

# The Neo4j Graph Data Science Library

## Manual v1.2

# Table of Contents

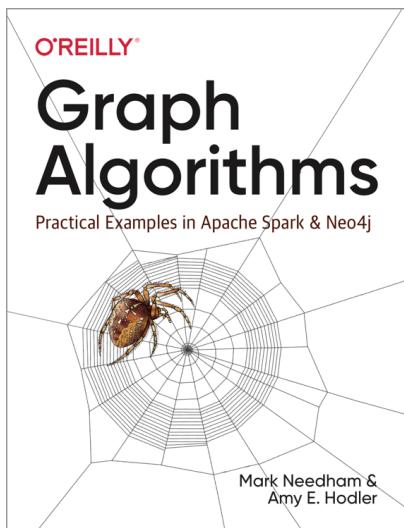
Introduction .....	2
Algorithms .....	2
Graph Catalog .....	2
Editions .....	2
Installation .....	3
Supported Neo4j versions .....	3
Neo4j Desktop .....	3
Neo4j Server .....	4
Neo4j Causal Cluster .....	5
System Requirements .....	6
Common usage .....	8
Memory Estimation .....	9
Creating graphs .....	13
Running algorithms .....	13
Graph management .....	15
Graph Catalog .....	15
Native projection .....	21
Cypher projection .....	35
Anonymous graphs .....	41
Utility functions .....	43
Algorithms .....	48
Syntax overview .....	48
Centrality algorithms .....	50
Community detection algorithms .....	96
Similarity algorithms .....	187
Path finding algorithms .....	267
Link Prediction algorithms .....	314
Auxiliary procedures .....	327
Production deployment .....	332
Transaction Handling .....	332
Appendix A: Procedures and functions reference .....	334
Graph Operations .....	334
Production-quality tier .....	334
Beta tier .....	336
Alpha tier .....	337
Appendix B: Migration from Graph Algorithms v3.5 .....	340
Who should read this guide .....	340
Syntax Changes .....	340

***This is the manual for Neo4j Graph Data Science library version 1.2.***

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.
- [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.
- [Procedures and functions 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.

For further reading resources, we recommend studying the free Graph Algorithms book.



Graph Algorithms: Practical Examples in Apache Spark and Neo4j, by Mark Needham & Amy E. Hodler and published by O'Reilly Media is available now.

Download it for free at [neo4j.com/graph-algorithms-book/](https://neo4j.com/graph-algorithms-book/).

# Introduction

*This chapter provides an introduction to graph algorithms.*

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

## 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.

## 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](#).

## 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.

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

# Installation

*This chapter provides instructions for installation and basic usage of the Neo4j Graph Data Science library.*

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 whitelisted in the Neo4j configuration. There are two main ways of achieving this, which we will detail in this chapter.

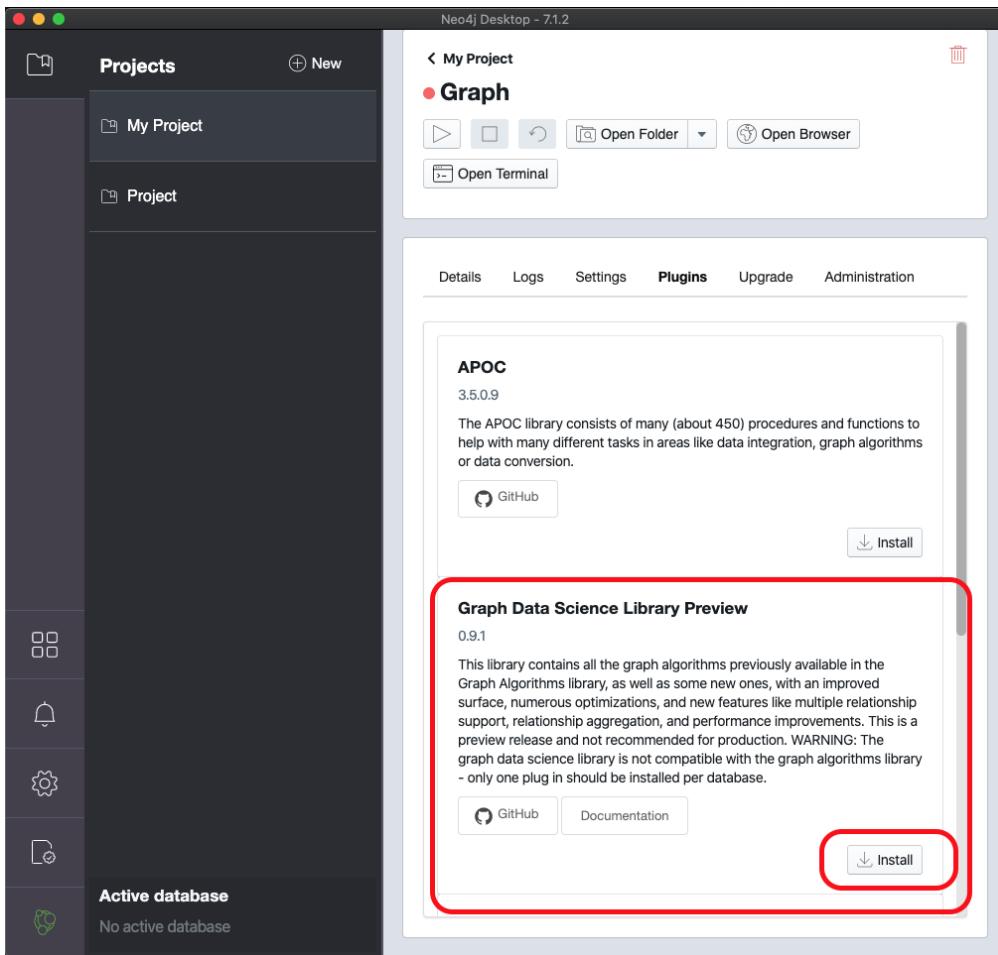
## Supported Neo4j versions

The GDS library supports the following Neo4j versions:

Neo4j Graph Data Science	Neo4j version
	4.0.0
	4.0.1
1.2.x	4.0.2
	4.0.3
	4.0.4
	3.5.9
	3.5.10
	3.5.11
	3.5.12
	3.5.13
1.0.x, 1.1.x	3.5.14
	3.5.15
	3.5.16
	3.5.17
	3.5.18

## 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 whitelist is configured, make sure to also include procedures from the GDS library:

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

## 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-\[version\]-standalone.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 whitelist is enabled in the `$NEO4J_HOME/conf/neo4j.conf` file and add the GDS library if necessary:

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

4. Restart Neo4j

## 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()
```

## 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, which

requires:

- installing GDS on the Read Replica
- managing cluster synchronisation events during GDS algorithm execution
- avoiding use of GDS write-back features
- consuming results from GDS workloads directly via Cypher

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.

## System Requirements

### Main Memory

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

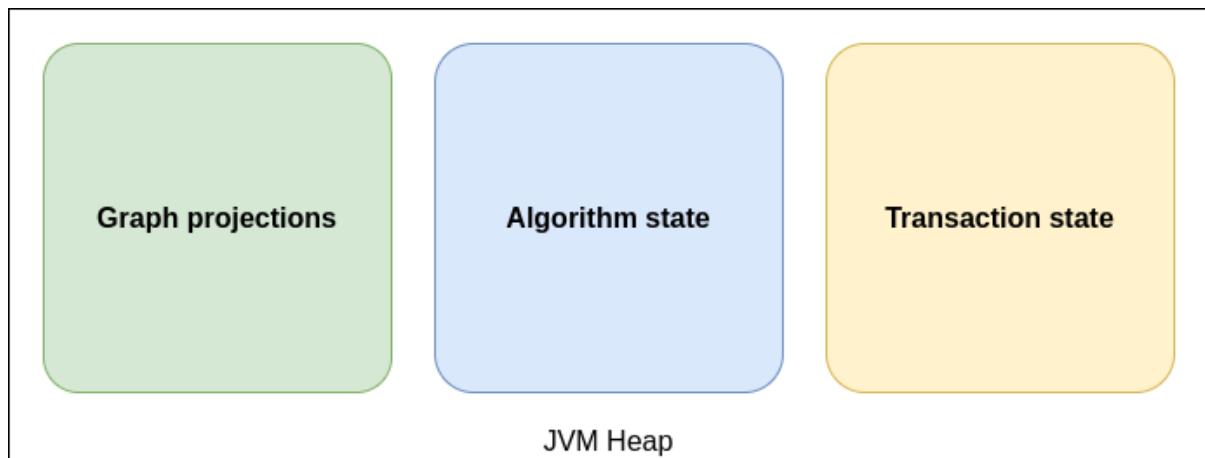


Figure 1. GDS heap memory usage

### 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 using [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.

## 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 [Configuration options](#)

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

- Neo4j Community Edition
  - The maximum concurrency in the library is 4.
- Neo4j Enterprise Edition
  - The maximum concurrency in the library is 4.
- Neo4j Graph Data Science Edition
  - The concurrency in the library is unlimited. To register for a license, please contact Neo4j at <https://neo4j.com/contact-us/?ref=graph-analytics>.

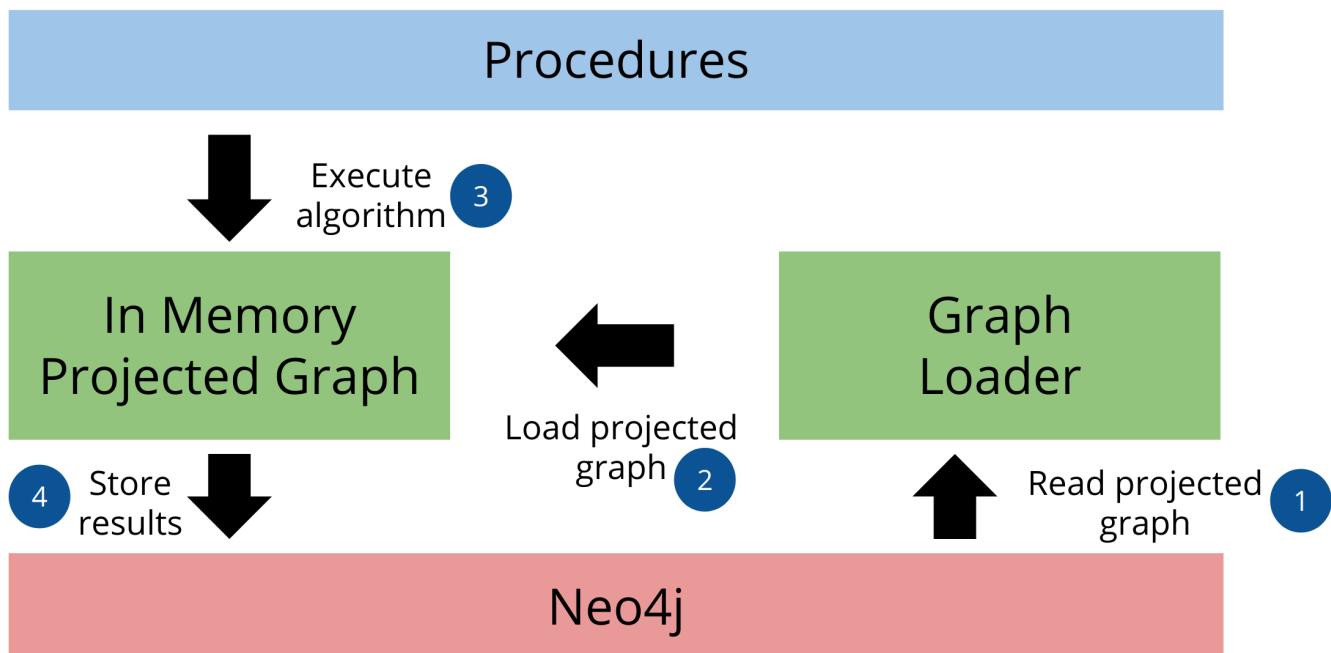
# Common usage

*This chapter explains the common usage patterns and operations that constitute the core of the Neo4j Graph Data Science library.*

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:



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:

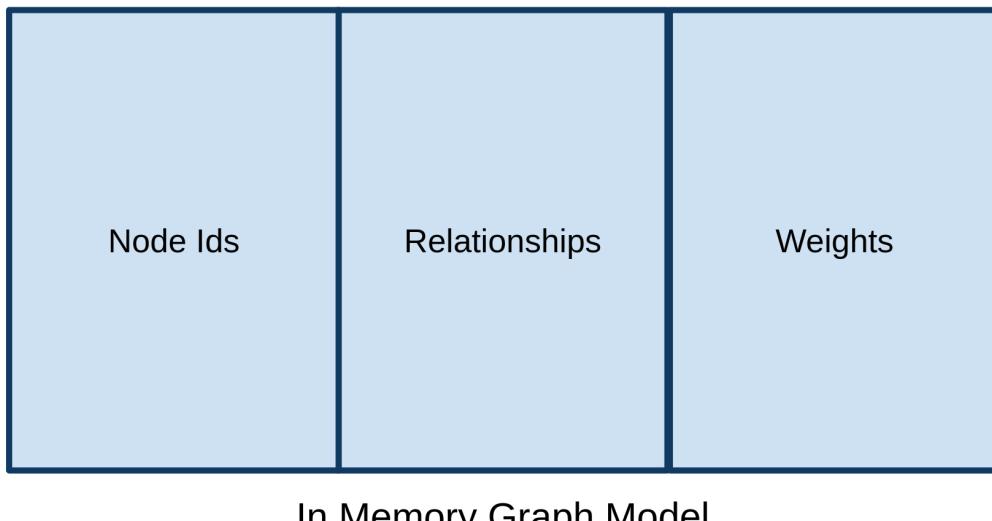
- [Memory Estimation](#)
- [Creating graphs](#)

- [Running algorithms](#)

## Memory Estimation

*This section describes how to estimate memory requirements for the projected graph model used by the Neo4j Graph Data Science library.*

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:



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.

This section includes:

- [Estimating memory requirements for algorithms](#)
- [Estimating memory requirements for graphs](#)
- [Automatic estimation and execution blocking](#)

### Estimating memory requirements for algorithms

In many use cases it will be useful to estimate the required memory of a graph and an algorithm before running it in order to make sure that the workload can run on the available hardware. To make this process easier every algorithm supports the `.estimate` mode, which returns an estimate of the amount of memory required to run graph algorithms.

## Syntax

