness Centrality with a sampling size of two. The seed value is an arbitrary integer, where using the same value will yield the same results between different runs of the procedure.

The following will run the algorithm in `stream` mode with a sampling size of two:

```
CALL gds.betweenness.stream('myGraph', {samplingSize: 2, samplingSeed: 0})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY name ASC
```

Table 255. Results

name	score
"Alice"	0.0
"Bob"	0.0



name	score
"Carol"	4.0
"Dan"	2.0
"Eve"	2.0
"Frank"	2.0
"Gale"	0.0

Here we can see that the 'Carol' node has the highest score, followed by a three-way tie between the 'Dan', 'Eve', and 'Frank' nodes. We are only sampling from two nodes, where the probability of a node being picked for the sampling is proportional to its outgoing degree. The 'Carol' node has the maximum degree and is the most likely to be picked. The 'Gale' node has an outgoing degree of zero and is very unlikely to be picked. The other nodes all have the same probability to be picked.

With our selected sampling seed of 0, we seem to have selected either of the 'Alice' and 'Bob' nodes, as well as the 'Carol' node. We can see that because either of 'Alice' and 'Bob' would add four to the score of the 'Carol' node, and each of 'Alice', 'Bob', and 'Carol' adds one to all of 'Dan', 'Eve', and 'Frank'.

To increase the accuracy of our approximation, the sampling size could be increased. In fact, setting the `samplingSize` to the node count of the graph (seven, in our case) will produce exact results.

## Undirected

Betweenness Centrality can also be run on undirected graphs. To illustrate this, we will project our example graph using the `UNDIRECTED` orientation.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myUndirectedGraph'.

```
CALL gds.graph.create('myUndirectedGraph', 'User', {FOLLOWS: {orientation: 'UNDIRECTED'}})
```

Now we can run Betweenness Centrality on our undirected graph. The algorithm automatically figures out that the graph is undirected.



Running the algorithm on an undirected graph is about twice as computationally intensive compared to a directed graph.

The following will run the algorithm in `stream` mode on the undirected graph:

```
CALL gds.betweenness.stream('myUndirectedGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY name ASC
```

Table 256. Results

name	score
"Alice"	0.0

name	score
"Bob"	0.0
"Carol"	9.5
"Dan"	3.0
"Eve"	3.0
"Frank"	5.5
"Gale"	0.0

The central nodes now have slightly higher scores, due to the fact that there are more shortest paths in the graph, and these are more likely to pass through the central nodes. The 'Dan' and 'Eve' nodes retain the same centrality scores as in the directed case.

## 7.2.5. Degree Centrality

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

The Degree Centrality algorithm can be used to find popular nodes within a graph. Degree centrality measures the number of incoming or outgoing (or both) relationships from a node, depending on the orientation of a relationship projection. For more information on relationship orientations, see the [relationship projection syntax section](#). It can be applied to either weighted or unweighted graphs. In the weighted case the algorithm computes the sum of all positive weights of adjacent relationships of a node, for each node in the graph. Non-positive weights are ignored.

For more information on this algorithm, see:

- [Linton C. Freeman: Centrality in Social Networks Conceptual Clarification, 1979.](#)

### Use-cases

The Degree Centrality algorithm has been shown to be useful in many different applications. For example:

- Degree centrality is an important component of any attempt to determine the most important people in a social network. For example, in BrandWatch's [most influential men and women on Twitter 2017](#) the top 5 people in each category have over 40m followers each, which is a lot higher than the average degree.

- Weighted degree centrality has been used to help separate fraudsters from legitimate users of an online auction. The weighted centrality for fraudsters is significantly higher because they tend to collude with each other to artificially increase the price of items. Read more in [Two Step graph-based semi-supervised Learning for Online Auction Fraud Detection](#)

## Syntax

This section covers the syntax used to execute the Degree Centrality algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

Run Degree Centrality in stream mode on a named graph.

```
CALL gds.degree.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  score: Float
```

Table 257. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 258. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 259. Algorithm specific configuration

Name	Type	Default	Optional	Description
orientation	String	NATURAL	yes	The orientation used to compute node degrees. Supported orientations are NATURAL, REVERSE and UNDIRECTED.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

Table 260. Results

Name	Type	Description
nodeId	Integer	Node ID.
score	Float	Degree Centrality score.

Run Degree Centrality in stats mode on a named graph.

```
CALL gds.degree.stats(
  graphName: String,
  configuration: Map
) YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  configuration: Map
```

Table 261. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 262. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 263. Algorithm specific configuration

Name	Type	Default	Optional	Description
orientation	String	NATURAL	yes	The orientation used to compute node degrees. Supported orientations are NATURAL, REVERSE and UNDIRECTED.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

Table 264. Results

Name	Type	Description
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the statistics.

Name	Type	Description
configuration	Map	Configuration used for running the algorithm.

Run Degree Centrality in mutate mode on a named graph.

```
CALL gds.degree.mutate(  
  graphName: String,  
  configuration: Map  
) YIELD  
  centralityDistribution: Map,  
  createMillis: Integer,  
  computeMillis: Integer,  
  postProcessingMillis: Integer,  
  mutateMillis: Integer,  
  nodePropertiesWritten: Integer,  
  configuration: Map
```

Table 265. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 266. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 267. Algorithm specific configuration

Name	Type	Default	Optional	Description
orientation	String	NATURAL	yes	The orientation used to compute node degrees. Supported orientations are NATURAL, REVERSE and UNDIRECTED.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

Table 268. Results

Name	Type	Description
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

Name	Type	Description
postProcessingMillis	Integer	Milliseconds for computing the statistics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	Number of properties added to the in-memory graph.
configuration	Map	Configuration used for running the algorithm.



Run Degree Centrality in write mode on a named graph.

```
CALL gds.degree.write(
  graphName: String,
  configuration: Map
) YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 269. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 270. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 271. Algorithm specific configuration

Name	Type	Default	Optional	Description
orientation	String	NATURAL	yes	The orientation used to compute node degrees. Supported orientations are NATURAL, REVERSE and UNDIRECTED.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

Table 272. Results

Name	Type	Description
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the statistics.
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	Number of properties written to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Degree Centrality in write mode on an anonymous graph:

```
CALL gds.degree.write(
  configuration: Map
) YIELD
  centralityDistribution: Map,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 273. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.

Name	Type	Default	Optional	Description
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

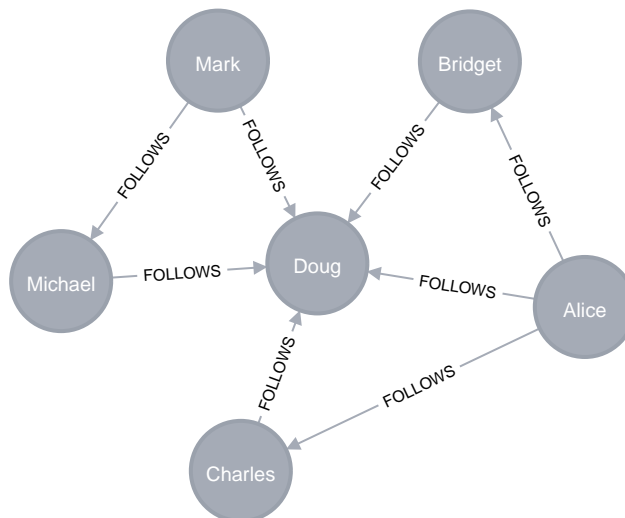
Table 274. Algorithm specific configuration

Name	Type	Default	Optional	Description
orientation	String	NATURAL	yes	The orientation used to compute node degrees. Supported orientations are NATURAL, REVERSE and UNDIRECTED.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted degree computation. If unspecified, the algorithm runs unweighted.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Degree Centrality algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
  (alice:User {name: 'Alice'}),
  (bridget:User {name: 'Bridget'}),
  (charles:User {name: 'Charles'}),
  (doug:User {name: 'Doug'}),
  (mark:User {name: 'Mark'}),
  (michael:User {name: 'Michael'}),

  (alice)-[:FOLLOWS {score: 1}]->(doug),
  (alice)-[:FOLLOWS {score: -2}]->(bridget),
  (alice)-[:FOLLOWS {score: 5}]->(charles),
  (mark)-[:FOLLOWS {score: 1.5}]->(doug),
  (mark)-[:FOLLOWS {score: 4.5}]->(michael),
  (bridget)-[:FOLLOWS {score: 1.5}]->(doug),
  (charles)-[:FOLLOWS {score: 2}]->(doug),
  (michael)-[:FOLLOWS {score: 1.5}]->(doug)
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `User` nodes and the `FOLLOWS` relationships.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a reverse projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'User',
  {
    FOLLOWS: {
      orientation: 'REVERSE',
      properties: ['score']
    }
  }
)
```

The graph is projected in a `REVERSE` orientation in order to retrieve people with the most followers in the following examples. This will be demonstrated using the Degree Centrality algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.degree.write.estimate('myGraph', { writeProperty: 'degree' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 275. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	8	40	40	"40 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the degree centrality for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode:

```
CALL gds.degree.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score AS followers
ORDER BY followers DESC, name DESC
```

Table 276. Results

name	followers
"Doug"	5.0
"Michael"	1.0
"Charles"	1.0
"Bridget"	1.0
"Mark"	0.0
"Alice"	0.0

We can see that Doug is the most popular user in our imaginary social network graph, with 5 followers - all other users follow them, but they don't follow anybody back. In a real social network, celebrities have very high follower counts but tend to follow only very few people. We could therefore consider Doug quite the celebrity!

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm in `stats` mode:

```
CALL gds.degree.stats('myGraph')
YIELD centralityDistribution
RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore
```

Table 277. Results

minimumScore	meanScore
0.0	1.3333358764648438

Comparing this to the results we saw in the [stream example](#), we can find our minimum and mean values from the table.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the degree centrality for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.degree.mutate('myGraph', { mutateProperty: 'degree' })
YIELD centralityDistribution, nodePropertiesWritten
RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore,
nodePropertiesWritten
```

Table 278. Results

minimumScore	meanScore	nodePropertiesWritten
0.0	1.3333358764648438	6

The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `degree` which stores the degree centrality score for each node. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs in the catalog](#).

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the degree centrality for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in **write** mode:

```
CALL gds.degree.write('myGraph', { writeProperty: 'degree' })
YIELD centralityDistribution, nodePropertiesWritten
RETURN centralityDistribution.min AS minimumScore, centralityDistribution.mean AS meanScore,
nodePropertiesWritten
```

Table 279. Results

minimumScore	meanScore	nodePropertiesWritten
0.0	1.3333358764648438	6

The returned result is the same as in the **stats** example. Additionally, each of the seven nodes now has a new property **degree** in the Neo4j database, containing the degree centrality score for that node.

## Weighted Degree Centrality example

This example will explain the weighted Degree Centrality algorithm. This algorithm is a variant of the Degree Centrality algorithm, that measures the sum of positive weights of incoming and outgoing relationships.

The following will run the algorithm in **stream** mode, showing which users have the highest weighted degree centrality:

```
CALL gds.degree.stream(
  'myGraph',
  { relationshipWeightProperty: 'score' }
)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score AS weightedFollowers
ORDER BY weightedFollowers DESC, name DESC
```

Table 280. Results

name	weightedFollowers
"Doug"	7.5
"Charles"	5.0
"Michael"	4.5
"Mark"	0.0
"Bridget"	0.0
"Alice"	0.0

Doug still remains our most popular user, but there isn't such a big gap to the next person. Charles and Michael both only have one follower, but those relationships have a high relationship weight. Note that Bridget also has a weighted score of 0.0, despite having a connection from Alice. That is because the **score** property value between Bridget and Alice is negative and will be ignored by the algorithm.

## Setting an orientation

By default, node centrality uses the **NATURAL** orientation to compute degrees. For some use-cases it makes

sense to analyze a different orientation, for example, if we want to find out how many users follow another user. In order to change the orientation, we can use the `orientation` configuration key. Supported values are `NATURAL` (default), `REVERSE` and `UNDIRECTED`.

The following will run the algorithm in `stream` mode, showing which users have the highest in-degree centrality using the reverse orientation of the relationships:

```
CALL gds.degree.stream(  
  'myGraph',  
  { orientation: 'REVERSE' }  
)  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score AS followees  
ORDER BY followees DESC, name DESC
```

Table 281. Results

name	followees
"Alice"	3.0
"Mark"	2.0
"Michael"	1.0
"Charles"	1.0
"Bridget"	1.0
"Doug"	0.0

The example shows that when looking at the reverse orientation, `Alice` is more central in the network than `Doug`.

## 7.2.6. Closeness Centrality Alpha

Closeness centrality is a way of detecting nodes that are able to spread information very efficiently through a graph.

The closeness centrality of a node measures its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances to all other nodes.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

For each node  $u$ , the Closeness Centrality algorithm calculates the sum of its distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then inverted to determine the closeness centrality score for that node.

The raw closeness centrality of a node  $u$  is calculated using the following formula:

$$\text{raw closeness centrality}(u) = 1 / \text{sum}(\text{distance from } u \text{ to all other nodes})$$

It is more common to normalize this score so that it represents the average length of the shortest paths rather than their sum. This adjustment allow comparisons of the closeness centrality of nodes of graphs of



different sizes

The formula for **normalized closeness centrality** of node  $u$  is as follows:

$$\text{normalized closeness centrality}(u) = (\text{number of nodes} - 1) / \text{sum}(\text{distance from } u \text{ to all other nodes})$$

Wasserman and Faust have proposed an improved formula for dealing with unconnected graphs. Assuming that  $n_u$  is the number of nodes reachable from  $u$  (counting also itself), their corrected formula for a given node  $u$  is given as follows

$$\text{Wasserman-Faust normalized closeness centrality}(u) = (n-1)^2 / (\text{number of nodes} - 1) * \text{sum}(\text{distance from } u \text{ to all other nodes})$$

Note that in the case of a directed graph, closeness centrality is defined alternatively. That is, rather than considering distances from  $u$  to every other node, we instead sum and average the distance from every other node to  $u$ .

## Use-cases - when to use the Closeness Centrality algorithm

- Closeness centrality is used to research organizational networks, where individuals with high closeness centrality are in a favourable position to control and acquire vital information and resources within the organization. One such study is "[Mapping Networks of Terrorist Cells](#)" by Valdis E. Krebs.
- Closeness centrality can be interpreted as an estimated time of arrival of information flowing through telecommunications or package delivery networks where information flows through shortest paths to a predefined target. It can also be used in networks where information spreads through all shortest paths simultaneously, such as infection spreading through a social network. Find more details in "[Centrality and network flow](#)" by Stephen P. Borgatti.
- Closeness centrality has been used to estimate the importance of words in a document, based on a graph-based keyphrase extraction process. This process is described by Florian Boudin in "[A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction](#)".

## Constraints - when not to use the Closeness Centrality algorithm

- Academically, closeness centrality works best on connected graphs. If we use the original formula on an unconnected graph, we can end up with an infinite distance between two nodes in separate connected components. This means that we'll end up with an infinite closeness centrality score when we sum up all the distances from that node.

In practice, a variation on the original formula is used so that we don't run into these issues.

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.closeness.write(configuration: Map)
YIELD nodes, createMillis, computeMillis, writeMillis, centralityDistribution
```

Table 282. Parameters

Name	Type	Default	Optional	Description
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurr ency	int	value of 'concurr ency'	yes	The number of concurrent threads used for reading the graph.
writeConcur rency	int	value of 'concurr ency'	yes	The number of concurrent threads used for writing the result.
writePropert y	string	'centrality'	yes	The property name written back to.
improved	boolean	false	yes	Denotes whether the Wasserman-Faust formula is used.

Table 283. Results

Name	Type	Description
nodes	int	The number of nodes considered.
createMillis	int	Milliseconds for loading data.
computeMillis	int	Milliseconds for running the algorithm.
writeMillis	int	Milliseconds for writing result data back.
writeProperty	string	The property name written back to.
centralityDistr ibution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.

The following will run the algorithm and stream results:

```
CALL gds.alpha.closeness.stream(configuration: Map)
YIELD nodeId, centrality
```

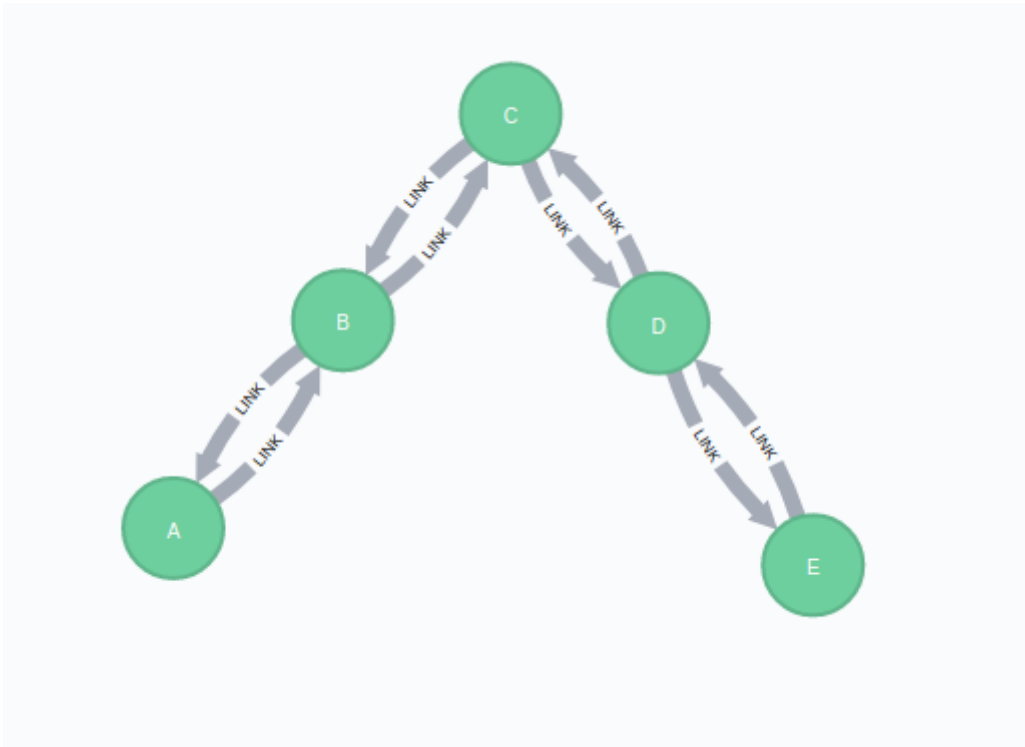
Table 284. Parameters

Name	Type	Default	Optional	Description
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurr ency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

Table 285. Results

Name	Type	Description
node	long	Node ID
centrality	float	Closeness centrality score

## Closeness Centrality algorithm sample



The following will create a sample graph:

```

CREATE (a:Node{id:"A"}),
      (b:Node{id:"B"}),
      (c:Node{id:"C"}),
      (d:Node{id:"D"}),
      (e:Node{id:"E"}),
      (a)-[:LINK]->(b),
      (b)-[:LINK]->(a),
      (b)-[:LINK]->(c),
      (c)-[:LINK]->(b),
      (c)-[:LINK]->(d),
      (d)-[:LINK]->(c),
      (d)-[:LINK]->(e),
      (e)-[:LINK]->(d);
  
```

The following will run the algorithm and stream results:

```

CALL gds.alpha.closeness.stream({
  nodeProjection: 'Node',
  relationshipProjection: 'LINK'
})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).name AS user, centrality
ORDER BY centrality DESC
  
```

Table 286. Results

Name	Centrality weight
C	0.6666666666666666
B	0.5714285714285714
D	0.5714285714285714
A	0.4

Name	Centrality weight
E	0.4

C is the best connected node in this graph, although B and D aren't far behind. A and E don't have close ties to many other nodes, so their scores are lower. Any node that has a direct connection to all other nodes would score 1.

The following will run the algorithm and write back results:

```
CALL gds.alpha.closeness.write({
  nodeProjection: 'Node',
  relationshipProjection: 'LINK',
  writeProperty: 'centrality'
}) YIELD nodes, writeProperty
```

Table 287. Results

nodes	writeProperty
5	"centrality"

## Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the [Creating graphs using Cypher](#) section of the manual.

```
CALL gds.alpha.closeness.write({
  nodeQuery: 'MATCH (p:Node) RETURN id(p) AS id',
  relationshipQuery: 'MATCH (p1:Node)-[:LINK]->(p2:Node) RETURN id(p1) AS source, id(p2) AS target'
}) YIELD nodes, writeProperty
```

Table 288. Results

nodes	writeProperty
5	"centrality"

Calculation:

- count farness in each msbfs-callback
- divide by N-1

$N = 5$  // number of nodes

$k = N-1 = 4$  // used for normalization

	A	B	C	D	E	
A	0	1	2	3	4	// fairness between each pair of nodes
B	1	0	1	2	3	
C	2	1	0	1	2	
D	3	2	1	0	1	
E	4	3	2	1	0	
S	10	7	6	7	10	// raw closeness centrality
k/S	0.4	0.57	0.67	0.57	0.4	// normalized closeness centrality

## 7.2.7. Harmonic Centrality Alpha

Harmonic centrality (also known as valued centrality) is a variant of closeness centrality, that was invented to solve the problem the original formula had when dealing with unconnected graphs. As with many of the centrality algorithms, it originates from the field of social network analysis.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

Harmonic centrality was proposed by Marchiori and Latora in [Harmony in the Small World](#) while trying to come up with a sensible notion of "average shortest path".

They suggested a different way of calculating the average distance to that used in the Closeness Centrality algorithm. Rather than summing the distances of a node to all other nodes, the harmonic centrality algorithm sums the inverse of those distances. This enables it deal with infinite values.

The raw harmonic centrality for a node is calculated using the following formula:

```
raw harmonic centrality(node) = sum(1 / distance from node to every other node excluding itself)
```

As with closeness centrality, we can also calculate a normalized harmonic centrality with the following formula:

```
normalized harmonic centrality(node) = sum(1 / distance from node to every other node excluding itself) / (number of nodes - 1)
```

In this formula,  $\infty$  values are handled cleanly.

### Use-cases - when to use the Harmonic Centrality algorithm

Harmonic centrality was proposed as an alternative to closeness centrality, and therefore has similar use cases.

For example, we might use it if we're trying to identify where in the city to place a new public service so that it's easily accessible for residents. If we're trying to spread a message on social media we could use the algorithm to find the key influencers that can help us achieve our goal.

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.closeness.harmonic.write(configuration: Map)
YIELD nodes, createMillis, computeMillis, writeMillis, centralityDistribution
```

Table 289. Parameters

Name	Type	Default	Optional	Description
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
writeConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for writing the result.
writeProperty	string	'centrality'	yes	The property name written back to.

Table 290. Results

Name	Type	Description
nodes	int	The number of nodes considered.
createMillis	int	Milliseconds for loading data.
computeMillis	int	Milliseconds for running the algorithm.
writeMillis	int	Milliseconds for writing result data back.
writeProperty	string	The property name written back to.
centralityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of centrality values.

The following will run the algorithm and stream results:

```
CALL gds.alpha.closeness.harmonic.stream(configuration: Map)
YIELD nodeId, centrality
```

Table 291. Parameters

Name	Type	Default	Optional	Description
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

Table 292. Results

Name	Type	Description
node	long	Node ID
centrality	float	Harmonic centrality score

## Harmonic Centrality algorithm sample

The following will create a sample graph:

```
CREATE (a:Node{id:"A"}),
      (b:Node{id:"B"}),
      (c:Node{id:"C"}),
      (d:Node{id:"D"}),
      (e:Node{id:"E"}),
      (a)-[:LINK]->(b),
      (b)-[:LINK]->(c),
      (d)-[:LINK]->(e)
```

The following will run the algorithm and stream results:

```
CALL gds.alpha.closeness.harmonic.stream({
  nodeProjection: 'Node',
  relationshipProjection: 'LINK'
})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).name AS user, centrality
ORDER BY centrality DESC
```

Table 293. Results

Name	Centrality weight
B	0.5
A	0.375
c	0.375
D	0.25
E	0.25

The following will run the algorithm and write back results:

```
CALL gds.alpha.closeness.harmonic.write({
  nodeProjection: 'Node',
  relationshipProjection: 'LINK',
  writeProperty: 'centrality'
}) YIELD nodes, writeProperty
```

Table 294. Results

nodes	writeProperty
5	"centrality"

## 7.2.8. HITS Alpha

### Introduction

The Hyperlink-Induced Topic Search (HITS) is a link analysis algorithm that rates nodes based on two scores, a **hub** score and an **authority** score. The **authority** score estimates the importance of the node within the network. The **hub** score estimates the value of its relationships to other nodes. The GDS implementation is based on the [Authoritative Sources in a Hyperlinked Environment](#) publication by Jon M. Kleinberg.

### Syntax

This section covers the syntax used to execute the HITS algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run HITS in stream mode on a named graph.

```
CALL gds.alpha.hits.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  values: Map
```

Table 295. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 296. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 297. Algorithm specific configuration

Name	Type	Default	Optional	Description
hitsIterations	Integer	n/a	no	The number of hits iterations to run. The number of pregel iterations will be equal to $hitsIterations * 4 + 1$
authProperty	String	"auth"	yes	The name that is used for the auth property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.
hubProperty	String	"hub"	yes	The name that is used for the hub property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.

Table 298. Results

Name	Type	Description
nodeId	Integer	Node ID.
values	Map	A map containing the <b>auth</b> and <b>hub</b> keys.

Run HITS in stats mode on a named graph.

```
CALL gds.alpha.hits.stats(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  configuration: Map
```

Table 299. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 300. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 301. Algorithm specific configuration

Name	Type	Default	Optional	Description
hitsIterations	Integer	n/a	no	The number of hits iterations to run. The number of pregel iterations will be equal to $hitsIterations * 4 + 1$
authProperty	String	"auth"	yes	The name that is used for the auth property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.
hubProperty	String	"hub"	yes	The name that is used for the hub property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.

Table 302. Results

Name	Type	Description
ranIterations	Integer	Number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

Name	Type	Description
configuration	Map	Configuration used for running the algorithm.

Run HITS in mutate mode on a named graph.

```
CALL gds.alpha.hits.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 303. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 304. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 305. Algorithm specific configuration

Name	Type	Default	Optional	Description
hitsIterations	Integer	n/a	no	The number of hits iterations to run. The number of pregel iterations will be equal to $hitsIterations * 4 + 1$
authProperty	String	"auth"	yes	The name that is used for the auth property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.
hubProperty	String	"hub"	yes	The name that is used for the hub property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.

Table 306. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMilliseconds	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
configuration	Map	The configuration used for running the algorithm.

Run HITS in write mode on a named graph.

```
CALL gds.alpha.hits.write(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 307. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 308. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 309. Algorithm specific configuration

Name	Type	Default	Optional	Description
hitsIterations	Integer	n/a	no	The number of hits iterations to run. The number of pregel iterations will be equal to $hitsIterations * 4 + 1$
authProperty	String	"auth"	yes	The name that is used for the auth property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.
hubProperty	String	"hub"	yes	The name that is used for the hub property when using <b>STREAM</b> , <b>MUTATE</b> or <b>WRITE</b> modes.

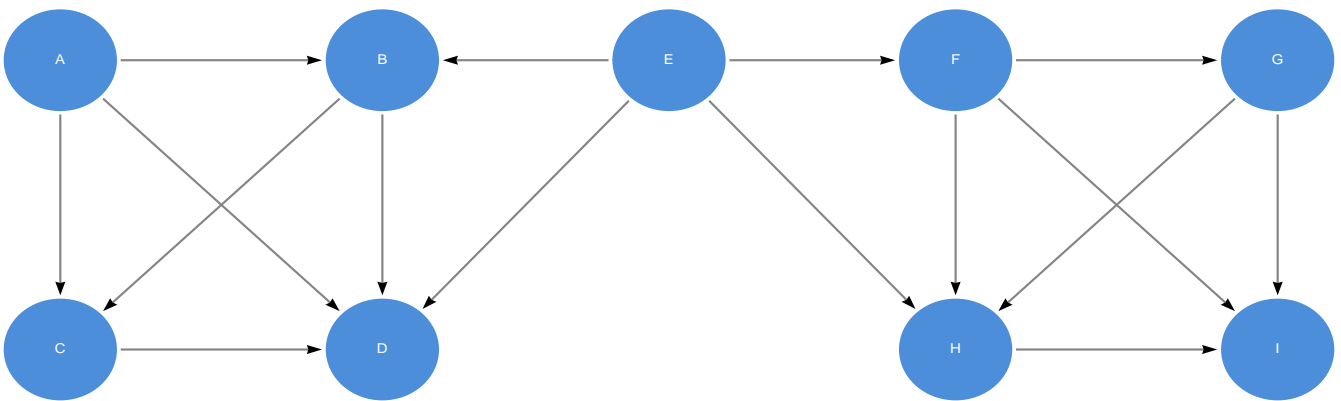
Table 310. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.

Name	Type	Description
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Examples

In this section we will show examples of running the HITS algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(a:Website {name: 'A'}),
(b:Website {name: 'B'}),
(c:Website {name: 'C'}),
(d:Website {name: 'D'}),
(e:Website {name: 'E'}),
(f:Website {name: 'F'}),
(g:Website {name: 'G'}),
(h:Website {name: 'H'}),
(i:Website {name: 'I'}),

(a)-[:LINK]->(b),
(a)-[:LINK]->(c),
(a)-[:LINK]->(d),
(b)-[:LINK]->(c),
(b)-[:LINK]->(d),
(c)-[:LINK]->(d),

(e)-[:LINK]->(b),
(e)-[:LINK]->(d),
(e)-[:LINK]->(f),
(e)-[:LINK]->(h),

(f)-[:LINK]->(g),
(f)-[:LINK]->(i),
(f)-[:LINK]->(h),
(g)-[:LINK]->(h),
(g)-[:LINK]->(i),
(h)-[:LINK]->(i);
```

In the example, we will use the HITS algorithm to calculate the authority and hub scores.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  'Website',
  'LINK'
);
```

In the following examples we will demonstrate using the HITS algorithm on this graph.

## Stream

In the `stream` execution mode, the algorithm returns the authority and hub scores for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.alpha.hits.stream('myGraph', {hitsIterations: 20})
YIELD nodeId, values
RETURN gds.util.asNode(nodeId).name AS Name, values.auth AS auth, values.hub AS hub
ORDER BY Name ASC
```

Table 311. Results



Name	auth	hub
"A"	0.0	0.5147630377521207
"B"	0.42644630743935796	0.3573686670593437
"C"	0.3218729455718005	0.23857061715828276
"D"	0.6463862608483191	0.0
"E"	0.0	0.640681017095129
"F"	0.23646490227616518	0.2763222153580397
"G"	0.10200264424057169	0.23867470447760597
"H"	0.426571816146601	0.0812340105698113
"I"	0.22009646020698218	0.0

## 7.2.9. Influence Maximization

The objective of influence maximization is to find a small subset of  $k$  nodes from a network in order to achieve maximization to the total number of nodes influenced by these  $k$  nodes. The Neo4j GDS library includes the following alpha influence maximization algorithms:

- Alpha
  - [Greedy](#)
  - [CELF](#)

### CELF Alpha

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Stream](#)

#### Introduction

The CELF algorithm for influence maximization aims to find  $k$  nodes that maximize the expected spread of influence in the network. It simulates the influence spread using the Independent Cascade model, which calculates the expected spread by taking the average spread over the  $mc$  Monte-Carlo simulations. In the propagation process, a node is influenced in case that a uniform random draw is less than the probability  $p$ .

Leskovec et al. 2007 introduced the CELF algorithm in their study [Cost-effective Outbreak Detection in Networks](#) to deal with the NP-hard problem of influence maximization. The CELF algorithm is based on a

"lazy-forward" optimization. The CELF algorithm dramatically improves the efficiency of the [Greedy](#) algorithm and should be preferred for large networks.

Syntax

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

Run CELLF in stream mode on a named graph.

```
CALL gds.alpha.influenceMaximization.celf.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  spread: Float
```

Table 312. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 313. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 314. Algorithm specific configuration

Name	Type	Default	Optional	Description
seedSetSize	Integer	n/a	no	The number of nodes that maximize the expected spread in the network.
monteCarlo Simulations	Integer	1000	yes	The number of Monte-Carlo simulations.
propagation Probability	Float	0.1	yes	The probability of a node being activated by an active neighbour node.

Table 315. Results

Name	Type	Description
nodeId	Integer	Node ID.
spread	Float	The spread gained by selecting the node.

Run CELF in stats mode on a named graph.

```
CALL gds.alpha.influenceMaximization.celf.stats(
  graphName: String,
  configuration: Map
)
YIELD
  nodes: Integer,
  computeMillis: Integer,
```

Table 316. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 317. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 318. Algorithm specific configuration

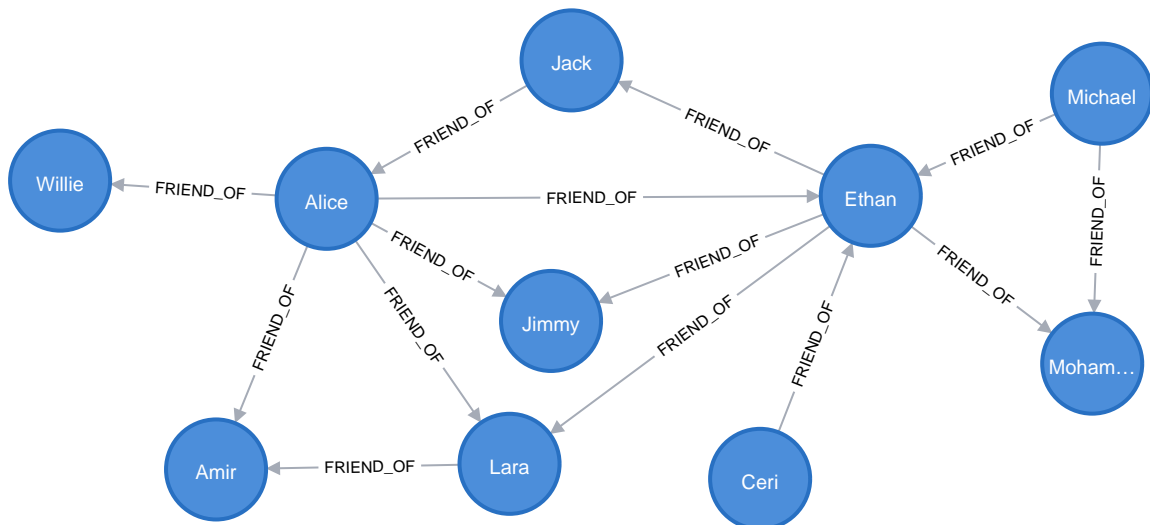
Name	Type	Default	Optional	Description
seedSetSize	Integer	n/a	no	The number of nodes that maximize the expected spread in the network.
monteCarlo Simulations	Integer	1000	yes	The number of Monte-Carlo simulations.
propagation Probability	Float	0.1	yes	The probability of a node being activated by an active neighbour node.

Table 319. Results

Name	Type	Description
nodes	Integer	The number of nodes in the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

## Examples

In this section we will show examples of running the CELF algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(a:Person {name: 'Jimmy'}),
(b:Person {name: 'Jack'}),
(c:Person {name: 'Alice'}),
(d:Person {name: 'Ceri'}),
(e:Person {name: 'Mohammed'}),
(f:Person {name: 'Michael'}),
(g:Person {name: 'Ethan'}),
(h:Person {name: 'Lara'}),
(i:Person {name: 'Amir'}),
(j:Person {name: 'Willie'}),

(b)-[:FRIEND_OF]->(c),
(c)-[:FRIEND_OF]->(a),
(c)-[:FRIEND_OF]->(g),
(c)-[:FRIEND_OF]->(h),
(c)-[:FRIEND_OF]->(i),
(c)-[:FRIEND_OF]->(j),
(d)-[:FRIEND_OF]->(g),
(f)-[:FRIEND_OF]->(e),
(f)-[:FRIEND_OF]->(g),
(g)-[:FRIEND_OF]->(a),
(g)-[:FRIEND_OF]->(b),
(g)-[:FRIEND_OF]->(h),
(g)-[:FRIEND_OF]->(e),
(h)-[:FRIEND_OF]->(i);
```

In the example, we will use the CELF algorithm to find  $k$  nodes subset.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  'FRIEND_OF'
);
```

In the following examples we will demonstrate using the CELF algorithm on this graph.

## Stream

In the `stream` execution mode, the algorithm returns the spread for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.alpha.influenceMaximization.celf.stream('myGraph', {seedSetSize: 3, concurrency: 4})
YIELD nodeId, spread
RETURN gds.util.asNode(nodeId).name AS Name, spread
ORDER BY spread ASC
```

Table 320. Results

Name	spread
"Alice"	1.519
"Ethan"	2.701
"Michael"	3.8

## Greedy Alpha

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Stream](#)

### Introduction

The Greedy algorithm for influence maximization aims to find  $k$  nodes that maximize the expected spread of influence in a network. It simulates the influence spread using the Independent Cascade model, which calculates the expected spread by taking the average spread over the `mc` Monte-Carlo simulations. In the propagation process, a node is influenced in case that a uniform random draw is less than the probability  $p$ .

Kempe et al. 2003 introduced the Greedy algorithm in their study [Maximizing the Spread of Influence through a Social Network](#) to deal with the NP-hard problem of influence maximization. The Greedy algorithm successively selecting the node within the maximum marginal gain approximation in polynomial time. For large networks [CELF](#) algorithm should be used.

Syntax

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

Run Greedy in stream mode on a named graph.

```
CALL gds.alpha.influenceMaximization.greedy.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  spread : Float
```

Table 321. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 322. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 323. Algorithm specific configuration

Name	Type	Default	Optional	Description
seedSetSize	Integer	n/a	no	The number of nodes that maximize the expected spread in the network.
monteCarlo Simulations	Integer	1000	yes	The number of Monte-Carlo simulations.
propagation Probability	Float	0.1	yes	The probability of a node being activated by an active neighbour node.

Table 324. Results

Name	Type	Description
nodeId	Integer	Node ID.
spread	Float	The spread gained by selecting the node.



Run Greedy in stats mode on a named graph.

```
CALL gds.alpha.influenceMaximization.greedy.stats(
  graphName: String,
  configuration: Map
)
YIELD
  nodes: Integer,
  computeMillis: Integer,
```

Table 325. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 326. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 327. Algorithm specific configuration

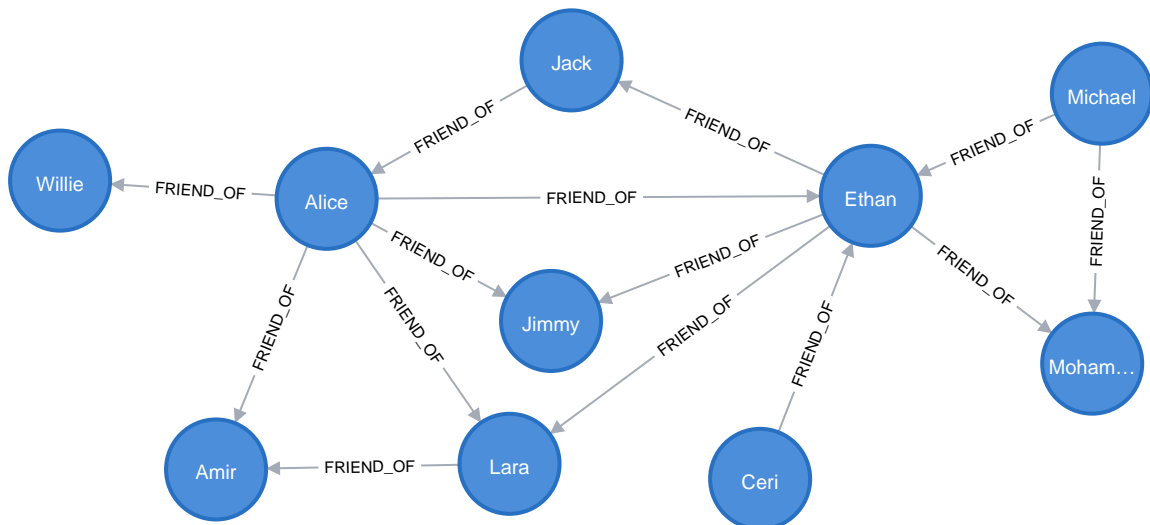
Name	Type	Default	Optional	Description
seedSetSize	Integer	n/a	no	The number of nodes that maximize the expected spread in the network.
monteCarlo Simulations	Integer	1000	yes	The number of Monte-Carlo simulations.
propagation Probability	Float	0.1	yes	The probability of a node being activated by an active neighbour node.

Table 328. Results

Name	Type	Description
nodes	Integer	The number of nodes in the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

## Examples

In this section we will show examples of running the Greedy algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(a:Person {name: 'Jimmy'}),
(b:Person {name: 'Jack'}),
(c:Person {name: 'Alice'}),
(d:Person {name: 'Ceri'}),
(e:Person {name: 'Mohammed'}),
(f:Person {name: 'Michael'}),
(g:Person {name: 'Ethan'}),
(h:Person {name: 'Lara'}),
(i:Person {name: 'Amir'}),
(j:Person {name: 'Willie'}),

(b)-[:FRIEND_OF]->(c),
(c)-[:FRIEND_OF]->(a),
(c)-[:FRIEND_OF]->(g),
(c)-[:FRIEND_OF]->(h),
(c)-[:FRIEND_OF]->(i),
(c)-[:FRIEND_OF]->(j),
(d)-[:FRIEND_OF]->(g),
(f)-[:FRIEND_OF]->(e),
(f)-[:FRIEND_OF]->(g),
(g)-[:FRIEND_OF]->(a),
(g)-[:FRIEND_OF]->(b),
(g)-[:FRIEND_OF]->(h),
(g)-[:FRIEND_OF]->(e),
(h)-[:FRIEND_OF]->(i);
```

In the example, we will use the Greedy algorithm to find  $k$  nodes subset.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  'FRIEND_OF'
);
```

In the following examples we will demonstrate using the Greedy algorithm on this graph.

## Stream

In the `stream` execution mode, the algorithm returns the spread for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.alpha.influenceMaximization.greedy.stream('myGraph', {seedSetSize: 3, concurrency: 4})
YIELD nodeId, spread
RETURN gds.util.asNode(nodeId).name AS Name, spread
ORDER BY spread ASC
```

Table 329. Results

Name	spread
"Alice"	1.519
"Ethan"	2.701
"Michael"	3.8

## 7.3. Community detection

Community detection algorithms are used to evaluate how groups of nodes are clustered or partitioned, as well as their tendency to strengthen or break apart. The Neo4j GDS library includes the following community detection algorithms, grouped by quality tier:

- Production-quality
  - [Louvain](#)
  - [Label Propagation](#)
  - [Weakly Connected Components](#)
  - [Triangle Count](#)
  - [Local Clustering Coefficient](#)
- Beta
  - [K-1 Coloring](#)
  - [Modularity Optimization](#)
- Alpha
  - [Strongly Connected Components](#)
  - [Speaker-Listener Label Propagation](#)
  - [Approximate Maximum k-cut](#)
  - [Conductance metric](#)

## 7.3.1. Louvain

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

The Louvain method is an algorithm to detect communities in large networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities. This means evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network.

The Louvain algorithm is a hierarchical clustering algorithm, that recursively merges communities into a single node and executes the modularity clustering on the condensed graphs.

For more information on this algorithm, see:

- [Lu, Hao, Mahantesh Halappanavar, and Ananth Kalyanaraman "Parallel heuristics for scalable community detection."](#)
- [https://en.wikipedia.org/wiki/Louvain\\_modularity](https://en.wikipedia.org/wiki/Louvain_modularity)



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

### Syntax

This section covers the syntax used to execute the Louvain algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Louvain in stream mode on a named graph.

```
CALL gds.louvain.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  communityId: Integer,
  intermediateCommunityIds: List of Integer
```

Table 330. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 331. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 332. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial community for a node. The property value needs to be a number.
maxLevels	Integer	10	yes	The maximum number of levels in which the graph is clustered and then condensed.
maxIterations	Integer	10	yes	The maximum number of iterations that the modularity optimization will run for each level.
tolerance	Float	0.0001	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.
includeIntermediateCommunities	Boolean	false	yes	Indicates whether to write intermediate communities. If set to false, only the final community is persisted.

Name	Type	Default	Optional	Description
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the <code>includeIntermediateCommunities</code> flag.

Table 333. Results

Name	Type	Description
nodeId	Integer	Node ID.
communityId	Integer	The community ID of the final level.
intermediateCommunityIds	List of Integer	Community IDs for each level. Null if <code>includeIntermediateCommunities</code> is set to false.

Run Louvain in stats mode on a named graph.

```
CALL gds.louvain.stats(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  createMillis: Integer,  
  computeMillis: Integer,  
  postProcessingMillis: Integer,  
  communityCount: Integer,  
  ranLevels: Integer,  
  modularity: Float,  
  modularities: List of Float,  
  communityDistribution: Map,  
  configuration: Map
```

Table 334. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 335. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 336. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial community for a node. The property value needs to be a number.
maxLevels	Integer	10	yes	The maximum number of levels in which the graph is clustered and then condensed.
maxIterations	Integer	10	yes	The maximum number of iterations that the modularity optimization will run for each level.
tolerance	Float	0.0001	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.



Name	Type	Default	Optional	Description
includeIntermediateCommunities	Boolean	false	yes	Indicates whether to write intermediate communities. If set to false, only the final community is persisted.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the <code>includeIntermediateCommunities</code> flag.

Table 337. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
communityCount	Integer	The number of communities found.
ranLevels	Integer	The number of supersteps the algorithm actually ran.
modularity	Float	The final modularity score.
modularities	List of Float	The modularity scores for each level.
communityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size for the last level.
configuration	Map	The configuration used for running the algorithm.

Run Louvain in mutate mode on a named graph.

```
CALL gds.louvain.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  postProcessingMillis: Integer,
  communityCount: Integer,
  ranLevels: Integer,
  modularity: Float,
  modularities: List of Float,
  nodePropertiesWritten: Integer,
  communityDistribution: Map,
  configuration: Map
```

Table 338. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 339. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 340. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial community for a node. The property value needs to be a number.
maxLevels	Integer	10	yes	The maximum number of levels in which the graph is clustered and then condensed.
maxIterations	Integer	10	yes	The maximum number of iterations that the modularity optimization will run for each level.
tolerance	Float	0.0001	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.

Name	Type	Default	Optional	Description
includeIntermediateCommunities	Boolean	false	yes	Indicates whether to write intermediate communities. If set to false, only the final community is persisted.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the <code>includeIntermediateCommunities</code> flag.

Table 341. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
communityCount	Integer	The number of communities found.
ranLevels	Integer	The number of supersteps the algorithm actually ran.
modularity	Float	The final modularity score.
modularities	List of Float	The modularity scores for each level.
nodePropertiesWritten	Integer	Number of properties added to the in-memory graph.
communityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size for the last level.
configuration	Map	The configuration used for running the algorithm.

Run Louvain in write mode on a named graph.

```
CALL gds.louvain.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodePropertiesWritten: Integer,
  communityCount: Integer,
  ranLevels: Integer,
  modularity: Float,
  modularities: List of Float,
  communityDistribution: Map,
  configuration: Map
```

Table 342. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 343. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 344. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial community for a node. The property value needs to be a number.
maxLevels	Integer	10	yes	The maximum number of levels in which the graph is clustered and then condensed.
maxIterations	Integer	10	yes	The maximum number of iterations that the modularity optimization will run for each level.

Name	Type	Default	Optional	Description
<code>tolerance</code>	Float	<code>0.0001</code>	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.
<code>includeIntermediateCommunities</code>	Boolean	<code>false</code>	yes	Indicates whether to write intermediate communities. If set to false, only the final community is persisted.
<code>consecutiveIds</code>	Boolean	<code>false</code>	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the <code>includeIntermediateCommunities</code> flag.
<code>minCommunitySize</code>	Integer	<code>0</code>	yes	Only community ids of communities with a size greater than or equal to the given value are written to Neo4j.

Table 345. Results

Name	Type	Description
<code>createMillis</code>	Integer	Milliseconds for loading data.
<code>computeMillis</code>	Integer	Milliseconds for running the algorithm.
<code>writeMillis</code>	Integer	Milliseconds for writing result data back.
<code>postProcessingMillis</code>	Integer	Milliseconds for computing percentiles and community count.
<code>nodePropertiesWritten</code>	Integer	The number of node properties written.
<code>communityCount</code>	Integer	The number of communities found.
<code>ranLevels</code>	Integer	The number of supersteps the algorithm actually ran.
<code>modularity</code>	Float	The final modularity score.
<code>modularities</code>	List of Float	The modularity scores for each level.
<code>communityDistribution</code>	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size for the last level.
<code>configuration</code>	Map	The configuration used for running the algorithm.

## Anonymous graphs

Run Louvain in write mode on an anonymous graph.

```
CALL gds.louvain.write(configuration: Map)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodePropertiesWritten: Integer,
  communityCount: Integer,
  ranLevels: Integer,
  modularity: Float,
  modularities: List of Float,
  communityDistribution: Map,
  configuration: Map
```

Table 346. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 347. Algorithm specific configuration

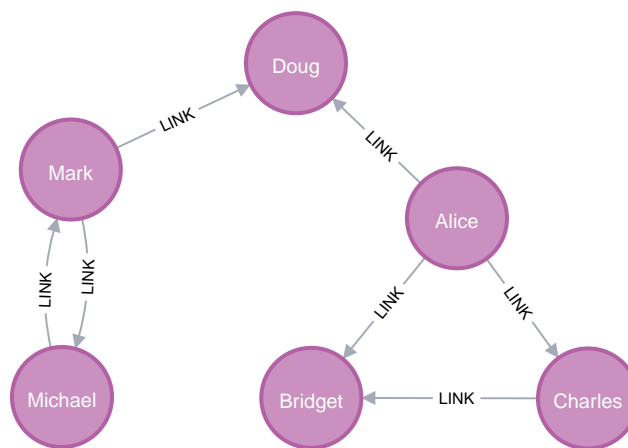
Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial community for a node. The property value needs to be a number.
maxLevels	Integer	10	yes	The maximum number of levels in which the graph is clustered and then condensed.

Name	Type	Default	Optional	Description
<code>maxIterations</code>	Integer	<code>10</code>	yes	The maximum number of iterations that the modularity optimization will run for each level.
<code>tolerance</code>	Float	<code>0.0001</code>	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.
<code>includeIntermediateCommunities</code>	Boolean	<code>false</code>	yes	Indicates whether to write intermediate communities. If set to false, only the final community is persisted.
<code>consecutiveIds</code>	Boolean	<code>false</code>	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory). Cannot be used in combination with the <code>includeIntermediateCommunities</code> flag.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Louvain community detection algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
  (nAlice:User {name: 'Alice', seed: 42}),
  (nBridget:User {name: 'Bridget', seed: 42}),
  (nCharles:User {name: 'Charles', seed: 42}),
  (nDoug:User {name: 'Doug'}),
  (nMark:User {name: 'Mark'}),
  (nMichael:User {name: 'Michael'}),

  (nAlice)-[:LINK {weight: 1}]->(nBridget),
  (nAlice)-[:LINK {weight: 1}]->(nCharles),
  (nCharles)-[:LINK {weight: 1}]->(nBridget),

  (nAlice)-[:LINK {weight: 5}]->(nDoug),

  (nMark)-[:LINK {weight: 1}]->(nDoug),
  (nMark)-[:LINK {weight: 1}]->(nMichael),
  (nMichael)-[:LINK {weight: 1}]->(nMark);

```

This graph has two clusters of *Users*, that are closely connected. Between those clusters there is one single edge. The relationships that connect the nodes in each component have a property `weight` which determines the strength of the relationship.

We can now create the graph and store it in the graph catalog. We load the `LINK` relationships with orientation set to `UNDIRECTED` as this works best with the Louvain algorithm.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(  
  'myGraph',  
  'User',  
  {  
    LINK: {  
      orientation: 'UNDIRECTED'  
    }  
  },  
  {  
    nodeProperties: 'seed',  
    relationshipProperties: 'weight'  
  }  
)
```

In the following examples we will demonstrate using the Louvain algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.louvain.write.estimate('myGraph', { writeProperty: 'community' })  
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 348. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	14	5321	563904	"[5321 Bytes ... 550 KiB]"



## Stream

In the `stream` execution mode, the algorithm returns the community ID for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
CALL gds.louvain.stream('myGraph')
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds
ORDER BY name ASC
```

Table 349. Results

name	communityId	intermediateCommunityIds
"Alice"	2	null
"Bridget"	2	null
"Charles"	2	null
"Doug"	5	null
"Mark"	5	null
"Michael"	5	null

We use default values for the procedure configuration parameter. Levels and `innerIterations` are set to 10 and the tolerance value is 0.0001. Because we did not set the value of `includeIntermediateCommunities` to `true`, the column `communities` is always `null`.

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm and returns the result in form of statistical and measurement values

```
CALL gds.louvain.stats('myGraph')
YIELD communityCount
```

Table 350. Results

communityCount
2

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the community ID for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm and store the results in `myGraph`:

```
CALL gds.louvain.mutate('myGraph', { mutateProperty: 'communityId' })
YIELD communityCount, modularity, modularities
```

Table 351. Results

communityCount	modularity	modularities
2	0.3571428571428571	[0.3571428571428571]

In `mutate` mode, only a single row is returned by the procedure. The result contains meta information, like the number of identified communities and the modularity values. In contrast to the `write` mode the result is written to the GDS in-memory graph instead of the Neo4j database.

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the community ID for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following run the algorithm, and write back results:

```
CALL gds.louvain.write('myGraph', { writeProperty: 'community' })
YIELD communityCount, modularity, modularities
```

Table 352. Results

communityCount	modularity	modularities
2	0.3571428571428571	[0.3571428571428571]

When writing back the results, only a single row is returned by the procedure. The result contains meta information, like the number of identified communities and the modularity values.

## Weighted

The Louvain algorithm can also run on weighted graphs, taking the given relationship weights into concern when calculating the modularity.

The following will run the algorithm on a weighted graph and stream results:

```
CALL gds.louvain.stream('myGraph', { relationshipWeightProperty: 'weight' })
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds
ORDER BY name ASC
```

Table 353. Results

name	communityId	intermediateCommunityIds
"Alice"	3	null
"Bridget"	2	null
"Charles"	2	null
"Doug"	3	null
"Mark"	5	null
"Michael"	5	null

Using the weighted relationships, we see that **Alice** and **Doug** have formed their own community, as their link is much stronger than all the others.

## Seeded

The Louvain algorithm can be run incrementally, by providing a seed property. With the seed property an initial community mapping can be supplied for a subset of the loaded nodes. The algorithm will try to keep the seeded community IDs.

The following will run the algorithm and stream results:

```
CALL gds.louvain.stream('myGraph', { seedProperty: 'seed' })
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds
ORDER BY name ASC
```

Table 354. Results

name	communityId	intermediateCommunityIds
"Alice"	42	null
"Bridget"	42	null
"Charles"	42	null
"Doug"	47	null
"Mark"	47	null

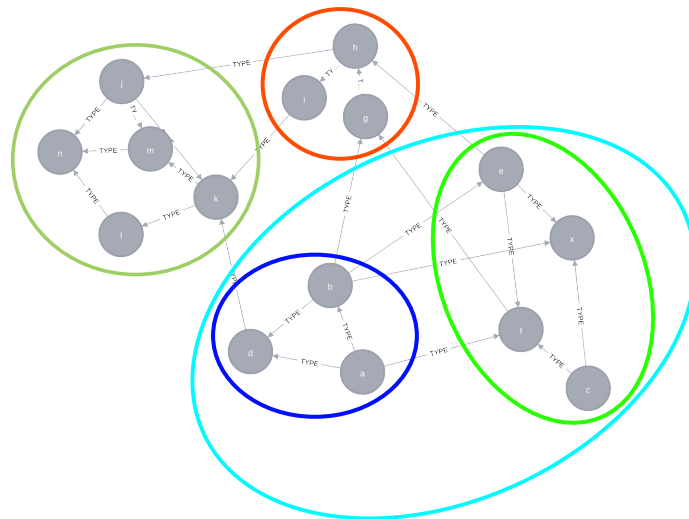
name	communityId	intermediateCommunityIds
"Michael"	47	null

Using the seeded graph, we see that the community around **Alice** keeps its initial community ID of **42**. The other community is assigned a new community ID, which is guaranteed to be larger than the largest seeded community ID. Note that the **consecutiveIds** configuration option cannot be used in combination with seeding in order to retain the seeding values.

### Stream intermediate communities

As described before, Louvain is a hierarchical clustering algorithm. That means that after every clustering step all nodes that belong to the same cluster are reduced to a single node. Relationships between nodes of the same cluster become self-relationships, relationships to nodes of other clusters connect to the clusters representative. This condensed graph is then used to run the next level of clustering. The process is repeated until the clusters are stable.

In order to demonstrate this iterative behavior, we need to construct a more complex graph.



```

CREATE (a:Node {name: 'a'})
CREATE (b:Node {name: 'b'})
CREATE (c:Node {name: 'c'})
CREATE (d:Node {name: 'd'})
CREATE (e:Node {name: 'e'})
CREATE (f:Node {name: 'f'})
CREATE (g:Node {name: 'g'})
CREATE (h:Node {name: 'h'})
CREATE (i:Node {name: 'i'})
CREATE (j:Node {name: 'j'})
CREATE (k:Node {name: 'k'})
CREATE (l:Node {name: 'l'})
CREATE (m:Node {name: 'm'})
CREATE (n:Node {name: 'n'})
CREATE (x:Node {name: 'x'})

CREATE (a)-[:TYPE]->(b)
CREATE (a)-[:TYPE]->(d)
CREATE (a)-[:TYPE]->(f)
CREATE (b)-[:TYPE]->(d)
CREATE (b)-[:TYPE]->(x)
CREATE (b)-[:TYPE]->(g)
CREATE (b)-[:TYPE]->(e)
CREATE (c)-[:TYPE]->(x)
CREATE (c)-[:TYPE]->(f)
CREATE (d)-[:TYPE]->(k)
CREATE (e)-[:TYPE]->(x)
CREATE (e)-[:TYPE]->(f)
CREATE (e)-[:TYPE]->(h)
CREATE (f)-[:TYPE]->(g)
CREATE (g)-[:TYPE]->(h)
CREATE (h)-[:TYPE]->(i)
CREATE (h)-[:TYPE]->(j)
CREATE (i)-[:TYPE]->(k)
CREATE (j)-[:TYPE]->(k)
CREATE (j)-[:TYPE]->(m)
CREATE (j)-[:TYPE]->(n)
CREATE (k)-[:TYPE]->(m)
CREATE (k)-[:TYPE]->(l)
CREATE (l)-[:TYPE]->(n)
CREATE (m)-[:TYPE]->(n);

```

The following will load the example graph, run the algorithm and stream results including the intermediate communities:

```

CALL gds.louvain.stream({
  nodeProjection: 'Node',
  relationshipProjection: {
    TYPE: {
      type: 'TYPE',
      orientation: 'undirected',
      aggregation: 'NONE'
    }
  },
  includeIntermediateCommunities: true
}) YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds
ORDER BY name ASC

```

Table 355. Results

name	communityId	intermediateCommunityIds
"a"	14	[3, 14]
"b"	14	[3, 14]
"c"	14	[14, 14]
"d"	14	[3, 14]

name	communityId	intermediateCommunityIds
"e"	14	[14, 14]
"f"	14	[14, 14]
"g"	7	[7, 7]
"h"	7	[7, 7]
"i"	7	[7, 7]
"j"	12	[12, 12]
"k"	12	[12, 12]
"l"	12	[12, 12]
"m"	12	[12, 12]
"n"	12	[12, 12]
"x"	14	[14, 14]

In this example graph, after the first iteration we see 4 clusters, which in the second iteration are reduced to three.

## 7.3.2. Label Propagation

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

The Label Propagation algorithm (LPA) is a fast algorithm for finding communities in a graph. It detects these communities using network structure alone as its guide, and doesn't require a pre-defined objective function or prior information about the communities.

LPA works by propagating labels throughout the network and forming communities based on this process of label propagation.

The intuition behind the algorithm is that a single label can quickly become dominant in a densely connected group of nodes, but will have trouble crossing a sparsely connected region. Labels will get trapped inside a densely connected group of nodes, and those nodes that end up with the same label when the algorithms finish can be considered part of the same community.

The algorithm works as follows:

- Every node is initialized with a unique community label (an identifier).
- These labels propagate through the network.
- At every iteration of propagation, each node updates its label to the one that the maximum numbers of its neighbours belongs to. Ties are broken arbitrarily but deterministically.
- LPA reaches convergence when each node has the majority label of its neighbours.
- LPA stops if either convergence, or the user-defined maximum number of iterations is achieved.

As labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label. At the end of the propagation only a few labels will remain - most will have disappeared. Nodes that have the same community label at convergence are said to belong to the same community.

One interesting feature of LPA is that nodes can be assigned preliminary labels to narrow down the range of solutions generated. This means that it can be used as semi-supervised way of finding communities where we hand-pick some initial communities.

For more information on this algorithm, see:

- ["Near linear time algorithm to detect community structures in large-scale networks"](#)
- Use cases:
  - [Twitter polarity classification with label propagation over lexical links and the follower graph](#)
  - [Label Propagation Prediction of Drug-Drug Interactions Based on Clinical Side Effects](#)
  - ["Feature Inference Based on Label Propagation on Wikidata Graph for DST"](#)



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

## Syntax

This section covers the syntax used to execute the Label Propagation algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).





Run Label Propagation in stream mode on a named graph.

```
CALL gds.labelPropagation.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  communityId: Integer
```

Table 356. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{ }	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 357. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 358. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
nodeWeightProperty	String	null	yes	The name of a node property that contains node weights.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	The name of a node property that defines an initial numeric label.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 359. Results

Name	Type	Description
nodeId	Integer	Node ID.
communityId	Integer	Community ID.

Run Label Propagation in stats mode on a named graph.

```
CALL gds.labelPropagation.stats(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  communityCount: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  communityDistribution: Map,
  configuration: Map
```

Table 360. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 361. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 362. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
nodeWeightProperty	String	null	yes	The name of a node property that contains node weights.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	The name of a node property that defines an initial numeric label.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 363. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
communityCount	Integer	The number of communities found.
ranIterations	Integer	The number of iterations that were executed.
didConverge	Boolean	True if the algorithm did converge to a stable labelling within the provided number of maximum iterations.
communityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size.
configuration	Map	The configuration used for running the algorithm.

Run Label Propagation in mutate mode on a named graph.

```
CALL gds.labelPropagation.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  postProcessingMillis: Integer,
  nodePropertiesWritten: Integer,
  communityCount: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  communityDistribution: Map,
  configuration: Map
```

Table 364. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 365. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 366. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
nodeWeightProperty	String	null	yes	The name of a node property that contains node weights.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	The name of a node property that defines an initial numeric label.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 367. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
nodePropertiesWritten	Integer	The number of node properties written.
communityCount	Integer	The number of communities found.
ranIterations	Integer	The number of iterations that were executed.
didConverge	Boolean	True if the algorithm did converge to a stable labelling within the provided number of maximum iterations.
communityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size.
configuration	Map	The configuration used for running the algorithm.

Run Label Propagation in write mode on a named graph.

```
CALL gds.labelPropagation.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodePropertiesWritten: Integer,
  communityCount: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  communityDistribution: Map,
  configuration: Map
```

Table 368. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 369. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 370. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
nodeWeightProperty	String	null	yes	The name of a node property that contains node weights.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	The name of a node property that defines an initial numeric label.

Name	Type	Default	Optional	Description
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).
minCommunitySize	Integer	0	yes	Only community ids of communities with a size greater than or equal to the given value are written to Neo4j.

Table 371. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
nodePropertiesWritten	Integer	The number of node properties written.
communityCount	Integer	The number of communities found.
ranIterations	Integer	The number of iterations that were executed.
didConverge	Boolean	True if the algorithm did converge to a stable labelling within the provided number of maximum iterations.
communityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of community size.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Label Propagation in write mode on an anonymous graph:

```
CALL gds.labelPropagation.write(
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodePropertiesWritten: Integer,
  communityCount: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  communityDistribution: Map,
  configuration: Map
```

Table 372. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 373. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
nodeWeightProperty	String	null	yes	The name of a node property that contains node weights.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

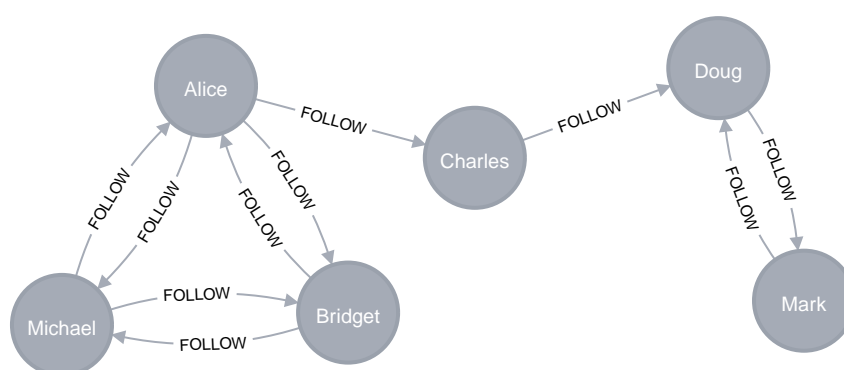


Name	Type	Default	Optional	Description
<a href="#">seedProperty</a>	String	n/a	yes	The name of a node property that defines an initial numeric label.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Label Propagation algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
  (alice:User {name: 'Alice', seed_label: 52}),
  (bridget:User {name: 'Bridget', seed_label: 21}),
  (charles:User {name: 'Charles', seed_label: 43}),
  (doug:User {name: 'Doug', seed_label: 21}),
  (mark:User {name: 'Mark', seed_label: 19}),
  (michael:User {name: 'Michael', seed_label: 52}),

  (alice)-[:FOLLOW {weight: 1}]->(bridget),
  (alice)-[:FOLLOW {weight: 10}]->(charles),
  (mark)-[:FOLLOW {weight: 1}]->(doug),
  (bridget)-[:FOLLOW {weight: 1}]->(michael),
  (doug)-[:FOLLOW {weight: 1}]->(mark),
  (michael)-[:FOLLOW {weight: 1}]->(alice),
  (alice)-[:FOLLOW {weight: 1}]->(michael),
  (bridget)-[:FOLLOW {weight: 1}]->(alice),
  (michael)-[:FOLLOW {weight: 1}]->(bridget),
  (charles)-[:FOLLOW {weight: 1}]->(doug)

```

This graph represents six users, some of whom follow each other. Besides a `name` property, each user also has a `seed_label` property. The `seed_label` property represents a value in the graph used to seed the node with a label. For example, this can be a result from a previous run of the Label Propagation algorithm. In addition, each relationship has a `weight` property.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'User',
  'FOLLOW',
  {
    nodeProperties: 'seed_label',
    relationshipProperties: 'weight'
  }
)
```

In the following examples we will demonstrate using the Label Propagation algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
CALL gds.labelPropagation.write.estimate('myGraph', { writeProperty: 'community' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 374. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	10	1608	1608	"1608 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the community ID for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
CALL gds.labelPropagation.stream('myGraph')
YIELD nodeId, communityId AS Community
RETURN gds.util.asNode(nodeId).name AS Name, Community
ORDER BY Community, Name
```

Table 375. Results

Name	Community
"Alice"	1
"Bridget"	1
"Michael"	1
"Charles"	4
"Doug"	4
"Mark"	4

In the above example we can see that our graph has two communities each containing three nodes. The default behaviour of the algorithm is to run `unweighted`, e.g. without using `node` or `relationship` weights. The `weighted` option will be demonstrated in [Weighted](#)

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm in `stats` mode:

```
CALL gds.labelPropagation.stats('myGraph')
YIELD communityCount, ranIterations, didConverge
```

Table 376. Results

communityCount	ranIterations	didConverge
2	3	true

As we can see from the example above the algorithm finds two communities and converges in three iterations. Note that we ran the algorithm `unweighted`.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the community ID for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm and write back results:

```
CALL gds.labelPropagation.mutate('myGraph', { mutateProperty: 'community' })
YIELD communityCount, ranIterations, didConverge
```

Table 377. Results

communityCount	ranIterations	didConverge
2	3	true

The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `community` which stores the community ID for each node. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs](#).

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the community ID for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm and write back results:

```
CALL gds.labelPropagation.write('myGraph', { writeProperty: 'community' })
YIELD communityCount, ranIterations, didConverge
```

Table 378. Results

communityCount	ranIterations	didConverge
2	3	true

The returned result is the same as in the `stats` example. Additionally, each of the six nodes now has a new property `community` in the Neo4j database, containing the community ID for that node.

## Weighted

The Label Propagation algorithm can also be configured to use node and/or relationship weights into account. By specifying a node weight via the `nodeWeightProperty` key, we can control the influence of a nodes community onto its neighbors. During the computation of the weight of a specific community, the node property will be multiplied by the weight of that nodes relationships.

When we created `myGraph`, we projected the relationship property `weight`. In order to tell the algorithm to consider this property as a relationship weight, we have to set the `relationshipWeightProperty` configuration parameter to `weight`.

The following will run the algorithm on a graph with weighted relationships and stream results:

```
CALL gds.labelPropagation.stream('myGraph', { relationshipWeightProperty: 'weight' })
YIELD nodeId, communityId AS Community
RETURN gds.util.asNode(nodeId).name AS Name, Community
ORDER BY Community, Name
```

Table 379. Results

Name	Community
"Bridget"	2
"Michael"	2
"Alice"	4
"Charles"	4
"Doug"	4
"Mark"	4

Compared to the [unweighted run](#) of the algorithm we still have two communities, but they contain two and four nodes respectively. Using the weighted relationships, the nodes [Alice](#) and [Charles](#) are now in the same community as there is a strong link between them.



We have used the `stream` mode to demonstrate running the algorithm using weights, the configuration parameters are available for all the modes of the algorithm.

## Seeded communities

At the beginning of the algorithm computation, every node is initialized with a unique label, and the labels propagate through the network.

An initial set of labels can be provided by setting the `seedProperty` configuration parameter. When we created `myGraph`, we projected the node property `seed_label`. We can use this node property as `seedProperty`.

The algorithm first checks if there is a seed label assigned to the node. If no seed label is present, the algorithm assigns new unique label to the node. Using this preliminary set of labels, it then sequentially updates each node's label to a new one, which is the most frequent label among its neighbors at every iteration of label propagation.



The `consecutiveIds` configuration option cannot be used in combination with `seedProperty` in order to retain the seeding values.

The following will run the algorithm with pre-defined labels:

```
CALL gds.labelPropagation.stream('myGraph', { seedProperty: 'seed_label' })
YIELD nodeId, communityId AS Community
RETURN gds.util.asNode(nodeId).name AS Name, Community
ORDER BY Community, Name
```

Table 380. Results

Name	Community
"Charles"	19
"Doug"	19
"Mark"	19
"Alice"	21
"Bridget"	21
"Michael"	21

As we can see, the communities are based on the `seed_label` property, concretely 19 is from the node `Mark` and 21 from `Doug`.



We have used the `stream` mode to demonstrate running the algorithm using `seedProperty`, this configuration parameter is available for all the modes of the algorithm.

### 7.3.3. Weakly Connected Components

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

#### Introduction

The WCC algorithm finds sets of connected nodes in an undirected graph, where all nodes in the same set form a connected component. WCC is often used early in an analysis to understand the structure of a graph. Using WCC to understand the graph structure enables running other algorithms independently on an identified cluster. As a preprocessing step for directed graphs, it helps quickly identify disconnected groups.

For more information on this algorithm, see:

- ["An efficient domain-independent algorithm for detecting approximately duplicate database records"](#).
- One study uses WCC to work out how well connected the network is, and then to see whether the connectivity remains if 'hub' or 'authority' nodes are moved from the graph: ["Characterizing and Mining Citation Graph of Computer Science Literature"](#)



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

## Syntax

This section covers the syntax used to execute the Weakly Connected Components algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).





Run WCC in stream mode on a named graph.

```
CALL gds.wcc.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  componentId: Integer
```

Table 381. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 382. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 383. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial component for a node. The property value needs to be a number.
threshold	Float	null	yes	The value of the weight above which the relationship is considered in the computation.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 384. Results

Name	Type	Description
nodeId	Integer	Node ID.
componentId	Integer	Component ID.

Run WCC in stats mode on a named graph.

```
CALL gds.wcc.stats(
  graphName: String,
  configuration: Map
)
YIELD
  componentCount: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  componentDistribution: Map,
  configuration: Map
```

Table 385. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 386. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 387. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial component for a node. The property value needs to be a number.
threshold	Float	null	yes	The value of the weight above which the relationship is considered in the computation.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 388. Results

Name	Type	Description
componentCount	Integer	The number of computed components.

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing component count and distribution statistics.
component Distribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of component sizes.
configuration	Map	The configuration used for running the algorithm.

Run WCC in mutate mode on a named graph.

```
CALL gds.wcc.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  componentCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  postProcessingMillis: Integer,
  componentDistribution: Map,
  configuration: Map
```

Table 389. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 390. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 391. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial component for a node. The property value needs to be a number.
threshold	Float	null	yes	The value of the weight above which the relationship is considered in the computation.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 392. Results

Name	Type	Description
componentCount	Integer	The number of computed components.
nodePropertiesWritten	Integer	The number of node properties written.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for computing component count and distribution statistics.
componentDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of component sizes.
configuration	Map	The configuration used for running the algorithm.

Run WCC in write mode on a named graph.

```
CALL gds.wcc.write(
  graphName: String,
  configuration: Map
)
YIELD
  componentCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  componentDistribution: Map,
  configuration: Map
```

Table 393. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 394. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 395. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
seedProperty	String	n/a	yes	Used to set the initial component for a node. The property value needs to be a number.
threshold	Float	null	yes	The value of the weight above which the relationship is considered in the computation.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Name	Type	Default	Optional	Description
minComponentSize	Integer	0	yes	Only component ids of components with a size greater than or equal to the given value are written to Neo4j.

Table 396. Results

Name	Type	Description
componentCount	Integer	The number of computed components.
nodePropertiesWritten	Integer	The number of node properties written.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result back to Neo4j.
postProcessingMillis	Integer	Milliseconds for computing component count and distribution statistics.
componentDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of component sizes.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run WCC in write mode on an anonymous graph:

```
CALL gds.wcc.write(
  configuration: Map
)
YIELD
  componentCount: Integer,
  nodePropertiesWritten: Integer,
  relationshipPropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  componentDistribution: Map,
  configuration: Map
```

Table 397. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	<code>null</code>	yes	The relationship properties to project during anonymous graph creation.
<code>concurrency</code>	Integer	<code>4</code>	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
<code>readConcurrency</code>	Integer	<code>value of 'concurrency'</code>	yes	The number of concurrent threads used for creating the graph.
<code>writeConcurrency</code>	Integer	<code>value of 'concurrency'</code>	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 398. Algorithm specific configuration

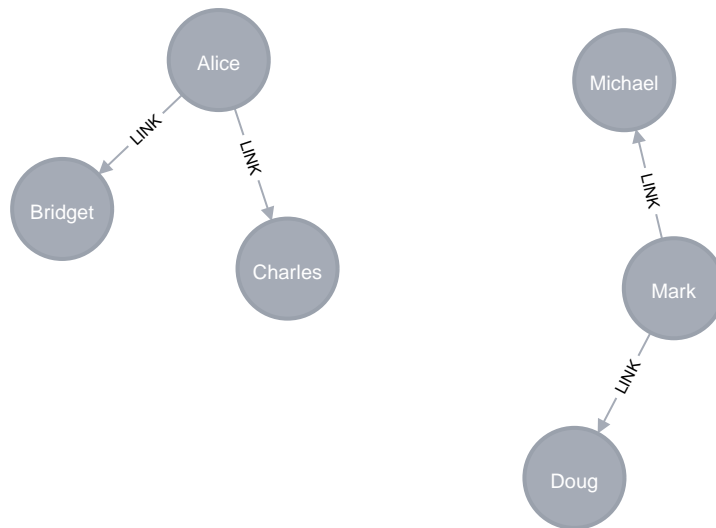
Name	Type	Default	Optional	Description
<code>relationshipWeightProperty</code>	String	<code>null</code>	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
<code>seedProperty</code>	String	<code>n/a</code>	yes	Used to set the initial component for a node. The property value needs to be a number.
<code>threshold</code>	Float	<code>null</code>	yes	The value of the weight above which the relationship is considered in the computation.
<code>consecutiveIds</code>	Boolean	<code>false</code>	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Weakly Connected Components algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small user network graph of a handful nodes connected in a particular pattern. The example graph looks like this:





The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
  (nAlice:User {name: 'Alice'}),
  (nBridget:User {name: 'Bridget'}),
  (nCharles:User {name: 'Charles'}),
  (nDoug:User {name: 'Doug'}),
  (nMark:User {name: 'Mark'}),
  (nMichael:User {name: 'Michael'}),

  (nAlice)-[:LINK {weight: 0.5}]->(nBridget),
  (nAlice)-[:LINK {weight: 4}]->(nCharles),
  (nMark)-[:LINK {weight: 1.1}]->(nDoug),
  (nMark)-[:LINK {weight: 2}]->(nMichael);
  
```

This graph has two connected components, each with three nodes. The relationships that connect the nodes in each component have a property `weight` which determines the strength of the relationship.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```

CALL gds.graph.create(
  'myGraph',
  'User',
  'LINK',
  {
    relationshipProperties: 'weight'
  }
)
  
```

In the following examples we will demonstrate using the Weakly Connected Components algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later

actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
CALL gds.wcc.write.estimate('myGraph', { writeProperty: 'component' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 399. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	4	176	176	"176 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the component ID for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
CALL gds.wcc.stream('myGraph')
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS name, componentId
ORDER BY componentId, name
```

Table 400. Results

name	componentId
"Alice"	0
"Bridget"	0
"Charles"	0
"Doug"	3
"Mark"	3
"Michael"	3

The result shows that the algorithm identifies two components. This can be verified in the [example graph](#).

The default behaviour of the algorithm is to run `unweighted`, e.g. without using `relationship` weights. The `weighted` option will be demonstrated in [Weighted](#)

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm in `stats` mode:

```
CALL gds.wcc.stats('myGraph')
YIELD componentCount
```

Table 401. Results

componentCount
2

The result shows that `myGraph` has two components and this can be verified by looking at the [example graph](#).

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the component ID for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.wcc.mutate('myGraph', { mutateProperty: 'componentId' })
YIELD nodePropertiesWritten, componentCount;
```

Table 402. Results

nodePropertiesWritten	componentCount
6	2

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the component ID for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.wcc.write('myGraph', { writeProperty: 'componentId' })
YIELD nodePropertiesWritten, componentCount;
```

Table 403. Results

nodePropertiesWritten	componentCount
6	2

As we can see from the results, the nodes connected to one another are calculated by the algorithm as belonging to the same connected component.

## Weighted

By configuring the algorithm to use a weight we can increase granularity in the way the algorithm calculates component assignment. We do this by specifying the property key with the `relationshipWeightProperty` configuration parameter. Additionally, we can specify a threshold for the weight value. Then, only weights greater than the threshold value will be considered by the algorithm. We do this by specifying the threshold value with the `threshold` configuration parameter.

If a relationship does not have the specified weight property, the algorithm falls back to using a default value of zero.

The following will run the algorithm and stream results:

```
CALL gds.wcc.stream('myGraph', {
  relationshipWeightProperty: 'weight',
  threshold: 1.0
}) YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS Name, componentId AS ComponentId
ORDER BY ComponentId, Name
```

Table 404. Results

Name	ComponentId
"Alice"	0
"Charles"	0
"Bridget"	1
"Doug"	3
"Mark"	3
"Michael"	3

As we can see from the results, the node named 'Bridget' is now in its own component, due to its relationship weight being less than the configured threshold and thus ignored.



We are using stream mode to illustrate running the algorithm as weighted or unweighted, all the other algorithm modes also support this configuration parameter.

## Seeded components

It is possible to define preliminary component IDs for nodes using the `seedProperty` configuration parameter. This is helpful if we want to retain components from a previous run and it is known that no components have been split by removing relationships. The property value needs to be a number.

The algorithm first checks if there is a seeded component ID assigned to the node. If there is one, that component ID is used. Otherwise, a new unique component ID is assigned to the node.

Once every node belongs to a component, the algorithm merges components of connected nodes. When components are merged, the resulting component is always the one with the lower component ID. Note that the `consecutiveIds` configuration option cannot be used in combination with seeding in order to retain the seeding values.



The algorithm assumes that nodes with the same seed value do in fact belong to the same component. If any two nodes in different components have the same seed, behavior is undefined. It is then recommended running WCC without seeds.

To demonstrate this in practice, we will go through a few steps:

1. We will run the algorithm and write the results to Neo4j.
2. Then we will add another node to our graph, this node will not have the property computed in Step 1.
3. We will create a new in-memory graph that has the result from Step 1 as `nodeProperty`
4. And then we will run the algorithm again, this time in `stream` mode, and we will use the `seedProperty` configuration parameter.

We will use the weighted variant of WCC.