```
CALL gds.<ALGO>.<MODE>.estimate(graphNameOrConfig: String|Map, configuration: Map)
YIELD requiredMemory, treeView, mapView, bytesMin, bytesMax, heapPercentageMin,
heapPercentageMax, nodeCount, relationshipCount
```

Table 1. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	-	no	The name of the projected graph or the algorithm configuration in case of implicit loading.
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 parameter accepts the same configuration parameters as the estimated algorithm. See the algorithm documentation for more information.

Table 2. Results

Name	Type	Description
requiredMemory	String	An estimation of the required memory in a human readable format.
treeView	String	A more detailed, human readable representation of the required memory, including estimates of the different components.
mapView	String	A more detailed representation of the required memory, including estimates of the different components.
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.
nodeCount	Integer	The estimated number of nodes in the graph
relationshipCount	Integer	The estimated number of relationships in the graph

## 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
"[561 KiB ... 564 KiB]"	574768	577952	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.
relationshipsQuery	String	-	no	The relationship query to estimate for.
configuration	Map	{}	yes	Additional configuration, such as concurrency.

## Automatic estimation and execution blocking

All algorithm procedures in the GDS library, 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.

## 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.

## Running algorithms

**This section describes the common execution modes for algorithms: `stream`, `write`, `mutate` and `stats`.**

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, the main execution modes to consider are `stream`, `write`, `mutate` and `stats`, 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](#).

### 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.

## 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 as a Cypher result row. This is the only execution mode that will make any 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 in another algorithm, a new graph must be created from the Neo4j database with the updated graph.

## Mutate

The `mutate` mode is very similar to the `write` mode but instead of writing results to the Neo4j database, they are made available at the in-memory graph. Note that the `mutateProperty` 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 as a Cypher result row. As for the `write` mode, mutated data can be node properties (such as Page Rank scores), new relationships (such as Node Similarity similarities), or relationship properties.

## Stats

The `stats` mode returns statistical results for the algorithm computation like counts or a percentile distribution. The same results can be acquired from the `write` mode of the procedure, but an algorithm running in `stats` mode makes no modifications to the underlying Neo4j graph.

# Graph management

This chapter explains the graph catalog, the different graph projection variants and utility functions in the Neo4j Graph Data Science library.

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

This chapter is divided into the following sections:

- [Graph Catalog](#)
- [Native projection](#)
- [Cypher projection](#)
- [Anonymous graphs](#)
- [Utility functions](#)

## 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 view over the stored graph, containing only analytical 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 that 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 a <a href="#">Native projection</a> .
<a href="#">gds.graph.create.cypher</a>	Creates a graph in the catalog using a <a href="#">Cypher projection</a> .
<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.

Name	Description
<code>gds.graph.removeNodeProperties</code>	Removes node properties from a named graph.
<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.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.beta.graph.export</code>	Exports a named graph into a new offline Neo4j database.



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.

## Creating graphs in the catalog

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.

There are two variants of projecting a graph from the Neo4j database into main memory:

- [Native projection](#)
  - Provides the best performance by reading from the Neo4j store files. Recommended to be used during both the development and the production phase.
- [Cypher projection](#)
  - The more flexible, expressive approach with lesser focus on performance. Recommended to be primarily used during the development phase.

In this section, we will give brief examples on how to create a graph using either variant. For detailed information about the configuration of each variant, we refer to the dedicated sections.

In the following two examples we show how to create a graph called `my-native-graph` that contains `Person` nodes and `LIKES` relationships.

*Create a graph using a native projection:*

```
CALL gds.graph.create(
    'my-native-graph',
    'Person',
    'LIKES'
)
YIELD graphName, nodeCount, relationshipCount, createMillis;
```

We can also use Cypher to select the nodes and relationships to be projected into the in-memory graph.

*Create a graph using a Cypher projection:*

```
CALL gds.graph.create.cypher(
    'my-cypher-graph',
    'MATCH (n:Person) RETURN id(n) AS id',
    'MATCH (a:Person)-[:LIKES]->(b:Person) RETURN id(a) AS source, id(b) AS target'
)
YIELD graphName, nodeCount, relationshipCount, createMillis;
```

After creating the graphs in the catalog, we can refer to them in algorithms by using their name.

*Run Page Rank on one of our created graphs:*

```
CALL gds.algo.pageRank.stream('my-native-graph') YIELD nodeId, score;
```

## Listing graphs in the catalog

Once we have created graphs in the catalog, we can list information about either all of them or a single graph using its name.

*List information about all graphs in the catalog:*

```
CALL gds.graph.list()
YIELD graphName, nodeProjection, relationshipProjection, nodeQuery, relationshipQuery,
      nodeCount, relationshipCount, schema, degreeDistribution, creationTime,
      modificationTime;
```

*List information about a named graph in the catalog:*

```
CALL gds.graph.list(graphName)
YIELD graphName, nodeProjection, relationshipProjection, nodeQuery, relationshipQuery,
      nodeCount, relationshipCount, schema, degreeDistribution, creationTime,
      modificationTime, sizeInBytes, memoryUsage;
```

The `nodeProjection` and `relationshipProjection` columns are primarily applicable to [Native projection](#). The `nodeQuery` and `relationshipQuery` columns are applicable only to [Cypher projection](#) and are `null` for graphs created with Native projection.

The `degreeDistribution` is more time-consuming to compute than the other return columns. It is however only computed when included in the `YIELD` subclause.

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` or `Float`.

The `creationTime` indicates when the graph was created in memory. The `modificationTime` indicates when the graph was updated by an algorithm running in `mutate` mode. The `sizeInBytes` yields the number of bytes used in the Java Heap to store that graph. The `memoryUsage` is the same information in a human readable format.

*List information about the degree distribution of a specific graph:*

```
CALL gds.graph.list('my-cypher-graph')
YIELD graphName, degreeDistribution;
```

## Check if a graph exists in the catalog

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

*Check if a graph exists in the catalog:*

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

## Removing node properties from a named graph

We can 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.

*Remove multiple node properties from a named graph:*

```
CALL gds.graph.removeNodeProperties('my-graph', ['pageRank', 'communityId'])
```

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

The procedure can be configured to remove just the properties for some specific node projections. In the following example, we ran an algorithm on a sub-graph and subsequently remove the newly created property.

*Remove node properties of a specific node projection:*

```
CALL gds.graph.create('my-graph', ['A', 'B'], '*')
CALL gds.wcc.mutate('my-graph', {nodeLabels: ['A'], mutateProperty: 'componentId'})
CALL gds.graph.removeNodeProperties('my-graph', ['componentId'], ['A'])
```

When a list of projections that are not `*` is specified, as in the example above, a different validation and execution is applied; It is then required that all projections have all of the given properties, and they will be removed from all of the projections.

If any of the given projections is `'*'`, the procedure behaves like in the first example.

## Deleting relationship types from a named graph

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.

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

```
CALL gds.graph.deleteRelationships('my-graph', 'T')
YIELD graphName, relationshipType, deletedRelationships, deletedProperties
```

## Removing graphs from the catalog

Once we have finished using the named graph we can remove it from the catalog to free up memory.

*Remove a graph from the catalog:*

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

## Write node properties to Neo4j

We can write node properties stored in a named in-memory graph back to Neo4j. This is useful if we ran multiple algorithms in `mutate` mode and want to write back some or all of the results. This is similar to what the `write` execution mode does, but allows more fine-grained control over the operations.

The properties to write are typically the `writeProperty` values that were used when running algorithms. Properties that were added to the created graph at creation time will often already be present in the Neo4j database.

*Write multiple node properties to Neo4j:*

```
CALL gds.graph.writeNodeProperties('my-graph', ['componentId', 'pageRank',
'communityId'])
```

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

The procedure can be configured to write just the properties for some specific node projections. In the following example, we ran an algorithm on a sub-graph and subsequently wrote the newly created property to Neo4j.

*Write node properties of a specific node projection to Neo4j:*

```
CALL gds.graph.create('my-graph', ['A', 'B'], '*')
CALL gds.wcc.mutate('my-graph', {nodeLabels: ['A'], mutateProperty: 'componentId'})
CALL gds.graph.writeNodeProperties('my-graph', ['componentId'], ['A'])
```

When a list of projections that are not `*` is specified, as in the example above, a different validation and execution is applied; It is then required that all projections have all of the given properties, and they will be written to Neo4j for all of the projections.

If any of the given projections is `'*'`, the procedure behaves like in the first example.

## Write relationships to Neo4j

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. This can either be an element identifier used in a relationship projection during graph construction or the `writeRelationshipType` used in algorithms that create relationships.

*Write relationships to Neo4j:*

```
CALL gds.graph.writeRelationship('my-graph', 'SIMILAR_TO')
```

By default, no relationship properties will be written. To write relationship properties, these have to be explicitly specified.

*Write relationships and their properties to Neo4j:*

```
CALL gds.graph.writeRelationship('my-graph', 'SIMILAR_TO', 'similarityScore')
```

## Create Neo4j databases from named graphs

This procedure is in the beta tier. For more information on this tier of algorithm, see [here](#).

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.

*Export a named graph to a new database in the Neo4j databases directory:*

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

The procedure yields information about the number of nodes, relationships and properties written. Optional parameters are `writeConcurrency`, `enableDebugLog` and `batchSize`.

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

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

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

## Native projection

*This section explains native projections in the Neo4j Graph Data Science library.*

A native projection allows us to project a graph from Neo4j into an in-memory graph. The projected graph can be specified in terms of node labels, relationship types and properties. Node labels and node properties are projected using [Node projections](#). Relationship types and relationship properties are projected using [Relationship projections](#).

The main benefit of native projections is their performance. In contrast, a [Cypher projection](#) is more flexible from the declaration point of view, but less performant. In most cases it is possible to structure your Neo4j graph model in a way that enables native projections to be used.

This section includes:

- [Syntax](#)
- [Node projections](#)
- [Relationship projections](#)

### Syntax

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

```

CALL gds.graph.create(
    graphName: String,
    nodeProjection: String, List or Map,
    relationshipProjection: String, List or Map,
    configuration: Map
)

```

*Table 7. Parameters*

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

*Table 8. Configuration*

Name	Type	Default	Description
readConcurrency	Integer	4	The number of concurrent threads used for creating the graph.
nodeProperties	String, List or Map	empty map	Node properties to load for all node projections.
relationshipProperties	String, List or Map	empty map	Relationship properties to load for all relationship projections.
validateRelationships	Boolean	false	Whether to throw an error if relationships contain nodes not included in the nodeProjection.

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

## Node projections

A node projection enables mapping Neo4j nodes into the in-memory graph. When specifying a node projection, we can declare one or more node labels that we want to project.

The following map-like syntax shows the general way of defining node projections:

```
{
    <node-label-1>: {
        label: <neo4j-label>,
        properties: <node-property-mappings>
    },
    <node-label-2>: {
        label: <neo4j-label>,
        properties: <node-property-mappings>
    },
    // ...
    <node-label-n>: {
        label: <neo4j-label>,
        properties: <node-property-mappings>
    }
}
```

- **node-label-i** denotes the node label used in the projected graph
  - **neo4j-label** denotes the node label in the Neo4j graph
    - The label must exist in the Neo4j database
    - If not specified, **neo4j-label** defaults to **node-label-i**
  - **node-property-mappings** denotes a set of mappings between Neo4j and in-memory properties

In the following example, we want to project **Person** nodes into the in-memory graph. The resulting graph contains all Neo4j nodes that have the label **Person**.

```
CALL gds.graph.create(
    'my-graph', {
        Person: { label: 'Person' }
    },
    '*'
)
YIELD graphName, nodeCount, relationshipCount;
```

We can use the following shorthand syntax to project a single node label.

```
CALL gds.graph.create('my-graph', 'Person', '*')
YIELD graphName, nodeCount, relationshipCount;
```

It is often useful to create an in-memory graph representing more than one node label. Let's assume, the database contains **Person** and **City** nodes which we want to project.

```

CALL gds.graph.create(
  'my-graph', {
    Person: { label: 'Person' },
    City: { label: 'City' }
  },
  '*'
)
YIELD graphName, nodeCount, relationshipCount;

```

The following shorthand syntax can be used to project multiple labels.

```

CALL gds.graph.create('my-graph', ['Person', 'City'], '*')
YIELD graphName, nodeCount, relationshipCount;

```

Projecting multiple node labels enables algorithms to only use a subset of those.

```

// Uses 'Person' nodes for computing Page Rank scores between persons
CALL gds.pageRank.stream('my-graph', { nodeLabels: ['Person'] }) YIELD nodeId, score;

```

To project all nodes in the Neo4j graph, we can use the special `*` ('star') node projection.

```

CALL gds.graph.create('my-graph', '*', '*')
YIELD graphName, nodeCount, relationshipCount;

```

When a graph uses the special star projection, additional nodes from named node projections are not added additionally as the star projection already contains them. This also applies to filtering. For example, if a graph was projected with labels `'A'`, `'B'` and also with the special star `'*'`, any node filtering that includes the star is equivalent to using only the star. It is however still possible to filter for nodes with label `'A'` or `'B'` in order to retrieve a subgraph containing only those nodes.

*Example usage of node label filters:*

```

CALL gds.graph.create('myGraph', ['Person', 'City', '*'])

// using all nodes projected in 'myGraph' (the star projection)
CALL gds.pageRank.stats('myGraph', {nodeLabels: ['*']})
CALL gds.pageRank.stats('myGraph', {nodeLabels: ['Person', 'City', '*']}) // equivalent
CALL gds.pageRank.stats('myGraph') // equivalent

// will use only nodes projected as 'Person' in 'myGraph'
CALL gds.pageRank.stats('myGraph', {nodeLabels: ['Person']})

// will use nodes projected as 'Person' or 'City' in 'myGraph'
CALL gds.pageRank.stats('myGraph', {nodeLabels: ['Person', 'City']})

```

## Node properties

It is often useful to load an in-memory graph with more than one node property. A typical scenario is running different seedable algorithms on the same graph, but with different node properties as seed. We can load multiple node properties for each node projection using node property mappings. A node property mapping maps a user-defined property key to a property key in the Neo4j database. Any algorithm that supports node properties can refer to these user-defined property keys.

```
{  
    <node-label>: {  
        label: <neo4j-label>,  
        properties: {  
            <property-key-1>: {  
                property: <neo-property-key>,  
                defaultValue: <numeric-value>  
            },  
            <property-key-2>: {  
                property: <neo-property-key>,  
                defaultValue: <numeric-value>  
            },  
            // ...  
            <property-key-n>: {  
                property: <neo-property-key>,  
                defaultValue: <numeric-value>  
            }  
        }  
    }  
}
```

- **property-key-i** denotes the property key in the projected graph
  - **neo-property-key** denotes the property key in the Neo4j graph
    - The property key must exist in the Neo4j database
    - If not specified, **neo-property-key** defaults to **property-key-i**
  - **numeric-value** is used if the property does not exist for a node
    - If not specified, **numeric-value** defaults to **NaN**

For the following example, let's assume that each **City** node stores two properties: the **population** of the city and an optional **stateId** that identifies the state in which the city is located. We want to project both properties and project **stateId** to the custom property key **community**.

Create a graph with multiple node properties:

```
CALL gds.graph.create(
  'my-graph', {
    City: {
      properties: {
        stateId: {
          property: 'stateId'
        },
        population: {
          property: 'population'
        }
      }
    },
    '*'
)
YIELD graphName, nodeCount, relationshipCount;
```

If we do not need to rename the node property keys or give a default value, we can use the following shorthand syntax.

```
CALL gds.graph.create('my-graph', 'City', '*', {
  nodeProperties: ['population', 'stateId']
}
)
YIELD graphName, nodeCount, relationshipCount;
```

It is also possible to rename the property key during projection. In the example, we project the property key `stateId` to a custom property key `community`. When we use the projected graph in an algorithm, we refer to the custom property key instead.

*Project node properties for all projected node labels:*

```
CALL gds.graph.create('my-graph', 'City', '*', {
  nodeProperties: ['population', { community: 'stateId' }]
}
)
YIELD graphName, nodeCount, relationshipCount;
```

The projected properties can be referred to by any algorithm that uses properties as input, for example, [Label Propagation](#).

```

CALL gds.labelPropagation.stream(
  'my-graph', {
    seedProperty: 'community'
  }
) YIELD nodeId, communityId;

```

## Relationship projections

A relationship projection defines how a specific subset of Neo4j relationships is projected into the in-memory graph.

The following map-like syntax shows the general way of defining relationship projections:

```

{
  <relationship-type-1>: {
    type: <neo4j-type>,
    orientation: <orientation>,
    aggregation: <aggregation-type>,
    properties: <relationship-property-mappings>
  },
  <relationship-type-2>: {
    type: <neo4j-type>,
    orientation: <orientation>,
    aggregation: <aggregation-type>,
    properties: <relationship-property-mappings>
  },
  // ...
  <relationship-type-n>: {
    type: <neo4j-type>,
    orientation: <orientation>,
    aggregation: <aggregation-type>,
    properties: <relationship-property-mappings>
  }
}

```

- **relationship-type-i** denotes the relationship type in the projected graph
  - **neo4j-type** denotes the relationship type in the Neo4j graph
    - The relationship type must exist in the Neo4j database
    - If not specified, **neo4j-type** defaults to **relationship-type-i**
  - **orientation** denotes how Neo4j relationships are represented in the projected graph. The following values are allowed:
    - **NATURAL**: each relationship is projected the same way as it is stored in Neo4j (default)
    - **REVERSE**: each relationship is reversed during graph projection
    - **UNDIRECTED**: each relationship is projected in both natural and reverse orientation

- **aggregation-type** denotes how parallel relationships and their properties are handled. The specified value is applied to all property mappings that have no aggregation specified. The following values are allowed:
  - **NONE**: parallel relationships are not aggregated (default)
  - **MIN**, **MAX**, **SUM**: applied to the numeric properties of parallel relationships
  - **SINGLE**: a single, arbitrary relationship out of the parallel relationships is projected
  - **COUNT**: counts the number of non-null numeric properties
    - If the special property name '**\***' is used, **COUNT** will count parallel relationships
- **relationship-property-mappings** denotes a set of mappings between Neo4j and in-memory relationship properties

In the following example, we want to project **City** nodes as well as **ROAD** and **RAIL** relationships into the in-memory graph.

```
CALL gds.graph.create(
  'my-graph',
  'City',
  {
    ROAD: {
      type: 'ROAD',
      orientation: 'NATURAL'
    },
    RAIL: {
      type: 'RAIL',
      orientation: 'NATURAL'
    }
  }
)
YIELD graphName, nodeCount, relationshipCount;
```

In the above example, we are using the same relationship type as in the Neo4j database as well as the default **orientation**. In that case we can use the following syntactic sugar, similar to node projections.

```
CALL gds.graph.create( 'my-graph', 'City', ['ROAD', 'RAIL'])
YIELD graphName, nodeCount, relationshipCount;
```

Projecting multiple relationship types enables algorithms to only use a subset of those.

```

// Uses 'ROAD' relationships for computing Page Rank of cities
CALL gds.pageRank.stream('my-graph', { relationshipTypes: ['ROAD'] }) YIELD nodeId,
score;

// Uses 'RAIL' relationships for computing Page Rank of cities
CALL gds.pageRank.stream('my-graph', { relationshipTypes: ['RAIL'] }) YIELD nodeId,
score;

```

## Projection orientation

By default, relationships are projected in their natural representation, i.e., in the same way as they are stored in Neo4j. Using the `orientation` key within a relationship projection definition, we can alter that behaviour. There are three possible values: `NATURAL`, `REVERSE` and `UNDIRECTED` which can be best described from a node's perspective:

- `NATURAL` is the default behaviour and projects relationships that are pointing away from a node.
- `REVERSE` projects relationships that are pointing towards a node.
- `UNDIRECTED` projects relationships in both, natural and reversed order.

Consider the following graph containing `Person` nodes connected by `KNOWS` relationships. A `KNOWS` relationship is directed, as one person might know another person, but not necessarily the other way around.

```

CREATE (alice:Person {name: 'Alice'})
CREATE (bob:Person {name: 'Bob'})
CREATE (eve:Person {name: 'Eve'})

CREATE (alice)-[:KNOWS]->(bob)
CREATE (bob)-[:KNOWS]->(eve)
CREATE (eve)-[:KNOWS]->(bob);

```

In a `NATURAL` projection, Alice has one relationship to Bob, Bob has one relationship to Eve who in turn also has one relationship to Bob. In a `REVERSE` projection, Alice has no relationships as there is no relationship pointing towards Alice. Bob and Eve would have one relationship each, as they point to each other. In an `UNDIRECTED` projection, Alice would have one relationship representing the outgoing relationship. However, Bob and Eve would have two relationships each as the outgoing and incoming relationships are viewed independently.

To create a graph projection with different projection types, we use the following syntax:

```

CALL gds.graph.create(
  'my-graph',
  'Person',
  {
    KNOWS: {
      type: 'KNOWS',
      orientation: 'NATURAL'
    },
    KNOWN_BY: {
      type: 'KNOWS',
      orientation: 'REVERSE'
    },
    FRIEND_OF: {
      type: 'KNOWS',
      orientation: 'UNDIRECTED'
    }
  }
)
YIELD graphName, nodeCount, relationshipCount;

```

As in the previous example, we can refer to a subset of the projected relationships when running an algorithm. If we run the examples, we can see different ranks for the individual nodes. The Page Rank algorithm evenly distributes ranks along the relationships of a node. In the reverse case, Alice has no relationships which leads to a different result.

```

// Uses 'KNOWS' relationships for computing Page Rank of persons
CALL gds.pageRank.stream('my-graph', { relationshipTypes: ['KNOWS'] }) YIELD nodeId,
score;

// Uses 'KNOWN_BY' relationships for computing Page Rank based on reversed
relationships
CALL gds.pageRank.stream('my-graph', { relationshipTypes: ['KNOWN_BY'] }) YIELD
nodeId, score;

// Uses 'FRIEND_OF' relationships for computing Page Rank based on both projection
types
CALL gds.pageRank.stream('my-graph', { relationshipTypes: ['FRIEND_OF'] }) YIELD
nodeId, score;

```

 Creating a projection consumes additional memory as those projections are stored in individual in-memory data structures. Sometimes it is possible to combine relationship projections instead of creating a new one. In the above example, the `FRIEND_OF` projection is equivalent to using `['KNOWS', 'KNOWN_BY']` as a relationship type predicate. This is not possible, if we use different aggregations for the single projections.

## Relationship properties

Similar to node properties, relationship projections support specifying relationship properties. We can specify multiple relationship properties for each relationship projection using relationship property mappings. A relationship property mapping maps a user-defined property key to a property key in the Neo4j database. The parameter is configured using a map in which each key refers to a user-defined property key.

The following map-like syntax shows the general way of defining relationship property mappings:

```
{  
    <relationship-type-1>: {  
        type: <neo4j-type>,  
        orientation: <orientation-type>,  
        aggregation: <aggregation-type>,  
        properties: {  
            <property-key-1>: {  
                property: <neo4j-property-key>,  
                defaultValue: <numeric-value>,  
                aggregation: <aggregation-type>  
            },  
            <property-key-2>: {  
                property: <neo4j-property-key>,  
                defaultValue: <numeric-value>,  
                aggregation: <aggregation-type>  
            },  
            // ...  
            <property-key-n>: {  
                property: <neo4j-property-key>,  
                defaultValue: <numeric-value>,  
                aggregation: <aggregation-type>  
            }  
        }  
    }  
}
```

- **property-key-i** denotes the name of the property in the projected graph
  - **neo4j-property-key** denotes the name of the property in the Neo4j graph
    - The property key must exist in the Neo4j database
    - **neo4j-property-key** defaults to **property-key-i**
    - The special property key '**\***' is allowed in combination with the **COUNT** aggregation
  - **numeric-value** is used if the property does not exist for a relationship
    - **numeric-value** defaults to **NaN**
  - **aggregation-type** denotes how properties of parallel relationships are handled. The specified value overrides the aggregation type specified for the enclosing relationship projection. The following values are allowed:

- **NONE**: parallel relationships are not aggregated (default)
- **MIN, MAX, SUM**: applied to the numeric properties of parallel relationships
- **SINGLE**: a single, arbitrary relationship out of the parallel relationships is projected
- **COUNT**: counts the number of non-null numeric properties
  - If the special property name '\*' is used, **COUNT** will count parallel relationships

In the following example, we want to project **City** nodes and **ROAD** relationships. For nodes we project the **stateId** property.

*Create a graph with multiple node and relationship properties:*

```
CALL gds.graph.create(
  'my-graph', {
    City: {
      properties: {
        community: {
          property: 'stateId'
        }
      }
    },
    {
      ROAD: {
        properties: {
          quality: {
            property: 'condition'
          },
          distance: {
            property: 'length'
          }
        }
      }
    }
  )
YIELD graphName, nodeCount, relationshipCount;
```

We can use the following shorthand syntax to express the same projection.

```
CALL gds.graph.create(
  'my-graph', 'City', 'ROAD', {
    nodeProperties: { community: 'stateId' },
    relationshipProperties: [{ quality: 'condition' }, { distance: 'length' }]
  }
)
YIELD graphName, nodeCount, relationshipCount;
```

The projected properties can be referred to by any algorithm that uses properties as input, for example [Label Propagation](#).

```

// Option 1: Use the road quality as relationship weight
CALL gds.labelPropagation.stream(
    'my-graph', {
        seedProperty: 'community',
        relationshipWeightProperty: 'quality'
    }
) YIELD nodeId, communityId;
// Option 2: Use the distance between cities as relationship weight
CALL gds.labelPropagation.stream(
    'my-graph', {
        seedProperty: 'community',
        relationshipWeightProperty: 'distance'
    }
) YIELD nodeId, communityId;

```

## Relationship aggregations

Relationship projections offer different ways of handling multiple - so called "parallel" - relationships between a given pair of nodes. The default is the **NONE** aggregation which keeps all parallel relationships and directly projects them into the in-memory graph. All other aggregations project all the parallel relationships between a pair of nodes into a single relationship.

In the following example, we want to aggregate all **ROAD** relationships between two cities to a single relationship. While doing so, we compute the maximum quality of the parallel relationships and store it on the resulting relationship.

Create a graph with aggregated parallel relationships using the maximum value of the `condition` property:

```
CALL gds.graph.create(
  'my-graph', {
    City: {
      properties: {
        community: {
          property: 'stateId'
        }
      }
    },
    ROAD: {
      properties: {
        maxQuality: {
          property: 'condition',
          aggregation: 'MAX',
          defaultValue: 1.0
        }
      }
    }
  }
)
YIELD graphName, nodeCount, relationshipCount;
```

Create a graph with aggregated relationships using the parallel relationship count as a relationship property:

```
CALL gds.graph.create(
  'my-graph', {
    City: {
      properties: {
        community: {
          property: 'stateId'
        }
      }
    },
    ROAD: {
      properties: {
        roadCount: {
          property: 'condition',
          aggregation: 'COUNT'
        }
      }
    }
  }
)
YIELD graphName, nodeCount, relationshipCount;
```

Since we have only one node projection and one relationship projection, we can use the following shorthand syntax.

```
CALL gds.graph.create(
    'my-graph', 'City', 'ROAD', {
        nodeProperties: { community: 'stateId' },
        relationshipProperties: { maxQuality: { property: 'condition', aggregation: 'MAX', defaultValue: 1.0 } }
    }
)
YIELD graphName, nodeCount, relationshipCount;
```

As before, the projected properties can be referred to by any algorithm that uses properties as input, for example [Label Propagation](#).

```
CALL gds.labelPropagation.stream(
    'my-graph', {
        seedProperty: 'community',
        relationshipWeightProperty: 'maxQuality'
    }
)
YIELD nodeId, communityId;
```

## Cypher projection

*This chapter explains how to create a graph using a Cypher projection.*

If the [Native projection](#) is not expressive enough to describe the in-memory graph, we can instead use Cypher queries to select nodes and relationships. One benefit of using Cypher queries is the possibility to form the graph from data that exists only at query time. A common use case is the reduction of paths into single relationships between the start and end node of the path.

*The following query reduces a 2-hop path to a single relationship effectively representing co-authors:*

```
MATCH (p1:Author)-[:WROTE]->(a:Article)<-[ :WROTE ]-(p2:Author)
RETURN id(p1) AS source, id(p2) AS target, count(a) AS weight
```

Cypher projections are especially useful during the development phase. Their flexibility is convenient when exploring data and algorithms, and designing a workflow. However, creating a graph from a Cypher projection can be significantly slower than creating it directly from the Neo4j store files. For production, it is recommended to adapt the domain model in a way that it can take advantage of the loading speed of [native projections](#).



During graph creation, GDS performs automatic [memory estimation and potential execution blocking](#) by default. This feature is off by default for Cypher projections as the memory consumption might be overestimated.

This section includes:

- [Syntax](#)
- [Cypher query constraints](#)
- [Relationship types](#)
- [Relationship orientation](#)
- [Relationship aggregation](#)
- [Using query parameters](#)

## 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  
)
```

Table 9. Parameters

Name	Optional	Description
graphName	no	The name under which the graph is stored in the catalog.
nodeQuery	no	Cypher query to project nodes.
relationshipQuery	no	Cypher query to project relationships.
configuration	yes	Additional parameters to configure the Cypher projection.