### Step 1

The following will run the algorithm in `write` mode:

```
CALL gds.wcc.write('myGraph', {
  writeProperty: 'componentId',
  relationshipWeightProperty: 'weight',
  threshold: 1.0
})
YIELD nodePropertiesWritten, componentCount;
```

Table 405. Results

nodePropertiesWritten	componentCount
6	3

### Step 2

After the algorithm has finished writing to Neo4j we want to create a new node in the database.

The following will create a new node in the Neo4j graph, with no component ID:

```
MATCH (b:User {name: 'Bridget'})
CREATE (b)-[:LINK {weight: 2.0}]->(new:User {name: 'Mats'})
```

### Step 3

Note, that we cannot use our already created graph as it does not contain the component id. We will therefore create a second in-memory graph that contains the previously computed component id.

The following will create a new graph containing the previously computed component id:

```
CALL gds.graph.create(
  'myGraph-seeded',
  'User',
  'LINK',
  {
    nodeProperties: 'componentId',
    relationshipProperties: 'weight'
  }
)
```

### Step 4

The following will run the algorithm in `stream` mode using `seedProperty`:

```
CALL gds.wcc.stream('myGraph-seeded', {
  seedProperty: 'componentId',
  relationshipWeightProperty: 'weight',
  threshold: 1.0
}) YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS name, componentId
ORDER BY componentId, name
```

Table 406. Results

name	componentId
"Alice"	0
"Charles"	0
"Bridget"	1
"Mats"	1
"Doug"	3
"Mark"	3
"Michael"	3

The result shows that despite not having the `seedProperty` when it was created, the node 'Mats' has been assigned to the same component as the node 'Bridget'. This is correct because these two nodes are connected.

## Writing Seeded components

In the [previous section](#) we demonstrated the `seedProperty` usage in `stream` mode. It is also available in the other modes of the algorithm. Below is an example on how to use `seedProperty` in `write` mode. Note that the example below relies on [Steps 1 - 3](#) from the previous section.

The following will run the algorithm in `write` mode using `seedProperty`:

```
CALL gds.wcc.write('myGraph-seeded', {
  seedProperty: 'componentId',
  writeProperty: 'componentId',
  relationshipWeightProperty: 'weight',
  threshold: 1.0
})
YIELD nodePropertiesWritten, componentCount;
```

Table 407. Results

nodePropertiesWritten	componentCount
1	3



If the `seedProperty` configuration parameter has the same value as `writeProperty`, the algorithm only writes properties for nodes where the component ID has changed. If they differ, the algorithm writes properties for all nodes.

### 7.3.4. Triangle Count

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

#### Introduction

The Triangle Count algorithm counts the number of triangles for each node in the graph. A triangle is a set of three nodes where each node has a relationship to the other two. In graph theory terminology, this is sometimes referred to as a 3-clique. The Triangle Count algorithm in the GDS library only finds triangles in undirected graphs.

Triangle counting has gained popularity in social network analysis, where it is used to detect communities and measure the cohesiveness of those communities. It can also be used to determine the stability of a graph, and is often used as part of the computation of network indices, such as clustering coefficients. The Triangle Count algorithm is also used to compute the [Local Clustering Coefficient](#).

For more information on this algorithm, see:

- Triangle count and clustering coefficient have been shown to be useful as features for classifying a given website as spam, or non-spam, content. This is described in "[Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs](#)".

## Syntax

This section covers the syntax used to execute the Triangle Count algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



The named graphs must be projected in the **UNDIRECTED** orientation for the Triangle Count algorithm.



Run Triangle Count in stream mode on a named graph:

```
CALL gds.triangleCount.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  triangleCount: Integer
```

Table 408. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 409. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 410. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxDegree	Integer	$2^{63} - 1$	Yes	If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

Table 411. Results

Name	Type	Description
nodeId	Integer	Node ID.
triangleCount	Integer	Number of triangles the node is part of. Is -1 if the node has been excluded from computation using the <code>maxDegree</code> configuration parameter.

Run Triangle Count in stats mode on a named graph:

```
CALL gds.triangleCount.stats(
  graphName: String,
  configuration: Map
)
YIELD
  globalTriangleCount: Integer,
  nodeCount: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  configuration: Map
```

Table 412. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 413. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 414. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxDegree	Integer	$2^{63} - 1$	Yes	If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

Table 415. Results

Name	Type	Description
globalTriangleCount	Integer	Total number of triangles in the graph.
nodeCount	Integer	Number of nodes in the graph.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.

Name	Type	Description
configuration	Map	The configuration used for running the algorithm.

Run Triangle Count in mutate mode on a named graph:

```
CALL gds.triangleCount.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  globalTriangleCount: Integer,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  configuration: Map
```

Table 416. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 417. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 418. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxDegree	Integer	$2^{63} - 1$	Yes	If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

Table 419. Results

Name	Type	Description
globalTriangleCount	Integer	Total number of triangles in the graph.
nodeCount	Integer	Number of nodes in the graph.
nodePropertiesWritten	Integer	Number of properties added to the in-memory graph.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMilliseconds	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
configuration	Map	The configuration used for running the algorithm.

Run Triangle Count in write mode on a named graph:

```
CALL gds.triangleCount.write(
  graphName: String,
  configuration: Map
)
YIELD
  globalTriangleCount: Integer,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 420. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 421. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 422. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxDegree	Integer	$2^{63} - 1$	Yes	If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

Table 423. Results

Name	Type	Description
globalTriangleCount	Integer	Total number of triangles in the graph.
nodeCount	Integer	Number of nodes in the graph.
nodePropertiesWritten	Integer	Number of properties written to Neo4j.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
writeMillis	Integer	Milliseconds for writing results back to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run *Triangle Count* in *write* mode on an anonymous graph:

```
CALL gds.triangleCount.write(
  configuration: Map
)
YIELD
  globalTriangleCount: Integer,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 424. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.

Name	Type	Default	Optional	Description
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 425. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxDegree	Integer	$2^{63} - 1$	Yes	If a node has a degree higher than this it will not be considered by the algorithm. The triangle count for these nodes will be -1.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Triangles listing

In addition to the standard execution modes there is an `alpha` procedure `gds.alpha.triangles` that can be used to list all triangles in the graph.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

The following will return a stream of node IDs for each triangle:

```
CALL gds.alpha.triangles(
  graphName: String,
  configuration: Map
)
YIELD nodeA, nodeB, nodeC
```

Table 426. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 427. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.



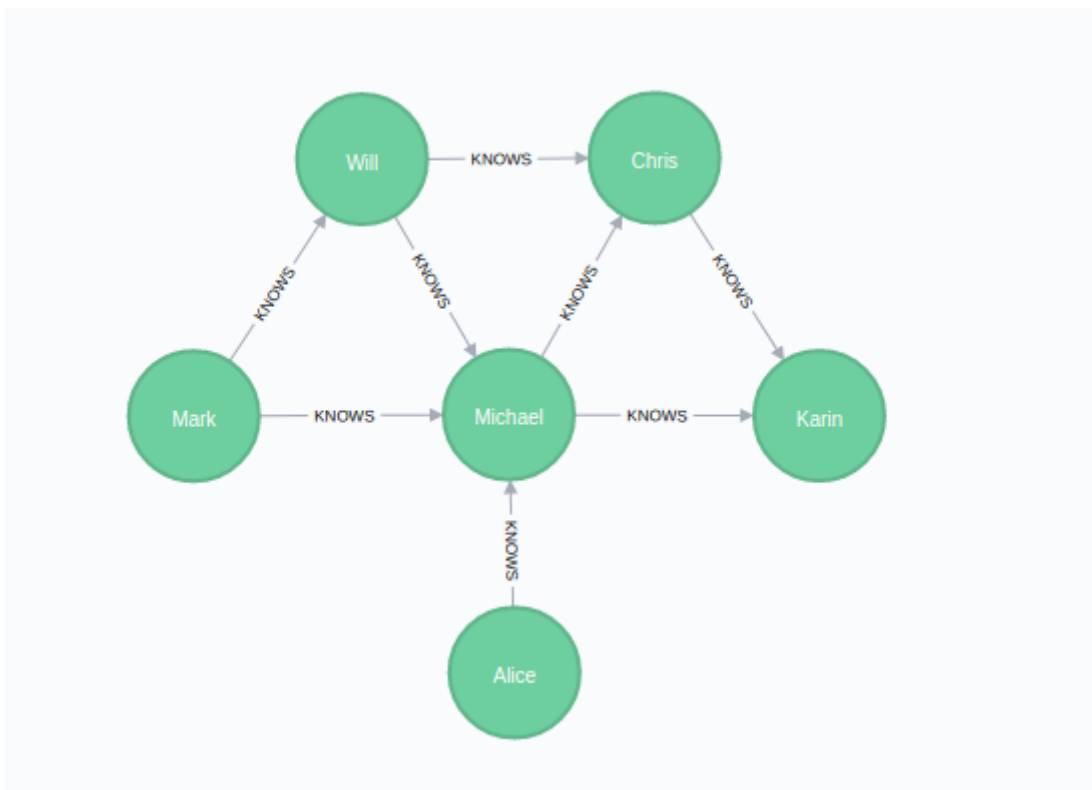
Name	Type	Default	Optional	Description
<a href="#">relationshipTypes</a>	List of String	[ '*' ]	yes	Filter the named graph using the given relationship types.
<a href="#">concurrency</a>	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 428. Results

Name	Type	Description
nodeA	Integer	The ID of the first node in the given triangle.
nodeB	Integer	The ID of the second node in the given triangle.
nodeC	Integer	The ID of the third node in the given triangle.

## Examples

In this section we will show examples of running the Triangle Count algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
  (alice:Person {name: 'Alice'}),
  (michael:Person {name: 'Michael'}),
  (karin:Person {name: 'Karin'}),
  (chris:Person {name: 'Chris'}),
  (will:Person {name: 'Will'}),
  (mark:Person {name: 'Mark'}),

  (michael)-[:KNOWS]->(karin),
  (michael)-[:KNOWS]->(chris),
  (will)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(will),
  (alice)-[:KNOWS]->(michael),
  (will)-[:KNOWS]->(chris),
  (chris)-[:KNOWS]->(karin)
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. For the relationships we must use the `UNDIRECTED` orientation. This is because the Triangle Count algorithm is defined only for undirected graphs.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)
```



The Triangle Count algorithm requires the graph to be created using the `UNDIRECTED` orientation for relationships.

In the following examples we will demonstrate using the Triangle Count algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
CALL gds.triangleCount.write.estimate('myGraph', { writeProperty: 'triangleCount' })
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 429. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	16	152	152	"152 Bytes"

Note that the relationship count is 16 although we only created 8 relationships in the original Cypher statement. This is because we used the **UNDIRECTED** orientation, which will project each relationship in each direction, effectively doubling the number of relationships.

## Stream

In the **stream** execution mode, the algorithm returns the triangle count for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the **stream** mode in general, see [Stream](#).

The following will run the algorithm in **stream** mode:

```
CALL gds.triangleCount.stream('myGraph')
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).name AS name, triangleCount
ORDER BY triangleCount DESC
```

Table 430. Results

name	triangleCount
"Michael"	3
"Chris"	2
"Will"	2
"Karin"	1
"Mark"	1
"Alice"	0

Here we find that the 'Michael' node has the most triangles. This can be verified in the [example graph](#). Since the 'Alice' node only **KNOWS** one other node, it can not be part of any triangle, and indeed the algorithm reports a count of zero.

## Stats

In the **stats** execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the **computeMillis** return item. In the examples below we will omit returning

the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm in `stats` mode:

```
CALL gds.triangleCount.stats('myGraph')
YIELD globalTriangleCount, nodeCount
```

Table 431. Results

globalTriangleCount	nodeCount
3	6

Here we can see that the graph has six nodes with a total number of three triangles. Comparing this to the [stream example](#) we can see that the 'Michael' node has a triangle count equal to the global triangle count. In other words, that node is part of all of the triangles in the graph and thus has a very central position in the graph.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the triangle count for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.triangleCount.mutate('myGraph', {
  mutateProperty: 'triangles'
})
YIELD globalTriangleCount, nodeCount
```

Table 432. Results

globalTriangleCount	nodeCount
3	6

The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `triangles` which stores the triangle count for each node. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs](#).

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the triangle count for each node as a property to the Neo4j database. The name of the new property is specified using

the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.triangleCount.write('myGraph', {
  writeProperty: 'triangles'
})
YIELD globalTriangleCount, nodeCount
```

Table 433. Results

globalTriangleCount	nodeCount
3	6

The returned result is the same as in the `stats` example. Additionally, each of the six nodes now has a new property `triangles` in the Neo4j database, containing the triangle count for that node.

## Maximum Degree

The Triangle Count algorithm supports a `maxDegree` configuration parameter that can be used to exclude nodes from processing if their degree is greater than the configured value. This can be useful to speed up the computation when there are nodes with a very high degree (so-called super nodes) in the graph. Super nodes have a great impact on the performance of the Triangle Count algorithm. To learn about the degree distribution of your graph, see [Listing graphs](#).

The nodes excluded from the computation get assigned a triangle count of `-1`.

The following will run the algorithm in `stream` mode with the `maxDegree` parameter:

```
CALL gds.triangleCount.stream('myGraph', {
  maxDegree: 4
})
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).name AS name, triangleCount
ORDER BY name ASC
```

Table 434. Results

name	triangleCount
"Alice"	0
"Chris"	0
"Karin"	0
"Mark"	0
"Michael"	-1
"Will"	0

Running the algorithm on the example graph with `maxDegree: 4` excludes the 'Michael' node from the computation, as it has a degree of 5.

As this node is part of all the triangles in the example graph excluding it results in no triangles.

## Triangles listing

It is also possible to list all the triangles in the graph. To do this we make use of the `alpha` procedure `gds.alpha.triangles`.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

The following will compute a stream of node IDs for each triangle and return the name property of the nodes:

```
CALL gds.alpha.triangles('myGraph')
YIELD nodeA, nodeB, nodeC
RETURN
  gds.util.asNode(nodeA).name AS nodeA,
  gds.util.asNode(nodeB).name AS nodeB,
  gds.util.asNode(nodeC).name AS nodeC
```

Table 435. Results

nodeA	nodeB	nodeC
"Michael"	"Karin"	"Chris"
"Michael"	"Chris"	"Will"
"Michael"	"Will"	"Mark"

We can see that there are three triangles in the graph: "Will, Michael, and Chris", "Will, Mark, and Michael", and "Michael, Karin, and Chris". The node "Alice" is not part of any triangle and thus does not appear in the triangles listing.

## 7.3.5. Local Clustering Coefficient

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

The Local Clustering Coefficient algorithm computes the local clustering coefficient for each node in the graph. The local clustering coefficient  $C_n$  of a node  $n$  describes the likelihood that the neighbours of  $n$  are

also connected. To compute  $C_n$  we use the number of triangles a node is a part of  $T_n$ , and the degree of the node  $d_n$ . The formula to compute the local clustering coefficient is as follows:

$$C_n = \frac{2T_n}{d_n(d_n - 1)}$$

As we can see the triangle count is required to compute the local clustering coefficient. To do this the [Triangle Count](#) algorithm is utilised.

Additionally, the algorithm can compute the average *clustering coefficient* for the whole graph. This is the normalised sum over all the local clustering coefficients.

For more information, see [Clustering Coefficient](#).

## Syntax

This section covers the syntax used to execute the Local Clustering Coefficient algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

Run Local Clustering Coefficient in stream mode on a named graph:

```
CALL gds.localClusteringCoefficient.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  localClusteringCoefficient: Double
```

Table 436. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 437. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 438. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 439. Results

Name	Type	Description
nodeId	Integer	Node ID.
localClusteringCoefficient	Double	Local clustering coefficient.



Run Local Clustering Coefficient in stats mode on a named graph:

```
CALL gds.localClusteringCoefficient.stats(
  graphName: String,
  configuration: Map
)
YIELD
  averageClusteringCoefficient: Double,
  nodeCount: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  configuration: Map
```

Table 440. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 441. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 442. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 443. Results

Name	Type	Description
averageClusteringCoefficient	Double	The average clustering coefficient.
nodeCount	Integer	Number of nodes in the graph.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
configuration	Map	The configuration used for running the algorithm.

Run Local Clustering Coefficient in mutate mode on a named graph:

```
CALL gds.localClusteringCoefficient.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  averageClusteringCoefficient: Double,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  configuration: Map
```

Table 444. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 445. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 446. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 447. Results

Name	Type	Description
averageClusteringCoefficient	Double	The average clustering coefficient.
nodeCount	Integer	Number of nodes in the graph.
nodePropertiesWritten	Integer	Number of properties added to the in-memory graph.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

Name	Type	Description
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
configuration	Map	The configuration used for running the algorithm.

Run Local Clustering Coefficient in write mode on a named graph:

```
CALL gds.localClusteringCoefficient.write(
  graphName: String,
  configuration: Map
)
YIELD
  averageClusteringCoefficient: Double,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 448. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 449. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 450. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 451. Results

Name	Type	Description
averageClusteringCoefficient	Double	The average clustering coefficient.
nodeCount	Integer	Number of nodes in the graph.
nodePropertiesWritten	Integer	Number of properties written to Neo4j.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
writeMillis	Integer	Milliseconds for writing results back to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Local Clustering Coefficient in write mode on an anonymous graph:

```
CALL gds.localClusteringCoefficient.write(
  configuration: Map
)
YIELD
  averageClusteringCoefficient: Double,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 452. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation via a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.

Name	Type	Default	Optional	Description
relationshipPr operties	String, List of String or Map	<code>null</code>	yes	The relationship properties to project during anonymous graph creation.
<code>concurrency</code>	Integer	<code>4</code>	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurre ncy	Integer	<code>value of 'concurrency ,</code>	yes	The number of concurrent threads used for creating the graph.
<code>writeConcurre ncy</code>	Integer	<code>value of 'concurrency ,</code>	yes	The number of concurrent threads used for writing the result to Neo4j.

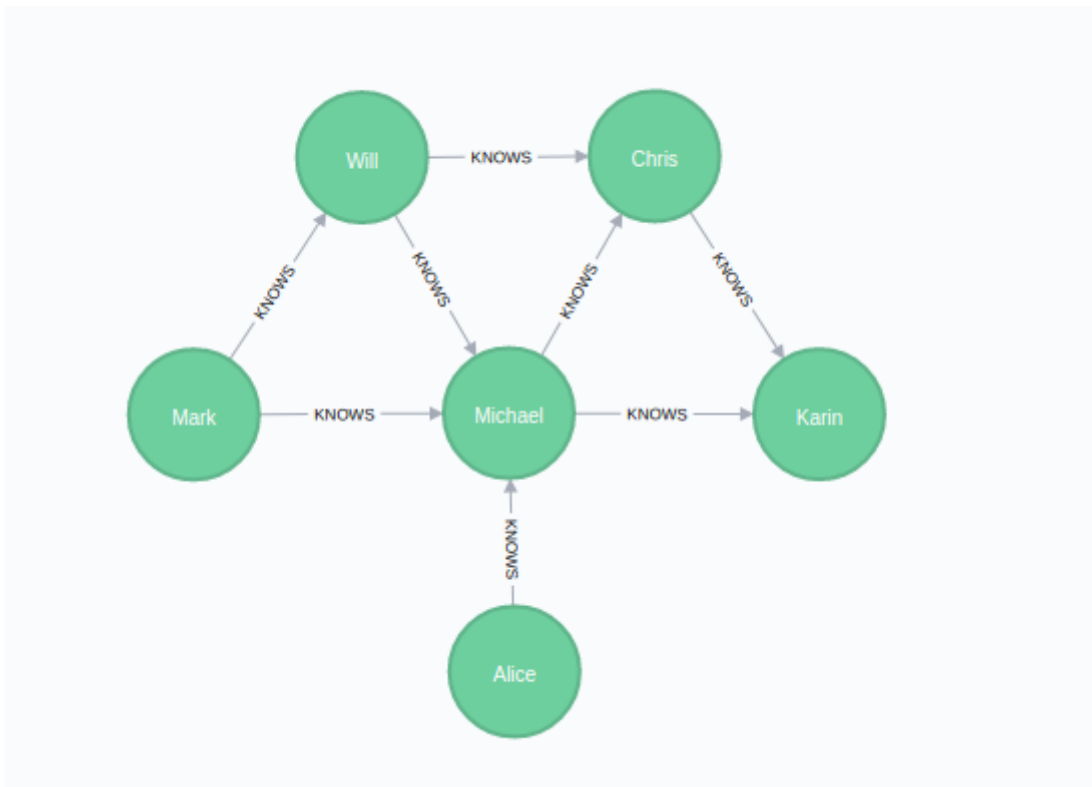
Table 453. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCount Property	String	<code>n/a</code>	Yes	Node property that contains pre-computed triangle count.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Local Clustering Coefficient algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
  (alice:Person {name: 'Alice'}),
  (michael:Person {name: 'Michael'}),
  (karin:Person {name: 'Karin'}),
  (chris:Person {name: 'Chris'}),
  (will:Person {name: 'Will'}),
  (mark:Person {name: 'Mark'}),

  (michael)-[:KNOWS]->(karin),
  (michael)-[:KNOWS]->(chris),
  (will)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(will),
  (alice)-[:KNOWS]->(michael),
  (will)-[:KNOWS]->(chris),
  (chris)-[:KNOWS]->(karin)
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. For the relationships we must use the `UNDIRECTED` orientation. This is because the Local Clustering Coefficient algorithm is defined only for undirected graphs.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)
```



The Local Clustering Coefficient algorithm requires the graph to be created using the `UNDIRECTED` orientation for relationships.

In the following examples we will demonstrate using the Local Clustering Coefficient algorithm on 'myGraph'.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.localClusteringCoefficient.write.estimate('myGraph', {
  writeProperty: 'localClusteringCoefficient'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 454. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	16	296	296	"296 Bytes"

Note that the relationship count is 16 although we only created 8 relationships in the original Cypher statement. This is because we used the **UNDIRECTED** orientation, which will project each relationship in each direction, effectively doubling the number of relationships.

## Stream

In the **stream** execution mode, the algorithm returns the local clustering coefficient for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the **stream** mode in general, see [Stream](#).

The following will run the algorithm in **stream** mode:

```
CALL gds.localClusteringCoefficient.stream('myGraph')
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).name AS name, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

Table 455. Results

name	localClusteringCoefficient
"Karin"	1.0
"Mark"	1.0
"Chris"	0.6666666666666666
"Will"	0.6666666666666666
"Michael"	0.3
"Alice"	0.0

From the results we can see that the nodes 'Karin' and 'Mark' have the highest local clustering coefficients. This shows that they are the best at introducing their friends - all the people who know them, know each other! This can be verified in the [example graph](#).

## Stats

In the **stats** execution mode, the algorithm returns a single row containing a summary of the algorithm



result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm in `stats` mode:

```
CALL gds.localClusteringCoefficient.stats('myGraph')
YIELD averageClusteringCoefficient, nodeCount
```

Table 456. Results

averageClusteringCoefficient	nodeCount
0.6055555555555555	6

The result shows that on average each node of our example graph has approximately 60% of its neighbours connected.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the local clustering coefficient for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.localClusteringCoefficient.mutate('myGraph', {
  mutateProperty: 'localClusteringCoefficient'
})
YIELD averageClusteringCoefficient, nodeCount
```

Table 457. Results

averageClusteringCoefficient	nodeCount
0.6055555555555555	6

The returned result is the same as in the `stats` example. Additionally, the graph 'myGraph' now has a node property `localClusteringCoefficient` which stores the local clustering coefficient for each node. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs](#).

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the local clustering coefficient for each node as a property to the Neo4j database. The name of the new property is

specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.localClusteringCoefficient.write('myGraph', {
  writeProperty: 'localClusteringCoefficient'
})
YIELD averageClusteringCoefficient, nodeCount
```

Table 458. Results

averageClusteringCoefficient	nodeCount
0.6055555555555555	6

The returned result is the same as in the `stats` example. Additionally, each of the six nodes now has a new property `localClusteringCoefficient` in the Neo4j database, containing the local clustering coefficient for that node.

## Pre-computed Counts

By default, the Local Clustering Coefficient algorithm executes [Triangle Count](#) as part of its computation. It is also possible to avoid the triangle count computation by configuring the Local Clustering Coefficient algorithm to read the triangle count from a node property. In order to do that we specify the `triangleCountProperty` configuration parameter. Please note that the Local Clustering Coefficient algorithm depends on the property holding actual triangle counts and not another number for the results to be actual local clustering coefficients.

To illustrate this we make use of the [Triangle Count algorithm](#) in `mutate` mode. The Triangle Count algorithm is going to store its result back into 'myGraph'. It is also possible to obtain the property value from the Neo4j database using a graph projection with a node property when creating the in-memory graph.

The following computes the triangle counts and stores the result into the in-memory graph:

```
CALL gds.triangleCount.mutate('myGraph', {
  mutateProperty: 'triangles'
})
```

The following will run the algorithm in `stream` mode using pre-computed triangle counts:

```
CALL gds.localClusteringCoefficient.stream('myGraph', {
  triangleCountProperty: 'triangles'
})
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).name AS name, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

Table 459. Results

name	localClusteringCoefficient
"Karin"	1.0
"Mark"	1.0
"Chris"	0.6666666666666666
"Will"	0.6666666666666666
"Michael"	0.3
"Alice"	0.0

As we can see the results are the same as in [the stream example](#) where we did not specify a `triangleCountProperty`.

### 7.3.6. K-1 Coloring Beta

This algorithm is in the beta tier. For more information on algorithm tiers, see [Algorithms](#).

#### Introduction

The K-1 Coloring algorithm assigns a color to every node in the graph, trying to optimize for two objectives:

1. To make sure that every neighbor of a given node has a different color than the node itself.
2. To use as few colors as possible.

Note that the graph coloring problem is proven to be NP-complete, which makes it intractable on anything but trivial graph sizes. For that reason the implemented algorithm is a greedy algorithm. Thus it is neither guaranteed that the result is an optimal solution, using as few colors as theoretically possible, nor does it always produce a correct result where no two neighboring nodes have different colors. However the precision of the latter can be controlled by the number of iterations this algorithm runs.

For more information on this algorithm, see:

- [Çatalyürek, Ümit V., et al. "Graph coloring algorithms for multi-core and massively multithreaded architectures."](#)
- [https://en.wikipedia.org/wiki/Graph\\_coloring#Vertex\\_coloring](https://en.wikipedia.org/wiki/Graph_coloring#Vertex_coloring)



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

#### Syntax

The following describes the API for running the algorithm and stream results:

```
CALL gds.beta.k1coloring.stream(graphName: String, configuration: Map)
YIELD nodeId, color
```

Table 460. Parameters

Name	Type	Default	Optional	Description
graphName	String	<code>null</code>	yes	The name of an existing graph on which to run the algorithm. If no graph name is provided, the configuration map must contain configuration for creating a graph.
configuration	Map	<code>{}</code>	yes	Additional configuration, see below.

Table 461. Configuration

Name	Type	Default	Optional	Description
nodeProjection	String	<code>null</code>	yes	The projection of nodes to use when creating the implicit graph.
relationshipProjection	String	<code>null</code>	yes	The projection of relationships to use when creating the implicit graph.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see <a href="#">CPU</a> .
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
<a href="#">maxIterations</a>	Integer	10	yes	The maximum number of iterations of K1 Coloring to run.

Table 462. Results

Name	Type	Description
nodeId	Integer	The ID of the Node
color	Integer	The color of the Node

The following describes the API for running the algorithm and returning the computation statistics:

```
CALL gds.beta.k1coloring.stats(
  graphName: String,
  configuration: Map
)
YIELD
  nodeCount,
  colorCount,
  ranIterations,
  didConverge,
  configuration,
  createMillis,
  computeMillis
```

Table 463. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{ }	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

Table 464. Configuration

Name	Type	Default	Optional	Description
nodeProjection	String	null	yes	The projection of nodes to use when creating the implicit graph.
relationshipProjection	String	null	yes	The projection of relationships to use when creating the implicit graph.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see <a href="#">CPU</a> .
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
maxIterations	Integer	10	yes	The maximum number of iterations of K1 Coloring to run.

Table 465. Results

Name	Type	Description
nodeCount	Integer	The number of nodes considered.
ranIterations	Integer	The actual number of iterations the algorithm ran.
didConverge	Boolean	An indicator of whether the algorithm found a correct coloring.

Name	Type	Description
colorCount	Integer	The number of colors used.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
configuration	Map	The configuration used for running the algorithm.

The following describes the API for running the algorithm and mutating the in-memory graph:

```
CALL gds.beta.k1coloring.mutate(graphName: String, configuration: Map)
YIELD nodeCount, colorCount, ranIterations, didConverge, configuration, createMillis,
computeMillis, mutateMillis
```

Table 466. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

The configuration for the `mutate` mode is similar to the `write` mode. Instead of specifying a `writeProperty`, we need to specify a `mutateProperty`. Also, specifying `writeConcurrency` is not possible in `mutate` mode.

Table 467. Results

Name	Type	Description
nodeCount	Integer	The number of nodes considered.
ranIterations	Integer	The actual number of iterations the algorithm ran.
didConverge	Boolean	An indicator of whether the algorithm found a correct coloring.
colorCount	Integer	The number of colors used.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
configuration	Map	The configuration used for running the algorithm.

The following describes the API for running the algorithm and writing results back to Neo4j:

```
CALL gds.beta.k1coloring.write(graphName: String, configuration: Map)
YIELD nodeCount, colorCount, ranIterations, didConverge, configuration, createMillis,
computeMillis, writeMillis
```

Table 468. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

Table 469. Configuration

Name	Type	Default	Optional	Description
nodeProjection	String	null	yes	The projection of nodes to use when creating the implicit graph.
relationshipProjection	String	null	yes	The projection of relationships to use when creating the implicit graph.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see <a href="#">CPU</a> .
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result.
maxIterations	Integer	10	yes	The maximum number of iterations of K1 Coloring to run.
writeProperty	String	n/a	no	The node property this procedure writes the color to.

Table 470. Results

Name	Type	Description
nodeCount	Integer	The number of nodes considered.
ranIterations	Integer	The actual number of iterations the algorithm ran.
didConverge	Boolean	An indicator of whether the algorithm found a correct coloring.
colorCount	Integer	The number of colors used.

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Examples

Consider the graph created by the following Cypher statement:

```
CREATE (alice:User {name: 'Alice'}),
      (bridget:User {name: 'Bridget'}),
      (charles:User {name: 'Charles'}),
      (doug:User {name: 'Doug'}),

      (alice)-[:LINK]->(bridget),
      (alice)-[:LINK]->(charles),
      (alice)-[:LINK]->(doug),
      (bridget)-[:LINK]->(charles)
```

This graph has a super node with name "Alice" that connects to all other nodes. It should therefore not be possible for any other node to be assigned the same color as the Alice node.

```
CALL gds.graph.create(
  'myGraph',
  'User',
  {
    LINK : {
      orientation: 'UNDIRECTED'
    }
  }
)
```

We can now go ahead and create an in-memory graph with all the `User` nodes and the `LINK` relationships with `UNDIRECTED` orientation.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create('myGraph', 'Person', 'LIKES')
```

In the following examples we will demonstrate using the K-1 Coloring algorithm on this graph.



Running the K-1 Coloring algorithm in stream mode:

```
CALL gds.beta.k1coloring.stream('myGraph')
YIELD nodeId, color
RETURN gds.util.asNode(nodeId).name AS name, color
ORDER BY name
```

Table 471. Results

name	color
"Alice"	0
"Bridget"	1
"Charles"	2
"Doug"	1

It is also possible to write the assigned colors back to the database using the `write` mode.

Running the K-1 Coloring algorithm in write mode:

```
CALL gds.beta.k1coloring.write('myGraph', {writeProperty: 'color'})
YIELD nodeCount, colorCount, ranIterations, didConverge
```

Table 472. Results

nodeCount	colorCount	ranIterations	didConverge
4	3	1	true

When using `write` mode the procedure will return information about the algorithm execution. In this example we return the number of processed nodes, the number of colors used to color the graph, the number of iterations and information whether the algorithm converged.

To instead mutate the in-memory graph with the assigned colors, the `mutate` mode can be used as follows.

Running the K-1 Coloring algorithm in mutate mode:

```
CALL gds.beta.k1coloring.mutate('myGraph', {mutateProperty: 'color'})
YIELD nodeCount, colorCount, ranIterations, didConverge
```

Table 473. Results

nodeCount	colorCount	ranIterations	didConverge
4	3	1	true

Similar to the `write` mode, `stats` mode can run the algorithm and return only the execution statistics without persisting the results.

Running the K-1 Coloring algorithm in stats mode:

```
CALL gds.beta.k1coloring.stats('myGraph')
YIELD nodeCount, colorCount, ranIterations, didConverge
```

Table 474. Results

nodeCount	colorCount	ranIterations	didConverge
4	3	1	true

### 7.3.7. Modularity Optimization Beta

This algorithm is in the beta tier. For more information on algorithm tiers, see [Algorithms](#).

#### Introduction

The Modularity Optimization algorithm tries to detect communities in the graph based on their *modularity*. *Modularity* is a measure of the structure of a graph, measuring the density of connections within a module or community. Graphs with a high modularity score will have many connections within a community but only few pointing outwards to other communities. The algorithm will explore for every node if its modularity score might increase if it changes its community to one of its neighboring nodes.

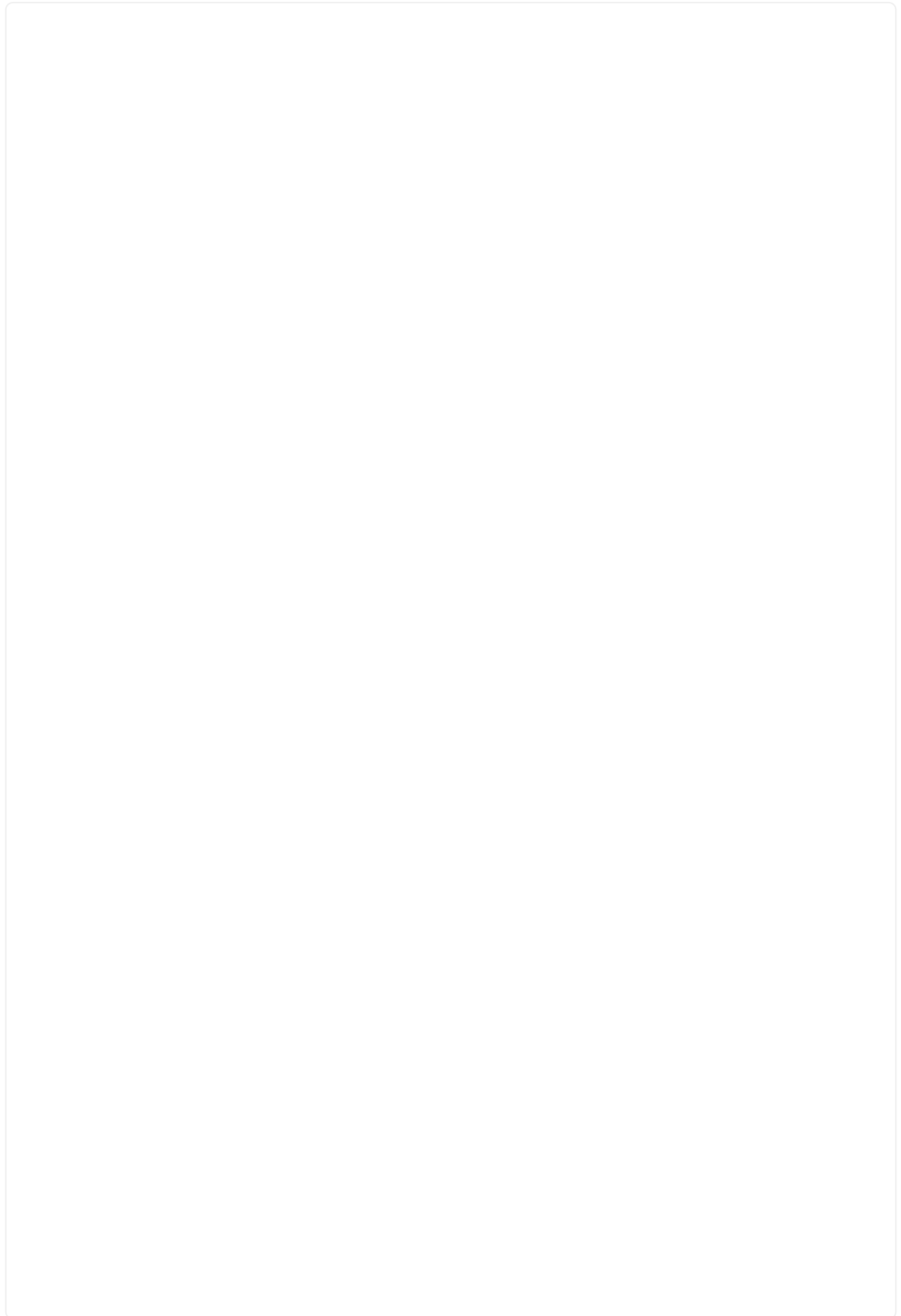
For more information on this algorithm, see:

- [MEJ Newman, M Girvan "Finding and evaluating community structure in networks"](#)
- [https://en.wikipedia.org/wiki/Modularity\\_\(networks\)](https://en.wikipedia.org/wiki/Modularity_(networks))



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).

#### Syntax



Run Modularity Optimization in stream mode on a named graph.

```
CALL gds.beta.modularityOptimization.stream(graphName: String, configuration: Map)
YIELD
  nodeId: Integer,
  communityId: Integer
```

Table 475. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

Table 476. General configuration

Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result (applicable in WRITE mode).
nodeProjection	Map or List	null	yes	The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph.
relationshipProjection	Map or List	null	yes	The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for implicit graph loading via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for implicit graph loading via a Cypher projection.
nodeProperties	Map or List	null	yes	The node properties to load during implicit graph loading.
relationshipProperties	Map or List	null	yes	The relationship properties to load during implicit graph loading.

Table 477. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.

Name	Type	Default	Optional	Description
<a href="#">tolerance</a>	Float	0.0001	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.
<a href="#">seedProperty</a>	String	n/a	yes	Used to define initial set of labels (must be a number).
<a href="#">consecutiveIds</a>	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 478. Results

Name	Type	Description
<a href="#">nodeId</a>	Integer	Node ID
<a href="#">communityId</a>	Integer	Community ID

Run Modularity Optimization in mutate mode on a named graph.

```
CALL gds.beta.modularityOptimization.mutate(graphName: String|Map, configuration: Map})
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  communityCount: Integer,
  communityDistribution: Map,
  modularity: Float,
  ranIterations: Integer,
  didConverge: Boolean,
  nodes: Integer,
  configuration: Map
```

Table 479. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

The configuration for the `mutate` mode is similar to the `write` mode. Instead of specifying a `writeProperty`, we need to specify a `mutateProperty`. Also, specifying `writeConcurrency` is not possible in `mutate` mode.

Table 480. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
nodes	Integer	The number of nodes considered.
didConverge	Boolean	True if the algorithm did converge to a stable modularity score within the provided number of maximum iterations.
ranIterations	Integer	The number of iterations run.
modularity	Float	The final modularity score.
communityCount	Integer	The number of communities found.
communityDistribution	Map	The containing min, max, mean as well as 50, 75, 90, 95, 99 and 999 percentile of community size.

Name	Type	Description
configuration	Map	The configuration used for running the algorithm.

Run Modularity Optimization in write mode on a named graph.

```
CALL gds.beta.modularityOptimization.write(graphName: String|Map, configuration: Map))
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  communityCount: Integer,
  communityDistribution: Map,
  modularity: Float,
  ranIterations: Integer,
  didConverge: Boolean,
  nodes: Integer,
  configuration: Map
```

Table 481. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

Table 482. General configuration

Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result (applicable in WRITE mode).
nodeProjection	Map or List	null	yes	The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph.
relationshipProjection	Map or List	null	yes	The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for implicit graph loading via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for implicit graph loading via a Cypher projection.
nodeProperties	Map or List	null	yes	The node properties to load during implicit graph loading.
relationshipProperties	Map or List	null	yes	The relationship properties to load during implicit graph loading.



Table 483. Algorithm specific configuration

Name	Type	Default	Optional	Description
seedProperty	String	n/a	yes	Used to set the initial community for a node. The property value needs to be a number.
writeProperty	String	n/a	yes	The property name written back the ID of the partition particular node belongs to.
maxIterations	Integer	10	yes	The maximum number of iterations that the modularity optimization will run for each level.
tolerance	Float	0.0001	yes	Minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns.
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 484. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
nodes	Integer	The number of nodes considered.
didConverge	Boolean	True if the algorithm did converge to a stable modularity score within the provided number of maximum iterations.
ranIterations	Integer	The number of iterations run.
modularity	Float	The final modularity score.
communityCount	Integer	The number of communities found.
communityDistribution	Map	The containing min, max, mean as well as 50, 75, 90, 95, 99 and 999 percentile of community size.
configuration	Map	The configuration used for running the algorithm.

## Examples

Consider the graph created by the following Cypher statement:

```
CREATE
  (a:Person {name:'Alice'})
  , (b:Person {name:'Bridget'})
  , (c:Person {name:'Charles'})
  , (d:Person {name:'Doug'})
  , (e:Person {name:'Elton'})
  , (f:Person {name:'Frank'})
  , (a)-[:KNOWS {weight: 0.01}]->(b)
  , (a)-[:KNOWS {weight: 5.0}]->(e)
  , (a)-[:KNOWS {weight: 5.0}]->(f)
  , (b)-[:KNOWS {weight: 5.0}]->(c)
  , (b)-[:KNOWS {weight: 5.0}]->(d)
  , (c)-[:KNOWS {weight: 0.01}]->(e)
  , (f)-[:KNOWS {weight: 0.01}]->(d)
```

This graph consists of two center nodes "Alice" and "Bridget" each of which have two more neighbors. Additionally, each neighbor of "Alice" is connected to one of the neighbors of "Bridget". Looking at the weights of the relationships, it can be seen that the connections from the two center nodes to their neighbors are very strong, while connections between those groups are weak. Therefore the Modularity Optimization algorithm should detect two communities: "Alice" and "Bob" together with their neighbors respectively.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  {
    KNOWS: {
      type: 'KNOWS',
      orientation: 'UNDIRECTED',
      properties: ['weight']
    }
  }
)
```

The following example demonstrates using the Modularity Algorithm on this weighted graph.

Running the Modularity Optimization algorithm in stream mode:

```
CALL gds.beta.modularityOptimization.stream('myGraph', { relationshipWeightProperty: 'weight' })
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId
ORDER BY name
```

Table 485. Results

name	communityId
"Alice"	4
"Bridget"	1
"Charles"	1

name	communityId
"Doug"	1
"Elton"	4
"Frank"	4

It is also possible to write the assigned community ids back to the database using the `write` mode.

Running the Modularity Optimization algorithm in write mode:

```
CALL gds.beta.modularityOptimization.write('myGraph', { relationshipWeightProperty: 'weight',
writeProperty: 'community' })
YIELD nodes, communityCount, ranIterations, didConverge
```

Table 486. Results

nodes	communityCount	ranIterations	didConverge
6	2	3	true

When using `write` mode the procedure will return information about the algorithm execution. In this example we return the number of processed nodes, the number of communities assigned to the nodes in the graph, the number of iterations and information whether the algorithm converged.

Running the algorithm without specifying the `relationshipWeightProperty` will default all relationship weights to 1.0.

To instead mutate the in-memory graph with the assigned community ids, the `mutate` mode is used.

Running the Modularity Optimization algorithm in mutate mode:

```
CALL gds.beta.modularityOptimization.mutate('myGraph', { relationshipWeightProperty: 'weight',
mutateProperty: 'community' })
YIELD nodes, communityCount, ranIterations, didConverge
```

Table 487. Results

nodes	communityCount	ranIterations	didConverge
6	2	3	true

When using `mutate` mode the procedure will return information about the algorithm execution as in `write` mode.

### 7.3.8. Strongly Connected Components Alpha

The Strongly Connected Components (SCC) algorithm finds maximal sets of connected nodes in a directed graph. A set is considered a strongly connected component if there is a directed path between each pair of nodes within the set. It is often used early in a graph analysis process to help us get an idea of how our graph is structured.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

## History and explanation

SCC is one of the earliest graph algorithms, and the first linear-time algorithm was described by Tarjan in 1972. Decomposing a directed graph into its strongly connected components is a classic application of the depth-first search algorithm.

## Use-cases - when to use the Strongly Connected Components algorithm

- In the analysis of powerful transnational corporations, SCC can be used to find the set of firms in which every member owns directly and/or indirectly owns shares in every other member. Although it has benefits, such as reducing transaction costs and increasing trust, this type of structure can weaken market competition. Read more in "[The Network of Global Corporate Control](#)".
- SCC can be used to compute the connectivity of different network configurations when measuring routing performance in multihop wireless networks. Read more in "[Routing performance in the presence of unidirectional links in multihop wireless networks](#)".
- Strongly Connected Components algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph. In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages, or play common games. The SCC algorithms can be used to find such groups, and suggest the commonly liked pages or games to the people in the group who have not yet liked those pages or games.

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.scc.write(graphName: String|Map, configuration: Map)
YIELD createMillis, computeMillis, writeMillis, setCount, maxSetSize, minSetSize
```

Table 488. Parameters

Name	Type	Default	Optional	Description
writeProperty	String	'componentId'	yes	The property name written back to.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result.

Table 489. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.

Name	Type	Description
writeMillis	Integer	Milliseconds for writing result data back.
postProcessingMillis	Integer	Milliseconds for computing percentiles and community count.
nodes	Integer	The number of nodes considered.
communityCount	Integer	The number of communities found.
p1	Float	The 1 percentile of community size.
p5	Float	The 5 percentile of community size.
p10	Float	The 10 percentile of community size.
p25	Float	The 25 percentile of community size.
p50	Float	The 50 percentile of community size.
p75	Float	The 75 percentile of community size.
p90	Float	The 90 percentile of community size.
p95	Float	The 95 percentile of community size.
p99	Float	The 99 percentile of community size.
p100	Float	The 100 percentile of community size.
writeProperty	String	The property name written back to.

The following will run the algorithm and stream results:

```
CALL gds.alpha.scc.stream(graphName: String, configuration: Map)
YIELD nodeId, componentId
```

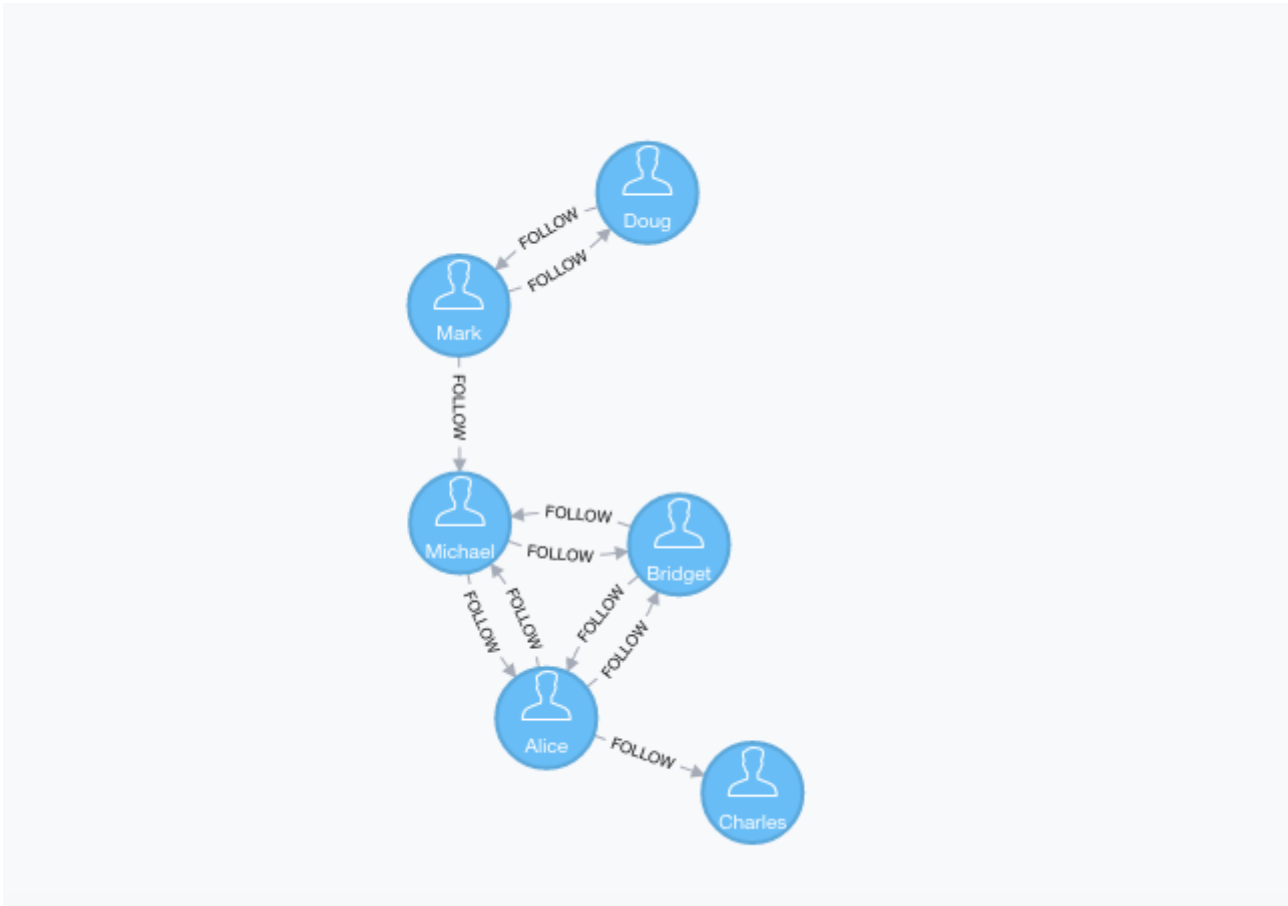
Table 490. Parameters

Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

Table 491. Results

Name	Type	Description
nodeId	Integer	Node ID.
componentId	Integer	Component ID.

## Strongly Connected Components algorithm example



The following will create a sample graph:

```
CREATE (nAlice:User {name:'Alice'})
CREATE (nBridget:User {name:'Bridget'})
CREATE (nCharles:User {name:'Charles'})
CREATE (nDoug:User {name:'Doug'})
CREATE (nMark:User {name:'Mark'})
CREATE (nMichael:User {name:'Michael'})

CREATE (nAlice)-[:FOLLOW]->(nBridget)
CREATE (nAlice)-[:FOLLOW]->(nCharles)
CREATE (nMark)-[:FOLLOW]->(nDoug)
CREATE (nMark)-[:FOLLOW]->(nMichael)
CREATE (nBridget)-[:FOLLOW]->(nMichael)
CREATE (nDoug)-[:FOLLOW]->(nMark)
CREATE (nMichael)-[:FOLLOW]->(nAlice)
CREATE (nAlice)-[:FOLLOW]->(nMichael)
CREATE (nBridget)-[:FOLLOW]->(nAlice)
CREATE (nMichael)-[:FOLLOW]->(nBridget);
```

The following will run the algorithm and write back results:

```
CALL gds.alpha.scc.write({
  nodeProjection: 'User',
  relationshipProjection: 'FOLLOW',
  writeProperty: 'componentId'
})
YIELD setCount, maxSetSize, minSetSize;
```

Table 492. Results

setCount	maxSetSize	minSetSize
3	3	1

The following will run the algorithm and stream back results:

```
CALL gds.alpha.scc.stream({
  nodeProjection: 'User',
  relationshipProjection: 'FOLLOW'
})
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS Name, componentId AS Component
ORDER BY Component DESC
```

Table 493. Results

Name	Component
"Doug"	3
"Mark"	3
"Charles"	2
"Alice"	0
"Bridget"	0
"Michael"	0

We have 3 strongly connected components in our sample graph.

The first, and biggest, component has members Alice, Bridget, and Michael, while the second component has Doug and Mark. Charles ends up in his own component because there isn't an outgoing relationship from that node to any of the others.

The following will find the largest partition:

```
MATCH (u:User)
RETURN u.componentId AS Component, count(*) AS ComponentSize
ORDER BY ComponentSize DESC
LIMIT 1
```

Table 494. Results

Component	ComponentSize
0	3

## Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the [Creating graphs using Cypher](#) section of the manual.

Use `nodeQuery` and `relationshipQuery` in the config:

```
CALL gds.alpha.scc.stream({
  nodeQuery: 'MATCH (u:User) RETURN id(u) AS id',
  relationshipQuery: 'MATCH (u1:User)-[:FOLLOW]->(u2:User) RETURN id(u1) AS source, id(u2) AS target' })
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS Name, componentId AS Component
ORDER BY Component DESC
```

Table 495. Results

Name	Component
"Doug"	3
"Mark"	3
"Charles"	2
"Alice"	0
"Bridget"	0
"Michael"	0

## References

- <https://pdfs.semanticscholar.org/61db/6892a92d1d5bdc83e52cc18041613cf895fa.pdf>
- <http://code.activestate.com/recipes/578507-strongly-connected-components-of-a-directed-graph/>
- [http://www.sandia.gov/~srajama/publications/BFS\\_and\\_Coloring.pdf](http://www.sandia.gov/~srajama/publications/BFS_and_Coloring.pdf)

## 7.3.9. Speaker-Listener Label Propagation Alpha

### Introduction

The Speaker-Listener Label Propagation Algorithm (SLLPA) is a variation of the Label Propagation algorithm that is able to detect multiple communities per node. The GDS implementation is based on the [SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process](#) publication by Xie et al.

The algorithm is randomized in nature and will not produce deterministic results. To accommodate this, we recommend using a higher number of iterations.

### Syntax

This section covers the syntax used to execute the SLLPA algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run SLLPA in stream mode on a named graph.

```
CALL gds.alpha.sllpa.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  values: Map {
    communitiyIds: List of Integer
  }
```

Table 496. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 497. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 498. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	n/a	no	Maximum number of iterations to run.
minAssociationStrength	String	0.2	yes	Minimum influence required for a community to retain a node.

Table 499. Results

Name	Type	Description
nodeId	Integer	Node ID.
values	Map	A map that contains the key <code>communityIds</code> .

Run SLLPA in stats mode on a named graph.

```
CALL gds.alpha.sllpa.stats(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  configuration: Map
```

Table 500. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 501. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 502. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	n/a	no	Maximum number of iterations to run.
minAssociationStrength	String	0.2	yes	Minimum influence required for a community to retain a node.

Table 503. Results

Name	Type	Description
ranIterations	Integer	Number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
configuration	Map	Configuration used for running the algorithm.

Run SLLPA in mutate mode on a named graph.

```
CALL gds.alpha.sllpa.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 504. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 505. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 506. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	n/a	no	Maximum number of iterations to run.
minAssociationStrength	String	0.2	yes	Minimum influence required for a community to retain a node.

Table 507. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.

Name	Type	Description
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
configuration	Map	The configuration used for running the algorithm.

Run SLLPA in write mode on a named graph.

```
CALL gds.alpha.sllpa.write(
  graphName: String,
  configuration: Map
)
YIELD
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 508. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 509. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 510. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	n/a	no	Maximum number of iterations to run.
minAssociationStrength	String	0.2	yes	Minimum influence required for a community to retain a node.

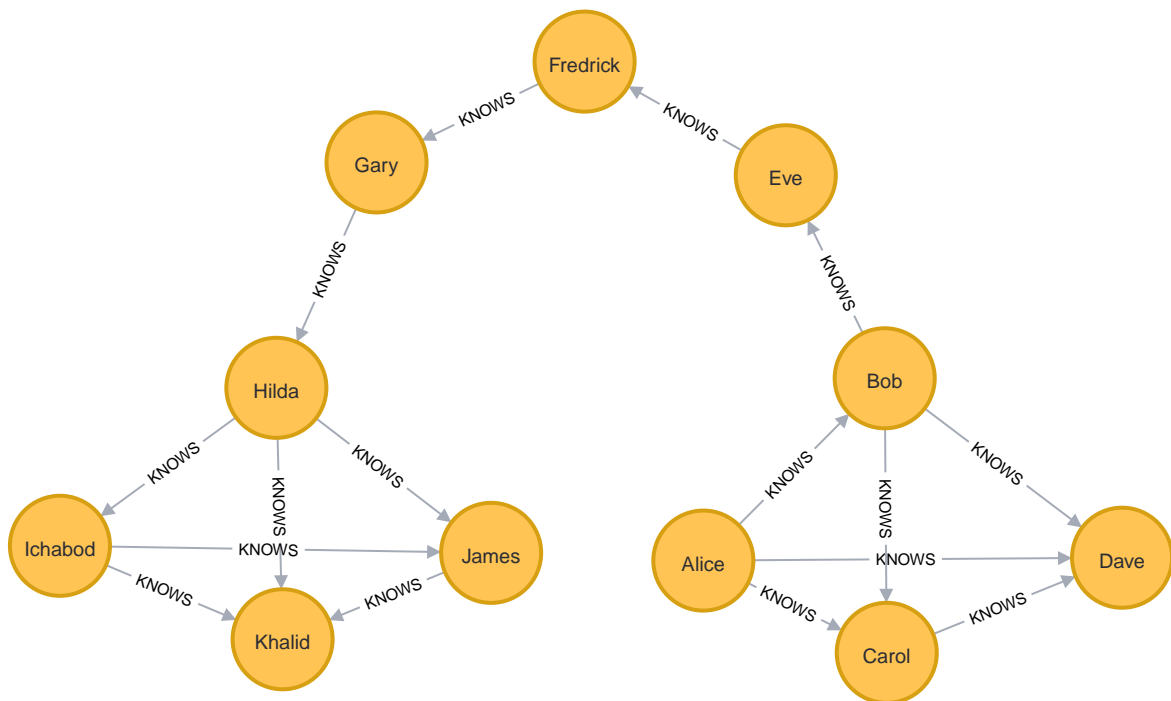
Table 511. Results

Name	Type	Description
ranIterations	Integer	The number of iterations run.
didConverge	Boolean	Indicates if the algorithm converged.
createMillis	Integer	Milliseconds for creating the graph.

Name	Type	Description
computeMilliseconds	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Examples

In this section we will show examples of running the SLLPA algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(a:Person {name: 'Alice'}),
(b:Person {name: 'Bob'}),
(c:Person {name: 'Carol'}),
(d:Person {name: 'Dave'}),
(e:Person {name: 'Eve'}),
(f:Person {name: 'Fredrick'}),
(g:Person {name: 'Gary'}),
(h:Person {name: 'Hilda'}),
(i:Person {name: 'Ichabod'}),
(j:Person {name: 'James'}),
(k:Person {name: 'Khalid'}),

(a)-[:KNOWS]->(b),
(a)-[:KNOWS]->(c),
(a)-[:KNOWS]->(d),
(b)-[:KNOWS]->(c),
(b)-[:KNOWS]->(d),
(c)-[:KNOWS]->(d),

(b)-[:KNOWS]->(e),
(e)-[:KNOWS]->(f),
(f)-[:KNOWS]->(g),
(g)-[:KNOWS]->(h),

(h)-[:KNOWS]->(i),
(h)-[:KNOWS]->(j),
(h)-[:KNOWS]->(k),
(i)-[:KNOWS]->(j),
(i)-[:KNOWS]->(k),
(j)-[:KNOWS]->(k);
```

In the example, we will use the SLLPA algorithm to find the communities in the graph.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
);
```

In the following examples we will demonstrate using the SLLPA algorithm on this graph.

## Stream

In the `stream` execution mode, the algorithm returns the community IDs for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.alpha.sllpa.stream('myGraph', {maxIterations: 100, minAssociationStrength: 0.1})
YIELD nodeId, values
RETURN gds.util.asNode(nodeId).name AS Name, values.communityIds AS communityIds
ORDER BY Name ASC
```

Table 512. Results

Name	communityIds
"Alice"	[0]
"Bob"	[0]
"Carol"	[0]
"Dave"	[0]
"Eve"	[0, 1]
"Fredrick"	[0, 1]
"Gary"	[0, 1]
"Hilda"	[1]
"Ichabod"	[1]
"James"	[1]
"Khalid"	[1]

Due to the randomness of the algorithm, the results will tend to vary between runs.

### 7.3.10. Approximate Maximum k-cut Alpha

#### Introduction

A k-cut of a graph is an assignment of its nodes into k disjoint communities. So for example a 2-cut of a graph with nodes *a, b, c, d* could be the communities *{a, b, c}* and *{d}*.

A Maximum k-cut is a k-cut such that the total weight of relationships between nodes from different communities in the k-cut is maximized. That is, a k-cut that maximizes the sum of weights of relationships whose source and target nodes are assigned to different communities in the k-cut. Suppose in the simple *a, b, c, d* node set example above we only had one relationship *b → c*, and it was of weight *1.0*. The 2-cut we outlined above would then not be a maximum 2-cut (with a cut cost of *0.0*), whereas for example the 2-cut with communities *{a, b}* and *{c, d}* would be one (with a cut cost of *1.0*).



Maximum k-cut is the same as [Maximum Cut](#) when  $k = 2$ .

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

#### Applications

Finding the maximum k-cut for a graph has several known applications, for example it is used to:

- analyze protein interaction
- design circuit (VLSI) layouts
- solve wireless communication problems



- analyze cryptocurrency transaction patterns
- design computer networks

## Approximation

In practice, finding the best cut is not feasible for larger graphs and only an approximation can be computed in reasonable time.

The approximate heuristic algorithm implemented in GDS is a parallelized [GRASP](#) style algorithm optionally enhanced (via config) with [variable neighborhood search \(VNS\)](#).

For detailed information about a serial version of the algorithm, with a slightly different construction phase, when  $k = 2$  see [GRASP+VNR](#) in the paper:

- [Festa et al. Randomized Heuristics for the Max-Cut Problem, 2002.](#)

To see how the algorithm above performs in terms of solution quality compared to other algorithms when  $k = 2$  see [FES02GV](#) in the paper:

- [Dunning et al. What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO, 2018.](#)



By the stochastic nature of the algorithm, the results it yields will not be deterministic unless running single-threaded (`concurrency = 1`) and using the same random seed (`randomSeed = SOME_FIXED_VALUE`).

## Tuning the algorithm parameters

There are two important algorithm specific parameters which lets you trade solution quality for shorter runtime.

### Iterations

GRASP style algorithms are iterative by nature. Every iteration they run the same well-defined steps to derive a solution, but each time with a different random seed yielding solutions that (highly likely) are different too. In the end the highest scoring solution is picked as the winner.

### VNS max neighborhood order

Variable neighborhood search (VNS) works by slightly perturbing a locally optimal solution derived from the previous steps in an iteration of the algorithm, followed by locally optimizing this perturbed solution. Perturb in this case means to randomly move some nodes from their current (locally optimal) community to another community.

VNS will in turn move `1, 2, ..., vnsMaxNeighborhoodOrder` random nodes and using each of the resulting solutions try to find a new locally optimal solution that's better. This means that although potentially better

solutions can be derived using VNS it will take more time, and additionally some more memory will be needed to temporarily store the perturbed solutions.

By default, VNS is not used (`vnsMaxNeighborhoodOrder = 0`). To use it, experimenting with a maximum order equal to `20` is a good place to start.

## Syntax

This section covers the syntax used to execute the Approximate Maximum k-cut algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

Example 1. Approximate Maximum k-cut syntax per mode



Run Approximate Maximum  $k$ -cut in stream mode on a named graph.

```
CALL gds.alpha.maxkcut.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  communityId: Integer
```

Table 513. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 514. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 515. Algorithm specific configuration

Name	Type	Default	Optional	Description
k	Integer	2	yes	The number of disjoint communities the nodes will be divided into.
iterations	Integer	8	yes	The number of iterations the algorithm will run before returning the best solution among all the iterations.
vnsMaxNeighborhoodOrder	Integer	0 (VNS off)	yes	The maximum number of nodes VNS will swap when perturbing solutions.
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in the computation. Requires <code>concurrency = 1</code> .
relationshipWeightProperty	String	null	yes	If set, the values stored at the given property are used as relationship weights during the computation. If not set, the graph is considered unweighted.

Table 516. Results

Name	Type	Description
nodeId	Integer	Node ID.

Name	Type	Description
communityId	Integer	Community ID.

Run Approximate Maximum  $k$ -cut in mutate mode on a named graph.

```
CALL gds.alpha.maxkcut.mutate(
  graphName: String,
  configuration: Map
) YIELD
  cutCost: Float,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 517. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 518. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 519. Algorithm specific configuration

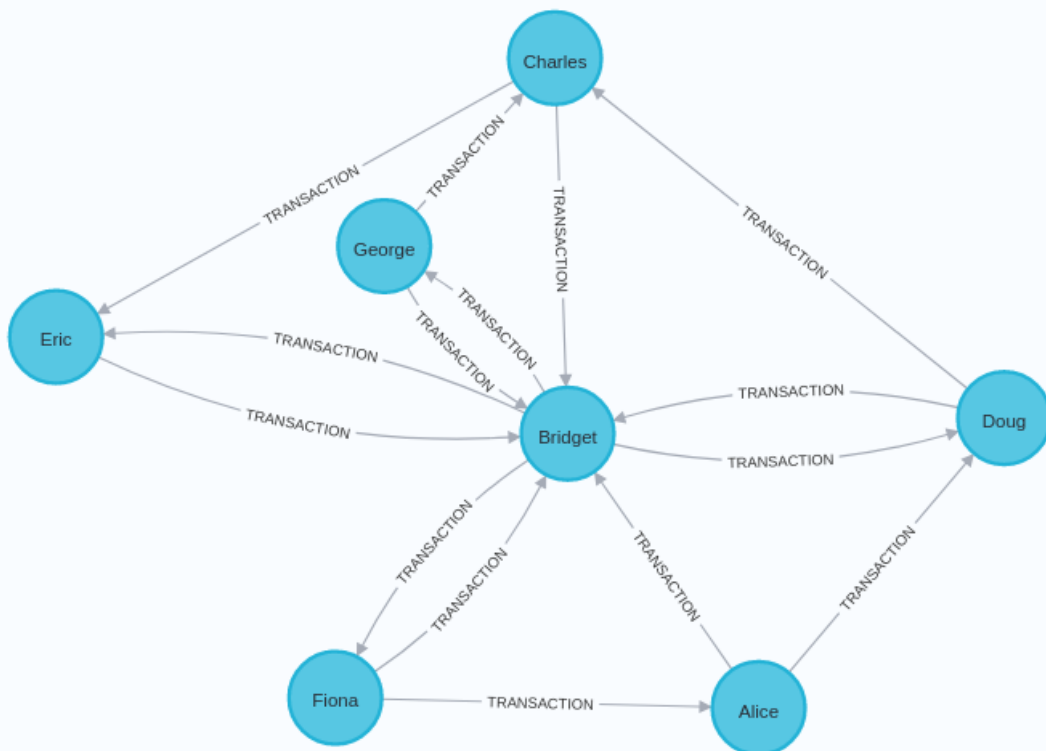
Name	Type	Default	Optional	Description
k	Integer	2	yes	The number of disjoint communities the nodes will be divided into.
iterations	Integer	8	yes	The number of iterations the algorithm will run before returning the best solution among all the iterations.
vnsMaxNeighborhoodOrder	Integer	0 (VNS off)	yes	The maximum number of nodes VNS will swap when perturbing solutions.
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in the computation. Requires <code>concurrency = 1</code> .
relationshipWeightProperty	String	null	yes	If set, the values stored at the given property are used as relationship weights during the computation. If not set, the graph is considered unweighted.

Table 520. Results

Name	Type	Description
cutCost	Float	Sum of weights of all relationships connecting nodes from different communities.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the statistics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	Number of properties added to the in-memory graph.
configuration	Map	Configuration used for running the algorithm.

## Examples

In this section we will show examples of running the Approximate Maximum k-cut algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small Bitcoin transactions graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(alice:Person {name: 'Alice'}),
(bridget:Person {name: 'Bridget'}),
(charles:Person {name: 'Charles'}),
(doug:Person {name: 'Doug'}),
(eric:Person {name: 'Eric'}),
(fiona:Person {name: 'Fiona'}),
(george:Person {name: 'George'}),
(alice)-[:TRANSACTION {value: 81.0}]->(bridget),
(alice)-[:TRANSACTION {value: 7.0}]->(doug),
(bridget)-[:TRANSACTION {value: 1.0}]->(doug),
(bridget)-[:TRANSACTION {value: 1.0}]->(eric),
(bridget)-[:TRANSACTION {value: 1.0}]->(fiona),
(bridget)-[:TRANSACTION {value: 1.0}]->(george),
(charles)-[:TRANSACTION {value: 45.0}]->(bridget),
(charles)-[:TRANSACTION {value: 3.0}]->(eric),
(doug)-[:TRANSACTION {value: 3.0}]->(charles),
(doug)-[:TRANSACTION {value: 1.0}]->(bridget),
(eric)-[:TRANSACTION {value: 1.0}]->(bridget),
(fiona)-[:TRANSACTION {value: 3.0}]->(alice),
(fiona)-[:TRANSACTION {value: 1.0}]->(bridget),
(george)-[:TRANSACTION {value: 1.0}]->(bridget),
(george)-[:TRANSACTION {value: 4.0}]->(charles)
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `TRANSACTION` relationships.

The following statement will create a graph store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Person',
  {
    TRANSACTION: {
      properties: ['value']
    }
  }
)
```

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `mutate` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.alpha.maxkcut.mutate.estimate('myGraph', {mutateProperty: 'community'})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 521. Results



nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
7	15	488	488	"488 Bytes"

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the approximate maximum k-cut for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.alpha.maxkcut.mutate('myGraph', {mutateProperty: 'community'})
YIELD cutCost, nodePropertiesWritten
```

Table 522. Results

cutCost	nodePropertiesWritten
13.0	7

We can see that when relationship weight is not taken into account we derive a cut into two (since we didn't override the default `k = 2`) communities of cost `13.0`. The total cost is represented by the `cutCost` column here. This is the value we want to be as high as possible. Additionally, the graph 'myGraph' now has a node property `community` which stores the community to which each node belongs.

To inspect which community each node belongs to we can [stream node properties](#).

Stream node properties:

```
CALL gds.graph.streamNodeProperty('myGraph', 'community')
YIELD nodeId, propertyValue
RETURN gds.util.asNode(nodeId).name AS name, propertyValue AS community
```

Table 523. Results

name	community
"Alice"	0
"Bridget"	0
"Charles"	0
"Doug"	1
"Eric"	1
"Fiona"	1
"George"	1

Looking at our graph topology we can see that there are no relationships between the nodes of community 1, and two relationships between the nodes of community 0, namely *Alice* → *Bridget* and *Charles* → *Bridget*. However, since there are a total of eight relationships between *Bridget* and nodes of community 1, and our graph is unweighted assigning *Bridget* to community 1 would not yield a cut of a higher total weight. Thus, since the number of relationships connecting nodes of different communities greatly outnumber the number of relationships connecting nodes of the same community it seems like a good solution. In fact, this is the maximum 2-cut for this graph.



Because of the inherent randomness in the Approximate Maximum k-Cut algorithm (unless having `concurrency = 1` and fixed `randomSeed`), running it another time might yield a different solution. For our case here it would be equally plausible to get the inverse solution, i.e. when our community 0 nodes are mapped to community 1 instead, and vice versa. Note however, that for that solution the cut cost would remain the same.

### Mutate with relationship weights

In this example we will have a look at how adding relationship weight can affect our solution.

The following will run the algorithm in `mutate` mode, diving our nodes into two communities once again:

```
CALL gds.alpha.maxkcut.mutate(
  'myGraph',
  {
    relationshipWeightProperty: 'value',
    mutateProperty: 'weightedCommunity'
  }
)
YIELD cutCost, nodePropertiesWritten
```

Table 524. Results

cutCost	nodePropertiesWritten
146.0	7

Since the `value` properties on our `TRANSACTION` relationships were all at least `1.0` and several of a larger value it's not surprising that we obtain a cut with a larger cost in the weighted case.

Let us now `stream node properties` to once again inspect the node community distribution.

Stream node properties:

```
CALL gds.graph.streamNodeProperty('myGraph', 'weightedCommunity')
YIELD nodeId, propertyValue
RETURN gds.util.asNode(nodeId).name AS name, propertyValue AS weightedCommunity
```

Table 525. Results

name	weightedCommunity
"Alice"	0
"Bridget"	1

name	weightedCommunity
"Charles"	0
"Doug"	1
"Eric"	1
"Fiona"	1
"George"	1

Comparing this result with that of [unweighted case](#) we can see that **Bridget** has moved to another community but the output is otherwise the same. Indeed, this makes sense by looking at our graph. **Bridget** is connected to nodes of community 1 by eight relationships, but these relationships all have weight 1.0. And although **Bridget** is only connected to two community 0 nodes, these relationships are of weight 81.0 and 45.0. Moving **Bridget** back to community 0 would lower the total cut cost of  $81.0 + 45.0 - 8 * 1.0 = 118.0$ . Hence, it does make sense that **Bridget** is now in community 1. In fact, this is the maximum 2-cut in the weighted case.



Because of the inherent randomness in the Approximate Maximum k-Cut algorithm (unless having `concurrency = 1` and fixed `randomSeed`), running it another time might yield a different solution. For our case here it would be equally plausible to get the inverse solution, i.e. when our community 0 nodes are mapped to community 1 instead, and vice versa. Note however, that for that solution the cut cost would remain the same.

## Stream

In the `stream` execution mode, the algorithm returns the approximate maximum k-cut for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode using default configuration parameters:

```
CALL gds.alpha.maxkcut.stream('myGraph')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId
```

Table 526. Results

name	communityId
"Alice"	0
"Bridget"	0
"Charles"	0
"Doug"	1
"Eric"	1
"Fiona"	1

name	communityId
"George"	1

We can see that the result is what we expect, namely the same as in the [mutate unweighted](#) example.



Because of the inherent randomness in the Approximate Maximum k-Cut algorithm (unless having `concurrency = 1` and fixed `randomSeed`), running it another time might yield a different solution. For our case here it would be equally plausible to get the inverse solution, i.e. when our community `0` nodes are mapped to community `1` instead, and vice versa. Note however, that for that solution the cut cost would remain the same.

### 7.3.11. Conductance metric Alpha

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

#### Introduction

[Conductance](#) is a metric that allows you to evaluate the quality of a community detection. Relationships of nodes in a community `C` connect to nodes either within `C` or outside `C`. The conductance is the ratio between relationships that point outside `C` and the total number of relationships of `C`. The lower the conductance, the more "well-knit" a community is.

It was shown by Yang and Leskovec in the paper "[Defining and Evaluating Network Communities based on Ground-truth](#)" that conductance is a very good metric for evaluating actual communities of real world graphs.

The algorithm runs in time linear to the number of relationships in the graph.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

#### Syntax

This section covers the syntax used to execute the Conductance algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

## Example 2. Conductance syntax per mode

Run Conductance in stream mode on a named graph.

```
CALL gds.alpha.conductance.stream(
  graphName: String,
  configuration: Map
) YIELD
  community: Integer,
  conductance: Float
```

Table 527. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 528. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 529. Algorithm specific configuration

Name	Type	Default	Optional	Description
communityProperty	String	n/a	no	The node property that holds the community ID as an integer for each node. Note that only non-negative community IDs are considered valid and will have their conductance computed.

Table 530. Results

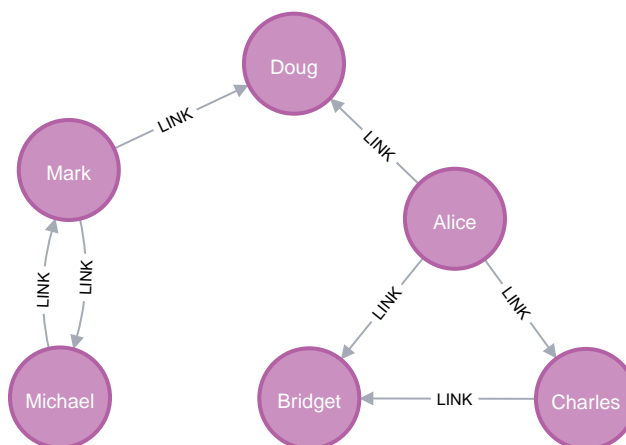
Name	Type	Description
community	Integer	Community ID.
conductance	Float	Conductance of the community.



Only non-negative community IDs are valid for identifying communities. Nodes with a negative community ID will only take part in the computation to the extent that they are connected to nodes in valid communities, and thus contribute to those valid communities' outward relationship counts.

## Examples

In this section we will show examples of running the Conductance algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(nAlice:User {name: 'Alice', seed: 42}),
(nBridget:User {name: 'Bridget', seed: 42}),
(nCharles:User {name: 'Charles', seed: 42}),
(nDoug:User {name: 'Doug'}),
(nMark:User {name: 'Mark'}),
(nMichael:User {name: 'Michael'}),

(nAlice)-[:LINK {weight: 1}]->(nBridget),
(nAlice)-[:LINK {weight: 1}]->(nCharles),
(nCharles)-[:LINK {weight: 1}]->(nBridget),

(nAlice)-[:LINK {weight: 5}]->(nDoug),

(nMark)-[:LINK {weight: 1}]->(nDoug),
(nMark)-[:LINK {weight: 1}]->(nMichael),
(nMichael)-[:LINK {weight: 1}]->(nMark);
```

This graph has two clusters of *Users*, that are closely connected. Between those clusters there is one single edge. The relationships that connect the nodes in each component have a property `weight` which determines the strength of the relationship.

We can now create the graph and store it in the graph catalog. We load the `LINK` relationships with orientation set to `UNDIRECTED` as this works best with the Louvain algorithm which we will use to create the communities that we evaluate using Conductance.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(  
  'myGraph',  
  'User',  
  {  
    LINK: {  
      orientation: 'UNDIRECTED'  
    }  
  },  
  {  
    nodeProperties: 'seed',  
    relationshipProperties: 'weight'  
  }  
)
```

We now run the [Louvain algorithm](#) to create a division of the nodes into communities that we can then evaluate.

The following will run the Louvain algorithm and store the results in `myGraph`:

```
CALL gds.louvain.mutate('myGraph', { mutateProperty: 'community', relationshipWeightProperty: 'weight' })  
YIELD communityCount
```

Table 531. Results

communityCount
3

Now our in-memory graph `myGraph` is populated with node properties under the key `community` that we can set as input for our evaluation using Conductance. The nodes are now assigned to communities in the following way:

Table 532. Community assignments

name	community
"Alice"	3
"Bridget"	2
"Charles"	2
"Doug"	3
"Mark"	5
"Michael"	5

Please see the [stream node properties](#) procedure for how to obtain such an assignment table.

For more information about Louvain, see its [algorithm page](#).

## Stream

Since we now have a community detection, we can evaluate how good it is under the conductance metric. Note that we in this case we use the feature of relationships being weighted by a relationship property.

The Conductance stream procedure returns the conductance for each community. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the stream mode in general, see [Stream](#).

The following will run the Conductance algorithm in `stream` mode:

```
CALL gds.alpha.conductance.stream('myGraph', { communityProperty: 'community', relationshipWeightProperty: 'weight' })
YIELD community, conductance
```

Table 533. Results

community	conductance
2	0.5
3	0.23076923076923078
5	0.2

We can see that the community of the weighted graph with the lowest conductance is community 5. This means that 5 is the community that is most "well-knit" in the sense that most of its relationship weights are internal to the community.

## 7.4. Similarity

Similarity algorithms compute the similarity of pairs of nodes using different vector-based metrics. The Neo4j GDS library includes the following similarity algorithms, grouped by quality tier:

- Production-quality
  - [Node Similarity](#)
- Beta
  - [K-Nearest Neighbors](#)
- Alpha
  - [Approximate Nearest Neighbors](#)
  - [Cosine Similarity](#)
  - [Euclidean Similarity](#)
  - [Jaccard Similarity](#)
  - [Overlap Similarity](#)
  - [Pearson Similarity](#)

### 7.4.1. Node Similarity

Supported algorithm traits:

[Directed](#)



Undirected

Homogeneous

Heterogeneous

Weighted

## Introduction

The Node Similarity algorithm compares a set of nodes based on the nodes they are connected to. Two nodes are considered similar if they share many of the same neighbors. Node Similarity computes pair-wise similarities based on the Jaccard metric, also known as the Jaccard Similarity Score.

Given two sets  $A$  and  $B$ , the Jaccard Similarity is computed using the following formula:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The input of this algorithm is a bipartite, connected graph containing two disjoint node sets. Each relationship starts from a node in the first node set and ends at a node in the second node set.

The Node Similarity algorithm compares each node that has outgoing relationships with each other such node. For every node  $n$ , we collect the outgoing neighborhood  $N(n)$  of that node, that is, all nodes  $m$  such that there is a relationship from  $n$  to  $m$ . For each pair  $n, m$ , the algorithm computes a similarity for that pair which is the Jaccard similarity of  $N(n)$  and  $N(m)$ .

The complexity of this comparison grows quadratically with the number of nodes to compare. The algorithm reduces the complexity by ignoring disconnected nodes.

In addition to computational complexity, the memory requirement for producing results also scales roughly quadratically. In order to bound memory usage, the algorithm requires an explicit limit on the number of results to compute per node. This is the 'topK' parameter. It can be set to any value, except 0.

The output of the algorithm are new relationships between pairs of the first node set. Similarity scores are expressed via relationship properties.

A related function for computing Jaccard similarity is described in [Jaccard Similarity](#).

For more information on this algorithm, see:

- [Structural equivalence \(Wikipedia\)](#)
- [The Jaccard index \(Wikipedia\)](#).

- [Bipartite graphs \(Wikipedia\)](#)



Running this algorithm requires sufficient available memory. Before running this algorithm, we recommend that you read [Memory Estimation](#).

## Syntax

This section covers the syntax used to execute the Node Similarity algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Node Similarity in stream mode on a named graph.

```
CALL gds.nodeSimilarity.stream(
  graphName: String,
  configuration: Map
) YIELD
  node1: Integer,
  node2: Integer,
  similarity: Float
```

Table 534. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 535. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 536. Algorithm specific configuration

Name	Type	Default	Optional	Description
similarityCutoff	Float	1E-42	yes	Lower limit for the similarity score to be present in the result. Values must be between 0 and 1.
degreeCutoff	Integer	1	yes	Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.
topK	Integer	10	yes	Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1.
bottomK	Integer	10	yes	Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1.
topN	Integer	0	yes	Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit.
bottomN	Integer	0	yes	Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 537. Results

Name	Type	Description
node1	Integer	Node ID of the first node.
node2	Integer	Node ID of the second node.
similarity	Float	Similarity score for the two nodes.

Run Node Similarity in stats mode on a named graph.

```
CALL gds.nodeSimilarity.stats(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  nodesCompared: Integer,
  similarityPairs: Integer,
  similarityDistribution: Map,
  configuration: Map
```

Table 538. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 539. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 540. Algorithm specific configuration

Name	Type	Default	Optional	Description
similarityCutoff	Float	1E-42	yes	Lower limit for the similarity score to be present in the result. Values must be between 0 and 1.
degreeCutoff	Integer	1	yes	Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.
topK	Integer	10	yes	Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1.
bottomK	Integer	10	yes	Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1.
topN	Integer	0	yes	Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit.
bottomN	Integer	0	yes	Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit.

Name	Type	Default	Optional	Description
<a href="#">relationshipWeightProperty</a>	String	<code>null</code>	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 541. Results

Name	Type	Description
<code>createMillis</code>	Integer	Milliseconds for loading data.
<code>computeMillis</code>	Integer	Milliseconds for running the algorithm.
<code>nodesCompared</code>	Integer	The number of nodes compared.
<code>postProcessingMillis</code>	Integer	Milliseconds for computing component count and distribution statistics.
<code>similarityPairs</code>	Integer	The number of pairs of similar nodes computed.
<code>similarityDistribution</code>	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of the computed similarity results.
<code>configuration</code>	Map	The configuration used for running the algorithm.

Run Node Similarity in mutate mode on a graph stored in the catalog.

```
CALL gds.nodeSimilarity.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  createMillis: Integer,  
  computeMillis: Integer,  
  mutateMillis: Integer,  
  postProcessingMillis: Integer,  
  relationshipsWritten: Integer,  
  nodesCompared: Integer,  
  similarityDistribution: Map,  
  configuration: Map
```

Table 542. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 543. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 544. Algorithm specific configuration

Name	Type	Default	Optional	Description
similarityCutoff	Float	1E-42	yes	Lower limit for the similarity score to be present in the result. Values must be between 0 and 1.
degreeCutoff	Integer	1	yes	Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.
topK	Integer	10	yes	Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1.
bottomK	Integer	10	yes	Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1.
topN	Integer	0	yes	Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit.



Name	Type	Default	Optional	Description
bottomN	Integer	0	yes	Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit.
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 545. Results

Name	Type	Description
nodesCompared	Integer	The number of nodes compared.
relationshipsWritten	Integer	The number of relationships created.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for computing percentiles.
similarityDistribution	Map	Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results.
configuration	Map	The configuration used for running the algorithm.

Run Node Similarity in write mode on a graph stored in the catalog.

```
CALL gds.nodeSimilarity.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodesCompared: Integer,
  relationshipsWritten: Integer,
  similarityDistribution: Map,
  configuration: Map
```

Table 546. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 547. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 548. Algorithm specific configuration

Name	Type	Default	Optional	Description
similarityCutoff	Float	1E-42	yes	Lower limit for the similarity score to be present in the result. Values must be between 0 and 1.
degreeCutoff	Integer	1	yes	Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.
topK	Integer	10	yes	Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1.
bottomK	Integer	10	yes	Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1.

Name	Type	Default	Optional	Description
topN	Integer	0	yes	Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit.
bottomN	Integer	0	yes	Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit.
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 549. Results

Name	Type	Description
nodesCompared	Integer	The number of nodes compared.
relationshipsWritten	Integer	The number of relationships created.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
postProcessingMillis	Integer	Milliseconds for computing percentiles.
similarityDistribution	Map	Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Node Similarity in write mode on an anonymous graph.

```
CALL gds.nodeSimilarity.write(
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodesCompared: Integer,
  relationshipsWritten: Integer,
  similarityDistribution: Map,
  configuration: Map
```

Table 550. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 551. Algorithm specific configuration

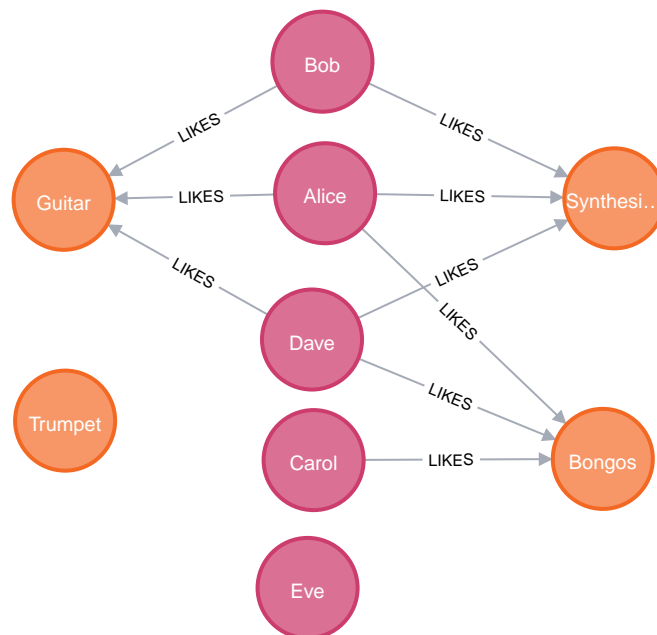
Name	Type	Default	Optional	Description
similarityCutoff	Float	1E-42	yes	Lower limit for the similarity score to be present in the result. Values must be between 0 and 1.
degreeCutoff	Integer	1	yes	Lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.
topK	Integer	10	yes	Limit on the number of scores per node. The K largest results are returned. This value cannot be lower than 1.
bottomK	Integer	10	yes	Limit on the number of scores per node. The K smallest results are returned. This value cannot be lower than 1.

Name	Type	Default	Optional	Description
topN	Integer	0	yes	Global limit on the number of scores computed. The N largest total results are returned. This value cannot be negative, a value of 0 means no global limit.
bottomN	Integer	0	yes	Global limit on the number of scores computed. The N smallest total results are returned. This value cannot be negative, a value of 0 means no global limit.
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the Node Similarity algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small knowledge graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
  (alice:Person {name: 'Alice'}),
  (bob:Person {name: 'Bob'}),
  (carol:Person {name: 'Carol'}),
  (dave:Person {name: 'Dave'}),
  (eve:Person {name: 'Eve'}),
  (guitar:Instrument {name: 'Guitar'}),
  (synth:Instrument {name: 'Synthesizer'}),
  (bongos:Instrument {name: 'Bongos'}),
  (trumpet:Instrument {name: 'Trumpet'}),

  (alice)-[:LIKES]->(guitar),
  (alice)-[:LIKES]->(synth),
  (alice)-[:LIKES {strength: 0.5}]->(bongos),
  (bob)-[:LIKES]->(guitar),
  (bob)-[:LIKES]->(synth),
  (carol)-[:LIKES]->(bongos),
  (dave)-[:LIKES]->(guitar),
  (dave)-[:LIKES]->(synth),
  (dave)-[:LIKES]->(bongos);
```

This bipartite graph has two node sets, Person nodes and Instrument nodes. The two node sets are connected via LIKES relationships. Each relationship starts at a Person node and ends at an Instrument node.

In the example, we want to use the Node Similarity algorithm to compare people based on the instruments they like.

The Node Similarity algorithm will only compute similarity for nodes that have a degree of at least 1. In the example graph, the Eve node will not be compared to other Person nodes.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  ['Person', 'Instrument'],
  {
    LIKES: {
      type: 'LIKES',
      properties: {
        strength: {
          property: 'strength',
          defaultValue: 1.0
        }
      }
    }
  }
);
```

In the following examples we will demonstrate using the Node Similarity algorithm on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful

to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.nodeSimilarity.write.estimate('myGraph', {
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 552. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
9	9	2592	2808	"[2592 Bytes ... 2808 Bytes]"

## Stream

In the `stream` execution mode, the algorithm returns the similarity score for each relationship. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.nodeSimilarity.stream('myGraph')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

Table 553. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Dave"	"Alice"	1.0
"Alice"	"Bob"	0.6666666666666666
"Bob"	"Alice"	0.6666666666666666
"Bob"	"Dave"	0.6666666666666666
"Dave"	"Bob"	0.6666666666666666
"Alice"	"Carol"	0.3333333333333333
"Carol"	"Alice"	0.3333333333333333
"Carol"	"Dave"	0.3333333333333333

Person1	Person2	similarity
"Dave"	"Carol"	0.3333333333333333

We use default values for the procedure configuration parameter. TopK is set to 10, topN is set to 0. Because of that the result set contains the top 10 similarity scores for each node.



If we would like to instead compare the Instruments to each other, we would then project the **LIKES** relationship type using **REVERSE** orientation. This would return similarities for pairs of Instruments and not compute any similarities between Persons.

## Stats

In the **stats** execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the **computeMillis** return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the **stats** mode in general, see [Stats](#).

The following will run the algorithm and returns the result in form of statistical and measurement values

```
CALL gds.nodeSimilarity.stats('myGraph')
YIELD nodesCompared, similarityPairs
```

Table 554. Results

nodesCompared	similarityPairs
4	10

## Mutate

The **mutate** execution mode extends the **stats** mode with an important side effect: updating the named graph with a new relationship property containing the similarity score for that relationship. The name of the new property is specified using the mandatory configuration parameter **mutateProperty**. The result is a single summary row, similar to **stats**, but with some additional metrics. The **mutate** mode is especially useful when multiple algorithms are used in conjunction.

For more details on the **mutate** mode in general, see [Mutate](#).

The following will run the algorithm, and write back results to the in-memory graph:

```
CALL gds.nodeSimilarity.mutate('myGraph', {
  mutateRelationshipType: 'SIMILAR',
  mutateProperty: 'score'
})
YIELD nodesCompared, relationshipsWritten
```

Table 555. Results



nodesCompared	relationshipsWritten
4	10

As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

## Write

The `write` execution mode extends the `stats` mode with an important side effect: for each pair of nodes we create a relationship with the Jaccard similarity score as a property to the Neo4j database. The type of the new relationship is specified using the mandatory configuration parameter `writeRelationshipType`. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm, and write back results:

```
CALL gds.nodeSimilarity.write('myGraph', {
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score'
})
YIELD nodesCompared, relationshipsWritten
```

Table 556. Results

nodesCompared	relationshipsWritten
4	10

As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

## Limit results

There are four limits that can be applied to the similarity results. Top limits the result to the highest similarity scores. Bottom limits the result to the lowest similarity scores. Both top and bottom limits can apply to the result as a whole ("N"), or to the result per node ("K").



There must always be a "K" limit, either `bottomK` or `topK`, which is a positive number. The default value for `topK` and `bottomK` is 10.

Table 557. Result limits

	total results	results per node
highest score	topN	topK
lowest score	bottomN	bottomK

## topK and bottomK

TopK and bottomK are limits on the number of scores computed per node. For topK, the K largest similarity scores per node are returned. For bottomK, the K smallest similarity scores per node are returned. TopK and bottomK cannot be 0, used in conjunction, and the default value is 10. If neither is specified, topK is used.

The following will run the algorithm, and stream the top 1 result per node:

```
CALL gds.nodeSimilarity.stream('myGraph', { topK: 1 })
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY Person1
```

Table 558. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Bob"	"Alice"	0.6666666666666666
"Carol"	"Alice"	0.3333333333333333
"Dave"	"Alice"	1.0

The following will run the algorithm, and stream the bottom 1 result per node:

```
CALL gds.nodeSimilarity.stream('myGraph', { bottomK: 1 })
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY Person1
```

Table 559. Results

Person1	Person2	similarity
"Alice"	"Carol"	0.3333333333333333
"Bob"	"Alice"	0.6666666666666666
"Carol"	"Alice"	0.3333333333333333
"Dave"	"Carol"	0.3333333333333333

## topN and bottomN

TopN and bottomN limit the number of similarity scores across all nodes. This is a limit on the total result set, in addition to the topK or bottomK limit on the results per node. For topN, the N largest similarity scores are returned. For bottomN, the N smallest similarity scores are returned. A value of 0 means no global limit is imposed and all results from topK or bottomK are returned.

The following will run the algorithm, and stream the 3 highest out of the top 1 results per node:

```
CALL gds.nodeSimilarity.stream('myGraph', { topK: 1, topN: 3 })
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY similarity DESC, Person1, Person2
```

Table 560. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Dave"	"Alice"	1.0
"Bob"	"Alice"	0.6666666666666666

### Degree cutoff and similarity cutoff

Degree cutoff is a lower limit on the node degree for a node to be considered in the comparisons. This value can not be lower than 1.

The following will ignore nodes with less than 3 LIKES relationships:

```
CALL gds.nodeSimilarity.stream('myGraph', { degreeCutoff: 3 })
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY Person1
```

Table 561. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Dave"	"Alice"	1.0

Similarity cutoff is a lower limit for the similarity score to be present in the result. The default value is very small (1E-42) to exclude results with a similarity score of 0.



Setting similarity cutoff to 0 may yield a very large result set, increased runtime and memory consumption.

The following will ignore node pairs with a similarity score less than 0.5:

```
CALL gds.nodeSimilarity.stream('myGraph', { similarityCutoff: 0.5 })
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY Person1
```

Table 562. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0

Person1	Person2	similarity
"Alice"	"Bob"	0.6666666666666666
"Bob"	"Dave"	0.6666666666666666
"Bob"	"Alice"	0.6666666666666666
"Dave"	"Alice"	1.0
"Dave"	"Bob"	0.6666666666666666

## Weighted Jaccard Similarity

Relationship properties can be used to modify the similarity induced by certain relationships. For example a relationship value of 2 is equal to counting that relationship twice while computing the jaccard similarity.



Weighted jaccard similarity is only defined for values greater or equal to 0.

The following query will respect relationship properties in the similarity computation:

```
CALL gds.nodeSimilarity.stream('myGraph', { relationshipWeightProperty: 'strength', similarityCutoff: 0.5
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY Person1
```

Table 563. Results

Person1	Person2	similarity
"Alice"	"Dave"	0.8333333333333334
"Alice"	"Bob"	0.8
"Bob"	"Alice"	0.8
"Bob"	"Dave"	0.6666666666666666
"Dave"	"Alice"	0.8333333333333334
"Dave"	"Bob"	0.6666666666666666

It can be seen that the similarity between Alice and Dave decreased compared to the non-weighted version of this algorithm. This is the case as the strength of the relationship between Alice and Bongos is reduced and both persons now only share 2.5 out of 3 possible instruments. Analogous the similarity between Alice and Bob increased as the missing liked instrument has a lower impact on the similarity score.

## 7.4.2. K-Nearest Neighbors Beta

### Introduction

The K-Nearest Neighbors algorithm computes a distance value for all node pairs in the graph and creates new relationships between each node and its k nearest neighbors. The distance is calculated based on

node properties.

The input of this algorithm is a monopartite graph. The graph does not need to be connected, in fact, existing relationships between nodes will be ignored. New relationships are created between each node and its  $k$  nearest neighbors.

The K-Nearest Neighbors algorithm compares a given property of each node. The  $k$  nodes where this property is most similar are the  $k$ -nearest neighbors.

The initial set of neighbors is picked at random and verified and refined in multiple iterations. The number of iterations is limited by the configuration parameter `maxIterations`. The algorithm may stop earlier if the neighbor lists only change by a small amount, which can be controlled by the configuration parameter `deltaThreshold`.

The particular implementation is based on [Efficient k-nearest neighbor graph construction for generic similarity measures](#) by Wei Dong et al. Instead of comparing every node with every other node, the algorithm selects possible neighbors based on the assumption, that the neighbors-of-neighbors of a node are most likely already the nearest one. The algorithm scales quasi-linear with respect to the node count, instead of being quadratic.

Furthermore, the algorithm only compares a sample of all possible neighbors on each iteration, assuming that eventually all possible neighbors will be seen. This can be controlled with the configuration parameter `sampleRate`:

- A valid sample rate must be in between 0 (exclusive) and 1 (inclusive).
- The default value is `0.5`.
- The parameter is used to control the trade-off between accuracy and runtime-performance.
- A higher sample rate will increase the accuracy of the result.
  - The algorithm will also require more memory and will take longer to compute.
- A lower sample rate will increase the runtime-performance.
  - Some potential nodes may be missed in the comparison and may not be included in the result.

The output of the algorithm are new relationships between nodes and their  $k$ -nearest neighbors. Similarity scores are expressed via relationship properties.

For more information on this algorithm, see:

- [Efficient k-nearest neighbor graph construction for generic similarity measures](#)
- [Nearest neighbor graph \(Wikipedia\)](#)



Running this algorithm requires sufficient available memory. Before running this algorithm, we recommend that you read [Memory Estimation](#).

## Similarity measures

The similarity measure used in the KNN algorithm depends on the type of the configured node property.

KNN supports both scalar numeric values as well as lists of numbers.

#### Scalar numeric property

When the property is a scalar number, the similarity is computed as one divided by one plus the absolute difference between the values:

$$\frac{1}{1 + |p_s - p_t|}$$

#### List of integers

When the property is a list of integers, the similarity is computed as one divided by one plus the number of unequal numbers in the list:

$$\frac{1}{1 + \Delta(p_s, p_t)}$$

#### List of floating-point numbers

When the property is a list of floating-point numbers, the similarity is computed using the cosine similarity metric. See the [Cosine Similarity](#) algorithm for more details. If the cosine similarity is negative, we clip the value to 0, i.e.,  $\max(\text{cosine}(a, b), 0)$ .

## Syntax

This section covers the syntax used to execute the K-Nearest Neighbors algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run K-Nearest Neighbors in stream mode on a named graph.

```
CALL gds.beta.knn.stream(
  graphName: String,
  configuration: Map
) YIELD
  node1: Integer,
  node2: Integer,
  similarity: Float
```

Table 564. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 565. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 566. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeWeightProperty	String	n/a	no	The name of a node property that contains node weights which will be used for similarity computation.
topK	Integer	10	yes	The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1.
sampleRate	Float	0.5	yes	Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive).
deltaThreshold	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).
maxIterations	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.
randomJoins	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.



Name	Type	Default	Optional	Description
randomSeed	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <code>concurrency</code> must be set to 1 when setting this parameter.

Table 567. Results

Name	Type	Description
node1	Integer	Node ID of the first node.
node2	Integer	Node ID of the second node.
similarity	Float	Similarity score for the two nodes.

Run K-Nearest Neighbors in stats mode on a named graph.

```
CALL gds.beta.knn.stats(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  nodesCompared: Integer,
  similarityPairs: Integer,
  similarityDistribution: Map,
  configuration: Map
```

Table 568. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 569. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 570. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeWeightProperty	String	n/a	no	The name of a node property that contains node weights which will be used for similarity computation.
topK	Integer	10	yes	The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1.
sampleRate	Float	0.5	yes	Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive).
deltaThreshold	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).
maxIterations	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.

Name	Type	Default	Optional	Description
randomJoins	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.
randomSeed	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <code>concurrency</code> must be set to 1 when setting this parameter.

Table 571. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing similarity value distribution statistics.
nodesCompared	Integer	The number of nodes compared.
similarityPairs	Integer	The number of pairs of similar nodes computed.
similarityDistribution	Map	Map containing min, max, mean as well as p50, p75, p90, p95, p99 and p999 percentile values of the computed similarity results.
configuration	Map	The configuration used for running the algorithm.

Run K-Nearest Neighbors in mutate mode on a graph stored in the catalog.

```
CALL gds.beta.knn.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  createMillis: Integer,  
  computeMillis: Integer,  
  mutateMillis: Integer,  
  postProcessingMillis: Integer,  
  relationshipsWritten: Integer,  
  nodesCompared: Integer,  
  similarityDistribution: Map,  
  configuration: Map
```

Table 572. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 573. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 574. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeWeightProperty	String	n/a	no	The name of a node property that contains node weights which will be used for similarity computation.
topK	Integer	10	yes	The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1.
sampleRate	Float	0.5	yes	Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive).
deltaThreshold	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).
maxIterations	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.

Name	Type	Default	Optional	Description
randomJoins	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.
randomSeed	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <code>concurrency</code> must be set to 1 when setting this parameter.

Table 575. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for computing similarity value distribution statistics.
nodesCompared	Integer	The number of nodes compared.
relationshipsWritten	Integer	The number of relationships created.
similarityDistribution	Map	Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results.
configuration	Map	The configuration used for running the algorithm.

Run K-Nearest Neighbors in write mode on a graph stored in the catalog.

```
CALL gds.beta.knn.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodesCompared: Integer,
  relationshipsWritten: Integer,
  similarityDistribution: Map,
  configuration: Map
```

Table 576. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 577. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 578. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeWeightProperty	String	n/a	no	The name of a node property that contains node weights which will be used for similarity computation.
topK	Integer	10	yes	The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1.
sampleRate	Float	0.5	yes	Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive).
deltaThreshold	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).

Name	Type	Default	Optional	Description
<code>maxIterations</code>	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.
<code>randomJoins</code>	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.
<code>randomSeed</code>	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <code>concurrency</code> must be set to 1 when setting this parameter.

Table 579. Results

Name	Type	Description
<code>createMillis</code>	Integer	Milliseconds for loading data.
<code>computeMillis</code>	Integer	Milliseconds for running the algorithm.
<code>writeMillis</code>	Integer	Milliseconds for writing result data back to Neo4j.
<code>postProcessingMillis</code>	Integer	Milliseconds for computing similarity value distribution statistics.
<code>nodesCompared</code>	Integer	The number of nodes compared.
<code>relationshipsWritten</code>	Integer	The number of relationships created.
<code>similarityDistribution</code>	Map	Map containing min, max, mean, stdDev and p1, p5, p10, p25, p75, p90, p95, p99, p100 percentile values of the computed similarity results.
<code>configuration</code>	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run K-Nearest Neighbors in write mode on an anonymous graph.

```
CALL gds.beta.knn.write(
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  postProcessingMillis: Integer,
  nodesCompared: Integer,
  relationshipsWritten: Integer,
  similarityDistribution: Map,
  configuration: Map
```

Table 580. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.



The KNN algorithm does not read any relationships, but the values for **relationshipProjection** or **relationshipQuery** are still being used and respected for the graph loading.

Table 581. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeWeightProperty	String	n/a	no	The name of a node property that contains node weights which will be used for similarity computation.



Name	Type	Default	Optional	Description
topK	Integer	10	yes	The number of neighbors to find for each node. The K-nearest neighbors are returned. This value cannot be lower than 1.
sampleRate	Float	0.5	yes	Sample rate to limit the number of comparisons per node. Value must be between 0 (exclusive) and 1 (inclusive).
deltaThreshold	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).
maxIterations	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.
randomJoins	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.
randomSeed	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <code>concurrency</code> must be set to 1 when setting this parameter.

The results are the same as running write mode on a named graph, see [write mode syntax above](#).



To get a deterministic result when running the algorithm:

- the `concurrency` parameter must be set to one
- the `randomSeed` must be explicitly set to something other than -1.

## Examples

Consider the graph created by the following Cypher statement:

```
CREATE (alice:Person {name: 'Alice', age: 24})
CREATE (bob:Person {name: 'Bob', age: 73})
CREATE (carol:Person {name: 'Carol', age: 24})
CREATE (dave:Person {name: 'Dave', age: 48})
CREATE (eve:Person {name: 'Eve', age: 67});
```

In the example, we want to use the K-Nearest Neighbors algorithm to compare people based on their age.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(  
  'myGraph',  
  {  
    Person: {  
      label: 'Person',  
      properties: 'age'  
    }  
  },  
  '*'  
);
```

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.beta.knn.write.estimate('myGraph', {  
  nodeWeightProperty: 'age',  
  writeRelationshipType: 'SIMILAR',  
  writeProperty: 'score',  
  topK: 1  
})  
YIELD nodeCount, bytesMin, bytesMax, requiredMemory
```

Table 582. Results

nodeCount	bytesMin	bytesMax	requiredMemory
5	1944	3000	"[1944 Bytes ... 3000 Bytes]"

## Stream

In the `stream` execution mode, the algorithm returns the similarity score for each relationship. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.beta.knn.stream('myGraph', {
  topK: 1,
  nodeWeightProperty: 'age',
  // The following parameters are set to produce a deterministic result
  randomSeed: 1337,
  concurrency: 1,
  sampleRate: 1.0,
  deltaThreshold: 0.0
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1, gds.util.asNode(node2).name AS Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

Table 583. Results

Person1	Person2	similarity
"Alice"	"Carol"	1.0
"Carol"	"Alice"	1.0
"Bob"	"Eve"	0.14285714285714285
"Eve"	"Bob"	0.14285714285714285
"Dave"	"Eve"	0.05

We use default values for the procedure configuration parameter for most parameters. The `randomSeed` and `concurrency` is set to produce the same result on every invocation. The `topK` parameter is set to 1 to only return the single nearest neighbor for every node.

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm and return the result in form of statistical and measurement values:

```
CALL gds.beta.knn.stats('myGraph', {topK: 1, concurrency: 1, randomSeed: 42, nodeWeightProperty: 'age'})
YIELD nodesCompared, similarityPairs
```

Table 584. Results

nodesCompared	similarityPairs
5	5

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new relationship property containing the similarity score for that relationship. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm, and write back results to the in-memory graph:

```
CALL gds.beta.knn.mutate('myGraph', {
  mutateRelationshipType: 'SIMILAR',
  mutateProperty: 'score',
  topK: 1,
  randomSeed: 42,
  concurrency: 1,
  nodeWeightProperty: 'age'
})
YIELD nodesCompared, relationshipsWritten
```

Table 585. Results

nodesCompared	relationshipsWritten
5	5

As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

## Write

The `write` execution mode extends the `stats` mode with an important side effect: for each pair of nodes we create a relationship with the similarity score as a property to the Neo4j database. The type of the new relationship is specified using the mandatory configuration parameter `writeRelationshipType`. Each new relationship stores the similarity score between the two nodes it represents. The relationship property key is set using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm, and write back results:

```
CALL gds.beta.knn.write('myGraph', {
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score',
  topK: 1,
  randomSeed: 42,
  concurrency: 1,
  nodeWeightProperty: 'age'
})
YIELD nodesCompared, relationshipsWritten
```

Table 586. Results

nodesCompared	relationshipsWritten
5	5

As we can see from the results, the number of created relationships is equal to the number of rows in the streaming example.

### 7.4.3. Jaccard Similarity Alpha

Jaccard Similarity (coefficient), a term coined by [Paul Jaccard](#), measures similarities between sets. It is defined as the size of the intersection divided by the size of the union of two sets. This notion has been generalized for multisets, where duplicate elements are counted as weights.

The GDS Jaccard Similarity function is defined for lists, which are interpreted as multisets.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

A related procedure for computing Jaccard similarity is described in [Node Similarity](#).

#### History and explanation

Given two sets **A** and **B**, the Jaccard Similarity is computed using the following formula:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The library contains functions to calculate similarity between sets of data. The Jaccard Similarity function is best used when calculating the similarity between small numbers of sets.

#### Use-cases - when to use the Jaccard Similarity algorithm

We can use the Jaccard Similarity algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, you can use the Jaccard Similarity algorithm to show the products that were purchased by similar customers, in terms of previous products purchased.

#### Jaccard Similarity algorithm function sample

The Jaccard Similarity function computes the similarity of two lists of numbers.

We can use it to compute the similarity of two hardcoded lists.

The following will return the Jaccard Similarity of two lists of numbers:

```
RETURN gds.alpha.similarity.jaccard([1,2,3], [1,2,4,5]) AS similarity
```

Table 587. Results

similarity
0.4

These two lists of numbers have a Jaccard Similarity of 0.4. We can see how this result is derived by breaking down the formula:

```
J(A,B) = A ∩ B / A + B - A ∩ B|
J(A,B) = 2 / 3 + 4 - 2
        = 2 / 5
        = 0.4
```

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

The following will create a sample graph:

```
CREATE
  (french:Cuisine {name:'French'}),
  (italian:Cuisine {name:'Italian'}),
  (indian:Cuisine {name:'Indian'}),
  (lebanese:Cuisine {name:'Lebanese'}),
  (portuguese:Cuisine {name:'Portuguese'}),

  (zhen:Person {name: 'Zhen'}),
  (praveena:Person {name: 'Praveena'}),
  (michael:Person {name: 'Michael'}),
  (arya:Person {name: 'Arya'}),
  (karin:Person {name: 'Karin'}),

  (praveena)-[:LIKES]->(indian),
  (praveena)-[:LIKES]->(portuguese),

  (zhen)-[:LIKES]->(french),
  (zhen)-[:LIKES]->(indian),

  (michael)-[:LIKES]->(french),
  (michael)-[:LIKES]->(italian),
  (michael)-[:LIKES]->(indian),

  (arya)-[:LIKES]->(lebanese),
  (arya)-[:LIKES]->(italian),
  (arya)-[:LIKES]->(portuguese),

  (karin)-[:LIKES]->(lebanese),
  (karin)-[:LIKES]->(italian)
```

The following will return the Jaccard Similarity of Karin and Arya:

```
MATCH (p1:Person {name: 'Karin'})-[:LIKES]->(cuisine1)
WITH p1, collect(id(cuisine1)) AS p1Cuisine
MATCH (p2:Person {name: "Arya"})-[:LIKES]->(cuisine2)
WITH p1, p1Cuisine, p2, collect(id(cuisine2)) AS p2Cuisine
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.jaccard(p1Cuisine, p2Cuisine) AS similarity
```

Table 588. Results

from	to	similarity
"Karin"	"Arya"	0.6666666666666666

The following will return the Jaccard Similarity of Karin and the other people that have a cuisine in common:

```
MATCH (p1:Person {name: 'Karin'})-[:LIKES]->(cuisine1)
WITH p1, collect(id(cuisine1)) AS p1Cuisine
MATCH (p2:Person)-[:LIKES]->(cuisine2) WHERE p1 <> p2
WITH p1, p1Cuisine, p2, collect(id(cuisine2)) AS p2Cuisine
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.jaccard(p1Cuisine, p2Cuisine) AS similarity
ORDER BY to, similarity DESC
```

Table 589. Results

from	to	similarity
"Karin"	"Arya"	0.6666666666666666
"Karin"	"Michael"	0.25
"Karin"	"Praveena"	0.0
"Karin"	"Zhen"	0.0

#### 7.4.4. Cosine Similarity Alpha

**Cosine similarity** is the cosine of the angle between two  $n$ -dimensional vectors in an  $n$ -dimensional space. It is the dot product of the two vectors divided by the product of the two vectors' lengths (or magnitudes).

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

#### History and explanation

Cosine similarity is computed using the following formula:

$$\text{similarity}(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n A_i^2} \times \sqrt{\sum_{i=1}^n B_i^2}}$$

Values range between -1 and 1, where -1 is perfectly dissimilar and 1 is perfectly similar.

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation and are therefore more appropriate for computing similarities on bigger datasets.

## Use-cases - when to use the Cosine Similarity algorithm

We can use the Cosine Similarity algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, to get movie recommendations based on the preferences of users who have given similar ratings to other movies that you've seen.

## Syntax

The following will create an anonymous graph to run the algorithm on and write back results:

```
CALL gds.alpha.similarity.cosine.write(configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75,
p90, p95, p99, p999, p100
```

Table 590. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 591. Configuration

Name	Type	Default	Optional	Description
data	List of String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for <code>'writeConcurrency'</code> .
writeConcurrency	Integer	value of <code>'concurrency'</code>	yes	The number of concurrent threads used for writing the result.
graph	String	dense	yes	The graph type ('dense' or 'cypher').
writeBatchSize	Integer	10000	yes	The batch size to use when storing results.
writeRelationshipType	String	SIMILAR	yes	The relationship type to use when storing results.



Name	Type	Default	Optional	Description
writeProperty	String	score	yes	The property to use when storing results.
sourceIds	List of Integer	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of Integer	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 592. Results

Name	Type	Description
nodes	Integer	The number of nodes passed in.
similarityPairs	Integer	The number of pairs of similar nodes computed.
writeRelationshipType	String	The relationship type used when storing results.
writeProperty	String	The property used when storing results.
min	Float	The minimum similarity score computed.
max	Float	The maximum similarity score computed.
mean	Float	The mean of similarities scores computed.
stdDev	Float	The standard deviation of similarities scores computed.
p25	Float	The 25 percentile of similarities scores computed.
p50	Float	The 50 percentile of similarities scores computed.
p75	Float	The 75 percentile of similarities scores computed.
p90	Float	The 90 percentile of similarities scores computed.
p95	Float	The 95 percentile of similarities scores computed.
p99	Float	The 99 percentile of similarities scores computed.
p999	Float	The 99.9 percentile of similarities scores computed.
p100	Float	The 100 percentile of similarities scores computed.

The following will create an anonymous graph to run the algorithm on and stream results:

```
CALL gds.alpha.similarity.cosine.stream(configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```

Table 593. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 594. Configuration

Name	Type	Default	Optional	Description
data	List of String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	null	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
graph	String	dense	yes	The graph type ('dense' or 'cypher').
sourceIds	List of Integer	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of Integer	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 595. Results

Name	Type	Description
item1	Integer	The ID of one node in the similarity pair.
item2	Integer	The ID of other node in the similarity pair.
count1	Integer	The size of the <code>targets</code> list of one node.
count2	Integer	The size of the <code>targets</code> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <code>targets</code> lists.
similarity	Integer	The cosine similarity of the two nodes.

## Cosine Similarity algorithm function sample

The Cosine Similarity function computes the similarity of two lists of numbers.



Cosine Similarity is only calculated over non-NULL dimensions. When calling the function, we should provide lists that contain the overlapping items.

We can use it to compute the similarity of two hardcoded lists.

The following will return the cosine similarity of two lists of numbers:

```
RETURN gds.alpha.similarity.cosine([3,8,7,5,2,9], [10,8,6,6,4,5]) AS similarity
```

Table 596. Results

similarity
0.8638935626791597

These two lists of numbers have a Cosine similarity of 0.863. We can see how this result is derived by breaking down the formula:

$$\left( \text{similarity}(A,B) = \frac{3 \cdot 10 + 8 \cdot 8 + 7 \cdot 6 + 5 \cdot 6 + 2 \cdot 4 + 9 \cdot 5}{\sqrt{3^2 + 8^2 + 7^2 + 5^2 + 2^2 + 9^2} \times \sqrt{10^2 + 8^2 + 6^2 + 6^2 + 4^2 + 5^2}} = \frac{219}{15.2315 \times 16.6433} = 0.8639 \right)$$

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

The following will create a sample graph:

```
CREATE (french:Cuisine {name:'French'})
CREATE (italian:Cuisine {name:'Italian'})
CREATE (indian:Cuisine {name:'Indian'})
CREATE (lebanese:Cuisine {name:'Lebanese'})
CREATE (portuguese:Cuisine {name:'Portuguese'})
CREATE (british:Cuisine {name:'British'})
CREATE (mauritian:Cuisine {name:'Mauritian'})

CREATE (zhen:Person {name:"Zhen"})
CREATE (praveena:Person {name:"Praveena"})
CREATE (michael:Person {name:"Michael"})
CREATE (arya:Person {name:"Arya"})
CREATE (karin:Person {name:"Karin"})

CREATE (praveena)-[:LIKES {score: 9}]->(indian)
CREATE (praveena)-[:LIKES {score: 7}]->(portuguese)
CREATE (praveena)-[:LIKES {score: 8}]->(british)
CREATE (praveena)-[:LIKES {score: 1}]->(mauritian)

CREATE (zhen)-[:LIKES {score: 10}]->(french)
CREATE (zhen)-[:LIKES {score: 6}]->(indian)
CREATE (zhen)-[:LIKES {score: 2}]->(british)

CREATE (michael)-[:LIKES {score: 8}]->(french)
CREATE (michael)-[:LIKES {score: 7}]->(italian)
CREATE (michael)-[:LIKES {score: 9}]->(indian)
CREATE (michael)-[:LIKES {score: 3}]->(portuguese)

CREATE (arya)-[:LIKES {score: 10}]->(lebanese)
CREATE (arya)-[:LIKES {score: 10}]->(italian)
CREATE (arya)-[:LIKES {score: 7}]->(portuguese)
CREATE (arya)-[:LIKES {score: 9}]->(mauritian)

CREATE (karin)-[:LIKES {score: 9}]->(lebanese)
CREATE (karin)-[:LIKES {score: 7}]->(italian)
CREATE (karin)-[:LIKES {score: 10}]->(portuguese)
```

The following will return the Cosine similarity of Michael and Arya:

```
MATCH (p1:Person {name: 'Michael'})-[likes1:LIKES]->(cuisine)
MATCH (p2:Person {name: "Arya"})-[likes2:LIKES]->(cuisine)
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.cosine(collect(likes1.score), collect(likes2.score)) AS similarity
```

Table 597. Results

from	to	similarity
"Michael"	"Arya"	0.9788908326303921

The following will return the Cosine similarity of Michael and the other people that have a cuisine in common:

```
MATCH (p1:Person {name: 'Michael'})-[likes1:LIKES]->(cuisine)
MATCH (p2:Person)-[likes2:LIKES]->(cuisine) WHERE p2 <> p1
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.cosine(collect(likes1.score), collect(likes2.score)) AS similarity
ORDER BY similarity DESC
```


Table 598. Results

from	to	similarity
"Michael"	"Arya"	0.9788908326303921
"Michael"	"Zhen"	0.9542262139256075
"Michael"	"Praveena"	0.9429903335828894
"Michael"	"Karin"	0.8498063272285821

## Cosine Similarity algorithm procedures examples

The Cosine Similarity procedure computes similarity between all pairs of items. It is a symmetrical algorithm, which means that the result from computing the similarity of Item A to Item B is the same as computing the similarity of Item B to Item A. We can therefore compute the score for each pair of nodes once. We don't compute the similarity of items to themselves.

The number of computations is  $((\# \text{ items})^2 / 2) - \# \text{ items}$ , which can be very computationally expensive if we have a lot of items.



Cosine Similarity is only calculated over non-NULL dimensions. The procedures expect to receive the same length lists for all items. Otherwise, longer lists will be trimmed to the length of the shortest list.

The following will create a sample graph:

```
CREATE (french:Cuisine {name:'French'})
CREATE (italian:Cuisine {name:'Italian'})
CREATE (indian:Cuisine {name:'Indian'})
CREATE (lebanese:Cuisine {name:'Lebanese'})
CREATE (portuguese:Cuisine {name:'Portuguese'})
CREATE (british:Cuisine {name:'British'})
CREATE (mauritian:Cuisine {name:'Mauritian'})

CREATE (zhen:Person {name: "Zhen"})
CREATE (praveena:Person {name: "Praveena"})
CREATE (michael:Person {name: "Michael"})
CREATE (arya:Person {name: "Arya"})
CREATE (karin:Person {name: "Karin"})

CREATE (praveena)-[:LIKES {score: 9}]->(indian)
CREATE (praveena)-[:LIKES {score: 7}]->(portuguese)
CREATE (praveena)-[:LIKES {score: 8}]->(british)
CREATE (praveena)-[:LIKES {score: 1}]->(mauritian)

CREATE (zhen)-[:LIKES {score: 10}]->(french)
CREATE (zhen)-[:LIKES {score: 6}]->(indian)
CREATE (zhen)-[:LIKES {score: 2}]->(british)

CREATE (michael)-[:LIKES {score: 8}]->(french)
CREATE (michael)-[:LIKES {score: 7}]->(italian)
CREATE (michael)-[:LIKES {score: 9}]->(indian)
CREATE (michael)-[:LIKES {score: 3}]->(portuguese)

CREATE (arya)-[:LIKES {score: 10}]->(lebanese)
CREATE (arya)-[:LIKES {score: 10}]->(italian)
CREATE (arya)-[:LIKES {score: 7}]->(portuguese)
CREATE (arya)-[:LIKES {score: 9}]->(mauritian)

CREATE (karin)-[:LIKES {score: 9}]->(lebanese)
CREATE (karin)-[:LIKES {score: 7}]->(italian)
CREATE (karin)-[:LIKES {score: 10}]->(portuguese)
```

## Stream

The following will return a stream of node pairs along with their Cosine similarities:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[:likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.cosine.stream({data: data})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC
```

Table 599. Results

from	to	similarity
"Praveena"	"Karin"	1.0
"Michael"	"Arya"	0.9788908326303921
"Arya"	"Karin"	0.9610904115204073
"Zhen"	"Michael"	0.9542262139256075
"Praveena"	"Michael"	0.9429903335828895
"Zhen"	"Praveena"	0.9191450300180579

from	to	similarity
"Michael"	"Karin"	0.8498063272285821
"Praveena"	"Arya"	0.7194014606174091
"Zhen"	"Arya"	0.0
"Zhen"	"Karin"	0.0

Praveena and Karin have the most similar food tastes, with a score of 1.0, and there are also several other pairs of users with similar tastes. The scores here are unusually high because our users haven't liked many of the same cuisines. We also have 2 pairs of users who are not similar at all. We'd probably want to filter those out, which we can do by passing in the `similarityCutoff` parameter.

The following will return a stream of node pairs that have a similarity of at least 0.1, along with their cosine similarities:

```

MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.cosine.stream({
  data: data,
  similarityCutoff: 0.0
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC

```

Table 600. Results

from	to	similarity
"Praveena"	"Karin"	1.0
"Michael"	"Arya"	0.9788908326303921
"Arya"	"Karin"	0.9610904115204073
"Zhen"	"Michael"	0.9542262139256075
"Praveena"	"Michael"	0.9429903335828895
"Zhen"	"Praveena"	0.9191450300180579
"Michael"	"Karin"	0.8498063272285821
"Praveena"	"Arya"	0.7194014606174091

We can see that those users with no similarity have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar `k` users for a given user. We can do that by passing in the `topK` parameter.

The following will return a stream of users along with the most similar user to them (i.e. k=1):

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.cosine.stream({
  data: data,
  similarityCutoff: 0.0,
  topK: 1
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY from
```

Table 601. Results

from	to	similarity
"Arya"	"Michael"	0.9788908326303921
"Karin"	"Praveena"	1.0
"Michael"	"Arya"	0.9788908326303921
"Praveena"	"Karin"	1.0
"Zhen"	"Michael"	0.9542262139256075

These results will not be symmetrical. For example, the person most similar to Zhen is Michael, but the person most similar to Michael is Arya.

Write

The following will find the most similar user for each user, and store a relationship between those users:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.cosine.write({
  data: data,
  topK: 1,
  similarityCutoff: 0.1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50,
p75, p90, p95, p99, p999, p100
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

Table 602. Results

nodes	similarityPairs	writeRelationshipType	writeProperty	min	max	mean	p95
5	5	"SIMILAR"	"score"	0.9542236328125	1.0000038146972656	0.9824020385742187	1.0000038146972656

We then could write a query to find out what types of cuisine that other people similar to us might like.

The following will find the most similar user to Praveena, and return their favourite cuisines that Praveena doesn't (yet!) like:

```
MATCH (p:Person {name: "Praveena"})-[:SIMILAR]->(other),
      (other)-[:LIKES]->(cuisine)
WHERE not((p)-[:LIKES]->(cuisine))
RETURN cuisine.name AS cuisine
```

Table 603. Results

cuisine
Italian
Lebanese

## Stats

The following will run the algorithm and returns the result in form of statistical and measurement values

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[:likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.cosine.stats({
  data: data,
  topK: 1,
  similarityCutoff: 0.1
})
YIELD nodes, similarityPairs, min, max, mean, p95
RETURN nodes, similarityPairs, min, max, mean, p95
```

## Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

The following will find the most similar person (i.e.  $k=1$ ) to Arya and Praveena:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[:likes:LIKES]->(c)
WITH {item:id(p), name: p.name, weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS personCuisines
WITH personCuisines,
     [value in personCuisines WHERE value.name IN ["Praveena", "Arya"] | value.item ] AS sourceIds
CALL gds.alpha.similarity.cosine.stream({
  data: personCuisines,
  sourceIds: sourceIds,
  topK: 1
})
YIELD item1, item2, similarity
WITH gds.util.asNode(item1) AS from, gds.util.asNode(item2) AS to, similarity
RETURN from.name AS from, to.name AS to, similarity
ORDER BY similarity DESC
```

Table 604. Results



from	to	similarity
Praveena	Karin	1.0
Arya	Michael	0.9788908326303921

## Skipping values

The algorithm checks every value in the input vectors against the `skipValue` to determine whether that value should be considered as part of the similarity computation. Vectors of different length are padded with `NaN` values which are skipped by default. Setting a `skipValue` allows skipping an additional value. A common value to skip is `0.0`.

The following will create a sample graph storing an embedding vector for each node:

```
CREATE (french:Cuisine {name:'French'})      SET french.embedding = [0.0, 0.33, 0.81, 0.52, 0.41]
CREATE (italian:Cuisine {name:'Italian'})    SET italian.embedding = [0.31, 0.72, 0.58, 0.67, 0.31]
CREATE (indian:Cuisine {name:'Indian'})      SET indian.embedding = [0.43, 0.0, 0.98, 0.51, 0.76]
CREATE (lebanese:Cuisine {name:'Lebanese'})  SET lebanese.embedding = [0.12, 0.23, 0.35, 0.31, 0.39]
CREATE (portuguese:Cuisine {name:'Portuguese'}) SET portuguese.embedding = [0.47, 0.98, 0.0, 0.72, 0.89]
CREATE (british:Cuisine {name:'British'})    SET british.embedding = [0.94, 0.12, 0.23, 0.4, 0.71]
CREATE (mauritian:Cuisine {name:'Mauritian'}) SET mauritian.embedding = [0.31, 0.56, 0.98, 0.0, 0.62]
```

The following will find the top 3 similarities between cuisines based on the `embedding` property:

```
MATCH (c:Cuisine)
WITH {item:id(c), weights: c.embedding} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.cosine.stream({
  data: data,
  skipValue: 0.0
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC, from ASC
LIMIT 3
```

Table 605. Results with skipping `0.0` values:

from	to	similarity
"Mauritian"	"Portuguese"	0.9955829148132149
"Portuguese"	"Mauritian"	0.9955829148132149
"Indian"	"Portuguese"	0.9954426605601884

Without skipping `0.0` values the result would look different:

Table 606. Results without skipping `0.0` values:

from	to	similarity
"Lebanese"	"French"	0.9372771447068958
"French"	"Lebanese"	0.9372771447068958
"Indian"	"Lebanese"	0.9110882139221992

## Cypher projection

If the similarity lists are very large they can take up a lot of memory. For cases where those lists contain lots of values that should be skipped, you can use the less memory-intensive approach of using Cypher statements to project the graph instead.

The Cypher projection expects to receive 3 fields:

- `item` - should contain node ids, which we can return using the `id` function.
- `category` - should contain node ids, which we can return using the `id` function.
- `weight` - should contain a double value.

Set `graph: 'cypher'` in the config:

```
WITH 'MATCH (person:Person)-[likes:LIKES]->(c)
      RETURN id(person) AS item, id(c) AS category, likes.score AS weight' AS query
CALL gds.alpha.similarity.cosine.write({
  data: query,
  graph: 'cypher',
  topK: 1,
  similarityCutoff: 0.1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

### 7.4.5. Pearson Similarity Alpha

[Pearson similarity](#) is the covariance of the two  $n$ -dimensional vectors divided by the product of their standard deviations.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

#### History and explanation

Pearson similarity is computed using the following formula:

$$\text{similarity}(A, B) = \frac{\text{cov}(A, B)}{\sigma_A \sigma_B} = \frac{\sum_{i=1}^n (A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^n (A_i - \bar{A})^2 (B_i - \bar{B})^2}}$$

Values range between -1 and 1, where -1 is perfectly dissimilar and 1 is perfectly similar.

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation and are therefore more appropriate for computing similarities on bigger datasets.

#### Use-cases - when to use the Pearson Similarity algorithm

We can use the Pearson Similarity algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, to get movie

recommendations based on the preferences of users who have given similar ratings to other movies that you've seen.

## Pearson Similarity algorithm function sample

The Pearson Similarity function computes the similarity of two lists of numbers.



Pearson Similarity is only calculated over non-NULL dimensions. When calling the function, we should provide lists that contain the overlapping items.

We can use it to compute the similarity of two hardcoded lists.

The following will return the Pearson similarity of two lists of numbers:

```
RETURN gds.alpha.similarity.pearson([5,8,7,5,4,9], [7,8,6,6,4,5]) AS similarity
```

Table 607. Results

similarity

0.28767798089123053

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

The following will create a sample graph:

```
MERGE (home_alone:Movie {name:'Home Alone'})
MERGE (matrix:Movie {name:'The Matrix'})
MERGE (good_men:Movie {name:'A Few Good Men'})
MERGE (top_gun:Movie {name:'Top Gun'})
MERGE (jerry:Movie {name:'Jerry Maguire'})
MERGE (gruffalo:Movie {name:'The Gruffalo'})

MERGE (zhen:Person {name: 'Zhen'})
MERGE (praveena:Person {name: 'Praveena'})
MERGE (michael:Person {name: 'Michael'})
MERGE (arya:Person {name: 'Arya'})
MERGE (karin:Person {name: 'Karin'})

MERGE (zhen)-[:RATED {score: 2}]->(home_alone)
MERGE (zhen)-[:RATED {score: 2}]->(good_men)
MERGE (zhen)-[:RATED {score: 3}]->(matrix)
MERGE (zhen)-[:RATED {score: 6}]->(jerry)

MERGE (praveena)-[:RATED {score: 6}]->(home_alone)
MERGE (praveena)-[:RATED {score: 7}]->(good_men)
MERGE (praveena)-[:RATED {score: 8}]->(matrix)
MERGE (praveena)-[:RATED {score: 9}]->(jerry)

MERGE (michael)-[:RATED {score: 7}]->(home_alone)
MERGE (michael)-[:RATED {score: 9}]->(good_men)
MERGE (michael)-[:RATED {score: 3}]->(jerry)
MERGE (michael)-[:RATED {score: 4}]->(top_gun)

MERGE (arya)-[:RATED {score: 8}]->(top_gun)
MERGE (arya)-[:RATED {score: 1}]->(matrix)
MERGE (arya)-[:RATED {score: 10}]->(jerry)
MERGE (arya)-[:RATED {score: 10}]->(gruffalo)

MERGE (karin)-[:RATED {score: 9}]->(top_gun)
MERGE (karin)-[:RATED {score: 7}]->(matrix)
MERGE (karin)-[:RATED {score: 7}]->(home_alone)
MERGE (karin)-[:RATED {score: 9}]->(gruffalo)
```

The following will return the Pearson similarity of Arya and Karin:

```
MATCH (p1:Person {name: 'Arya'})-[rated:RATED]->(movie)
WITH p1, gds.alpha.similarity.asVector(movie, rated.score) AS p1Vector
MATCH (p2:Person {name: 'Karin'})-[rated:RATED]->(movie)
WITH p1, p2, p1Vector, gds.alpha.similarity.asVector(movie, rated.score) AS p2Vector
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.pearson(p1Vector, p2Vector, {vectorType: "maps"}) AS similarity
```

Table 608. Results

from	to	similarity
"Arya"	"Karin"	0.8194651785206903

In this example, we pass in `vectorType: "maps"` as an extra parameter, as well as using the `gds.alpha.similarity.asVector` function to construct a vector of maps containing each movie and the corresponding rating. We do this because the Pearson Similarity algorithm needs to compute the average of all the movies that a user has reviewed, not just the ones that they have in common with the user we're comparing them to. We can't therefore just pass in collections of the ratings of movies that have been reviewed by both people.

The following will return the Pearson similarity of Arya and other people that have rated at least one movie:

```
MATCH (p1:Person {name: 'Arya'})-[rated:RATED]->(movie)
WITH p1, gds.alpha.similarity.asVector(movie, rated.score) AS p1Vector
MATCH (p2:Person)-[rated:RATED]->(movie) WHERE p2 <> p1
WITH p1, p2, p1Vector, gds.alpha.similarity.asVector(movie, rated.score) AS p2Vector
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.pearson(p1Vector, p2Vector, {vectorType: "maps"}) AS similarity
ORDER BY similarity DESC
```

Table 609. Results

from	to	similarity
"Arya"	"Karin"	0.8194651785206903
"Arya"	"Zhen"	0.4839533792540704
"Arya"	"Praveena"	0.09262336892949784
"Arya"	"Michael"	-0.9551953674747637

## Pearson Similarity algorithm procedures sample

The Pearson Similarity procedure computes similarity between all pairs of items. It is a symmetrical algorithm, which means that the result from computing the similarity of Item A to Item B is the same as computing the similarity of Item B to Item A. We can therefore compute the score for each pair of nodes once. We don't compute the similarity of items to themselves.

The number of computations is  $((\# \text{ items})^2 / 2) - \# \text{ items}$ , which can be very computationally expensive if we have a lot of items.



Pearson Similarity is only calculated over non-NULL dimensions. The procedures expect to receive the same length lists for all items. Otherwise, longer lists will be trimmed to the length of the shortest list.

The following will create a sample graph:

```

MERGE (home_alone:Movie {name:'Home Alone'})
MERGE (matrix:Movie {name:'The Matrix'})
MERGE (good_men:Movie {name:'A Few Good Men'})
MERGE (top_gun:Movie {name:'Top Gun'})
MERGE (jerry:Movie {name:'Jerry Maguire'})
MERGE (gruffalo:Movie {name:'The Gruffalo'})

MERGE (zhen:Person {name:'Zhen'})
MERGE (praveena:Person {name:'Praveena'})
MERGE (michael:Person {name:'Michael'})
MERGE (arya:Person {name:'Arya'})
MERGE (karin:Person {name:'Karin'})

MERGE (zhen)-[:RATED {score: 2}]->(home_alone)
MERGE (zhen)-[:RATED {score: 2}]->(good_men)
MERGE (zhen)-[:RATED {score: 3}]->(matrix)
MERGE (zhen)-[:RATED {score: 6}]->(jerry)

MERGE (praveena)-[:RATED {score: 6}]->(home_alone)
MERGE (praveena)-[:RATED {score: 7}]->(good_men)
MERGE (praveena)-[:RATED {score: 8}]->(matrix)
MERGE (praveena)-[:RATED {score: 9}]->(jerry)

MERGE (michael)-[:RATED {score: 7}]->(home_alone)
MERGE (michael)-[:RATED {score: 9}]->(good_men)
MERGE (michael)-[:RATED {score: 3}]->(jerry)
MERGE (michael)-[:RATED {score: 4}]->(top_gun)

MERGE (arya)-[:RATED {score: 8}]->(top_gun)
MERGE (arya)-[:RATED {score: 1}]->(matrix)
MERGE (arya)-[:RATED {score: 10}]->(jerry)
MERGE (arya)-[:RATED {score: 10}]->(gruffalo)

MERGE (karin)-[:RATED {score: 9}]->(top_gun)
MERGE (karin)-[:RATED {score: 7}]->(matrix)
MERGE (karin)-[:RATED {score: 7}]->(home_alone)
MERGE (karin)-[:RATED {score: 9}]->(gruffalo)

```

## Stream

The following will return a stream of node pairs along with their Pearson similarities:

```

MATCH (p:Person), (m:Movie)
OPTIONAL MATCH (p)-[:rated:RATED]->(m)
WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.pearson.stream({
  data: data,
  topK: 0
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC

```

Table 610. Results

from	to	similarity
"Zhen"	"Praveena"	0.8865926413116155

from	to	similarity
"Zhen"	"Karin"	0.8320502943378437
"Arya"	"Karin"	0.8194651785206903
"Zhen"	"Arya"	0.4839533792540704
"Praveena"	"Karin"	0.4472135954999579
"Praveena"	"Arya"	0.09262336892949784
"Praveena"	"Michael"	-0.788492846568306
"Zhen"	"Michael"	-0.9091365607973364
"Michael"	"Arya"	-0.9551953674747637
"Michael"	"Karin"	-0.9863939238321437

Zhen and Praveena are the most similar with a score of 0.88. The maximum score is 1.0 We also have 4 pairs of users who are not similar at all. We'd probably want to filter those out, which we can do by passing in the `similarityCutoff` parameter.

The following will return a stream of node pairs that have a similarity of at least 0.1, along with their Pearson similarities:

```
MATCH (p:Person), (m:Movie)
OPTIONAL MATCH (p)-[rated:RATED]->(m)
WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.pearson.stream({
  data: data,
  similarityCutoff: 0.1,
  topK: 0
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC
```

Table 611. Results

from	to	similarity
"Zhen"	"Praveena"	0.8865926413116155
"Zhen"	"Karin"	0.8320502943378437
"Arya"	"Karin"	0.8194651785206903
"Zhen"	"Arya"	0.4839533792540704
"Praveena"	"Karin"	0.4472135954999579

We can see that those users with no similarity have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar `k` users for a given user. We can do that by passing in the `topK` parameter.

The following will return a stream of users along with the most similar user to them (i.e. k=1):

```
MATCH (p:Person), (m:Movie)
OPTIONAL MATCH (p)-[rated:RATED]->(m)
WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.pearson.stream({
  data: data,
  topK:1,
  similarityCutoff: 0.0
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC
```

Table 612. Results

from	to	similarity
"Zhen"	"Praveena"	0.8865926413116155
"Praveena"	"Zhen"	0.8865926413116155
"Karin"	"Zhen"	0.8320502943378437
"Arya"	"Karin"	0.8194651785206903

These results will not necessarily be symmetrical. For example, the person most similar to Arya is Karin, but the person most similar to Karin is Zhen.

Write

The following will find the most similar user for each user, and store a relationship between those users:

```
MATCH (p:Person), (m:Movie)
OPTIONAL MATCH (p)-[rated:RATED]->(m)
WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.pearson.write({
  data: data,
  topK: 1,
  similarityCutoff: 0.1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75,
p90, p95, p99, p999, p100
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

Table 613. Results

nodes	similarityPairs	writeRelationshipType	writeProperty	min	max	mean	p95
5	4	"SIMILAR"	"score"	0.8194618225097656	0.8865890502929688	0.8561716079711914	0.8865890502929688

We then could write a query to find out which are the movies that other people similar to us liked.

The following will find the most similar user to Karin, and return their movies that Karin didn't (yet!) rate:

```
MATCH (p:Person {name: 'Karin'})-[:SIMILAR]->(other),
      (other)-[:RATED]->(movie)
WHERE not((p)-[:RATED]->(movie)) and r.score >= 5
RETURN movie.name AS movie
```

Table 614. Results

movie
Jerry Maguire

## Stats

The following will run the algorithm and returns the result in form of statistical and measurement values

```
MATCH (p:Person), (m:Movie)
OPTIONAL MATCH (p)-[:rated:RATED]->(m)
WITH {item:id(p), weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.pearson.stats({
  data: data,
  topK: 1,
  similarityCutoff: 0.1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

## Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

The following will find the most similar person (i.e. `k=1`) to Arya and Praveena:

```
MATCH (p:Person), (m:Movie)
OPTIONAL MATCH (p)-[:rated:RATED]->(m)
WITH {item:id(p), name: p.name, weights: collect(coalesce(rated.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS personCuisines
WITH personCuisines,
     [value in personCuisines WHERE value.name IN ["Praveena", "Arya"] | value.item ] AS sourceIds
CALL gds.alpha.similarity.pearson.stream({
  data: personCuisines,
  sourceIds: sourceIds,
  topK: 1
})
YIELD item1, item2, similarity
WITH gds.util.asNode(item1) AS from, gds.util.asNode(item2) AS to, similarity
RETURN from.name AS from, to.name AS to, similarity
ORDER BY similarity DESC
```

Table 615. Results

from	to	similarity
Praveena	Zhen	0.8865926413116155
Arya	Karin	0.8194651785206903



## Skipping values

By default the `skipValue` parameter is `gds.util.NaN()`. The algorithm checks every value against the `skipValue` to determine whether that value should be considered as part of the similarity result. For cases where no values should be skipped, skipping can be disabled by setting `skipValue` to `null`.

The following will create a sample graph:

```
MERGE (home_alone:Movie {name:'Home Alone'}) SET home_alone.embedding = [0.71, 0.33, 0.81, 0.52, 0.41]
MERGE (matrix:Movie {name:'The Matrix'}) SET matrix.embedding = [0.31, 0.72, 0.58, 0.67, 0.31]
MERGE (good_men:Movie {name:'A Few Good Men'}) SET good_men.embedding = [0.43, 0.26, 0.98, 0.51, 0.76]
MERGE (top_gun:Movie {name:'Top Gun'}) SET top_gun.embedding = [0.12, 0.23, 0.35, 0.31, 0.3]
MERGE (jerry:Movie {name:'Jerry Maguire'}) SET jerry.embedding = [0.47, 0.98, 0.81, 0.72, 0]
```

The following will find the similarity between movies based on the `embedding` property:

```
MATCH (m:Movie)
WITH {item:id(m), weights: m.embedding} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.pearson.stream({
  data: data,
  skipValue: null
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC
```

Table 616. Results

from	to	similarity
The Matrix	Jerry Maguire	0.8689113641953199
A Few Good Men	Top Gun	0.6846566091701214
Home Alone	A Few Good Men	0.556559508845268
The Matrix	Top Gun	0.39320549183813097
Home Alone	Jerry Maguire	0.10026787755714502
Top Gun	Jerry Maguire	0.056232940630734043
Home Alone	Top Gun	0.006048691083898151
Home Alone	The Matrix	-0.23435051666541426
The Matrix	A Few Good Men	-0.2545273235448378
A Few Good Men	Jerry Maquire	-0.31099199179883635

## Cypher projection

If the similarity lists are very large they can take up a lot of memory. For cases where those lists contain lots of values that should be skipped, you can use the less memory-intensive approach of using Cypher statements to project the graph instead.

The Cypher projection expects to receive 3 fields:

- `item` - should contain node ids, which we can return using the `id` function.

- `category` - should contain node ids, which we can return using the `id` function.
- `weight` - should contain a double value.

Set `graph: 'cypher'` in the config:

```
WITH "MATCH (person:Person)-[rated:RATED]->(c)
      RETURN id(person) AS item, id(c) AS category, rated.score AS weight" AS query
CALL gds.alpha.similarity.pearson({
  data: query,
  graph: 'cypher',
  topK: 1,
  similarityCutoff: 0.1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

Table 617. Results

nodes	similarityPairs	writeRelationshipType	writeProperty	min	max	mean	p95
5	4	"SIMILAR"	"score"	0.8194618225097656	0.8865890502929688	0.8561716079711914	0.8865890502929688

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.similarity.pearson.write(configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100
```

Table 618. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 619. Configuration

Name	Type	Default	Optional	Description
data	List or String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.

Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
writeBatchSize	Integer	10000	yes	The batch size to use when storing results.
writeRelationshipType	String	SIMILAR	yes	The relationship type to use when storing results.
writeProperty	String	score	yes	The property to use when storing results.
sourceIds	List of String	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of String	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 620. Results

Name	Type	Description
nodes	Integer	The number of nodes passed in.
similarityPairs	Integer	The number of pairs of similar nodes computed.
writeRelationshipType	String	The relationship type used when storing results.
writeProperty	String	The property used when storing results.
min	Float	The minimum similarity score computed.
max	Float	The maximum similarity score computed.
mean	Float	The mean of similarities scores computed.
stdDev	Float	The standard deviation of similarities scores computed.
p25	Float	The 25 percentile of similarities scores computed.
p50	Float	The 50 percentile of similarities scores computed.
p75	Float	The 75 percentile of similarities scores computed.
p90	Float	The 90 percentile of similarities scores computed.
p95	Float	The 95 percentile of similarities scores computed.
p99	Float	The 99 percentile of similarities scores computed.
p999	Float	The 99.9 percentile of similarities scores computed.
p100	Float	The 100 percentile of similarities scores computed.

The following will run the algorithm and stream results:

```
CALL gds.alpha.similarity.pearson.stream(configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```

Table 621. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 622. Configuration

Name	Type	Default	Optional	Description
data	List or String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
sourceIds	List of Integer	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of Integer	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 623. Results

Name	Type	Description
item1	Integer	The ID of one node in the similarity pair.
item2	Integer	The ID of other node in the similarity pair.
count1	Integer	The size of the <code>targets</code> list of one node.
count2	Integer	The size of the <code>targets</code> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <code>targets</code> lists.
similarity	Integer	The pearson similarity of the two nodes.

## 7.4.6. Euclidean Distance Alpha

Euclidean distance measures the straight line distance between two points in n-dimensional space.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

Euclidean distance is computed using the following formula:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation and are therefore more appropriate for computing similarities on bigger datasets.

### Use-cases - when to use the Euclidean Distance algorithm

We can use the Euclidean Distance algorithm to work out the similarity between two things. We might then use the computed similarity as part of a recommendation query. For example, to get movie recommendations based on the preferences of users who have given similar ratings to other movies that you've seen.

### Euclidean Distance algorithm function sample

The Euclidean Distance function computes the similarity of two lists of numbers.



Euclidean Distance is only calculated over non-NULL dimensions. When calling the function, we should provide lists that contain the overlapping items.

We can use it to compute the similarity of two hardcoded lists.

The following will return the euclidean similarity of two lists of numbers:

```
RETURN gds.alpha.similarity.euclideanDistance([3,8,7,5,2,9], [10,8,6,6,4,5]) AS similarity
```

Table 624. Results

similarity
8.426149773176359

These two lists of numbers have a euclidean distance of 8.42.

We can also use it to compute the similarity of nodes based on lists computed by a Cypher query.

The following will create a sample graph:

```

MERGE (french:Cuisine {name:'French'})
MERGE (italian:Cuisine {name:'Italian'})
MERGE (indian:Cuisine {name:'Indian'})
MERGE (lebanese:Cuisine {name:'Lebanese'})
MERGE (portuguese:Cuisine {name:'Portuguese'})
MERGE (british:Cuisine {name:'British'})
MERGE (mauritian:Cuisine {name:'Mauritian'})

MERGE (zhen:Person {name: "Zhen"})
MERGE (praveena:Person {name: "Praveena"})
MERGE (michael:Person {name: "Michael"})
MERGE (arya:Person {name: "Arya"})
MERGE (karin:Person {name: "Karin"})

MERGE (praveena)-[:LIKES {score: 9}]->(indian)
MERGE (praveena)-[:LIKES {score: 7}]->(portuguese)
MERGE (praveena)-[:LIKES {score: 8}]->(british)
MERGE (praveena)-[:LIKES {score: 1}]->(mauritian)

MERGE (zhen)-[:LIKES {score: 10}]->(french)
MERGE (zhen)-[:LIKES {score: 6}]->(indian)
MERGE (zhen)-[:LIKES {score: 2}]->(british)

MERGE (michael)-[:LIKES {score: 8}]->(french)
MERGE (michael)-[:LIKES {score: 7}]->(italian)
MERGE (michael)-[:LIKES {score: 9}]->(indian)
MERGE (michael)-[:LIKES {score: 3}]->(portuguese)

MERGE (arya)-[:LIKES {score: 10}]->(lebanese)
MERGE (arya)-[:LIKES {score: 10}]->(italian)
MERGE (arya)-[:LIKES {score: 7}]->(portuguese)
MERGE (arya)-[:LIKES {score: 9}]->(mauritian)

MERGE (karin)-[:LIKES {score: 9}]->(lebanese)
MERGE (karin)-[:LIKES {score: 7}]->(italian)
MERGE (karin)-[:LIKES {score: 10}]->(portuguese)

```

The following will return the Euclidean distance of Zhen and Praveena:

```

MATCH (p1:Person {name: 'Zhen'})-[:likes1:LIKES]->(cuisine)
MATCH (p2:Person {name: 'Praveena'})-[:likes2:LIKES]->(cuisine)
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.euclideanDistance(collect(likes1.score), collect(likes2.score)) AS similarity

```

Table 625. Results

from	to	similarity
"Zhen"	"Praveena"	6.708203932499369

The following will return the Euclidean distance of Zhen and the other people that have a cuisine in common:

```

MATCH (p1:Person {name: 'Zhen'})-[:likes1:LIKES]->(cuisine)
MATCH (p2:Person)-[:likes2:LIKES]->(cuisine) WHERE p2 <> p1
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.euclideanDistance(collect(likes1.score), collect(likes2.score)) AS similarity
ORDER BY similarity DESC

```

Table 626. Results

from	to	similarity
"Zhen"	"Praveena"	6.708203932499369
"Zhen"	"Michael"	3.605551275463989

## Euclidean Distance algorithm procedures sample

The Euclidean Distance procedure computes similarity between all pairs of items. It is a symmetrical algorithm, which means that the result from computing the similarity of Item A to Item B is the same as computing the similarity of Item B to Item A. We can therefore compute the score for each pair of nodes once. We don't compute the similarity of items to themselves.

The number of computations is  $((\# \text{ items})^2 / 2) - \# \text{ items}$ , which can be very computationally expensive if we have a lot of items.



Euclidean Distance is only calculated over non-NULL dimensions. The procedures expect to receive the same length lists for all items. Otherwise, longer lists will be trimmed to the length of the shortest list.

The following will create a sample graph:

```

MERGE (french:Cuisine {name:'French'})
MERGE (italian:Cuisine {name:'Italian'})
MERGE (indian:Cuisine {name:'Indian'})
MERGE (lebanese:Cuisine {name:'Lebanese'})
MERGE (portuguese:Cuisine {name:'Portuguese'})
MERGE (karin:Person {name: "Karin"})

MERGE (praveena)-[:LIKES {score: 9}]->(indian)
MERGE (praveena)-[:LIKES {score: 7}]->(portuguese)
MERGE (praveena)-[:LIKES {score: 8}]->(british)
MERGE (praveena)-[:LIKES {score: 1}]->(mauritian)

MERGE (zhen)-[:LIKES {score: 10}]->(french)
MERGE (zhen)-[:LIKES {score: 6}]->(indian)
MERGE (zhen)-[:LIKES {score: 2}]->(british)

MERGE (british:Cuisine {name:'British'})
MERGE (mauritian:Cuisine {name:'Mauritian'})

MERGE (zhen:Person {name: "Zhen"})
MERGE (praveena:Person {name: "Praveena"})
MERGE (michael:Person {name: "Michael"})
MERGE (arya:Person {name: "Arya"})
MERGE (michael)-[:LIKES {score: 8}]->(french)
MERGE (michael)-[:LIKES {score: 7}]->(italian)
MERGE (michael)-[:LIKES {score: 9}]->(indian)
MERGE (michael)-[:LIKES {score: 3}]->(portuguese)

MERGE (arya)-[:LIKES {score: 10}]->(lebanese)
MERGE (arya)-[:LIKES {score: 10}]->(italian)
MERGE (arya)-[:LIKES {score: 7}]->(portuguese)
MERGE (arya)-[:LIKES {score: 9}]->(mauritian)

MERGE (karin)-[:LIKES {score: 9}]->(lebanese)
MERGE (karin)-[:LIKES {score: 7}]->(italian)
MERGE (karin)-[:LIKES {score: 10}]->(portuguese)

```

## Stream

The following will return a stream of node pairs, along with their intersection and euclidean similarities:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.stream({
  data: data,
  topK: 0
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity
```

Table 627. Results

from	to	similarity
"Praveena"	"Karin"	3.0
"Zhen"	"Michael"	3.605551275463989
"Praveena"	"Michael"	4.0
"Arya"	"Karin"	4.358898943540674
"Michael"	"Arya"	5.0
"Zhen"	"Praveena"	6.708203932499369
"Michael"	"Karin"	7.0
"Praveena"	"Arya"	8.0
"Zhen"	"Arya"	NaN
"Zhen"	"Karin"	NaN

Praveena and Karin have the most similar food preferences, with a euclidean distance of 3.0. Lower scores are better here; a score of 0 would indicate that users have exactly the same preferences.

We can also see at the bottom of the list that Zhen and Arya and Zhen and Karin have a similarity of **NaN**. We get this result because there is no overlap in their food preferences.

We can filter those results out using the `gds.util.isFinite` function.

The following will return a stream of node pairs, along with their intersection and finite euclidean similarities:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.stream({
  data: data,
  topK: 0
})
YIELD item1, item2, count1, count2, similarity
WHERE gds.util.isFinite(similarity)
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity
```



Table 628. Results

from	to	similarity
"Praveena"	"Karin"	3.0
"Zhen"	"Michael"	3.605551275463989
"Praveena"	"Michael"	4.0
"Arya"	"Karin"	4.358898943540674
"Michael"	"Arya"	5.0
"Zhen"	"Praveena"	6.708203932499369
"Michael"	"Karin"	7.0
"Praveena"	"Arya"	8.0

We can see in these results that Zhen and Arya and Zhen and Karin have been removed.

We might decide that we don't want to see users with a similarity above 4 returned in our results. If so, we can filter those out by passing in the `similarityCutoff` parameter.

The following will return a stream of node pairs that have a similarity of at most 4, along with their euclidean distance:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.stream({
  data: data,
  similarityCutoff: 4.0,
  topK: 0
})
YIELD item1, item2, count1, count2, similarity
WHERE gds.util.isFinite(similarity)
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity
```

Table 629. Results

from	to	similarity
"Praveena"	"Karin"	3.0
"Zhen"	"Michael"	3.605551275463989
"Praveena"	"Michael"	4.0

We can see that those users with a high score have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar `k` users for a given user. We can do that by passing in the `topK` parameter.

The following will return a stream of users along with the most similar user to them (i.e. k=1):

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.stream({
  data: data,
  topK: 1
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY from
```

Table 630. Results

from	to	similarity
"Arya"	"Karin"	4.358898943540674
"Karin"	"Praveena"	3.0
"Michael"	"Zhen"	3.605551275463989
"Praveena"	"Karin"	3.0
"Zhen"	"Michael"	3.605551275463989

These results will not necessarily be symmetrical. For example, the person most similar to Arya is Karin, but the person most similar to Karin is Praveena.

Write

The following will find the most similar user for each user, and store a relationship between those users:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.write({
  data: data,
  topK: 1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50,
p75, p90, p95, p99, p999, p100
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

Table 631. Results

nodes	similarityPairs	writeRelationshipType	writeProperty	min	max	mean	p95
5	5	"SIMILAR"	"score"	3.0	4.358901977 5390625	3.513998413 0859374	4.358901977 5390625

We then could write a query to find out what types of cuisine that other people similar to us might like.

The following will find the most similar user to Praveena, and return their favorite cuisines that Praveena doesn't (yet!) like:

```
MATCH (p:Person {name: "Praveena"})-[:SIMILAR]->(other),
      (other)-[:LIKES]->(cuisine)
WHERE not((p)-[:LIKES]->(cuisine))
RETURN cuisine.name AS cuisine
```

Table 632. Results

cuisine
Italian
Lebanese

## Stats

The following will run the algorithm and returns the result in form of statistical and measurement values

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[:likes:LIKES]->(c)
WITH {item:id(p), weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.stats({
  data: data,
  topK: 1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

## Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

The following will find the most similar person (i.e. `k=1`) to Arya and Praveena:

```
MATCH (p:Person), (c:Cuisine)
OPTIONAL MATCH (p)-[:likes:LIKES]->(c)
WITH {item:id(p), name: p.name, weights: collect(coalesce(likes.score, gds.util.NaN()))} AS userData
WITH collect(userData) AS personCuisines
WITH personCuisines,
     [value in personCuisines WHERE value.name IN ["Praveena", "Arya"] | value.item ] AS sourceIds
CALL gds.alpha.similarity.euclidean.stream({
  data: personCuisines,
  sourceIds: sourceIds,
  topK: 1
})
YIELD item1, item2, similarity
WITH gds.util.asNode(item1) AS from, gds.util.asNode(item2) AS to, similarity
RETURN from.name AS from, to.name AS to, similarity
ORDER BY similarity DESC
```

Table 633. Results

from	to	similarity
"Arya"	"Karin"	4.358898943540674
"Praveena"	"Karin"	3.0

## Skipping values

By default the `skipValue` parameter is `gds.util.NaN()`. The algorithm checks every value against the `skipValue` to determine whether that value should be considered as part of the similarity result. For cases where no values should be skipped, skipping can be disabled by setting `skipValue` to `null`.

The following will create a sample graph:

```

MERGE (french:Cuisine {name:'French'})      SET french.embedding = [0.71, 0.33, 0.81, 0.52, 0.41]
MERGE (italian:Cuisine {name:'Italian'})    SET italian.embedding = [0.31, 0.72, 0.58, 0.67, 0.31]
MERGE (indian:Cuisine {name:'Indian'})      SET indian.embedding = [0.43, 0.26, 0.98, 0.51, 0.76]
MERGE (lebanese:Cuisine {name:'Lebanese'})  SET lebanese.embedding = [0.12, 0.23, 0.35, 0.31, 0.39]
MERGE (portuguese:Cuisine {name:'Portuguese'}) SET portuguese.embedding = [0.47, 0.98, 0.81, 0.72, 0.89]
MERGE (british:Cuisine {name:'British'})    SET british.embedding = [0.94, 0.12, 0.23, 0.4, 0.71]
MERGE (mauritian:Cuisine {name:'Mauritian'}) SET mauritian.embedding = [0.31, 0.56, 0.98, 0.21, 0.62]

```

The following will find the similarity between cuisines based on the `embedding` property:

```

MATCH (c:Cuisine)
WITH {item:id(c), weights: c.embedding} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.euclidean.stream({
  data: data,
  skipValue: null
})
YIELD item1, item2, count1, count2, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC

```

## Cypher projection

If the similarity lists are very large they can take up a lot of memory. For cases where those lists contain lots of values that should be skipped, you can use the less memory-intensive approach of using Cypher statements to project the graph instead.

The Cypher projection expects to receive 3 fields:

- `item` - should contain node ids, which we can return using the `id` function.
- `category` - should contain node ids, which we can return using the `id` function.
- `weight` - should contain a double value.

Set `graph: 'cypher'` in the config:

```
WITH "MATCH (person:Person)-[likes:LIKES]->(c)
      RETURN id(person) AS item, id(c) AS category, likes.score AS weight" AS query
CALL gds.alpha.similarity.euclidean.write({
  data: query,
  graph: 'cypher',
  topK: 1,
  similarityCutoff: 4.0
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.similarity.euclidean.write(configuration: Map)
YIELD nodes, similarityPair, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75,
p90, p95, p99, p999, p100
```

Table 634. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 635. Configuration

Name	Type	Default	Optional	Description
data	List or String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for <code>'writeConcurrency'</code> .
writeConcurrency	Integer	value of <code>'concurrency'</code>	yes	The number of concurrent threads used for writing the result.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
writeBatchSize	Integer	10000	yes	The batch size to use when storing results.

Name	Type	Default	Optional	Description
writeRelationshipType	String	SIMILAR	yes	The relationship type to use when storing results.
writeProperty	String	score	yes	The property to use when storing results.
sourceIds	List of String	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of String	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 636. Results

Name	Type	Description
nodes	Integer	The number of nodes passed in.
similarityPairs	Integer	The number of pairs of similar nodes computed.
writeRelationshipType	String	The relationship type used when storing results.
writeProperty	String	The property used when storing results.
min	Float	The minimum similarity score computed.
max	Float	The maximum similarity score computed.
mean	Float	The mean of similarities scores computed.
stdDev	Float	The standard deviation of similarities scores computed.
p25	Float	The 25 percentile of similarities scores computed.
p50	Float	The 50 percentile of similarities scores computed.
p75	Float	The 75 percentile of similarities scores computed.
p90	Float	The 90 percentile of similarities scores computed.
p95	Float	The 95 percentile of similarities scores computed.
p99	Float	The 99 percentile of similarities scores computed.
p999	Float	The 99.9 percentile of similarities scores computed.
p100	Float	The 100 percentile of similarities scores computed.