Table 10. Configuration

Name	Type	Default	Description
readConcurrency	Integer	4	The number of concurrent threads used for creating the graph.
relationshipProperties	Map	empty map	Mappings between the RETURN items and relationship properties in the graph projection.
validateRelationships	Boolean	true	Whether to throw an error if relationships contain nodes not included in the nodeQuery.
parameters	Map	empty map	A map of user-defined query parameters that are passed into the node and relationship query.

To get information about a stored named graph, including its schema, one can use `gds.graph.list`.

## Query constraints

The node query projects nodes and optionally their properties to an in-memory graph. Each row in the query result represents a node in the projected graph.

The query result must contain a column called `id`. The value in that column is used to uniquely identify the node.

*Simple example of a node query used for Cypher projection.*

```
MATCH (n) RETURN id(n) AS id
```

The relationship query projects relationships and optionally their type and properties to an in-memory graph. Each row in the query result represents a relationship in the projected graph.

The query result must contain a column called `source` and a column called `target`. The values in those columns represent the source node id and the target node id of the relationship. The values are used to connect the relationships to the nodes selected by the node query. If either the source or the target value can not be mapped to a node, the relationship is not projected.

*Simple example of a relationship query used for Cypher projection.*

```
MATCH (n)-->(m) RETURN id(n) AS source, id(m) AS target
```

Using both example queries in a Cypher projection, we can project the whole Neo4j graph into an in-memory graph and store it in the catalog:

```
CALL gds.graph.create.cypher(
    'my-cypher-graph',
    'MATCH (n) RETURN id(n) AS id',
    'MATCH (n)-->(m) RETURN id(n) AS source, id(m) AS target'
)
```

 Cypher projections allow creating graphs from arbitrary query results, regardless of whether these map to actual identifiers in the Neo4j graph. **Executing an algorithm on such a graph in `write` mode may lead to unexpected changes in the Neo4j database.**

## Node and relationship properties

Similar to the default native projection, we can load node and relationship properties using a Cypher projection.

Both node and relationship queries must return their respective mandatory columns, i.e., `id`, `source` and `target`. If a query returns additional columns, those columns are used as node and relationship

properties, respectively.

The values stored in property columns need to be numeric. If a value is `null` a default value (`Double.NaN`) is loaded instead. If we want to use a different default value, the `coalesce` function can be used.

The following Cypher projection loads multiple node and relationship properties:

*Projecting query columns into node and relationship properties.*

```
CALL gds.graph.create.cypher(
    'my-cypher-graph',
    'MATCH (n:City) RETURN id(n) AS id, n.stateId AS community, n.population AS
population',
    'MATCH (n:City)-[r:ROAD]->(m:City) RETURN id(n) AS source, id(m) AS target,
r.distance AS distance, coalesce(r.condition, 1.0) AS quality'
)
```

The projected properties can be referred to by any algorithm that uses properties as input, for example, [Label Propagation](#).

```
CALL gds.labelPropagation.stream(
    'my-cypher-graph', {
        seedProperty: 'community',
        relationshipWeightProperty: 'quality'
    }
)
```

## Node labels

Native projections supports specifying multiple node labels which can be filtered in an individual algorithm execution. Cypher projections can achieve the same feature by returning the node label in the node query. If a column called `labels` is present in the node query result, we use the values in that column to distinguish node labels. This column is expected to return a list of strings.

Consider the following example where `Author` nodes are connected by `WRITED` relationships to either `Article` or `Book` nodes.

*Using the `labels` column to distinguish between node labels.*

```
CALL gds.graph.create.cypher(
    'my-cypher-graph',
    'MATCH (n) WHERE n:Author OR n:Article OR n:Book RETURN id(n) AS id, labels(n) AS
labels',
    'MATCH (n:Author)-[r:WRITED]->(m) RETURN id(n) AS source, id(m) AS target'
)
```

The created graph will be composed of nodes labeled with either `:Book`, `:Article`, or `:Author`. This

allows us to apply a node filter during algorithm execution:

*Using a node filter to run the algorithm on a subgraph.*

```
CALL gds.labelPropagation.stream(
  'my-cypher-graph', {
    nodeLabels: ['Author', 'Book']
  }
)
```

## Relationship types

The native projection supports loading multiple relationship types which can be filtered in an individual algorithm execution. The Cypher projection can achieve the same feature by returning the relationship type in the query. If the `type` column is present in the query result, we use the values in that column to distinguish relationship types.

For the following example, let's assume `City` nodes to be connected by either `ROAD` or `RAIL` relationships.

*Using the `type` column to distinguish between multiple relationship types.*

```
CALL gds.graph.create.cypher(
  'my-cypher-graph',
  'MATCH (n:City) RETURN id(n) AS id',
  'MATCH (n:City)-[r:ROAD|RAIL]->(m:City) RETURN id(n) AS source, id(m) AS target,
  type(r) AS type'
)
```

The loaded graph will be composed of the two relationship types. This allows us to apply a relationship filter during algorithm execution:

*Using a relationship filter to run the algorithm on a subgraph.*

```
CALL gds.labelPropagation.stream(
  'my-cypher-graph', {
    relationshipTypes: ['ROAD']
  }
)
```

## Relationship orientation

The native projection supports specifying an orientation per relationship type. The cypher projection can achieve the same feature by adjusting the `MATCH` clause of the relationship query.

*Loading the relationships with orientation NATURAL*

```
CALL gds.graph.create.cypher(  
  'my-cypher-graph',  
  'MATCH (n:City) RETURN id(n) AS id',  
  'MATCH (n:City)-[r:ROAD|RAIL]->(m:City) RETURN id(n) AS source, id(m) AS target,  
  type(r) AS type'  
)
```

*Loading the relationships with orientation UNDIRECTED*

```
CALL gds.graph.create.cypher(  
  'my-cypher-graph',  
  'MATCH (n:City) RETURN id(n) AS id',  
  'MATCH (n:City)-[r:ROAD|RAIL]-(m:City) RETURN id(n) AS source, id(m) AS target,  
  type(r) AS type'  
)
```

Note the missing arrow in the **Match** clause of the relationship query.

*Loading the relationships with orientation REVERSE*

```
CALL gds.graph.create.cypher(  
  'my-cypher-graph',  
  'MATCH (n:City) RETURN id(n) AS id',  
  'MATCH (n:City)<-[r:ROAD|RAIL]-(m:City) RETURN id(n) AS source, id(m) AS target,  
  type(r) AS type'  
)
```

The **REVERSE** orientation can also be achieved by swapping source and target in the **RESULT** clause.

## Relationship aggregation

The property graph model supports parallel relationships, which means two nodes can be connected by multiple relationships of the same relationship type. For some algorithms, we want the projected graph to contain at most one relationship between two nodes.

The simplest way to achieve this is to use the **DISTINCT** operator in the relationship query:

```
MATCH (n:City)-[r:ROAD]->(m:City)  
RETURN DISTINCT id(n) AS source, id(m) AS target
```

If we also want to load relationship properties, aggregating the values of parallel edges can also be achieved using Cypher.

```
MATCH (n:City)-[r:ROAD]->(m:City)
RETURN
    id(n) AS source,
    id(m) AS target,
    min(r.distance) AS minDistance,
    coalesce(max(r.condition), 1.0) AS maxQuality
```

## 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.

```
CALL gds.graph.create.cypher(
    'my-cypher-graph',
    'MATCH (n:City) WHERE n.name IN $cities RETURN id(n) AS id',
    'MATCH (n:City)-[r:ROAD]->(m:City) WHERE n.name IN $cities AND m.name IN $cities
    RETURN id(n) AS source, id(m) AS target',
    {
        parameters: { cities: ["Leipzig", "Malmö"] }
    }
)
```

## Anonymous graphs

*This chapter explains how to create an anonymous graph for a single algorithm execution.*

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 keys `nodeQuery` and `relationshipQuery`.

*Anonymous cypher projection syntax*

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

## Utility functions

This section describes the utility functions and shows their usage in concrete examples.

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

*Usage:*

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

*Table 11. Results*

<code>version</code>
"1.2.2"

## Numeric Functions

*Table 12. 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 true iff the given argument is a finite value (not ±Infinity, NaN, or null).
<code>gds.util.isInfinite</code>	Return true iff the given argument is not a finite value (not ±Infinity, NaN, or null).

## Syntax

Name	Parameter
<code>gds.util.NaN()</code>	-

Name	Parameter
<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 13. 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 14. Results

isInfinite
false
true
true

The utility function `gds.util.NaN` can be used as an 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](#).

## Node and Path Functions

Table 15. Node and Path 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: 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 16. 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 17. 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.

## Catalog Functions

Catalog functions allow accessing in-memory graphs directly from a Cypher query.

Table 18. 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

Create a graph in the catalog:

```
CALL gds.graph.create('my-graph', 'User', 'LINK');
```

Run an algorithm that updates the named-graph:

```
CALL gds.pageRank.mutate('my-graph', { mutateProperty: 'score' })
```

We can now access the property `score` without writing the data to Neo4j.

Access a property node property for Alice:

```
MATCH (alice:User)
WHERE alice.name = 'Alice'
RETURN
  alice.name AS name,
  gds.util.nodeProperty('my-graph', id(alice), 'score') AS score
```

Table 19. Results

name	score
"Alice"	0.15000000000000002

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

*Access a property node property from User for Alice:*

```
MATCH (alice:User)
WHERE alice.name = 'Alice'
RETURN
  alice.name AS name,
  gds.util.nodeProperty('my-graph', id(alice), 'score', 'User') AS score
```

*Table 20. Results*

<b>name</b>	<b>score</b>
"Alice"	0.15000000000000002

# 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 algorithms](#)
- [Community detection algorithms](#)
- [Similarity algorithms](#)
- [Path finding algorithms](#)
- [Link Prediction algorithms](#)
- [Auxiliary procedures](#)

## 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.

- **write**

- Writes the results of the algorithm to the Neo4j database and returns a single record of summary statistics.

- **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.

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>](<br/>graphName: String,<br/>configuration: Map<br/>)
```

*Syntax composition for the anonymous graph variant:*

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

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

## Configuration options

All algorithms allow adjustment of their runtime characteristics through a set of configuration parameters. Some of these parameters are algorithm specific, however, many of them are shared between the 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 [CPU](#) of the System

Requirements.

#### **relationshipTypes - 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 then 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.

#### **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**

## **Centrality algorithms**

*This chapter provides explanations and examples for each of the centrality algorithms in the Neo4j Graph Data Science library.*

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](#)

- Alpha
  - ArticleRank
  - Betweenness Centrality
  - Closeness Centrality
  - Degree Centrality
  - Eigenvector Centrality

## Page Rank

***This section describes the Page Rank algorithm in the Neo4j Graph Data Science library.***

Page Rank is an algorithm that measures the influence or importance of nodes in a directed graph.

It is computed by either iteratively distributing each node's score (originally based on degree) over its neighbours. Theoretically this is equivalent traversing the graph in a random fashion and counting the frequency of hitting each node during these walks.

This section includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Unweighted](#)
  - [Weighted](#)
  - [Personalized](#)
  - [Memory Estimation](#)
  - [Stats](#)
- [Usage](#)

### Introduction

The Page Rank 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.

Page Rank is defined 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 (i.e., are citations).
- $d$  is a damping factor which can be set between 0 and 1. 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](#).

## Syntax

### Write mode

*Run PageRank in write mode on a graph stored in the catalog.*

```
CALL gds.pageRank.write(
    graphName: String,
    configuration: Map
)
YIELD
    // general write return columns
    ranIterations: Integer,
    didConverge: Boolean
```

*Table 21. 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 22. General configuration for algorithm execution on a named graph.*

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	String[]	[ '*' ]	yes	Filter the named graph using the given relationship types.

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	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 score is written to.

Run PageRank in write mode on an anonymous graph.

```
CALL gds.pageRank.write(configuration: Map)
YIELD
    // general write return columns
    ranIterations: Integer,
    didConverge: Boolean
```

Table 23. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 24. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, String[] or Map	null	yes	The relationship properties to project during anonymous graph creation.

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 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 score is written to.

Table 25. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation.
maxIterations	Integer	20	yes	The maximum number of iterations of Page Rank 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	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.

Table 26. 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.
writeMillis	Integer	Milliseconds for writing result data back.
nodePropertiesWritten	Integer	The number of properties that were written to Neo4j.

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.
configuration	Map	The configuration used for running the algorithm.

### Mutate mode

Run PageRank in mutate mode on a graph stored in the catalog.

```
CALL gds.pageRank.mutate(
    graphName: String,
    configuration: Map
)
YIELD
    // general mutate return columns
    ranIterations: Integer,
    didConverge: Boolean
```

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.

The following will run the algorithm and store the results in `myGraph`:

```
CALL gds.pageRank.mutate('myGraph', { mutateProperty: 'pagerank' })
```

### Stream mode

Run PageRank in stream mode on a graph stored in the catalog.

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

Table 27. 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 28. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 score is written to.

Run PageRank in stream mode on an anonymous graph.

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

Table 29. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 30. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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.

Name	Type	Default	Optional	Description
nodeProperties	String, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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 score is written to.

Table 31. Algorithm specific configuration

Name	Type	Default	Optional	Description
dampingFactor	Float	0.85	yes	The damping factor of the Page Rank calculation.
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	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.

Table 32. Results

Name	Type	Description
nodeId	Integer	Node ID
score	Float	Page Rank

## Stats mode

*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
```

*Table 33. 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.

*Run PageRank in stats mode on an anonymous graph.*

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

*Table 34. Parameters*

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

The configuration is the same as for the `write` mode.

The `mode` can be substituted with the available modes (`stream`, `write` and `stats`).