The following will run the algorithm and stream results:

```
CALL gds.alpha.similarity.euclidean.stream(configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```

Table 637. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 638. Configuration

Name	Type	Default	Optional	Description
data	List or String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
graph	String	<code>dense</code>	yes	The graph name ('dense' or 'cypher').
sourceIds	List of Integer	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of Integer	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 639. Results

Name	Type	Description
item1	Integer	The ID of one node in the similarity pair.
item2	Integer	The ID of other node in the similarity pair.
count1	Integer	The size of the <code>targets</code> list of one node.
count2	Integer	The size of the <code>targets</code> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <code>targets</code> lists.
similarity	Integer	The euclidean similarity of the two nodes.

### 7.4.7. Overlap Similarity Alpha

[Overlap similarity](#) measures overlap between two sets. It is defined as the size of the intersection of two sets, divided by the size of the smaller of the two sets.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

#### History and explanation

Overlap similarity is computed using the following formula:

$$O(A,B) = \frac{|A \cap B|}{\min(|A|, |B|)}$$

The library contains both procedures and functions to calculate similarity between sets of data. The function is best used when calculating the similarity between small numbers of sets. The procedures parallelize the computation, and are therefore more appropriate for computing similarities on bigger datasets.

## Use-cases - when to use the Overlap Similarity algorithm

We can use the Overlap Similarity algorithm to work out which things are subsets of others. We might then use these computed subsets to [learn a taxonomy from tagged data](#), as described by Jesús Barrasa.

## Overlap Similarity algorithm function sample

The following will return the Overlap similarity of two lists of numbers:

```
RETURN gds.alpha.similarity.overlap([1,2,3], [1,2,4,5]) AS similarity
```

Table 640. Results

similarity

0.6666666666666666

These two lists of numbers have an overlap similarity of 0.66. We can see how this result is derived by breaking down the formula:

```
O(A,B) = ( A ∩ B ) / (min( |A|, |B| ))
O(A,B) = 2 / min(3,4)
        = 2 / 3
        = 0.66
```

## Overlap Similarity algorithm procedures sample



The following will create a sample graph:

```
CREATE
(fahrenheit451:Book {title:'Fahrenheit 451'}),
(dune:Book {title:'Dune'}),
(hungerGames:Book {title:'The Hunger Games'}),
(nineteen84:Book {title:'1984'}),
(gatsby:Book {title:'The Great Gatsby'}),

(scienceFiction:Genre {name: "Science Fiction"}),
(fantasy:Genre {name: "Fantasy"}),
(dystopia:Genre {name: "Dystopia"}),
(classics:Genre {name: "Classics"}),

(fahrenheit451)-[:HAS_GENRE]->(dystopia),
(fahrenheit451)-[:HAS_GENRE]->(scienceFiction),
(fahrenheit451)-[:HAS_GENRE]->(fantasy),
(fahrenheit451)-[:HAS_GENRE]->(classics),

(hungerGames)-[:HAS_GENRE]->(scienceFiction),
(hungerGames)-[:HAS_GENRE]->(fantasy),

(nineteen84)-[:HAS_GENRE]->(scienceFiction),
(nineteen84)-[:HAS_GENRE]->(dystopia),
(nineteen84)-[:HAS_GENRE]->(classics),

(dune)-[:HAS_GENRE]->(scienceFiction),
(dune)-[:HAS_GENRE]->(fantasy),
(dune)-[:HAS_GENRE]->(classics),

(gatsby)-[:HAS_GENRE]->(classics)
```

Stream

The following will return a stream of node pairs, along with their intersection and overlap similarities:

```
MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), categories: collect(id(book))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.overlap.stream({data: data})
YIELD item1, item2, count1, count2, intersection, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to,
       count1, count2, intersection, similarity
ORDER BY similarity DESC
```

Table 641. Results

from	to	count1	count2	intersection	similarity
Fantasy	Science Fiction	3	4	3	1.0
Dystopia	Science Fiction	2	4	2	1.0
Dystopia	Classics	2	4	2	1.0
Science Fiction	Classics	4	4	3	0.75
Fantasy	Classics	3	4	2	0.66
Dystopia	Fantasy	2	3	1	0.5

Fantasy and Dystopia are both clear subgenres of Science Fiction - 100% of the books that list those as genres also list Science Fiction as a genre. Dystopia is also a subgenre of Classics. The others are less obvious; Dystopia probably isn't a subgenre of Fantasy, but the other two pairs could be subgenres.

The following will return a stream of node pairs that have a similarity of at least 0.75, along with their intersection and overlap similarities:

```
MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), categories: collect(id(book))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.overlap.stream({
  data: data,
  similarityCutoff: 0.75
})
YIELD item1, item2, count1, count2, intersection, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to,
       count1, count2, intersection, similarity
ORDER BY similarity DESC
```

Table 642. Results

from	to	count1	count2	intersection	similarity
Fantasy	Science Fiction	3	4	3	1.0
Dystopia	Classics	2	4	2	1.0
Dystopia	Science Fiction	2	4	2	1.0
Science Fiction	Classics	4	4	3	0.75

We can see that those genres with lower similarity have been filtered out. If we're implementing a k-Nearest Neighbors type query we might instead want to find the most similar k super genres for a given genre. We can do that by passing in the topK parameter.

The following will return a stream of genres, along with the two most similar super genres to them (i.e. k=2):

```
MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), categories: collect(id(book))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.overlap.stream({
  data: data,
  topK: 2
})
YIELD item1, item2, count1, count2, intersection, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to,
       count1, count2, intersection, similarity
ORDER BY from
```

Table 643. Results

from	to	count1	count2	intersection	similarity
Dystopia	Classics	2	4	2	1.0
Dystopia	Science Fiction	2	4	2	1.0
Fantasy	Science Fiction	3	4	3	1.0
Fantasy	Classics	3	4	2	0.6666666666666666
Science Fiction	Classics	4	4	3	0.75

## Write

The following will find the most similar genre for each genre, and store a relationship between those genres:

```
MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), categories: collect(id(book))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.overlap.write({
  data: data,
  topK: 2,
  similarityCutoff: 0.5
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50,
p75, p90, p95, p99, p999, p100
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

Table 644. Results

nodes	similarityPairs	writeRelationshipType	writeProperty	min	max	mean	p95
4	5	NARROWER_THAN	score	0.6666641235351562	1.0000038146972656	0.8833351135253906	1.0000038146972656

We then could write a query to find out the genre hierarchy for a specific genre.

The following will find the genre hierarchy for the Fantasy genre

```
MATCH path = (fantasy:Genre {name: "Fantasy"})-[:NARROWER_THAN*]->(genre)
RETURN [node in nodes(path) | node.name] AS hierarchy
ORDER BY length(path)
```

Table 645. Results

hierarchy
["Fantasy", "Science Fiction"]
["Fantasy", "Classics"]
["Fantasy", "Science Fiction", "Classics"]

## Stats

The following will run the algorithm and returns the result in form of statistical and measurement values

```
MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), categories: collect(id(book))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.similarity.overlap.stats({
  data: data,
  topK: 2,
  similarityCutoff: 0.5
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
```

## Specifying source and target ids

Sometimes, we don't want to compute all pairs similarity, but would rather specify subsets of items to compare to each other. We do this using the `sourceIds` and `targetIds` keys in the config.

We could use this technique to compute the similarity of a subset of items to all other items.

The following will return the super genres for the `Fantasy` and `Classics` genres:

```
MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), name: genre.name, categories: collect(id(book))} AS userData
WITH collect(userData) AS data
WITH data,
     [value in data WHERE value.name IN ["Fantasy", "Classics"] | value.item ] AS sourceIds
CALL gds.alpha.similarity.overlap.stream({
  data: data,
  sourceIds: sourceIds
})
YIELD item1, item2, count1, count2, intersection, similarity
RETURN gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY similarity DESC
```

Table 646. Results

from	to	similarity
Fantasy	Science Fiction	1.0
Classics	Science Fiction	0.75
Fantasy	Classics	0.6666666666666666

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.similarity.overlap.write(configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75,
p90, p95, p99, p999, p100
```

Table 647. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 648. Configuration

Name	Type	Default	Optional	Description
data	List or String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.

Name	Type	Default	Optional	Description
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
writeBatchSize	Integer	10000	yes	The batch size to use when storing results.
writeRelationshipType	String	SIMILAR	yes	The relationship type to use when storing results.
writeProperty	String	score	yes	The property to use when storing results.
sourceIds	List of String	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of String	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 649. Results

Name	Type	Description
nodes	Integer	The number of nodes passed in.
similarityPairs	Integer	The number of pairs of similar nodes computed.
writeRelationshipType	String	The relationship type used when storing results.
writeProperty	String	The property used when storing results.
min	Float	The minimum similarity score computed.
max	Float	The maximum similarity score computed.
mean	Float	The mean of similarities scores computed.
stdDev	Float	The standard deviation of similarities scores computed.
p25	Float	The 25 percentile of similarities scores computed.
p50	Float	The 50 percentile of similarities scores computed.
p75	Float	The 75 percentile of similarities scores computed.
p90	Float	The 90 percentile of similarities scores computed.

Name	Type	Description
p95	Float	The 95 percentile of similarities scores computed.
p99	Float	The 99 percentile of similarities scores computed.
p999	Float	The 99.9 percentile of similarities scores computed.
p100	Float	The 100 percentile of similarities scores computed.

The following will run the algorithm and stream results:

```
CALL gds.alpha.similarity.overlap.stream(configuration: Map)
YIELD item1, item2, count1, count2, similarity
```

Table 650. Parameters

Name	Type	Default	Optional	Description
configuration	Map	n/a	no	Algorithm-specific configuration.

Table 651. Configuration

Name	Type	Default	Optional	Description
data	List or String	null	no	A list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If 0, it will return as many as it finds.
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	<code>gds.util.NaN()</code>	yes	Value to skip when executing similarity computation. A value of <code>null</code> means that skipping is disabled.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
sourceIds	List of Integer	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.
targetIds	List of Integer	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <code>data</code> parameter.

Table 652. Results

Name	Type	Description
item1	Integer	The ID of one node in the similarity pair.

Name	Type	Description
item2	Integer	The ID of other node in the similarity pair.
count1	Integer	The size of the <code>targets</code> list of one node.
count2	Integer	The size of the <code>targets</code> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <code>targets</code> lists.
similarity	Integer	The overlap similarity of the two nodes.

## 7.4.8. Approximate Nearest Neighbors (ANN) Alpha

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

The Approximate Nearest Neighbors algorithm constructs a k-Nearest Neighbors Graph for a set of objects based on a provided similarity algorithm. The similarity of items is computed based on [Jaccard Similarity](#), [Cosine Similarity](#), [Euclidean Distance](#), or [Pearson Similarity](#).

The implementation in the library is based on Dong, Charikar, and Li's paper [Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures](#).

### Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.ml.ann.write(configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, stdDev, p25, p50, p75, p90, p95, p99, p999, p100
```

Table 653. Configuration

Name	Type	Default	Optional	Description
algorithm	String	null	no	The similarity algorithm to use. Valid values: <code>jaccard</code> , <code>cosine</code> , <code>pearson</code> , <code>euclidean</code> .
data	List	null	no	If algorithm is <code>jaccard</code> , a list of maps of the following structure: <code>{item: nodeId, categories: [nodeId, nodeId, nodeId]}</code> . Otherwise a list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node.
randomSeed	Integer	n/a	yes	The random-seed used for neighbor-sampling.
sampling	Boolean	true	yes	Whether the potential neighbors should be sampled.
p	Float	0.5	yes	Influences the sample size: $\min(1.0, p) *  topK $ .
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.

Name	Type	Default	Optional	Description
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <b>targets</b> list. If the list contains less than this amount, that node will be excluded from the calculation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result.
writeBatchSize	Integer	10000	yes	The batch size to use when storing results.
writeRelationshipType	String	SIMILAR	yes	The relationship type to use when storing results.
writeProperty	String	score	yes	The property to use when storing results.

Table 654. Results

Name	Type	Description
nodes	Integer	The number of nodes passed in.
similarityPairs	Integer	The number of pairs of similar nodes computed.
writeRelationshipType	String	The relationship type used when storing results.
writeProperty	String	The property used when storing results.
min	Float	The minimum similarity score computed.
max	Float	The maximum similarity score computed.
mean	Float	The mean of similarities scores computed.
stdDev	Float	The standard deviation of similarities scores computed.
p25	Float	The 25 percentile of similarities scores computed.
p50	Float	The 50 percentile of similarities scores computed.
p75	Float	The 75 percentile of similarities scores computed.
p90	Float	The 90 percentile of similarities scores computed.
p95	Float	The 95 percentile of similarities scores computed.
p99	Float	The 99 percentile of similarities scores computed.
p999	Float	The 99.9 percentile of similarities scores computed.
p100	Float	The 25 percentile of similarities scores computed.

The following will run the algorithm and stream results:

```
CALL gds.alpha.ml.ann.stream(configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```



Table 655. Configuration

Name	Type	Default	Optional	Description
algorithm	String	null	no	The similarity algorithm to use. Valid values: 'jaccard', 'cosine', 'pearson', 'euclidean'
data	List	null	no	If algorithm is 'jaccard', a list of maps of the following structure: <code>{item: nodeId, categories: [nodeId, nodeId, nodeId]}</code> . Otherwise a list of maps of the following structure: <code>{item: nodeId, weights: [double, double, double]}</code> or a Cypher query.
top	Integer	0	yes	The number of similar pairs to return. If 0, it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node.
randomSeed	Integer	1	yes	The random-seed used for neighbor-sampling.
sampling	Boolean	true	yes	Whether the potential neighbors should be sampled.
p	Float	0.5	yes	Influences the sample size: $\min(1.0, p) *  \text{topK} $
similarityCutoff	Integer	-1	yes	The threshold for similarity. Values below this will not be returned.
degreeCutoff	Integer	0	yes	The threshold for the number of items in the <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 656. Results

Name	Type	Description
item1	Integer	The ID of one node in the similarity pair.
item2	Integer	The ID of other node in the similarity pair.
count1	Integer	The size of the <code>targets</code> list of one node.
count2	Integer	The size of the <code>targets</code> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <code>targets</code> lists.
similarity	Integer	The similarity of the two nodes.

## Use-cases - when to use the Approximate Nearest Neighbors algorithm

We can use the Approximate Nearest Neighbors algorithm to work out the approximate k most similar items to each other. The corresponding k-Nearest Neighbors Graph can then be used as part of recommendation queries.

## Approximate Nearest Neighbors algorithm sample

The following will create a sample graph:

```
CREATE
(french:Cuisine {name:'French'}),
(italian:Cuisine {name:'Italian'}),
(indian:Cuisine {name:'Indian'}),
(lebanese:Cuisine {name:'Lebanese'}),
(portuguese:Cuisine {name:'Portuguese'}),

(zhen:Person {name:'Zhen'}),
(praveena:Person {name:'Praveena'}),
(michael:Person {name:'Michael'}),
(arya:Person {name:'Arya'}),
(karin:Person {name:'Karin'}),

(praveena)-[:LIKES]->(indian),
(praveena)-[:LIKES]->(portuguese),

(zhen)-[:LIKES]->(french),
(zhen)-[:LIKES]->(indian),

(michael)-[:LIKES]->(french),
(michael)-[:LIKES]->(italian),
(michael)-[:LIKES]->(indian),

(arya)-[:LIKES]->(lebanese),
(arya)-[:LIKES]->(italian),
(arya)-[:LIKES]->(portuguese),

(karin)-[:LIKES]->(lebanese),
(karin)-[:LIKES]->(italian)
```

The following will return a stream of nodes, along with up to the 3 most similar nodes to them based on Jaccard Similarity:

```
MATCH (p:Person)-[:LIKES]->(cuisine)
WITH {item:id(p), categories: collect(id(cuisine))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.ml.ann.stream({
  data: data,
  algorithm: 'jaccard',
  similarityCutoff: 0.1,
  randomSeed: 1,
  concurrency: 1
})
YIELD item1, item2, similarity
return gds.util.asNode(item1).name AS from, gds.util.asNode(item2).name AS to, similarity
ORDER BY from
```

Table 657. Results

from	to	similarity
Arya	Karin	0.6666666666666666
Arya	Praveena	0.25
Arya	Michael	0.2
Karin	Arya	0.6666666666666666
Karin	Michael	0.25
Michael	Karin	0.25
Michael	Praveena	0.25
Michael	Arya	0.2

from	to	similarity
Praveena	Arya	0.25
Praveena	Michael	0.25
Zhen	Michael	0.6666666666666666

Arya and Karin, and Zhen and Michael have the most similar food preferences, with two overlapping cuisines for a similarity of 0.66. We also have 3 pairs of users who are not similar at all. We'd probably want to filter those out, which we can do by passing in the `similarityCutoff` parameter.

The following will find up to 3 similar users for each user, and store a relationship between those users:

```

MATCH (p:Person)-[:LIKES]->(cuisine)
WITH {item:id(p), categories: collect(id(cuisine))} AS userData
WITH collect(userData) AS data
CALL gds.alpha.ml.ann.write({
  algorithm: 'jaccard',
  data: data,
  similarityCutoff: 0.1,
  showComputations: true,
  randomSeed: 1,
  concurrency: 1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean, p95

```

Table 658. Results

nodes	similarityPairs	writeRelationshipType	writeProperty	min	max	mean	p95
5	13	"SIMILAR"	"score"	0.199999809 26513672	0.666666984 5581055	0.351282266 4701022	0.666666984 5581055

We then could write a query to find out what types of cuisine that other people similar to us might like.

The following will find the most similar user to Praveena, and return their favorite cuisines that Praveena doesn't (yet!) like:

```

MATCH (p:Person {name: 'Praveena'})-[:SIMILAR]->(other),
      (other)-[:LIKES]->(cuisine)
WHERE not((p)-[:LIKES]->(cuisine))
RETURN cuisine.name AS cuisine, count(*) AS count
ORDER BY cuisine DESC

```

Table 659. Results

cuisine	count
"French"	1
"Italian"	2
"Lebanese"	1

## Usage

When executing `ApproximateNearestNeighbors` in parallel, it is possible that results are flaky because of the asynchronous execution fashion of the algorithm.

## 7.5. Path finding

Path finding algorithms find the shortest path between two or more nodes or evaluate the availability and quality of paths. The Neo4j GDS library includes the following path finding algorithms, grouped by quality tier:

- Production-quality
  - [Dijkstra Source-Target](#)
  - [Dijkstra Single-Source](#)
  - [A\\*](#)
  - [Yen's algorithm](#)
- Beta
  - [Random Walk](#)
- Alpha
  - [Minimum Weight Spanning Tree](#)
  - [Single Source Shortest Path](#)
  - [All Pairs Shortest Path](#)
  - [Breadth First Search](#)
  - [Depth First Search](#)

### 7.5.1. Dijkstra Source-Target

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

## Introduction

The Dijkstra Shortest Path algorithm computes the shortest path between nodes. The algorithm supports weighted graphs with positive relationship weights. The Dijkstra Source-Target algorithm computes the shortest path between a source and a target node. To compute all paths from a source node to all reachable nodes, [Dijkstra Single-Source](#) can be used.

The GDS implementation is based on the [original description](#) and uses a binary heap as priority queue. The implementation is also used for the [A\\*](#) and [Yen's](#) algorithms. The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

## Syntax

This section covers the syntax used to execute the Dijkstra algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Dijkstra in stream mode on a named graph.

```
CALL gds.shortestPath.dijkstra.stream(
  graphName: String,
  configuration: Map
)
YIELD
  index: Integer,
  sourceNode: Integer,
  targetNode: Integer,
  totalCost: Float,
  nodeIds: List of Integer,
  costs: List of Float,
  path: Path
```

Table 660. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 661. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 662. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 663. Results

Name	Type	Description
index	Integer	0-based index of the found path.
sourceNode	Integer	Source node of the path.
targetNode	Integer	Target node of the path.
totalCost	Float	Total cost from source to target.
nodeIds	List of Integer	Node ids on the path in traversal order.

Name	Type	Description
costs	List of Float	Accumulated costs for each node on the path.
path	Path	The path represented as Cypher entity.



The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

Run Dijkstra in mutate mode on a named graph.

```
CALL gds.shortestPath.dijkstra.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  configuration: Map
```

Table 664. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 665. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 666. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.

Table 667. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
mutateMillis	Integer	Milliseconds for adding relationships to the in-memory graph.

Name	Type	Description
relationships Written	Integer	The number of relationships that were added.
configuratio n	Map	The configuration used for running the algorithm.

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

Run Dijkstra in write mode on a named graph.

```
CALL gds.shortestPath.dijkstra.write(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 668. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 669. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 670. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
writeNodeIds	Boolean	false	yes	If true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	If true, the written relationship has a costs list property.

Table 671. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
writeMillis	Integer	Milliseconds for writing relationships to Neo4j.
relationshipsWritten	Integer	The number of relationships that were written.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Dijkstra in write mode on an anonymous graph:

```
CALL gds.shortestPath.dijkstra.write(
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 672. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.

Name	Type	Default	Optional	Description
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	<code>null</code>	yes	The relationship properties to project during anonymous graph creation.
<code>concurrency</code>	Integer	<code>4</code>	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
<code>readConcurrency</code>	Integer	<code>value of 'concurrency'</code>	yes	The number of concurrent threads used for creating the graph.
<code>writeConcurrency</code>	Integer	<code>value of 'concurrency'</code>	yes	The number of concurrent threads used for writing the result to Neo4j.

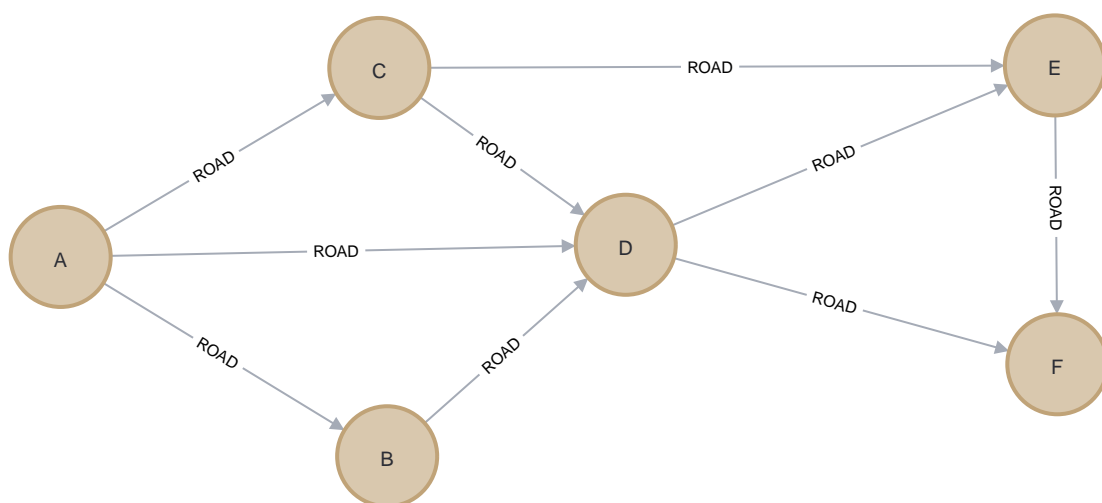
Table 673. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	<code>n/a</code>	no	The Neo4j node id of the source node.
targetNode	Integer	<code>n/a</code>	no	The Neo4j node id of the target node.
writeNodeIds	Boolean	<code>false</code>	yes	Iff true, the written relationship has a nodeIds list property.
writeCosts	Boolean	<code>false</code>	yes	Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

## Examples

In this section we will show examples of running the Dijkstra algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small transport network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE (a:Location {name: 'A'}),
      (b:Location {name: 'B'}),
      (c:Location {name: 'C'}),
      (d:Location {name: 'D'}),
      (e:Location {name: 'E'}),
      (f:Location {name: 'F'}),
      (a)-[:ROAD {cost: 50}]->(b),
      (a)-[:ROAD {cost: 50}]->(c),
      (a)-[:ROAD {cost: 100}]->(d),
      (b)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 80}]->(e),
      (d)-[:ROAD {cost: 30}]->(e),
      (d)-[:ROAD {cost: 80}]->(f),
      (e)-[:ROAD {cost: 40}]->(f);
```

This graph builds a transportation network with roads between locations. Like in the real world, the roads in the graph have different lengths. These lengths are represented by the `cost` relationship property.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Location',
  'ROAD',
  {
    relationshipProperties: 'cost'
  }
)
```

In the following example we will demonstrate the use of the Dijkstra Shortest Path algorithm using this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.dijkstra.write.estimate('myGraph', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'cost',
  writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 674. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	9	696	696	"696 Bytes"

## Stream

In the `stream` execution mode, the algorithm returns the shortest path for each source-target-pair. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index
```

Table 675. Results

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs	path
0	"A"	"F"	160.0	[A, B, D, E, F]	[0.0, 50.0, 90.0, 120.0, 160.0]	[Node[0], Node[1], Node[3], Node[4], Node[5]]

The result shows the total cost of the shortest path between node `A` and node `F`. It also shows an ordered list of node ids that were traversed to find the shortest path as well as the accumulated costs of the visited nodes. This can be verified in the [example graph](#). Cypher Path objects can be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property.

## Mutate

The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property.

The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.dijkstra.mutate('myGraph', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'cost',
  mutateRelationshipType: 'PATH'
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 676. Results

relationshipsWritten
1

After executing the above query, the in-memory graph will be updated with a new relationship of type `PATH`. The new relationship will store a single property `totalCost`.

## Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.dijkstra.write('myGraph', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'cost',
  writeRelationshipType: 'PATH',
  writeNodeIds: true,
  writeCosts: true
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 677. Results



```
relationshipsWritten
```

```
1
```

The above query will write a single relationship of type `PATH` back to Neo4j. The relationship stores three properties describing the path: `totalCost`, `nodeIds` and `costs`.

## 7.5.2. Dijkstra Single-Source

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

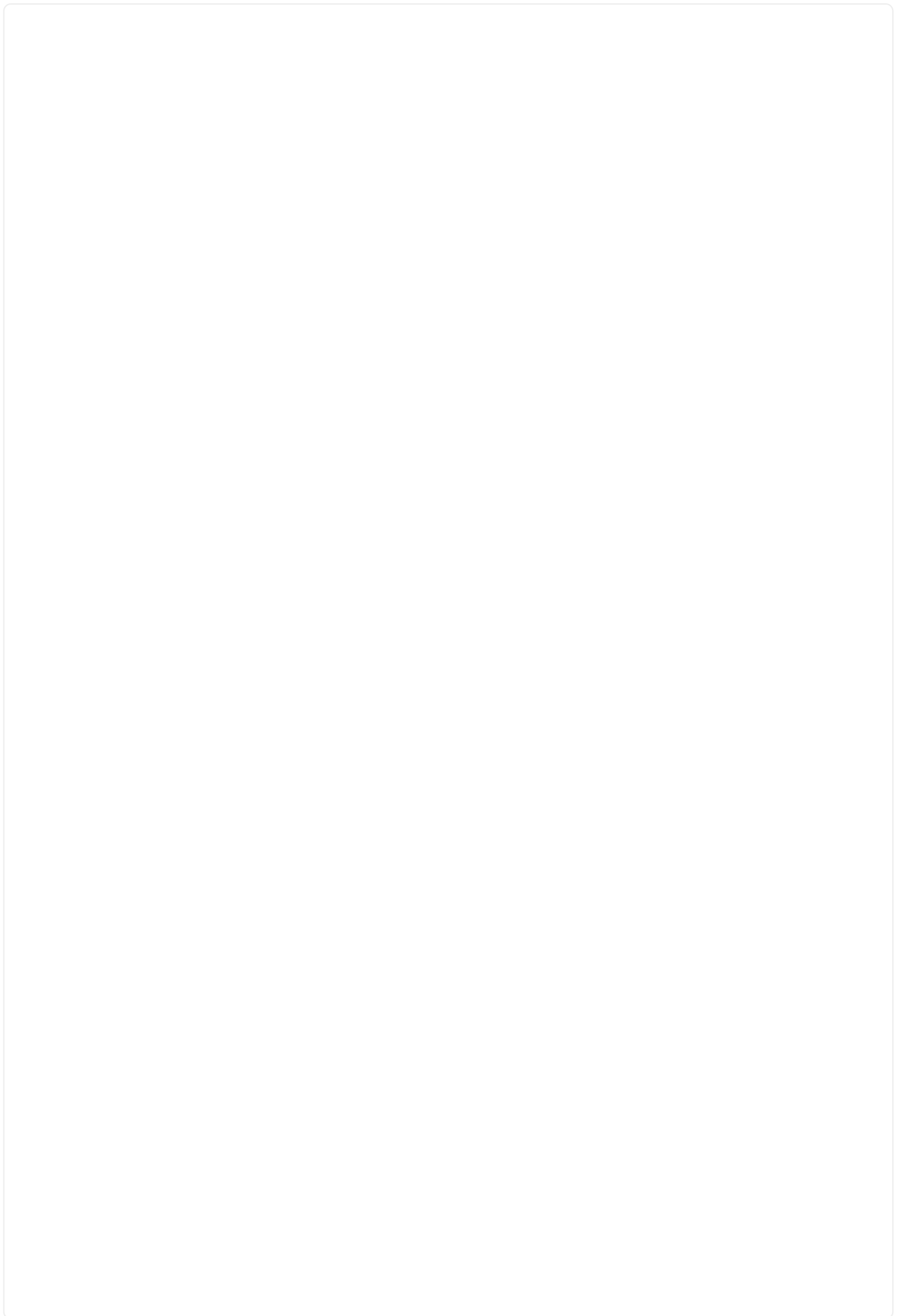
### Introduction

The Dijkstra Shortest Path algorithm computes the shortest path between nodes. The algorithm supports weighted graphs with positive relationship weights. The Dijkstra Single-Source algorithm computes the shortest paths between a source node and all nodes reachable from that node. To compute the shortest path between a source and a target node, [Dijkstra Source-Target](#) can be used.

The GDS implementation is based on the [original description](#) and uses a binary heap as priority queue. The implementation is also used for the [A\\*](#) and [Yen's](#) algorithms. The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

### Syntax

This section covers the syntax used to execute the Dijkstra algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Dijkstra in stream mode on a named graph.

```
CALL gds.allShortestPaths.dijkstra.stream(
  graphName: String,
  configuration: Map
)
YIELD
  index: Integer,
  sourceNode: Integer,
  targetNode: Integer,
  totalCost: Float,
  nodeIds: List of Integer,
  costs: List of Float,
  path: Path
```

Table 678. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 679. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 680. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 681. Results

Name	Type	Description
index	Integer	0-based index of the found path.
sourceNode	Integer	Source node of the path.
targetNode	Integer	Target node of the path.
totalCost	Float	Total cost from source to target.
nodeIds	List of Integer	Node ids on the path in traversal order.

Name	Type	Description
costs	List of Float	Accumulated costs for each node on the path.
path	Path	The path represented as Cypher entity.

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

Run Dijkstra in mutate mode on a named graph.

```
CALL gds.allShortestPaths.dijkstra.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  configuration: Map
```

Table 682. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 683. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 684. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.

Table 685. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
mutateMillis	Integer	Milliseconds for adding relationships to the in-memory graph.

Name	Type	Description
relationships Written	Integer	The number of relationships that were added.
configuratio n	Map	The configuration used for running the algorithm.

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

Run Dijkstra in write mode on a named graph.

```
CALL gds.allShortestPaths.dijkstra.write(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 686. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 687. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 688. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
writeNodeIds	Boolean	false	yes	If true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	If true, the written relationship has a costs list property.

Table 689. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
writeMillis	Integer	Milliseconds for writing relationships to Neo4j.
relationshipsWritten	Integer	The number of relationships that were written.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Dijkstra in write mode on an anonymous graph:

```
CALL gds.allShortestPaths.dijkstra.write(
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 690. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.



Name	Type	Default	Optional	Description
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

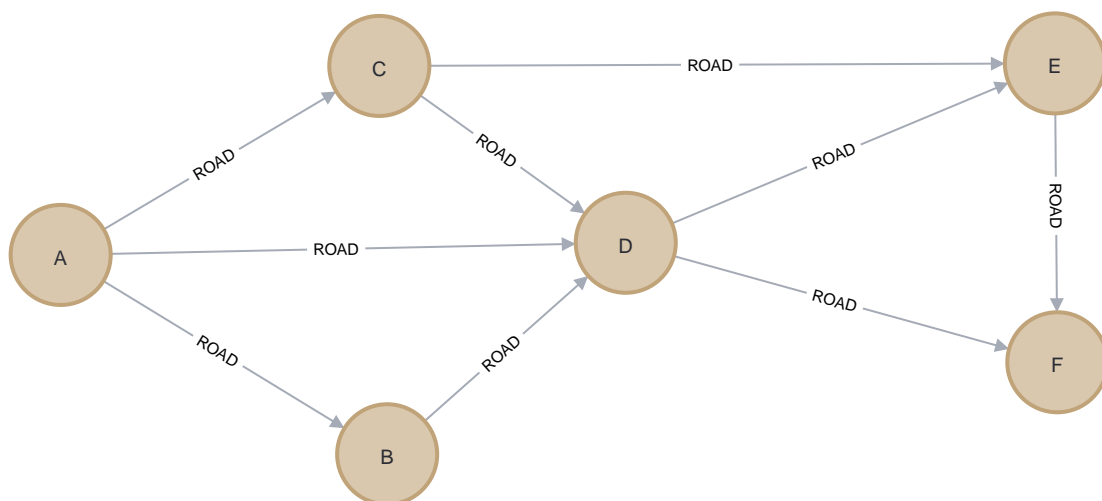
Table 691. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j node id of the source node.
targetNode	Integer	n/a	no	The Neo4j node id of the target node.
writeNodeIds	Boolean	false	yes	Iff true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

## Examples

In this section we will show examples of running the Dijkstra algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small transport network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE (a:Location {name: 'A'}),
      (b:Location {name: 'B'}),
      (c:Location {name: 'C'}),
      (d:Location {name: 'D'}),
      (e:Location {name: 'E'}),
      (f:Location {name: 'F'}),
      (a)-[:ROAD {cost: 50}]->(b),
      (a)-[:ROAD {cost: 50}]->(c),
      (a)-[:ROAD {cost: 100}]->(d),
      (b)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 80}]->(e),
      (d)-[:ROAD {cost: 30}]->(e),
      (d)-[:ROAD {cost: 80}]->(f),
      (e)-[:ROAD {cost: 40}]->(f);
```

This graph builds a transportation network with roads between locations. Like in the real world, the roads in the graph have different lengths. These lengths are represented by the `cost` relationship property.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Location',
  'ROAD',
  {
    relationshipProperties: 'cost'
  }
);
```

In the following example we will demonstrate the use of the Dijkstra Shortest Path algorithm using this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
MATCH (source:Location {name: 'A'})
CALL gds.allShortestPaths.dijkstra.write.estimate('myGraph', {
  sourceNode: source,
  relationshipWeightProperty: 'cost',
  writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 692. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	9	696	696	"696 Bytes"

### Stream

In the `stream` execution mode, the algorithm returns the shortest path for each source-target-pair. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
MATCH (source:Location {name: 'A'})
CALL gds.allShortestPaths.dijkstra.stream('myGraph', {
  sourceNode: source,
  relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index
```

Table 693. Results

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs	path
0	"A"	"A"	0.0	[A]	[0.0]	[Node[0]]
1	"A"	"B"	50.0	[A, B]	[0.0, 50.0]	[Node[0], Node[1]]
2	"A"	"C"	50.0	[A, C]	[0.0, 50.0]	[Node[0], Node[2]]
3	"A"	"D"	90.0	[A, B, D]	[0.0, 50.0, 90.0]	[Node[0], Node[1], Node[3]]

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs	path
4	"A"	"E"	120.0	[A, B, D, E]	[0.0, 50.0, 90.0, 120.0]	[Node[0], Node[1], Node[3], Node[4]]
5	"A"	"F"	160.0	[A, B, D, E, F]	[0.0, 50.0, 90.0, 120.0, 160.0]	[Node[0], Node[1], Node[3], Node[4], Node[5]]

The result shows the total cost of the shortest path between node **A** and all other reachable nodes in the graph. It also shows ordered lists of node ids that were traversed to find the shortest paths as well as the accumulated costs of the visited nodes. This can be verified in the [example graph](#). Cypher Path objects can be returned by the **path** return field. The Path objects contain the node objects and virtual relationships which have a **cost** property.

## Mutate

The **mutate** execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the **mutateRelationshipType** option. The total path cost is stored using the **totalCost** property.

The **mutate** mode is especially useful when multiple algorithms are used in conjunction.

For more details on the **mutate** mode in general, see [Mutate](#).

The following will run the algorithm in **mutate** mode:

```
MATCH (source:Location {name: 'A'})
CALL gds.allShortestPaths.dijkstra.mutate('myGraph', {
  sourceNode: source,
  relationshipWeightProperty: 'cost',
  mutateRelationshipType: 'PATH'
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 694. Results

relationshipsWritten
6

After executing the above query, the in-memory graph will be updated with new relationships of type **PATH**. The new relationships will store a single property **totalCost**.

## Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
MATCH (source:Location {name: 'A'})
CALL gds.allShortestPaths.dijkstra.write('myGraph', {
  sourceNode: source,
  relationshipWeightProperty: 'cost',
  writeRelationshipType: 'PATH',
  writeNodeIds: true,
  writeCosts: true
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 695. Results

relationshipsWritten
6

The above query will write 6 relationships of type `PATH` back to Neo4j. The relationships store three properties describing the path: `totalCost`, `nodeIds` and `costs`.

### 7.5.3. A\*

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

## Introduction

The A\* (pronounced "A-Star") Shortest Path algorithm computes the shortest path between two nodes. A\* is an informed search algorithm as it uses a heuristic function to guide the graph traversal. The algorithm supports weighted graphs with positive relationship weights.

Unlike [Dijkstra's shortest path algorithm](#), the next node to search from is not solely picked on the already computed distance. Instead, the algorithm combines the already computed distance with the result of a heuristic function. That function takes a node as input and returns a value that corresponds to the cost to

reach the target node from that node. In each iteration, the graph traversal is continued from the node with the lowest combined cost.

In GDS, the A\* algorithm is based on the [Dijkstra's shortest path algorithm](#). The heuristic function is the haversine distance, which defines the distance between two points on a sphere. Here, the sphere is the earth and the points are geo-coordinates stored on the nodes in the graph.

The algorithm implementation is executed using a single thread. Altering the concurrency configuration has no effect.

## Requirements

In GDS, the heuristic function used to guide the search is the [haversine formula](#). The formula computes the distance between two points on a sphere given their longitudes and latitudes. The distance is computed in nautical miles.

In order to guarantee finding the optimal solution, i.e., the shortest path between two points, the heuristic must be admissible. To be admissible, the function must not overestimate the distance to the target, i.e., the lowest possible cost of a path must always be greater or equal to the heuristic.

This leads to a requirement on the relationship weights of the input graph. Relationship weights must represent the distance between two nodes and ideally scaled to nautical miles. Kilometers or miles also work, but the heuristic works best for nautical miles.

## Syntax

This section covers the syntax used to execute the A\* algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

A\* syntax per mode



Run A\* in stream mode on a named graph.

```
CALL gds.shortestPath.astar.stream(
  graphName: String,
  configuration: Map
)
YIELD
  index: Integer,
  sourceNode: Integer,
  targetNode: Integer,
  totalCost: Float,
  nodeIds: List of Integer,
  costs: List of Float,
  path: Path
```

Table 696. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 697. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 698. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 699. Results

Name	Type	Description
index	Integer	0-based index of the found path.
sourceNode	Integer	Source node of the path.
targetNode	Integer	Target node of the path.
totalCost	Float	Total cost from source to target.
nodeIds	List of Integer	Node ids on the path in traversal order.



Name	Type	Description
costs	List of Float	Accumulated costs for each node on the path.
path	Path	The path represented as Cypher entity.

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

Run A\* in mutate mode on a named graph.

```
CALL gds.shortestPath.aster.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  configuration: Map
```

Table 700. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 701. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 702. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.

Table 703. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
mutateMillis	Integer	Milliseconds for adding relationships to the in-memory graph.

Name	Type	Description
relationships Written	Integer	The number of relationships that were added.
configuratio n	Map	The configuration used for running the algorithm.

The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

Run A\* in write mode on a named graph.

```
CALL gds.shortestPath.astar.write(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 704. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 705. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 706. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
writeNodeIds	Boolean	false	yes	If true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	If true, the written relationship has a costs list property.

Table 707. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
writeMillis	Integer	Milliseconds for writing relationships to Neo4j.
relationshipsWritten	Integer	The number of relationships that were written.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run A\* in write mode on an anonymous graph:

```
CALL gds.shortestPath.aster.write(
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 708. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.

Name	Type	Default	Optional	Description
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

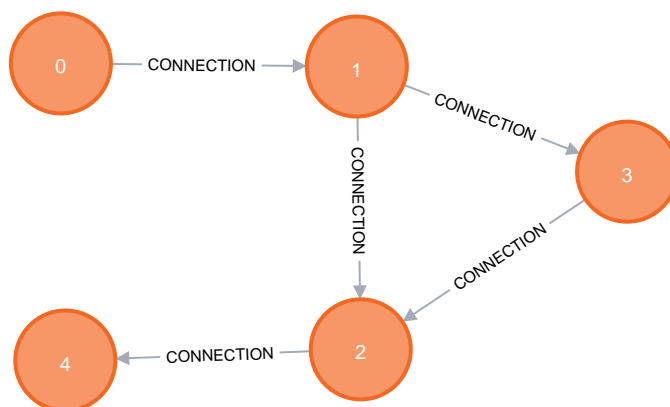
Table 709. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j node id of the source node.
targetNode	Integer	n/a	no	The Neo4j node id of the target node.
writeNodeIds	Boolean	false	yes	Iff true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

## Examples

In this section we will show examples of running the A\* algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small transport network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE (a:Station {name: 'Kings Cross', latitude: 51.5308, longitude: -0.1238}),
      (b:Station {name: 'Euston', latitude: 51.5282, longitude: -0.1337}),
      (c:Station {name: 'Camden Town', latitude: 51.5392, longitude: -0.1426}),
      (d:Station {name: 'Mornington Crescent', latitude: 51.5342, longitude: -0.1387}),
      (e:Station {name: 'Kentish Town', latitude: 51.5507, longitude: -0.1402}),
      (a)-[:CONNECTION {distance: 0.7}]->(b),
      (b)-[:CONNECTION {distance: 1.3}]->(c),
      (b)-[:CONNECTION {distance: 0.7}]->(d),
      (d)-[:CONNECTION {distance: 0.6}]->(c),
      (c)-[:CONNECTION {distance: 1.3}]->(e)
```

The graph represents a transport network of stations. Each station has a geo-coordinate, expressed by `latitude` and `longitude` properties. Stations are connected via connections. We use the `distance` property as relationship weight which represents the distance between stations in kilometers. The algorithm will pick the next node in the search based on the already traveled distance and the distance to the target station.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Station',
  'CONNECTION',
  {
    nodeProperties: ['latitude', 'longitude'],
    relationshipProperties: 'distance'
  }
)
```

In the following example we will demonstrate the use of the A\* Shortest Path algorithm using this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.astar.write.estimate('myGraph', {
  sourceNode: source,
  targetNode: target,
  latitudeProperty: 'latitude',
  longitudeProperty: 'longitude',
  writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 710. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
5	5	984	984	"984 Bytes"

### Stream

In the `stream` execution mode, the algorithm returns the shortest path for each source-target-pair. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.astar.stream('myGraph', {
  sourceNode: source,
  targetNode: target,
  latitudeProperty: 'latitude',
  longitudeProperty: 'longitude',
  relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index
```

Table 711. Results

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs	path
0	"Kings Cross"	"Kentish Town"	3.3	[Kings Cross, Euston, Camden Town, Kentish Town]	[0.0, 0.7, 2.0, 3.3]	[Node[0], Node[1], Node[2], Node[4]]

The result shows the total cost of the shortest path between node `King's Cross` and `Kentish Town` in the graph. It also shows ordered lists of node ids that were traversed to find the shortest paths as well as the accumulated costs of the visited nodes. This can be verified in the [example graph](#). Cypher Path objects can



be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property.

## Mutate

The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property.

The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.astar.mutate('myGraph', {
  sourceNode: source,
  targetNode: target,
  latitudeProperty: 'latitude',
  longitudeProperty: 'longitude',
  relationshipWeightProperty: 'distance',
  mutateRelationshipType: 'PATH'
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 712. Results

relationshipsWritten
1

After executing the above query, the in-memory graph will be updated with new relationships of type `PATH`. The new relationships will store a single property `totalCost`.

## Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.astar.write('myGraph', {
  sourceNode: source,
  targetNode: target,
  latitudeProperty: 'latitude',
  longitudeProperty: 'longitude',
  relationshipWeightProperty: 'distance',
  writeRelationshipType: 'PATH',
  writeNodeIds: true,
  writeCosts: true
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 713. Results

relationshipsWritten
1

The above query will write one relationship of type `PATH` back to Neo4j. The relationship stores three properties describing the path: `totalCost`, `nodeIds` and `costs`.

## 7.5.4. Yen's algorithm

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

Yen's Shortest Path algorithm computes a number of shortest paths between two nodes. The algorithm is often referred to as Yen's k-Shortest Path algorithm, where  $k$  is the number of shortest paths to compute. The algorithm supports weighted graphs with positive relationship weights. It also respects parallel relationships between the same two nodes when computing multiple shortest paths.

For  $k = 1$ , the algorithm behaves exactly like [Dijkstra's shortest path algorithm](#) and returns the shortest path. For  $k = 2$ , the algorithm returns the shortest path and the second shortest path between the same source and target node. Generally, for  $k = n$ , the algorithm computes at most  $n$  paths which are discovered in the order of their total cost.

The GDS implementation is based on the [original description](#). For the actual path computation, Yen's algorithm uses [Dijkstra's shortest path algorithm](#). The algorithm makes sure that an already discovered shortest path will not be traversed again.

The algorithm implementation is executed using a single thread. Altering the concurrency configuration

has no effect.

## Syntax

This section covers the syntax used to execute the Yen's algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Yen's in stream mode on a named graph.

```
CALL gds.shortestPath.yens.stream(
  graphName: String,
  configuration: Map
)
YIELD
  index: Integer,
  sourceNode: Integer,
  targetNode: Integer,
  totalCost: Float,
  nodeIds: List of Integer,
  costs: List of Float,
  path: Path
```

Table 714. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 715. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 716. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

Table 717. Results

Name	Type	Description
index	Integer	0-based index of the found path.
sourceNode	Integer	Source node of the path.
targetNode	Integer	Target node of the path.
totalCost	Float	Total cost from source to target.
nodeIds	List of Integer	Node ids on the path in traversal order.

Name	Type	Description
costs	List of Float	Accumulated costs for each node on the path.
path	Path	The path represented as Cypher entity.

The mutate mode creates new relationships in the in-memory graph. Each relationship represents a path from the source node to the target node. The total cost of a path is stored via the `totalCost` relationship property.

Run Yen's in mutate mode on a named graph.

```
CALL gds.shortestPath.yens.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  configuration: Map
```

Table 718. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 719. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 720. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.

Table 721. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
mutateMillis	Integer	Milliseconds for adding relationships to the in-memory graph.

Name	Type	Description
relationships Written	Integer	The number of relationships that were added.
configuratio n	Map	The configuration used for running the algorithm.



The write mode creates new relationships in the Neo4j database. Each relationship represents a path from the source node to the target node. Additional path information is stored using relationship properties. By default, the write mode stores a `totalCost` property. Optionally, one can also store `nodeIds` and `costs` of intermediate nodes on the path.

Run Yen's in write mode on a named graph.

```
CALL gds.shortestPath.yens.write(
  graphName: String,
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 722. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 723. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 724. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j source node or node id.
writeNodeIds	Boolean	false	yes	If true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	If true, the written relationship has a costs list property.

Table 725. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Unused.
writeMillis	Integer	Milliseconds for writing relationships to Neo4j.
relationshipsWritten	Integer	The number of relationships that were written.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run Yen's in write mode on an anonymous graph:

```
CALL gds.shortestPath.yens.write(
  configuration: Map
)
YIELD
  relationshipsWritten: Integer,
  ranIterations: Integer,
  didConverge: Boolean,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 726. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.

Name	Type	Default	Optional	Description
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

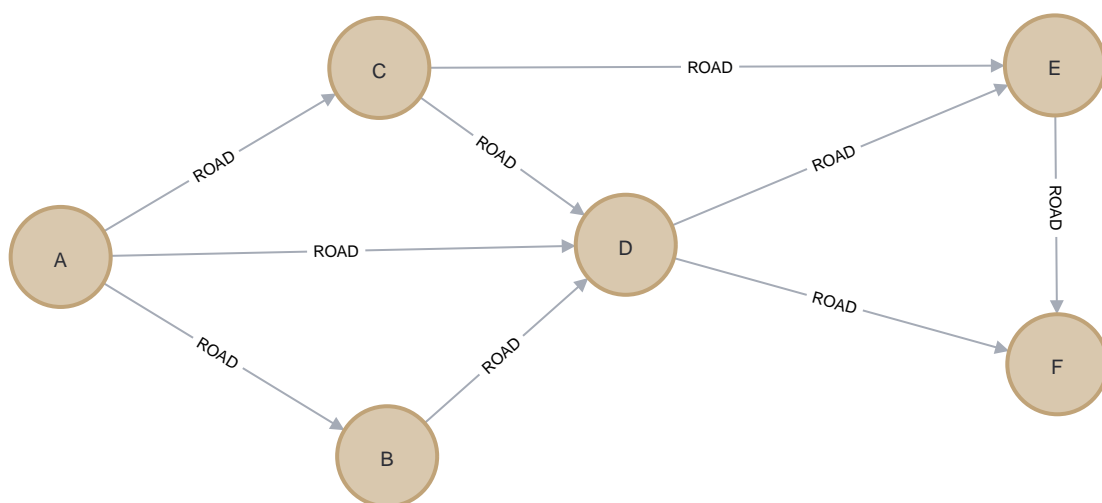
Table 727. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNode	Integer	n/a	no	The Neo4j node id of the source node.
targetNode	Integer	n/a	no	The Neo4j node id of the target node.
writeNodeIds	Boolean	false	yes	Iff true, the written relationship has a nodeIds list property.
writeCosts	Boolean	false	yes	Iff true, the written relationship has a costs list property.

The results are the same as for running write mode with a named graph, see the write mode syntax above.

## Examples

In this section we will show examples of running the Yen's algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small transport network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE (a:Location {name: 'A'}),
      (b:Location {name: 'B'}),
      (c:Location {name: 'C'}),
      (d:Location {name: 'D'}),
      (e:Location {name: 'E'}),
      (f:Location {name: 'F'}),
      (a)-[:ROAD {cost: 50}]->(b),
      (a)-[:ROAD {cost: 50}]->(c),
      (a)-[:ROAD {cost: 100}]->(d),
      (b)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 40}]->(d),
      (c)-[:ROAD {cost: 80}]->(e),
      (d)-[:ROAD {cost: 30}]->(e),
      (d)-[:ROAD {cost: 80}]->(f),
      (e)-[:ROAD {cost: 40}]->(f);
```

This graph builds a transportation network with roads between locations. Like in the real world, the roads in the graph have different lengths. These lengths are represented by the `cost` relationship property.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Location',
  'ROAD',
  {
    relationshipProperties: 'cost'
  }
)
```

In the following example we will demonstrate the use of the Yen's Shortest Path algorithm using this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `write` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.yens.write.estimate('myGraph', {
  sourceNode: source,
  targetNode: target,
  k: 3,
  relationshipWeightProperty: 'cost',
  writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 728. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	9	1008	1008	"1008 Bytes"

### Stream

In the `stream` execution mode, the algorithm returns the shortest path for each source-target-pair. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm and stream results:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.yens.stream('myGraph', {
  sourceNode: source,
  targetNode: target,
  k: 3,
  relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  nodes(path) as path
ORDER BY index
```

Table 729. Results

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs	path
0	"A"	"F"	160.0	[A, B, D, E, F]	[0.0, 50.0, 90.0, 120.0, 160.0]	[Node[0], Node[1], Node[3], Node[4], Node[5]]

index	sourceNodeName	targetNodeName	totalCost	nodeNames	costs	path
1	"A"	"F"	160.0	[A, C, D, E, F]	[0.0, 50.0, 90.0, 120.0, 160.0]	[Node[0], Node[2], Node[3], Node[4], Node[5]]
2	"A"	"F"	170.0	[A, B, D, F]	[0.0, 50.0, 90.0, 170.0]	[Node[0], Node[1], Node[3], Node[5]]

The result shows the three shortest paths between node **A** and node **F**. The first two paths have the same total cost, however the first one traversed from **A** to **D** via the **B** node, while the second traversed via the **C** node. The third path has a higher total cost as it goes directly from **D** to **F** using the relationship with a cost of **80**, whereas the detour via **E** for the first two paths costs **70**. This can be verified in the [example graph](#). Cypher Path objects can be returned by the `path` return field. The Path objects contain the node objects and virtual relationships which have a `cost` property.

## Mutate

The `mutate` execution mode updates the named graph with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `mutateRelationshipType` option. The total path cost is stored using the `totalCost` property.

The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.yens.mutate('myGraph', {
  sourceNode: source,
  targetNode: target,
  k: 3,
  relationshipWeightProperty: 'cost',
  mutateRelationshipType: 'PATH'
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 730. Results

relationshipsWritten
3

After executing the above query, the in-memory graph will be updated with a new relationship of type `PATH`. The new relationship will store a single property `totalCost`.

## Write

The `write` execution mode updates the Neo4j database with new relationships. Each new relationship represents a path from source node to target node. The relationship type is configured using the `writeRelationshipType` option. The total path cost is stored using the `totalCost` property. The intermediate node ids are stored using the `nodeIds` property. The accumulated costs to reach an intermediate node are stored using the `costs` property.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
MATCH (source:Location {name: 'A'}), (target:Location {name: 'F'})
CALL gds.shortestPath.yens.write('myGraph', {
  sourceNode: source,
  targetNode: target,
  k: 3,
  relationshipWeightProperty: 'cost',
  writeRelationshipType: 'PATH',
  writeNodeIds: true,
  writeCosts: true
})
YIELD relationshipsWritten
RETURN relationshipsWritten
```

Table 731. Results

relationshipsWritten
3

The above query will write a single relationship of type `PATH` back to Neo4j. The relationship stores three properties describing the path: `totalCost`, `nodeIds` and `costs`.

## 7.5.5. Minimum Weight Spanning Tree Alpha

The Minimum Weight Spanning Tree (MST) starts from a given node, and finds all its reachable nodes and the set of relationships that connect the nodes together with the minimum possible weight. Prim's algorithm is one of the simplest and best-known minimum spanning tree algorithms. The K-Means variant of this algorithm can be used to detect clusters in the graph.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

The first known algorithm for finding a minimum spanning tree was developed by the Czech scientist Otakar Borůvka in 1926, while trying to find an efficient electricity network for Moravia. Prim's algorithm was invented by Jarník in 1930 and rediscovered by Prim in 1957. It is similar to Dijkstra's shortest path algorithm but, rather than minimizing the total length of a path ending at each relationship, it minimizes the length of each relationship individually. Unlike Dijkstra's, Prim's can tolerate negative-weight relationships.

The algorithm operates as follows:

- Start with a tree containing only one node (and no relationships).
- Select the minimal-weight relationship coming from that node, and add it to our tree.
- Repeatedly choose a minimal-weight relationship that joins any node in the tree to one that is not in the tree, adding the new relationship and node to our tree.
- When there are no more nodes to add, the tree we have built is a minimum spanning tree.

## Use-cases - when to use the Minimum Weight Spanning Tree algorithm

- Minimum spanning tree was applied to analyze airline and sea connections of Papua New Guinea, and minimize the travel cost of exploring the country. It could be used to help design low-cost tours that visit many destinations across the country. The research mentioned can be found in ["An Application of Minimum Spanning Trees to Travel Planning"](#).
- Minimum spanning tree has been used to analyze and visualize correlations in a network of currencies, based on the correlation between currency returns. This is described in ["Minimum Spanning Tree Application in the Currency Market"](#).
- Minimum spanning tree has been shown to be a useful tool to trace the history of transmission of infection, in an outbreak supported by exhaustive clinical research. For more information, see [Use of the Minimum Spanning Tree Model for Molecular Epidemiological Investigation of a Nosocomial Outbreak of Hepatitis C Virus Infection](#).

## Constraints - when not to use the Minimum Weight Spanning Tree algorithm

The MST algorithm only gives meaningful results when run on a graph, where the relationships have different weights. If the graph has no weights, or all relationships have the same weight, then any spanning tree is a minimum spanning tree.

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.spanningTree.write(configuration: Map)
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
```

The following will compute the minimum weight spanning tree and write the results:

```
CALL gds.alpha.spanningTree.minimum.write(configuration: Map)
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
```

The following will compute the maximum weight spanning tree and write the results:

```
CALL gds.alpha.spanningTree.maximum.write(configuration: Map)
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
```

Table 732. Configuration

Name	Type	Default	Optional	Description
startNodeID	Integer	null	no	The start node ID



Name	Type	Default	Optional	Description
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
writeProperty	String	'mst'	yes	The relationship type written back as result
weightWriteProperty	String	n/a	no	The weight property of the <a href="#">writeProperty</a> relationship type written back

Table 733. Results

Name	Type	Description
effectiveNodeCount	Integer	The number of visited nodes
createMillis	Integer	Milliseconds for loading data
computeMillis	Integer	Milliseconds for running the algorithm
writeMillis	Integer	Milliseconds for writing result data back

The following will run the  $k$ -spanning tree algorithms and write back results:

```
CALL gds.alpha.spanningTree.kmin.write(configuration: Map)
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
```

```
CALL gds.alpha.spanningTree.kmax.write(configuration: Map)
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
```

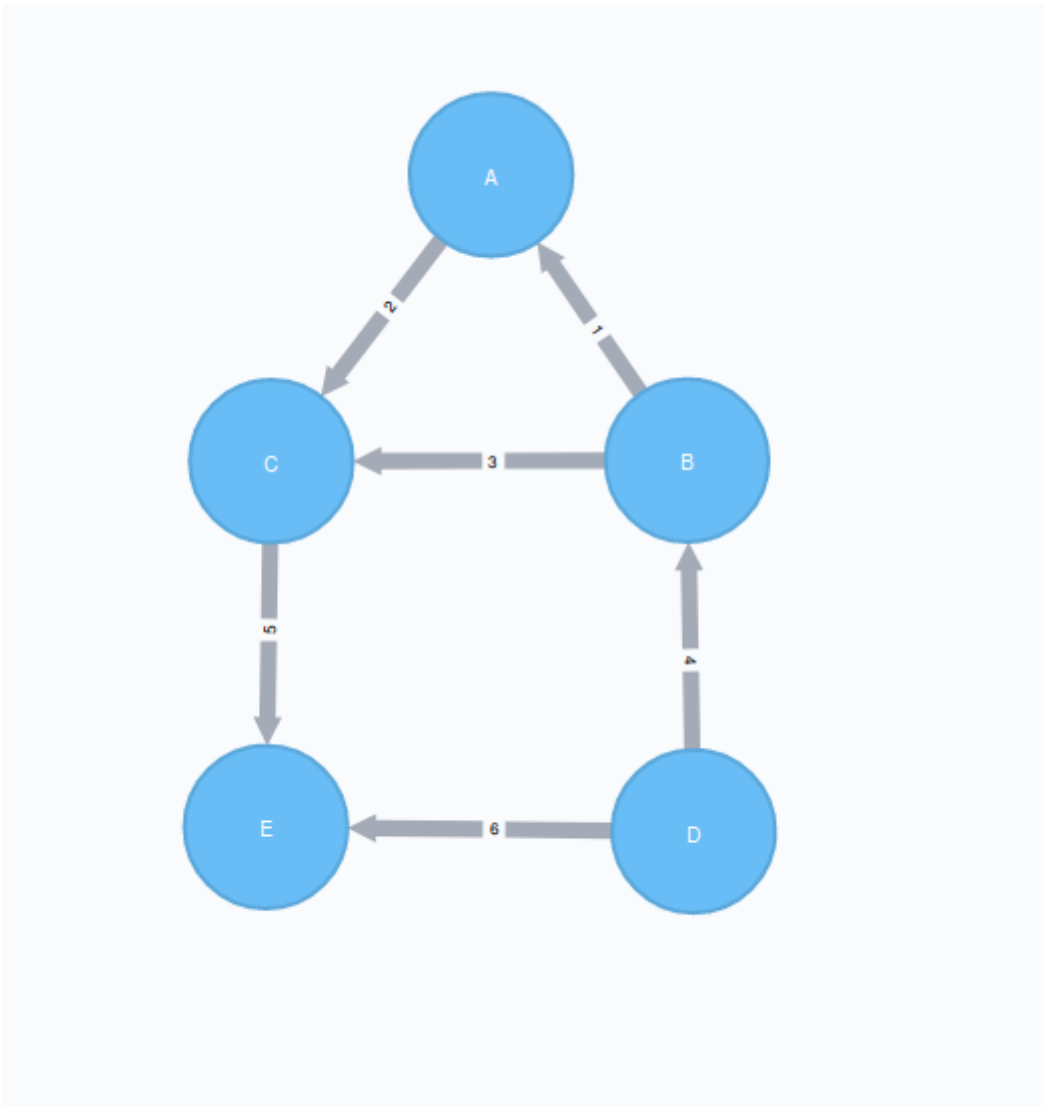
Table 734. Configuration

Name	Type	Default	Optional	Description
k	Integer	null	no	The result is a tree with $k$ nodes and $k - 1$ relationships
startNodeid	Integer	null	no	The start node ID
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
writeProperty	String	'MST'	yes	The relationship type written back as result
weightWriteProperty	String	n/a	no	The weight property of the <a href="#">writeProperty</a> relationship type written back

Table 735. Results

Name	Type	Description
effectiveNodeCount	Integer	The number of visited nodes
createMillis	Integer	Milliseconds for loading data
computeMillis	Integer	Milliseconds for running the algorithm
writeMillis	Integer	Milliseconds for writing result data back

## Minimum Weight Spanning Tree algorithm sample



The following will create a sample graph:

```
CREATE (a:Place {id: 'A'}),
       (b:Place {id: 'B'}),
       (c:Place {id: 'C'}),
       (d:Place {id: 'D'}),
       (e:Place {id: 'E'}),
       (f:Place {id: 'F'}),
       (g:Place {id: 'G'}),
       (d)-[:LINK {cost:4}]->(b),
       (d)-[:LINK {cost:6}]->(e),
       (b)-[:LINK {cost:1}]->(a),
       (b)-[:LINK {cost:3}]->(c),
       (a)-[:LINK {cost:2}]->(c),
       (c)-[:LINK {cost:5}]->(e),
       (f)-[:LINK {cost:1}]->(g);
```

Minimum weight spanning tree visits all nodes that are in the same connected component as the starting node, and returns a spanning tree of all nodes in the component where the total weight of the relationships is minimized.

The following will run the Minimum Weight Spanning Tree algorithm and write back results:

```
MATCH (n:Place {id: 'D'})
CALL gds.alpha.spanningTree.minimum.write({
  nodeProjection: 'Place',
  relationshipProjection: {
    LINK: {
      type: 'LINK',
      properties: 'cost',
      orientation: 'UNDIRECTED'
    }
  },
  startNodeId: id(n),
  relationshipWeightProperty: 'cost',
  writeProperty: 'MINST',
  weightWriteProperty: 'writeCost'
})
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

To find all pairs of nodes included in our minimum spanning tree, run the following query:

```
MATCH path = (n:Place {id: 'D'})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source, endNode(rel).id AS destination, rel.writeCost AS cost
```

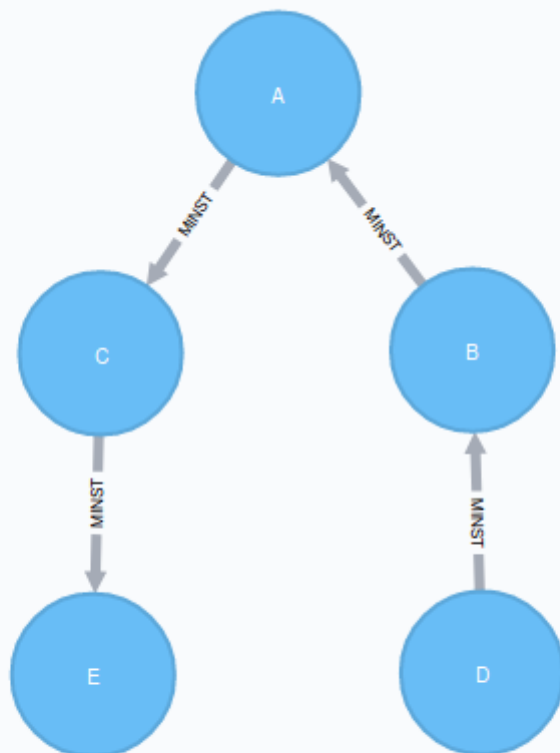


Figure 2. Results

Table 736. Results

Source	Destination	Cost
D	B	4
B	A	1
A	C	2
C	E	5

The minimum spanning tree excludes the relationship with cost 6 from D to E, and the one with cost 3 from B to C. Nodes F and G aren't included because they're unreachable from D.

Maximum weighted tree spanning algorithm is similar to the minimum one, except that it returns a spanning tree of all nodes in the component where the total weight of the relationships is maximized.

The following will run the maximum weight spanning tree algorithm and write back results:

```
MATCH (n:Place{id: 'D'})
CALL gds.alpha.spanningTree.maximum.write({
  nodeProjection: 'Place',
  relationshipProjection: {
    LINK: {
      type: 'LINK',
      properties: 'cost'
    }
  },
  startNodeId: id(n),
  relationshipWeightProperty: 'cost',
  writeProperty: 'MAXST',
  weightWriteProperty: 'writeCost'
})
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

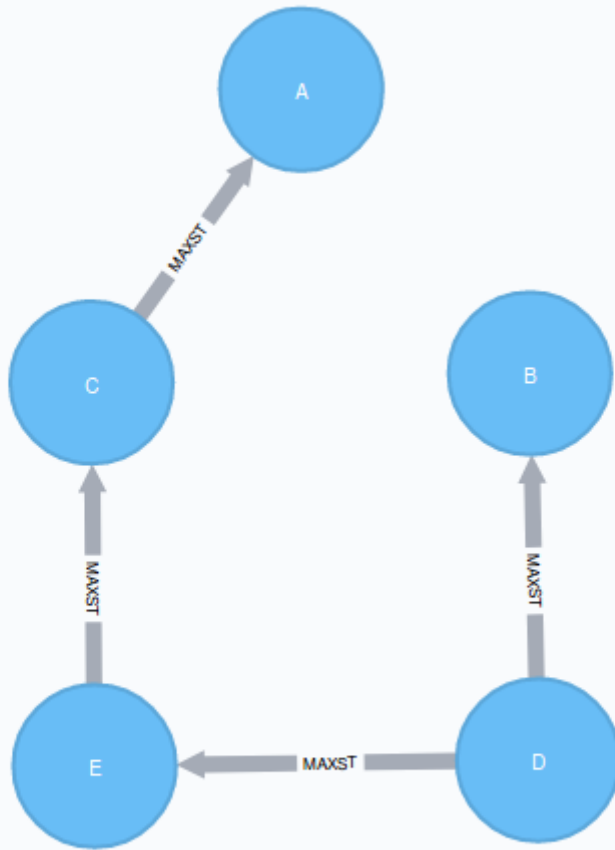


Figure 3. Results

### K-Spanning tree

Sometimes we want to limit the size of our spanning tree result, as we are only interested in finding a smaller tree within our graph that does not span across all nodes. K-Spanning tree algorithm returns a tree with  $k$  nodes and  $k - 1$  relationships.

In our sample graph we have 5 nodes. When we ran MST above, we got a 5-minimum spanning tree returned, that covered all five nodes. By setting the  $k=3$ , we define that we want to get returned a 3-minimum spanning tree that covers 3 nodes and has 2 relationships.

The following will run the k-minimum spanning tree algorithm and write back results:

```
MATCH (n:Place{id: 'D'})
CALL gds.alpha.spanningTree.kmin.write({
  nodeProjection: 'Place',
  relationshipProjection: {
    LINK: {
      type: 'LINK',
      properties: 'cost'
    }
  },
  k: 3,
  startNodeId: id(n),
  relationshipWeightProperty: 'cost',
  writeProperty: 'kminst'
})
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

Find nodes that belong to our k-spanning tree result:

```
MATCH (n:Place)
WITH n.id AS Place, n.kminst AS Partition, count(*) AS count
WHERE count = 3
RETURN Place, Partition
```

Table 737. Results

Place	Partition
A	1
B	1
C	1
D	3
E	4

Nodes A, B, and C are the result 3-minimum spanning tree of our graph.

The following will run the k-maximum spanning tree algorithm and write back results:

```
MATCH (n:Place{id: 'D'})
CALL gds.alpha.spanningTree.kmax.write({
  nodeProjection: 'Place',
  relationshipProjection: {
    LINK: {
      type: 'LINK',
      properties: 'cost'
    }
  },
  k: 3,
  startNodeId: id(n),
  relationshipWeightProperty: 'cost',
  writeProperty: 'kmaxst'
})
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

Find nodes that belong to our  $k$ -spanning tree result:

```
MATCH (n:Place)
WITH n.id AS Place, n.kmaxst AS Partition, count(*) AS count
WHERE count = 3
RETURN Place, Partition
```

Table 738. Results

Place	Partition
A	0
B	1
C	3
D	3
E	3

Nodes C, D, and E are the result 3-maximum spanning tree of our graph.

## 7.5.6. Single Source Shortest Path Alpha

The Single Source Shortest Path (SSSP) algorithm calculates the shortest (weighted) path from a node to all other nodes in the graph.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

SSSP came into prominence at the same time as the shortest path algorithm and Dijkstra's algorithm can act as an implementation for both problems.

We implement a delta-stepping algorithm that has been [shown to outperform Dijkstra's](#).

### Use-cases - when to use the Single Source Shortest Path algorithm

- [Open Shortest Path First](#) is a routing protocol for IP networks. It uses Dijkstra's algorithm to help detect changes in topology, such as link failures, and [come up with a new routing structure in seconds](#).

### Constraints - when not to use the Single Source Shortest Path algorithm

Delta stepping does not support negative weights. The algorithm assumes that adding a relationship to a path can never make a path shorter - an invariant that would be violated with negative weights.

### Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.shortestPath.deltaStepping.write(configuration: Map)
YIELD nodeCount, loadDuration, evalDuration, writeDuration
```

Table 739. Configuration

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
delta	Float	null	yes	The grade of concurrency to use.
writeProperty	String	'sssp'	yes	The property name written back to the node sequence of the node in the path. The property contains the cost it takes to get from the start node to the specific node.

Table 740. Results

Name	Type	Description
nodeCount	Integer	The number of nodes considered
loadDuration	Integer	Milliseconds for loading data
evalDuration	Integer	Milliseconds for running the algorithm
writeDuration	Integer	Milliseconds for writing result data back

The following will run the algorithm and stream results:

```
CALL gds.alpha.shortestPath.deltaStepping.stream(configuration: Map)
YIELD nodeId, distance
```

Table 741. Parameters

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node
delta	Float	null	no	The grade of concurrency to use.
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.

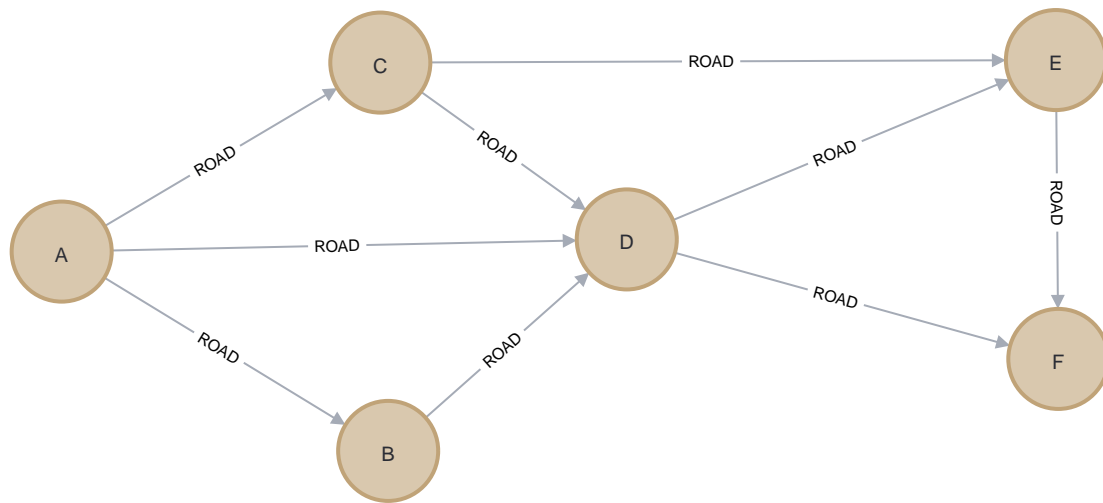
Table 742. Results

Name	Type	Description
nodeId	Integer	Node ID
distance	Integer	The cost it takes to get from the start node to the specific node.

## Single Source Shortest Path algorithm sample

In this section we will show examples of running the Single Source Shortest Path algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small transport network graph of a handful nodes connected in a particular pattern. The example graph looks like this:





The following will create a sample graph:

```

CREATE (a:Loc {name: "A"}),
       (b:Loc {name: "B"}),
       (c:Loc {name: "C"}),
       (d:Loc {name: "D"}),
       (e:Loc {name: "E"}),
       (f:Loc {name: "F"}),
       (a)-[:ROAD {cost: 50}]->(b),
       (a)-[:ROAD {cost: 50}]->(c),
       (a)-[:ROAD {cost: 100}]->(d),
       (b)-[:ROAD {cost: 40}]->(d),
       (c)-[:ROAD {cost: 40}]->(d),
       (c)-[:ROAD {cost: 80}]->(e),
       (d)-[:ROAD {cost: 30}]->(e),
       (d)-[:ROAD {cost: 80}]->(f),
       (e)-[:ROAD {cost: 40}]->(f);
  
```

## Delta stepping algorithm

The following will run the algorithm and stream results:

```

MATCH (n:Loc {name: 'A'})
CALL gds.alpha.shortestPath.deltaStepping.stream({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost'
    }
  },
  startNode: n,
  relationshipWeightProperty: 'cost',
  delta: 3.0
})
YIELD nodeId, distance
RETURN gds.util.asNode(nodeId).name AS Name, distance AS Cost
  
```

Table 743. Results

Name	Cost
"A"	0.0
"B"	50.0
"C"	50.0

Name	Cost
"D"	90.0
"E"	120.0
"F"	160.0

The above table shows the cost of going from A to each of the other nodes, including itself at a cost of 0.

The following will run the algorithm and write back results:

```
MATCH (n:Loc {name: 'A'})
CALL gds.alpha.shortestPath.deltaStepping.write({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost'
    }
  },
  startNode: n,
  relationshipWeightProperty: 'cost',
  delta: 3.0,
  writeProperty: 'sssp'
})
YIELD nodeCount
RETURN nodeCount
```

Table 744. Results

nodeCount
6

### Cypher projection

If node labels and relationship types are not selective enough to project a graph, you can use Cypher queries instead. Cypher projections can also be used to run algorithms on a virtual graph. You can learn more in the [Creating graphs using Cypher](#) section of the manual.

```
MATCH (start:Loc {name: 'A'})
CALL gds.alpha.shortestPath.deltaStepping.write({
  nodeQuery: 'MATCH(n:Loc) WHERE not n.name = "C" RETURN id(n) AS id',
  relationshipQuery: 'MATCH(n:Loc)-[r:ROAD]->(m:Loc) WHERE not (n.name = "C" OR m.name = "C") RETURN id(n) AS source, id(m) AS target, r.cost AS weight',
  startNode: start,
  relationshipWeightProperty: 'weight',
  delta: 3.0,
  writeProperty: 'sssp'
})
YIELD nodeCount
RETURN nodeCount
```

Table 745. Results

nodeCount
5

## 7.5.7. All Pairs Shortest Path Alpha

The All Pairs Shortest Path (APSP) calculates the shortest (weighted) path between all pairs of nodes. This algorithm has optimizations that make it quicker than calling the Single Source Shortest Path algorithm for every pair of nodes in the graph.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

Some pairs of nodes might not be reachable between each other, so no shortest path exists between these pairs. In this scenario, the algorithm will return **Infinity** value as a result between these pairs of nodes.

Plain cypher does not support filtering **Infinity** values, so `gds.util.isFinite` function was added to help filter **Infinity** values from results.

### Use-cases - when to use the All Pairs Shortest Path algorithm

- The All Pairs Shortest Path algorithm is used in urban service system problems, such as the location of urban facilities or the distribution or delivery of goods. One example of this is determining the traffic load expected on different segments of a transportation grid. For more information, see [Urban Operations Research](#).
- All pairs shortest path is used as part of the REWIRE data center design algorithm that finds a network with maximum bandwidth and minimal latency. There are more details about this approach in ["REWIRE: An Optimization-based Framework for Data Center Network Design"](#)

### Syntax

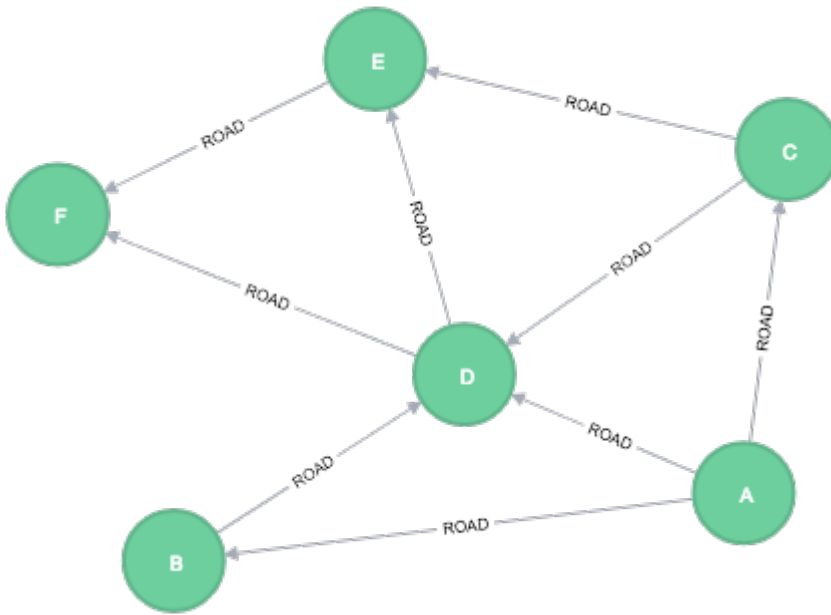
The following will run the algorithm and stream results:

```
CALL gds.alpha.allShortestPaths.stream(configuration: Map)
YIELD startNodeId, targetNodeId, distance
```

Table 746. Parameters

Name	Type	Default	Optional	Description
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'. This is dependent on the Neo4j edition; for more information, see <a href="#">CPU</a> .
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

## All Pairs Shortest Path algorithm sample



The following will create a sample graph:

```
CREATE (a:Loc {name: 'A'}),
       (b:Loc {name: 'B'}),
       (c:Loc {name: 'C'}),
       (d:Loc {name: 'D'}),
       (e:Loc {name: 'E'}),
       (f:Loc {name: 'F'}),
       (a)-[:ROAD {cost: 50}]->(b),
       (a)-[:ROAD {cost: 50}]->(c),
       (a)-[:ROAD {cost: 100}]->(d),
       (b)-[:ROAD {cost: 40}]->(d),
       (c)-[:ROAD {cost: 40}]->(d),
       (c)-[:ROAD {cost: 80}]->(e),
       (d)-[:ROAD {cost: 30}]->(e),
       (d)-[:ROAD {cost: 80}]->(f),
       (e)-[:ROAD {cost: 40}]->(f);
```

The following will run the algorithm and stream results:

```
CALL gds.alpha.allShortestPaths.stream({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost'
    }
  },
  relationshipWeightProperty: 'cost'
})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE gds.util.isFinite(distance) = true

MATCH (source:Loc) WHERE id(source) = sourceNodeId
MATCH (target:Loc) WHERE id(target) = targetNodeId
WITH source, target, distance WHERE source <> target

RETURN source.name AS source, target.name AS target, distance
ORDER BY distance DESC, source ASC, target ASC
LIMIT 10
```

Table 747. Results

Source	Target	Cost
A	F	160
A	E	120
B	F	110
C	F	110
A	D	90
B	E	70
C	E	70
D	F	70
A	B	50
A	C	50

This query returned the top 10 pairs of nodes that are the furthest away from each other. F and E appear to be quite distant from the others.

For now, only single-source shortest path support loading the relationship as undirected, but we can use Cypher loading to help us solve this. Undirected graph can be represented as [Bidirected graph](#), which is a directed graph in which the reverse of every relationship is also a relationship.

We do not have to save this reversed relationship, we can project it using Cypher loading. Note that relationship query does not specify direction of the relationship. This is applicable to all other algorithms that use Cypher loading.

The following will run the algorithm, treating the graph as undirected:

```
CALL gds.alpha.allShortestPaths.stream({
  nodeQuery: 'MATCH (n:Loc) RETURN id(n) AS id',
  relationshipQuery: 'MATCH (n:Loc)-[r:ROAD]-(p:Loc) RETURN id(n) AS source, id(p) AS target, r.cost AS cost',
  relationshipWeightProperty: 'cost'
})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE gds.util.isFinite(distance) = true

MATCH (source:Loc) WHERE id(source) = sourceNodeId
MATCH (target:Loc) WHERE id(target) = targetNodeId
WITH source, target, distance WHERE source <> target

RETURN source.name AS source, target.name AS target, distance
ORDER BY distance DESC, source ASC, target ASC
LIMIT 10
```

Table 748. Results

Source	Target	Cost
A	F	160
F	A	160
A	E	120

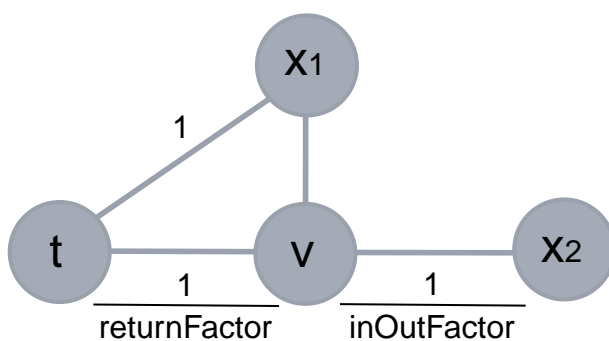
Source	Target	Cost
E	A	120
B	F	110
C	F	110
F	B	110
F	C	110
A	D	90
D	A	90

## 7.5.8. Random Walk Beta

Random Walk is an algorithm that provides random paths in a graph.

A random walk simulates a traversal of the graph in which the traversed relationships are chosen at random. In a classic random walk, each relationship has the same, possibly weighted, probability of being picked. This probability is not influenced by the previously visited nodes. The random walk implementation of the Neo4j Graph Data Science library supports the concept of second order random walks. This method tries to model the transition probability based on the currently visited node  $v$ , the node  $t$  visited before the current one, and the node  $x$  which is the target of a candidate relationship. Random walks are thus influenced by two parameters: the `returnFactor` and the `inOutFactor`:

- The `returnFactor` is used if  $t$  equals  $x$ , i.e., the random walk returns to the previously visited node.
- The `inOutFactor` is used if the distance from  $t$  to  $x$  is equal to 2, i.e., the walk traverses further away from the node  $t$



The probabilities for traversing a relationship during a random walk can be further influenced by specifying a `relationshipWeightProperty`. A relationship property value greater than 1 will increase the likelihood of a relationship being traversed, a property value between 0 and 1 will decrease that probability.



To obtain a random walk where the transition probability is independent of the previously visited nodes both the `returnFactor` and the `inOutFactor` can be set to 1.0.



Running this algorithm requires sufficient memory availability. Before running this algorithm, we recommend that you read [Memory Estimation](#).







Run `RandomWalk` in stream mode on a named graph.

```
CALL gds.beta.randomWalk.stream(
  graphName: String,
  configuration: Map
) YIELD
YIELD
  nodeIds: List of Integer,
  path: Path
```

Table 749. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 750. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 751. Algorithm specific configuration

Name	Type	Default	Optional	Description
sourceNodes	List of Integer	List of all nodes	yes	The list of nodes from which to do a random walk.
walkLength	Integer	80	yes	The number of steps in a single random walk.
walksPerNode	Integer	10	yes	The number of random walks generated for each node.
inOutFactor	Float	1.0	yes	Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local.
returnFactor	Float	1.0	yes	Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be $\geq 0$ . If unspecified, the algorithm runs unweighted.
randomSeed	Integer	random	yes	Seed value for the random number generator used to generate the random walks.

Name	Type	Default	Optional	Description
walkBuffer Size	Integer	1000	yes	The number of random walks to complete before starting training.

Table 752. Results

Name	Type	Description
nodeIds	List of Integer	The nodes of the random walk.
path	Path	A <code>Path</code> object of the random walk.

## Examples

Consider the graph created by the following Cypher statement:

```
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]->(about),
       (about)-[:LINKS]->(home),
       (product)-[:LINKS]->(home),
       (home)-[:LINKS]->(product),
       (links)-[:LINKS]->(home),
       (home)-[:LINKS]->(links),
       (links)-[:LINKS]->(a),
       (a)-[:LINKS]->(home),
       (links)-[:LINKS]->(b),
       (b)-[:LINKS]->(home),
       (links)-[:LINKS]->(c),
       (c)-[:LINKS]->(home),
       (links)-[:LINKS]->(d),
       (d)-[:LINKS]->(home)
```

```
CALL gds.graph.create(
  'myGraph',
  '*',
  { LINKS: { orientation: 'UNDIRECTED' } }
);
```

Without specified source nodes

Run the RandomWalk algorithm on `myGraph`

```
CALL gds.beta.randomWalk.stream(
  'myGraph',
  {
    walkLength: 3,
    walksPerNode: 1,
    randomSeed: 42,
    concurrency: 1
  }
)
YIELD nodeIds, path
RETURN nodeIds, [node IN nodes(path) | node.name ] AS pages
```

Table 753. Results

nodeIds	pages
[0, 5, 3]	[Home, Site B, Links]
[1, 0, 6]	[About, Home, Site C]
[2, 0, 5]	[Product, Home, Site B]
[3, 6, 3]	[Links, Site C, Links]
[4, 3, 4]	[Site A, Links, Site A]
[5, 3, 5]	[Site B, Links, Site B]
[6, 3, 7]	[Site C, Links, Site D]
[7, 3, 0]	[Site D, Links, Home]

With specified source nodes

Run the RandomWalk algorithm on `myGraph` with specified `sourceNodes`

```
MATCH (page:Page)
WHERE page.name IN ['Home', 'About']
WITH COLLECT(page) as sourceNodes
CALL gds.beta.randomWalk.stream(
  'myGraph',
  {
    sourceNodes: sourceNodes,
    walkLength: 3,
    walksPerNode: 1,
    randomSeed: 42,
    concurrency: 1
  }
)
YIELD nodeIds, path
RETURN nodeIds, [node IN nodes(path) | node.name ] AS pages
```

Table 754. Results

nodeIds	pages
[0, 5, 3]	[Home, Site B, Links]
[1, 0, 6]	[About, Home, Site C]

## 7.5.9. Breadth First Search Alpha

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### Introduction

The Breadth First Search algorithm is a graph traversal algorithm that given a start node visits nodes in order of increasing distance, see [https://en.wikipedia.org/wiki/Breadth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search). A related algorithm is the Depth First Search algorithm, [Depth First Search](#). This algorithm is useful for searching when the likelihood of finding the node searched for decreases with distance. There are multiple termination conditions supported for the traversal, based on either reaching one of several target nodes, reaching a maximum depth, exhausting a given budget of traversed relationship cost, or just traversing the whole graph. The output of the procedure contains information about which nodes were visited and in what order.

### Syntax

The following describes the API for running the algorithm and stream results:

```
CALL gds.alpha.bfs.stream(  
  graphName: string,  
  configuration: map  
)  
YIELD  
  // general stream return columns  
  startNodeId: int,  
  nodeIds: int,  
  path: Path
```

Table 755. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

Table 756. General configuration

Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result (applicable in WRITE mode).
nodeProjection	Map or List	null	yes	The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph.

Name	Type	Default	Optional	Description
relationshipProjection	Map or List	<code>null</code>	yes	The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for implicit graph loading via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for implicit graph loading via a Cypher projection.
nodeProperties	Map or List	<code>null</code>	yes	The node properties to load during implicit graph loading.
relationshipProperties	Map or List	<code>null</code>	yes	The relationship properties to load during implicit graph loading.

Table 757. Algorithm specific configuration

Name	Type	Default	Optional	Description
startNodeid	Integer	<code>n/a</code>	no	The node id of the node where to start the traversal.
targetNodes	List of Integer	<code>empty list</code>	yes	Ids for target nodes. Traversal terminates when any target node is visited.
maxDepth	Integer	<code>-1</code>	yes	The maximum distance from the start node at which nodes are visited.

Table 758. Results

Name	Type	Description
startNodeid	Integer	The node id of the node where to start the traversal.
nodeids	List of Integer	The ids of all nodes that were visited during the traversal.
path	Path	A path containing all the nodes that were visited during the traversal.

## Examples

Consider the graph created by the following Cypher statement:

```
CREATE
  (nA:Node {tag: 'a'}),
  (nB:Node {tag: 'b'}),
  (nC:Node {tag: 'c'}),
  (nD:Node {tag: 'd'}),
  (nE:Node {tag: 'e'}),

  (nA)-[:REL {cost: 8.0}]->(nB),
  (nA)-[:REL {cost: 9.0}]->(nC),
  (nB)-[:REL {cost: 1.0}]->(nE),
  (nC)-[:REL {cost: 5.0}]->(nD)
```

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create('myGraph', 'Node', 'REL', { relationshipProperties: 'cost' })
```

In the following examples we will demonstrate using the Breadth First Search algorithm on this graph.

Running the Breadth First Search algorithm:

```
MATCH (a:Node{tag:'a'})
WITH id(a) AS startNode
CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags
```

Table 759. Results

tags
"a"
"b"
"c"
"d"
"e"

Since none of the options for early termination are specified, the whole graph is visited during the traversal.

Running the Breadth First Search algorithm with target nodes:

```
MATCH (a:Node{tag:'a'}), (d:Node{tag:'d'}), (e:Node{tag:'e'})
WITH id(a) AS startNode, [id(d), id(e)] AS targetNodes
CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, targetNodes: targetNodes})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags
```

Table 760. Results

tags
"a"
"b"
"c"
"e"

Running the Breadth First Search algorithm with maxDepth:

```
MATCH (a:Node{tag:'a'})
WITH id(a) AS startNode
CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, maxDepth: 1})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags
```

Table 761. Results

tags
"a"
"b"
"c"

In the above example, nodes d and e were not visited since they are at distance 2 from a.

## 7.5.10. Depth First Search Alpha

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### Introduction

The Depth First Search algorithm is a graph traversal that starts at a given node and explores as far as possible along each branch before backtracking, see [https://en.wikipedia.org/wiki/Depth-first\\_search](https://en.wikipedia.org/wiki/Depth-first_search). A related algorithm is the Breath First Search algorithm, [Breath First Search](#). This algorithm can be preferred over Breath First Search for example if one wants to find a target node at a large distance and exploring a random path has decent probability of success. There are multiple termination conditions supported for the traversal, based on either reaching one of several target nodes, reaching a maximum depth, exhausting a given budget of traversed relationship cost, or just traversing the whole graph. The output of the procedure contains information about which nodes were visited and in what order.

### Syntax

The following describes the API for running the algorithm and stream results:

```
CALL gds.alpha.dfs.stream(
  graphName: String,
  configuration: Map
)
YIELD
  // general stream return columns
  startNodeId: Integer,
  nodeIds: Integer,
  path: Path
```

Table 762. Parameters

Name	Type	Default	Optional	Description
graphName	String or Map	n/a	no	Either the name of a graph stored in the catalog or a Map configuring the graph creation and algorithm execution.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering. Must be empty if graphNameOrConfig is a Map.

Table 763. General configuration

Name	Type	Default	Optional	Description
<code>concurrency</code>	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
<code>readConcurrency</code>	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
<code>writeConcurrency</code>	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result (applicable in WRITE mode).
<code>nodeProjection</code>	Map or List	null	yes	The node projection used for implicit graph loading or filtering nodes of an explicitly loaded graph.
<code>relationshipProjection</code>	Map or List	null	yes	The relationship projection used for implicit graph loading or filtering relationship of an explicitly loaded graph.
<code>nodeQuery</code>	String	null	yes	The Cypher query used to select the nodes for implicit graph loading via a Cypher projection.
<code>relationshipQuery</code>	String	null	yes	The Cypher query used to select the relationships for implicit graph loading via a Cypher projection.
<code>nodeProperties</code>	Map or List	null	yes	The node properties to load during implicit graph loading.
<code>relationshipProperties</code>	Map or List	null	yes	The relationship properties to load during implicit graph loading.

Table 764. Algorithm specific configuration

Name	Type	Default	Optional	Description
<code>startNodeid</code>	Integer	n/a	no	The node id of the node where to start the traversal.
<code>targetNodes</code>	List of Integer	empty list	yes	Ids for target nodes. Traversal terminates when any target node is visited.
<code>maxDepth</code>	Integer	-1	yes	The maximum distance from the start node at which nodes are visited.

Table 765. Results

Name	Type	Description
<code>startNodeid</code>	Integer	The node id of the node where to start the traversal.
<code>nodeids</code>	List of Integer	The ids of all nodes that were visited during the traversal.
<code>path</code>	Path	A path containing all the nodes that were visited during the traversal.

## Examples

Consider the graph created by the following Cypher statement:



```

CREATE
  (nA:Node {tag: 'a'}),
  (nB:Node {tag: 'b'}),
  (nC:Node {tag: 'c'}),
  (nD:Node {tag: 'd'}),
  (nE:Node {tag: 'e'}),

  (nA)-[:REL {cost: 8.0}]->(nB),
  (nA)-[:REL {cost: 9.0}]->(nC),
  (nB)-[:REL {cost: 1.0}]->(nE),
  (nC)-[:REL {cost: 5.0}]->(nD)

```

The following statement will create the graph and store it in the graph catalog.

```

CALL gds.graph.create('myGraph', 'Node', 'REL', { relationshipProperties: 'cost' })

```

In the following examples we will demonstrate using the Depth First Search algorithm on this graph. If we do not specify any of the options for early termination, the whole graph is visited:

Running the Depth First Search algorithm:

```

MATCH (a:Node{tag:'a'})
WITH id(a) AS startNode
CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags

```

Table 766. Results

tags
"a"
"b"
"c"
"d"
"e"

If specifying d and e as target nodes, not all nodes at distance 1 will be visited due to the depth first traversal order, in which node d is reached before b:

Running the Depth First Search algorithm with target nodes:

```

MATCH (a:Node{tag:'a'}), (d:Node{tag:'d'}), (e:Node{tag:'e'})
WITH id(a) AS startNode, [id(d), id(e)] AS targetNodes
CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode, targetNodes: targetNodes})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags

```

Table 767. Results

tags
"a"

tags
"c"
"d"

Running the Depth First Search algorithm with maxDepth:

```
MATCH (a:Node{tag:'a'})
WITH id(a) AS startNode
CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode, maxDepth: 1})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags
```


Table 768. Results

tags
"a"
"b"
"c"

In the above case, nodes d and e were not visited since they are at distance 2 from a.

## 7.6. Topological link prediction

Link prediction algorithms help determine the closeness of a pair of nodes using the topology of the graph. The computed scores can then be used to predict new relationships between them.



The following algorithms use only the topology of the graph to make predictions about relationships between nodes. To make predictions also utilizing node properties one can use the machine learning based methods [Link prediction](#) and [Link prediction pipelines](#).

The Neo4j GDS library includes the following link prediction algorithms, grouped by quality tier:

- Alpha
  - [Adamic Adar](#)
  - [Common Neighbors](#)
  - [Preferential Attachment](#)
  - [Resource Allocation](#)
  - [Same Community](#)
  - [Total Neighbors](#)

### 7.6.1. Adamic Adar Alpha

[Adamic Adar](#) is a measure used to compute the closeness of nodes based on their shared neighbors.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

## History and explanation

The Adamic Adar algorithm was introduced in 2003 by Lada Adamic and Eytan Adar to [predict links in a social network](#). It is computed using the following formula:

$$A(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{\log |N(u)|}$$

where  $N(u)$  is the set of nodes adjacent to  $u$ .

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate closeness between two nodes.

## Syntax

The following will run the algorithm and return the result:

```
RETURN gds.alpha.linkprediction.adamicAdar(node1:Node, node2:Node, {  
  relationshipQuery:String,  
  direction:String  
})
```

Table 769. Parameters

Name	Type	Default	Optional	Description
<code>node1</code>	Node	null	no	A node
<code>node2</code>	Node	null	no	Another node
<code>relationshipQuery</code>	String	null	yes	The relationship type used to compute similarity between <code>node1</code> and <code>node2</code>
<code>direction</code>	String	BOTH	yes	The relationship direction used to compute similarity between <code>node1</code> and <code>node2</code> . Possible values are <code>OUTGOING</code> , <code>INCOMING</code> and <code>BOTH</code> .

## Adamic Adar algorithm sample

The following will create a sample graph:

```
CREATE
(zhen:Person {name: 'Zhen'}),
(praveena:Person {name: 'Praveena'}),
(michael:Person {name: 'Michael'}),
(arya:Person {name: 'Arya'}),
(karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]->(arya),
(zhen)-[:FRIENDS]->(praveena),
(praveena)-[:WORKS_WITH]->(karin),
(praveena)-[:FRIENDS]->(michael),
(michael)-[:WORKS_WITH]->(karin),
(arya)-[:FRIENDS]->(karin)
```

The following will return the Adamic Adar score for Michael and Karin:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.adamicAdar(p1, p2) AS score
```

Table 770. Results

score
0.9102392266268373

We can also compute the score of a pair of nodes based on a specific relationship type.

The following will return the Adamic Adar score for Michael and Karin based only on the **FRIENDS** relationships:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.adamicAdar(p1, p2, {relationshipQuery: 'FRIENDS'}) AS score
```

Table 771. Results

score
0.0

## 7.6.2. Common Neighbors Alpha

Common neighbors captures the idea that two strangers who have a friend in common are more likely to be introduced than those who don't have any friends in common.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

It is computed using the following formula:

$$CN(x, y) = |N(x) \cap N(y)|$$

where  $N(x)$  is the set of nodes adjacent to node  $x$ , and  $N(y)$  is the set of nodes adjacent to node  $y$ .

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate closeness between two nodes.

## Syntax

The following will run the algorithm and return the result:

```
RETURN gds.alpha.linkprediction.commonNeighbors(node1:Node, node2:Node, {
  relationshipQuery:String,
  direction:String
})
```

Table 772. Parameters

Name	Type	Default	Optional	Description
node1	Node	null	no	A node
node2	Node	null	no	Another node
relationshipQuery	String	null	yes	The relationship type used to compute similarity between node1 and node2.
direction	String	BOTH	yes	The relationship direction used to compute similarity between node1 and node2. Possible values are OUTGOING, INCOMING and BOTH.

## Common Neighbors algorithm sample

The following will create a sample graph:

```
CREATE
(zhen:Person {name: 'Zhen'}),
(praveena:Person {name: 'Praveena'}),
(michael:Person {name: 'Michael'}),
(arya:Person {name: 'Arya'}),
(karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]->(arya),
(zhen)-[:FRIENDS]->(praveena),
(praveena)-[:WORKS_WITH]->(karin),
(praveena)-[:FRIENDS]->(michael),
(michael)-[:WORKS_WITH]->(karin),
(arya)-[:FRIENDS]->(karin)
```

The following will return the number of common neighbors for Michael and Karin:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.commonNeighbors(p1, p2) AS score
```

Table 773. Results

score
1.0

We can also compute the score of a pair of nodes based on a specific relationship type.

The following will return the number of common neighbors for Michael and Karin based only on the **FRIENDS** relationships:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.commonNeighbors(p1, p2, {relationshipQuery: "FRIENDS"}) AS score
```

Table 774. Results

score
0.0

### 7.6.3. Preferential Attachment Alpha

Preferential Attachment is a measure used to compute the closeness of nodes, based on their shared neighbors.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

#### History and explanation

Preferential attachment means that the more connected a node is, the more likely it is to receive new links. This algorithm was popularised by [Albert-László Barabási and Réka Albert](#) through their work on scale-free networks. It is computed using the following formula:

$$PA(x, y) = |N(x)| * |N(y)|$$

where  $N(u)$  is the set of nodes adjacent to  $u$ .

A value of 0 indicates that two nodes are not close, while higher values indicate that nodes are closer.

The library contains a function to calculate closeness between two nodes.

#### Syntax

The following will run the algorithm and return the result:

```
RETURN gds.alpha.linkprediction.preferentialAttachment(node1:Node, node2:Node, {
  relationshipQuery:String,
  direction:String
})
```

Table 775. Parameters

Name	Type	Default	Optional	Description
<code>node1</code>	Node	null	no	A node
<code>node2</code>	Node	null	no	Another node
<code>relationshipQuery</code>	String	null	yes	The relationship type used to compute similarity between <code>node1</code> and <code>node2</code>

Name	Type	Default	Optional	Description
<code>direction</code>	String	BOTH	yes	The relationship direction used to compute similarity between <code>node1</code> and <code>node2</code> . Possible values are <code>OUTGOING</code> , <code>INCOMING</code> and <code>BOTH</code> .

## Preferential Attachment algorithm sample

The following will create a sample graph:

```
CREATE
(zhen:Person {name: 'Zhen'}),
(praveena:Person {name: 'Praveena'}),
(michael:Person {name: 'Michael'}),
(arya:Person {name: 'Arya'}),
(karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]->(arya),
(zhen)-[:FRIENDS]->(praveena),
(praveena)-[:WORKS_WITH]->(karin),
(praveena)-[:FRIENDS]->(michael),
(michael)-[:WORKS_WITH]->(karin),
(arya)-[:FRIENDS]->(karin)
```

The following will return the Preferential Attachment score for Michael and Karin:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.preferentialAttachment(p1, p2) AS score
```

Table 776. Results

score
6.0

We can also compute the score of a pair of nodes based on a specific relationship type.

The following will return the Preferential Attachment score for Michael and Karin based only on the `FRIENDS` relationship:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.preferentialAttachment(p1, p2, {relationshipQuery: "FRIENDS"}) AS score
```

Table 777. Results

score
1.0

## 7.6.4. Resource Allocation Alpha

[Resource Allocation](#) is a measure used to compute the closeness of nodes based on their shared neighbors.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

## History and explanation

The Resource Allocation algorithm was introduced in 2009 by Tao Zhou, Linyuan Lü, and Yi-Cheng Zhang as part of a study to predict links in various networks. It is computed using the following formula:

$$RA(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|N(u)|}$$

where  $N(u)$  is the set of nodes adjacent to  $u$ .

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate closeness between two nodes.

## Syntax

The following will run the algorithm and return the result:

```
RETURN gds.alpha.linkprediction.resourceAllocation(node1:Node, node2:Node, {
  relationshipQuery:String,
  direction:String
})
```

Table 778. Parameters

Name	Type	Default	Optional	Description
<code>node1</code>	Node	null	no	A node
<code>node2</code>	Node	null	no	Another node
<code>relationshipQuery</code>	String	null	yes	The relationship type to use to compute similarity between <code>node1</code> and <code>node2</code>
<code>direction</code>	String	BOTH	yes	The relationship direction used to compute similarity between <code>node1</code> and <code>node2</code> . Possible values are <code>OUTGOING</code> , <code>INCOMING</code> and <code>BOTH</code> .

## Resource Allocation algorithm sample

The following will create a sample graph:

```
CREATE
(zhen:Person {name: 'Zhen'}),
(praveena:Person {name: 'Praveena'}),
(michael:Person {name: 'Michael'}),
(arya:Person {name: 'Arya'}),
(karin:Person {name: 'Karin'}),

(zhen)-[:FRIENDS]->(arya),
(zhen)-[:FRIENDS]->(praveena),
(praveena)-[:WORKS_WITH]->(karin),
(praveena)-[:FRIENDS]->(michael),
(michael)-[:WORKS_WITH]->(karin),
(arya)-[:FRIENDS]->(karin)
```



The following will return the Resource Allocation score for Michael and Karin:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.resourceAllocation(p1, p2) AS score
```

Table 779. Results

score
0.3333333333333333

We can also compute the score of a pair of nodes based on a specific relationship type.

The following will return the Resource Allocation score for Michael and Karin based only on the **FRIENDS** relationships:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.resourceAllocation(p1, p2, {relationshipQuery: "FRIENDS"}) AS score
```

Table 780. Results

score
0.0

## 7.6.5. Same Community Alpha

Same Community is a way of determining whether two nodes belong to the same community. These communities could be computed by using one of the [Community detection](#).

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

If two nodes belong to the same community, there is a greater likelihood that there will be a relationship between them in future, if there isn't already.

A value of 0 indicates that two nodes are not in the same community. A value of 1 indicates that two nodes are in the same community.

The library contains a function to calculate closeness between two nodes.

### Syntax

The following will run the algorithm and return the result:

```
RETURN gds.alpha.linkprediction.sameCommunity(node1:Node, node2:Node, communityProperty:String)
```

Table 781. Parameters

Name	Type	Default	Optional	Description
node1	Node	null	no	A node
node2	Node	null	no	Another node
communityProperty	String	'community'	yes	The property that contains the community to which nodes belong

## Same Community algorithm sample

The following will create a sample graph:

```
CREATE (zhen:Person {name: 'Zhen', community: 1}),
       (praveena:Person {name: 'Praveena', community: 2}),
       (michael:Person {name: 'Michael', community: 1}),
       (arya:Person {name: 'Arya', partition: 5}),
       (karin:Person {name: 'Karin', partition: 5}),
       (jennifer:Person {name: 'Jennifer'})
```

The following will indicate that Michael and Zhen belong to the same community:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Zhen'})
RETURN gds.alpha.linkprediction.sameCommunity(p1, p2) AS score
```

Table 782. Results

score
1.0

The following will indicate that Michael and Praveena do not belong to the same community:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Praveena'})
RETURN gds.alpha.linkprediction.sameCommunity(p1, p2) AS score
```

Table 783. Results

score
0.0

If one of the nodes doesn't have a community, this means it doesn't belong to the same community as any other node.

The following will indicate that Michael and Jennifer do not belong to the same community:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Jennifer'})
RETURN gds.alpha.linkprediction.sameCommunity(p1, p2) AS score
```

Table 784. Results

score
0.0

By default, the community is read from the `community` property, but it is possible to explicitly state which property to read from.

The following will indicate that Arya and Karin belong to the same community, based on the `partition` property:

```
MATCH (p1:Person {name: 'Arya'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.sameCommunity(p1, p2, 'partition') AS score
```

Table 785. Results

score
1.0

## 7.6.6. Total Neighbors Alpha

Total Neighbors computes the closeness of nodes, based on the number of unique neighbors that they have. It is based on the idea that the more connected a node is, the more likely it is to receive new links.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### History and explanation

Total Neighbors is computed using the following formula:

$$TN(x, y) = |N(x) \cup N(y)|$$

where  $N(x)$  is the set of nodes adjacent to  $x$ , and  $N(y)$  is the set of nodes adjacent to  $y$ .

A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

The library contains a function to calculate the closeness between two nodes.

### Syntax

The following will run the algorithm and return the result:

```
RETURN gds.alpha.linkprediction.totalNeighbors(node1:Node, node2:Node, {
  relationshipQuery: null,
  direction: "BOTH"
})
```

Table 786. Parameters

Name	Type	Default	Optional	Description
<code>node1</code>	Node	null	no	A node
<code>node2</code>	Node	null	no	Another node
<code>relationshipQuery</code>	String	null	yes	The relationship type used to compute similarity between <code>node1</code> and <code>node2</code>

Name	Type	Default	Optional	Description
<code>direction</code>	String	BOTH	yes	The relationship direction used to compute similarity between <code>node1</code> and <code>node2</code> . Possible values are <code>OUTGOING</code> , <code>INCOMING</code> and <code>BOTH</code> .

## Total Neighbors algorithm sample

The following will create a sample graph:

```
CREATE (zhen:Person {name: 'Zhen'}),
      (praveena:Person {name: 'Praveena'}),
      (michael:Person {name: 'Michael'}),
      (arya:Person {name: 'Arya'}),
      (karin:Person {name: 'Karin'}),

      (zhen)-[:FRIENDS]->(arya),
      (zhen)-[:FRIENDS]->(praveena),
      (praveena)-[:WORKS_WITH]->(karin),
      (praveena)-[:FRIENDS]->(michael),
      (michael)-[:WORKS_WITH]->(karin),
      (arya)-[:FRIENDS]->(karin)
```

The following will return the Total Neighbors score for Michael and Karin:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.totalNeighbors(p1, p2) AS score
```

Table 787. Results

score
4.0

We can also compute the score of a pair of nodes, based on a specific relationship type.

The following will return the Total Neighbors score for Michael and Karin based only on the `FRIENDS` relationship:

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN gds.alpha.linkprediction.totalNeighbors(p1, p2, {relationshipQuery: "FRIENDS"}) AS score
```

Table 788. Results

score
2.0

## 7.7. Node embeddings

Node embedding algorithms compute low-dimensional vector representations of nodes in a graph. These vectors, also called embeddings, can be used for machine learning. The Neo4j Graph Data Science library contains the following node embedding algorithms:

- Production-quality

- [FastRP](#)
- Beta
  - [GraphSAGE](#)
  - [Node2Vec](#)

## 7.7.1. Fast Random Projection

Supported algorithm traits:

[Directed](#)

[Undirected](#)

[Homogeneous](#)

[Heterogeneous](#)

[Weighted](#)

### Introduction

Fast Random Projection, or FastRP for short, is a node embedding algorithm in the family of random projection algorithms. These algorithms are theoretically backed by the Johnson-Lindenstrauss lemma according to which one can project  $n$  vectors of arbitrary dimension into  $O(\log(n))$  dimensions and still approximately preserve pairwise distances among the points. In fact, a linear projection chosen in a random way satisfies this property.

Such techniques therefore allow for aggressive dimensionality reduction while preserving most of the distance information. The FastRP algorithm operates on graphs, in which case we care about preserving similarity between nodes and their neighbors. This means that two nodes that have similar neighborhoods should be assigned similar embedding vectors. Conversely, two nodes that are not similar should not be assigned similar embedding vectors.

The FastRP algorithm initially assigns random vectors to all nodes using a technique called *very sparse random projection*, see (Achlioptas, 2003) below. Moreover, in GDS it is possible to use node properties for the creation of these *initial random vectors* in a way described [below](#). We will also use projection of a node synonymously with the initial random vector of a node.

Starting with these random vectors and iteratively averaging over node neighborhoods, the algorithm constructs a sequence of *intermediate embeddings*  $e_n^i$  for each node  $n$ . More precisely,

$$e_n^i = \text{avg}(e_m^{i-1}),$$

where  $m$  ranges over neighbors of  $n$  and  $e_n^0$  is the node's initial random vector.

The embedding  $e_n$  of node  $n$ , which is the output of the algorithm, is a combination of the vectors and embeddings defined above:

$$e_n = w_0 \cdot \text{normalize}(r_n) + \sum_{i=1}^{i=k} w_i \cdot \text{normalize}(e_n^i),$$

where `normalize` is the function which divides a vector with its `L2 norm`, the value of `nodeSelfInfluence` is  $w_0$ , and the values of `iterationWeights` are  $[w_1, w_2, \dots, w_k]$ . We will return to `Node Self Influence` later on.

Therefore, each node's embedding depends on a neighborhood of radius equal to the number of iterations. This way FastRP exploits higher-order relationships in the graph while still being highly scalable.

The present implementation extends the original algorithm to support weighted graphs, which computes weighted averages of neighboring embeddings using the relationship weights. In order to make use of this, the `relationshipWeightProperty` parameter should be set to an existing relationship property.

The original algorithm is intended only for undirected graphs. We support running on both on directed graphs and undirected graph. For directed graphs we consider only the outgoing neighbors when computing the intermediate embeddings for a node. Therefore, using the orientations `NATURAL`, `REVERSE` or `UNDIRECTED` will all give different embeddings. In general, it is recommended to first use `UNDIRECTED` as this is what the original algorithm was evaluated on.

For more information on this algorithm see:

- [H. Chen, S.F. Sultan, Y. Tian, M. Chen, S. Skiena: Fast and Accurate Network Embeddings via Very Sparse Random Projection, 2019.](#)
- [Dimitris Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. Journal of Computer and System Sciences, 66\(4\):671–687, 2003.](#)

## Node properties

Most real-world graphs contain node properties which store information about the nodes and what they represent. The FastRP algorithm in the GDS library extends the original FastRP algorithm with a capability to take node properties into account. The resulting embeddings can therefore represent the graph more accurately.

The node property aware aspect of the algorithm is configured via the parameters `featureProperties` and `propertyRatio`. Each node property in `featureProperties` is associated with a randomly generated vector of dimension `propertyDimension`, where `propertyDimension = embeddingDimension * propertyRatio`. Each node is then initialized with a vector of size `embeddingDimension` formed by concatenation of two parts:

1. The first part is formed like in the standard FastRP algorithm,
2. The second one is a linear combination of the property vectors, using the property values of the node as weights.

The algorithm then proceeds with the same logic as the FastRP algorithm. Therefore, the algorithm will output arrays of size `embeddingDimension`. The last `propertyDimension` coordinates in the embedding captures information about property values of nearby nodes (the "property part" below), and the remaining coordinates (`embeddingDimension - propertyDimension` of them; "topology part") captures information about nearby presence of nodes.

```

[0, 1, ... | ..., N - 1, N]
^^^^^^^^^^ | ^^^^^^^^^^^
topology part | property part
              ^
              property ratio

```

## Tuning algorithm parameters

In order to improve the embedding quality using FastRP on one of your graphs, it is possible to tune the algorithm parameters. This process of finding the best parameters for your specific use case and graph is typically referred to as [hyperparameter tuning](#). We will go through each of the configuration parameters and explain how they behave.

For statistically sound results, it is a good idea to reserve a test set excluded from parameter tuning. After selecting a set of parameter values, the embedding quality can be evaluated using a downstream machine learning task on the test set. By varying the parameter values and studying the precision of the machine learning task, it is possible to deduce the parameter values that best fit the concrete dataset and use case. To construct such a set you may want to use a dedicated node label in the graph to denote a subgraph without the test data.

### Embedding dimension

The embedding dimension is the length of the produced vectors. A greater dimension offers a greater precision, but is more costly to operate over.

The optimal embedding dimension depends on the number of nodes in the graph. Since the amount of information the embedding can encode is limited by its dimension, a larger graph will tend to require a greater embedding dimension. A typical value is a power of two in the range 128 - 1024. A value of at least 256 gives good results on graphs in the order of  $10^5$  nodes, but in general increasing the dimension improves results. Increasing embedding dimension will however increase memory requirements and runtime linearly.

### Normalization strength

The normalization strength is used to control how node degrees influence the embedding. Using a negative value will downplay the importance of high degree neighbors, while a positive value will instead increase their importance. The optimal normalization strength depends on the graph and on the task that the embeddings will be used for. In the original paper, hyperparameter tuning was done in the range of `[-1, 0]` (no positive values), but we have found cases where a positive normalization strengths gives better results.

### Iteration weights

The iteration weights parameter control two aspects: the number of iterations, and their relative impact on the final node embedding. The parameter is a list of numbers, indicating one iteration per number where the number is the weight applied to that iteration.

In each iteration, the algorithm will expand across all relationships in the graph. This has some implications:

- With a single iteration, only direct neighbors will be considered for each node embedding.
- With two iterations, direct neighbors and second-degree neighbors will be considered for each node embedding.
- With three iterations, direct neighbors, second-degree neighbors, and third-degree neighbors will be considered for each node embedding. Direct neighbors may be reached twice, in different iterations.
- In general, the embedding corresponding to the  $i$ :th iteration contains features depending on nodes reachable with paths of length  $i$ . If the graph is undirected, then a node reachable with a path of length  $L$  can also be reached with length  $L+2k$ , for any integer  $k$ .
- In particular, a node may reach back to itself on each even iteration (depending on the direction in the graph).

It is good to have at least one non-zero weight in an even and in an odd position. Typically, using at least a few iterations, for example three, is recommended. However, a too high value will consider nodes far away and may not be informative or even be detrimental. The intuition here is that as the projections reach further away from the node, the less specific the neighborhood becomes. Of course, a greater number of iterations will also take more time to complete.

## Node Self Influence

Node Self Influence is a variation of the original FastRP algorithm.

How much a node's embedding is affected by the intermediate embedding at iteration  $i$  is controlled by the  $i$ 'th element of `iterationWeights`. This can also be seen as how much the initial random vectors, or projections, of nodes that can be reached in  $i$  hops from a node affect the embedding of the node. Similarly, `nodeSelfInfluence` behaves like an iteration weight for a 0th iteration, or the amount of influence the projection of a node has on the embedding of the same node.

A reason for setting this parameter to a non-zero value is if your graph has low connectivity or a significant amount of isolated nodes. Isolated nodes combined with using `propertyRatio = 0.0` leads to embeddings that contain all zeros. However using node properties along with node self influence can thus produce more meaningful embeddings for such nodes. This can be seen as producing fallback features when graph structure is (locally) missing. Moreover, sometimes a node's own properties are simply informative features and are good to include even if connectivity is high. Finally, node self influence can be used for pure dimensionality reduction to compress node properties used for node classification.

If node properties are not used, using `nodeSelfInfluence` may also have a positive effect, depending on other settings and on the problem.

## Orientation

Choosing the right orientation when creating the graph may have the single greatest impact. The FastRP algorithm is designed to work with undirected graphs, and we expect this to be the best in most cases. If



you expect only outgoing or incoming relationships to be informative for a prediction task, then you may want to try using the orientations **NATURAL** or **REVERSE** respectively.

## Syntax

This section covers the syntax used to execute the FastRP algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run FastRP in stream mode on a named graph.

```
CALL gds.fastRP.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  embedding: List of Float
```

Table 789. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 790. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 791. Algorithm specific configuration

Name	Type	Default	Optional	Description
propertyRatio	Float	0.0	yes	The desired ratio of the property embedding dimension to the total <code>embeddingDimension</code> . A positive value requires <code>featureProperties</code> to be non-empty.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
embeddingDimension	Integer	n/a	no	The dimension of the computed node embeddings. Minimum value is 1.
iterationWeights	List of Float	[0.0, 1.0, 1.0]	yes	Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding.
nodeSelfInfluence	Float	0.0	yes	Controls for each node how much its initial random vector contributes to its final embedding.
normalizationStrength	Float	0.0	yes	The initial random vector for each node is scaled by its degree to the power of <code>normalizationStrength</code> .
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in computing the embeddings.

Name	Type	Default	Optional	Description
<a href="#">relationshipWeightProperty</a>	String	<code>null</code>	yes	Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted.
The number of iterations is equal to the length of <code>iterationWeights</code> .				
It is required that <code>iterationWeights</code> is non-empty or <code>nodeSelfInfluence</code> is non-zero.				

Table 792. Results

Name	Type	Description
<code>nodeId</code>	Integer	Node ID.
<code>embedding</code>	List of Float	FastRP node embedding.

Run FastRP in stats mode on a named graph.

```
CALL gds.fastRP.stats(  
  graphName: String,  
  configuration: Map  
) YIELD  
  nodeCount: Integer,  
  createMillis: Integer,  
  computeMillis: Integer,  
  configuration: Map
```

Table 793. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 794. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 795. Algorithm specific configuration

Name	Type	Default	Optional	Description
propertyRatio	Float	0.0	yes	The desired ratio of the property embedding dimension to the total <code>embeddingDimension</code> . A positive value requires <code>featureProperties</code> to be non-empty.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
embeddingDimension	Integer	n/a	no	The dimension of the computed node embeddings. Minimum value is 1.
iterationWeights	List of Float	[0.0, 1.0, 1.0]	yes	Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding.
nodeSelfInfluence	Float	0.0	yes	Controls for each node how much its initial random vector contributes to its final embedding.
normalizationStrength	Float	0.0	yes	The initial random vector for each node is scaled by its degree to the power of <code>normalizationStrength</code> .
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in computing the embeddings.

Name	Type	Default	Optional	Description
<a href="#">relationshipWeightProperty</a>	String	<code>null</code>	yes	Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted.
The number of iterations is equal to the length of <code>iterationWeights</code> .				
It is required that <code>iterationWeights</code> is non-empty or <code>nodeSelfInfluence</code> is non-zero.				

Table 796. Results

Name	Type	Description
<code>nodeCount</code>	Integer	Number of nodes processed.
<code>createMillis</code>	Integer	Milliseconds for creating the graph.
<code>computeMillis</code>	Integer	Milliseconds for running the algorithm.
<code>configuration</code>	Map	Configuration used for running the algorithm.

Run FastRP in mutate mode on a named graph.

```
CALL gds.fastRP.mutate(  
  graphName: String,  
  configuration: Map  
) YIELD  
  nodeCount: Integer,  
  nodePropertiesWritten: Integer,  
  createMillis: Integer,  
  computeMillis: Integer,  
  mutateMillis: Integer,  
  configuration: Map
```

Table 797. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 798. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 799. Algorithm specific configuration

Name	Type	Default	Optional	Description
propertyRatio	Float	0.0	yes	The desired ratio of the property embedding dimension to the total <code>embeddingDimension</code> . A positive value requires <code>featureProperties</code> to be non-empty.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
embeddingDimension	Integer	n/a	no	The dimension of the computed node embeddings. Minimum value is 1.
iterationWeights	List of Float	[0.0, 1.0, 1.0]	yes	Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding.
nodeSelfInfluence	Float	0.0	yes	Controls for each node how much its initial random vector contributes to its final embedding.
normalizationStrength	Float	0.0	yes	The initial random vector for each node is scaled by its degree to the power of <code>normalizationStrength</code> .

Name	Type	Default	Optional	Description
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in computing the embeddings.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted.

The number of iterations is equal to the length of `iterationWeights`.

It is required that `iterationWeights` is non-empty or `nodeSelfInfluence` is non-zero.

Table 800. Results

Name	Type	Description
nodeCount	Integer	Number of nodes processed.
nodePropertiesWritten	Integer	Number of node properties written.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
configuration	Map	Configuration used for running the algorithm.



Run FastRP in write mode on a named graph.

```
CALL gds.fastRP.write(
  graphName: String,
  configuration: Map
) YIELD
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 801. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 802. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 803. Algorithm specific configuration

Name	Type	Default	Optional	Description
propertyRatio	Float	0.0	yes	The desired ratio of the property embedding dimension to the total embeddingDimension. A positive value requires featureProperties to be non-empty.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
embeddingDimension	Integer	n/a	no	The dimension of the computed node embeddings. Minimum value is 1.
iterationWeights	List of Float	[0.0, 1.0, 1.0]	yes	Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding.
nodeSelfInfluence	Float	0.0	yes	Controls for each node how much its initial random vector contributes to its final embedding.

Name	Type	Default	Optional	Description
normalizationStrength	Float	0.0	yes	The initial random vector for each node is scaled by its degree to the power of <code>normalizationStrength</code> .
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in computing the embeddings.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted.

The number of iterations is equal to the length of `iterationWeights`.

It is required that `iterationWeights` is non-empty or `nodeSelfInfluence` is non-zero.

Table 804. Results

Name	Type	Description
nodeCount	Integer	Number of nodes processed.
nodePropertiesWritten	Integer	Number of node properties written.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
configuration	Map	Configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run FastRP in write mode on an anonymous graph.

```
CALL gds.fastRP.write(
  configuration: Map
)
YIELD
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 805. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	null	yes	The relationship properties to project during anonymous graph creation.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
writeConcurrency	Integer	value of 'concurrency'	yes	WRITE mode only: The number of concurrent threads used for writing the result.
writeProperty	String	n/a	no	WRITE mode only: The node property to which the embedding is written to.

Table 806. Algorithm specific configuration

Name	Type	Default	Optional	Description
propertyRatio	Float	0.0	yes	The desired ratio of the property embedding dimension to the total embeddingDimension. A positive value requires featureProperties to be non-empty.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
embeddingDimension	Integer	n/a	no	The dimension of the computed node embeddings. Minimum value is 1.
iterationWeights	List of Float	[0.0, 1.0, 1.0]	yes	Contains a weight for each iteration. The weight controls how much the intermediate embedding from the iteration contributes to the final embedding.
nodeSelfInfluence	Float	0.0	yes	Controls for each node how much its initial random vector contributes to its final embedding.
normalizationStrength	Float	0.0	yes	The initial random vector for each node is scaled by its degree to the power of normalizationStrength.

Name	Type	Default	Optional	Description
randomSeed	Integer	n/a	yes	A random seed which is used for all randomness in computing the embeddings.
<a href="#">relationshipWeightProperty</a>	String	null	yes	Name of the relationship property to use for weighted random projection. If unspecified, the algorithm runs unweighted.

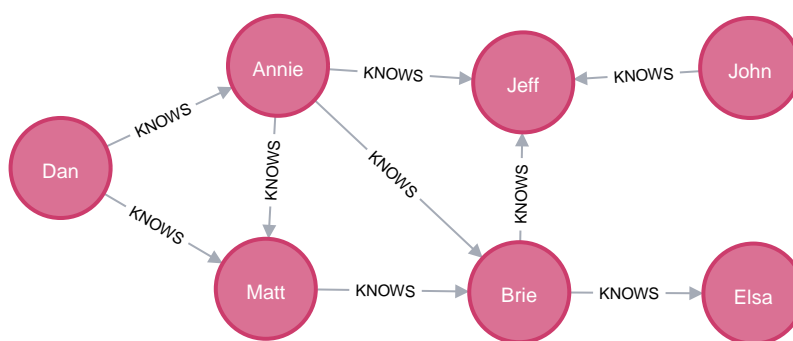
The number of iterations is equal to the length of `iterationWeights`.

It is required that `iterationWeights` is non-empty or `nodeSelfInfluence` is non-zero.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the FastRP node embedding algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
  (dan:Person {name: 'Dan', age: 18}),
  (annie:Person {name: 'Annie', age: 12}),
  (matt:Person {name: 'Matt', age: 22}),
  (jeff:Person {name: 'Jeff', age: 51}),
  (brie:Person {name: 'Brie', age: 45}),
  (elsa:Person {name: 'Elsa', age: 65}),
  (john:Person {name: 'John', age: 64}),

  (dan)-[:KNOWS {weight: 1.0}]->(annie),
  (dan)-[:KNOWS {weight: 1.0}]->(matt),
  (annie)-[:KNOWS {weight: 1.0}]->(matt),
  (annie)-[:KNOWS {weight: 1.0}]->(jeff),
  (annie)-[:KNOWS {weight: 1.0}]->(brie),
  (matt)-[:KNOWS {weight: 3.5}]->(brie),
  (brie)-[:KNOWS {weight: 1.0}]->(elsa),
  (brie)-[:KNOWS {weight: 2.0}]->(jeff),
  (john)-[:KNOWS {weight: 1.0}]->(jeff);

```

This graph represents seven people who know one another. A relationship property `weight` denotes the strength of the knowledge between two persons.

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. For the

relationships we will use the **UNDIRECTED** orientation. This is because the FastRP algorithm has been measured to compute more predictive node embeddings in undirected graphs. We will also add the **weight** relationship property which we will make use of when running the weighted version of FastRP.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'persons'.

```
CALL gds.graph.create(  
  'persons',  
  'Person',  
  {  
    KNOWS: {  
      orientation: 'UNDIRECTED',  
      properties: 'weight'  
    }  
  },  
  { nodeProperties: ['age'] }  
)
```

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the **estimate** procedure. This can be done with any execution mode. We will use the **stream** mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on **estimate** in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.fastRP.stream.estimate('persons', {embeddingDimension: 128})  
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 807. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
7	18	11424	11424	"11424 Bytes"

## Stream

In the **stream** execution mode, the algorithm returns the embedding for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the **stream** mode in general, see [Stream](#).

The following will run the algorithm, and stream results:

```
CALL gds.fastRP.stream('persons',
{
  embeddingDimension: 4,
  randomSeed: 42
})
YIELD nodeId, embedding
```

Table 808. Results

nodeId	embedding
0	[0.4774002134799957, -0.6602408289909363, -0.36686956882476807, -1.7089111804962158]
1	[0.7989360094070435, -0.4918718934059143, -0.41281944513320923, -1.6314401626586914]
2	[0.47275322675704956, -0.49587157368659973, -0.3340468406677246, -1.7141895294189453]
3	[0.8290714025497437, -0.3260476291179657, -0.3317275643348694, -1.4370529651641846]
4	[0.7749264240264893, -0.4773247539997101, 0.0675133764743805, -1.5248265266418457]
5	[0.8408374190330505, -0.37151476740837097, 0.12121132016181946, -1.530960202217102]
6	[1.0, -0.11054422706365585, -0.3697933852672577, -0.9225144982337952]

The results of the algorithm are not very intuitively interpretable, as the node embedding format is a mathematical abstraction of the node within its neighborhood, designed for machine learning programs. What we can see is that the embeddings have four elements (as configured using `embeddingDimension`) and that the numbers are relatively small (they all fit in the range of `[-2, 2]`). The magnitude of the numbers is controlled by the `embeddingDimension`, the number of nodes in the graph, and by the fact that FastRP performs euclidean normalization on the intermediate embedding vectors.



Due to the random nature of the algorithm the results will vary between the runs. However, this does not necessarily mean that the pairwise distances of two node embeddings vary as much.

## Stats

In the `stats` execution mode, the algorithm returns a single row containing a summary of the algorithm result. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the `computeMillis` return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](#).

For more details on the `stats` mode in general, see [Stats](#).

The following will run the algorithm and returns the result in form of statistical and measurement values

```
CALL gds.fastRP.stats('persons', { embeddingDimension: 8 })
YIELD nodeCount
```

Table 809. Results

nodeCount
7

The `stats` mode does not currently offer any statistical results for the embeddings themselves. We can however see that the algorithm has successfully processed all seven nodes in our example graph.

## Mutate

The `mutate` execution mode extends the `stats` mode with an important side effect: updating the named graph with a new node property containing the embedding for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

The following will run the algorithm in `mutate` mode:

```
CALL gds.fastRP.mutate(
  'persons',
  {
    embeddingDimension: 8,
    mutateProperty: 'fastrp-embedding'
  }
)
YIELD nodePropertiesWritten
```

Table 810. Results

nodePropertiesWritten
7

The returned result is similar to the `stats` example. Additionally, the graph 'persons' now has a node property `fastrp-embedding` which stores the node embedding for each node. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs](#).

## Write

The `write` execution mode extends the `stats` mode with an important side effect: writing the embedding for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row, similar to `stats`, but with some additional metrics. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

The following will run the algorithm in `write` mode:

```
CALL gds.fastRP.write(  
  'persons',  
  {  
    embeddingDimension: 8,  
    writeProperty: 'fastrp-embedding'  
  }  
)  
YIELD nodePropertiesWritten
```

Table 811. Results

nodePropertiesWritten
7

The returned result is similar to the `stats` example. Additionally, each of the seven nodes now has a new property `fastrp-embedding` in the Neo4j database, containing the node embedding for that node.

## Weighted

By default, the algorithm is considering the relationships of the graph to be unweighted. To change this behaviour we can use configuration parameter called `relationshipWeightProperty`. Below is an example of running the weighted variant of algorithm.

The following will run the algorithm, and stream results:

```
CALL gds.fastRP.stream(  
  'persons',  
  {  
    embeddingDimension: 4,  
    randomSeed: 42,  
    relationshipWeightProperty: 'weight'  
  }  
)  
YIELD nodeId, embedding
```

Table 812. Results

nodeId	embedding
0	[0.10945529490709305, -0.5032674074172974, 0.464673787355423, -1.7539862394332886]
1	[0.3639600872993469, -0.39210301637649536, 0.46271592378616333, -1.829423427581787]
2	[0.12314096093177795, -0.3213110864162445, 0.40100979804992676, -1.471055269241333]
3	[0.30704641342163086, -0.24944794178009033, 0.3947891891002655, -1.3463698625564575]
4	[0.23112300038337708, -0.30148714780807495, 0.584831714630127, -1.2822188138961792]



nodeId	embedding
5	[0.14497177302837372, -0.2312137484550476, 0.5552002191543579, -1.2605633735656738]
6	[0.5139184594154358, -0.07954332232475281, 0.3690345287322998, -0.9176374077796936]

Since the initial state of the algorithm is randomised, it isn't possible to intuitively analyse the effect of the relationship weights.

### Using node properties as features

To explain the novel initialization using node properties, let us consider an example where `embeddingDimension` is 10, `propertyRatio` is 0.2. The dimension of the embedded properties, `propertyDimension` is thus 2. Assume we have a property `f1` of scalar type, and a property `f2` storing arrays of length 2. This means that there are 3 features which we order like `f1` followed by the two values of `f2`. For each of these three features we sample a two dimensional random vector. Let's say these are `p1=[0.0, 2.4]`, `p2=[-2.4, 0.0]` and `p3=[2.4, 0.0]`. Consider now a node (`n {f1: 0.5, f2: [1.0, -1.0]}`). The linear combination mentioned above, is in concrete terms  $0.5 * p1 + 1.0 * p2 - 1.0 * p3 = [-4.8, 1.2]$ . The initial random vector for the node `n` contains first 8 values sampled as in the original FastRP paper, and then our computed values `-4.8` and `1.2`, totalling 10 entries.

In the example below, we again set the embedding dimension to 2, but we set `propertyRatio` to 1, which means the embedding is computed from node properties only.

The following will run FastRP with feature properties:

```
CALL gds.fastRP.stream('persons', {
  randomSeed: 42,
  embeddingDimension: 2,
  propertyRatio: 1.0,
  featureProperties: ['age'],
  iterationWeights: [1.0]
}) YIELD nodeId, embedding
```

Table 813. Results

nodeId	embedding
0	[0.0, -1.0]
1	[0.0, -1.0]
2	[0.0, -0.9999999403953552]
3	[0.0, -1.0]
4	[0.0, -0.9999999403953552]
5	[0.0, -1.0]
6	[0.0, -1.0]

In this example, the embeddings are based on the `age` property. Because of L2 normalization which is applied to each iteration (here only one iteration), all nodes have the same embedding despite having

different age values (apart from rounding errors).

## 7.7.2. GraphSAGE Beta

GraphSAGE is an *inductive* algorithm for computing node embeddings. GraphSAGE is using node feature information to generate node embeddings on unseen nodes or graphs. Instead of training individual embeddings for each node, the algorithm learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood.



The algorithm is defined for UNDIRECTED graphs.

For more information on this algorithm see:

- [William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs." 2018.](#)
- [Amit Pande, Kai Ni and Venkataramani Kini. "SWAG: Item Recommendations using Convolutions on Weighted Graphs." 2019.](#)

Syntax



Run GraphSAGE in train mode on a named graph.

```
CALL gds.beta.graphSage.train(
  graphName: String,
  configuration: Map
) YIELD
  graphName: String,
  graphCreateConfig: Map,
  modelInfo: Map,
  configuration: Map,
  trainMillis: Integer
```

Table 814. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 815. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 816. Algorithm specific configuration

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of the model to train, must not exist in the Model Catalog.
featureProperties	List of String	n/a	no	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
embeddingDimension	Integer	64	yes	The dimension of the generated node embeddings as well as their hidden layer representations.
aggregator	String	"mean"	yes	The aggregator to be used by the layers. Supported values are "mean" and "pool".
activationFunction	String	"sigmoid"	yes	The activation function to be used in the model architecture. Supported values are "sigmoid" and "relu".

Name	Type	Default	Optional	Description
sampleSizes	List of Integer	[25, 10]	yes	A list of Integer values, the size of the list determines the number of layers and the values determine how many nodes will be sampled by the layers.
projectedFeatureDimension	Integer	n/a	yes	The dimension of the projected <code>featureProperties</code> . This enables multi-label GraphSage, where each label can have a subset of the <code>featureProperties</code> .
batchSize	Integer	100	yes	The number of nodes per batch.
tolerance	Float	1e-4	yes	Tolerance used for the early convergence of an epoch.
learningRate	Float	0.1	yes	The learning rate determines the step size at each iteration while moving toward a minimum of a loss function.
epochs	Integer	1	yes	Number of times to traverse the graph.
maxIterations	Integer	10	yes	Maximum number of weight updates per batch. Batches can also converge early based on <code>tolerance</code> .
searchDepth	Integer	5	yes	Maximum depth of the RandomWalks to sample nearby nodes for the training.
negativeSampleWeight	Integer	20	yes	The weight of the negative samples. Higher values increase the impact of negative samples in the loss.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
randomSeed	Integer	random	yes	A random seed which is used to control the randomness in computing the embeddings.

Table 817. Results

Name	Type	Description
graphName	String	The name of the in-memory graph used during training.
graphCreateConfig	Map	Configuration used to create in-memory graph. Only has value if <code>anonymous graph</code> was used.
modelInfo	Map	Details of the trained model.
configuration	Map	The configuration used to run the procedure.
trainMillis	Integer	Milliseconds to train the model.

Table 818. Details on `modelInfo`

Name	Type	Description
<code>name</code>	String	The name of the trained model.
<code>type</code>	String	The type of the trained model. Always <code>graphSage</code> .
<code>metrics</code>	Map	Metrics related to running the training, details in the table below.

Table 819. Metrics collected during training

Name	Type	Description
<code>ranEpochs</code>	Integer	The number of ran epochs during training.
<code>epochLosses</code>	List	Ordered list of the losses after each epoch.
<code>didConverge</code>	Boolean	Indicates if the training has converged.

Run GraphSAGE in stream mode on a named graph.

```
CALL gds.beta.graphSage.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  embedding: List
```

Table 820. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 821. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 822. Algorithm specific configuration

Name	Type	Default	Optional	Description
batchSize	Integer	100	yes	The number of nodes per batch.

Table 823. Results

Name	Type	Description
nodeId	Integer	The Neo4j node ID.
embedding	List of Float	The computed node embedding.

Run GraphSAGE in mutate mode on a graph stored in the catalog.

```
CALL gds.beta.graphSage.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  nodeCount: Integer,  
  nodePropertiesWritten: Integer,  
  createMillis: Integer,  
  computeMillis: Integer,  
  mutateMillis: Integer,  
  configuration: Map
```

Table 824. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 825. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 826. Algorithm specific configuration

Name	Type	Default	Optional	Description
batchSize	Integer	100	yes	The number of nodes per batch.

Table 827. Results

Name	Type	Description
nodeCount	Integer	The number of nodes processed.
nodePropertiesWritten	Integer	The number of node properties written.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for writing result data back to the in-memory graph.
configuration	Map	The configuration used for running the algorithm.



Run GraphSAGE in write mode on a graph stored in the catalog.

```
CALL gds.beta.graphSage.write(
  graphName: String,
  configuration: Map
)
YIELD
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  configuration: Map
```

Table 828. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 829. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 830. Algorithm specific configuration

Name	Type	Default	Optional	Description
batchSize	Integer	100	yes	The number of nodes per batch.

Table 831. Results

Name	Type	Description
nodeCount	Integer	The number of nodes processed.
nodePropertiesWritten	Integer	The number of node properties written.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.

Name	Type	Description
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run GraphSAGE in write mode on an anonymous graph.

```
CALL gds.beta.graphSage.write(
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 832. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation a Native projection.
nodeQuery	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipQuery	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, List of String or Map	<code>null</code>	yes	The relationship properties to project during anonymous graph creation.
<code>concurrency</code>	Integer	<code>4</code>	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.

Name	Type	Default	Optional	Description
<code>writeConcurrency</code>	Integer	value of 'concurrency'	yes	WRITE mode only: The number of concurrent threads used for writing the result.
<code>writeProperty</code>	String	n/a	no	WRITE mode only: The node property to which the embedding is written to.

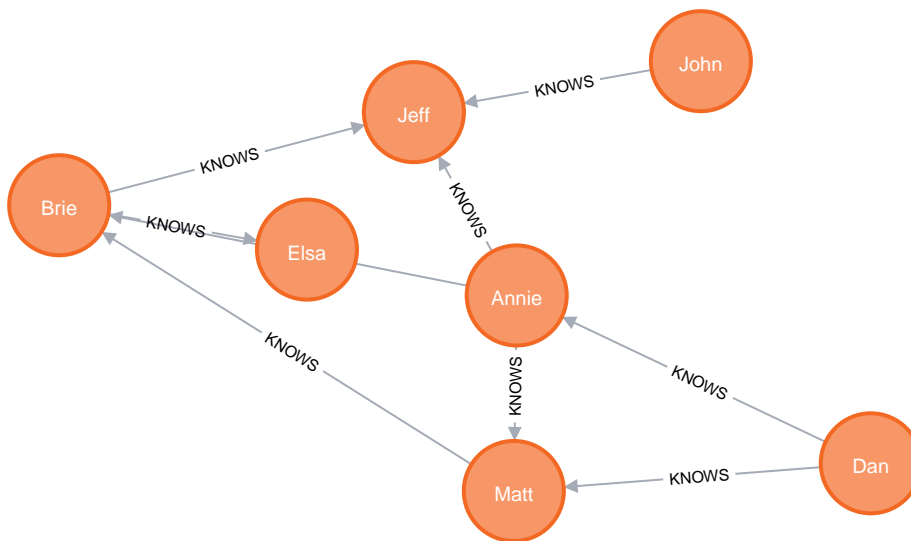
Table 833. Algorithm specific configuration

Name	Type	Default	Optional	Description
<code>batchSize</code>	Integer	100	yes	The number of nodes per batch.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

In this section we will show examples of running the GraphSAGE algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small friends network graph of a handful of nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
// Persons
( dan:Person {name: 'Dan', age: 20, heightAndWeight: [185, 75]}),
( annie:Person {name: 'Annie', age: 12, heightAndWeight: [124, 42]}),
( matt:Person {name: 'Matt', age: 67, heightAndWeight: [170, 80]}),
( jeff:Person {name: 'Jeff', age: 45, heightAndWeight: [192, 85]}),
( brie:Person {name: 'Brie', age: 27, heightAndWeight: [176, 57]}),
( elsa:Person {name: 'Elsa', age: 32, heightAndWeight: [158, 55]}),
( john:Person {name: 'John', age: 35, heightAndWeight: [172, 76]}),

(dan)-[:KNOWS {relWeight: 1.0}]->(annie),
(dan)-[:KNOWS {relWeight: 1.6}]->(matt),
(annie)-[:KNOWS {relWeight: 0.1}]->(matt),
(annie)-[:KNOWS {relWeight: 3.0}]->(jeff),
(annie)-[:KNOWS {relWeight: 1.2}]->(brie),
(matt)-[:KNOWS {relWeight: 10.0}]->(brie),
(brie)-[:KNOWS {relWeight: 1.0}]->(elsa),
(brie)-[:KNOWS {relWeight: 2.2}]->(jeff),
(john)-[:KNOWS {relWeight: 5.0}]->(jeff)
```

```
CALL gds.graph.create(
  'persons',
  {
    Person: {
      label: 'Person',
      properties: ['age', 'heightAndWeight']
    }
  }, {
    KNOWS: {
      type: 'KNOWS',
      orientation: 'UNDIRECTED',
      properties: ['relWeight']
    }
  }
)
```



The algorithm is defined for **UNDIRECTED** graphs.

## Train

Before we are able to generate node embeddings we need to train a model and store it in the model catalog. Below is an example of how to do that.



The names specified in the **featureProperties** configuration parameter must exist in the in-memory graph.

```
CALL gds.beta.graphSage.train(
  'persons',
  {
    modelName: 'exampleTrainModel',
    featureProperties: ['age', 'heightAndWeight'],
    aggregator: 'mean',
    activationFunction: 'sigmoid',
    randomSeed: 1337,
    sampleSizes: [25, 10]
  }
) YIELD modelInfo as info
RETURN
  info.modelName as modelName,
  info.metrics.didConverge as didConverge,
  info.metrics.ranEpochs as ranEpochs,
  info.metrics.epochLosses as epochLosses
```

Table 834. Results

modelName	didConverge	ranEpochs	epochLosses
"exampleTrainModel"	false	1	[186.04946807210226]



Due to the random initialisation of the weight variables the results may vary between different runs.

Looking at the results we can draw the following conclusions, the training converged after a single epoch, the losses are almost identical. Tuning the algorithm parameters, such as trying out different `sampleSizes`, `searchDepth`, `embeddingDimension` or `batchSize` can improve the losses. For different datasets, GraphSAGE may require different train parameters for producing good models.

The trained model is automatically registered in the [model catalog](#).

### Train with multiple node labels

In this section we describe how to train on a graph with multiple labels. The different labels may have different sets of properties. To run on such a graph, GraphSAGE is run in *multi-label mode*, in which the feature properties are projected into a common feature space. Therefore, all nodes have feature vectors of the same dimension after the projection.

The projection for a label is linear and given by a matrix of weights. The weights for each label are learned jointly with the other weights of the GraphSAGE model.

In the multi-label mode, the following is applied prior to the usual aggregation layers:

1. A property representing the label is added to the feature properties for that label
2. The feature properties for each label are projected into a feature vector of a shared dimension

The projected feature dimension is configured with `projectedFeatureDimension`, and specifying it enables the multi-label mode.

The feature properties used for a label are those present in the `featureProperties` configuration parameter which exist in the graph for that label. In the multi-label mode, it is no longer required that all labels have all the specified properties.

### Assumptions

- A requirement for multi-label mode is that each node belongs to exactly one label.
- A GraphSAGE model trained in this mode must be applied on graphs with the same schema with regards to node labels and properties.

## Examples

In order to demonstrate GraphSAGE with multiple labels, we add instruments and relationships of type **LIKE** between person and instrument to the example graph.



The following Cypher statement will extend the example graph in the Neo4j database:

```
MATCH
  (dan:Person {name: "Dan"}),
  (annie:Person {name: "Annie"}),
  (matt:Person {name: "Matt"}),
  (brie:Person {name: "Brie"}),
  (john:Person {name: "John"})
CREATE
  (guitar:Instrument {name: 'Guitar', cost: 1337.0}),
  (synth:Instrument {name: 'Synthesizer', cost: 1337.0}),
  (bongos:Instrument {name: 'Bongos', cost: 42.0}),
  (trumpet:Instrument {name: 'Trumpet', cost: 1337.0}),
  (dan)-[:LIKES]->(guitar),
  (dan)-[:LIKES]->(synth),
  (dan)-[:LIKES]->(bongos),
  (annie)-[:LIKES]->(guitar),
  (annie)-[:LIKES]->(synth),
  (matt)-[:LIKES]->(bongos),
  (brie)-[:LIKES]->(guitar),
  (brie)-[:LIKES]->(synth),
  (brie)-[:LIKES]->(bongos),
  (john)-[:LIKES]->(trumpet)
```

```
CALL gds.graph.create(
  'persons_with_instruments',
  {
    Person: {
      label: 'Person',
      properties: ['age', 'heightAndWeight']
    },
    Instrument: {
      label: 'Instrument',
      properties: ['cost']
    }
  }, {
    KNOWS: {
      type: 'KNOWS',
      orientation: 'UNDIRECTED'
    },
    LIKES: {
      type: 'LIKES',
      orientation: 'UNDIRECTED'
    }
  }
})
```

We can now run GraphSAGE in multi-label mode on that graph by specifying the `projectedFeatureDimension` parameter. Multi-label GraphSAGE removes the requirement, that each node in the in-memory graph must have all `featureProperties`. However, the projections are independent per label and even if two labels have the same `featureProperty` they are considered as different features before projection. The `projectedFeatureDimension` equals the maximum length of the feature-array, i.e., `age` and `cost` both are scalar features plus the list feature `heightAndWeight` which has a length of two. For each node its unique labels properties is projected using a label specific projection to vector space of dimension `projectedFeatureDimension`. Note that the `cost` feature is only defined for the instrument nodes, while `age` and `heightAndWeight` are only defined for persons.

```
CALL gds.beta.graphSage.train(
  'persons_with_instruments',
  {
    modelName: 'multiLabelModel',
    featureProperties: ['age', 'heightAndWeight', 'cost'],
    projectedFeatureDimension: 4
  }
)
```

## Train with relationship weights

The GraphSAGE implementation supports training using relationship weights. Greater relationship weight between nodes signifies that the nodes should have more similar embedding values.

The following Cypher query trains a GraphSAGE model using relationship weights

```
CALL gds.beta.graphSage.train(
  'persons',
  {
    modelName: 'weightedTrainedModel',
    featureProperties: ['age', 'heightAndWeight'],
    relationshipWeightProperty: 'relWeight',
    nodeLabels: ['Person'],
    relationshipTypes: ['KNOWS']
  }
)
```

Train when there are no node properties present in the graph

In the case when you have a graph that does not have node properties we recommend to use existing algorithm in `mutate` mode to create node properties. Good candidates are [Centrality algorithms](#) or [Community algorithms](#).

The following example illustrates calling Degree Centrality in `mutate` mode and then using the mutated property as feature of GraphSAGE training. For the purpose of this example we are going to use the `Persons` graph, but we will not load any properties to the in-memory graph.

Create the in-memory graph without loading any node properties

```
CALL gds.graph.create(  
  'noPropertiesGraph',  
  'Person', {  
    KNOWS: {  
      type: 'KNOWS',  
      orientation: 'UNDIRECTED'  
    }  
  }  
)
```

Run `DegreeCentrality mutate` to create a new property for each node

```
CALL gds.degree.mutate(  
  'noPropertiesGraph',  
  {  
    mutateProperty: 'degree'  
  }  
) YIELD nodePropertiesWritten
```

Run GraphSAGE train using the property produced by `DegreeCentrality` as feature property

```
CALL gds.beta.graphSage.train(  
  'noPropertiesGraph',  
  {  
    modelName: 'myModel',  
    featureProperties: ['degree']  
  }  
)  
YIELD trainMillis  
RETURN trainMillis
```

`gds.degree.mutate` will create a new node property `degree` for each of the nodes in the in-memory graph, which then can be used as `featureProperty` in the `GraphSAGE.train` mode.



Using separate algorithms to produce `featureProperties` can also be very useful to capture graph topology properties.

## Stream

To generate embeddings and stream them back to the client we can use the stream mode. We must first train a model, which we do using the `gds.beta.graphSage.train` procedure.



```
CALL gds.beta.graphSage.train(
  'persons',
  {
    modelName: 'graphSage',
    featureProperties: ['age', 'heightAndWeight'],
    embeddingDimension: 3,
    randomSeed: 19
  }
)
```

Once we have trained a model (named 'graphSage') we can use it to generate and stream the embeddings.

```
CALL gds.beta.graphSage.stream(
  'persons',
  {
    modelName: 'graphSage'
  }
)
YIELD nodeId, embedding
```

Table 835. Results

nodeId	embedding
0	[0.5285002502143177, 0.4682181762801141, 0.7081378570737874]
1	[0.5285002502147674, 0.46821817628034773, 0.7081378570732975]
2	[0.5285002502143014, 0.46821817628010554, 0.7081378570738053]
3	[0.5285002502129178, 0.46821817627938667, 0.7081378570753134]
4	[0.5285002502572376, 0.46821817630241636, 0.7081378570270093]
5	[0.5285002503196665, 0.46821817633485613, 0.7081378569589678]
6	[0.528500250213112, 0.46821817627948753, 0.7081378570751017]



Due to the random initialisation of the weight variables the results may vary slightly between the runs.

## Mutate

The [model trained as part of the stream example](#) can be reused to write the results to the in-memory graph using the `mutate` mode of the procedure. Below is an example of how to achieve this.

```
CALL gds.beta.graphSage.mutate(
  'persons',
  {
    mutateProperty: 'inMemoryEmbedding',
    modelName: 'graphSage'
  }
)
YIELD
  nodeCount,
  nodePropertiesWritten
```

Table 836. Results

nodeCount	nodePropertiesWritten
7	7

Write

The [model trained as part of the stream example](#) can be reused to write the results to Neo4j. Below is an example of how to achieve this.

```
CALL gds.beta.graphSage.write(
  'persons',
  {
    writeProperty: 'embedding',
    modelName: 'graphSage'
  }
) YIELD
nodeCount,
nodePropertiesWritten
```

Table 837. Results

nodeCount	nodePropertiesWritten
7	7

## Caveats

If you are embedding a graph that has an isolated node, the aggregation step in GraphSAGE can only draw information from the node itself. When all the properties of that node are `0.0`, and the activation function is `relu`, this leads to an all-zero vector for that node. However, since GraphSAGE normalizes node embeddings using the L2-norm, and a zero vector cannot be normalized, we assign all-zero embeddings to such nodes under these special circumstances. In scenarios where you generate all-zero embeddings for orphan nodes, that may have impacts on downstream tasks such as nearest neighbor or other similarity algorithms. It may be more appropriate to filter out these disconnected nodes prior to running GraphSAGE.

When running `gds.beta.graphSage.train.estimate`, the feature dimension is computed as if each feature property is scalar.

### 7.7.3. Node2Vec Beta

Node2Vec is a node embedding algorithm that computes a vector representation of a node based on random walks in the graph. The neighborhood is sampled through random walks. Using a number of random neighborhood samples, the algorithm trains a single hidden layer neural network. The neural network is trained to predict the likelihood that a node will occur in a walk based on the occurrence of another node.

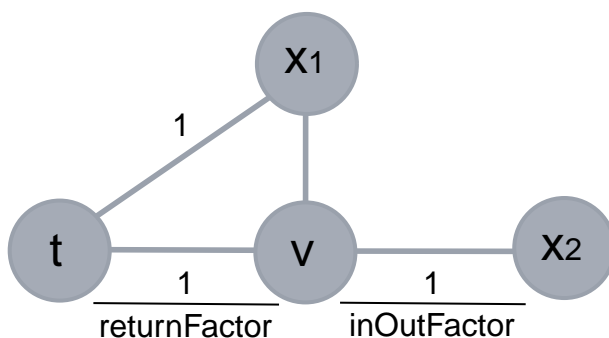
For more information on this algorithm, see:

- [Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016.](#)
- <https://snap.stanford.edu/node2vec/>

## Random Walks

A main concept of the Node2Vec algorithm are the second order random walks. A random walk simulates a traversal of the graph in which the traversed relationships are chosen at random. In a classic random walk, each relationship has the same, possibly weighted, probability of being picked. This probability is not influenced by the previously visited nodes. The concept of second order random walks, however, tries to model the transition probability based on the currently visited node  $v$ , the node  $t$  visited before the current one, and the node  $x$  which is the target of a candidate relationship. Node2Vec random walks are thus influenced by two parameters: the `returnFactor` and the `inOutFactor`:

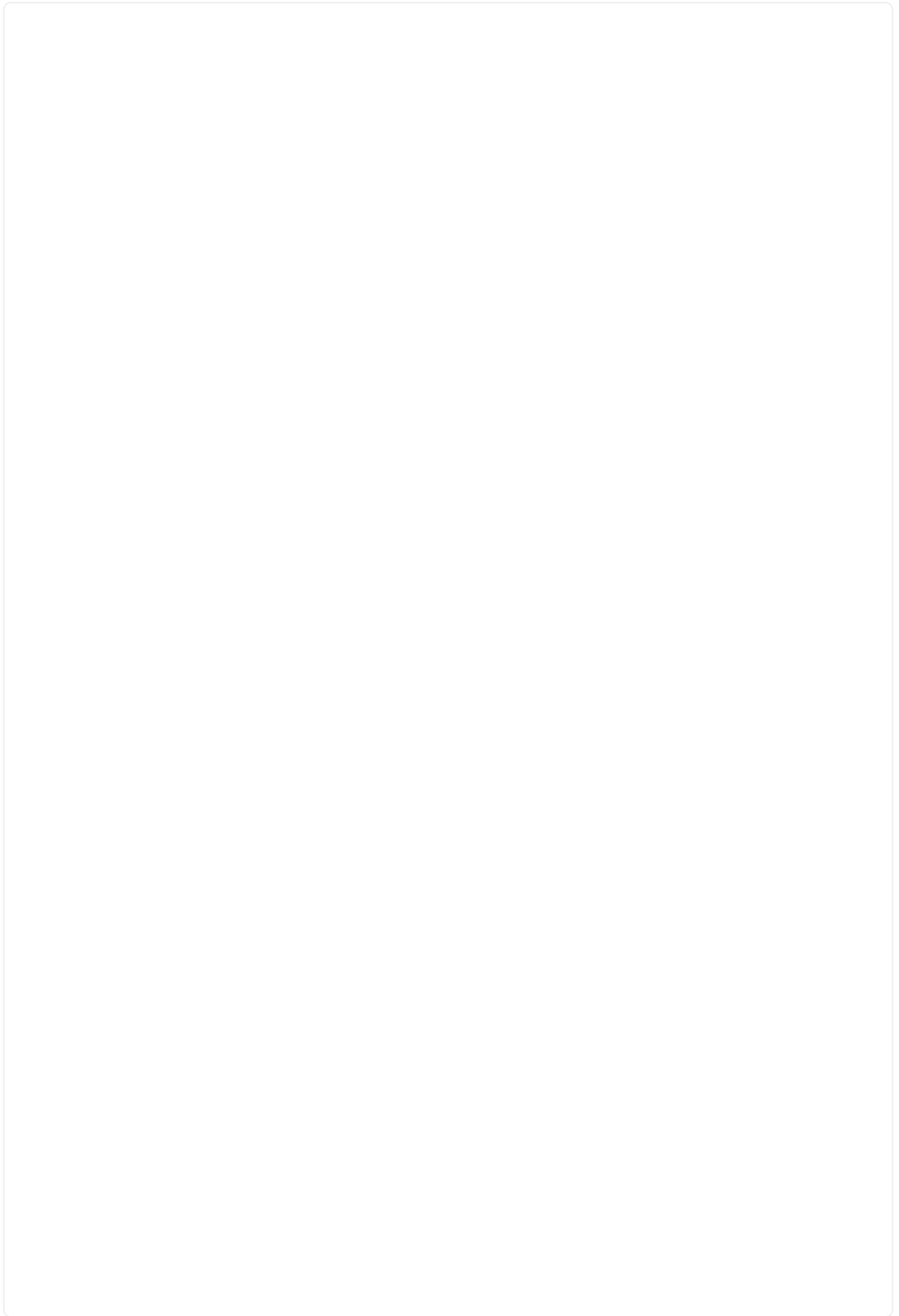
- The `returnFactor` is used if  $t$  equals  $x$ , i.e., the random walk returns to the previously visited node.
- The `inOutFactor` is used if the distance from  $t$  to  $x$  is equal to 2, i.e., the walk traverses further away from the node  $t$



The probabilities for traversing a relationship during a random walk can be further influenced by specifying a `relationshipWeightProperty`. A relationship property value greater than 1 will increase the likelihood of a relationship being traversed, a property value between 0 and 1 will decrease that probability.

For every node in the graph Node2Vec generates a series of random walks with the particular node as start node. The number of random walks per node can be influenced by the `walkPerNode` configuration parameters, the walk length is controlled by the `walkLength` parameter.

## Syntax



Run Node2Vec in stream mode on a named graph.

```
CALL gds.beta.node2vec.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  embedding: List of Float
```

Table 838. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 839. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 840. Algorithm specific configuration

Name	Type	Default	Optional	Description
walkLength	Integer	80	yes	The number of steps in a single random walk.
walksPerNode	Integer	10	yes	The number of random walks generated for each node.
inOutFactor	Float	1.0	yes	Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local.
returnFactor	Float	1.0	yes	Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be $\geq 0$ . If unspecified, the algorithm runs unweighted.
windowSize	Integer	10	yes	Size of the context window when training the neural network.
negativeSamplingRate	Integer	5	yes	Number of negative samples to produce for each positive sample.

Name	Type	Default	Optional	Description
positiveSamplingFactor	Float	0.001	yes	Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled.
negativeSamplingExponent	Float	0.75	yes	Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally.
embeddingDimension	Integer	128	yes	Size of the computed node embeddings.
iterations	Integer	1	yes	Number of training iterations.
initialLearningRate	Float	0.01	yes	Learning rate used initially for training the neural network. The learning rate decreases after each training iteration.
minLearningRate	Float	0.0001	yes	Lower bound for learning rate as it is decreased during training.
randomSeed	Integer	random	yes	Seed value used to generate the random walks, which are used as the training set of the neural network. Note, that the generated embeddings are still nondeterministic.
walkBufferSize	Integer	1000	yes	The number of random walks to complete before starting training.

Table 841. Results

Name	Type	Description
nodeId	Integer	The Neo4j node ID.
embedding	List of Float	The computed node embedding.

Run Node2Vec in mutate mode on a graph stored in the catalog.

```
CALL gds.beta.node2vec.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 842. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 843. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 844. Algorithm specific configuration

Name	Type	Default	Optional	Description
walkLength	Integer	80	yes	The number of steps in a single random walk.
walksPerNode	Integer	10	yes	The number of random walks generated for each node.
inOutFactor	Float	1.0	yes	Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local.
returnFactor	Float	1.0	yes	Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be $\geq 0$ . If unspecified, the algorithm runs unweighted.
windowSize	Integer	10	yes	Size of the context window when training the neural network.

Name	Type	Default	Optional	Description
negativeSamplingRate	Integer	5	yes	Number of negative samples to produce for each positive sample.
positiveSamplingFactor	Float	0.001	yes	Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled.
negativeSamplingExponent	Float	0.75	yes	Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally.
embeddingDimension	Integer	128	yes	Size of the computed node embeddings.
iterations	Integer	1	yes	Number of training iterations.
initialLearningRate	Float	0.01	yes	Learning rate used initially for training the neural network. The learning rate decreases after each training iteration.
minLearningRate	Float	0.0001	yes	Lower bound for learning rate as it is decreased during training.
randomSeed	Integer	random	yes	Seed value used to generate the random walks, which are used as the training set of the neural network. Note, that the generated embeddings are still nondeterministic.
walkBufferSize	Integer	1000	yes	The number of random walks to complete before starting training.

Table 845. Results

Name	Type	Description
nodeCount	Integer	The number of nodes processed.
nodePropertiesWritten	Integer	The number of node properties written.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Milliseconds for post-processing of the results.
configuration	Map	The configuration used for running the algorithm.



Run Node2Vec in write mode on a graph stored in the catalog.

```
CALL gds.beta.node2vec.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodeCount: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 846. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 847. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 848. Algorithm specific configuration

Name	Type	Default	Optional	Description
walkLength	Integer	80	yes	The number of steps in a single random walk.
walksPerNode	Integer	10	yes	The number of random walks generated for each node.
inOutFactor	Float	1.0	yes	Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local.
returnFactor	Float	1.0	yes	Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency.

Name	Type	Default	Optional	Description
<a href="#">relationshipWeightProperty</a>	String	<code>null</code>	yes	Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be $\geq 0$ . If unspecified, the algorithm runs unweighted.
windowSize	Integer	<code>10</code>	yes	Size of the context window when training the neural network.
negativeSamplingRate	Integer	<code>5</code>	yes	Number of negative samples to produce for each positive sample.
positiveSamplingFactor	Float	<code>0.001</code>	yes	Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled.
negativeSamplingExponent	Float	<code>0.75</code>	yes	Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally.
embeddingDimension	Integer	<code>128</code>	yes	Size of the computed node embeddings.
iterations	Integer	<code>1</code>	yes	Number of training iterations.
initialLearningRate	Float	<code>0.01</code>	yes	Learning rate used initially for training the neural network. The learning rate decreases after each training iteration.
minLearningRate	Float	<code>0.0001</code>	yes	Lower bound for learning rate as it is decreased during training.
randomSeed	Integer	<code>random</code>	yes	Seed value used to generate the random walks, which are used as the training set of the neural network. Note, that the generated embeddings are still nondeterministic.
walkBufferSize	Integer	<code>1000</code>	yes	The number of random walks to complete before starting training.