#### Estimate mode

The following will estimate the memory requirements for running the algorithm. The `mode` can be substituted with the available modes (`stream`, `write` and `stats`).

Run PageRank in estimate mode on a named graph.

```
CALL gds.pageRank.<mode>.estimate(  
    graphName: String,  
    configuration: Map  
)
```

Table 35. 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.

Run PageRank in estimate mode on an anonymous graph.

```
CALL gds.pageRank.<mode>.estimate(configuration: Map)
```

Table 36. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

For memory estimation on a named graph the configuration for creating that graph is used. For the anonymous graph, configuration is used as for [Native projection](#) or [Cypher projection](#). The graph estimation on an anonymous graph is based on the configuration pertaining to anonymous creation or so-called fictive estimation controlled by the options in the table below.

Table 37. Configuration

Name	Type	Default	Optional	Description
nodeCount	Integer	-1	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	-1	yes	The number of relationships in a fictive graph.

Setting the `nodeCount` and `relationshipCount` parameters results in fictive graph estimation which allows a memory estimation without loading the graph. Additionally algorithm specific parameters can also be provided as config which influence the estimation of memory usage specific to the algorithm.

Table 38. Memory estimation results.

Name	Type	Description
nodeCount	Integer	Node count of the graph used in the estimation.
relationshipCount	Integer	Relationship count of the graph used in the estimation.

Name	Type	Description
requiredMemory	Integer	Human readable version for required memory.
bytesMin	Integer	Minimum number of bytes to be consumed.
bytesMax	Integer	Maximum number of bytes to be consumed.
heapPercentageMin	Float	The minimum percentage of the configured max heap (-Xmx) to be consumed.
heapPercentageMax	Float	The maximum percentage of the configured max heap (-Xmx) to be consumed.
treeView	Map	Human readable version of memory estimation.
mapView	Map	Detailed information on memory consumption.

## Examples

Consider the graph created by the following Cypher statement:

```

CREATE (home:Page {name:'Home'})
CREATE (about:Page {name:'About'})
CREATE (product:Page {name:'Product'})
CREATE (links:Page {name:'Links'})
CREATE (a:Page {name:'Site A'})
CREATE (b:Page {name:'Site B'})
CREATE (c:Page {name:'Site C'})
CREATE (d:Page {name:'Site D'})

CREATE (home)-[:LINKS {weight: 0.2}]->(about)
CREATE (home)-[:LINKS {weight: 0.2}]->(links)
CREATE (home)-[:LINKS {weight: 0.6}]->(product)
CREATE (about)-[:LINKS {weight: 1.0}]->(home)
CREATE (product)-[:LINKS {weight: 1.0}]->(home)
CREATE (a)-[:LINKS {weight: 1.0}]->(home)
CREATE (b)-[:LINKS {weight: 1.0}]->(home)
CREATE (c)-[:LINKS {weight: 1.0}]->(home)
CREATE (d)-[:LINKS {weight: 1.0}]->(home)
CREATE (links)-[:LINKS {weight: 0.8}]->(home)
CREATE (links)-[:LINKS {weight: 0.05}]->(a)
CREATE (links)-[:LINKS {weight: 0.05}]->(b)
CREATE (links)-[:LINKS {weight: 0.05}]->(c)
CREATE (links)-[:LINKS {weight: 0.05}]->(d)

```

This graph represents seven pages, linking to 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 standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could also be used.

The following statement will create the graph and store it in the graph catalog.

```
CALL gds.graph.create(
  'myGraph',
  'Page',
  'LINKS',
  {
    relationshipProperties: 'weight'
  }
)
```

In the following examples we will demonstrate using the Page Rank algorithm on this graph.

### Unweighted

The following will run the algorithm and stream results:

```
CALL gds.pageRank.stream('myGraph', { maxIterations: 20, dampingFactor: 0.85 })
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC, name ASC
```

Table 39. Results

name	score
"Home"	3.2362017153762284
"About"	1.0611098567023873
"Links"	1.0611098567023873
"Product"	1.0611098567023873
"Site A"	0.3292259009438567
"Site B"	0.3292259009438567
"Site C"	0.3292259009438567
"Site D"	0.3292259009438567

To instead write the page-rank score to a node property in the Neo4j graph, use this query:

The following will run the algorithm and write back results:

```
CALL gds.pageRank.write('myGraph', {
  maxIterations: 20,
  dampingFactor: 0.85,
  writeProperty: 'pagerank'
})
YIELD nodePropertiesWritten AS writtenProperties, ranIterations
```

Table 40. Results

writtenProperties	ranIterations
8	20

## Weighted

The following will run the algorithm and stream results:

```
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 41. Results

name	score
"Home"	3.5528567278757683
"Product"	1.9541301048360766
"About"	0.7513767024036497
"Links"	0.7513767024036497
"Site A"	0.18167360233856014
"Site B"	0.18167360233856014
"Site C"	0.18167360233856014
"Site D"	0.18167360233856014

To instead write the page-rank score to a node property in the Neo4j graph, use this query:

The following will run the algorithm and write back results:

```
CALL gds.pageRank.write('myGraph', {
    maxIterations: 20,
    dampingFactor: 0.85,
    writeProperty: 'pagerank',
    relationshipWeightProperty: 'weight'
})
YIELD nodePropertiesWritten AS writtenProperties, ranIterations
```

Table 42. Results

writtenProperties	ranIterations
8	20

## Personalized

Personalized Page Rank is a variation of Page Rank which is biased towards a set of [sourceNodes](#). This variant of Page Rank is often used as part of [recommender systems](#).

The following examples show how to run Page Rank 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 43. Results

name	score
"Home"	0.4015879109501838
"Site A"	0.1690742586266424
"About"	0.11305649263085797
"Links"	0.11305649263085797
"Product"	0.11305649263085797
"Site B"	0.01907425862664241
"Site C"	0.01907425862664241
"Site D"	0.01907425862664241

*The following will run the algorithm and write back results:*

```
MATCH (siteA:Page {name: 'Site A'})
CALL gds.pageRank.write('myGraph', {
    maxIterations: 20,
    dampingFactor: 0.85,
    writeProperty: 'pagerank',
    sourceNodes: [siteA]
})
YIELD nodePropertiesWritten, ranIterations
RETURN nodePropertiesWritten AS writtenProperties, ranIterations
```

Table 44. Results

writtenProperties	ranIterations
8	20

## 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 45. Results

nodeCount	relationshipCoun t	bytesMin	bytesMax	requiredMemory
8	14	1560	1560	"1560 Bytes"

## 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 46. Results

max
3.236204147338867

The above query returned a so called centrality histogram which can be used to monitor the distribution of page rank values across all computed nodes. This can be useful for inspecting the computed scores or perform normalizations.

## Usage

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

- If there are no links from within a group of pages to outside of the group, then the group is considered a spider trap.
- Rank sink can occur when a network of pages form an infinite cycle.
- Dead-ends occur when pages have no out-links. If a page contains a link to another page which has no out-links, the link would be known as a dangling link.

## ArticleRank

**This section describes the ArticleRank algorithm in the Neo4j Graph Data Science library.**

ArticleRank is a variant of the [Page Rank algorithm](#), which measures the **transitive** influence or connectivity of nodes.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [ArticleRank algorithm sample](#)
- [Syntax](#)
- [Graph type support](#)

### History and explanation

Where ArticleRank differs to Page Rank is that Page Rank assumes that relationships from nodes that have a low out-degree are more important than relationships from nodes with a higher out-degree. ArticleRank weakens this assumption.

ArticleRank is defined in [ArticleRank: a PageRank-based alternative to numbers of citations for analysing citation networks](#) as follows:

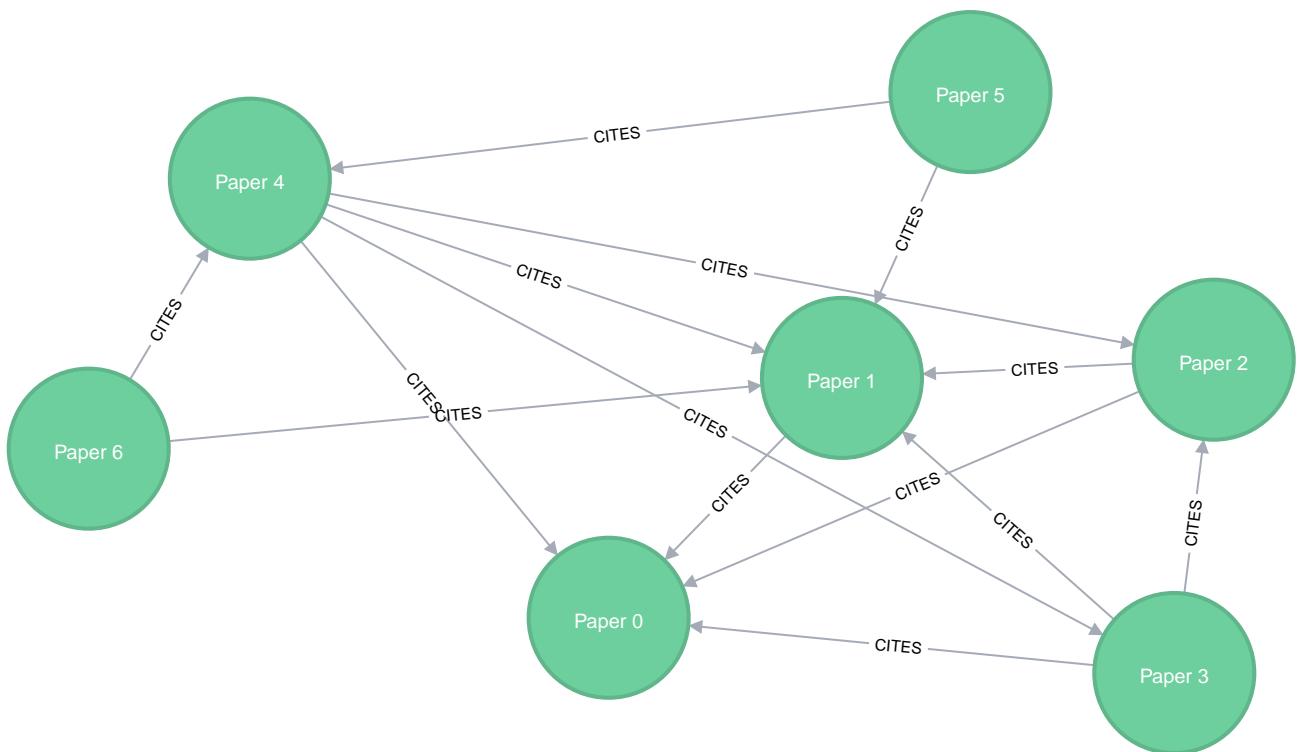
$$AR(A) = (1-d) + d \left( \frac{AR(T_1)}{C(T_1) + C(\text{AVG})} + \dots + \frac{AR(T_n)}{C(T_n) + C(\text{AVG})} \right)$$

where,

- we assume that a page **A** has pages **T<sub>1</sub>** to **T<sub>n</sub>** which point to it (i.e., are citations).
- **d** is a damping factor which can be set between 0 and 1. It is usually set to 0.85.
- **C(A)** is defined as the number of links going out of page **A**.
- **C(AVG)** is defined as the average number of links going out of all pages.

### ArticleRank algorithm sample

This sample will explain the ArticleRank algorithm, using a simple graph:



The following will create a sample graph:

```

MERGE (paper0:Paper {name:'Paper 0'})
MERGE (paper1:Paper {name:'Paper 1'})
MERGE (paper2:Paper {name:'Paper 2'})
MERGE (paper3:Paper {name:'Paper 3'})
MERGE (paper4:Paper {name:'Paper 4'})
MERGE (paper5:Paper {name:'Paper 5'})
MERGE (paper6:Paper {name:'Paper 6'})

```

```
MERGE (paper1)-[:CITES]->(paper0)
```

```

MERGE (paper2)-[:CITES]->(paper0)
MERGE (paper2)-[:CITES]->(paper1)

```

```

MERGE (paper3)-[:CITES]->(paper0)
MERGE (paper3)-[:CITES]->(paper1)
MERGE (paper3)-[:CITES]->(paper2)

```

```

MERGE (paper4)-[:CITES]->(paper0)
MERGE (paper4)-[:CITES]->(paper1)
MERGE (paper4)-[:CITES]->(paper2)
MERGE (paper4)-[:CITES]->(paper3)

```

```

MERGE (paper5)-[:CITES]->(paper1)
MERGE (paper5)-[:CITES]->(paper4)

```

```

MERGE (paper6)-[:CITES]->(paper1)
MERGE (paper6)-[:CITES]->(paper4)

```

The following will run the algorithm and stream results:

```
CALL gds.alpha.articleRank.stream({
  nodeProjection: 'Paper',
  relationshipProjection: 'CITES',
  iterations: 20,
  dampingFactor: 0.85
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS page, score
ORDER BY score DESC
```

The following will run the algorithm and write back results:

```
CALL gds.alpha.articleRank.write({
  nodeProjection: 'Paper',
  relationshipProjection: 'CITES',
  iterations: 20, dampingFactor: 0.85,
  writeProperty: "pagerank"
})
YIELD nodes, iterations, createMillis, computeMillis, writeMillis, dampingFactor,
writeProperty
```

Table 47. Results

Name	ArticleRank
Paper 0	0.34616300000000005
Paper 1	0.319422
Paper 4	0.213733
Paper 2	0.21089400000000003
Paper 3	0.18026850000000003
Paper 5	0.15000000000000002
Paper 6	0.15000000000000002

Paper 0 is the most important paper, but it's only the 2nd most cited paper - Paper 1 has more citations. However, Paper 1 cites Paper 0, which lets us see that it's not only the number of incoming links that is important, but also the importance of the papers behind those links. Papers 5 and 6 are not cited by any other papers, so their score doesn't increase above the initial score of 1 - dampingFactor.

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.articleRank.write(graphNameOrConfig: String|Map, configuration: Map)
YIELD nodes, iterations, createMillis, computeMillis, writeMillis, dampingFactor,
writeProperty
```

Table 48. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes.
relationship	string	null	yes	The relationship type to load from the graph. If null, load all relationships.
iterations	int	20	yes	How many iterations of Page Rank to run.
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.
dampingFactor	float	0.85	yes	The damping factor of the Page Rank calculation.
relationshipWeightProperty	string	null	yes	The name of the relationship property that represents weight. If null, treats the graph as unweighted. Must be numeric.
defaultValue	float	0.0	yes	The default value of the weight in case it is missing or invalid.
graph	string	'huge'	yes	Use 'huge' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node statement and relationship statement.