Table 849. Results

Name	Type	Description
nodeCount	Integer	The number of nodes processed.
nodePropertiesWritten	Integer	The number of node properties written.
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
configuration	Map	The configuration used for running the algorithm.

## Anonymous graphs

It is also possible to execute the algorithm on a graph that is projected in conjunction with the algorithm execution. In this case, the graph does not have a name, and we call it anonymous. When executing over an anonymous graph the configuration map contains a graph projection configuration as well as an algorithm configuration. All execution modes support execution on anonymous graphs, although we only show syntax and mode-specific configuration for the `write` mode for brevity.

For more information on syntax variants, see [Syntax overview](#).

Run `Node2Vec` in `write` mode on an anonymous graph.

```
CALL gds.beta.node2vec.write(  
  configuration: Map  
)  
YIELD  
  createMillis: Integer,  
  computeMillis: Integer,  
  writeMillis: Integer,  
  nodeCount: Integer,  
  nodePropertiesWritten: Integer,  
  configuration: Map
```

Table 850. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
<code>nodeProjection</code>	String, List of String or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.
<code>relationshipProjection</code>	String, List of String or Map	<code>null</code>	yes	The relationship projection used for anonymous graph creation via a Native projection.
<code>nodeQuery</code>	String	<code>null</code>	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
<code>relationshipQuery</code>	String	<code>null</code>	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
<code>nodeProperties</code>	String, List of String or Map	<code>null</code>	yes	The node properties to project during anonymous graph creation.
<code>relationshipProperties</code>	String, List of String or Map	<code>null</code>	yes	The relationship properties to project during anonymous graph creation.
<code>concurrency</code>	Integer	<code>4</code>	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency' and 'writeConcurrency'.
<code>readConcurrency</code>	Integer	value of 'concurrency'	yes	The number of concurrent threads used for creating the graph.
<code>writeConcurrency</code>	Integer	value of 'concurrency'	yes	WRITE mode only: The number of concurrent threads used for writing the result.
<code>writeProperty</code>	String	<code>n/a</code>	no	WRITE mode only: The node property to which the embedding is written to.

Table 851. Algorithm specific configuration

Name	Type	Default	Optional	Description
walkLength	Integer	80	yes	The number of steps in a single random walk.
walksPerNode	Integer	10	yes	The number of random walks generated for each node.
inOutFactor	Float	1.0	yes	Tendency of the random walk to stay close to the start node or fan out in the graph. Higher value means stay local.
returnFactor	Float	1.0	yes	Tendency of the random walk to return to the last visited node. A value below 1.0 means a higher tendency.
relationshipWeightProperty	String	null	yes	Name of the relationship property to use as weights to influence the probabilities of the random walks. The weights need to be $\geq 0$ . If unspecified, the algorithm runs unweighted.
windowSize	Integer	10	yes	Size of the context window when training the neural network.
negativeSamplingRate	Integer	5	yes	Number of negative samples to produce for each positive sample.
positiveSamplingFactor	Float	0.001	yes	Factor for influencing the distribution for positive samples. A higher value increases the probability that frequent nodes are down-sampled.
negativeSamplingExponent	Float	0.75	yes	Exponent applied to the node frequency to obtain the negative sampling distribution. A value of 1.0 samples proportionally to the frequency. A value of 0.0 samples each node equally.
embeddingDimension	Integer	128	yes	Size of the computed node embeddings.
iterations	Integer	1	yes	Number of training iterations.
initialLearningRate	Float	0.01	yes	Learning rate used initially for training the neural network. The learning rate decreases after each training iteration.
minLearningRate	Float	0.0001	yes	Lower bound for learning rate as it is decreased during training.
randomSeed	Integer	random	yes	Seed value used to generate the random walks, which are used as the training set of the neural network. Note, that the generated embeddings are still nondeterministic.
walkBufferSize	Integer	1000	yes	The number of random walks to complete before starting training.

The results are the same as for running write mode with a named graph, see the [write mode syntax above](#).

## Examples

Consider the graph created by the following Cypher statement:

```

CREATE (alice:Person {name: 'Alice'})
CREATE (bob:Person {name: 'Bob'})
CREATE (carol:Person {name: 'Carol'})
CREATE (dave:Person {name: 'Dave'})
CREATE (eve:Person {name: 'Eve'})
CREATE (guitar:Instrument {name: 'Guitar'})
CREATE (synth:Instrument {name: 'Synthesizer'})
CREATE (bongos:Instrument {name: 'Bongos'})
CREATE (trumpet:Instrument {name: 'Trumpet'})

CREATE (alice)-[:LIKES]->(guitar)
CREATE (alice)-[:LIKES]->(synth)
CREATE (alice)-[:LIKES]->(bongos)
CREATE (bob)-[:LIKES]->(guitar)
CREATE (bob)-[:LIKES]->(synth)
CREATE (carol)-[:LIKES]->(bongos)
CREATE (dave)-[:LIKES]->(guitar)
CREATE (dave)-[:LIKES]->(synth)
CREATE (dave)-[:LIKES]->(bongos);

```

```
CALL gds.graph.create('myGraph', ['Person', 'Instrument'], 'LIKES');
```

Run the Node2Vec algorithm on `myGraph`

```

CALL gds.beta.node2vec.stream('myGraph', {embeddingDimension: 2})
YIELD nodeId, embedding
RETURN nodeId, embedding

```

Table 852. Results

nodeId	embedding
0	[-0.14295829832553864, 0.08884537220001221]
1	[0.016700705513358116, 0.2253911793231964]
2	[-0.06589698046445847, 0.042405471205711365]
3	[0.05862073227763176, 0.1193704605102539]
4	[0.10888434946537018, -0.18204474449157715]
5	[0.16728264093399048, 0.14098615944385529]
6	[-0.007779224775731564, 0.02114257402718067]
7	[-0.213893860578537, 0.06195802614092827]
8	[0.2479933649301529, -0.137322798371315]

## 7.8. Machine learning models

The machine learning procedures in Neo4j GDS allow you to train supervised machine learning models. Models can then be accessed via the [Model catalog](#) and used to make predictions about your graph.

To help with working with the ML models, there are additional guides for pre-processing and hyperparameter tuning available in:

- [Pre-processing](#)
- [Tuning parameters for training](#)

The Neo4j GDS library includes the following machine learning models, grouped by quality tier:

- Alpha
  - [Node Classification](#)
  - [Node Classification Pipelines](#)
  - [Link Prediction](#)
  - [Link Prediction Pipelines](#)

### 7.8.1. Pre-processing

In most machine learning scenarios, several pre-processing steps are applied to produce data that is amenable to machine learning algorithms. This is also true for graph data. The goal of pre-processing is to provide good features for the learning algorithm. In GDS some options include:

- [Node embeddings](#)
- [Centrality algorithms](#)
- [Auxiliary algorithms](#)
  - Of special interest are [Scale Properties](#) and
  - [Split Relationships](#) for [Link Prediction](#).

### 7.8.2. Tuning parameters

Both [Node Classification](#) and [Link Prediction](#) have training parameters that can be tuned automatically given a set of allowed values. The parameters `maxEpochs`, `tolerance` and `patience` control for how long the training will run until termination. These parameters give ways to limit a computational budget. In general, higher `maxEpochs` and `patience` and lower `tolerance` lead to longer training but higher quality models. It is however well-known that restricting the computational budget can serve the purpose of regularization and mitigate overfitting.

When faced with a heavy training task, a strategy to perform hyperparameter optimization faster, is to initially use lower values for the budget related parameters while exploring better ranges for other general or algorithm specific parameters.

More precisely, `maxEpochs` is the maximum number of epochs trained until termination. Whether the training exhausted the maximum number of epochs or converged prior is reported in the neo4j debug log.

As for `patience` and `tolerance`, the former is the maximum number of consecutive epochs that do not improve the training loss at least by a `tolerance` fraction of the current loss. After `patience` such unproductive epochs, the training is terminated. In our experience, reasonable values for `patience` are in the range 1 to 3.

It is also possible, via `minEpochs`, to control a minimum number of epochs before the above termination criteria enter into play.

The training algorithm applied to the above algorithms is gradient descent. The gradient updates are computed batch-wise on batches of `batchSize` examples, and batches are computed concurrently on

**concurrency** threads. Thus, **batchSize** can affect the convergence rate, but since the algorithms above optimize convex functions, the resulting model is in theory (approximately) unique.

## 7.8.3. Node classification

### Introduction

Node Classification is a common machine learning task applied to graph: training a model to learn in which class a node belongs. There are two major classes of classification problems: binary and multiclass. In Binary-class classifications, the given dataset is categorized into two classes and in Multi-class classification, the given dataset is categorized into several classes. Neo4j GDS supports both of the above. Neo4j GDS trains supervised machine learning models based on node properties (features) in your graph to predict what class an unseen or future node would belong to. Node Classification can be used favorably together with [pre-processing algorithms](#).

Concretely, Node Classification models are used to predict a non-existing node property based on other node properties. The non-existing node property represents the class, and is referred to as the target property. The specified node properties are used as input features. The Node Classification model does not rely on relationship information. However, a node embedding algorithm could embed the neighborhoods of nodes as a node property, to transfer this information into the Node Classification model (see [Node embeddings](#)).

Models are trained on parts of the input graph and evaluated using specified metrics. Splitting of the graph into a train and a test graph is performed internally by the algorithm, and the test graph is used to evaluate model performance.

The training process follows this outline:

1. The input graph is split into two parts: the train graph and the test graph.
2. The train graph is further divided into a number of validation folds, each consisting of a train part and a validation part.
3. Each model candidate is trained on each train part and evaluated on the respective validation part.
4. The training process uses a logistic regression algorithm, and the evaluation uses the specified metrics. The first metric is the primary metric.
5. The model with the highest average score according to the primary metric will win the training.
6. The winning model will then be retrained on the entire train graph.
7. The winning model is evaluated on the train graph as well as the test graph.
8. The winning model is retrained on the entire original graph.
9. Finally, the winning model will be registered in the [Model Catalog](#).

Trained models may then be used to predict the value of the **target** property (class) of previously unseen nodes. In addition to the predicted class for each node, the predicted probability for each class may also be retained on the nodes. The order of the probabilities matches the order of the classes registered in the model.

## Metrics

The Node Classification model in the Neo4j GDS library supports the following evaluation metrics:

- Global metrics
  - `F1_WEIGHTED`
  - `F1_MACRO`
  - `ACCURACY`
- Per-class metrics
  - `F1(class=<number>)` or `F1(class=*)`
  - `PRECISION(class=<number>)` or `PRECISION(class=*)`
  - `RECALL(class=<number>)` or `RECALL(class=*)`
  - `ACCURACY(class=<number>)` or `ACCURACY(class=*)`

The `*` is syntactic sugar for reporting the metric for each class in the graph. When using a per-class metric, the reported metrics contain keys like for example `ACCURACY_class_1`.

More than one metric can be specified during training but only the first specified — the `primary` one — is used for evaluation, the results of all are present in the train results. The primary metric may not be a `*` expansion due to the ambiguity of which of the expanded metrics should be the `primary` one.

## Syntax

This section covers the syntax used to execute the Node Classification algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).





Run Node Classification in train mode on a named graph:

```
CALL gds.alpha.ml.nodeClassification.train(
  graphName: String,
  configuration: Map
) YIELD
  trainMillis: Integer,
  modelInfo: Map,
  configuration: Map
```

Table 853. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 854. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of the model to train, must not exist in the Model Catalog.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 855. Algorithm specific configuration

Name	Type	Default	Optional	Description
targetProperty	String	n/a	no	The class of the node. Must be of type Integer.
holdoutFraction	Float	n/a	no	Fraction of the graph reserved for testing. Must be in the range (0, 1).
validationFolds	Integer	n/a	no	Number of divisions of the train graph used for model selection.
metrics	List of String	n/a	no	Metrics used to evaluate the models.
params	List of Map	n/a	no	List of model configurations to be trained. See next table for details.
randomSeed	Integer	n/a	yes	Seed for the random number generator used during training.

Table 856. Model configuration

Name	Type	Default	Optional	Description
penalty	Float	0.0	yes	Penalty used for the logistic regression.
batchSize	Integer	100	yes	Number of nodes per batch.
minEpochs	Integer	1	yes	Minimum number of training epochs.
maxEpochs	Integer	100	yes	Maximum number of training epochs.
patience	Integer	1	yes	Maximum number of iterations that do not improve the loss before stopping.
tolerance	Float	0.001	yes	Minimum acceptable loss before stopping.

For hyperparameter tuning ideas, look [here](#).

Table 857. Results

Name	Type	Description
trainMillis	Integer	Milliseconds used for training.
modelInfo	Map	Information about the training and the winning model.
configuration	Map	Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the [Model List Procedure](#). The `modelInfo` return field has the following algorithm-specific subfields:

Table 858. Model info fields

Name	Type	Description
classes	List of Integer	Sorted list of class ids which are the distinct values of <code>targetProperty</code> over the entire graph.
bestParameters	Map	The model parameters which performed best on average on validation folds according to the primary metric.
metrics	Map	Map from metric description to evaluated metrics for various models and subsets of the data, see below.

The structure of `modelInfo` is:

```

{
  bestParameters: Map,           ①
  classes: List of Integer,     ②
  metrics: {                    ③
    <METRIC_NAME>: {           ④
      test: Float,             ⑤
      outerTrain: Float,      ⑥
      train: [{                ⑦
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      {
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      ...
    ],
    validation: [{             ⑧
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    {
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    ...
  ]
}
}
}

```

- ① The best scoring model candidate configuration.
- ② Sorted list of class ids which are the distinct values of `targetProperty` over the entire graph.
- ③ The `metrics` map contains an entry for each metric description, and the corresponding results for that metric.
- ④ A metric name specified in the configuration of the procedure, e.g., `F1_MACRO` or `RECALL(class=4)`.
- ⑤ Numeric value for the evaluation of the best model on the test set.
- ⑥ Numeric value for the evaluation of the best model on the outer train set.
- ⑦ The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.
- ⑧ The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.

### Run Node Classification in stream mode on a named graph:

```
CALL gds.alpha.ml.nodeClassification.predict.stream(
  graphName: String,
  configuration: Map
) YIELD
  nodeId: Integer,
  predictedClass: Integer,
  predictedProbabilities: List of Float
```

Table 859. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 860. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 861. Algorithm specific configuration

Name	Type	Default	Optional	Description
includePredictedProbabilities	Boolean	false	yes	Whether to return the probability for each class. If false then null is returned in predictedProbabilities.
batchSize	Integer	100	yes	Number of nodes per batch.

Table 862. Results

Name	Type	Description
nodeId	Integer	Node ID.
predictedClass	Integer	Predicted class for this node.
predictedProbabilities	List of Float	Probabilities for all classes, for this node.

Run Node Classification in mutate mode on a named graph:

```
CALL gds.alpha.ml.nodeClassification.predict.mutate(
  graphName: String,
  configuration: Map
) YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 863. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 864. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 865. Algorithm specific configuration

Name	Type	Default	Optional	Description
predictedProbabilityProperty	String	n/a	yes	The node property in which the class probability list is stored. If omitted, the probability list is discarded.
batchSize	Integer	100	yes	Number of nodes per batch.

Table 866. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
nodePropertiesWritten	Integer	Number of node properties written.

Name	Type	Description
configuration	Map	Configuration used for running the algorithm.

### Run Node Classification in write mode on a named graph:

```
CALL gds.alpha.ml.nodeClassification.predict.write(
  graphName: String,
  configuration: Map
) YIELD
  createMillis: Integer,
  computeMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 867. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 868. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 869. Algorithm specific configuration

Name	Type	Default	Optional	Description
predictedProbabilityProperty	String	n/a	yes	The node property in which the class probability list is stored. If omitted, the probability list is discarded.
batchSize	Integer	100	yes	Number of nodes per batch.

Table 870. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
writeMillis	Integer	Milliseconds for writing result back to Neo4j.



Name	Type	Description
nodePropertiesWritten	Integer	Number of relationships created.
configuration	Map	Configuration used for running the algorithm.

## Examples

In this section we will show examples of training a Node Classification Model on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the model in a real setting. We will do this on a small graph of a handful of nodes representing houses. This is an example of Multi-class classification, the `class` node property distinct values determine the number of classes, in this case three (0, 1 and 2). The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(:House {color: 'Gold', sizePerStory: [15.5, 23.6, 33.1], class: 0}),
(:House {color: 'Red', sizePerStory: [15.5, 23.6, 100.0], class: 0}),
(:House {color: 'Blue', sizePerStory: [11.3, 35.1, 22.0], class: 0}),
(:House {color: 'Green', sizePerStory: [23.2, 55.1, 0.0], class: 1}),
(:House {color: 'Gray', sizePerStory: [34.3, 24.0, 0.0], class: 1}),
(:House {color: 'Black', sizePerStory: [71.66, 55.0, 0.0], class: 1}),
(:House {color: 'White', sizePerStory: [11.1, 111.0, 0.0], class: 1}),
(:House {color: 'Teal', sizePerStory: [80.8, 0.0, 0.0], class: 2}),
(:House {color: 'Beige', sizePerStory: [106.2, 0.0, 0.0], class: 2}),
(:House {color: 'Magenta', sizePerStory: [99.9, 0.0, 0.0], class: 2}),
(:House {color: 'Purple', sizePerStory: [56.5, 0.0, 0.0], class: 2}),
(:UnknownHouse {color: 'Pink', sizePerStory: [23.2, 55.1, 56.1]}),
(:UnknownHouse {color: 'Tan', sizePerStory: [22.32, 102.0, 0.0]}),
(:UnknownHouse {color: 'Yellow', sizePerStory: [39.0, 0.0, 0.0]});
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `House` and `UnknownHouse` labels. We will also project the `sizeOfStory` property to use as a model feature, and the `class` property to use as a target feature.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create('myGraph', {
  House: { properties: ['sizePerStory', 'class'] },
  UnknownHouse: { properties: 'sizePerStory' }
},
'x'
)
```

In the following examples we will demonstrate using the Node Classification model on this graph.

## Memory Estimation

First off, we will estimate the cost of running the algorithm using the `estimate` procedure. This can be done with any execution mode. We will use the `train` mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Automatic estimation and execution blocking](#).

For more details on `estimate` in general, see [Memory Estimation](#).

The following will estimate the memory requirements for running the algorithm in write mode:

```
CALL gds.alpha.ml.nodeClassification.train.estimate('myGraph', {
  nodeLabels: ['House'],
  modelName: 'nc-model',
  featureProperties: ['sizePerStory'],
  targetProperty: 'class',
  randomSeed: 2,
  holdoutFraction: 0.2,
  validationFolds: 5,
  metrics: [ 'F1_WEIGHTED' ],
  params: [
    {penalty: 0.0625},
    {penalty: 0.5},
    {penalty: 1.0},
    {penalty: 4.0}
  ]
})
YIELD bytesMin, bytesMax, requiredMemory
```

Table 871. Results

bytesMin	bytesMax	requiredMemory
66874376	66906336	"[63 MiB ... 63 MiB]"

## Train

In this example we will train a model to predict the class in which a house belongs, based on its `sizePerStory` property.

## Train a Node Classification model:

```
CALL gds.alpha.ml.nodeClassification.train('myGraph', {
  nodeLabels: ['House'],
  modelName: 'nc-model',
  featureProperties: ['sizePerStory'],
  targetProperty: 'class',
  randomSeed: 2,
  holdoutFraction: 0.2,
  validationFolds: 5,
  metrics: [ 'F1_WEIGHTED' ],
  params: [
    {penalty: 0.0625},
    {penalty: 0.5},
    {penalty: 1.0},
    {penalty: 4.0}
  ]
}) YIELD modelInfo
RETURN
{penalty: modelInfo.bestParameters.penalty} AS winningModel,
modelInfo.metrics.F1_WEIGHTED.outerTrain AS trainGraphScore,
modelInfo.metrics.F1_WEIGHTED.test AS testGraphScore
```

Table 872. Results

winningModel	trainGraphScore	testGraphScore
{penalty=0.0625}	0.999999990909091	0.6363636286363638

Here we can observe that the model candidate with penalty `0.0625` performed the best in the training phase, with a score of almost 100% over the train graph. On the test graph, the model scores a bit lower at about 64%. This indicates that the model reacted very well to the train graph, and was able to generalize fairly well to unseen data. In order to achieve a higher test score, we may need to use better features, a larger graph, or different model configuration.

## Stream

In the `stream` execution mode, the algorithm returns the predicted property for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

In this example we will show how to use a trained model to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the [train example](#) which we gave the name `'nc-model'`.

```
CALL gds.alpha.ml.nodeClassification.predict.stream('myGraph', {
  nodeLabels: ['House', 'UnknownHouse'],
  modelName: 'nc-model',
  includePredictedProbabilities: true
}) YIELD nodeId, predictedClass, predictedProbabilities
WITH gds.util.asNode(nodeId) AS houseNode, predictedClass, predictedProbabilities
WHERE houseNode:UnknownHouse
RETURN
  houseNode.color AS classifiedHouse,
  predictedClass,
  floor(predictedProbabilities[predictedClass] * 100) AS confidence
ORDER BY classifiedHouse
```

Table 873. Results

classifiedHouse	predictedClass	confidence
"Pink"	0	98.0
"Tan"	1	98.0
"Yellow"	2	79.0

As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the confidence is  $\geq 79\%$  in all cases.

## Mutate

The `mutate` execution mode updates the named graph with a new node property containing the predicted class for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row including information about timings and how many properties were written. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

In this example we will show how to use a trained model to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the [train example](#) which we gave the name `'nc-model'`.

```
CALL gds.alpha.ml.nodeClassification.predict.mutate('myGraph', {
  nodeLabels: ['House', 'UnknownHouse'],
  modelName: 'nc-model',
  mutateProperty: 'predictedClass',
  predictedProbabilityProperty: 'predictedProbabilities'
}) YIELD nodePropertiesWritten
```

Table 874. Results

nodePropertiesWritten
28

Since we specified also the `predictedProbabilityProperty` we are writing two properties for each of the 14 nodes. In order to analyse our predicted classes we stream the properties from the in-memory graph:

```
CALL gds.graph.streamNodeProperties(
  'myGraph', ['predictedProbabilities', 'predictedClass'], ['UnknownHouse']
) YIELD nodeId, nodeProperty, propertyValue
RETURN gds.util.asNode(nodeId).color AS classifiedHouse, nodeProperty, propertyValue
ORDER BY classifiedHouse, nodeProperty
```

Table 875. Results

classifiedHouse	nodeProperty	propertyValue
"Pink"	"predictedClass"	0
"Pink"	"predictedProbabilities"	[0.9866455686217779, 0.01311656378786989, 2.3786759035214687E-4]
"Tan"	"predictedClass"	1
"Tan"	"predictedProbabilities"	[0.01749164563726576, 0.9824922482993587, 1.610606337562594E-5]
"Yellow"	"predictedClass"	2
"Yellow"	"predictedProbabilities"	[0.0385634113659007, 0.16350471177895198, 0.7979318768551473]

As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the highest class probability is >75% in all cases.

## Write

The `write` execution mode writes the predicted property for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row including information about timings and how many properties were written. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

In this example we will show how to use a trained model to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the [train example](#) which we gave the name `'nc-model'`.

```
CALL gds.alpha.ml.nodeClassification.predict.write('myGraph', {
  nodeLabels: ['House', 'UnknownHouse'],
  modelName: 'nc-model',
  writeProperty: 'predictedClass',
  predictedProbabilityProperty: 'predictedProbabilities'
}) YIELD nodePropertiesWritten
```

Table 876. Results

nodePropertiesWritten
28

Since we specified also the `predictedProbabilityProperty` we are writing two properties for each of the

14 nodes. In order to analyse our predicted classes we stream the properties from the in-memory graph:

```
MATCH (house:UnknownHouse)
RETURN house.color AS classifiedHouse, house.predictedClass AS predictedClass,
house.predictedProbabilities AS predictedProbabilities
```

Table 877. Results

classifiedHouse	predictedClass	predictedProbabilities
"Pink"	0	[0.9866455686217779, 0.01311656378786989, 2.3786759035214687E-4]
"Tan"	1	[0.01749164563726576, 0.9824922482993587, 1.610606337562594E-5]
"Yellow"	2	[0.0385634113659007, 0.16350471177895198, 0.7979318768551473]

As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the highest class probability is >75% in all cases.

## 7.8.4. Node classification pipelines

### Introduction

Node Classification is a common machine learning task applied to graphs: training models to classify nodes. The GDS library also provides a standalone version of [Node Classification](#). Here we describe Node Classification Pipelines, which facilitate an end-to-end workflow, from features extraction to node classification. There are two kinds of pipelines: training pipelines and classification pipelines, both of which reside in the [model catalog](#). When a training pipeline is executed, a classification pipeline is created and stored in the model catalog.

A training pipeline is a sequence of two phases:

- I. The graph is augmented with new node properties in a series of steps.
- II. The augmented graph is used for training a node classification model.

One can [configure](#) which steps should be included above. The steps execute GDS algorithms that create new node properties. After configuring the node property steps, one can [select](#) a subset of node properties to be used as features. The training phase (II) proceeds in a manner akin to the standalone version of [Node Classification](#), where it can train multiple models, select the best one, and report relevant performance metrics.

After [training the pipeline](#), a classification pipeline is created. This new pipeline inherits the node property steps and feature configuration from the training pipeline and uses them to generate the relevant features

for classifying unlabeled nodes.



**Classification** can only be done with a trained classification pipeline (not with a training pipeline).

The motivation for using pipelines:

- easier to get splits right and prevent data leakage
- ensuring that the same feature creation steps are applied at classification and train time
- applying the trained model with a single procedure call
- persisting the pipeline as a whole

The rest of this page is divided as follows:

- [Creating a pipeline](#)
- [Adding node properties](#)
- [Adding features](#)
- [Configuring the node splits](#)
- [Configuring the model parameters](#)
- [Training the pipeline](#)
- [Applying a classification pipeline to make predictions](#)

## Creating a pipeline

The first step of building a new pipeline is to create one using `gds.alpha.ml.pipeline.nodeClassification.create`. This stores a trainable model object in the model catalog of type `Node classification training pipeline`. This represents a configurable pipeline that can later be invoked for training, which in turn creates a classification pipeline. The latter is also a model which is stored in the catalog with type `Node classification pipeline`.

## Syntax

### Create pipeline syntax

```
CALL gds.alpha.ml.pipeline.nodeClassification.create(  
  pipelineName: String  
)  
YIELD  
  name: String,  
  nodePropertySteps: List of Map,  
  featureProperties: List of String,  
  splitConfig: Map,  
  parameterSpace: List of Map
```

Table 878. Parameters

Name	Type	Description
pipelineName	String	The name of the created pipeline.

Table 879. Results

Name	Type	Description
name	String	Name of the pipeline.
nodeProperty Steps	List of Map	List of configurations for node property steps.
featureProperties	List of String	List of node properties to be used as features.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will create a pipeline:

```
CALL gds.alpha.ml.pipeline.nodeClassification.create('pipe')
```

Table 880. Results

name	nodePropertySteps	featureProperties	splitConfig	parameterSpace
"pipe"	[]	[]	{testFraction=0.3, validationFolds=3}	[[maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001]]

This shows that the newly created pipeline does not contain any steps yet, and has defaults for the split and train parameters.

## Adding node properties

A node classification pipeline can execute one or several GDS algorithms in mutate mode that create node properties in the in-memory graph. Such steps producing node properties can be chained one after another and created properties can later be used as [features](#). Moreover, the node property steps that are added to the training pipeline will be executed both when [training](#) a model and when the classification pipeline is [applied for classification](#).

The name of the procedure that should be added can be a fully qualified GDS procedure name ending with `.mutate`. The ending `.mutate` may be omitted and one may also use shorthand forms such as `node2vec` instead of `gds.beta.node2vec.mutate`.



For example, [pre-processing algorithms](#) can be used as node property steps.

## Syntax

### Add node property syntax

```
CALL gds.alpha.ml.pipeline.nodeClassification.addNodeProperty(  
  pipelineName: String,  
  procedureName: String,  
  procedureConfiguration: Map  
)  
YIELD  
  name: String,  
  nodePropertySteps: List of Map,  
  featureProperties: List of String,  
  splitConfig: Map,  
  parameterSpace: List of Map
```

Table 881. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
procedureName	String	The name of the procedure to be added to the pipeline.
procedureConfiguration	Map	The configuration of the procedure, excluding <code>graphName</code> , <code>nodeLabels</code> and <code>relationshipTypes</code> .

Table 882. Results

Name	Type	Description
name	String	Name of the pipeline.
nodePropertySteps	List of Map	List of configurations for node property steps.
featureProperties	List of String	List of node properties to be used as features.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will add a node property step to the pipeline. Here we assume that the input graph contains a property `sizePerStory`.

```
CALL gds.alpha.ml.pipeline.nodeClassification.addNodeProperty('pipe', 'scaleProperties', {  
  nodeProperties: 'sizePerStory',  
  scaler: 'L1Norm',  
  mutateProperty: 'scaledSizes'  
})  
YIELD name, nodePropertySteps
```

Table 883. Results

name	nodePropertySteps
"pipe"	[[name=gds.alpha.scaleProperties.mutate, config={scaler=L1Norm, mutateProperty=scaledSizes, nodeProperties=sizePerStory}]]

The `scaledSizes` property can be later used as a feature.

## Adding features

A Node Classification Pipeline allows you to select a subset of the available node properties to be used as features for the machine learning model. When executing the pipeline, the selected `nodeProperties` must be either present in the input graph, or created by a previous node property step. For example, the `embedding` property could be created by the previous example, and we expect `numberOfPosts` to already be present in the in-memory graph used as input, at train and predict time.

## Syntax

### Adding a feature to a pipeline syntax

```
CALL gds.alpha.ml.pipeline.nodeClassification.selectFeatures(
  pipelineName: String,
  nodeProperties: List or String
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureProperties: List of String,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 884. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
nodeProperties	List or String	Configuration for splitting the relationships.

Table 885. Results

Name	Type	Description
name	String	Name of the pipeline.
nodeProperty Steps	List of Map	List of configurations for node property steps.
featureProperties	List of String	List of node properties to be used as features.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will select features for the pipeline. Here we assume that the input graph contains a property `sizePerStory` and `scaledSizes` was created in a `nodePropertyStep`.

```
CALL gds.alpha.ml.pipeline.nodeClassification.selectFeatures('pipe', ['scaledSizes', 'sizePerStory'])
YIELD name, featureProperties
```

Table 886. Results

name	featureProperties
"pipe"	[scaledSizes, sizePerStory]

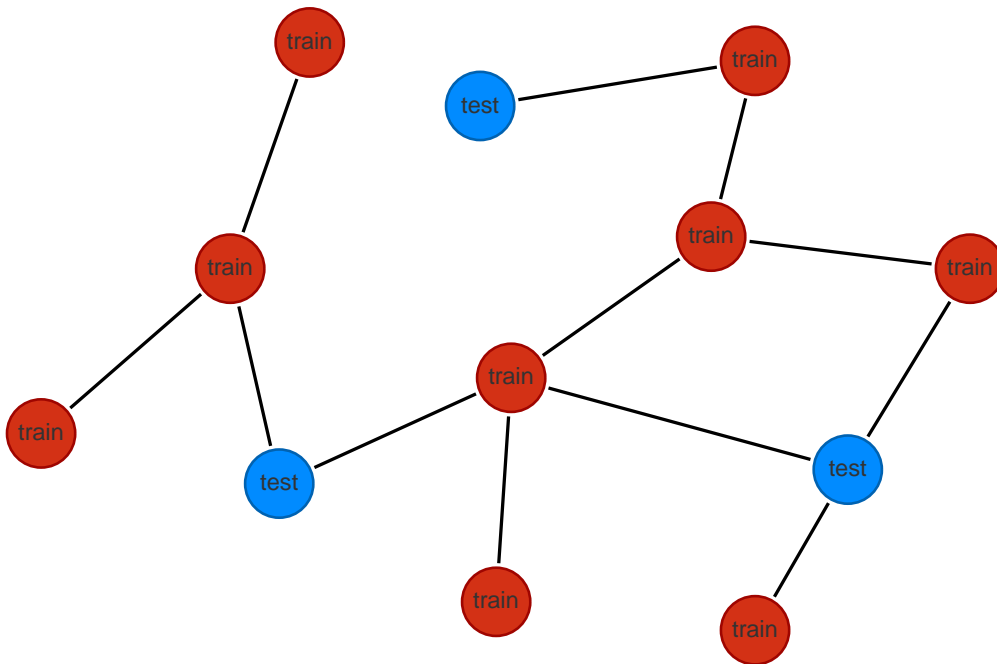
## Configuring the node splits

Node Classification Pipelines manage splitting the nodes into several sets for training, testing and validating the models defined in the [parameter space](#). Configuring the splitting is optional, and if omitted, splitting will be done using default settings. The splitting configuration of a pipeline can be inspected by using `gds.beta.model.list` and possibly only yielding `splitConfig`.

The node splits are used in the training process as follows:

1. The input graph is split into two parts: the train graph and the test graph. See the [example below](#).
2. The train graph is further divided into a number of validation folds, each consisting of a train part and a validation part. See the [animation below](#).
3. Each model candidate is trained on each train part and evaluated on the respective validation part.
4. The model with the highest average score according to the primary metric will win the training.
5. The winning model will then be retrained on the entire train graph.
6. The winning model is evaluated on the train graph as well as the test graph.
7. The winning model is retrained on the entire original graph.

Below we illustrate an example for a graph with 12 nodes. First we use a `holdoutFraction` of 0.25 to split into train and test subgraphs.



Then we carry out three validation folds, where we first split the train subgraph into 3 disjoint subsets (s1, s2 and s3), and then alternate which subset is used for validation. For each fold, all candidate models are trained in the red nodes, and validated in the green nodes.

[validation-folds-image] | [train-test-splitting/validation-folds-node-classification.gif](#)

## Syntax

Configure the node split syntax

```
CALL gds.alpha.ml.pipeline.nodeClassification.configureSplit(
  pipelineName: String,
  configuration: Map
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureProperties: List of Strings,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 887. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
configuration	Map	Configuration for splitting the relationships.

Table 888. Configuration

Name	Type	Default	Description
validationFolds	Integer	3	Number of divisions of the training graph used during <a href="#">model selection</a> .
testFraction	Double	0.3	Fraction of the graph reserved for testing. Must be in the range (0, 1). The fraction used for the training is $1 - \text{testFraction}$ .

Table 889. Results

Name	Type	Description
name	String	Name of the pipeline.
nodeProperty Steps	List of Map	List of configurations for node property steps.
featureProperties	List of String	List of node properties to be used as features.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will configure the splitting of the pipeline:

```
CALL gds.alpha.ml.pipeline.nodeClassification.configureSplit('pipe', {
  testFraction: 0.2,
  validationFolds: 5
})
YIELD splitConfig
```

Table 890. Results

splitConfig
{testFraction=0.2, validationFolds=5}

We now reconfigured the splitting of the pipeline, which will be applied during [training](#).

## Configuring the model parameters

The `gds.alpha.ml.pipeline.nodeClassification.configureParams` mode is used to set up the train mode with a list of configurations of logistic regression models. The set of model configurations is called the *parameter space* which parametrizes a set of model candidates. The parameter space can be configured by passing this procedure a list of maps, where each map configures the training of one logistic regression model. In [Training the pipeline](#), we explain further how the configured model candidates are trained, evaluated and compared.

The allowed model parameters are listed in the table [Model configuration](#).

If `configureParams` is not used, then a single model with defaults for all the model parameters is used. The parameter space of a pipeline can be inspected using `gds.beta.model.list` and optionally yielding only `parameterSpace`.

## Syntax

## Configure the train parameters syntax

```
CALL gds.alpha.ml.pipeline.nodeClassification.configureParams(
  pipelineName: String,
  parameterSpace: List of Map
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureProperties: List of String,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 891. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
parameterSpace	List of Map	The parameter space used to select the best model from. Each Map corresponds to potential model. The allowed parameters for a model are defined in the next table.

Table 892. Model configuration

Name	Type	Default	Optional	Description
penalty	Float	0.0	yes	Penalty used for the logistic regression. By default, no penalty is applied.
batchSize	Integer	100	yes	Number of nodes per batch.
minEpochs	Integer	1	yes	Minimum number of training epochs.
maxEpochs	Integer	100	yes	Maximum number of training epochs.
patience	Integer	1	yes	Maximum number of unproductive consecutive epochs.
tolerance	Float	0.001	yes	The minimal improvement of the loss to be considered productive.

Table 893. Results

Name	Type	Description
name	String	Name of the pipeline.
nodePropertySteps	List of Map	List of configurations for node property steps.
featureProperties	List of String	List of node properties to be used as features.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will configure the parameter space of the pipeline:

```
CALL gds.alpha.ml.pipeline.nodeClassification.configureParams('pipe',
  [{penalty: 0.0625}, {tolerance: 0.01}, {maxEpochs: 500}]
) YIELD parameterSpace
```

Table 894. Results

parameterSpace
[[{maxEpochs=100, minEpochs=1, penalty=0.0625, patience=1, batchSize=100, tolerance=0.001}, {maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.01}, {maxEpochs=500, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001}]]

The `parameterSpace` in the pipeline now contains the three different model parameters, expanded with the default values. Each specified model configuration will be tried out during the model selection in [training](#).

## Training the pipeline

The train mode, `gds.alpha.ml.pipeline.nodeClassification.train`, is responsible for splitting data, feature extraction, model selection, training and storing a model for future use. Running this mode results in a classification pipeline of type `Node classification pipeline`, which is then stored in the [model catalog](#). The classification pipeline can be [applied](#) to a possibly different graph which classifies nodes.

More precisely, the training proceeds as follows:

1. Apply `nodeLabels` and `relationshipType` filters to the graph.
2. Apply the node property steps, added according to [Adding node properties](#), on the whole graph.
3. Select node properties to be used as features, as specified in [Adding features](#).
4. Split the input graph into two parts: the train graph and the test graph. This is described in [Configuring the node splits](#). These graphs are internally managed and exist only for the duration of the training.
5. Split the nodes in the train graph using stratified k-fold cross-validation. The number of folds `k` can be configured as described in [Configuring the node splits](#).
6. Each model candidate defined in the `parameter space` is trained on each train set and evaluated on the respective validation set for every fold. The training process uses a logistic regression algorithm, and the evaluation uses the specified `metric`.
7. Choose the best performing model according to the highest average score for the primary metric.
8. Retrain the winning model on the entire train graph.
9. Evaluate the performance of the winning model on the whole train graph as well as the test graph.
10. Retrain the winning model on the entire original graph.
11. Register the winning model in the [Model Catalog](#).



The above steps describe what the procedure does logically. The actual steps as well as their ordering in the implementation may differ.



A step can only use node properties that are already present in the input graph or produced by steps, which were added before.

## Metrics

The Node Classification model in the Neo4j GDS library supports the following evaluation metrics:

- Global metrics
  - `F1_WEIGHTED`
  - `F1_MACRO`
  - `ACCURACY`
- Per-class metrics
  - `F1(class=<number>)` or `F1(class=*)`
  - `PRECISION(class=<number>)` or `PRECISION(class=*)`
  - `RECALL(class=<number>)` or `RECALL(class=*)`
  - `ACCURACY(class=<number>)` or `ACCURACY(class=*)`

The `*` is syntactic sugar for reporting the metric for each class in the graph. When using a per-class metric, the reported metrics contain keys like for example `ACCURACY_class_1`.

More than one metric can be specified during training but only the first specified — the `primary` one — is used for evaluation, the results of all are present in the train results. The primary metric may not be a `*` expansion due to the ambiguity of which of the expanded metrics should be the `primary` one.

## Syntax

Run Node Classification in train mode on a named graph:

```
CALL gds.alpha.ml.pipeline.nodeClassification.train(
  graphName: String,
  configuration: Map
) YIELD
  trainMillis: Integer,
  modelInfo: Map,
  configuration: Map
```

Table 895. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 896. Configuration



Name	Type	Default	Optional	Description
pipeline	String	n/a	no	The name of the pipeline to execute.
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
targetProperty	String	n/a	no	The class of the node. Must be of type Integer.
metrics	List of String	n/a	no	<a href="#">Metrics</a> used to evaluate the models.
randomSeed	Integer	n/a	yes	Seed for the random number generator used during training.
modelName	String	n/a	no	The name of the model to train, must not exist in the Model Catalog.

Table 897. Results

Name	Type	Description
trainMillis	Integer	Milliseconds used for training.
modelInfo	Map	Information about the training and the winning model.
configuration	Map	Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the [Model List Procedure](#). The `modelInfo` return field has the following algorithm-specific subfields:

Table 898. Model info fields

Name	Type	Description
classes	List of Integer	Sorted list of class ids which are the distinct values of <code>targetProperty</code> over the entire graph.
bestParameters	Map	The model parameters which performed best on average on validation folds according to the primary metric.
metrics	Map	Map from metric description to evaluated metrics for various models and subsets of the data, see below.
trainingPipeline	Map	The pipeline used for the training.

The structure of `modelInfo` is:

```

{
  bestParameters: Map,           ①
  trainingPipeline: Map         ②
  classes: List of Integer,     ③
  metrics: {                    ④
    <METRIC_NAME>: {           ⑤
      test: Float,             ⑥
      outerTrain: Float,       ⑦
      train: [{                ⑧
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      }],
      {
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      ...
    ],
    validation: [{             ⑨
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    }],
    {
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    ...
  ]
}
}

```

- ① The best scoring model candidate configuration.
- ② The pipeline used for the training.
- ③ Sorted list of class ids which are the distinct values of `targetProperty` over the entire graph.
- ④ The `metrics` map contains an entry for each metric description, and the corresponding results for that metric.
- ⑤ A metric name specified in the configuration of the procedure, e.g., `F1_MACRO` or `RECALL(class=4)`.
- ⑥ Numeric value for the evaluation of the winning model on the test set.
- ⑦ Numeric value for the evaluation of the winning model on the outer train set.
- ⑧ The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.
- ⑨ The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.

## Example

In this section we will show examples of running a Node Classification training pipeline on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the model in a real setting. We will do this on a small graph of a handful of nodes representing houses.

This is an example of Multi-class classification, the `class` node property distinct values determine the number of classes, in this case three (0, 1 and 2). The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
(:House {color: 'Gold', sizePerStory: [15.5, 23.6, 33.1], class: 0}),
(:House {color: 'Red', sizePerStory: [15.5, 23.6, 100.0], class: 0}),
(:House {color: 'Blue', sizePerStory: [11.3, 35.1, 22.0], class: 0}),
(:House {color: 'Green', sizePerStory: [23.2, 55.1, 0.0], class: 1}),
(:House {color: 'Gray', sizePerStory: [34.3, 24.0, 0.0], class: 1}),
(:House {color: 'Black', sizePerStory: [71.66, 55.0, 0.0], class: 1}),
(:House {color: 'White', sizePerStory: [11.1, 111.0, 0.0], class: 1}),
(:House {color: 'Teal', sizePerStory: [80.8, 0.0, 0.0], class: 2}),
(:House {color: 'Beige', sizePerStory: [106.2, 0.0, 0.0], class: 2}),
(:House {color: 'Magenta', sizePerStory: [99.9, 0.0, 0.0], class: 2}),
(:House {color: 'Purple', sizePerStory: [56.5, 0.0, 0.0], class: 2}),
(:UnknownHouse {color: 'Pink', sizePerStory: [23.2, 55.1, 56.1]}),
(:UnknownHouse {color: 'Tan', sizePerStory: [22.32, 102.0, 0.0]}),
(:UnknownHouse {color: 'Yellow', sizePerStory: [39.0, 0.0, 0.0]});
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for the pipeline execution. We do this using a native projection targeting the `House` and `UnknownHouse` labels. We will also project the `sizeOfStory` property to use as a model feature, and the `class` property to use as a target feature.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create('myGraph', {
  House: { properties: ['sizePerStory', 'class'] },
  UnknownHouse: { properties: 'sizePerStory' }
},
'*)
)
```

In the following examples we will demonstrate running the Node Classification training pipeline on this graph. We will train a model to predict the class in which a house belongs, based on its `sizePerStory` property.

The following will train a model using a pipeline:

```
CALL gds.alpha.ml.pipeline.nodeClassification.train('myGraph', {
  pipeline: 'pipe',
  nodeLabels: ['House'],
  modelName: 'nc-pipeline-model',
  targetProperty: 'class',
  randomSeed: 42,
  concurrency: 1,
  metrics: ['F1_WEIGHTED']
}) YIELD modelInfo
RETURN
modelInfo.bestParameters AS winningModel,
modelInfo.metrics.F1_WEIGHTED.outerTrain AS trainGraphScore,
modelInfo.metrics.F1_WEIGHTED.test AS testGraphScore
```

Table 899. Results

winningModel	trainGraphScore	testGraphScore
{maxEpochs=100, minEpochs=1, penalty=0.0625, patience=1, batchSize=100, tolerance=0.001}	0.9999999912121211	0.9999999850000002

Here we can observe that the model candidate with penalty `0.0625` performed the best in the training phase, with an `F1_WEIGHTED` score nearing 1 over the train graph as well as on the test graph. This indicates that the model reacted very well to the train graph, and was able to generalize fairly well to unseen data. Notice that this is just a toy example on a very small graph. In order to achieve a higher test score, we may need to use better features, a larger graph, or different model configuration.

## Applying a trained model for prediction

In the previous sections we have seen how to build up a Node Classification training pipeline and train it to produce a classification pipeline. After [training](#), the runnable model is of type `Node classification pipeline` and resides in the [model catalog](#).

The classification pipeline can be executed with a graph in the graph catalog to predict the value of the `target` property (class) of previously unseen nodes. In addition to the predicted class for each node, the predicted probability for each class may also be retained on the nodes. The order of the probabilities matches the order of the classes registered in the model.

Since the model has been trained on features which are created using the feature pipeline, the same feature pipeline is stored within the model and executed at prediction time. As during training, intermediate node properties created by the node property steps in the feature pipeline are transient and not visible after execution.

The predict graph must contain the properties that the pipeline requires and the used array properties must have the same dimensions as in the train graph. If the predict and train graphs are distinct, it is also beneficial that they have similar origins and semantics, so that the model is able to generalize well.

## Syntax

Run Node Classification in stream mode on a named graph:

```
CALL gds.alpha.ml.pipeline.nodeClassification.predict.stream(
  graphName: String,
  configuration: Map
)
YIELD
  nodeId: Integer,
  predictedClass: Integer,
  predictedProbabilities: List of Float
```

Table 900. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 901. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 902. Algorithm specific configuration

Name	Type	Default	Optional	Description
includePredictedProbabilities	Boolean	false	yes	Whether to return the probability for each class. If false then null is returned in predictedProbabilities.
batchSize	Integer	100	yes	Number of nodes per batch.

Table 903. Results

Name	Type	Description
nodeId	Integer	Node ID.
predictedClass	Integer	Predicted class for this node.
predictedProbabilities	List of Float	Probabilities for all classes, for this node.

Run Node Classification in mutate mode on a named graph:

```
CALL gds.alpha.ml.pipeline.nodeClassification.predict.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 904. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 905. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 906. Algorithm specific configuration

Name	Type	Default	Optional	Description
predictedProbabilityProperty	String	n/a	yes	The node property in which the class probability list is stored. If omitted, the probability list is discarded.
batchSize	Integer	100	yes	Number of nodes per batch.

Table 907. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.

Name	Type	Description
nodePropertiesWritten	Integer	Number of node properties written.
configuration	Map	Configuration used for running the algorithm.

Run Node Classification in write mode on a named graph:

```
CALL gds.alpha.ml.pipeline.nodeClassification.predict.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 908. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 909. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'writeConcurrency'.
writeConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for writing the result to Neo4j.

Table 910. Algorithm specific configuration

Name	Type	Default	Optional	Description
predictedProbabilityProperty	String	n/a	yes	The node property in which the class probability list is stored. If omitted, the probability list is discarded.
batchSize	Integer	100	yes	Number of nodes per batch.

Table 911. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMilliseconds	Integer	Milliseconds for running the algorithm.



Name	Type	Description
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
writeMillis	Integer	Milliseconds for writing result back to Neo4j.
nodePropertiesWritten	Integer	Number of node properties written.
configuration	Map	Configuration used for running the algorithm.

## Example

In the following examples we will show how to use a classification pipeline to predict the class of a node in your in-memory graph. In addition to the predicted class, we will also produce the probability for each class in another node property. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the [train example](#) which we gave the name 'nc-pipeline-model'.

## Stream

```
CALL gds.alpha.ml.pipeline.nodeClassification.predict.stream('myGraph', {
  modelName: 'nc-pipeline-model',
  includePredictedProbabilities: true,
  nodeLabels: ['UnknownHouse']
})
YIELD nodeId, predictedClass, predictedProbabilities
WITH gds.util.asNode(nodeId) AS houseNode, predictedClass, predictedProbabilities
RETURN
  houseNode.color AS classifiedHouse,
  predictedClass,
  floor(predictedProbabilities[predictedClass] * 100) AS confidence
ORDER BY classifiedHouse
```

Table 912. Results

classifiedHouse	predictedClass	confidence
"Pink"	0	98.0
"Tan"	1	98.0
"Yellow"	2	79.0

As we can see, the model was able to predict the pink house into class 0, tan house into class 1, and yellow house into class 2. This makes sense, as all houses in class 0 had three stories, class 1 two stories and class 2 one story, and the same is true of the pink, tan and yellow houses, respectively. Additionally, we see that the model is confident in these predictions, as the confidence is  $\geq 79\%$  in all cases.

## Mutate

The `mutate` execution mode updates the named graph with a new node property containing the predicted class for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row including information about timings and how many properties were written. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

```
CALL gds.alpha.ml.pipeline.nodeClassification.predict.mutate('myGraph', {
  nodeLabels: ['UnknownHouse'],
  modelName: 'nc-pipeline-model',
  mutateProperty: 'predictedClass',
  predictedProbabilityProperty: 'predictedProbabilities'
}) YIELD nodePropertiesWritten
```

Table 913. Results

nodePropertiesWritten
6

Since we specified also the `predictedProbabilityProperty` we are writing two properties for each of the 3 `UnknownHouse` nodes.

## Write

The `write` execution mode writes the predicted property for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter `writeProperty`. The result is a single summary row including information about timings and how many properties were written. The `write` mode enables directly persisting the results to the database.

For more details on the `write` mode in general, see [Write](#).

```
CALL gds.alpha.ml.pipeline.nodeClassification.predict.write('myGraph', {
  nodeLabels: ['UnknownHouse'],
  modelName: 'nc-pipeline-model',
  writeProperty: 'predictedClass',
  predictedProbabilityProperty: 'predictedProbabilities'
}) YIELD nodePropertiesWritten
```

Table 914. Results

nodePropertiesWritten
6

Since we specified also the `predictedProbabilityProperty` we are writing two properties for each of the 3 `UnknownHouse` nodes.

## 7.8.5. Link prediction

### Introduction

Link prediction is a common machine learning task applied to graphs: training a model to learn, between pairs of nodes in a graph, where relationships should exist. The predicted links are undirected. You can think of this as building a model to predict missing relationships in your dataset or relationships that are likely to form in the future. Neo4j GDS trains supervised machine learning models based on the relationships and node properties in your graph to predict the existence - and probability - of relationships.

Link Prediction can be used favorably together with [pre-processing algorithms](#).

The basic work flow of Link Prediction contains the following parts which are described below:

- [Creating training and test graphs](#)
- [Training and Evaluating model candidates](#)
- [Applying a model for prediction](#)

### Training, Model Selection and Evaluation

When building a model, it is possible to specify multiple model configurations and a model selection metric. The train mode, `gds.alpha.ml.linkPrediction.train`, is responsible for training and evaluating the models, selecting the best model, and storing it in the model catalog.

The train mode takes as input two relationship types representing the training graph and test graph respectively. The relationship types must have an integer property, with values being either `0` or `1`. If the value is `0` the relationship represents a negative example, meaning a node pair which is not connected in the original graph. If the value is `1` the relationship represents a positive example, meaning a relationship which does exist in the original graph.

To obtain the feature vector for an example, the algorithm first forms node feature vectors for the source and target nodes of the example in question. This is done by concatenating the property values of the specified feature properties in order into a node feature vector. Thus, for each relationship we have one feature vector for the source node,  $s = [s_1, s_2, \dots, s_d]$ , and one for the target node,  $t = [t_1, t_2, \dots, t_d]$ . Thereafter, the algorithm uses a *link feature combiner* to combine the two node feature vectors into a single feature vector  $f$  for the training example. There are three supported link feature combiners:

- **L2**
  - which gives a feature vector  $f = [(s_1 - t_1)^2, (s_2 - t_2)^2, \dots, (s_d - t_d)^2]$ .
- **HADAMARD**
  - which gives a feature vector  $f = [s_1 * t_1, s_2 * t_2, \dots, s_d * t_d]$ .
- **COSINE**
  - which gives a single scalar feature, using the [Cosine similarity](#) between  $s$  and  $t$  given by

$$f = \frac{\sum_{i=1}^d s_i t_i}{\sqrt{\sum_{i=1}^d s_i^2} \sqrt{\sum_{i=1}^d t_i^2}}$$

The precise steps of the train mode are:

1. The relationships of the training graph are divided into a number of folds, consisting of a training part and a validation part.
2. Each model candidate is trained on each train part and evaluated on the respective validation part. The training process uses a logistic regression algorithm, and the evaluation uses the [AUCPR metric](#).
3. The model with the highest average score according to the metric will win the training.
4. The winning model will then be re-trained on the whole training graph and evaluated on the training graph as well as on the test graph.
5. The winning model will be registered in the [Model Catalog](#).

Trained models may then be used to predict the probability of a relationship between two nodes.

### Applying a Link Prediction model

A previously trained model can be applied by invoking the `gds.alpha.ml.linkPrediction.predict.mutate` mode. This will retrieve the model by name from the model catalog. The model will thereby be used to predict the probability of relationships between all node pairs in the graph that are not connected. There are two mandatory configuration parameters, which limit the size of the output:

- `topN` retains the most probable predictions.
- `threshold` retains predictions whose probability is above the threshold.

### Train/Test Splitting

In order to train a Link Prediction model, one needs training and test graphs as described [above](#). The recommended way to obtain these is by using `gds.alpha.ml.splitRelationships()` procedure, once to produce the test graph, and another time for the training graph. By invoking this procedure the first time, one obtains two new relationship types that represent the test graph and a 'remaining' graph. One can then, invoke the procedure again, on the just created 'remaining' graph, which then creates a training graph and an even smaller 'remaining' graph. [Below](#), is an example usage of how the `splitRelationships` procedure can be used to prepare the required datasets for training.

The 'remaining' graph after the second split can optionally be used to create node embeddings without data leakage from test or validation sets.

Note that, after the first invocation, we cannot use the 'remaining' graph as the training graph, because it not guaranteed to have `0/1` value relationship labels nor negative link examples.

### Metrics

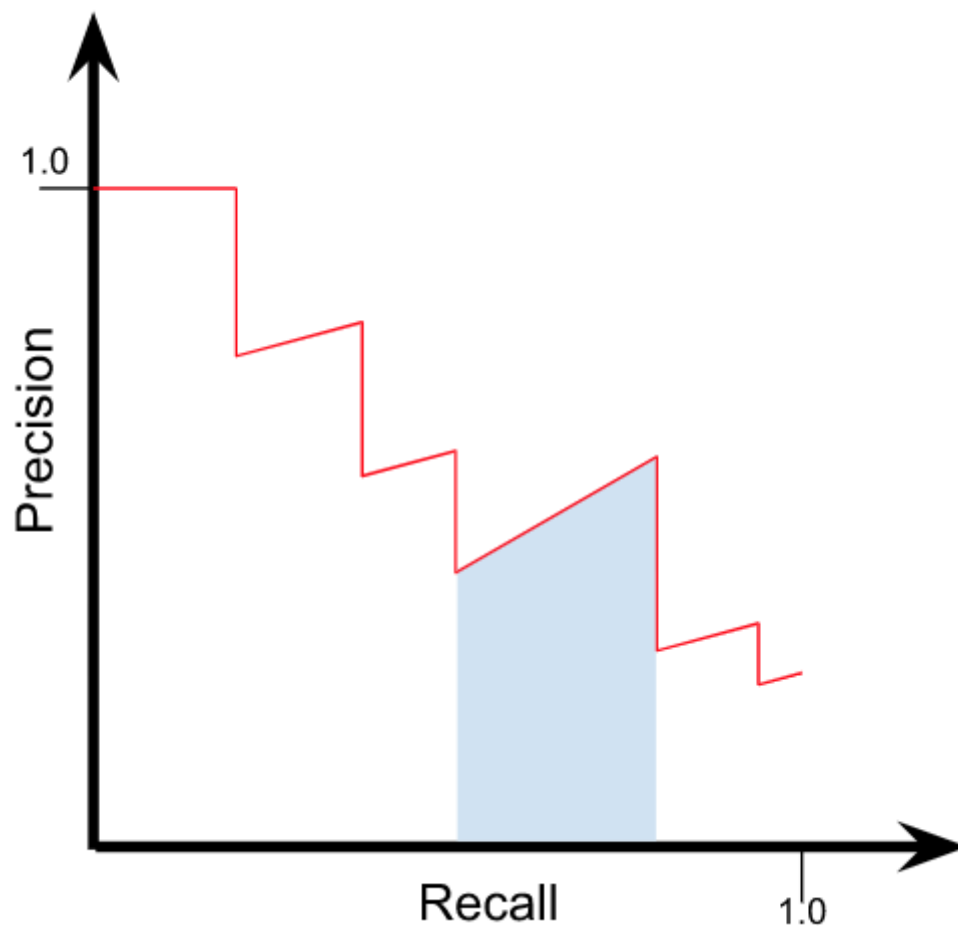
The Link Prediction model in the Neo4j GDS library supports only the Area Under the Precision-Recall Curve metric, abbreviated as AUCPR. In order to compute precision and recall we require a set of examples, each of which has a positive or negative target. For each example we have also a predicted

target. Given the true and predicted targets, we can compute precision and recall (for reference, see f.e. [Wikipedia](#)).

Then, to compute the AUCPR, we construct the precision-recall curve, as follows:

- Each predicted target is associated with a prediction strength, for example the predicted probability of a positive target. We sort the examples in descending order of prediction strength.
- For all prediction strengths that occur, we use that strength as a threshold and for all examples of that strength or higher predict that these examples have positive targets.
- We now compute precision  $p$  and recall  $r$  and consider the tuple  $(r, p)$  as a point on a curve, the precision-recall curve.
- Finally, the curve is linearly interpolated and the area is computed as a union of trapezoids with corners on the points.

The curve will have a shape that looks something like this:



Note here the blue area which shows one trapezoid under the curve.

The area under the Precision-Recall curve can also be interpreted as an average precision where the average is over different classification thresholds.

## Class imbalance

Most graphs have far more non-connected node pairs than connected ones (e.g. sparse graphs). Thus, typically we have an issue with class imbalance. There are multiple strategies to account for imbalanced data. In our procedure, the AUCPR metric is used which is considered more suitable than the commonly used AUROC (Area Under the Receiver Operating Characteristic) metric for imbalanced data. For the metric to appropriately reflect both positive (connected node pairs) and negative (non-connected pairs) examples, we provide the ability to both control the ratio of sampling between the classes, and to control the relative weight of classes via `negativeClassWeight`. The former is configured by the configuration parameter `negativeSamplingRatio` in `splitRelationships` when using that procedure to generate the test set. Tuning the `negativeClassWeight`, which is explained below, means weighting up or down the false positives when computing precision.

The recommended value for `negativeSamplingRatio` is the true class ratio of the graph, in other words, not applying *undersampling*. However, the higher the value, the bigger the test set and thus the time to evaluate. The ratio of total probability mass of negative versus positive examples in the test set is approximately `negativeSamplingRatio * negativeClassWeight`. Thus, both of these parameters can be adjusted in tandem to trade off evaluation accuracy with speed.

The true class ratio is computed as  $(q - r) / r$ , where  $q = n(n-1)/2$  is the number of possible undirected relationships, and  $r$  is the number of actual undirected relationships. Please note that the `relationshipCount` reported by the `graph list` procedure is the *directed* count of relationships summed over all existing relationship types. Thus, we recommend using Cypher to obtain  $r$  on the source Neo4j graph. For example, this query will count the number of relationships of type `T` or `R`:

```
MATCH (a)-[rel:T | R]-(b)
WHERE a < b
RETURN count(rel) AS r
```

When choosing a value for `negativeClassWeight`, two factors should be considered. First, the desired ratio of total probability mass of negative versus positive examples in the test set. Second, what the ratio of sampled negative examples to positive examples was in the test set. To be consistent with *traditional* evaluation, one should choose parameters so that `negativeSamplingRatio * negativeClassWeight = 1.0`, for example by setting the values to the true class ratio and its reciprocal, or both values to `1.0`.

Alternatively, one can aim for the ratio of total probability weight between the classes to be close to the true class ratio. That is, making sure `negativeSamplingRatio * negativeClassWeight` is close to the true class ratio. The reported metric (AUCPR) then better reflects the expected precision on unseen highly imbalanced data. With this type of evaluation one has to adjust expectations as the metric value then becomes much smaller.

## Syntax

This section covers the syntax used to execute the Link Prediction algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



The named graphs must be projected in the `UNDIRECTED` orientation for the Link Prediction model.



Run Link Prediction in train mode on a named graph:

```
CALL gds.alpha.ml.linkPrediction.train(
  graphName: String,
  configuration: Map
) YIELD
  trainMillis: Integer,
  modelInfo: Map,
  configuration: Map
```

Table 915. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 916. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of the model to train, must not exist in the Model Catalog.
featureProperties	List of String	[]	yes	The names of the node properties that should be used as input features. All property names must exist in the in-memory graph and be of type Float or List of Float.
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 917. Algorithm specific configuration

Name	Type	Default	Optional	Description
trainRelationshipType	String	n/a	no	Relationship type to use during model training.
testRelationshipType	String	n/a	no	Relationship type to use during model evaluation.
validationFolds	Integer	n/a	no	Number of divisions of the training graph used during model selection.
negativeClassWeight	Float	n/a	no	Weight of negative examples in model evaluation. Positive examples have weight 1.
params	List of Map	n/a	no	List of model configurations to be trained and compared. See next table for details.
randomSeed	Integer	n/a	yes	Seed for the random number generator used during training.



Table 918. Model configuration

Name	Type	Default	Optional	Description
penalty	Float	0.0	yes	Penalty used for the logistic regression. By default, no penalty is applied.
linkFeatureCombiner	String	"L2"	yes	Link feature combiner is used to combine two node feature vectors into the feature vector for the training. Available combiners are L2, HADAMARD and COSINE.
batchSize	Integer	100	yes	Number of nodes per batch.
minEpochs	Integer	1	yes	Minimum number of training epochs.
maxEpochs	Integer	100	yes	Maximum number of training epochs.
patience	Integer	1	yes	Maximum number of unproductive consecutive epochs.
tolerance	Float	0.001	yes	The minimal improvement of the loss to be considered productive.

For hyperparameter tuning ideas, look [here](#).

Table 919. Results

Name	Type	Description
trainMillis	Integer	Milliseconds used for training.
modelInfo	Map	Information about the training and the winning model.
configuration	Map	Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the [Model List Procedure](#). The `modelInfo` return field has the following algorithm-specific subfields:

Table 920. Model info fields

Name	Type	Description
bestParameters	Map	The model parameters which performed best on average on validation folds according to the primary metric.
metrics	Map	Map from metric description to evaluated metrics for various models and subsets of the data, see below.

The structure of `modelInfo` is:

```

{
  bestParameters: Map,           ①
  metrics: {                     ②
    AUCPR: {
      test: Float,              ③
      outerTrain: Float,        ④
      train: [{                 ⑤
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      {
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      ...
    ],
    validation: [{             ⑥
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    {
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    ...
  ]
}
}

```

- ① The best scoring model candidate configuration.
- ② The `metrics` map contains an entry for each metric description (currently only `AUCPR`) and the corresponding results for that metric.
- ③ Numeric value for the evaluation of the best model on the test set.
- ④ Numeric value for the evaluation of the best model on the outer train set.
- ⑤ The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.
- ⑥ The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.

Run Link Prediction in stream mode on a named graph:

```
CALL gds.alpha.ml.linkPrediction.predict.stream(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  node1: Integer,  
  node2: Integer,  
  probability: Float
```

Table 921. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 922. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 923. Algorithm specific configuration

Name	Type	Default	Optional	Description
topN	Integer	n/a	no	Limit on predicted relationships to output.
threshold	Float	n/a	no	Minimum predicted probability on relationships to output.

Table 924. Results

Name	Type	Description
node1	Integer	Node ID of the first node.
node2	Integer	Node ID of the second node.
probability	Float	Predicted probability of a link between the nodes.

Run Link Prediction in mutate mode on a named graph:

```
CALL gds.alpha.ml.linkPrediction.predict.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  createMillis: Integer,  
  computeMillis: Integer,  
  postProcessingMillis: Integer,  
  mutateMillis: Integer,  
  relationshipsWritten: Integer,  
  configuration: Map
```

Table 925. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 926. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a Link Prediction model in the model catalog.
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
mutateRelationshipType	String	n/a	no	The relationship type used for the new relationships written to the in-memory graph.
mutateProperty	String	'probability'	yes	The relationship property in the GDS graph to which the result is written.

Table 927. Algorithm specific configuration

Name	Type	Default	Optional	Description
topN	Integer	n/a	no	Limit on predicted relationships to output.
threshold	Float	n/a	no	Minimum predicted probability on relationships to output.

Table 928. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

Name	Type	Description
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
relationshipsWritten	Integer	Number of relationships created.
configuration	Map	Configuration used for running the algorithm.

Run Link Prediction in write mode on a named graph:

```
CALL gds.alpha.ml.linkPrediction.predict.write(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  writeMillis: Integer,
  relationshipsWritten: Integer,
  configuration: Map
```

Table 929. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 930. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a Link Prediction model in the model catalog.
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
writeRelationshipType	String	n/a	no	The relationship type used to persist the computed relationships in the Neo4j database.
writeProperty	String	n/a	no	The relationship property in the Neo4j database to which the result is written.

Table 931. Algorithm specific configuration

Name	Type	Default	Optional	Description
topN	Integer	n/a	no	Limit on predicted relationships to output.
threshold	Float	n/a	no	Minimum predicted probability on relationships to output.

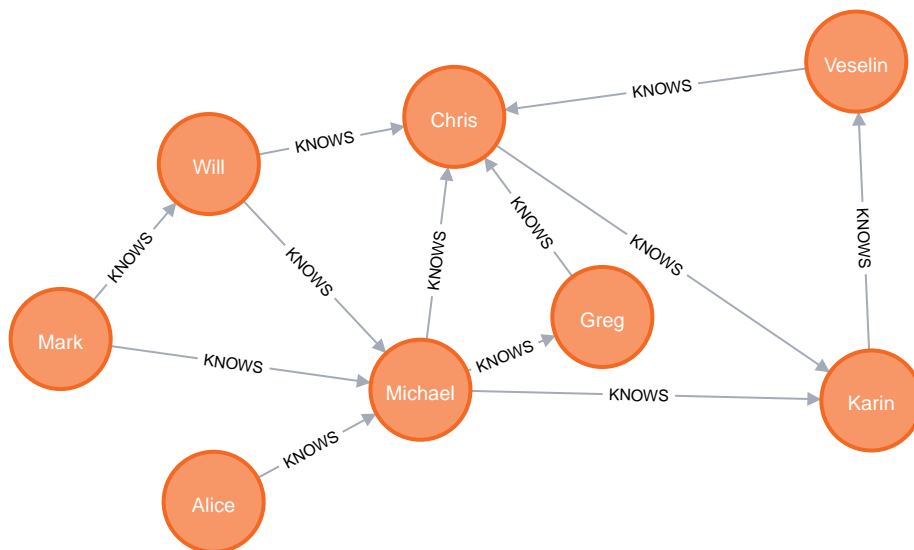
Table 932. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.

Name	Type	Description
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
writeMillis	Integer	Milliseconds for writing result data back to Neo4j.
relationshipsWritten	Integer	Number of relationships created.
configuration	Map	Configuration used for running the algorithm.

## Examples

In this section we will show examples of running the Link Prediction algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```
CREATE
  (alice:Person {name: 'Alice', numberOfPosts: 38}),
  (michael:Person {name: 'Michael', numberOfPosts: 67}),
  (karin:Person {name: 'Karin', numberOfPosts: 30}),
  (chris:Person {name: 'Chris', numberOfPosts: 132}),
  (will:Person {name: 'Will', numberOfPosts: 6}),
  (mark:Person {name: 'Mark', numberOfPosts: 32}),
  (greg:Person {name: 'Greg', numberOfPosts: 29}),
  (veselin:Person {name: 'Veselin', numberOfPosts: 3}),

  (alice)-[:KNOWS]->(michael),
  (michael)-[:KNOWS]->(karin),
  (michael)-[:KNOWS]->(chris),
  (michael)-[:KNOWS]->(greg),
  (will)-[:KNOWS]->(michael),
  (will)-[:KNOWS]->(chris),
  (mark)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(will),
  (greg)-[:KNOWS]->(chris),
  (veselin)-[:KNOWS]->(chris),
  (karin)-[:KNOWS]->(veselin),
  (chris)-[:KNOWS]->(karin);
```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. We will also project the `numberOfPosts` property, so we can use it as a model feature. For the relationships we must use the `UNDIRECTED` orientation. This is because the Link Prediction model is defined only for undirected graphs.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  {
    Person: {
      properties: ['numberOfPosts']
    }
  },
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
);
```



The Link Prediction model requires the graph to be created using the `UNDIRECTED` orientation for relationships.

In the following examples we will demonstrate using the Link Prediction model on this graph.

## Train

First, we must do the test/train splits. For this we will make use of the `gds.alpha.ml.splitRelationships` procedure. We will do one split to generate the test graph. We note that in the example graph there are



eight nodes and twelve directed relationships. Recall that we compute the class ratio as  $(q - r) / q$ , where we then have  $q = 8(8-1)/2$  and  $r = 12$  which gives us class ratio of  $(28 - 12) / 12 \approx 1.33$ . We use this to configure `negativeSampleRatio` to achieve a sampling proportional to the class ratio.

```
CALL gds.alpha.ml.splitRelationships.mutate('myGraph', {
  relationshipTypes: ['KNOWS'],
  remainingRelationshipType: 'KNOWS_REMAINING',
  holdoutRelationshipType: 'KNOWS_TESTGRAPH',
  holdoutFraction: 0.2,
  negativeSamplingRatio: 1.33,
  randomSeed: 1984
}) YIELD relationshipsWritten
```

Table 933. Results

relationshipsWritten
25

We will create copied relationships for each existing relationship, into either the `KNOWS_REMAINING` or the `KNOWS_TESTGRAPH` relationship types. All relationships in `KNOWS_TESTGRAPH` will have a `label` property. Additionally, a number of non-existing relationships will be created into the `KNOWS_TESTGRAPH` relationship type to be used as negative examples, with a `label` of `0`.

Next, we will create the train graph.

```
CALL gds.alpha.ml.splitRelationships.mutate('myGraph', {
  relationshipTypes: ['KNOWS_REMAINING'],
  remainingRelationshipType: 'KNOWS_IGNORED_FOR_TRAINING',
  holdoutRelationshipType: 'KNOWS_TRAINGRAPH',
  holdoutFraction: 0.2,
  negativeSamplingRatio: 1.33,
  randomSeed: 1984
}) YIELD relationshipsWritten
```

Table 934. Results

relationshipsWritten
20

With both training and test graphs, we are ready to train models. We will use 5 validation folds, meaning we will split the train graph into 5 pairs, using one part of each pair for training and one for validation. Since we set the `negativeSamplingRatio` to 1.33 (the class ratio of the graph) above, we'll set the `negativeClassWeight` during training to  $1 / 1.33$  to assign equal weight to both classes.

## Train a Link Prediction model:

```
CALL gds.alpha.ml.linkPrediction.train('myGraph', {
  trainRelationshipType: 'KNOWS_TRAINGRAPH',
  testRelationshipType: 'KNOWS_TESTGRAPH',
  modelName: 'lp-numberOfPosts-model',
  featureProperties: ['numberOfPosts'],
  validationFolds: 5,
  negativeClassWeight: 1.0 / 1.33,
  randomSeed: 2,
  concurrency: 1,
  params: [
    {penalty: 0.5, maxEpochs: 1000},
    {penalty: 1.0, maxEpochs: 1000},
    {penalty: 0.0, maxEpochs: 1000}
  ]
}) YIELD modelInfo
RETURN
{ maxEpochs: modelInfo.bestParameters.maxEpochs, penalty: modelInfo.bestParameters.penalty } AS
winningModel,
modelInfo.metrics.AUCPR.outerTrain AS trainGraphScore,
modelInfo.metrics.AUCPR.test AS testGraphScore
```

Table 935. Results

winningModel	trainGraphScore	testGraphScore
{maxEpochs=1000, penalty=0.5}	0.38525757517173825	0.46710171439292664

Here we can observe that the model candidate with penalty 0.5 performed the best in the training phase, with a score of about 71% over the train graph. On the test graph, the model scored much lower at about 35%. This indicates that the model reacted fairly well to the train graph, but did not generalise very well to unseen data. In order to achieve a higher test score, we may need to use better features, a larger graph, or different model configuration.

## Stream

In the `stream` execution mode, the algorithm returns the top predicted relationships. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

```
CALL gds.alpha.ml.linkPrediction.predict.stream('myGraph', {
  relationshipTypes: ['KNOWS'],
  modelName: 'lp-numberOfPosts-model',
  topN: 5,
  threshold: 0.45
})
YIELD node1, node2, probability
RETURN gds.util.asNode(node1).name AS person1, gds.util.asNode(node2).name AS person2, probability
ORDER BY probability DESC, person1
```

Table 936. Results

person1	person2	probability
"Karin"	"Greg"	0.4991363247445545
"Karin"	"Mark"	0.49896977670628373

person1	person2	probability
"Mark"	"Greg"	0.49869219716877955
"Will"	"Veselin"	0.49869219716877955
"Alice"	"Mark"	0.49719328593255546

We specified `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships. Note that the predicted link between the Karin and Greg nodes does not reflect any particular direction between them.

## Mutate

In this example we will show how to use a trained model to predict new relationships in your in-memory graph. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the [train example](#) which we gave the name `'lp-numberOfPosts-model'`.

We must also make sure that we do not include any of the relationships from the train or test graphs, which we do by specifying a relationship filter for the original relationship type `'KNOWS'`.

```
CALL gds.alpha.ml.linkPrediction.predict.mutate('myGraph', {
  relationshipTypes: ['KNOWS'],
  modelName: 'lp-numberOfPosts-model',
  mutateRelationshipType: 'KNOWS_PREDICTED',
  topN: 5,
  threshold: 0.45
}) YIELD relationshipsWritten
```

Table 937. Results

relationshipsWritten
10

We specified `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships. Because we are using the `UNDIRECTED` orientation, we will write twice as many relationships to the in-memory graph.

## Write

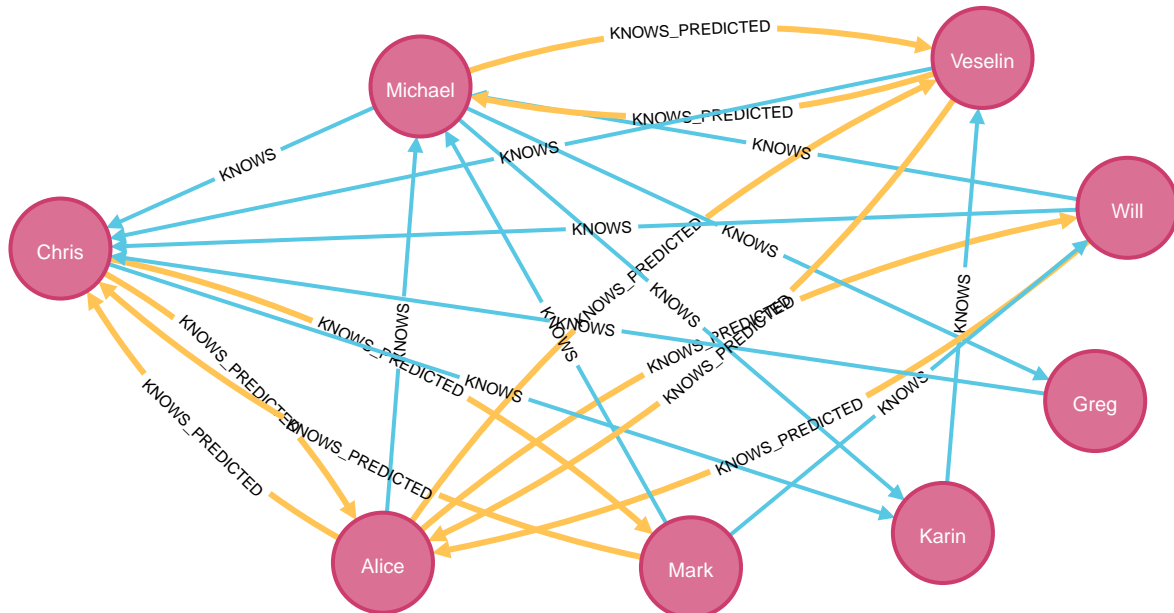
In this example we will show how to use a trained model to predict new relationships in your in-memory graph, and write the predictions back to Neo4j. We will again use the model `'lp-numberOfPosts-model'`, as in the [mutate example](#).

```
CALL gds.alpha.ml.linkPrediction.predict.write('myGraph', {
  relationshipTypes: ['KNOWS'],
  modelName: 'lp-numberOfPosts-model',
  writeRelationshipType: 'KNOWS_PREDICTED',
  topN: 5,
  threshold: 0.45
}) YIELD relationshipsWritten
```

Table 938. Results

relationshipsWritten
10

The end result looks like this:



In yellow we highlight the predicted relationships.

## 7.8.6. Link prediction pipelines

### Introduction

Link prediction is a common machine learning task applied to graphs: training a model to learn, between pairs of nodes in a graph, where relationships should exist. More precisely, the input of the machine learning model are examples of node pairs which are labeled as connected or not connected. The GDS library provides Link prediction, see [here](#). Here we describe an additional method that provides an end-to-end Link prediction experience. In addition to managing a predictive model, it also manages:

- splitting relationships into subsets for **test**, **train** and **feature-input**
- a pipeline of processing steps that supply custom features for the model

The motivation for using pipelines are:

- easier to get splits right and prevent data leakage
- ensuring that the same feature creation steps are applied at predict and train time
- applying the trained model with a single procedure call
- persisting the pipeline as a whole

The rest of this page is divided as follows:

- [Creating a pipeline](#)

- [Adding node properties](#)
- [Adding link features](#)
- [Configuring the relationship splits](#)
- [Configuring the model parameters](#)
- [Training the pipeline](#)
- [Applying a trained model for prediction](#)

## Creating a pipeline

The first step of building a new pipeline is to create one using `gds.alpha.ml.pipeline.linkPrediction.create`. This stores a trainable model object in the model catalog of type `Link prediction training pipeline`. This represents a configurable pipeline that can later be invoked for training, which in turn creates a trained pipeline. The latter is also a model which is stored in the catalog with type `Link prediction pipeline`.

## Syntax

### Create pipeline syntax

```
CALL gds.alpha.ml.pipeline.linkPrediction.create(
  pipelineName: String
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureSteps: List of Map,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 939. Parameters

Name	Type	Description
pipelineName	String	The name of the created pipeline.

Table 940. Results

Name	Type	Description
name	String	Name of the pipeline.
nodePropertySteps	List of Map	List of configurations for node property steps.
featureSteps	List of Map	List of configurations for feature steps.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will create a pipeline:

```
CALL gds.alpha.ml.pipeline.linkPrediction.create('pipe')
```

Table 941. Results

name	nodePropertySteps	featureSteps	splitConfig	parameterSpace
"pipe"	[]	[]	{negativeSamplingRatio=1.0, testFraction=0.1, validationFolds=3, trainFraction=0.1}	[[useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001]]

This shows that the newly created pipeline does not contain any steps yet, and has defaults for the split and train parameters.

## Adding node properties

A link prediction pipeline can execute one or several GDS algorithms in mutate mode that create node properties in the in-memory graph. Such steps producing node properties can be chained one after another and created properties can also be used to [add features](#). Moreover, the node property steps that are added to the pipeline will be executed both when [training](#) a model and when the trained model is [applied for prediction](#).

The name of the procedure that should be added can be a fully qualified GDS procedure name ending with `.mutate`. The ending `.mutate` may be omitted and one may also use shorthand forms such as `node2vec` instead of `gds.beta.node2vec.mutate`.

For example, [pre-processing algorithms](#) can be used as node property steps.