Table 49. Results

Name	Type	Description
nodes	int	The number of nodes considered.
iterations	int	The number of iterations run.
dampingFactor	float	The damping factor used.

Name	Type	Description
writeProperty	string	The property name written back to.
createMillis	int	Milliseconds for loading data.
computeMillis	int	Milliseconds for running the algorithm.
writeMillis	int	Milliseconds for writing result data back.

The following will run the algorithm and stream results:

```
CALL gds.alpha.articleRank.stream(graphNameOrConfig: String|Map, configuration: Map)
YIELD node, score
```

Table 50. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes.
relationship	string	null	yes	The relationship type to load from the graph. If null, load all nodes.
iterations	int	20	yes	Specify how many iterations of Page Rank to run.
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'.
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.
dampingFactor	float	0.85	yes	The damping factor of the Page Rank calculation.
relationshipWeightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
defaultValue	float	0.0	yes	The default value of the weight in case it is missing or invalid.

Name	Type	Default	Optional	Description
graph	string	'huge'	yes	Use 'huge' when describing the subset of the graph with label and relationship-type parameter. Use 'cypher' for describing the subset with cypher node statement and relationship statement.

Table 51. Results

Name	Type	Description
node	long	Node ID
score	float	Page Rank weight

## Graph type support

The ArticleRank algorithm supports the following graph types:

- directed, unweighted
- undirected, unweighted

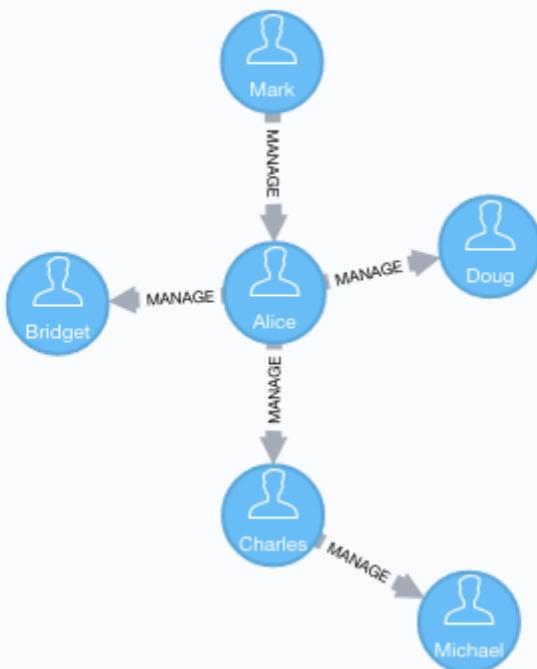
## Betweenness Centrality

***This section describes the Betweenness Centrality algorithm in the Neo4j Graph Data Science library.***

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.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

In the following example, Alice is the main connection in the graph:



If Alice is removed, all connections in the graph would be cut off. This makes Alice important, because she ensures that no nodes are isolated.

This section includes:

- History and explanation
- Use-cases - when to use the Betweenness Centrality algorithm
- Constraints - when not to use the Betweenness Centrality algorithm
- Syntax
- Betweenness Centrality example
- Sampled Betweenness Centrality
  - Sampled Betweenness Centrality example
- Cypher projection

## History and explanation

The Betweenness Centrality algorithm calculates the shortest (weighted) path between every pair of nodes in a connected graph, using the breadth-first search algorithm. Each node receives a score, based on the number of these shortest paths that pass through the node. Nodes that most frequently lie on these shortest paths will have a higher betweenness centrality score.

The algorithm was given its first formal definition by Linton Freeman, in his 1971 paper "A Set of Measures of Centrality Based on Betweenness". It was considered to be one of the "three distinct intuitive conceptions of centrality".

## Use-cases - when to use the Betweenness Centrality algorithm

- Betweenness centrality is used to research the network flow in a package delivery process, or telecommunications network. These networks are characterized by traffic that has a known target and takes the shortest path possible. This, and other scenarios, are described by Stephen P. Borgatti in "[Centrality and network flow](#)".
- Betweenness centrality is used to identify influencers in legitimate, or criminal, organizations. Studies show that influencers in organizations are not necessarily in management positions, but instead can be found in brokerage positions of the organizational network. Removal of such influencers could seriously destabilize the organization. More detail can be found in "[Brokerage qualifications in ringing operations](#)", by Carlo Morselli and Julie Roy.
- Betweenness centrality can be used to help microbloggers spread their reach on Twitter, with a recommendation engine that targets influencers that they should interact with in the future. This approach is described in "[Making Recommendations in a Microblog to Improve the Impact of a Focal User](#)".

## Constraints - when not to use the Betweenness Centrality algorithm

- Betweenness centrality makes the assumption that all communication between nodes happens along the shortest path and with the same frequency, which isn't the case in real life. Therefore, it doesn't give us a perfect view of the most influential nodes in a graph, but rather a good representation. Newman explains this in more detail on page 186 of [Networks: An Introduction](#).
- For large graphs, exact centrality computation isn't practical. The fastest known algorithm for exactly computing betweenness of all the nodes requires at least  $O(nm)$  time for unweighted graphs, where  $n$  is the number of nodes and  $m$  is the number of relationships. Instead, we can use an approximation algorithm that works with a subset of nodes.

## Syntax

The following will run the Betweenness Centrality algorithm and write back results:

```
CALL gds.alpha.betweenness.write(configuration: Map)
YIELD nodes, minCentrality, maxCentrality, sumCentrality, createMillis, computeMillis,
writeMillis
```

Table 52. Parameters

Name	Type	Default	Optional	Description
configuration	map	{}	no	Configuration parameters.

Table 53. Configuration

Name	Type	Default	Optional	Description
writeProperty	string	'centrality'	yes	The property name written back to.

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'. This is dependent on the Neo4j edition; for more information, see <a href="#">CPU</a> .
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.

Table 54. Results

Name	Type	Description
nodes	int	The number of nodes considered
minCentrality	int	The minimum centrality value
maxCentrality	int	The maximum centrality value
sumCentrality	int	The sum of all centrality values
createMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the Betweenness Centrality algorithm and stream results:

```
CALL gds.alpha.betweenness.stream(configuration: Map)
YIELD nodeId, centrality
```

Table 55. 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'. This is dependent on the Neo4j edition; for more information, see <a href="#">CPU</a> .

Name	Type	Default	Optional	Description
readConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

Table 56. Results

Name	Type	Description
node	long	Node ID
centrality	float	Betweenness centrality weight

The following will run the Sampled Betweenness Centrality algorithm and write back results:

```
CALL gds.alpha.betweenness.sampled.write(configuration: Map)
YIELD nodes, minCentrality, maxCentrality, sumCentrality, createMillis, computeMillis,
writeMillis
```

Table 57. Parameters

Name	Type	Default	Optional	Description
writeProperty	string	'centrality'	yes	The property name written back to.
strategy	string	'random'	yes	The node selection strategy.
probability	float	log10(N) / e^2	yes	The probability a node is selected. Values between 0 and 1. If 1, selects all nodes and works like original Brandes algorithm.
maxDepth	int	Integer.MAX_X	yes	The depth of the shortest paths traversal.
concurrency	int	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	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.

Table 58. Results

Name	Type	Description
nodes	int	The number of nodes considered

Name	Type	Description
minCentrality	int	The minimum centrality value
maxCentrality	int	The maximum centrality value
sumCentrality	int	The sum of all centrality values
createMillis	int	Milliseconds for loading data
computeMillis	int	Milliseconds for running the algorithm
writeMillis	int	Milliseconds for writing result data back

The following will run the Sampled Betweenness Centrality algorithm and stream results:

```
CALL gds.alpha.betweenness.sampled.stream(configuration: Map)
YIELD nodeId, centrality
```

Table 59. Parameters

Name	Type	Default	Optional	Description
label	string	null	yes	The label to load from the graph. If null, load all nodes.
relationship	string	null	yes	The relationship type to load from the graph. If null, load all relationships.
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'.
readConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
strategy	string	'random'	yes	The node selection strategy.
probability	float	$\log_{10}(N) / e^2$	yes	The probability a node is selected. Values between 0 and 1. If 1, selects all nodes and works like original Brandes algorithm.
maxDepth	int	Integer.MAX_X	yes	The depth of the shortest paths traversal.

Table 60. Results

Name	Type	Description
node	long	Node ID.
centrality	float	Betweenness centrality weight.

## Betweenness Centrality algorithm sample

People with high betweenness tend to be the innovators and brokers in social networks. They combine different perspectives, transfer ideas between groups, and get power from their ability to make introductions and pull strings.

The following will create a sample graph:

```
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)-[:MANAGE]->(bridget),
       (alice)-[:MANAGE]->(charles),
       (alice)-[:MANAGE]->(doug),
       (mark)-[:MANAGE]->(alice),
       (charles)-[:MANAGE]->(michael)
```

The following will run the algorithm and stream results:

```
CALL gds.alpha.betweenness.stream({
    nodeProjection: 'User',
    relationshipProjection: 'MANAGE'
})
YIELD nodeId, centrality
RETURN gds.util.asNode(nodeId).name AS user, centrality
ORDER BY centrality DESC
```

Table 61. Results

user	centrality
Alice	4
Charles	2
Bridget	0
Doug	0
Mark	0
Michael	0

We can see that Alice is the main broker in this network, and Charles is a minor broker. The others don't have any influence, because all the shortest paths between pairs of people go via Alice or Charles.

The following will run the algorithm and write back results:

```
CALL gds.alpha.betweenness.write({  
    nodeProjection: 'User',  
    relationshipProjection: 'MANAGE',  
    writeProperty: 'centrality'  
}) YIELD nodes, minCentrality, maxCentrality, sumCentrality
```

Table 62. Results

nodes	minCentrality	maxCentrality	sumCentrality
6	0.0	4.0	6.0

## Approximation of Betweenness Centrality

As mentioned above, calculating the exact betweenness centrality on large graphs can be very time consuming. Therefore, you might choose to use an approximation algorithm that will run much quicker, and still provide useful information.

### Sampled Betweenness Centrality algorithm

The RA-Brandes algorithm is the best known algorithm for calculating an approximate score for betweenness centrality. Rather than calculating the shortest path between every pair of nodes, the RA-Brandes algorithm considers only a subset of nodes. Two common strategies for selecting the subset of nodes are:

#### random

Nodes are selected uniformly, at random, with defined probability of selection. The default probability is `log10(N) / e^2`. If the probability is 1, then the algorithm works the same way as the normal Betweenness Centrality algorithm, where all nodes are loaded.

#### degree

First, the mean degree of the nodes is calculated, and then only the nodes whose degree is higher than the mean are visited (i.e. only dense nodes are visited).

As a further optimisation, you can choose to limit the depth used by the shortest path algorithm. This can be controlled by the `maxDepth` parameter.

The following will run the algorithm and stream results:

```
CALL gds.alpha.betweenness.sampled.stream({  
    nodeProjection: 'User',  
    relationshipProjection: 'MANAGE',  
    strategy: 'random',  
    probability: 1.0,  
    maxDepth: 1  
}) YIELD nodeId, centrality  
RETURN gds.util.asNode(nodeId).name AS user, centrality  
ORDER BY centrality DESC
```

Table 63. Results

user	centrality
Alice	3
Charles	1
Bridget	0
Doug	0
Mark	0
Michael	0

Alice is still the main broker in the network, and Charles is a minor broker, although their centrality score has reduced as the algorithm only considers relationships at a depth of 1. The others don't have any influence, because all the shortest paths between pairs of people go via Alice or Charles.

The following will run the algorithm and write back results:

```
CALL gds.alpha.betweenness.sampled.write({
  nodeProjection: 'User',
  relationshipProjection: 'MANAGE',
  strategy: 'random',
  probability: 1.0,
  writeProperty: 'centrality',
  maxDepth: 1
})
YIELD nodes, minCentrality, maxCentrality, sumCentrality
```

Table 64. Results

nodes	minCentrality	maxCentrality	sumCentrality
6	0.0	3.0	4.0

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

```
CALL gds.alpha.betweenness.write({
  nodeQuery: 'MATCH (p:User) RETURN id(p) AS id',
  relationshipQuery: 'MATCH (p1:User)-[:MANAGE]->(p2:User) RETURN id(p1) AS source,
  id(p2) AS target'
})
YIELD nodes, minCentrality, maxCentrality, sumCentrality
```

## Closeness Centrality

*This section describes the Closeness Centrality algorithm in the Neo4j Graph Data Science library.*

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Closeness Centrality algorithm](#)
- [Constraints - when not to use the Closeness Centrality algorithm](#)
- [Syntax](#)
- [Closeness Centrality algorithm sample](#)
- [Cypher projection](#)

### History and explanation

For each node, 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 is calculated using the following formula:

```
raw_closeness_centrality(node) = 1 / sum(distance from node 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** is as follows:

```
normalized_closeness_centrality(node) = (number of nodes - 1) / sum(distance from node to all other nodes)
```

### 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
```

Table 65. 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 66. Results

Name	Type	Description
nodes	int	The number of nodes considered.
createMillis	int	Milliseconds for loading data.

Name	Type	Description
computeMillis	int	Milliseconds for running the algorithm.
writeMillis	int	Milliseconds for writing result data back.
writeProperty	string	The property name written back to.

The following will run the algorithm and stream results:

```
CALL gds.alpha.closeness.stream(configuration: Map)
YIELD nodeId, centrality
```

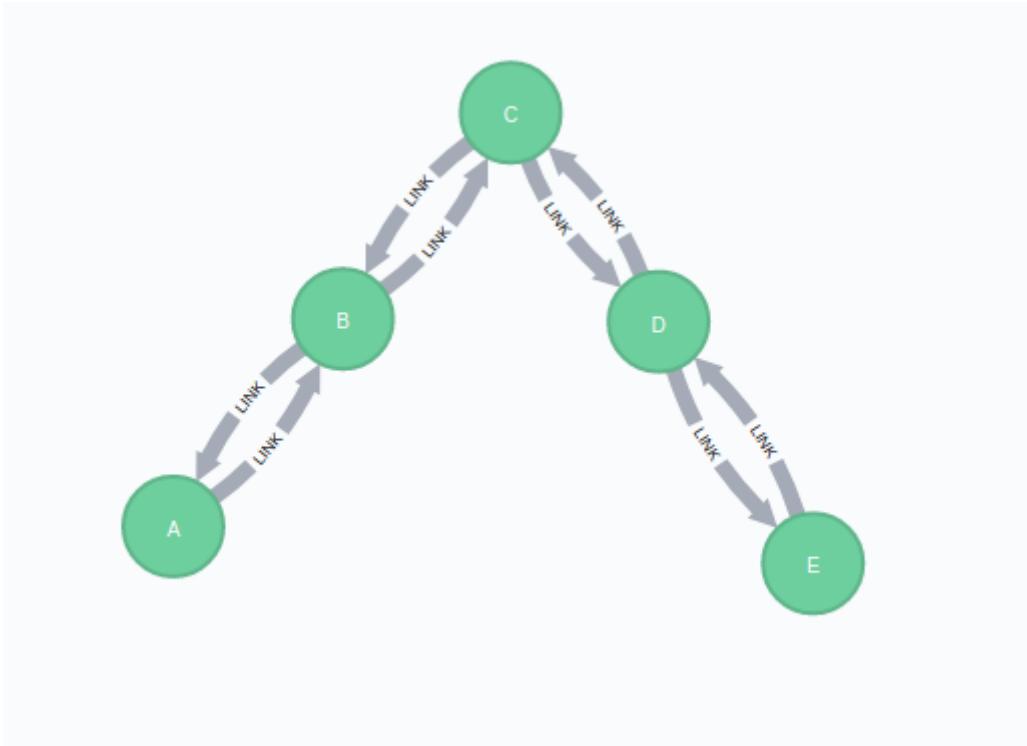
Table 67. Parameters

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 68. 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 69. Results

Name	Centrality weight
C	0.6666666666666666

Name	Centrality weight
B	0.5714285714285714
D	0.5714285714285714
A	0.4
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 70. Results

nodes	writeProperty
5	"centrality"

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) 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 71. 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	// farness 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

## Degree Centrality

**This section describes the Degree Centrality algorithm in the Neo4j Graph Data Science library.**

Degree centrality measures the number of incoming and outgoing relationships from a node.

The Degree Centrality algorithm can help us find popular nodes in a graph.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Degree Centrality algorithm](#)
- [Degree Centrality algorithm sample](#)
- [Weighted Degree Centrality algorithm sample](#)
- [Cypher projection](#)
- [Syntax](#)

### History and explanation

Degree Centrality was proposed by Linton C. Freeman in his 1979 paper [Centrality in Social Networks Conceptual Clarification](#). While the Degree Centrality algorithm can be used to find the popularity of individual nodes, it is often used as part of a global analysis where we calculate the minimum degree, maximum degree, mean degree, and standard deviation across the whole graph.

### Use-cases - when to use the Degree Centrality algorithm

- Degree centrality is an important component of any attempt to determine the most important people on 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.

- 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

The following will run the algorithm and write back results:

```
CALL gds.alpha.degree.write(configuration: Map)
YIELD nodes, createMillis, computeMillis, writeMillis, writeProperty
```

Table 72. Configuration

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.
relationshipWeightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
defaultValue	float	0.0	yes	The default value of the weight in case it is missing or invalid.

Table 73. Results

Name	Type	Description
nodes	int	The number of nodes considered.
writeProperty	string	The property name written back to.
createMillis	int	Milliseconds for loading data.
computeMillis	int	Milliseconds for running the algorithm.
writeMillis	int	Milliseconds for writing result data back.

The following will run the algorithm and stream results:

```
CALL gds.alpha.degree.stream(configuration: Map)
YIELD node, score
```

Table 74. Configuration

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'.
readConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
weightProperty	string	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
defaultValue	float	0.0	yes	The default value of the weight in case it is missing or invalid.

Table 75. Results

Name	Type	Description
nodeId	long	Node ID
score	float	Degree Centrality score

## Degree Centrality algorithm sample

This sample will explain the Degree Centrality algorithm, using a simple graph:

Create sample graph

```
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]->(doug),
       (alice)-[:FOLLOWS]->(brIDGET),
       (alice)-[:FOLLOWS]->(charles),
       (mark)-[:FOLLOWS]->(doug),
       (mark)-[:FOLLOWS]->(michael),
       (brIDGET)-[:FOLLOWS]->(doug),
       (charles)-[:FOLLOWS]->(doug),
       (michael)-[:FOLLOWS]->(doug)
```

*The following will run the algorithm and stream results, showing which users have the most followers:*

```
CALL gds.alpha.degree.stream({  
    nodeProjection: 'User',  
    relationshipProjection: {  
        FOLLOWS: {  
            type: 'FOLLOWS',  
            projection: 'REVERSE'  
        }  
    }  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score AS followers  
ORDER BY followers DESC
```

*The following will run the algorithm and store results, showing which users have the most followers:*

```
CALL gds.alpha.degree.write({  
    nodeProjection: 'User',  
    relationshipProjection: {  
        FOLLOWS: {  
            type: 'FOLLOWS',  
            projection: 'REVERSE'  
        }  
    },  
    writeProperty: 'followers'  
})
```

*Table 76. Results*

Name	Followers
Doug	5.0
Bridget	1.0
Charles	1.0
Michael	1.0
Mark	0.0
Alice	0.0

*The following will run the algorithm and stream results, showing which users follow the most other users:*

```
CALL gds.alpha.degree.stream({  
    nodeProjection: 'User',  
    relationshipProjection: 'FOLLOWS'  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score AS followers  
ORDER BY followers DESC
```

*The following will run the algorithm and store results, showing which users follow the most other users:*

```
CALL gds.alpha.degree.write({  
    nodeProjection: 'User',  
    relationshipProjection: 'FOLLOWS',  
    writeProperty: 'followers'  
})
```

*Table 77. Results*

Name	Following
Alice	3.0
Mark	2.0
Bridget	1.0
Charles	1.0
Michael	1.0
Doug	0.0

We can see that Doug is the most popular user in our imaginary Twitter graph, with 5 followers - all other users follow him, but he doesn't follow anybody back. In the real Twitter network celebrities have very high follower counts but tend to follow very few back people. We could therefore consider Doug a celebrity!

### **Weighted Degree Centrality algorithm sample**

This sample will explain the weighted Degree Centrality algorithm, using a simple graph:

*The following will create a sample graph:*

```
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)
```

This algorithm is a variant of the Degree Centrality algorithm, that measures the sum of the weights of incoming and outgoing relationships.

*The following will run the algorithm and stream results, showing which users have the most weighted followers:*

```
CALL gds.alpha.degree.stream({  
    nodeProjection: 'User',  
    relationshipProjection: {  
        FOLLOWS: {  
            type: 'FOLLOWS',  
            orientation: 'REVERSE',  
            properties: 'score'  
        }  
    },  
    relationshipWeightProperty: 'score'  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score AS weightedFollowers  
ORDER BY weightedFollowers DESC
```

*The following will run the algorithm and store results, showing which users have the most weighted followers:*

```
CALL gds.alpha.degree.write({  
    nodeProjection: 'User',  
    relationshipProjection: {  
        FOLLOWS: {  
            type: 'FOLLOWS',  
            orientation: 'REVERSE',  
            properties: 'score'  
        }  
    },  
    relationshipWeightProperty: 'score',  
    writeProperty: 'weightedFollowers'  
})  
YIELD nodes, writeProperty
```

*Table 78. Results*

Name	weightedFollowers
Doug	7.5
Charles	5.0
Michael	4.5
Bridget	2.0
Alice	0.0
Mark	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.

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

```
CALL gds.alpha.degree.write({  
    nodeQuery: 'MATCH (u:User) RETURN id(u) AS id',  
    relationshipQuery: 'MATCH (u1:User)<-[>:FOLLOWS]-(u2:User) RETURN id(u1) AS source,  
    id(u2) AS target',  
    writeProperty: 'followers'  
})
```

Note, that if we want to find the number of users that a user is following rather than their number of followers, we need to handle that in our Cypher query.

*The following will run the algorithm and store the results, calculating the number of users that a user follows:*

```
CALL gds.alpha.degree.write({  
    nodeQuery: 'MATCH (u:User) RETURN id(u) AS id',  
    relationshipQuery: 'MATCH (u1:User)-[>:FOLLOWS]-(u2:User) RETURN id(u1) AS source,  
    id(u2) AS target',  
    writeProperty: 'followers'  
})
```

## Eigenvector Centrality

**This section describes the Eigenvector Centrality algorithm in the Neo4j Graph Data Science library.**

Eigenvector Centrality is an algorithm that measures the **transitive** influence or connectivity of nodes.

Relationships to high-scoring nodes contribute more to the score of a node than connections to low-scoring nodes. A high score means that a node is connected to other nodes that have high scores.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Eigenvector Centrality algorithm](#)
- [Syntax](#)
- [Eigenvector Centrality algorithm sample](#)
- [Cypher projection](#)

- Graph type support

## History and explanation

Eigenvector Centrality was proposed by Phillip Bonacich, in his 1986 paper [Power and Centrality: A Family of Measures](#). It was the first of the centrality measures that considered the transitive importance of a node in a graph, rather than only considering its direct importance.

## Use-cases - when to use the Eigenvector Centrality algorithm

Eigenvector Centrality can be used in many of the [same use cases as the Page Rank algorithm](#).

## Syntax

*The following will run the algorithm and write back results:*

```
CALL gds.alpha.eigenvector.write(configuration: Map)
YIELD nodes, iterations, dampingFactor, writeProperty, createMillis, computeMillis,
writeMillis
```

Table 79. Configuration

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.
relationshipWeightProperty	string	null	yes	The name of the relationship property that contains weight. If null, treats the graph as unweighted. Must be numeric.
normalization	string	null	yes	The type of normalization to apply to the results. Valid values are <a href="#">max</a> , <a href="#">l1norm</a> , <a href="#">l2norm</a> .
maxIterations	int	20	yes	The maximum number of iterations of EigenvectorCentrality to run.
sourceNodes	list<node>	empty list	yes	A list of nodes to start the computation from.

Table 80. Results

Name	Type	Description
nodes	int	The number of nodes considered.

Name	Type	Description
iterations	int	The number of iterations run.
dampingFactor	float	The damping factor used.
writeProperty	string	The property name written back to.
createMillis	int	Milliseconds for loading data.
computeMillis	int	Milliseconds for running the algorithm.
writeMillis	int	Milliseconds for writing result data back.

The following will run the algorithm and stream results:

```
CALL gds.alpha.eigenvector.stream(configuration: Map)
YIELD node, score
```

Table 81. Configuration

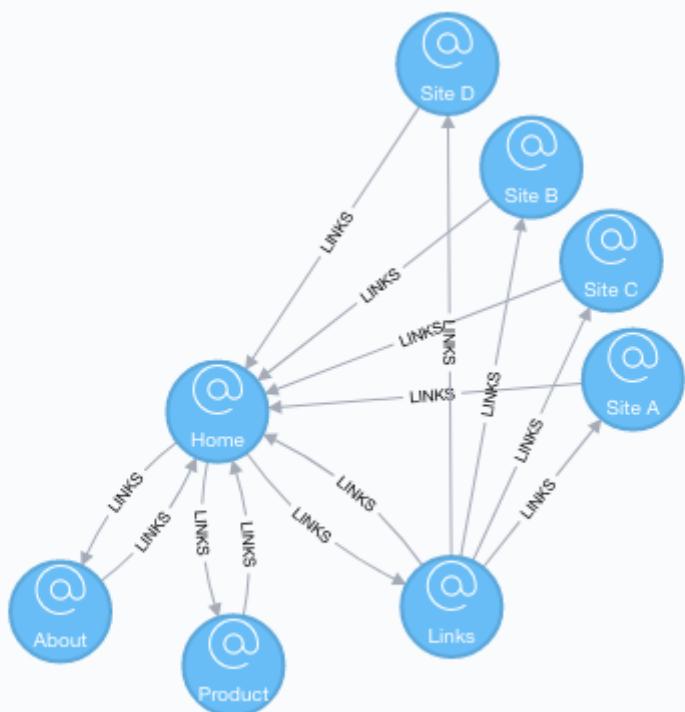
Name	Type	Default	Optional	Description
concurrency	int	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'.
readConcurrency	int	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
relationshipPropertyWeightProperty	string	null	yes	The name of the relationship property that contains weight. If null, treats the graph as unweighted. Must be numeric.
normalization	string	null	yes	The type of normalization to apply to the results. Valid values are <code>max</code> , <code>l1norm</code> , <code>l2norm</code> .
maxIterations	int	20	yes	The maximum number of iterations of EigenvectorCentrality to run.
sourceNodes	list<node>	empty list	yes	A list of nodes to start the computation from.

Table 82. Results

Name	Type	Description
nodeId	long	Node ID
score	float	Eigenvector Centrality weight

## Eigenvector Centrality algorithm sample

This sample will explain the Eigenvector Centrality algorithm, using a simple graph:



The following will create a sample graph:

```
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)
```

The following will run the algorithm and stream results:

```
CALL gds.alpha.eigenvector.stream({
  nodeProjection: 'Page',
  relationshipProjection: 'LINKS'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS page, score
ORDER BY score DESC
```

Table 83. Results

page	score
"Home"	31.458663403987885
"About"	14.403928011655807
"Product"	14.403928011655807
"Links"	14.403928011655807
"Site A"	6.572431668639183
"Site B"	6.572431668639183
"Site C"	6.572431668639183
"Site D"	6.572431668639183

As we might expect, the *Home* page has the highest Eigenvector Centrality because it has incoming links from all other pages. We can also see that it's not only the number of incoming links that is important, but also the importance of the pages behind those links.