## Syntax

Add node property syntax

```
CALL gds.alpha.ml.pipeline.linkPrediction.addNodeProperty(  
  pipelineName: String,  
  procedureName: String,  
  procedureConfiguration: Map  
)  
YIELD  
  name: String,  
  nodePropertySteps: List of Map,  
  featureSteps: List of Map,  
  splitConfig: Map,  
  parameterSpace: List of Map
```

Table 942. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
procedureName	String	The name of the procedure to be added to the pipeline.
procedureConfiguration	Map	The configuration of the procedure, excluding <code>graphName</code> , <code>nodeLabels</code> and <code>relationshipTypes</code> .

Table 943. Results

Name	Type	Description
name	String	Name of the pipeline.
nodePropertySteps	List of Map	List of configurations for node property steps.
featureSteps	List of Map	List of configurations for feature steps.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will add a node property step to the pipeline:

```
CALL gds.alpha.ml.pipeline.linkPrediction.addNodeProperty('pipe', 'fastRP', {
  mutateProperty: 'embedding',
  embeddingDimension: 256,
  randomSeed: 42
})
```

Table 944. Results

name	nodePropertySteps	featureSteps	splitConfig	parameterSpace
"pipe"	[[name=gds.fastRP.mutate, config={randomSeed=42, embeddingDimension=256, mutateProperty=embedding}]]	[]	{negativeSamplingRatio=1.0, testFraction=0.1, validationFolds=3, trainFraction=0.1}	[[useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001]]

The pipeline will now execute the [fastRP algorithm](#) in mutate mode both before [training](#) a model, and when the trained model is [applied for prediction](#). This ensures the [embedding](#) property can be used as an input for link features.

## Adding link features

A Link Prediction pipeline executes a sequence of steps to compute the features used by a machine learning model. A feature step computes a vector of features for given node pairs. For each node pair, the

results are concatenated into a single *link* feature vector. The order of the features in the link feature vector follows the order of the feature steps. Like with node property steps, the feature steps are also executed both at [training](#) and [prediction](#) time. The supported methods for obtaining features are described [below](#).

## Syntax

### Adding a link feature to a pipeline syntax

```
CALL gds.alpha.ml.pipeline.linkPrediction.addFeature(
  pipelineName: String,
  featureType: String,
  configuration: Map
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureSteps: List of Map,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 945. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
featureType	String	The featureType determines the method used for computing the link feature. See <a href="#">supported types</a> .
configuration	Map	Configuration for splitting the relationships.

Table 946. Configuration

Name	Type	Default	Description
nodeProperties	List of String	no	The names of the node properties that should be used as input.

Table 947. Results

Name	Type	Description
name	String	Name of the pipeline.
nodeProperty Steps	List of Map	List of configurations for node property steps.
featureSteps	List of Map	List of configurations for feature steps.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Supported feature types

A feature step can use node properties that exist in the input graph or are added by the pipeline. For each node in each potential link, the values of `nodeProperties` are concatenated, in the configured order, into a



vector  $f$ . That is, for each potential link the feature vector for the source node,  $s = [s_1, s_2, \dots, s_d]$ , is combined with the one for the target node,  $t = [t_1, t_2, \dots, t_d]$ , into a single feature vector  $f$ .

The supported types of features can then be described as follows:

Table 948. Supported feature types

Feature Type	Formula / Description
L2	$f = [(s_1 - t_1)^2, (s_2 - t_2)^2, \dots, (s_d - t_d)^2]$
HADAMARD	$f = [s_1 * t_1, s_2 * t_2, \dots, s_d * t_d]$
COSINE	$f = \frac{\sum_{i=1}^d s_i t_i}{\sqrt{\sum_{i=1}^d s_i^2} \sqrt{\sum_{i=1}^d t_i^2}}$

## Example

The following will add a feature step to the pipeline:

```
CALL gds.alpha.ml.pipeline.linkPrediction.addFeature('pipe', 'hadamard', {
  nodeProperties: ['embedding', 'numberOfPosts']
}) YIELD featureSteps
```

Table 949. Results

featureSteps
[[name=HADAMARD, config={nodeProperties=[embedding, numberOfPosts]}]]

When executing the pipeline, the `nodeProperties` must be either present in the input graph, or created by a previous node property step. For example, the `embedding` property could be created by the previous example, and we expect `numberOfPosts` to already be present in the in-memory graph used as input, at train and predict time.

## Configuring the relationship splits

Link Prediction pipelines manage splitting the relationships into several sets and add sampled negative relationships to some of these sets. Configuring the splitting is optional, and if omitted, splitting will be done using default settings.

The splitting configuration of a pipeline can be inspected by using `gds.beta.model.list` and possibly only yielding `splitConfig`.

The splitting of relationships proceeds internally in the following steps:

1. The graph is filtered according to specified `nodeLabels` and `relationshipTypes`, which are configured at train time.
2. The relationships remaining after filtering we call positive, and they are split into a `test` set and remaining relationships which we refer to as `test-complement` set.
  - The `test` set contains a `testFraction` fraction of the positive relationships.

- Random negative relationships are added to the `test` set. The number of negative relationships is the number of positive ones multiplied by the `negativeSamplingRatio`.
  - The negative relationships do not coincide with positive relationships.
3. The relationships in the `test-complement` set are split into a `train` set and a `feature-input` set.
- The `train` set contains a `trainFraction` fraction of the `test-complement` set.
  - The `feature-input` set contains  $(1 - \text{trainFraction})$  fraction of the `test-complement` set.
  - Random negative relationships are added to the `train` set. The number of negative relationships is the number of positive ones multiplied by the `negativeSamplingRatio`.
  - The negative relationships do not coincide with positive relationships, nor with test relationships.

The sampled positive and negative relationships are given relationship weights of `1.0` and `0.0` respectively so that they can be distinguished.

The `feature-input` graph is used, both in training and testing, for computing node properties and therefore also features which depend on node properties.

The `train` and `test` relationship sets are used for:

- determining the label (positive or negative) for each training or test example
- identifying the node pair for which link features are to be computed

However, they are not used by the algorithms run in the node property steps. The reason for this is that otherwise the model would use the prediction target (existence of a relationship) as a feature.

## Syntax

Configure the relationship split syntax

```
CALL gds.alpha.ml.pipeline.linkPrediction.configureSplit(
  pipelineName: String,
  configuration: Map
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureSteps: List of Map,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 950. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
configuration	Map	Configuration for splitting the relationships.

Table 951. Configuration

Name	Type	Default	Description
validationFolds	Integer	3	Number of divisions of the training graph used during <a href="#">model selection</a> .
testFraction	Double	0.1	Fraction of the graph reserved for testing. Must be in the range (0, 1).
trainFraction	Double	0.1	Fraction of the <a href="#">test-complement set</a> reserved for training. Must be in the range (0, 1).
negativeSamplingRatio	Double	1.0	The desired ratio of negative to positive samples in the test and train set.

Table 952. Results

Name	Type	Description
name	String	Name of the pipeline.
nodePropertySteps	List of Map	List of configurations for node property steps.
featureSteps	List of Map	List of configurations for feature steps.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will configure the splitting of the pipeline:

```
CALL gds.alpha.ml.pipeline.linkPrediction.configureSplit('pipe', {
  testFraction: 0.3,
  trainFraction: 0.3,
  validationFolds: 7
})
YIELD splitConfig
```

Table 953. Results

splitConfig
{negativeSamplingRatio=1.0, testFraction=0.3, validationFolds=7, trainFraction=0.3}

We now reconfigured the splitting of the pipeline, which will be applied during [training](#).

## Configuring the model parameters

The `gds.alpha.ml.pipeline.linkPrediction.configureParams` mode is used to set up the train mode with a list of configurations of logistic regression models. The set of model configurations is called the *parameter space* which parametrizes a set of model candidates. The parameter space can be configured by passing this procedure a list of maps, where each map configures the training of one logistic regression model. In [Training the pipeline](#), we explain further how the configured model candidates are trained, evaluated and compared.

The allowed model parameters are listed in the table [Model configuration](#).

If `configureParams` is not used, then a single model with defaults for all the model parameters is used. The parameter space of a pipeline can be inspected using `gds.beta.model.list` and optionally yielding only `parameterSpace`.

## Syntax

### Configure the train parameters syntax

```
CALL gds.alpha.ml.pipeline.linkPrediction.configureParams(
  pipelineName: String,
  parameterSpace: List of Map
)
YIELD
  name: String,
  nodePropertySteps: List of Map,
  featureSteps: List of Map,
  splitConfig: Map,
  parameterSpace: List of Map
```

Table 954. Parameters

Name	Type	Description
pipelineName	String	The name of the pipeline.
parameterSpace	List of Map	The parameter space used to select the best model from. Each Map corresponds to potential model. The allowed parameters for a model are defined in the next table.

Table 955. Model configuration

Name	Type	Default	Optional	Description
penalty	Float	0.0	yes	Penalty used for the logistic regression. By default, no penalty is applied.
batchSize	Integer	100	yes	Number of nodes per batch.
minEpochs	Integer	1	yes	Minimum number of training epochs.
maxEpochs	Integer	100	yes	Maximum number of training epochs.
patience	Integer	1	yes	Maximum number of unproductive consecutive epochs.
tolerance	Float	0.001	yes	The minimal improvement of the loss to be considered productive.
useBiasFeature	Boolean	true	yes	Whether the logistic regression model uses a bias feature.

Table 956. Results

Name	Type	Description
name	String	Name of the pipeline.

Name	Type	Description
nodeProperty Steps	List of Map	List of configurations for node property steps.
featureSteps	List of Map	List of configurations for feature steps.
splitConfig	Map	Configuration to define the split before the model training.
parameterSpace	List of Map	List of parameter configurations for models which the train mode uses for model selection.

## Example

The following will configure the parameter space of the pipeline:

```
CALL gds.alpha.ml.pipeline.linkPrediction.configureParams('pipe',
  [{tolerance: 0.001}, {tolerance: 0.01}, {maxEpochs: 500}]
) YIELD parameterSpace
```

Table 957. Results

parameterSpace
<pre>[[useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001], {useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.01}, {useBiasFeature=true, maxEpochs=500, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001}]</pre>

The `parameterSpace` in the pipeline now contains the three different model parameters, expanded with the default values. Each specified model configuration will be tried out during the model selection in [training](#).

## Training the pipeline

The train mode, `gds.alpha.ml.pipeline.linkPrediction.train`, is responsible for splitting data, feature extraction, model selection, training and storing a model for future use. Running this mode results in a [Link prediction pipeline](#) model being stored in the model catalog along with metrics collected during training. The trained pipeline can be [applied](#) to a possibly different graph which produces a relationship type of predicted links, each having a predicted probability stored as a property.

More precisely, the procedure will in order:

1. Apply `nodeLabels` and `relationshipType` filters to the graph. All subsequent graphs have the same node set.
2. Create a relationship split of the graph into `test`, `train` and `feature-input` sets as described in [Configuring the relationship splits](#). These graphs are internally managed and exist only for the duration of the training.
3. Apply the node property steps, added according to [Adding node properties](#), on the `feature-input` graph.
4. Apply the feature steps, added according to [Adding link features](#), to the `train` graph, which yields for each `train` relationship an *instance*, that is, a feature vector and a binary label.

5. Split the training instances using stratified k-fold cross-validation. The number of folds *k* can be configured using `validationFolds` in `gds.alpha.ml.pipeline.linkPrediction.configureSplit`.
6. Train each model candidate given by the [parameter space](#) for each of the folds and evaluate the model on the respective validation set. The training process uses a logistic regression algorithm, and the evaluation uses the [AUCPR metric](#).
7. Declare as winner the model with the highest average metric across the folds.
8. Re-train the winning model on the whole training set and evaluate it on both the `train` and `test` sets. In order to evaluate on the `test` set, the feature pipeline is first applied again as for the `train` set.
9. Register the winning model in the [Model Catalog](#).



The above steps describe what the procedure does logically. The actual steps as well as their ordering in the implementation may differ.



A step can only use node properties that are already present in the input graph or produced by steps, which were added before.

## Syntax

Run Link Prediction in train mode on a named graph:

```
CALL gds.alpha.ml.pipeline.linkPrediction.train(
  graphName: String,
  configuration: Map
) YIELD
  trainMillis: Integer,
  modelInfo: Map,
  configuration: Map
```

Table 958. Parameters

Name	Type	Default	Optional	Description
<code>graphName</code>	String	<code>n/a</code>	no	The name of a graph stored in the catalog.
<code>configuration</code>	Map	<code>{}</code>	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 959. Configuration

Name	Type	Default	Optional	Description
<code>modelName</code>	String	<code>n/a</code>	no	The name of the model to train, must not exist in the Model Catalog.
<code>pipeline</code>	String	<code>n/a</code>	no	The name of the pipeline to execute.
<code>negativeClass Weight</code>	Float	<code>1.0</code>	yes	Weight of negative examples in model evaluation. Positive examples have weight 1.
<code>randomSeed</code>	Integer	<code>n/a</code>	yes	Seed for the random number generator used during training.
<code>nodeLabels</code>	List of String	<code>['*']</code>	yes	Filter the named graph using the given node labels.

Name	Type	Default	Optional	Description
<a href="#">relationshipTypes</a>	List of String	[ '*' ]	yes	Filter the named graph using the given relationship types. The relationships must be undirected.
<a href="#">concurrency</a>	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 960. Results

Name	Type	Description
trainMillis	Integer	Milliseconds used for training.
modelInfo	Map	Information about the training and the winning model.
configuration	Map	Configuration used for the train procedure.

The `modelInfo` can also be retrieved at a later time by using the [Model List Procedure](#). The `modelInfo` return field has the following algorithm-specific subfields:

Table 961. Model info fields

Name	Type	Description
bestParameters	Map	The model parameters which performed best on average on validation folds according to the primary metric.
metrics	Map	Map from metric description to evaluated metrics for various models and subsets of the data, see below.
trainingPipeline	Map	The pipeline used for the training.

The structure of `modelInfo` is:

```

{
  bestParameters: Map,           ①
  trainingPipeline: Map         ②
  metrics: {                    ③
    AUCPR: {
      test: Float,              ④
      outerTrain: Float,        ⑤
      train: [{                 ⑥
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      {
        avg: Float,
        max: Float,
        min: Float,
        params: Map
      },
      ...
    ],
    validation: [{              ⑦
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    {
      avg: Float,
      max: Float,
      min: Float,
      params: Map
    },
    ...
  ]
}
}
}

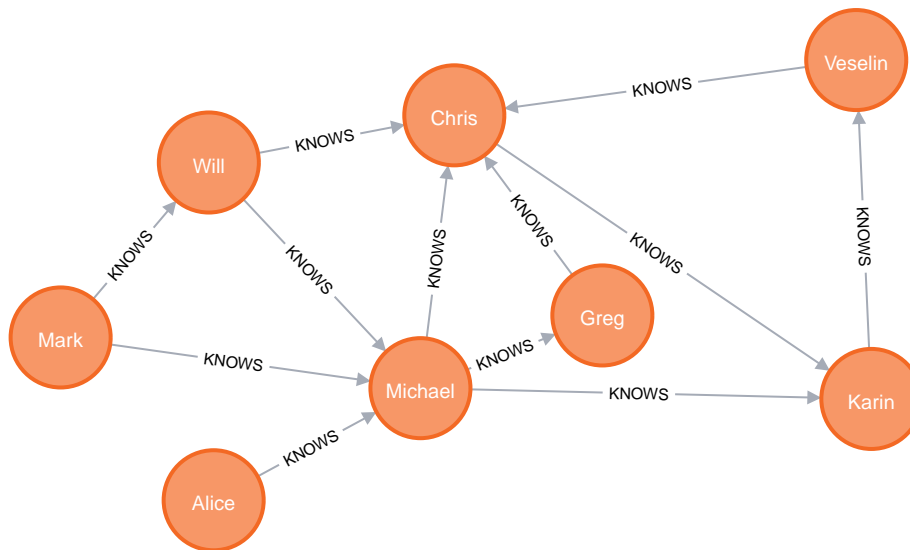
```

- ① The best scoring model candidate configuration.
- ② The pipeline used for the training.
- ③ The `metrics` map contains an entry for each metric description (currently only `AUCPR`) and the corresponding results for that metric.
- ④ Numeric value for the evaluation of the best model on the test set.
- ⑤ Numeric value for the evaluation of the best model on the outer train set.
- ⑥ The `train` entry lists the scores over the `train` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.
- ⑦ The `validation` entry lists the scores over the `validation` set for all candidate models (e.g., `params`). Each such result is in turn also a map with keys `params`, `avg`, `min` and `max`.

## Example

In this example we will create a small graph and train the pipeline we have built up thus far. The graph consists of a handful of nodes connected in a particular pattern. The example graph looks like this:





The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
  (alice:Person {name: 'Alice', numberOfPosts: 38}),
  (michael:Person {name: 'Michael', numberOfPosts: 67}),
  (karin:Person {name: 'Karin', numberOfPosts: 30}),
  (chris:Person {name: 'Chris', numberOfPosts: 132}),
  (will:Person {name: 'Will', numberOfPosts: 6}),
  (mark:Person {name: 'Mark', numberOfPosts: 32}),
  (greg:Person {name: 'Greg', numberOfPosts: 29}),
  (veselin:Person {name: 'Veselin', numberOfPosts: 3}),

  (alice)-[:KNOWS]->(michael),
  (michael)-[:KNOWS]->(karin),
  (michael)-[:KNOWS]->(chris),
  (michael)-[:KNOWS]->(greg),
  (will)-[:KNOWS]->(michael),
  (will)-[:KNOWS]->(chris),
  (mark)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(will),
  (greg)-[:KNOWS]->(chris),
  (veselin)-[:KNOWS]->(chris),
  (karin)-[:KNOWS]->(veselin),
  (chris)-[:KNOWS]->(karin);

```

With the graph in Neo4j we can now project it into the graph catalog. We do this using a native projection targeting the `Person` nodes and the `KNOWS` relationships. We will also project the `numberOfPosts` property, so it can be used when creating link features. For the relationships we must use the `UNDIRECTED` orientation. This is because the Link Prediction pipelines are defined only for undirected graphs.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```

CALL gds.graph.create(
  'myGraph',
  {
    Person: {
      properties: ['numberOfPosts']
    }
  },
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)

```



The Link Prediction model requires the graph to be created using the **UNDIRECTED** orientation for relationships.

The following will train a model using a pipeline:

```
CALL gds.alpha.ml.pipeline.linkPrediction.train('myGraph', {
  pipeline: 'pipe',
  modelName: 'lp-pipeline-model',
  randomSeed: 42
}) YIELD modelInfo
RETURN
modelInfo.bestParameters AS winningModel,
modelInfo.metrics.AUCPR.outerTrain AS trainGraphScore,
modelInfo.metrics.AUCPR.test AS testGraphScore
```

Table 962. Results

winningModel	trainGraphScore	testGraphScore
{useBiasFeature=true, maxEpochs=100, minEpochs=1, penalty=0.0, patience=1, batchSize=100, tolerance=0.001}	0.41666666666666666 3	0.76388888888888888

We can see the model configuration with **tolerance = 0.001** (and defaults filled for remaining parameters) was selected, and has a score of **0.76** on the test set. The score computed as the **AUCPR** metric, which is in the range [0, 1]. A model which gives higher score to all links than non-links will have a score of 1.0, and a model that assigns random scores will on average have a score of 0.5.

## Applying a trained model for prediction

In the previous sections we have seen how to build up a Link Prediction training pipeline and train it to produce a predictive model. After **training**, the runnable model is of type **Link prediction pipeline** and resides in the model catalog.

The trained model can then be applied to a graph in the graph catalog to create a new relationship type containing the predicted links. The relationships also have a property which stores the predicted probability of the link, which can be seen as a relative measure of the model's prediction confidence.

Since the model has been trained on features which are created using the feature pipeline, the same feature pipeline is stored within the model and executed at prediction time. As during training, intermediate node properties created by the node property steps in the feature pipeline are transient and not visible after execution.

When using the model for prediction, the relationships of the input graph are used in two ways. First, the input graph is fed into the feature pipeline and therefore influences predictions if there is at least one step in the pipeline which uses the input relationships (typically any node property step does). Second, predictions are carried out on each node pair that is not connected in the input graph.

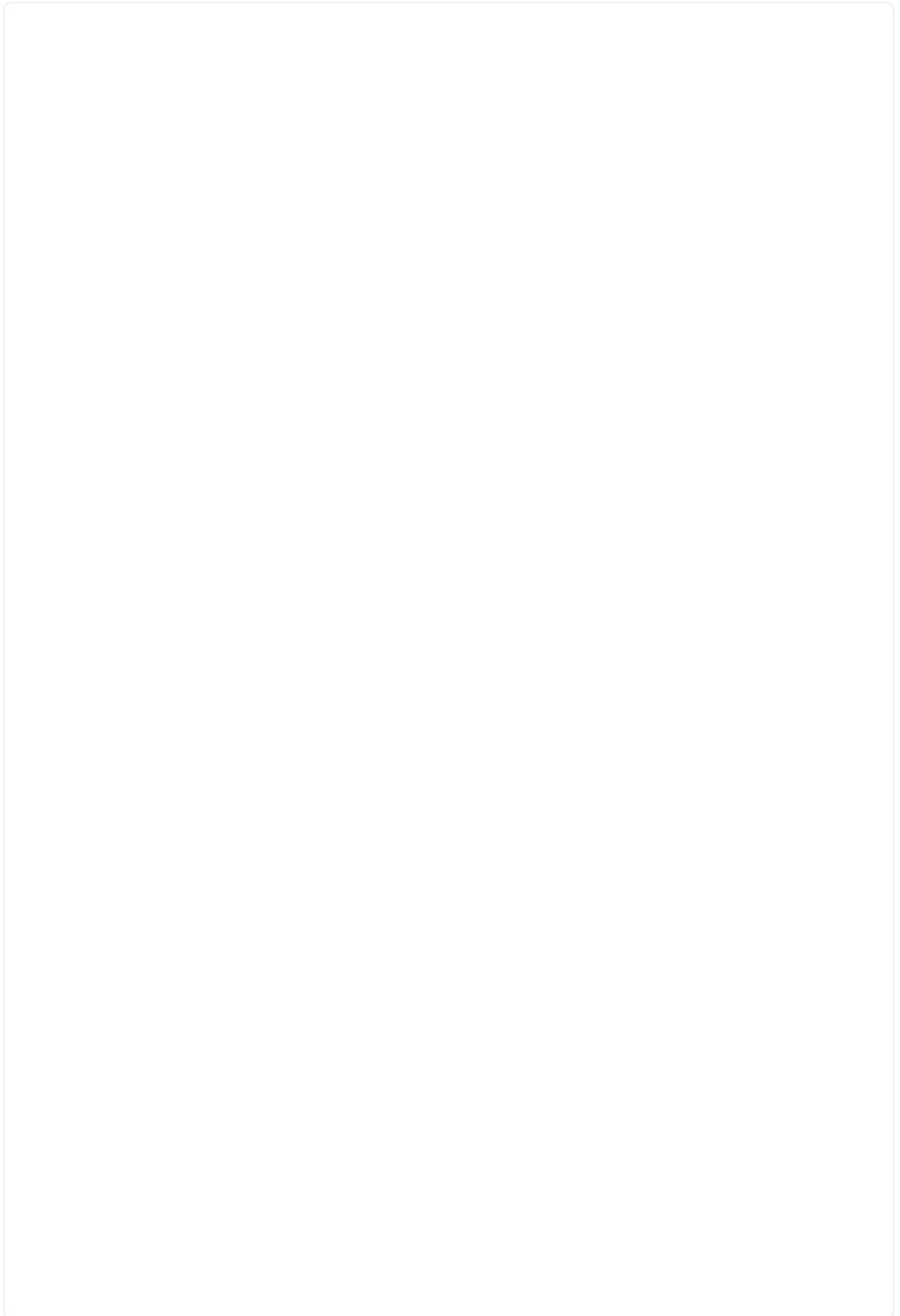
The predicted links are sorted by score before the ones having score below the configured **threshold** are discarded. Finally, the configured **topN** predictions are stored back to the in-memory graph.

It is necessary that the predict graph contains the properties that the pipeline requires and that the used array properties have the same dimensions as in the train graph. If the predict and train graphs are distinct, it is also beneficial that they have similar origins and semantics, so that the model is able to generalize

well.

Syntax

## Link Prediction syntax per mode



Run Link Prediction in mutate mode on a named graph:

```
CALL gds.alpha.ml.pipeline.linkPrediction.predict.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  postProcessingMillis: Integer,
  mutateMillis: Integer,
  relationshipsWritten: Integer,
  probabilityDistribution: Integer,
  samplingStats: Map,
  configuration: Map
```

Table 963. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 964. Configuration

Name	Type	Default	Optional	Description
modelName	String	n/a	no	The name of a Link Prediction model in the model catalog.
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.
mutateRelationshipType	String	n/a	no	The relationship type used for the new relationships written to the in-memory graph.
mutateProperty	String	'probability'	yes	The relationship property in the GDS graph to which the result is written.

Table 965. Algorithm specific configuration

Name	Type	Default	Optional	Description
sampleRate	Float	n/a	no	Sample rate to determine how many links are considered for each node. If set to 1, all possible links are considered, i.e., exhaustive search. Otherwise, a <a href="#">kNN-based</a> approximate search will be used. Value must be between 0 (exclusive) and 1 (inclusive).
topN <sup>[4]</sup>	Integer	n/a	no	Limit on predicted relationships to output.

Name	Type	Default	Optional	Description
threshold <sup>[4]</sup>	Float	0.0	yes	Minimum predicted probability on relationships to output.
topK <sup>[5]</sup>	Integer	10	yes	Limit on number of predicted relationships to output for each node. This value cannot be lower than 1.
deltaThreshold <sup>[5]</sup>	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).
maxIterations <sup>[5]</sup>	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.
randomJoins <sup>[5]</sup>	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.
randomSeed <sup>[5]</sup>	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <b>concurrency</b> must be set to 1 when setting this parameter.

Table 966. Results

Name	Type	Description
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing the global metrics.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
relationshipsWritten	Integer	Number of relationships created.
probabilityDistribution	Map	Description of distribution of predicted probabilities.
samplingStats	Map	Description of how predictions were sampled.
configuration	Map	Configuration used for running the algorithm.

Run Link Prediction in stream mode on a named graph:

```
CALL gds.alpha.ml.pipeline.linkPrediction.predict.stream(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  node1: Integer,  
  node2: Integer,  
  probability: Float
```

Table 967. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 968. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 969. Algorithm specific configuration

Name	Type	Default	Optional	Description
sampleRate	Float	n/a	no	Sample rate to determine how many links are considered for each node. If set to 1, all possible links are considered, i.e., exhaustive search. Otherwise, a <a href="#">kNN-based</a> approximate search will be used. Value must be between 0 (exclusive) and 1 (inclusive).
topN <sup>[6]</sup>	Integer	n/a	no	Limit on predicted relationships to output.
threshold <sup>[4]</sup>	Float	0.0	yes	Minimum predicted probability on relationships to output.
topK <sup>[7]</sup>	Integer	10	yes	Limit on number of predicted relationships to output for each node. This value cannot be lower than 1.
deltaThreshold <sup>[5]</sup>	Float	0.001	yes	Value as a percentage to determine when to stop early. If fewer updates than the configured value happen, the algorithm stops. Value must be between 0 (exclusive) and 1 (inclusive).
maxIterations <sup>[5]</sup>	Integer	100	yes	Hard limit to stop the algorithm after that many iterations.

Name	Type	Default	Optional	Description
randomJoins <sup>[5]</sup>	Integer	10	yes	Between every iteration, how many attempts are being made to connect new node neighbors based on random selection.
randomSeed <sup>[5]</sup>	Integer	n/a	yes	The seed value to control the randomness of the algorithm. Note that <b>concurrency</b> must be set to 1 when setting this parameter.

Table 970. Results

Name	Type	Description
node1	Integer	Node ID of the first node.
node2	Integer	Node ID of the second node.
probability	Float	Predicted probability of a link between the nodes.

## Example

In this example we will show how to use a trained model to predict new relationships in your in-memory graph. In order to do this, we must first have an already trained model registered in the Model Catalog. We will use the model which we trained in the [train example](#) which we gave the name `lp-pipeline-model`. The algorithm excludes predictions for existing relationships in the graph as well as self-loops.

There are two different strategies for choosing which node pairs to consider when predicting new links, exhaustive search and approximate search. Whereas the former considers all possible new links, the latter will use a randomized strategy that only considers a subset of them in order to run faster. We will explain each individually with examples in the [mutate examples](#) below.

## Stream

```
CALL gds.alpha.ml.pipeline.linkPrediction.predict.stream('myGraph', {
  modelName: 'lp-pipeline-model',
  topN: 5,
  threshold: 0.45
})
YIELD node1, node2, probability
RETURN gds.util.asNode(node1).name AS person1, gds.util.asNode(node2).name AS person2, probability
ORDER BY probability DESC, person1
```

We specified `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships.

Table 971. Results

person1	person2	probability
"Alice"	"Chris"	0.5422350772807373



person1	person2	probability
"Chris"	"Mark"	0.5364414066546659
"Alice"	"Mark"	0.5130009448848327
"Alice"	"Karin"	0.5123040606165334
"Alice"	"Greg"	0.51204718863418

We can see, that our model predicts the most likely link is between Alice and Chris.

### Mutate

In these examples we will show how to write the predictions to your in-memory graph. We will use the model `lp-pipeline-model`, that we trained in the [train example](#).

### Exhaustive search

The exhaustive search will simply run through all possible new links, that is, check all node pairs that are not already connected by a relationship. For each such node pair the model we trained will be used to predict whether there they should be connected be a link or not.

```
CALL gds.alpha.ml.pipeline.linkPrediction.predict.mutate('myGraph', {
  modelName: 'lp-pipeline-model',
  relationshipTypes: ['KNOWS'],
  mutateRelationshipType: 'KNOWS_EXHAUSTIVE_PREDICTED',
  topN: 5,
  threshold: 0.45
}) YIELD relationshipsWritten, samplingStats
```

We specify `threshold` to filter out predictions with probability less than 45%, and `topN` to further limit output to the top 5 relationships. Note that we omit setting the `sampleRate` in our configuration as it defaults to 1 implying that the exhaustive search strategy is used. Because we are using the `UNDIRECTED` orientation, we will write twice as many relationships to the in-memory graph.

Table 972. Results

relationshipsWritten	samplingStats
10	{linksConsidered=16, strategy=exhaustive}

As we can see in the `samplingStats`, we use the exhaustive search strategy and check 16 possible links during the prediction. Indeed, since there are a total of  $8 * (8 - 1) / 2 = 28$  possible links in the graph and we already have 12, that means we check all possible new links. Although 16 links were considered, we only mutate the best five (since `topN = 5`) that are above our threshold.

If our graph is very large there may be a lot of possible new links. As such it may take a very long time to run the predictions. It may therefore be a more viable option to use a search strategy that only looks at a subset of all possible new links.

## Approximate search

To avoid possibly having to run for a very long time considering all possible new links (due to the inherent quadratic complexity over node count) we can use an approximate search strategy.

The approximate search strategy lets us leverage the [K-Nearest Neighbors algorithm](#) with our model's prediction function as its similarity measure to trade off lower runtime for accuracy. Accuracy in this context refers to how close we are in finding the very best actual new possible links according to our models predictions, i.e. the best predictions that would be made by exhaustive search.

The initial set of considered links for each node is picked at random and then refined in multiple iterations based on previously predicted links. The number of iterations is limited by the configuration parameter `maxIterations`, and we also limit the number of random links considered between kNN iterations using `randomJoins`. The algorithm may stop earlier if the link predictions per node only change by a small amount, which can be controlled by the configuration parameter `deltaThreshold`. See the [K-Nearest Neighbors documentation](#) for more details on how the search works.

```
CALL gds.alpha.ml.pipeline.linkPrediction.predict.mutate('myGraph', {
  modelName: 'lp-pipeline-model',
  relationshipTypes: ['KNOWS'],
  mutateRelationshipType: 'KNOWS_APPROX_PREDICTED',
  sampleRate: 0.5,
  topK: 1,
  randomJoins: 2,
  maxIterations: 3,
  // necessary for deterministic results
  concurrency: 1,
  randomSeed: 42
})
YIELD relationshipsWritten, samplingStats
```

In order to use the approximate strategy we make sure to set the `sampleRate` explicitly to a value  $< 1.0$ . In this small example we set `topK = 1` to only get one link predicted for each node. Because we are using the `UNDIRECTED` orientation, we will write twice as many relationships to the in-memory graph.

Table 973. Results

relationshipsWritten	samplingStats
16	{linksConsidered=44, didConverge=true, strategy=approximate, ranIterations=2}

As we can see in the `samplingStats`, we use the approximate search strategy and check 44 possible links during the prediction. Though in this small example we actually consider more links than in the exhaustive case, this will typically not be the case for larger graphs. Since the relationships we write are undirected, reported `relationshipsWritten` is 16 when we search for the best (`topK = 1`) prediction for each node.

## 7.9. Auxiliary procedures

Auxiliary procedures are extra tools that can be useful in your workflow.

The Neo4j GDS library includes the following auxiliary procedures, grouped by quality tier:

- Beta
  - [Graph Generation](#)

- Alpha
  - [Collapse Path](#)
  - [Scale Properties](#)
  - [One Hot Encoding](#)
  - [Split Relationships](#)

## 7.9.1. Graph Generation Beta

In certain use cases it is useful to generate random graphs, for example, for testing or benchmarking purposes. For that reason the Neo4j Graph Algorithm library comes with a set of built-in graph generators. The generator stores the resulting graph in the [graph catalog](#). That graph can be used as input for any algorithm in the library.

This algorithm is in the beta tier. For more information on algorithm tiers, see [Algorithms](#).



It is currently not possible to persist these graphs in Neo4j. Running an algorithm in write mode on a generated graph will lead to unexpected results.

The graph generation is parameterized by three dimensions:

- node count - the number of nodes in the generated graph
- average degree - describes the average out-degree of the generated nodes
- relationship distribution function - the probability distribution method used to connect generated nodes

### Syntax

The following describes the API for running the algorithm

```
CALL gds.beta.graph.generate(graphName: String, nodeCount: Integer, averageDegree: Integer, {
  relationshipDistribution: String,
  relationshipProperty: Map
})
YIELD name, nodes, relationships, generateMillis, relationshipSeed, averageDegree,
relationshipDistribution, relationshipProperty
```

Table 974. Parameters

Name	Type	Default	Optional	Description
<code>graphName</code>	String	<code>null</code>	no	The name under which the generated graph is stored.
<code>nodeCount</code>	Integer	<code>null</code>	no	The number of generated nodes.
<code>averageDegree</code>	Integer	<code>null</code>	no	The average out-degree of generated nodes.
<code>configuration</code>	Map	<code>{}</code>	yes	Additional configuration, see below.

Table 975. Configuration

Name	Type	Default	Optional	Description
<code>relationshipDistribution</code>	String	<code>UNIFORM</code>	yes	The probability distribution method used to connect generated nodes. For more information see <a href="#">Relationship Distribution</a> .
<code>relationshipSeed</code>	Integer	<code>null</code>	yes	The seed used for generating relationships.
<code>relationshipProperty</code>	Map	<code>{}</code>	yes	Describes the method used to generate a relationship property. By default no relationship property is generated. For more information see <a href="#">Relationship Property</a> .
<code>aggregation</code>	String	<code>NONE</code>	yes	The relationship aggregation method cf. <a href="#">Relationship Projection</a> .
<code>orientation</code>	String	<code>NATURAL</code>	yes	The method of orienting edges. Allowed values are NATURAL, REVERSE and UNDIRECTED.
<code>allowSelfLoops</code>	Boolean	<code>false</code>	yes	Whether to allow relationships with identical source and target node.

Table 976. Results

Name	Type	Description
<code>name</code>	String	The name under which the stored graph was stored.
<code>nodes</code>	Integer	The number of nodes in the graph.
<code>relationships</code>	Integer	The number of relationships in the graph.
<code>generateMillis</code>	Integer	Milliseconds for generating the graph.
<code>relationshipSeed</code>	Integer	The seed used for generating relationships.
<code>averageDegree</code>	Float	The average out degree of the generated nodes.
<code>relationshipDistribution</code>	String	The probability distribution method used to connect generated nodes.
<code>relationshipProperty</code>	String	The configuration of the generated relationship property.

## Relationship Distribution

The `relationshipDistribution` parameter controls the statistical method used for the generation of new relationships. Currently there are three supported methods:

- **UNIFORM** - Distributes the outgoing relationships evenly, i.e., every node has exactly the same out degree (equal to the average degree). The target nodes are selected randomly.
- **RANDOM** - Distributes the outgoing relationships using a normal distribution with an average of `averageDegree` and a standard deviation of  $2 * \text{averageDegree}$ . The target nodes are selected randomly.
- **POWER\_LAW** - Distributes the incoming relationships using a power law distribution. The out degree is based on a normal distribution.

## Relationship Seed

The `relationshipSeed` parameter allows, to generate graphs with the same relationships, if they have no property. Currently the `relationshipProperty` is not seeded, therefore the generated graphs can differ in their property values. Hence generated graphs based on the same `relationshipSeed` are not identical.

## Relationship Property

The graph generator is capable of generating a relationship property. This can be controlled using the `relationshipProperty` parameter which accepts the following parameters:

Table 977. Configuration

Name	Type	Default	Optional	Description
<code>name</code>	String	null	no	The name under which the property values are stored.
<code>type</code>	String	null	no	The method used to generate property values.
<code>min</code>	Float	0.0	yes	Minimal value of the generated property (only supported by <code>RANDOM</code> ).
<code>max</code>	Float	1.0	yes	Maximum value of the generated property (only supported by <code>RANDOM</code> ).
<code>value</code>	Float	null	yes	Fixed value assigned to every relationship (only supported by <code>FIXED</code> ).

Currently, there are two supported methods to generate relationship properties:

- `FIXED` - Assigns a fixed value to every relationship. The `value` parameter must be set.
- `RANDOM` - Assigns a random value between the lower (`min`) and upper (`max`) bound.

### 7.9.2. Collapse Path Alpha

#### Introduction

The Collapse Path algorithm is a traversal algorithm capable of creating relationships between the start and end nodes of a traversal. In other words, the path between the start node and the end node is collapsed into a single relationship (a direct path). The algorithm is intended to support the creation of monopartite graphs required by many graph algorithms.

The main input for the algorithm is a list of relationship types. Starting from every node in the specified graph, these relationship types are traversed one after the other using the order specified in the configuration. Only nodes reached after traversing every relationship type specified are used as end nodes. Exactly one relationship is created for every pair of nodes for which at least one path from start to end node exists.

#### Syntax



Run Collapse Path in mutate mode on a named graph.

```
CALL gds.alpha.collapsePath.mutate(
  graphName: String,
  configuration: Map
)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  relationshipsWritten: Integer,
  configuration: Map
```

Table 978. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 979. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 980. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipTypes	List of String	n/a	no	Ordered list of relationship types used for the traversal. The same relationship type can be added multiple times, in order to traverse them as indicated.
mutateRelationshipType	String	n/a	no	Relationship type of the newly created relationships.
allowSelfLoops	Boolean	false	yes	Indicates whether it is possible to create self referencing relationships, i.e. relationships where the start and end node are identical.

Table 981. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
relationshipsWritten	Integer	The number of relationships created by the algorithm.

Name	Type	Description
configuration	Map	The configuration used for running the algorithm.

## Examples

Consider the graph created by the following Cypher statement:

```
CREATE
  (Dan:Person),
  (Annie:Person),
  (Matt:Person),
  (Jeff:Person),

  (Guitar:Instrument),
  (Flute:Instrument),

  (Dan)-[:PLAYS]->(Guitar),
  (Annie)-[:PLAYS]->(Guitar),

  (Matt)-[:PLAYS]->(Flute),
  (Jeff)-[:PLAYS]->(Flute)
```

In this example we want to create a relationship, called `PLAYS_SAME_INSTRUMENT`, between `Person` nodes that play the same instrument. To achieve that we have to traverse a path specified by the following Cypher pattern:

```
(p1:Person)-[:PLAYS]->(:Instrument)-[:PLAYED_BY]->(p2:Person)
```

In our source graph only the `PLAYS` relationship type exists. The `PLAYED_BY` relationship type can be created by loading the `PLAYS` relationship type in `REVERSE` direction. The following query will create such a graph:

```
CALL gds.graph.create(
  'persons',
  ['Person', 'Instrument'],
  {
    PLAYS: {
      type: 'PLAYS',
      orientation: 'NATURAL'
    },
    PLAYED_BY: {
      type: 'PLAYS',
      orientation: 'REVERSE'
    }
  }
})
```

Now we can run the algorithm by specifying the traversal `PLAYS`, `PLAYED_BY` in the `relationshipTypes` option.

```
CALL gds.alpha.collapsePath.mutate(
  'persons',
  {
    relationshipTypes: ['PLAYS', 'PLAYED_BY'],
    allowSelfLoops: false,
    mutateRelationshipType: 'PLAYS_SAME_INSTRUMENT'
  }
) YIELD relationshipsWritten
```



Table 982. Results

relationshipsWritten
4

The mutated graph will look like the following graph when filtered by the `PLAYS_SAME_INSTRUMENT` relationship

```
CREATE
  (Dan:Person),
  (Annie:Person),
  (Matt:Person),
  (Jeff:Person),

  (Guitar:Instrument),
  (Flute:Instrument),

  (Dan)-[:PLAYS_SAME_INSTRUMENT]->(Annie),
  (Annie)-[:PLAYS_SAME_INSTRUMENT]->(Dan),

  (Matt)-[:PLAYS_SAME_INSTRUMENT]->(Jeff),
  (Jeff)-[:PLAYS_SAME_INSTRUMENT]->(Matt),
```

### 7.9.3. Scale Properties Alpha

#### Introduction

The Scale Properties algorithm is a utility algorithm that is used to pre-process node properties for model training or post-process algorithm results such as PageRank scores. It scales the node properties based on the specified scaler. Multiple properties can be scaled at once and are returned in a list property.

The input properties must be numbers or lists of numbers. The lists must all have the same size. The output property will always be a list. The size of the output list is equal to the sum of length of the input properties. That is, if the input properties are two scalar numeric properties and one list property of length three, the output list will have a total length of five.

There are a number of supported scalers for the Scale Properties algorithm. These can be configured using the `scaler` configuration parameter.

List properties are scaled index-by-index. See [the list example](#) for more details.

In the following equations, `p` denotes the vector containing all property values for a single property across all nodes in the graph.

#### Min-max scaler

Scales all property values into the range `[0, 1]` where the minimum value(s) get the scaled value `0` and the maximum value(s) get the scaled value `1`, according to this formula:

$$p_{scaled} = \frac{p - \min(p)}{\max(p) - \min(p)}$$

## Max scaler

Scales all property values into the range [-1, 1] where the absolute maximum value(s) get the scaled value 1, according to this formula:

$$p_{scaled} = \frac{p}{|max(p)|}$$

## Mean scaler

Scales all property values into the range [-1, 1] where the average value(s) get the scaled value 0.

$$p_{scaled} = \frac{p - avg(p)}{max(p) - min(p)}$$

## Log scaler

Transforms all property values using the natural logarithm.

$$p_{scaled} = \ln(p)$$

## Standard Score

Scales all property values using the [Standard Score \(Wikipedia\)](#).

$$p_{scaled} = \frac{p - avg(p)}{std(p)}$$

## L1 Norm

Scales all property values into the range [0.0, 1.0].

$$p_{scaled} = \frac{p}{|p|_1}$$

## L2 Norm

Scales all property values using the [L2 Norm \(Wikipedia\)](#).

$$p_{scaled} = \frac{p}{||p||}$$

## Syntax

This section covers the syntax used to execute the Scale Properties algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).

Run Scale Properties in stream mode on a named graph.

```
CALL gds.alpha.scaleProperties.stream(  
  graphName: String,  
  configuration: Map  
) YIELD  
  nodeId: Integer,  
  scaledProperty: List of Float
```

Table 983. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 984. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 985. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeProperties	List of String	n/a	no	The names of the node properties that are to be scaled. All property names must exist in the in-memory graph.
scaler	String	n/a	no	The name of the scaler applied for the properties. Supported values are <code>MinMax</code> , <code>Max</code> , <code>Mean</code> , <code>Log</code> , <code>L1Norm</code> , <code>L2Norm</code> and <code>StdScore</code> .

Table 986. Results

Name	Type	Description
nodeId	Integer	Node ID.
scaledProperty	List of Float	Scaled values for each input node property.

Run Scale Properties in mutate mode on a named graph.

```
CALL gds.alpha.scaleProperties.mutate(
  graphName: String,
  configuration: Map
) YIELD
  createMillis: Integer,
  computeMillis: Integer,
  mutateMillis: Integer,
  postProcessingMillis: Integer,
  nodePropertiesWritten: Integer,
  configuration: Map
```

Table 987. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 988. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 989. Algorithm specific configuration

Name	Type	Default	Optional	Description
nodeProperties	List of String	n/a	no	The names of the node properties that are to be scaled. All property names must exist in the in-memory graph.
scaler	String	n/a	no	The name of the scaler applied for the properties. Supported values are <code>MinMax</code> , <code>Max</code> , <code>Mean</code> , <code>Log</code> , <code>L1Norm</code> , <code>L2Norm</code> and <code>StdScore</code> .

Table 990. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
postProcessingMillis	Integer	Unused.

Name	Type	Description
nodePropertiesWritten	Integer	Number of node properties written.
configuration	Map	Configuration used for running the algorithm.

## Examples

In this section we will show examples of running the Scale Properties algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small hotel graph of a handful nodes connected in a particular pattern. The example graph looks like this:



The following Cypher statement will create the example graph in the Neo4j database:

```

CREATE
(:Hotel {avgReview: 4.2, buildYear: 1978, storyCapacity: [32, 32, 0], name: 'East'}),
(:Hotel {avgReview: 8.1, buildYear: 1958, storyCapacity: [18, 20, 0], name: 'Plaza'}),
(:Hotel {avgReview: 19.0, buildYear: 1999, storyCapacity: [100, 100, 70], name: 'Central'}),
(:Hotel {avgReview: -4.12, buildYear: 2005, storyCapacity: [250, 250, 250], name: 'West'}),
(:Hotel {avgReview: 0.01, buildYear: 2020, storyCapacity: [1250, 1250, 900], name: 'Polar'}),
(:Hotel {avgReview: 3.3, buildYear: 1981, storyCapacity: [240, 240, 0], name: 'Beach'}),
(:Hotel {avgReview: 6.7, buildYear: 1984, storyCapacity: [80, 0, 0], name: 'Mountain'}),
(:Hotel {avgReview: -1.2, buildYear: 2010, storyCapacity: [55, 20, 0], name: 'Forest'})

```

With the graph in Neo4j we can now project it into the graph catalog to prepare it for algorithm execution. We do this using a native projection targeting the `Hotel` nodes, including their properties. Note that no relationships are necessary to scale the node properties. Thus we use a star projection (\*) for relationships.



In the examples below we will use named graphs and native projections as the norm. However, anonymous graphs and/or [Cypher projections](#) can also be used.

The following statement will create a graph using a native projection and store it in the graph catalog under the name 'myGraph'.

```
CALL gds.graph.create(
  'myGraph',
  'Hotel',
  '*',
  { nodeProperties: ['avgReview', 'buildYear', 'storyCapacity'] }
)
```

In the following examples we will demonstrate how to scale the node properties of this graph.

## Stream

In the `stream` execution mode, the algorithm returns the scaled properties for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects.

For more details on the `stream` mode in general, see [Stream](#).

The following will run the algorithm in `stream` mode:

```
CALL gds.alpha.scaleProperties.stream('myGraph', {
  nodeProperties: ['buildYear', 'avgReview'],
  scaler: 'MinMax'
}) YIELD nodeId, scaledProperty
RETURN gds.util.asNode(nodeId).name AS name, scaledProperty
ORDER BY name ASC
```

Table 991. Results

name	scaledProperty
"Beach"	[0.3709677419354839, 0.3209342560553633]
"Central"	[0.6612903225806451, 1.0]
"East"	[0.3225806451612903, 0.35986159169550175]
"Forest"	[0.8387096774193549, 0.12629757785467127]
"Mountain"	[0.41935483870967744, 0.4679930795847751]
"Plaza"	[0.0, 0.5285467128027681]
"Polar"	[1.0, 0.17863321799307957]
"West"	[0.7580645161290323, 0.0]

In the results we can observe that the first element in the resulting `scaledProperty` we get the min-max-scaled values for `buildYear`, where the `Plaza` hotel has the minimum value and is scaled to zero, while the `Polar` hotel has the maximum value and is scaled to one. This can be verified with the example graph. The second value in the `scaledProperty` result are the scaled values of the `avgReview` property.

## Mutate

The `mutate` execution mode enables updating the named graph with a new node property containing the scaled properties for that node. The name of the new property is specified using the mandatory configuration parameter `mutateProperty`. The result is a single summary row containing metrics from the computation. The `mutate` mode is especially useful when multiple algorithms are used in conjunction.

For more details on the `mutate` mode in general, see [Mutate](#).

In this example we will scale the two hotel properties of `buildYear` and `avgReview` using the [Mean scaler](#). The output is a list property which we will call `hotelFeatures`, imagining that we will use this as input for a machine learning model later on.

The following will run the algorithm in `mutate` mode:

```
CALL gds.alpha.scaleProperties.mutate('myGraph', {
  nodeProperties: ['buildYear', 'avgReview'],
  scaler: 'Mean',
  mutateProperty: 'hotelFeatures'
}) YIELD nodePropertiesWritten
```

Table 992. Results

nodePropertiesWritten
8

The result shows that there are now eight new node properties in the in-memory graph. These contain the scaled values from the input properties, where the scaled `buildYear` values are in the first list position and scaled `avgReview` values are in the second position. To find out how to inspect the new schema of the in-memory graph, see [Listing graphs in the catalog](#).

## List properties

The `storyCapacity` property models the amount of rooms on each story of the hotel. The property is normalized so that hotels with fewer stories have a zero value. This is because the Scale Properties algorithm requires that all values for the same property have the same length. In this example we will show how to scale the values in these lists using the Scale Properties algorithm. We imagine using the output as feature vector to input in a machine learning algorithm. Additionally, we will include the `avgReview` property in our feature vector.

The following will run the algorithm in `mutate` mode:

```
CALL gds.alpha.scaleProperties.stream('myGraph', {
  nodeProperties: ['avgReview', 'storyCapacity'],
  scaler: 'StdScore'
}) YIELD nodeId, scaledProperty
RETURN gds.util.asNode(nodeId).name AS name, scaledProperty AS features
ORDER BY name ASC
```

Table 993. Results

name	features
"Beach"	[-0.17956547594003253, -0.03401933556831381, 0.00254261210704973, -0.5187592498702616]
"Central"	[2.172199255871029, -0.3968922482969945, -0.3534230828799124, -0.2806402499298136]
"East"	[-0.0447509371737933, -0.5731448059080679, -0.526320706159294, -0.5187592498702616]
"Forest"	[-0.8536381697712284, -0.513529970245499, -0.5568320514438908, -0.5187592498702616]
"Mountain"	[0.32973389273242665, -0.4487312358296632, -0.6076842935848854, -0.5187592498702616]
"Plaza"	[0.5394453974799097, -0.609432097180936, -0.5568320514438908, -0.5187592498702616]
"Polar"	[-0.672387512096618, 2.583849534831454, 2.5705808402272767, 2.542770749364069]
"West"	[-1.2910364511016934, -0.00809984180197948, 0.027968733177547028, 0.3316657499170525]

The resulting feature vector contains the standard-score scaled value for the `avgReview` property in the first list position. We can see that some values are negative and that the maximum value sticks out for the `Central` hotel.

The other three list positions are the scaled values for the `storyCapacity` list property. Note that each list item is scaled only with respect to the corresponding item in the other lists. Thus, the `Polar` hotel has the greatest scaled value in all list positions.

## 7.9.4. One Hot Encoding Alpha

The One Hot Encoding function is used to convert categorical data into a numerical format that can be used by Machine Learning libraries.

This algorithm is in the alpha tier. For more information on algorithm tiers, see [Algorithms](#).

### One Hot Encoding sample

One hot encoding will return a list equal to the length of the `available values`. In the list, `selected values` are represented by `1`, and `unselected values` are represented by `0`.

The following will run the algorithm on hardcoded lists:

```
RETURN gds.alpha.ml.oneHotEncoding(['Chinese', 'Indian', 'Italian'], ['Italian']) AS embedding
```

Table 994. Results

embedding
[0,0,1]



The following will create a sample graph:

```
CREATE (french:Cuisine {name:'French'}),
      (italian:Cuisine {name:'Italian'}),
      (indian:Cuisine {name:'Indian'}),

      (zhen:Person {name: "Zhen"}),
      (praveena:Person {name: "Praveena"}),
      (michael:Person {name: "Michael"}),
      (arya:Person {name: "Arya"}),

      (praveena)-[:LIKES]->(indian),
      (zhen)-[:LIKES]->(french),
      (michael)-[:LIKES]->(french),
      (michael)-[:LIKES]->(italian)
```

The following will return a one hot encoding for each user and the types of cuisine that they like:

```
MATCH (cuisine:Cuisine)
WITH cuisine
ORDER BY cuisine.name
WITH collect(cuisine) AS cuisines
MATCH (p:Person)
RETURN p.name AS name, gds.alpha.ml.oneHotEncoding(cuisines, [(p)-[:LIKES]->(cuisine) | cuisine]) AS
embedding
ORDER BY name
```

Table 995. Results

name	embedding
Arya	[0,0,0]
Michael	[1,0,1]
Praveena	[0,1,0]
Zhen	[1,0,0]

Table 996. Parameters

Name	Type	Default	Optional	Description
<code>availableValues</code>	list	null	yes	The available values. If null, the function will return an empty list.
<code>selectedValues</code>	list	null	yes	The selected values. If null, the function will return a list of all 0's.

Table 997. Results

Type	Description
<code>list</code>	One hot encoding of the selected values.

## 7.9.5. Split Relationships Alpha

### Introduction

The Split relationships algorithm is a utility algorithm that is used to pre-process a graph for model training. It splits the relationships into a holdout set and a remaining set. The holdout set is divided into

two classes: positive, i.e., existing relationships, and negative, i.e., non-existing relationships. The class is indicated by a `label` property on the relationships. This enables the holdout set to be used for training or testing a machine learning model. Both, the holdout and the remaining relationships are added to the in-memory graph.

If the configuration option `relationshipWeightProperty` is specified, then the corresponding relationship property is preserved on the remaining set of relationships. Note however that the holdout set only has the `label` property; it is not possible to induce relationship weights on the holdout set as it also contains negative samples.

## Syntax

This section covers the syntax used to execute the Split Relationships algorithm in each of its execution modes. We are describing the named graph variant of the syntax. To learn more about general syntax variants, see [Syntax overview](#).



Run Split Relationships in mutate mode on a named graph.

```
CALL gds.alpha.ml.splitRelationships.mutate(  
  graphName: String,  
  configuration: Map  
)  
YIELD  
  createMillis: Integer,  
  computeMillis: Integer,  
  mutateMillis: Integer,  
  relationshipsWritten: Integer,  
  configuration: Map
```

Table 998. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph stored in the catalog.
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 999. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	List of String	['*']	yes	Filter the named graph using the given node labels.
relationshipTypes	List of String	['*']	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm.

Table 1000. Algorithm specific configuration

Name	Type	Default	Optional	Description
holdoutFraction	Float	n/a	no	The fraction of all relationships being used as holdout set.
negativeSamplingRatio	Float	n/a	no	The desired ratio of negative to positive samples in holdout set.
holdoutRelationshipType	String	n/a	no	Relationship type used for the holdout set. Each relationship has a property <code>label</code> indicating whether it is a positive or negative sample.
remainingRelationshipType	String	n/a	no	Relationship type used for the remaining set.
nonNegativeRelationshipTypes	List of String	n/a	yes	Additional relationship types that are used for negative sampling.
relationshipWeightProperty	String	null	yes	Name of the relationship property that is inherited by the <code>remainingRelationshipType</code> .

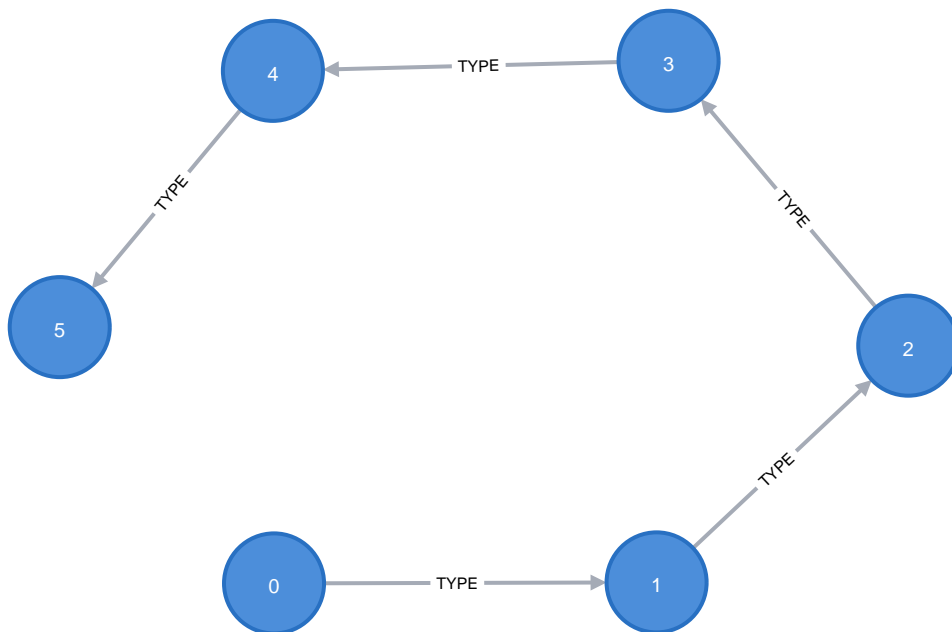
Name	Type	Default	Optional	Description
randomSeed	Integer	n/a	yes	An optional seed value for the random selection of relationships.

Table 1001. Results

Name	Type	Description
createMilliseconds	Integer	Milliseconds for loading data.
computeMilliseconds	Integer	Milliseconds for running the algorithm.
mutateMilliseconds	Integer	Milliseconds for adding properties to the in-memory graph.
relationshipsWritten	Integer	The number of relationships created by the algorithm.
configuration	Map	The configuration used for running the algorithm.

## Examples

In this section we will show examples of running the Split Relationships algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small graph of a handful nodes connected in a particular pattern. The example graph looks like this:



Consider the graph created by the following Cypher statement:

```

CREATE
  (n0:Label),
  (n1:Label),
  (n2:Label),
  (n3:Label),
  (n4:Label),
  (n5:Label),

  (n0)-[:TYPE { prop: 0} ]->(n1),
  (n1)-[:TYPE { prop: 1} ]->(n2),
  (n2)-[:TYPE { prop: 4} ]->(n3),
  (n3)-[:TYPE { prop: 9} ]->(n4),
  (n4)-[:TYPE { prop: 16} ]->(n5)

```

Given the above graph, we want to use 20% of the relationships as holdout set. The holdout set will be split into two same-sized classes: positive and negative. Positive relationships will be randomly selected from the existing relationships and marked with a property `label: 1`. Negative relationships will be randomly generated, i.e., they do not exist in the input graph, and are marked with a property `label: 0`.

```

CALL gds.graph.create(
  'graph',
  'Label',
  { TYPE: { orientation: 'UNDIRECTED' } }
)

```

Now we can run the algorithm by specifying the appropriate ratio and the output relationship types. We use a random seed value in order to produce deterministic results.

```

CALL gds.alpha.ml.splitRelationships.mutate('graph', {
  holdoutRelationshipType: 'TYPE_HOLDOUT',
  remainingRelationshipType: 'TYPE_REMAINING',
  holdoutFraction: 0.2,
  negativeSamplingRatio: 1.0,
  randomSeed: 1337
}) YIELD relationshipsWritten

```

Table 1002. Results

relationshipsWritten
10

The input graph consists of 5 relationships. We use 20% (1 relationship) of the relationships to create the 'TYPE\_HOLDOUT' relationship type (holdout set). This creates 1 relationship with positive label. Because of the `negativeSamplingRatio`, one relationship with negative label is also created. Finally, the `TYPE_REMAINING` relationship type is formed with the remaining 80% (4 relationships). These are written as orientation `UNDIRECTED` which counts as writing 8 relationships.

The mutated graph will look like the following graph when filtered by the **TEST** and **TRAIN** relationship.

```
CREATE
  (n0:Label),
  (n1:Label),
  (n2:Label),
  (n3:Label),
  (n4:Label),
  (n5:Label),

  (n2)-[:TYPE_HOLDOUT { label: 0 } ]->(n5), // negative, non-existing
  (n3)-[:TYPE_HOLDOUT { label: 1 } ]->(n2), // positive, existing

  (n0)-[:TYPE_REMAINING { prop: 0 } ]-(n1),
  (n1)-[:TYPE_REMAINING { prop: 1 } ]-(n2),
  (n3)-[:TYPE_REMAINING { prop: 9 } ]-(n4),
  (n4)-[:TYPE_REMAINING { prop: 16 } ]-(n5),
  (n0)-[:TYPE_REMAINING { prop: 0 } ]->(n1),
  (n1)-[:TYPE_REMAINING { prop: 1 } ]->(n2),
  (n3)-[:TYPE_REMAINING { prop: 9 } ]->(n4),
  (n4)-[:TYPE_REMAINING { prop: 16 } ]->(n5)
```

## 7.10. Pregel API

### 7.10.1. Introduction

Pregel is a vertex-centric computation model to define your own algorithms via a user-defined *compute* function. Node values can be updated within the compute function and represent the algorithm result. The input graph contains default node values or node values from a graph projection.

The compute function is executed in multiple iterations, also called *supersteps*. In each superstep, the compute function runs for each node in the graph. Within that function, a node can receive messages from other nodes, typically its neighbors. Based on the received messages and its currently stored value, a node can compute a new value. A node can also send messages to other nodes, typically its neighbors, which are received in the next superstep. The algorithm terminates after a fixed number of supersteps or if no messages are being sent between nodes.

A Pregel computation is executed in parallel. Each thread executes the compute function for a batch of nodes.

For more information about Pregel, have a look at [https://kowshik.github.io/JPregel/pregel\\_paper.pdf](https://kowshik.github.io/JPregel/pregel_paper.pdf).

To implement your own Pregel algorithm, the Graph Data Science library provides a Java API, which is described below.

The introduction of a new Pregel algorithm can be separated in two main steps. First, we need to implement the algorithm using the Pregel Java API. Second, we need to expose the algorithm via a Cypher procedure to make use of it.

For an example on how to expose a custom Pregel computation via a Neo4j procedure, have a look at the [Pregel examples](#).

### 7.10.2. Pregel Java API

The Pregel Java API allows us to easily build our own algorithm by implementing several interfaces.

## Computation

The first step is to implement the `org.neo4j.gds.beta.pregel.PregelComputation` interface. It is the main interface to express user-defined logic using the Pregel framework.

### The Pregel computation

```
public interface PregelComputation<C extends PregelConfig> {
    // The schema describes the node property layout.
    PregelSchema schema();
    // Called in the first superstep and allows initializing node state.
    default void init(PregelContext.InitContext<C> context) {}
    // Called in each superstep for each node and contains the main logic.
    void compute(PregelContext.ComputeContext<C> context, Pregel.Messages messages);
    // Called exactly once at the end of each superstep by a single thread.
    default void masterCompute(MasterComputeContext<C> context) {}
    // Used to combine all messages sent to a node to a single value.
    default Optional<Reducer> reducer() {
        return Optional.empty();
    }
    // Used to apply a relationship weight on a message.
    default double applyRelationshipWeight(double message, double relationshipWeight);
}
```

Pregel node values are composite values. The `schema` describes the layout of that composite value. Each element of the schema can represent either a primitive long or double value as well as arrays of those. The element is uniquely identified by a key, which is used to access the value during the computation. Details on schema declaration can be found in the [dedicated section](#).

The `init` method is called in the beginning of the first superstep of the Pregel computation and allows initializing node values. The interface defines an abstract `compute` method, which is called for each node in every superstep. Algorithm-specific logic is expressed within the `compute` method. The context parameter provides access to node properties of the in-memory graph and the algorithm configuration.

The `compute` method is called individually for each node in every superstep as long as the node receives messages or has not voted to halt yet. Since an implementation of `PregelComputation` is stateless, a node can only communicate with other nodes via messages. In each superstep, a node receives `messages` and can send new messages via the `context` parameter. Messages can be sent to neighbor nodes or any node if its identifier is known.

The `masterCompute` method is called exactly once at the end of each superstep. It is executed by a single thread and can be used to modify a global state based on the current computation state. Details on using a master computation can be found in the [dedicated section](#).

An optional `reducer` can be used to define a function that is being applied on messages sent to a single node. It takes two arguments, the current value and a message value, and produces a new value. The function is called repeatedly, once for each message that is sent to a node. Eventually, only one message will be received by the node in the next superstep. By defining a reducer, memory consumption and computation runtime can be improved significantly. Check the [dedicated section](#) for more details.

The `applyRelationshipWeight` method can be used to modify the message based on a relationship property. If the input graph has no relationship properties, i.e. is unweighted, the method is skipped.



## Pregel schema

In Pregel, each node is associated with a value which can be accessed from within the `compute` method. The value is typically used to represent intermediate computation state and eventually the computation result. To represent complex state, the node value is a composite type which consists of one or more named values. From the perspective of the `compute` function, each of these values can be accessed by its name.

When implementing a `PregelComputation`, one must override the `schema()` method. The following example shows the simplest possible example:

```
PregelSchema schema() {
    return PregelSchema.Builder().add("foobar", ValueType.LONG).build();
}
```

The node value consists of a single value named `foobar` which is of type `long`. A node value can be of any GDS-supported type, i.e. `long`, `double`, `long[]`, `double[]` and `float[]`.

We can add an arbitrary number of values to the schema:

```
PregelSchema schema() {
    return PregelSchema.Builder()
        .add("foobar", ValueType.LONG)
        .add("baz", ValueType.DOUBLE)
        .build();
}
```

Note, that each property consumes additional memory when executing the algorithm, which typically amounts to the number of nodes multiplied by the size of a single value (e.g. 64 Bit for a `long` or `double`).

The `add` method on the builder takes a third argument: `Visibility`. There are two possible values: `PUBLIC` (default) and `PRIVATE`. The visibility is considered during `procedure code generation` to indicate if the value is part of the Pregel result or not. Any value that has visibility `PUBLIC` will be part of the computation result and included in the result of the procedure, e.g., streamed to the caller, mutated to the in-memory graph or written to the database.

The following shows a schema where one value is used as result and a second value is only used during computation:

```
PregelSchema schema() {
    return PregelSchema.Builder()
        .add("result", ValueType.LONG, Visibility.PUBLIC)
        .add("tempValue", ValueType.DOUBLE, Visibility.PRIVATE)
        .build();
}
```

## Init context and compute context

The main purpose of the two context objects is to enable the computation to communicate with the Pregel framework. A context is stateful, and all its methods are subject to the current superstep and the currently processed node. Both context objects share a set of methods, e.g., to access the config and node state. Additionally, each context adds context-specific methods.

The `org.neo4j.gds.beta.pregel.PregelContext.InitContext` is available in the `init` method of a Pregel computation. It provides access to node properties stored in the in-memory graph. We can set the initial node state to a fixed value, e.g. the node id, or use graph properties and the user-defined configuration to initialize a context-dependent state.

### The InitContext

```
public final class InitContext {
    // The currently processed node id.
    public long nodeId();
    // User-defined Pregel configuration
    public PregelConfig config();
    // Sets a double node value for the given schema key.
    public void setNodeValue(String key, double value);
    // Sets a long node value for the given schema key.
    public void setNodeValue(String key, long value);
    // Sets a double array node value for the given schema key.
    public void setNodeValue(String key, double[] value);
    // Sets a long array node value for the given schema key.
    public void setNodeValue(String key, long[] value);
    // Number of nodes in the input graph.
    public long nodeCount();
    // Number of relationships in the input graph.
    public long relationshipCount();
    // Number of relationships of the current node.
    public int degree();
    // Available node property keys in the input graph.
    public Set<String> nodePropertyKeys();
    // Node properties stored in the input graph.
    public NodeProperties nodeProperties(String key);
}
```

In contrast, `org.neo4j.gds.beta.pregel.PregelContext.ComputeContext` can be accessed inside the `compute` method. The context provides methods to access the computation state, e.g. the current superstep, and to send messages to other nodes in the graph.

## The ComputeContext

```
public final class ComputeContext {
    // The currently processed node id.
    public long nodeId();
    // User-defined Pregel configuration
    public PregelConfig config();
    // Sets a double node value for the given schema key.
    public void setNodeValue(String key, double value);
    // Sets a long node value for the given schema key.
    public void setNodeValue(String key, long value);
    // Number of nodes in the input graph.
    public long nodeCount();
    // Number of relationships in the input graph.
    public long relationshipCount();
    // Indicates whether the input graph is a multi-graph.
    public boolean isMultiGraph();
    // Number of relationships of the current node.
    public int degree();
    // Double value for the given node schema key.
    public double doubleNodeValue(String key);
    // Double value for the given node schema key.
    public long longNodeValue(String key);
    // Double array value for the given node schema key.
    public double[] doubleArrayNodeValue(String key);
    // Long array value for the given node schema key.
    public long[] longArrayNodeValue(String key);
    // Notify the framework that the node intends to stop its computation.
    public void voteToHalt();
    // Indicates whether this is superstep 0.
    public boolean isInitialSuperstep();
    // 0-based superstep identifier.
    public int superstep();
    // Sends the given message to all neighbors of the node.
    public void sendToNeighbors(double message);
    // Sends the given message to the target node.
    public void sendTo(long targetNodeId, double message);
    // Stream of neighbor ids of the current node.
    public LongStream getNeighbours();
}
```

## Master Computation

Some Pregel programs may require logic that is executed after all threads have finished the current superstep, for example, to reset or evaluate a global data structure. This can be achieved by overriding the `org.neo4j.gds.beta.pregel.PregelComputation.masterCompute` function of the `PregelComputation`. This function will be called at the end of each superstep after all compute threads have finished. The master compute function will be called by a single thread.

The `masterCompute` function has access to the `org.neo4j.gds.beta.pregel.PregelContext.MasterComputeContext`. That context is similar to the `ComputeContext` but is not tied to a specific node and does not allow sending messages. Furthermore, the `MasterComputeContext` allows to run a function for every node in the graph and has access to the computation state of all nodes.

## The MasterComputeContext

```
public final class MasterComputeContext {
    // User-defined Pregel configuration
    public PregelConfig config();
    // Number of nodes in the input graph.
    public long nodeCount();
    // Number of relationships in the input graph.
    public long relationshipCount();
    // Indicates whether the input graph is a multi-graph.
    public boolean isMultiGraph();
    // Run the given consumer for every node in the graph.
    public void forEachNode(LongPredicate consumer);
    // Double value for the given node schema key.
    public double doubleNodeValue(long nodeId, String key);
    // Double value for the given node schema key.
    public long longNodeValue(long nodeId, String key);
    // Double array value for the given node schema key.
    public double[] doubleArrayNodeValue(long nodeId, String key);
    // Long array value for the given node schema key.
    public long[] longArrayNodeValue(long nodeId, String key);
    // Sets a double node value for the given schema key.
    public void setNodeValue(long nodeId, String key, double value);
    // Sets a long node value for the given schema key.
    public void setNodeValue(long nodeId, String key, long value);
    // Sets a double array node value for the given schema key.
    public void setNodeValue(long nodeId, String key, double[] value);
    // Sets a long array node value for the given schema key.
    public void setNodeValue(long nodeId, String key, long[] value);
    // Indicates whether this is superstep 0.
    public boolean isInitialSuperstep();
    // 0-based superstep identifier.
    public int superstep();
}
```

## Message reducer

Many Pregel computations rely on computing a single value from all messages being sent to a node. For example, the page rank algorithm computes the sum of all messages being sent to a single node. In those cases, a reducer can be used to combine all messages to a single value. If applicable, this optimization improves memory consumption and computation runtime.

By default, a Pregel computation does not make use of a reducer. All messages sent to a node are stored in a queue and received in the next superstep. To enable message reduction, one needs to implement the `reducer` method and provide either a custom or a pre-defined reducer.

The *Reducer* interface that needs to be implemented.

```
public interface Reducer {
    // The identity element is used as the initial value.
    double identity();
    // Computes a new value based on the current value and the message.
    double reduce(double current, double message);
}
```

The identity value is used as the initial value for the `current` argument in the `reduce` function. All subsequent calls use the result of the previous call as `current` value.

The framework already provides implementations for computing the minimum, maximum, sum and count of messages. The default implementations are part of the `Reducer` interface and can be applied as follows:

Applying the sum reducer in a custom computation.

```
public class CustomComputation implements PregelComputation<PregelConfig> {
    @Override
    public void compute(PregelContext.ComputeContext<CustomConfig> context, Pregel.Messages messages) {
        // ...
        for (var message : messages) {
            // ...
        }
    }

    @Override
    public Optional<Reducer> reducer() {
        return Optional.of(new Reducer.Sum());
    }
}
```

The implementation of the compute method does not need to be adapted. If a reducer is present, the `messages` iterator contains either zero or one message. Note, that defining a reducer precludes running the computation with asynchronous messaging. The `isAsynchronous` flag at the config is ignored in that case.

## Configuration

To configure the execution of a custom Pregel computation, the framework requires a configuration. The `org.neo4j.gds.beta.pregel.PregelConfig` provides the minimum set of options to execute a computation. The configuration options also map to the parameters that can later be set via a custom procedure. This is equivalent to all the other algorithms within the GDS library.

Table 1003. Pregel Configuration

Name	Type	Default Value	Description
<code>maxIterations</code>	Integer	-	Maximum number of supersteps after which the computation will terminate.
<code>isAsynchronous</code>	Boolean	false	Flag indicating if messages can be sent and received in the same superstep.
<code>partitioning</code>	String	"range"	Selects the partitioning of the input graph, can be either "range", "degree" or "auto".
<code>relationshipWeightProperty</code>	String	null	Name of the relationship property to use as weights. If unspecified, the algorithm runs unweighted.
<code>concurrency</code>	Integer	4	Concurrency used when executing the Pregel computation.
<code>writeConcurrency</code>	Integer	concurrency	Concurrency used when writing computation results to Neo4j.
<code>writeProperty</code>	String	"pregel_"	Prefix string that is prepended to node schema keys in write mode.
<code>mutateProperty</code>	String	"pregel_"	Prefix string that is prepended to node schema keys in mutate mode.

For some algorithms, we want to specify additional configuration options.

Typically, these options are algorithm specific arguments, such as thresholds. Another reason for a custom config relates to the initialization phase of the computation. If we want to init the node state based on a graph property, we need to access that property via its key. Since those keys are dynamic properties of the graph, we need to provide them to the computation. We can achieve that by declaring an option to set that key in a custom configuration.

If a user-defined Pregel computation requires custom options a custom configuration can be created by extending the `PregelConfig`.

*A custom configuration and how it can be used in the init phase.*

```
@ValueClass
@Configuration
public interface CustomConfig extends PregelConfig {
    // A property key that refers to a seed property.
    String seedProperty();
    // An algorithm specific parameter.
    int minDegree();
}

public class CustomComputation implements PregelComputation<CustomConfig> {

    @Override
    public void init(PregelContext.InitContext<CustomConfig> context) {
        // Use the custom config key to access a graph property.
        var seedProperties = context.nodeProperties(context.config().seedProperty());
        // Init the node state with the graph property for that node.
        context.setNodeValue("state", seedProperties.doubleValue(context.nodeId()));
    }

    @Override
    public void compute(PregelContext.ComputeContext<CustomConfig> context, Pregel.Messages messages) {
        if (context.degree() >= context.config().minDegree()) {
            // ...
        }
    }

    // ...
}
```

### 7.10.3. Run Pregel via Cypher

To make a custom Pregel computation accessible via Cypher, it needs to be exposed via the procedure API. The Pregel framework in GDS provides an easy way to generate procedures for all the default modes.

#### Procedure generation

To generate procedures for a computation, it needs to be annotated with the `@org.neo4j.gds.beta.pregel.annotation.PregelProcedure` annotation. In addition, the config parameter of the custom computation must be a subtype of `org.neo4j.gds.beta.pregel.PregelProcedureConfig`.

*Using the `@PregelProcedure` annotation to configure code generation.*

```
@PregelProcedure(
    name = "custom.pregel.proc",
    modes = {GDSMode.STREAM, GDSMode.WRITE},
    description = "My custom Pregel algorithm"
)
public class CustomComputation implements PregelComputation<PregelProcedureConfig> {
    // ...
}
```

The annotation provides a number of configuration options for the code generation.

Table 1004. Configuration

Name	Type	Default Value	Description
name	String	-	The prefix of the generated procedure name. It is appended by the mode.
modes	List	[STREAM, WRITE, MUTATE, STATS]	A procedure is generated for each of the specified modes.
description	String	""	Procedure description that is printed in <code>dbms.listProcedures()</code> .

For the above Code snippet, we generate four procedures:

- `custom.pregel.proc.stream`
- `custom.pregel.proc.stream.estimate`
- `custom.pregel.proc.write`
- `custom.pregel.proc.write.estimate`

Note that by default, all values specified in the `PregelSchema` are included in the procedure results. To change that behaviour, we can change the visibility for individual parts of the schema. For more details, please refer to the [dedicated documentation section](#).

## Building and installing a Neo4j plugin

In order to use a Pregel algorithm in Neo4j via a procedure, we need to package it as Neo4j plugin. The `pregel-bootstrap` project is a good starting point. The `build.gradle` file within the project contains all the dependencies necessary to implement a Pregel algorithm and to generate corresponding procedures.

Make sure to change the `gdsVersion` and `neo4jVersion` according to your setup. GDS and Neo4j are runtime dependencies. Therefore, GDS needs to be installed as a plugin on the Neo4j server.

To build the project and create a plugin jar, just run:

```
./gradlew shadowJar
```

You can find the `pregel-bootstrap.jar` in `build/libs`. The jar needs to be placed in the `plugins` directory within your Neo4j installation alongside a GDS plugin jar. In order to have access to the procedure in Cypher, its namespace potentially needs to be added to the `neo4j.conf` file.

Enabling an example procedure in `neo4j.conf`

```
dbms.security.procedures.unrestricted=custom.pregel.proc.*
dbms.security.procedures.allowlist=custom.pregel.proc.*
```



Before Neo4j 4.2, the configuration setting is called `dbms.security.procedures.whitelist`

## 7.10.4. Examples

The [pregel-examples](#) module contains a set of examples for Pregel algorithms. The algorithm implementations demonstrate the usage of the Pregel API. Along with each example, we provide test classes that can be used as a guideline on how to write tests for custom algorithms. To play around, we recommend copying one of the algorithms into the [pregel-bootstrap](#) project, build it and setup the plugin in Neo4j.

[4] Only applicable in the exhaustive search.

[5] Only applicable in the approximate strategy. For more details look at the [syntax section of kNN](#)

[6] Only applicable in the exhaustive search.

[7] Only applicable in the approximate strategy. For more details look at the [syntax section of kNN](#)



# Chapter 8. End-to-end examples

For each algorithm in the [Algorithms pages](#) we have small examples of limited scope that demonstrate the usage of that particular algorithm, typically only using that one algorithm. The purpose of this section is show how the algorithms in GDS can be used to solve fairly realistic use cases end-to-end, typically using several algorithms in each example.

- [Product recommendation engine using FastRP and kNN](#)

## 8.1. FastRP and kNN example

In this example we consider a graph of products and customers, and we want to find new products to recommend for each customer. We want to use the [K-Nearest Neighbors algorithm \(kNN\)](#) to identify similar customers and base our product recommendations on that. In order to be able to leverage topological information about the graph in kNN, we will first create node embeddings using [FastRP](#). These embeddings will then be the input to the kNN algorithm.

For each pair of similar customers we can then recommend products that have been purchased by one of the customers but not the other, using a simple cypher query.

### 8.1.1. Graph creation

We will start by creating our graph of products and customers in the database. The `amount` relationship property represents the average weekly amount of money spent by a customer on a given product.

Consider the graph created by the following Cypher statement:

```
CREATE
(dan:Person {name: 'Dan'}),
(annie:Person {name: 'Annie'}),
(matt:Person {name: 'Matt'}),
(jeff:Person {name: 'Jeff'}),
(brie:Person {name: 'Brie'}),
(elsa:Person {name: 'Elsa'}),

(cookies:Product {name: 'Cookies'}),
(tomatoes:Product {name: 'Tomatoes'}),
(cucumber:Product {name: 'Cucumber'}),
(celesty:Product {name: 'Celery'}),
(kale:Product {name: 'Kale'}),
(milk:Product {name: 'Milk'}),
(chocolate:Product {name: 'Chocolate'}),

(dan)-[:BUYS {amount: 1.2}]->(cookies),
(dan)-[:BUYS {amount: 3.2}]->(milk),
(dan)-[:BUYS {amount: 2.2}]->(chocolate),

(annie)-[:BUYS {amount: 1.2}]->(cucumber),
(annie)-[:BUYS {amount: 3.2}]->(milk),
(annie)-[:BUYS {amount: 3.2}]->(tomatoes),

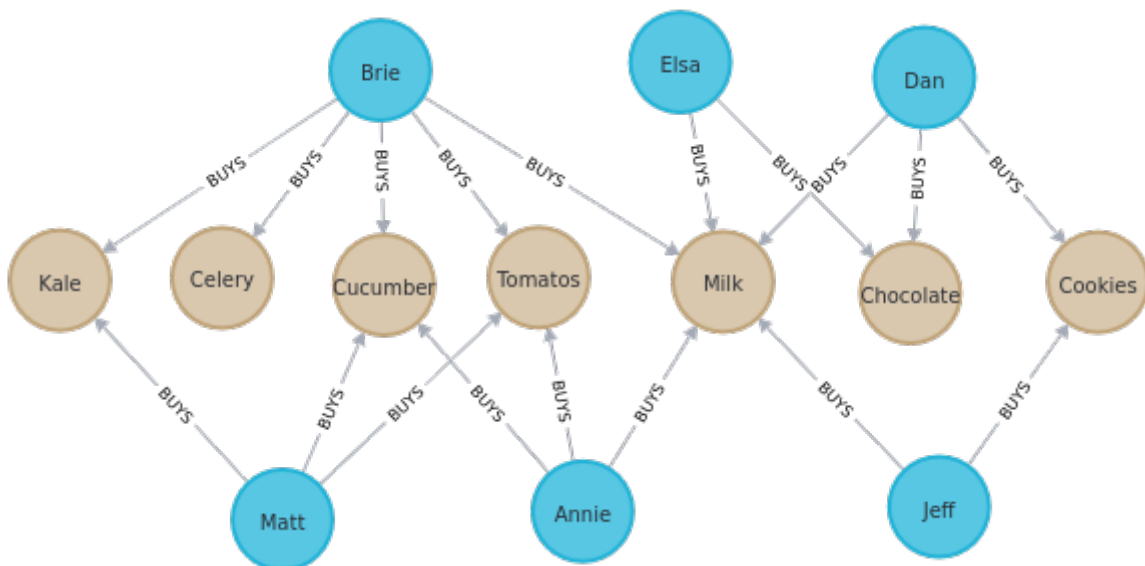
(matt)-[:BUYS {amount: 3}]->(tomatoes),
(matt)-[:BUYS {amount: 2}]->(kale),
(matt)-[:BUYS {amount: 1}]->(cucumber),

(jeff)-[:BUYS {amount: 3}]->(cookies),
(jeff)-[:BUYS {amount: 2}]->(milk),

(brie)-[:BUYS {amount: 1}]->(tomatoes),
(brie)-[:BUYS {amount: 2}]->(milk),
(brie)-[:BUYS {amount: 2}]->(kale),
(brie)-[:BUYS {amount: 3}]->(cucumber),
(brie)-[:BUYS {amount: 0.3}]->(celesty),

(elsa)-[:BUYS {amount: 3}]->(chocolate),
(elsa)-[:BUYS {amount: 3}]->(milk);
```

The graph can be visualized in the following way:



Now we can proceed to create an in-memory graph which we can run the algorithms on.

Create the in-memory graph and store it in the graph catalog:

```
CALL gds.graph.create(  
  'purchases',  
  ['Person', 'Product'],  
  {  
    BUYS: {  
      orientation: 'UNDIRECTED',  
      properties: 'amount'  
    }  
  }  
)
```

## 8.1.2. FastRP embedding

Now we run the FastRP algorithm to generate node embeddings that capture topological information from the graph. We choose to work with `embeddingDimension` set to 4 which is sufficient since our example graph is very small. The `iterationWeights` are chosen empirically to yield sensible results. Please see the [syntax section](#) of the FastRP documentation for more information on these parameters. Since we want to use the embeddings as input when we run kNN later we use FastRP's `mutate mode`.

Create node embeddings using FastRP:

```
CALL gds.fastRP.mutate('purchases',  
  {  
    embeddingDimension: 4,  
    randomSeed: 42,  
    mutateProperty: 'embedding',  
    relationshipWeightProperty: 'amount',  
    iterationWeights: [0.8, 1, 1, 1]  
  }  
)  
YIELD nodePropertiesWritten
```

Table 1005. Results

nodePropertiesWritten
13

## 8.1.3. Similarities with kNN

Now we can run kNN to identify similar nodes by using the node embeddings that we generated with FastRP as `nodeWeightProperty`. Since we are working with a small graph, we can set `sampleRate` to 1 and `deltaThreshold` to 0 without having to worry about long computation times. The `concurrency` parameter is set to 1 (along with the fixed `randomSeed`) in order to get a deterministic result. Please see the [syntax section](#) of the kNN documentation for more information on these parameters. Note that we use the algorithm's `write mode` to write the properties and relationships back to our database, so that we can analyze them later using Cypher.

Run kNN with FastRP node embeddings as input:

```
CALL gds.beta.knn.write('purchases', {
  topK: 2,
  nodeWeightProperty: 'embedding',
  randomSeed: 42,
  concurrency: 1,
  sampleRate: 1.0,
  deltaThreshold: 0.0,
  writeRelationshipType: "SIMILAR",
  writeProperty: "score"
})
YIELD nodesCompared, relationshipsWritten, similarityDistribution
RETURN nodesCompared, relationshipsWritten, similarityDistribution.mean as meanSimilarity
```

Table 1006. Results

nodesCompared	relationshipsWritten	meanSimilarity
13	26	0.8341259589562049

As we can see the mean similarity between nodes is quite high. This is due to the fact that we have a small example where there are no long paths between nodes leading to many similar FastRP node embeddings.

## 8.1.4. Results exploration

Let us now inspect the results of our kNN call by using Cypher. We can use the **SIMILARITY** relationship type to filter out the relationships we are interested in. And since we just care about similarities between people for our product recommendation engine, we make sure to only match nodes with the **Person** label.

List pairs of people that are similar:

```
MATCH (n:Person)-[r:SIMILAR]->(m:Person)
RETURN n.name as person1, m.name as person2, r.score as similarity
ORDER BY similarity DESCENDING, person1, person2
```

Table 1007. Results

person1	person2	similarity
"Annie"	"Matt"	0.9661740064620972
"Matt"	"Annie"	0.9661740064620972
"Dan"	"Elsa"	0.9606010317802429
"Elsa"	"Dan"	0.9606010317802429
"Jeff"	"Annie"	0.6309423446655273

Our kNN results indicate among other things that the **Person** nodes named "Annie" and "Matt" are very similar. Looking at the **BUYS** relationships for these two nodes we can see that such a conclusion makes sense. They both buy three products, two of which are the same (**Product** nodes named "Cucumber" and "Tomatoes") for both people and with similar amounts. We therefore have high confidence in our approach.

## 8.1.5. Making recommendations

Using the information we derived that the **Person** nodes named "Annie" and "Matt" are similar, we can make product recommendations for each of them. Since they are similar, we can assume that products purchased by only one of the people may be of interest to buy also for the other person not already buying the product. By this principle we can derive product recommendations for the **Person** named "Matt" using a simple Cypher query.

Product recommendations for **Person** node with name "Matt":

```
MATCH (:Person {name: "Annie"})-->(p1:Product)
WITH collect(p1) as products
MATCH (:Person {name: "Matt"})-->(p2:Product)
WHERE not p2 in products
RETURN p2.name as recommendation
```

Table 1008. Results

recommendation
"Kale"

Indeed, "Kale" is the one product that the **Person** named "Annie" buys that is also not purchased by the **Person** named "Matt".

## 8.1.6. Conclusion

Using two GDS algorithms and some basic Cypher we were easily able to derive some sensible product recommendations for a customer in our small example.

To make sure to get similarities to other customers for every customer in our graph with kNN, we could play around with increasing the **topK** parameter.

# Chapter 9. Production deployment

This chapter is divided into the following sections:

- [Transaction Handling](#)
- [Using GDS and Fabric](#)
- [GDS Feature Toggles](#)

## 9.1. Transaction Handling

### 9.1.1. During graph projection

During graph projection, new transactions are used that do not inherit the transaction state of the Cypher transaction. This means that changes from the Cypher transaction state are not visible to the graph projection transactions.

For example, the following statement will only create an empty graph (assuming the `MyLabel` label was not already present in the Neo4j database):

```
CREATE (n:MyLabel) // the new node is part of Cypher transaction state
WITH *
CALL gds.graph.create('myGraph', 'MyLabel', '*')
YIELD nodeCount
RETURN nodeCount
```

Table 1009. Results

nodeCount
0

The situation is the same when using an anonymous projection with an algorithm procedure:

```
CREATE (n:MyWccLabel) // the new node is part of Cypher transaction state
WITH *
CALL gds.wcc.stats({nodeProjection: 'MyWccLabel', relationshipProjection: '*'})
YIELD componentCount
RETURN componentCount
```

Table 1010. Results

componentCount
0

### 9.1.2. During results writing

Results from algorithms (node properties, for example) are written to the graph in new transactions. The number of transactions used depends on the size of the results and the `writeConcurrency` configuration parameter (for more details, please refer to sections [Write](#) and [Common Configuration parameters](#)). These

transactions are committed independently from the Cypher transaction. This means, if the Cypher transaction is terminated (either by the user or by the database system), already committed write transactions will not be rolled back.

## Transaction writing examples



The code in this section is for illustrative purposes. The goal is to demonstrate correct usage of the GDS library write functionality with Cypher Shell and Java API.

### Cypher Shell

Example for incorrect use.

```
:BEGIN

// Create the in-memory graph
CALL gds.graph.create.cypher(
  'test',
  'MATCH (n) WHERE n:Artist OR n:Genre RETURN id(n) AS id',
  'MATCH (a:Artist)-[:RELEASED_BY]-(:Album)-[:HAS_GENRE]->(g:Genre)
  RETURN id(g) AS source, id(a) AS target, "IS_ASSOCIATED_WITH" AS type'
);

// Delete the old stuff
MATCH ()-[r:SIMILAR_TO]->() DELETE r;

// Run the algorithm
CALL gds.nodeSimilarity.write(
  'test', {
    writeRelationshipType: 'SIMILAR_TO',
    writeProperty: 'score'
  }
);

:COMMIT
```

The issue with the above statement is that all the queries run in the same transaction.

A correct handling of the above statement would be to run each statement in its own transaction, which is shown below. Notice the reordering of the statements, this ensures that the in-memory graph will have the most recent changes after the removal of the relationships.

First remove the unwanted relationships.

```
:BEGIN

MATCH ()-[r:SIMILAR_TO]->() DELETE r;

:COMMIT
```

Create the in-memory graph.

```

:BEGIN

CALL gds.graph.create.cypher(
  'test',
  'MATCH (n) WHERE n:Artist OR n:Genre RETURN id(n) AS id',
  'MATCH (a:Artist)-[:RELEASED_BY]-(:Album)-[:HAS_GENRE]->(g:Genre)
  RETURN id(g) AS source, id(a) AS target, "IS_ASSOCIATED_WITH" AS type'
);

:COMMIT

```

Run the algorithm.

```

:BEGIN

CALL gds.nodeSimilarity.write(
  'test', {
    writeRelationshipType: 'SIMILAR_TO',
    writeProperty: 'score'
  }
);

:COMMIT

```

Java API

The same issue can be seen using the Java API, the examples are below.

Constants used throughout the examples below:

```

// Removes the in-memory graph (if exists) from the graph catalog
static final String CYPHER_DROP_GDS_GRAPH_IF_EXISTS =
  "CALL gds.graph.drop('test', false)";

// Creates the in-memory graph
static final String CYPHER_CREATE_GDS_GRAPH_ARTIST_GENRE =
  "CALL gds.graph.create.cypher(" +
  "  'test', " +
  "  'MATCH (n) WHERE n:Artist OR n:Genre RETURN id(n) AS id', " +
  "  'MATCH (a:Artist)-[:RELEASED_BY]-(:Album)-[:HAS_GENRE]->(g:Genre) " +
  "    RETURN id(g) AS source, id(a) AS target, \"IS_ASSOCIATED_WITH\" AS type'" +
  ")";

// Runs NodeSimilarity in `write` mode over the in-memory graph
static final String CYPHER_WRITE_SIMILAR_TO =
  "CALL gds.nodeSimilarity.write(" +
  "  'test', {" +
  "    writeRelationshipType: 'SIMILAR_TO'," +
  "    writeProperty: 'score'" +
  "  }" +
  ")";

```

Incorrect use:

```

try (var session = driver.session()) {
  var params = Map.<String, Object>of("graphName", "genre-related-to-artist");
  session.writeTransaction(tx -> {
    tx.run(CYPHER_DROP_GDS_GRAPH_IF_EXISTS, params).consume();
    tx.run(CYPHER_CREATE_GDS_GRAPH_ARTIST_GENRE, params).consume();
    tx.run("MATCH ()-[r:SIMILAR_TO]->() DELETE r").consume();
    return tx.run(CYPHER_WRITE_SIMILAR_TO, params).consume();
  });
}

```



Here we are facing the same issue with running everything in the same transaction. This can be written correctly by splitting each statement in its own transaction.

Correct handling of the statements:

```
try (var session = driver.session()) {  
    // First run the remove statement  
    session.writeTransaction(tx -> {  
        return tx.run("MATCH ()-[r:SIMILAR_TO]->() DELETE r").consume();  
    });  
  
    // Create the in-memory graph  
    var params = Map.<String, Object>of("graphName", "genre-related-to-artist");  
    session.writeTransaction(tx -> {  
        tx.run(CYPHER_DROP_GDS_GRAPH_IF_EXISTS, params).consume();  
        return tx.run(CYPHER_CREATE_GDS_GRAPH_ARTIST_GENRE, params).consume();  
    });  
  
    // Run the algorithm  
    session.writeTransaction(tx -> {  
        return tx.run(CYPHER_WRITE_SIMILAR_TO, params).consume();  
    });  
}
```

# Chapter 10. Transaction termination

The Cypher transaction can be terminated by either the user or the database system. This will eventually terminate all transactions that have been opened during graph projection, algorithm execution, or results writing. It is not immediately visible and can take a moment for the transactions to recognize that the Cypher transaction has been terminated.

## 10.1. Using GDS and Fabric

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](#).

A typical Neo4j Fabric setup consists of two components: one or more shards that hold the data and one or more Fabric proxies that coordinate the distributed queries. Currently, the way of running the Neo4j Graph Data Science library in a Fabric deployment is to run GDS on the shards. Executing GDS on a Fabric proxy is currently not supported.

### 10.1.1. Running GDS on the Shards

In this mode of using GDS in a Fabric environment, the GDS operations are executed on the shards. The graph projections and algorithms are then 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 use case is to replicate the graph's topology across multiple shards, where some shards act as operational and others as analytical databases.

#### Setup

In this scenario we need to set up the shards to run the Neo4j Graph Data Science library.

Every shard that will run the Graph Data Science library should be configured just as a standalone GDS database would be, for more information see [Installation](#).

The Fabric proxy nodes do not require any special configuration, i.e., the GDS library plugin does not need to be installed. However, the proxy nodes should be configured to handle the amount of data received from the shards.

#### Examples

Let's assume we have a Fabric setup with two shards. One shard functions as the operational database and holds a graph with the schema `(Person)-[KNOWS]->(Person)`. Every `Person` node also stores an identifying property `id` and the persons `name` and possibly other properties.

The other shard, the analytical database, stores a graph with the same data, except that the only property is the unique identifier.

Using Fabric, we can now calculate the PageRank score for each Person and join the results with the name

of that Person.

```
CALL {
  USE FABRIC_DB_NAME.ANALYTICS_DB
  CALL gds.pagerank.stream({nodeProjection: 'Person', relationshipProjection: 'KNOWS'})
  YIELD nodeId, score AS pageRank
  RETURN gds.util.asNode(nodeId).id AS personId, pageRank
}
CALL {
  USE FABRIC_DB_NAME.OPERATIONAL_DB
  WITH personId
  MATCH (p {id: personId})
  RETURN p.name AS name
}
RETURN name, personId, pageRank
```

The query first connects to the analytical database where the PageRank algorithm computes the rank for each node of an anonymous graph. The algorithm results are streamed to the proxy, together with the unique node id. For every row returned by the first subquery, the operational database is then queried for the persons name, again using the unique node id to identify the **Person** node across the shards.

## Limitations

- It is not possible to run algorithms across shards.

## 10.2. GDS Feature Toggles



Feature toggles are not considered part of the public API and can be removed or changed between minor releases of the GDS Library.

### 10.2.1. BitIdMap Feature Toggle Enterprise edition

GDS Enterprise Edition uses a different in-memory graph implementation that is consuming less memory compared to the GDS Community Edition. This in-memory graph implementation performance depends on the underlying graph size and topology. It can be slower for write procedures and graph creation of smaller graphs. To switch to the more memory intensive implementation used in GDS Community Edition you can disable this feature by using the following procedure call.

```
CALL gds.features.useBitIdMap(false)
```

### 10.2.2. Uncompressed Adjacency List Toggle

The in-memory graph for GDS is based on the [Compressed Sparse Row](#) (CSR) layout and uses compressed adjacency lists by default. The compression lowers the memory usage for a graph but requires additional computation time to decompress during algorithm execution. Using an uncompressed adjacency list will result in higher memory consumption in order to provide faster traversals. It can also have negative performance impacts due to the increased resident memory size. Using more memory requires a higher memory bandwidth to read the same adjacency list. Whether compressed or uncompressed is better heavily depends on the topology of the graph and the algorithm. Algorithms that are traversal heavy, such as triangle counting, have a higher chance of benefiting from an uncompressed

adjacency list. Very dense nodes in graphs with a very skewed degree distribution ("power law") often achieve a higher compression ratio. Using the uncompressed adjacency list on those graphs has a higher chance of running into memory bandwidth limitations.

To switch to uncompressed adjacency lists, use the following procedure call.

```
CALL gds.features.useUncompressedAdjacencyList(true)
```

To switch to compressed adjacency lists, use the following procedure call.

```
CALL gds.features.useUncompressedAdjacencyList(false)
```

To reset the setting to the default value, use the following procedure call.

```
CALL gds.features.useUncompressedAdjacencyList.reset() YIELD enabled
```

### 10.2.3. Reordered Adjacency List Toggle

The in-memory graph for GDS writes adjacency lists out of order due to the way the data is read from the underlying store. This feature toggle will add a step during graph creation in which the adjacency lists will be reordered to follow the internal node ids. That reordering results in a CSR representation that is closer to the [textbook layout](#), where the adjacency lists are written in node id order. Reordering can have benefits for some graphs and some algorithms because adjacency lists that will be traversed by the same thread are more likely to be stored close together in memory (caches). The order depends on the GDS internal node ids that are assigned in the in-memory graph and not on the node ids loaded from the underlying Neo4j store.

To enable reordering, use the following procedure call.

```
CALL gds.features.useReorderedAdjacencyList(true)
```

To disable reordering, use the following procedure call.

```
CALL gds.features.useReorderedAdjacencyList(false)
```

To reset the setting to the default value, use the following procedure call.

```
CALL gds.features.useReorderedAdjacencyList.reset() YIELD enabled
```

## Appendix A: Operations reference

The operations in the Graph Data Science library can be divided into the following categories:

- [Graph Catalog](#)
- [Model Catalog](#)

- Graph Algorithms
- Additional Operations

## 10.A.1. Graph Catalog

### Production-quality tier

Table 1011. List of all production-quality graph operations in the GDS library. Functions are written in *italic*.

Description	Operation
Create Graph	<i>gds.graph.create</i>
	<i>gds.graph.create.estimate</i>
	<i>gds.graph.create.cypher</i>
	<i>gds.graph.create.cypher.estimate</i>
Check if a named graph exists	<i>gds.graph.exists</i>
	<i>gds.graph.exists</i>
List graphs	<i>gds.graph.list</i>
Remove node properties from a named graph	<i>gds.graph.removeNodeProperties</i>
Delete relationships from a named graph	<i>gds.graph.deleteRelationships</i>
Remove a named graph from memory	<i>gds.graph.drop</i>
Stream a single node property to the procedure caller	<i>gds.graph.streamNodeProperty</i>
Stream node properties to the procedure caller	<i>gds.graph.streamNodeProperties</i>
Stream a single relationship property to the procedure caller	<i>gds.graph.streamRelationshipProperty</i>
Stream relationship properties to the procedure caller	<i>gds.graph.streamRelationshipProperties</i>
Write node properties to Neo4j	<i>gds.graph.writeNodeProperties</i>
Write relationships to Neo4j	<i>gds.graph.writeRelationship</i>
Graph Export	<i>gds.graph.export</i>

### Beta Tier

Table 1012. List of all beta graph operations in the GDS library. Functions are written in *italic*.

Description	Operation
Create a graph from a named graph	<i>gds.beta.graph.create.subgraph</i>
Generate Random Graph	<i>gds.beta.graph.generate</i>
CSV Export	<i>gds.beta.graph.export.csv</i>
	<i>gds.beta.graph.export.csv.estimate</i>

## 10.A.2. Model Catalog

### Beta Tier

Table 1013. List of all beta model catalog operations in the GDS library. Functions are written in *italic*.

Description	Operation
Check if a model exists	<i>gds.beta.model.exists</i>
Remove a model from memory	<i>gds.beta.model.drop</i>
List models	<i>gds.beta.model.list</i>

### Alpha Tier

Table 1014. List of all alpha model catalog operations in the GDS library. Functions are written in *italic*.

Description	Operation
Store a model	<i>gds.alpha.model.store</i>
Load a stored model	<i>gds.alpha.model.load</i>
Delete a stored model	<i>gds.alpha.model.delete</i>
Publish a model	<i>gds.alpha.model.publish</i>

## 10.A.3. Graph Algorithms

Algorithms exist in one of three tiers of maturity:

- Production-quality
  - Indicates that the algorithm has been tested with regards to stability and scalability.
  - Algorithms in this tier are prefixed with *gds.<algorithm>*.
- Beta
  - Indicates that the algorithm is a candidate for the production-quality tier.
  - Algorithms in this tier are prefixed with *gds.beta.<algorithm>*.
- Alpha
  - Indicates that the algorithm is experimental and might be changed or removed at any time.
  - Algorithms in this tier are prefixed with *gds.alpha.<algorithm>*.

### Production-quality tier

Table 1015. List of all production-quality algorithms in the GDS library. Functions are written in *italic*.

Algorithm name	Operation
Label Propagation	<code>gds.labelPropagation.mutate</code>
	<code>gds.labelPropagation.mutate.estimate</code>
	<code>gds.labelPropagation.write</code>
	<code>gds.labelPropagation.write.estimate</code>
	<code>gds.labelPropagation.stream</code>
	<code>gds.labelPropagation.stream.estimate</code>
	<code>gds.labelPropagation.stats</code>
	<code>gds.labelPropagation.stats.estimate</code>
Louvain	<code>gds.louvain.mutate</code>
	<code>gds.louvain.mutate.estimate</code>
	<code>gds.louvain.write</code>
	<code>gds.louvain.write.estimate</code>
	<code>gds.louvain.stream</code>
	<code>gds.louvain.stream.estimate</code>
	<code>gds.louvain.stats</code>
	<code>gds.louvain.stats.estimate</code>
Node Similarity	<code>gds.nodeSimilarity.mutate</code>
	<code>gds.nodeSimilarity.mutate.estimate</code>
	<code>gds.nodeSimilarity.write</code>
	<code>gds.nodeSimilarity.write.estimate</code>
	<code>gds.nodeSimilarity.stream</code>
	<code>gds.nodeSimilarity.stream.estimate</code>
	<code>gds.nodeSimilarity.stats</code>
	<code>gds.nodeSimilarity.stats.estimate</code>
PageRank	<code>gds.pageRank.mutate</code>
	<code>gds.pageRank.mutate.estimate</code>
	<code>gds.pageRank.write</code>
	<code>gds.pageRank.write.estimate</code>
	<code>gds.pageRank.stream</code>
	<code>gds.pageRank.stream.estimate</code>
	<code>gds.pageRank.stats</code>
	<code>gds.pageRank.stats.estimate</code>

Algorithm name	Operation
Weakly Connected Components	<code>gds.wcc.mutate</code>
	<code>gds.wcc.mutate.estimate</code>
	<code>gds.wcc.write</code>
	<code>gds.wcc.write.estimate</code>
	<code>gds.wcc.stream</code>
	<code>gds.wcc.stream.estimate</code>
	<code>gds.wcc.stats</code>
	<code>gds.wcc.stats.estimate</code>
Triangle Count	<code>gds.triangleCount.stream</code>
	<code>gds.triangleCount.stream.estimate</code>
	<code>gds.triangleCount.stats</code>
	<code>gds.triangleCount.stats.estimate</code>
	<code>gds.triangleCount.write</code>
	<code>gds.triangleCount.write.estimate</code>
	<code>gds.triangleCount.mutate</code>
	<code>gds.triangleCount.mutate.estimate</code>
Local Clustering Coefficient	<code>gds.localClusteringCoefficient.stream</code>
	<code>gds.localClusteringCoefficient.stream.estimate</code>
	<code>gds.localClusteringCoefficient.stats</code>
	<code>gds.localClusteringCoefficient.stats.estimate</code>
	<code>gds.localClusteringCoefficient.write</code>
	<code>gds.localClusteringCoefficient.write.estimate</code>
	<code>gds.localClusteringCoefficient.mutate</code>
	<code>gds.localClusteringCoefficient.mutate.estimate</code>
Betweenness Centrality	<code>gds.betweenness.stream</code>
	<code>gds.betweenness.stream.estimate</code>
	<code>gds.betweenness.stats</code>
	<code>gds.betweenness.stats.estimate</code>
	<code>gds.betweenness.mutate</code>
	<code>gds.betweenness.mutate.estimate</code>
	<code>gds.betweenness.write</code>
	<code>gds.betweenness.write.estimate</code>



Algorithm name	Operation
Fast Random Projection	<code>gds.fastRP.mutate</code>
	<code>gds.fastRP.mutate.estimate</code>
	<code>gds.fastRP.stats</code>
	<code>gds.fastRP.stats.estimate</code>
	<code>gds.fastRP.stream</code>
	<code>gds.fastRP.stream.estimate</code>
	<code>gds.fastRP.write</code>
	<code>gds.fastRP.write.estimate</code>
Degree Centrality	<code>gds.degree.mutate</code>
	<code>gds.degree.mutate.estimate</code>
	<code>gds.degree.stats</code>
	<code>gds.degree.stats.estimate</code>
	<code>gds.degree.stream</code>
	<code>gds.degree.stream.estimate</code>
	<code>gds.degree.write</code>
	<code>gds.degree.write.estimate</code>
ArticleRank	<code>gds.articleRank.mutate</code>
	<code>gds.articleRank.mutate.estimate</code>
	<code>gds.articleRank.write</code>
	<code>gds.articleRank.write.estimate</code>
	<code>gds.articleRank.stream</code>
	<code>gds.articleRank.stream.estimate</code>
	<code>gds.articleRank.stats</code>
	<code>gds.articleRank.stats.estimate</code>
Eigenvector	<code>gds.eigenvector.mutate</code>
	<code>gds.eigenvector.mutate.estimate</code>
	<code>gds.eigenvector.write</code>
	<code>gds.eigenvector.write.estimate</code>
	<code>gds.eigenvector.stream</code>
	<code>gds.eigenvector.stream.estimate</code>
	<code>gds.eigenvector.stats</code>
	<code>gds.eigenvector.stats.estimate</code>

Algorithm name	Operation
Shortest Path Dijkstra	<i>gds.shortestPath.dijkstra.stream</i>
	<i>gds.shortestPath.dijkstra.stream.estimate</i>
	<i>gds.shortestPath.dijkstra.write</i>
	<i>gds.shortestPath.dijkstra.write.estimate</i>
	<i>gds.shortestPath.dijkstra.mutate</i>
	<i>gds.shortestPath.dijkstra.mutate.estimate</i>
All Shortest Paths Dijkstra	<i>gds.allShortestPaths.dijkstra.stream</i>
	<i>gds.allShortestPaths.dijkstra.stream.estimate</i>
	<i>gds.allShortestPaths.dijkstra.write</i>
	<i>gds.allShortestPaths.dijkstra.write.estimate</i>
	<i>gds.allShortestPaths.dijkstra.mutate</i>
	<i>gds.allShortestPaths.dijkstra.mutate.estimate</i>
Shortest Paths Yens	<i>gds.shortestPath.yens.stream</i>
	<i>gds.shortestPath.yens.stream.estimate</i>
	<i>gds.shortestPath.yens.write</i>
	<i>gds.shortestPath.yens.write.estimate</i>
	<i>gds.shortestPath.yens.mutate</i>
	<i>gds.shortestPath.yens.mutate.estimate</i>
Shortest Path AStar	<i>gds.shortestPath.astar.stream</i>
	<i>gds.shortestPath.astar.stream.estimate</i>
	<i>gds.shortestPath.astar.write</i>
	<i>gds.shortestPath.astar.write.estimate</i>
	<i>gds.shortestPath.astar.mutate</i>
	<i>gds.shortestPath.astar.mutate.estimate</i>

## Beta tier

Table 1016. List of all beta algorithms in the GDS library. Functions are written in *italic*.

Algorithm name	Operation
GraphSAGE	<i>gds.beta.graphSage.stream</i>
	<i>gds.beta.graphSage.stream.estimate</i>
	<i>gds.beta.graphSage.mutate</i>
	<i>gds.beta.graphSage.mutate.estimate</i>
	<i>gds.beta.graphSage.write</i>
	<i>gds.beta.graphSage.write.estimate</i>
	<i>gds.beta.graphSage.train</i>
	<i>gds.beta.graphSage.train.estimate</i>

Algorithm name	Operation
K1Coloring	<i>gds.beta.k1coloring.mutate</i>
	<i>gds.beta.k1coloring.mutate.estimate</i>
	<i>gds.beta.k1coloring.stats</i>
	<i>gds.beta.k1coloring.stats.estimate</i>
	<i>gds.beta.k1coloring.stream</i>
	<i>gds.beta.k1coloring.stream.estimate</i>
	<i>gds.beta.k1coloring.write</i>
	<i>gds.beta.k1coloring.write.estimate</i>
K-Nearest Neighbors	<i>gds.beta.knn.mutate</i>
	<i>gds.beta.knn.mutate.estimate</i>
	<i>gds.beta.knn.stats</i>
	<i>gds.beta.knn.stats.estimate</i>
	<i>gds.beta.knn.stream</i>
	<i>gds.beta.knn.stream.estimate</i>
	<i>gds.beta.knn.write</i>
	<i>gds.beta.knn.write.estimate</i>
Modularity Optimization	<i>gds.beta.modularityOptimization.mutate</i>
	<i>gds.beta.modularityOptimization.mutate.estimate</i>
	<i>gds.beta.modularityOptimization.stream</i>
	<i>gds.beta.modularityOptimization.stream.estimate</i>
	<i>gds.beta.modularityOptimization.write</i>
	<i>gds.beta.modularityOptimization.write.estimate</i>
Node2Vec	<i>gds.beta.node2vec.mutate</i>
	<i>gds.beta.node2vec.mutate.estimate</i>
	<i>gds.beta.node2vec.stream</i>
	<i>gds.beta.node2vec.stream.estimate</i>
	<i>gds.beta.node2vec.write</i>
	<i>gds.beta.node2vec.write.estimate</i>
Random Walk	<i>gds.beta.randomWalk.stream</i>
	<i>gds.beta.randomWalk.stream.estimate</i>

## Alpha tier

Table 1017. List of all alpha algorithms in the GDS library. Functions are written in *italic*.

Algorithm name	Operation
All Shortest Paths	<i>gds.alpha.allShortestPaths.stream</i>

Algorithm name	Operation
Approximate Maximum k-cut	<code>gds.alpha.maxkcut.mutate</code>
	<code>gds.alpha.maxkcut.mutate.estimate</code>
	<code>gds.alpha.maxkcut.stream</code>
	<code>gds.alpha.maxkcut.stream.estimate</code>
Breadth First Search	<code>gds.alpha.bfs.stream</code>
Closeness Centrality	<code>gds.alpha.closeness.stream</code>
	<code>gds.alpha.closeness.write</code>
	<code>gds.alpha.closeness.harmonic.stream</code>
	<code>gds.alpha.closeness.harmonic.write</code>
Collapse Path	<code>gds.alpha.collapsePath.mutate</code>
Depth First Search	<code>gds.alpha.dfs.stream</code>
HITS	<code>gds.alpha.hits.mutate</code>
	<code>gds.alpha.hits.mutate.estimate</code>
	<code>gds.alpha.hits.stats</code>
	<code>gds.alpha.hits.stats.estimate</code>
	<code>gds.alpha.hits.stream</code>
	<code>gds.alpha.hits.stream.estimate</code>
	<code>gds.alpha.hits.write</code>
	<code>gds.alpha.hits.write.estimate</code>
Strongly Connected Components	<code>gds.alpha.scc.stream</code>
	<code>gds.alpha.scc.write</code>
Single Source Shortest Path	<code>gds.alpha.shortestPath.deltaStepping.write</code>
	<code>gds.alpha.shortestPath.deltaStepping.stream</code>
Scale Properties	<code>gds.alpha.scaleProperties.mutate</code>
	<code>gds.alpha.scaleProperties.stream</code>
Cosine Similarity	<code>gds.alpha.similarity.cosine.stats</code>
	<code>gds.alpha.similarity.cosine.stream</code>
	<code>gds.alpha.similarity.cosine.write</code>
	<code>gds.alpha.similarity.cosine</code>
Euclidean Similarity	<code>gds.alpha.similarity.euclidean.stats</code>
	<code>gds.alpha.similarity.euclidean.stream</code>
	<code>gds.alpha.similarity.euclidean.write</code>
	<code>gds.alpha.similarity.euclidean</code>
	<code>gds.alpha.similarity.euclideanDistance</code>
Jaccard Similarity	<code>gds.alpha.similarity.jaccard</code>

Algorithm name	Operation
Overlap Similarity	<code>gds.alpha.similarity.overlap.stats</code>
	<code>gds.alpha.similarity.overlap.stream</code>
	<code>gds.alpha.similarity.overlap.write</code>
	<code>gds.alpha.similarity.overlap</code>
Pearson Similarity	<code>gds.alpha.similarity.pearson.stats</code>
	<code>gds.alpha.similarity.pearson.stream</code>
	<code>gds.alpha.similarity.pearson.write</code>
	<code>gds.alpha.similarity.pearson</code>
Speaker-Listener Label Propagation	<code>gds.alpha.sllpa.mutate</code>
	<code>gds.alpha.sllpa.mutate.estimate</code>
	<code>gds.alpha.sllpa.stats</code>
	<code>gds.alpha.sllpa.stats.estimate</code>
	<code>gds.alpha.sllpa.stream</code>
	<code>gds.alpha.sllpa.stream.estimate</code>
	<code>gds.alpha.sllpa.write</code>
	<code>gds.alpha.sllpa.write.estimate</code>
Spanning Tree	<code>gds.alpha.spanningTree.write</code>
	<code>gds.alpha.spanningTree.kmax.write</code>
	<code>gds.alpha.spanningTree.kmin.write</code>
	<code>gds.alpha.spanningTree.maximum.write</code>
	<code>gds.alpha.spanningTree.minimum.write</code>
Approximate Nearest Neighbours	<code>gds.alpha.ml.ann.stream</code>
	<code>gds.alpha.ml.ann.write</code>
Link Prediction	<code>gds.alpha.ml.linkPrediction.predict.mutate</code>
	<code>gds.alpha.ml.linkPrediction.predict.mutate.estimate</code>
	<code>gds.alpha.ml.linkPrediction.predict.stream</code>
	<code>gds.alpha.ml.linkPrediction.predict.stream.estimate</code>
	<code>gds.alpha.ml.linkPrediction.predict.write</code>
	<code>gds.alpha.ml.linkPrediction.predict.write.estimate</code>
	<code>gds.alpha.ml.linkPrediction.train</code>
	<code>gds.alpha.ml.linkPrediction.train.estimate</code>

Algorithm name	Operation
Link Prediction Pipeline	<code>gds.alpha.ml.pipeline.linkPrediction.create</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.addNodeProperty</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.addFeature</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.configureParams</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.configureSplit</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.train</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.predict.mutate</code>
	<code>gds.alpha.ml.pipeline.linkPrediction.predict.stream</code>
Node Classification Pipeline	<code>gds.alpha.ml.pipeline.nodeClassification.create</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.addNodeProperty</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.selectFeatures</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.configureParams</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.configureSplit</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.predict.mutate</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.predict.stream</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.predict.write</code>
	<code>gds.alpha.ml.pipeline.nodeClassification.train</code>
Adamic Adar	<code>gds.alpha.linkprediction.adamicAdar</code>
Common Neighbors	<code>gds.alpha.linkprediction.commonNeighbors</code>
Preferential Attachment	<code>gds.alpha.linkprediction.preferentialAttachment</code>
Preferential Attachment	<code>gds.alpha.linkprediction.resourceAllocation</code>
Same Community	<code>gds.alpha.linkprediction.sameCommunity</code>
Total Neighbors	<code>gds.alpha.linkprediction.totalNeighbors</code>
Node Classification	<code>gds.alpha.ml.nodeClassification.predict.mutate</code>
	<code>gds.alpha.ml.nodeClassification.predict.mutate.estimate</code>
	<code>gds.alpha.ml.nodeClassification.predict.stream</code>
	<code>gds.alpha.ml.nodeClassification.predict.stream.estimate</code>
	<code>gds.alpha.ml.nodeClassification.predict.write</code>
	<code>gds.alpha.ml.nodeClassification.predict.write.estimate</code>
	<code>gds.alpha.ml.nodeClassification.train</code>
	<code>gds.alpha.ml.nodeClassification.train.estimate</code>
Split Relationships	<code>gds.alpha.ml.splitRelationships.mutate</code>

Algorithm name	Operation
Triangle Listing	<code>gds.alpha.triangles</code>
Influence Maximization - Greedy	<code>gds.alpha.influenceMaximization.greedy.stream</code>
Influence Maximization - CELF	<code>gds.alpha.influenceMaximization.celf.stream</code>
Conductance	<code>gds.alpha.conductance.stream</code>

## 10.A.4. Additional Operations

Table 1018. List of all additional operations. Functions are written in *italic*.

Description	Operation
List all operations in GDS	<code>gds.list</code>
List logged progress	<code>gds.beta.listProgress</code>
The version of the installed GDS	<code>gds.version</code>
Node id functions	<code>gds.util.asNode</code>
	<code>gds.util.asNodes</code>
Numeric Functions	<code>gds.util.NaN</code>
	<code>gds.util.infinity</code>
	<code>gds.util.isFinite</code>
	<code>gds.util.isInfinite</code>
Accessing a node property in a named graph	<code>gds.util.nodeProperty</code>
Builds a vector of maps containing items and weights	<code>gds.alpha.similarity.asVector</code>
One Hot Encoding	<code>gds.alpha.ml.oneHotEncoding</code>
Status of the system	<code>gds.debug.sysInfo</code>
Create an impermanent database backed by a named in-memory graph	<code>gds.alpha.create.cypherdb</code>
Get an overview of the system's workload and available resources	<code>gds.alpha.systemMonitor</code>

## Appendix B: Migration from Graph Algorithms v3.5

### 10.B.1. Who should read this guide

This documentation is intended for users who are familiar with the Graph Algorithms library. We assume that most of the mentioned operations and concepts can be understood with little explanation. Thus we are intentionally brief in the examples and comparisons. Please see the dedicated chapters in this manual for details on all the features in the Graph Data Science library.

## 10.B.2. Syntax Changes

In this section we will focus on side-by-side examples of operations using the syntax of the Graph Algorithms library and Graph Data Science library, respectively.

This section is divided into the following sub-sections:

- [Common Changes](#)
- [Memory estimation](#)
- [Graph creation - Named Graph](#)
- [Graph creation - Cypher Queries](#)
- [Graph listing](#)
- [Graph info](#)
- [Graph removal](#)
- [Production-quality algorithms](#)

## 10.B.3. Common changes

This section describes changes between Graph Algorithms library and Graph Data Science library that are common to all procedures.

Table 1019. Namespace

Graph Algorithms v3.5	Graph Data Science v1.0
algo.*	gds.*

Table 1020. Changes in Parameters

Graph Algorithms v3.5	Graph Data Science v1.0 Named Graph	Graph Data Science v1.0 Anonymous Graph
-	<code>graphName</code>	<code>graphConfiguration</code>
node label <sup>[9]</sup>	-	-
relationship type <sup>[9]</sup>	-	-
<code>direction</code>	-	-
<code>config</code>	<code>configuration</code>	-

Table 1021. Changes in configuration parameter map

Graph Algorithms v3.5	Graph Data Science v1.0
<code>write: true</code>	Replaced by dedicated <code>write</code> mode
<code>graph: 'cypher' 'huge'</code>	Removed. Always using <code>huge</code> graph <sup>[10]</sup>
<code>direction</code>	Replaced by <code>projection</code> parameter of <code>relationshipProjection</code>



Graph Algorithms v3.5	Graph Data Science v1.0
<code>direction: 'OUTGOING'</code>	<code>orientation: 'NATURAL'</code>
<code>direction: 'INCOMING'</code>	<code>orientation: 'REVERSE'</code>
<code>direction: 'BOTH'</code>	Removed <sup>[11]</sup>
<code>undirected: true</code>	Replaced by <code>orientation: 'UNDIRECTED'</code> parameter of <code>relationshipProjection</code>
<code>duplicateRelationships</code>	Replaced by <code>aggregation</code> parameter of <code>relationshipProjection</code>
<code>duplicateRelationships: 'SKIP'</code>	<code>aggregation: 'SINGLE'</code>
<code>iterations</code>	<code>maxIterations</code>

## 10.B.4. Memory estimation

Table 1022. Changes in the YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
<code>requiredMemory</code>	<code>requiredMemory</code>
<code>bytesMin</code>	<code>bytesMin</code>
<code>bytesMax</code>	<code>bytesMax</code>
<code>mapView</code>	<code>mapView</code>
-	<code>treeView</code>
-	<code>nodeCount</code>
-	<code>relationshipCount</code>

The most significant change in memory estimation is that in GDS to estimate an operation you suffix it with `.estimate` while in GA the operation had to be passed as parameter to `algo.memrec`.

Table 1023. Estimating the memory requirements of loading a named graph:

Graph Algorithms v3.5	Graph Data Science v1.0
Native Projections:	
<pre>CALL algo.memrec(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   'graph.load' )</pre>	<pre>CALL gds.graph.create.estimate(   'MyLabel',   'MY_RELATIONSHIP_TYPE' )</pre>
Cypher Projections:	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.memrec( 'MATCH (n:MyLabel) RETURN id(n) AS id', 'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t) RETURN id(s) AS source, id(t) AS target', 'graph.load', {   graph: 'cypher' } )</pre>	<pre>CALL gds.graph.create.cypher.estimate( 'MATCH (n:MyLabel) RETURN id(n) AS id', 'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t) RETURN id(s) AS source, id(t) AS target' )</pre>

## 10.B.5. Graph creation - Named Graph

Table 1024. Changes in the YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
name	graphName
graph	-
direction	-
undirected	-
sorted	-
nodes	nodesCount
loadMillis	createMillis
alreadyLoaded	-
nodeProperties	-
relationshipProperties	relationshipCount
relationshipWeight	-
loadNodes	-
loadRelationships	-
-	nodeProjection
-	relationshipProjection

Table 1025. Loading a named graph in the default way:

Graph Algorithms v3.5	Graph Data Science v1.0
Minimal Native Projection:	
<pre>CALL algo.graph.load( 'myGraph', 'MyLabel', 'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.graph.create( 'myGraph', 'MyLabel', 'MY_RELATIONSHIP_TYPE' )</pre>

Graph Algorithms v3.5	Graph Data Science v1.0
Native Projection with additional properties:	
<pre>CALL algo.graph.load(   'myGraph',   'MyLabel',   'MY_RELATIONSHIP_TYPE',   {     concurrency: 4,     graph: 'huge',     direction: 'INCOMING'   } )</pre>	<pre>CALL gds.graph.create(   'myGraph',   'MyLabel',   {     MY_RELATIONSHIP_TYPE: {       orientation: 'REVERSE'     }   },   {     readConcurrency: 4   } )</pre>
Native Projection with <code>direction: 'BOTH'</code> :	
<pre>CALL algo.graph.load(   'myGraph',   'MyLabel',   'MY_RELATIONSHIP_TYPE',   {     graph: 'huge',     direction: 'BOTH'   } )</pre>	<pre>CALL gds.graph.create(   'myGraph',   'MyLabel',   {     MY_RELATIONSHIP_TYPE_NATURAL: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'NATURAL'     },     MY_RELATIONSHIP_TYPE_REVERSE: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'REVERSE'     }   } )</pre>
Undirected Native Projection:	
<pre>CALL algo.graph.load(   'myGraph',   'MyLabel',   'MY_RELATIONSHIP_TYPE',   {     graph: 'huge',     undirected: true   } )</pre>	<pre>CALL gds.graph.create(   'myGraph',   'MyLabel',   {     MY_RELATIONSHIP_TYPE: {       orientation: 'UNDIRECTED'     }   } )</pre>

### 10.B.6. Graph creation - Cypher Queries

Table 1026. Loading a named graph using Cypher queries:

Graph Algorithms v3.5	Graph Data Science v1.0
Basic Cypher queries, defining source and target:	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.graph.load(   'myGraph',   'MATCH (n:MyLabel)   RETURN id(n) AS id',   'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t)   RETURN id(s) AS source, id(t) AS target',   {     graph: 'cypher'   } )</pre>	<pre>CALL gds.graph.create.cypher(   'myGraph',   'MATCH (n:MyLabel)   RETURN id(n) AS id',   'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t)   RETURN id(s) AS source, id(t) AS target' )</pre>

With concurrency property and Cypher query with relationship property:

<pre>CALL algo.graph.load(   'myGraph',   'MATCH (n:MyLabel)   RETURN id(n) AS id',   'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t)   RETURN     id(s) AS source,     id(t) AS target,     r.myProperty AS weight',   {     concurrency: 4,     graph: 'cypher'   } )</pre>	<pre>CALL gds.graph.create.cypher(   'myGraph',   'MATCH (n:MyLabel)   RETURN id(n) AS id',   'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t)   RETURN     id(s) AS source,     id(t) AS target,     r.myProperty AS weight',   {     readConcurrency: 4   } )</pre>
---	--

Parallel loading:

<pre>CALL algo.graph.load(   'myGraph',   'MATCH (n:MyLabel)   WITH * SKIP \$skip LIMIT \$limit   RETURN id(n) AS id',   'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]-&gt;(t)   WITH * SKIP \$skip LIMIT \$limit   RETURN     id(s) AS source,     id(t) AS target,     r.myProperty AS weight',   {     concurrency: 4,     graph: 'cypher'   } )</pre>	-
---	---

## 10.B.7. Graph listing

Table 1027. Changes in the YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
name	graphName
nodes	nodeCount
relationships	relationshipCount

Graph Algorithms v3.5	Graph Data Science v1.0
type	-
direction	-
-	nodeProjection <sup>[12]</sup>
-	relationshipProjection <sup>[12]</sup>
-	nodeQuery <sup>[13]</sup>
-	relationshipQuery <sup>[13]</sup>
-	degreeDistribution <sup>[14]</sup>

Table 1028. Listing named graphs:

Graph Algorithms v3.5	Graph Data Science v1.0
<code>CALL algo.graph.list()</code>	<code>CALL gds.graph.list()</code>

## 10.B.8. Graph info

Table 1029. Changes in the YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
name	graphName
nodes	nodeCount
relationships	relationshipCount
exists	-
removed	-
type	-
direction	-
-	nodeProjection <sup>[15]</sup>
-	relationshipProjection <sup>[15]</sup>
-	nodeQuery <sup>[16]</sup>
-	relationshipQuery <sup>[16]</sup>
-	degreeDistribution <sup>[17]</sup>
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[18]</sup>	-

Table 1030. Viewing information about a specific named graph:

Graph Algorithms v3.5	Graph Data Science v1.0
View information for a Named graph:	

Graph Algorithms v3.5	Graph Data Science v1.0
<code>CALL algo.graph.info('myGraph')</code>	<code>CALL gds.graph.list('myGraph')</code>
Check graph existence:	
<code>CALL algo.graph.info('myGraph') YIELD exists</code>	<code>CALL gds.graph.exists('myGraph') YIELD exists</code>
View graph statistics:	
<code>CALL algo.graph.info('myGraph', true) YIELD min, max, mean, p50</code>	<code>CALL gds.graph.list('myGraph') YIELD degreeDistribution AS dd RETURN dd.min, dd.max, dd.mean, dd.p50</code>

## 10.B.9. Removing named graphs

Table 1031. Changes in the YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
<code>name</code>	<code>graphName</code>
<code>nodes</code>	<code>nodeCount</code>
<code>relationships</code>	<code>relationshipCount</code>
<code>exists</code>	-
<code>removed</code>	-
<code>type</code>	-
<code>direction</code>	-
-	<code>nodeProjection</code> <sup>[19]</sup>
-	<code>relationshipProjection</code> <sup>[19]</sup>
-	<code>nodeQuery</code> <sup>[20]</sup>
-	<code>relationshipQuery</code> <sup>[20]</sup>
-	<code>degreeDistribution</code>

Table 1032. Removing a named graph:

Graph Algorithms v3.5	Graph Data Science v1.0
<code>CALL algo.graph.remove('myGraph')</code>	<code>CALL gds.graph.drop('myGraph')</code>

## 10.B.10. Production-ready algorithms

This section covers all algorithms that have been migrated to the production-ready tier of the Neo4j Graph Data Science library. Syntax changes in configuration, return columns, and execution modes are illustrated with side-by-side examples of queries.

- [Label Propagation](#)
- [Louvain](#)
- [Node Similarity](#)
- [PageRank](#)
- [Weakly Connected Components](#)
- [Triangle Count / Clustering Coefficient](#)
- [Betweenness Centrality \(exact and sampled\)](#)

### Label Propagation

Table 1033. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
<code>direction</code>	-
<code>iterations</code>	<code>maxIterations</code>
<code>concurrency</code>	<code>concurrency</code>
<code>readConcurrency</code>	<code>readConcurrency</code> <sup>[21]</sup>
<code>writeConcurrency</code>	<code>writeConcurrency</code> <sup>[22]</sup>
<code>weightProperty</code> <sup>[23]</sup>	-
-	<code>nodeWeightProperty</code>
-	<code>relationshipWeightProperty</code>
<code>seedProperty</code>	<code>seedProperty</code>
<code>partitionProperty</code>	-
<code>writeProperty</code>	<code>writeProperty</code> <sup>[22]</sup>
<code>write</code>	-
<code>graph</code>	-

Table 1034. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
<code>loadMillis</code>	<code>createMillis</code>
<code>computeMillis</code>	<code>computeMillis</code>
<code>writeMillis</code>	<code>writeMillis</code>
<code>postProcessingMillis</code>	<code>postProcessingMillis</code>

Graph Algorithms v3.5	Graph Data Science v1.0
nodes	nodePropertiesWritten
communityCount	communityCount
didConverge	didConverge
-	ranIterations
write	-
-	communityDistribution
-	configuration <sup>[24]</sup>
writeProperty <sup>[25]</sup>	-
weightProperty <sup>[26]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[27]</sup>	-

Table 1035. Label Propagation Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Streaming over a named graph:	
<pre>CALL algo.labelPropagation.stream(<b>null</b>, <b>null</b>, {graph: 'myGraph'}) <b>YIELD</b> nodeId, label</pre>	<pre>CALL gds.labelPropagation.stream('myGraph') <b>YIELD</b> nodeId, communityId</pre>
Streaming over a named graph using configuration for iterations and relationship weight property:	
<pre>CALL algo.labelPropagation.stream( <b>null</b>, <b>null</b>, { graph: 'myGraph', iterations: <b>15</b>, weightProperty: 'myWeightProperty' } )</pre>	<pre>CALL gds.labelPropagation.stream( 'myGraph', { maxIterations: <b>15</b>, relationshipWeightProperty: 'myWeightProperty' } )</pre>
Streaming over anonymous graph:	
<pre>CALL algo.labelPropagation.stream( 'MyLabel', 'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.labelPropagation.stream({ nodeProjection: 'MyLabel', relationshipProjection: 'MY_RELATIONSHIP_TYPE' })</pre>
Streaming over anonymous graph using relationship with <b>REVERSE</b> orientation:	



Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.labelPropagation.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'INCOMING' } )</pre>	<pre>CALL gds.labelPropagation.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       orientation: 'REVERSE'     }   } })</pre>

Streaming over anonymous graph using two way relationships <sup>[29]</sup>:

<pre>CALL algo.labelPropagation.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'BOTH' } )</pre>	<pre>CALL gds.labelPropagation.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE_NATURAL: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'NATURAL'     },     MY_RELATIONSHIP_TYPE_REVERSE: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'REVERSE'     }   } })</pre>
--	--

Table 1036. Label Propagation Write Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic write:	
<pre>CALL algo.labelPropagation(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     write: true   } ) YIELD   writeMillis,   iterations,   p1,   writeProperty</pre>	<pre>CALL gds.labelPropagation.write(   'myGraph',   { writeProperty: 'myWriteProperty' } ) YIELD   writeMillis,   ranIterations,   communityDistribution AS cd,   configuration AS conf RETURN   writeMillis,   ranIterations,   cd.p1 AS p1,   conf.writeProperty AS writeProperty</pre>
Write using weight properties <sup>[31]</sup> :	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.labelPropagation(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     weightProperty: 'myRelationshipWeightProperty',     write: true   } )</pre>	<pre>CALL gds.labelPropagation.write(   'myGraph',   {     writeProperty: 'myWriteProperty',     relationshipWeightProperty: 'myRelationshipWeightProperty',     nodeWeightProperty: 'myNodeWeightProperty'   } )</pre>
Memory estimation of the algorithm:	
<pre>CALL algo.memrec(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   'labelPropagation',   {     writeProperty: 'myWriteProperty',     weightProperty: 'myRelationshipWeightProperty',     write: true   } )</pre>	<pre>CALL gds.labelPropagation.write.estimate(   {     nodeProjection: 'MyLabel',     relationshipProjection: 'MY_RELATIONSHIP_TYPE',     writeProperty: 'myWriteProperty',     relationshipWeightProperty: 'myRelationshipWeightProperty',     nodeWeightProperty: 'myNodeWeightProperty'   } )</pre>

## Louvain

Table 1037. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
levels	maxLevels
concurrency	concurrency
readConcurrency	readConcurrency <sup>[32]</sup>
writeConcurrency	writeConcurrency <sup>[33]</sup>
weightProperty	relationshipWeightProperty
seedProperty	seedProperty
innerIterations	maxIterations
includeIntermediateCommunities	includeIntermediateCommunities
tolerance	tolerance
writeProperty	writeProperty <sup>[33]</sup>
write	-
graph	-

Table 1038. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
nodes	nodePropertiesWritten
communityCount	communityCount
levels	ranLevels
nodeId	nodeId <sup>[34]</sup>
community	communityId <sup>[34]</sup>
communities	intermediateCommunityIds <sup>[34]</sup>
modularity	modularity <sup>[35]</sup>
modularities	modularities <sup>[35]</sup>
write	-
-	communityDistribution
-	configuration <sup>[36]</sup>
includeIntermediateCommunities <sup>[37]</sup>	-
writeProperty <sup>[37]</sup>	-
weightProperty <sup>[38]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[39]</sup>	-

Table 1039. Louvain Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic streaming over named graph:	
<pre>CALL algo.beta.louvain.stream(<b>null</b>, <b>null</b>, {graph: 'myGraph'}) YIELD nodeId, community, communities</pre>	<pre>CALL gds.louvain.stream('myGraph') YIELD nodeId, communityId, intermediateCommunityIds</pre>
Streaming over named graph using additional properties - <code>maxLevels</code> and <code>maxIterations</code> :	
<pre>CALL algo.beta.louvain.stream(   <b>null</b>,   <b>null</b>,   {     graph: 'myGraph',     levels: <b>15</b>,     innerIterations: <b>30</b>   } )</pre>	<pre>CALL gds.louvain.stream(   'myGraph',   {     maxLevels: <b>15</b>,     maxIterations: <b>30</b>   } )</pre>

Graph Algorithms v3.5	Graph Data Science v1.0
Streaming over named graph with weight property:	
<pre>CALL algo.beta.louvain.stream(   null,   null,   {     graph: 'myGraph',     weightProperty: 'myWeightProperty'   } )</pre>	<pre>CALL gds.louvain.stream(   'myGraph',   {     relationshipWeightProperty: 'myWeightProperty'   } )</pre>
Minimalistic streaming over anonymous graph:	
<pre>CALL algo.beta.louvain.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.louvain.stream({   nodeProjection: 'MyLabel',   relationshipProjection: 'MY_RELATIONSHIP_TYPE' })</pre>
Streaming over anonymous graph with <b>REVERSE</b> relationship orientation:	
<pre>CALL algo.beta.louvain.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'INCOMING' } )</pre>	<pre>CALL gds.louvain.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       orientation: 'REVERSE'     }   } })</pre>
Streaming over anonymous graph using two way relationships <sup>[41]</sup> :	
<pre>CALL algo.louvain.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'BOTH' } )</pre>	<pre>CALL gds.louvain.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE_NATURAL: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'NATURAL'     },     MY_RELATIONSHIP_TYPE_REVERSE: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'REVERSE'     }   } })</pre>

Table 1040. Louvain Write Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic write with just <b>writeProperty</b> :	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.beta.louvain(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     write: true   } ) YIELD   nodes,   writeMillis,   levels,   iterations,   p1,   writeProperty</pre>	<pre>CALL gds.louvain.write(   'myGraph',   { writeProperty: 'myWriteProperty' } ) YIELD   nodePropertiesWritten,   writeMillis,   ranLevels,   ranIterations,   communityDistribution AS cd,   configuration AS conf RETURN   nodePropertiesWritten,   writeMillis,   ranLevels,   ranIterations,   cd.p1 AS p1,   conf.writeProperty AS writeProperty</pre>

Running in `write` mode over weighted graph:

<pre>CALL algo.beta.louvain(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     weightProperty: 'myWeightProperty',     write: true   } )</pre>	<pre>CALL gds.louvain.write(   'myGraph',   {     writeProperty: 'myWriteProperty',     relationshipWeightProperty: 'myWeightProperty'   } )</pre>
--	--

Memory estimation of the algorithm:

<pre>CALL algo.memrec(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   'beta.louvain',   {     writeProperty: 'myWriteProperty',     weightProperty: 'myRelationshipWeightProperty',     write: true   } )</pre>	<pre>CALL gds.louvain.write.estimate(   {     nodeProjection: 'MyLabel',     relationshipProjection: 'MY_RELATIONSHIP_TYPE',     writeProperty: 'myWriteProperty',     relationshipWeightProperty: 'myWeightProperty'   } )</pre>
---	---

## Node Similarity

Table 1041. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
<code>direction</code>	-
<code>concurrency</code>	<code>concurrency</code>
<code>readConcurrency</code>	<code>readConcurrency</code> <sup>[42]</sup>

Graph Algorithms v3.5	Graph Data Science v1.0
writeConcurrency	writeConcurrency <sup>[43]</sup>
topK	topK
bottomK	bottomK
topN	topN
bottomN	bottomN
similarityCutoff	similarityCutoff
degreeCutoff	degreeCutoff
writeProperty	writeProperty <sup>[43]</sup>
writeRelationshipType	writeRelationshipType <sup>[43]</sup>
write	-
graph	-

Table 1042. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
node1	node1 <sup>[44]</sup>
node2	node2 <sup>[44]</sup>
similarity	similarity <sup>[44]</sup>
nodesCompared	nodesCompared <sup>[45]</sup>
relationships	relationshipsWritten <sup>[45]</sup>
write	-
-	similarityDistribution
-	configuration <sup>[46]</sup>
writeProperty <sup>[47]</sup>	-
writeRelationshipType <sup>[47]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[48]</sup>	-

Table 1043. Node Similarity Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic streaming over named graph:	

## Graph Algorithms v3.5

```
CALL algo.nodeSimilarity.stream(null, null,
{graph: 'myGraph'})
YIELD node1, node2, similarity
```

## Graph Data Science v1.0

```
CALL gds.nodeSimilarity.stream('myGraph')
YIELD node1, node2, similarity
```

Streaming over named graph using `topK` and `similarityCutoff` configuration properties:

```
CALL algo.nodeSimilarity.stream(
  null,
  null,
  {
    graph: 'myGraph',
    topK: 1,
    similarityCutoff: 0.5
  }
)
```

```
CALL gds.nodeSimilarity.stream(
  'myGraph',
  {
    topK: 1,
    similarityCutoff: 0.5
  }
)
```

Streaming over named graph using `bottomK` configuration property:

```
CALL algo.nodeSimilarity.stream(
  null,
  null,
  {
    graph: 'myGraph',
    bottomK: 15
  }
)
```

```
CALL gds.nodeSimilarity.stream(
  'myGraph',
  {
    bottomK: 15
  }
)
```

Minimalistic streaming over anonymous graph:

```
CALL algo.nodeSimilarity.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

```
CALL gds.nodeSimilarity.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: 'MY_RELATIONSHIP_TYPE'
})
```

Streaming over anonymous graph using `REVERSE` relationship projection:

```
CALL algo.nodeSimilarity.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

```
CALL gds.nodeSimilarity.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

Streaming over anonymous graph using two way relationships <sup>[50]</sup>:

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.nodeSimilarity.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'BOTH' } )</pre>	<pre>CALL gds.nodeSimilarity.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE_NATURAL: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'NATURAL'     },     MY_RELATIONSHIP_TYPE_REVERSE: {       type: 'MY_RELATIONSHIP_TYPE',       orientation: 'REVERSE'     }   } })</pre>

Table 1044. Node Similarity Write Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic <code>write</code> with <code>writeRelationshipType</code> and <code>writeProperty</code> :	
<pre>CALL algo.nodeSimilarity(   null,   null,   {     graph: 'myGraph',     writeRelationshipType: 'MY_WRITE_REL_TYPE',     writeProperty: 'myWriteProperty',     write: true   } ) YIELD   nodesCompared,   relationships,   writeMillis,   iterations,   p1,   writeProperty</pre>	<pre>CALL gds.nodeSimilarity.write(   'myGraph',   {     writeRelationshipType: 'MY_WRITE_REL_TYPE',     writeProperty: 'myWriteProperty'   } ) YIELD   nodesCompared,   relationships,   writeMillis,   ranIterations,   similarityDistribution AS sd,   configuration AS conf RETURN   nodesCompared,   relationships,   writeMillis,   ranIterations,   sd.p1 AS p1,   conf.writeProperty AS writeProperty</pre>
Memory estimation of the algorithm:	
<pre>CALL algo.memrec(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   'nodeSimilarity',   {     writeRelationshipType: 'MY_WRITE_REL_TYPE',     writeProperty: 'myWriteProperty',     write: true   } )</pre>	<pre>CALL gds.nodeSimilarity.write.estimate(   {     nodeProjection: 'MyLabel',     relationshipProjection:   'MY_RELATIONSHIP_TYPE',     writeRelationshipType: 'MY_WRITE_REL_TYPE',     writeProperty: 'myWriteProperty'   } )</pre>



# PageRank

Table 1045. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
iterations	maxIterations
tolerance	tolerance
dampingFactor	dampingFactor
concurrency	concurrency
readConcurrency	readConcurrency <sup>[51]</sup>
writeConcurrency	writeConcurrency <sup>[52]</sup>
writeProperty	writeProperty <sup>[52]</sup>
weightProperty	relationshipWeightProperty
write	-
graph	-

Table 1046. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
node	nodeId <sup>[53]</sup>
score	score <sup>[53]</sup>
nodes	nodePropertiesWritten <sup>[54]</sup>
iterations	ranIterations
write	-
-	configuration <sup>[55]</sup>
writeProperty <sup>[56]</sup>	-
dampingFactor <sup>[56]</sup>	-
tolerance <sup>[56]</sup>	-
weightProperty <sup>[57]</sup>	-

Table 1047. PageRank Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic stream over named graph:	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.pageRank.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, score</pre>	<pre>CALL gds.pageRank.stream('myGraph') YIELD nodeId, score</pre>
Streaming over named graph with iteration limit:	
<pre>CALL algo.pageRank.stream(   null,   null,   {     graph: 'myGraph',     iterations: 20   } )</pre>	<pre>CALL gds.pageRank.stream(   'myGraph',   {     maxIterations: 20   } )</pre>
Minimalistic streaming over anonymous graph:	
<pre>CALL algo.pageRank.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.pageRank.stream({   nodeProjection: 'MyLabel',   relationshipProjection: 'MY_RELATIONSHIP_TYPE' })</pre>
Streaming over anonymous graph with REVERSE relationship orientation:	
<pre>CALL algo.pageRank.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'INCOMING' } )</pre>	<pre>CALL gds.pageRank.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       orientation: 'REVERSE'     }   } })</pre>
Streaming over anonymous graph with relationship weight property, assigning it a default value in case the property doesn't have value:	
<pre>CALL algo.pageRank.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   {     weightProperty: 'myWeightProperty',     defaultValue: 1.5   } )</pre>	<pre>CALL gds.pageRank.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       properties: {         myWeightProperty: {           defaultValue: 1.5         }       }     }   } })</pre>

Table 1048. PageRank Write Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Running <code>write</code> mode on named graph:	
<pre>CALL algo.pageRank(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     write: true   } ) YIELD   nodes,   loadMillis,   iterations,   p1,   writeProperty</pre>	<pre>CALL gds.pageRank.write(   'myGraph',   {     writeProperty: 'myWriteProperty'   } ) YIELD   nodePropertiesWritten,   createMillis,   ranIterations,   configuration AS conf RETURN   nodePropertiesWritten,   writeMillis,   ranIterations,   conf.writeProperty AS writeProperty</pre>

Memory estimation of the algorithm:

<pre>CALL algo.memrec(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   'pageRank',   {     writeProperty: 'myWriteProperty',     write: true   } )</pre>	<pre>CALL gds.pageRank.write.estimate(   {     nodeProjection: 'MyLabel',     relationshipProjection: 'MY_RELATIONSHIP_TYPE',     writeProperty: 'myWriteProperty'   } )</pre>
---	--

## Weakly Connected Components

Table 1049. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
<code>direction</code>	-
<code>concurrency</code>	<code>concurrency</code>
<code>readConcurrency</code>	<code>readConcurrency</code> <sup>[58]</sup>
<code>writeConcurrency</code>	<code>writeConcurrency</code> <sup>[59]</sup>
<code>writeProperty</code>	<code>writeProperty</code> <sup>[59]</sup>
<code>weightProperty</code>	<code>relationshipWeightProperty</code>
<code>defaultValue</code>	<code>defaultValue</code>
<code>seedProperty</code>	<code>seedProperty</code>
<code>threshold</code>	<code>threshold</code>
<code>consecutiveIds</code>	<code>consecutiveIds</code>
<code>write</code>	-

Graph Algorithms v3.5	Graph Data Science v1.0
graph	-

Table 1050. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
nodeId	nodeId <sup>[60]</sup>
setId	componentId <sup>[60]</sup>
nodes	nodePropertiesWritten <sup>[61]</sup>
-	relationshipPropertiesWritten <sup>[61]</sup>
write	-
-	componentDistribution
-	configuration <sup>[62]</sup>
writeProperty <sup>[63]</sup>	-
weightProperty <sup>[64]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[65]</sup>	-

Table 1051. Weakly Connected Components Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic stream over named graph:	
<pre>CALL algo.unionFind.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, setId</pre>	<pre>CALL gds.wcc.stream('myGraph') YIELD nodeId, componentId</pre>
Streaming over weighted named graph:	
<pre>CALL algo.unionFind.stream(   null,   null,   {     graph: 'myGraph',     weightProperty: 'myWeightProperty'   } )</pre>	<pre>CALL gds.wcc.stream(   'myGraph',   {     relationshipWeightProperty: 'myWeightProperty'   } )</pre>
Minimalistic streaming over anonymous graph:	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.unionFind.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.wcc.stream({   nodeProjection: 'MyLabel',   relationshipProjection: 'MY_RELATIONSHIP_TYPE' })</pre>
Streaming over anonymous graph with <b>REVERSE</b> relationship orientation:	
<pre>CALL algo.unionFind.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { direction: 'INCOMING' } )</pre>	<pre>CALL gds.wcc.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       orientation: 'REVERSE'     }   } })</pre>
Streaming over anonymous graph with relationship specifying default value for the weight property:	
<pre>CALL algo.unionFind.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   {     graph: 'myGraph',     weightProperty: 'myWeightProperty',     defaultValue: 2.0   } )</pre>	<pre>CALL gds.wcc.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       properties: {         myWeightProperty: {           defaultValue: 2         }       }     }   } })</pre>

Table 1052. Weakly Connected Components Write Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic <b>write</b> mode:	
<pre>CALL algo.unionFind(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     write: true   } ) YIELD   nodes,   loadMillis,   p1,   writeProperty</pre>	<pre>CALL gds.wcc.write(   'myGraph',   { writeProperty: 'myWriteProperty' } ) YIELD   nodePropertiesWritten,   createMillis,   componentDistribution AS cd,   configuration AS conf RETURN   nodePropertiesWritten,   createMillis,   cd.p1 AS p1,   conf.writeProperty AS writeProperty</pre>
Running <b>write</b> mode over weighted named graph:	

Graph Algorithms v3.5	Graph Data Science v1.0
<pre>CALL algo.unionFind(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     weightProperty: 'myWeightProperty',     write: true   } )</pre>	<pre>CALL gds.wcc.write(   'myGraph',   {     writeProperty: 'myWriteProperty',     relationshipWeightProperty: 'myWeightProperty'   } )</pre>
Memory estimation of the algorithm:	
<pre>CALL algo.memrec(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   'unionFind',   {     writeProperty: 'myWriteProperty',     weightProperty:   'myRelationshipWeightProperty',     write: true   } )</pre>	<pre>CALL gds.wcc.write.estimate(   {     nodeProjection: 'MyLabel',     relationshipProjection:   'MY_RELATIONSHIP_TYPE',     writeProperty: 'myWriteProperty',     relationshipWeightProperty: 'myWeightProperty'   } )</pre>

## Triangle Counting / Clustering Coefficient

The `alpha` procedures from the namespace `algo.triangleCount` are being replaced by a pair of procedure namespaces:

- `gds.triangleCount`
- `gds.localClusteringCoefficient`

Everything relating to clustering coefficients has been extracted into a separate algorithm backing `gds.localClusteringCoefficient` procedures. To compute both triangle count and local clustering coefficient values multiple procedures will be necessary.

The triangle enumeration procedure `algo.triangles.stream()` has been renamed to `gds.alpha.triangles()`.

Table 1053. Common changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.2
<code>direction</code>	-
<code>concurrency</code>	<code>concurrency</code>
<code>readConcurrency</code>	<code>readConcurrency</code> <sup>[66]</sup>
<code>writeConcurrency</code>	<code>writeConcurrency</code> <sup>[67]</sup>
<code>writeProperty</code>	<code>writeProperty</code> <sup>[67]</sup>

Graph Algorithms v3.5	Graph Data Science v1.2
write	-
graph	-

Table 1054. Changes in YIELD fields of `algo.triangleCount`

Graph Algorithms v3.5	Graph Data Science v1.2
nodeId	nodeId <sup>[68]</sup>
triangles	triangleCount <sup>[69]</sup>
triangleCount	globalTriangleCount <sup>[70]</sup>
nodeCount	nodeCount <sup>[70]</sup>
averageClusteringCoefficient <sup>[71]</sup>	-
clusteringCoefficientProperty <sup>[72]</sup>	-
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
write	-
-	configuration <sup>[73]</sup>
writeProperty <sup>[74]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999	-

Table 1055. TriangleCount Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.2
Streaming triangle counts over named graph:	
<pre>CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, triangles</pre>	<pre>CALL gds.triangleCount.stream('myGraph') YIELD nodeId, triangleCount</pre>
Streaming local clustering coefficients over named graph:	
<pre>CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, coefficient</pre>	<pre>CALL gds.localClusteringCoefficient.stream('myGraph') YIELD nodeId, localClusteringCoefficient</pre>
Streaming both triangle counts and local clustering coefficients:	

Graph Algorithms v3.5	Graph Data Science v1.2
<pre>CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, triangles, coefficient</pre>	<pre>CALL gds.triangleCount.mutate('myGraph', {mutateProperty: 'tc'}) YIELD globalTriangleCount CALL gds.localClusteringCoefficient.stream('myGraph', {   triangleCountProperty: 'tc' }) YIELD nodeId, localClusteringCoefficient WITH   nodeId,   localClusteringCoefficient,   gds.util.nodeProperty('myGraph', nodeId, 'tc') AS triangleCount RETURN nodeId, triangleCount, localClusteringCoefficient</pre>
Streaming triangle counts over anonymous graph:	
<pre>CALL algo.triangleCount.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.triangleCount.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       orientation: 'UNDIRECTED'     }   } })</pre>

Table 1056. TriangleCount Write Mode

Graph Algorithms v3.5	Graph Data Science v1.2
Writing triangle counts from named graph:	
<pre>CALL algo.triangleCount(null, null, {   graph: 'myGraph',   write: true,   writeProperty: 'tc' }) YIELD nodeCount, triangleCount</pre>	<pre>CALL gds.triangleCount.write('myGraph', {   writeProperty: 'tc' }) YIELD nodeCount, globalTriangleCount</pre>
Writing local clustering coefficients from named graph:	
<pre>CALL algo.triangleCount(null, null, {   graph: 'myGraph',   write: true,   clusteringCoefficientProperty: 'lcc' }) YIELD nodeCount, averageClusteringCoefficient</pre>	<pre>CALL gds.localClusteringCoefficient.write(   'myGraph', {     writeProperty: 'lcc'   }) YIELD nodeCount, averageClusteringCoefficient</pre>
Writing both triangle counts and local clustering coefficients:	



Graph Algorithms v3.5	Graph Data Science v1.2
<pre>CALL algo.triangleCount(null, null, {   graph: 'myGraph',   write: true,   writeProperty: 'tc',   clusteringCoefficientProperty: 'lcc' }) YIELD nodeCount, triangleCount, averageClusteringCoefficient</pre>	<pre>CALL gds.triangleCount.mutate('myGraph', {   mutateProperty: 'tc' }) YIELD globalTriangleCount CALL gds.localClusteringCoefficient.write ('myGraph', {   triangleCountProperty: 'tc',   writeProperty: 'lcc' }) YIELD nodeCount, averageClusteringCoefficient CALL gds.graph.writeNodeProperties('myGraph', ['tc']) YIELD propertiesWritten RETURN nodeCount, globalTriangleCount, averageClusteringCoefficient</pre>

## Betweenness Centrality

In Graph Algorithms v3.5, Betweenness Centrality was surfaced in two different procedures:

- `algo.betweenness` and `algo.betweenness.stream`
- `algo.betweenness.sampled` and `algo.betweenness.sampled.stream`

These two have been merged into a single procedure `gds.betweenness` with all four execution modes supported. The sampling is controlled via configuration parameter rather than explicit procedures. Setting the sampling size to the node count will produce exact results.

Table 1057. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.3
<code>stats</code>	-
<code>strategy</code>	-
<code>probability</code>	-
-	<code>samplingSize</code>
-	<code>samplingSeed</code>
<code>maxDepth</code>	-
<code>direction</code>	-
<code>concurrency</code>	<code>concurrency</code>
<code>readConcurrency</code>	<code>readConcurrency</code> <sup>[75]</sup>
<code>writeConcurrency</code>	<code>writeConcurrency</code> <sup>[76]</sup>
<code>writeProperty</code>	<code>writeProperty</code> <sup>[76]</sup>
<code>write</code>	-
<code>graph</code>	-

Table 1058. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.3
centrality	score <sup>[77]</sup>
nodes	-
minCentrality	minimumScore
maxCentrality	maximumScore
sumCentrality	scoreSum
loadMillis	createMillis
evalMillis	computeMillis
writeMillis	writeMillis
-	postProcessingMillis
-	configuration <sup>[78]</sup>
-	nodePropertiesWritten <sup>[79]</sup>
write	-
writeProperty <sup>[80]</sup>	-

Table 1059. Betweenness Centrality Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.3
Minimalistic stream over named graph:	
<pre>CALL algo.betweenness.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, centrality</pre>	<pre>CALL gds.betweenness.stream('myGraph') YIELD nodeId, score</pre>
Minimalistic stream over named graph, sampled:	
<pre>CALL algo.betweenness.sampled.stream(null, null, {graph: 'myGraph', probability: 0.5}) YIELD nodeId, centrality</pre>	<pre>CALL gds.betweenness.stream('myGraph', {samplingSize: 1000}) // assume 2000 nodes YIELD nodeId, score</pre>
Minimalistic streaming over anonymous graph:	
<pre>CALL algo.betweenness.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE' )</pre>	<pre>CALL gds.betweenness.stream({   nodeProjection: 'MyLabel',   relationshipProjection: 'MY_RELATIONSHIP_TYPE' })</pre>
Streaming over anonymous graph with <b>UNDIRECTED</b> relationship orientation:	

Graph Algorithms v3.5	Graph Data Science v1.3
<pre>CALL algo.betweenness.stream(   'MyLabel',   'MY_RELATIONSHIP_TYPE',   { undirected: true } )</pre>	<pre>CALL gds.betweenness.stream({   nodeProjection: 'MyLabel',   relationshipProjection: {     MY_RELATIONSHIP_TYPE: {       orientation: 'UNDIRECTED'     }   } })</pre>

Table 1060. Betweenness Centrality Write Mode

Graph Algorithms v3.5	Graph Data Science v1.3
<p>Running <b>write</b> mode on named graph:</p>	
<pre>CALL algo.betweenness(   null,   null,   {     graph: 'myGraph',     writeProperty: 'myWriteProperty',     write: true   } ) YIELD   nodes,   minCentrality,   maxCentrality,   sumCentrality,   loadMillis,   evalMillis,   writeMillis</pre>	<pre>CALL gds.betweenness.write(   'myGraph',   {     writeProperty: 'myWriteProperty'   } ) YIELD   nodePropertiesWritten,   minimumScore,   maximumScore,   scoreSum,   createMillis,   computeMillis,   writeMillis,   configuration</pre>

# 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>.

[8] Moved to `graphConfiguration` as `nodeProjection`

[9] Moved to `graphConfiguration` as `relationshipProjection`

[10] Graph creation with cypher queries has dedicate `gds.graph.create.cypher` procedure. There are parameters `nodeQuery` and `relationshipQuery` for anonymous graphs

[11] This behaviour can be achieved by creating two relationship projections - one with `orientation: 'NATURAL'` and one with `orientation: 'REVERSE'`

[12] Field will be `null` if a Cypher projection was used

[13] Field will be `null` unless a Cypher projection was used

- [14] Graph statistics map, i.e. min, max, percentiles, etc.
- [15] Field will be `null` if a Cypher projection was used
- [16] Field will be `null` unless a Cypher projection was used
- [17] Graph statistics map, i.e. min, max, percentiles, etc.
- [18] Inlined into `degreeDistribution`
- [19] Field will be `null` if a Cypher projection was used
- [20] Field will be `null` unless a Cypher projection was used
- [21] Only when using anonymous graph
- [22] Only for `write` mode
- [23] Can be configured separately by using `nodeWeightProperty` and `relationshipWeightProperty`
- [24] The configuration used to run the algorithm
- [25] Inlined into `configuration`
- [26] Inlined into `configuration` as `nodeWeightProperty` and/or `relationshipWeightProperty`
- [27] Inlined into `communityDistribution`
- [32] Only when using anonymous graph
- [33] Only for `write` mode
- [34] Only for `stream` mode
- [35] Only for `write` mode
- [36] The configuration used to run the algorithm
- [37] Inlined into `configuration`
- [38] Inlined into `configuration` as `relationshipWeightProperty`
- [39] Inlined into `communityDistribution`
- [42] Only when using anonymous graph
- [43] Only for `write` mode
- [44] Only for `stream` mode
- [45] Only for `write` mode
- [46] The configuration used to run the algorithm
- [47] Inlined into `configuration`
- [48] Inlined into `similarityDistribution`
- [51] Only when using anonymous graph
- [52] Only for `write` mode
- [53] Only for `stream` mode
- [54] Only for `write` mode
- [55] The configuration used to run the algorithm
- [56] Inlined into `configuration`
- [57] Inlined into `configuration` as `relationshipWeightProperty`
- [58] Only when using anonymous graph
- [59] Only for `write` mode
- [60] Only for `stream` mode
- [61] Only for `write` mode
- [62] The configuration used to run the algorithm
- [63] Inlined into `configuration`
- [64] Inlined into `configuration` as `relationshipWeightProperty`
- [65] Inlined into `componentDistribution`
- [66] Only when using anonymous graph
- [67] Only for `write` mode
- [68] Only for `stream` mode
- [69] Only for `stream` mode
- [70] Not present in `stream` mode
- [71] Moved to `gds.localClusteringCoefficient`
- [72] Moved as `writeProperty` to `gds.localClusteringCoefficient`
- [73] The configuration used to run the algorithm
- [74] Inlined into `configuration`
- [75] Only when using anonymous graph
- [76] Only for `write` mode

[77] Only for `stream` mode

[78] The configuration used to run the algorithm

[79] Only for `write` mode

[80] Inlined into `configuration`