The following will run the algorithm and write back results:

```
CALL gds.alpha.eigenvector.write({
  nodeProjection: 'Page',
  relationshipProjection: 'LINKS',
  writeProperty: 'eigenvector'
})
YIELD nodes, iterations, dampingFactor, writeProperty
```

Table 84. Results

nodes	iterations	dampingFactor	writeProperty
0	20	1.0	"eigenvector"

By default, the scores returned by the Eigenvector Centrality are not normalized. We can specify a normalization using the [normalization](#) parameter. The algorithm supports the following options:

- `max` - divide all scores by the maximum score
- `l1norm` - normalize scores so that they sum up to 1

- **$L_2$ norm** - divide each score by the square root of the squared sum of all scores

The following will run the algorithm and stream results using **max** normalization:

```
CALL gds.alpha.eigenvector.stream({
  nodeProjection: 'Page',
  relationshipProjection: 'LINKS',
  normalization: 'max'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS page, score
ORDER BY score DESC
```

Table 85. Results

page	score
"Home"	1.0
"About"	0.4578684042192931
"Product"	0.4578684042192931
"Links"	0.4578684042192931
"Site A"	0.20892278811203477
"Site B"	0.20892278811203477
"Site C"	0.20892278811203477
"Site D"	0.20892278811203477

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

Use **nodeQuery** and **relationshipQuery** in the config:

```
CALL gds.alpha.eigenvector.write({
  nodeQuery: 'MATCH (p:Page) RETURN id(p) AS id',
  relationshipQuery: 'MATCH (p1:Page)-[:LINKS]->(p2:Page) RETURN id(p1) AS source,
  id(p2) AS target',
  maxIterations: 5
})
YIELD nodes, iterations, dampingFactor, writeProperty
```

## Graph type support

The Eigenvector Centrality algorithm supports the following graph types:

- directed, unweighted

- [] directed, weighted
- undirected, unweighted
- [] undirected, weighted

## Community detection algorithms

*This chapter provides explanations and examples for each of the community detection algorithms in the Neo4j Graph Data Science library.*

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](#)

### Louvain

*This section describes the Louvain algorithm in the Neo4j Graph Data Science library.*

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Streaming](#)
  - [Writing](#)
  - [Mutate](#)
  - [Weights](#)

- Seeding
- Multi-Level
- Memory Estimation
- Stats

## 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

### Write mode

*Run Louvain in write mode on a graph stored in the catalog.*

```
CALL gds.louvain.write(
    graphName: String,
    configuration: Map
)
YIELD
    // general write return columns
    nodePropertiesWritten: Integer,
    communityCount: Integer,
    modularity: Float
```

Table 86. Parameters

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

Name	Type	Default	Optional	Description
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Table 87. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 community ID is written to.

Run Louvain in write mode on an anonymous graph.

```
CALL gds.louvain.write(configuration: Map)
YIELD
    // general write return columns
    nodePropertiesWritten: Integer,
    communityCount: Integer,
    modularity: Float
```

Table 88. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 89. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, String[] or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.

Name	Type	Default	Optional	Description
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipsQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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 community ID is written to.

Table 90. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipsWeightProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
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	no	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).

Table 91. 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.
ranLevels	Integer	The number of supersteps the algorithm actually ran.
modularity	Float	The final modularity score.
modularities	Integer[]	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.

#### Mutate mode

*Run Louvain in mutate mode on a graph stored in the catalog.*

```
CALL gds.louvain.mutate(
    graphName: String,
    configuration: Map
)
YIELD
    // general mutate return columns
    nodePropertiesWritten: Integer,
    communityCount: Integer,
    modularity: Float
```

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.

### Stream mode

*Run Louvain in stream mode on a graph stored in the catalog.*

```
CALL gds.louvain.stream(
    graphName: String,
    configuration: Map
)
YIELD
    nodeId: Integer,
    communityId: Integer,
    intermediateCommunityIds: Integer[]
```

*Table 92. 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 93. General configuration for algorithm execution on a named graph.*

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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'.

Name	Type	Default	Optional	Description
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 community ID is written to.

Run Louvain in stream mode on an anonymous graph.

```
CALL gds.louvain.stream(configuration: Map)
YIELD
  nodeId: Integer,
  communityId: Integer,
  intermediateCommunityIds: Integer[]
```

Table 94. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 95. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, String[] or Map	null	yes	The relationship properties to project during anonymous graph creation.

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 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 community ID is written to.

Table 96. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
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.
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 includeIntermediateCommunities flag.

Table 97. Results

Name	Type	Description
nodeId	Integer	Node ID.

Name	Type	Description
communityId	Integer	The community ID of the final level.
intermediateCommunityIds	Integer[]	Community IDs for each level. <b>Null</b> if <code>includeIntermediateCommunities</code> is set to false.

### Stats mode

Run Louvain in stats mode on a named graph.

```
CALL gds.louvain.stats(
    graphName: String,
    configuration: Map
)
YIELD
    computeMillis: Integer,
    postProcessingMillis: Integer,
    communityCount: Integer,
    modularity: Float,
```

Table 98. 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.

Run Louvain in stats mode on an anonymous graph.

```
CALL gds.louvain.stats(configuration: Map)
YIELD
    createMillis: Integer,
    computeMillis: Integer,
    postProcessingMillis: Integer,
    communityCount: Integer,
    modularity: Float,
```

Table 99. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

The configuration is the same as for the `write` mode.

Table 100. 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	Integer[]	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.

### Estimate mode

The following will estimate the memory requirements for running the algorithm. The `mode` can be substituted with the available modes (`stream`, `write` and `stats`).

*Run Louvain in estimate mode on a named graph.*

```
CALL gds.louvain.<mode>.estimate(
    graphName: String,
    configuration: Map
)
```

Table 101. 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.

*Run Louvain in estimate mode on an anonymous graph.*

```
CALL gds.louvain.<mode>.estimate(configuration: Map)
```

Table 102. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

For memory estimation on a named graph the configuration for creating that graph is used. For the anonymous graph, configuration is used as for [Native projection](#) or [Cypher projection](#). The graph estimation on an anonymous graph is based on the configuration pertaining to anonymous creation or so-called fictive estimation controlled by the options in the table below.

*Table 103. Configuration*

Name	Type	Default	Optional	Description
nodeCount	Integer	-1	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	-1	yes	The number of relationships in a fictive graph.

Setting the `nodeCount` and `relationshipCount` parameters results in fictive graph estimation which allows a memory estimation without loading the graph. Additionally algorithm specific parameters can also be provided as config which influence the estimation of memory usage specific to the algorithm.

*Table 104. Memory estimation results.*

Name	Type	Description
nodeCount	Integer	Node count of the graph used in the estimation.
relationshipCount	Integer	Relationship count of the graph used in the estimation.
requiredMemory	Integer	Human readable version for required memory.
bytesMin	Integer	Minimum number of bytes to be consumed.
bytesMax	Integer	Maximum number of bytes to be consumed.
heapPercentageMin	Float	The minimum percentage of the configured max heap (-Xmx) to be consumed.
heapPercentageMax	Float	The maximum percentage of the configured max heap (-Xmx) to be consumed.
treeView	Map	Human readable version of memory estimation.
mapView	Map	Detailed information on memory consumption.

## Examples

Consider the graph created by the following Cypher statement:

```

CREATE (nAlice:User {name: 'Alice', seed: 42})
CREATE (nBridget:User {name: 'Bridget', seed: 42})
CREATE (nCharles:User {name: 'Charles', seed: 42})
CREATE (nDoug:User {name: 'Doug'})
CREATE (nMark:User {name: 'Mark'})
CREATE (nMichael:User {name: 'Michael'})

CREATE (nAlice)-[:LINK {weight: 1}]->(nBridget)
CREATE (nAlice)-[:LINK {weight: 1}]->(nCharles)
CREATE (nCharles)-[:LINK {weight: 1}]->(nBridget)

CREATE (nAlice)-[:LINK {weight: 5}]->(nDoug)

CREATE (nMark)-[:LINK {weight: 1}]->(nDoug)
CREATE (nMark)-[:LINK {weight: 1}]->(nMichael)
CREATE (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 standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could 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.

### Streaming results

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 105. 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`.

### Writing results

To instead write the community results back to the graph in Neo4j, use the following query. For each node a property is written that holds the assigned community.

The following run the algorithm, and write back results:

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

Table 106. 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.

### 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 107. 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.

### Running on weighted graphs

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 108. 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.

### Running with seed communities

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 109. Results

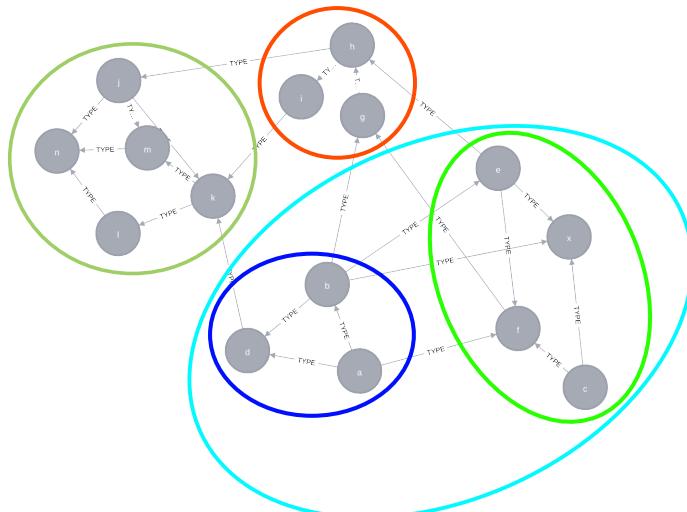
<b>name</b>	<b>communityId</b>	<b>intermediateCommunityIds</b>
"Alice"	42	null
"Bridget"	42	null
"Charles"	42	null
"Doug"	47	null
"Mark"	47	null
"Michael"	47	null

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.

### Streaming 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 110. Results

<b>name</b>	<b>communityId</b>	<b>intermediateCommunityIds</b>
"a"	14	[3, 14]
"b"	14	[3, 14]
"c"	14	[14, 14]
"d"	14	[3, 14]
"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.

## 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 111. Results

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
6	14	5353	580120	"[5353 Bytes ... 566 KiB]"

## 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 112. Results

communityCount
2

## Label Propagation

**This section describes the Label Propagation algorithm in the Neo4j Graph Data Science library.**

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.

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.

This section includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Unweighted](#)
  - [Weighted](#)
  - [Seeded](#)

## Introduction

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.

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

### Write mode

Run Label Propagation in write mode on a graph stored in the catalog.

```
CALL gds.labelPropagation.write(  
    graphName: String,  
    configuration: Map  
)  
YIELD  
    // general write return columns  
    ranIterations: Integer,  
    didConverge: Boolean
```

Table 113. 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 114. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 community ID is written to.

Run Label Propagation in write mode on an anonymous graph.

```
CALL gds.labelPropagation.write(configuration: Map)  
YIELD  
    // general write return columns  
    ranIterations: Integer,  
    didConverge: Boolean
```

Table 115. 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 116. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 community ID is written to.

Table 117. 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 the node property that represents weight.
relationshipWeightProperty	String	null	yes	The name of the relationship property that represents weight.
seedProperty	String	n/a	yes	Used to define initial set of labels (must be a number).
consecutiveIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 118. 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.
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.

## Mutate mode

Run Label Propagation in mutate mode on a graph stored in the catalog.

```
CALL gds.labelPropagation.mutate(
    graphName: String,
    configuration: Map
)
YIELD
    // general mutate return columns
    ranIterations: Integer,
    didConverge: Boolean
```

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.

The following will run the algorithm and store the results in `myGraph`:

```
CALL gds.labelPropagation.mutate('myGraph', { mutateProperty: 'communityId' })
```

## Stream mode

The following will run the algorithm and stream back results:

Run Label Propagation in stream mode on a graph stored in the catalog.

```
CALL gds.labelPropagation.stream(
    graphName: String,
    configuration: Map
)
YIELD
    nodeId: Integer,
    communityId: Integer
```

Table 119. 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 120. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 community ID is written to.

Run Label Propagation in stream mode on an anonymous graph.

```
CALL gds.labelPropagation.stream(configuration: Map)
YIELD
    nodeId: Integer,
    communityId: Integer
```

Table 121. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 122. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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 community ID is written to.

Table 123. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
nodeWeightProperty	String	null	yes	The property name of node that contain weight. Must be numeric.
relationshipWeightProperty	String	null	yes	The property name of relationship that contain weight. Must be numeric.

Name	Type	Default	Optional	Description
seedProper ty	String	n/a	yes	Used to define initial set of labels (must be a number).
consecutiv eIds	Boolean	false	yes	Flag to decide whether component identifiers are mapped into a consecutive id space (requires additional memory).

Table 124. Results

Name	Type	Description
nodeId	Integer	Node ID
communityId	Integer	Community ID

### Stats mode

Run Label Propagation in stats mode on a named graph.

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

Table 125. Parameters

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

Run Label Propagation in stats mode on an anonymous graph.

```
CALL gds.labelPropagation.stats(configuration: Map)
YIELD
    ranIterations: Integer,
    didConverge: Boolean,
    createMillis: Integer,
    computeMillis: Integer
```

Table 126. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

The configuration is the same as for the `write` mode.

Table 127. 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.

### Estimate mode

The following will estimate the memory requirements for running the algorithm. The `mode` can be substituted with the available modes (`stream`, `write` and `stats`).

*Run Label Propagation in estimate mode on a named graph.*

```
CALL gds.labelPropagation.<mode>.estimate(
    graphName: String,
    configuration: Map
)
```

Table 128. Parameters

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

Name	Type	Default	Optional	Description
configuration	Map	{}	yes	Configuration for algorithm-specifics and/or graph filtering.

Run Label Propagation in estimate mode on an anonymous graph.

```
CALL gds.labelPropagation.<mode>.estimate(configuration: Map)
```

Table 129. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

For memory estimation on a named graph the configuration for creating that graph is used. For the anonymous graph, configuration is used as for [Native projection](#) or [Cypher projection](#). The graph estimation on an anonymous graph is based on the configuration pertaining to anonymous creation or so-called fictive estimation controlled by the options in the table below.

Table 130. Configuration

Name	Type	Default	Optional	Description
nodeCount	Integer	-1	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	-1	yes	The number of relationships in a fictive graph.

Setting the `nodeCount` and `relationshipCount` parameters results in fictive graph estimation which allows a memory estimation without loading the graph. Additionally algorithm specific parameters can also be provided as config which influence the estimation of memory usage specific to the algorithm.

Table 131. Memory estimation results.

Name	Type	Description
nodeCount	Integer	Node count of the graph used in the estimation.
relationshipCount	Integer	Relationship count of the graph used in the estimation.
requiredMemory	Integer	Human readable version for required memory.
bytesMin	Integer	Minimum number of bytes to be consumed.
bytesMax	Integer	Maximum number of bytes to be consumed.
heapPercentageMin	Float	The minimum percentage of the configured max heap (-Xmx) to be consumed.
heapPercentageMax	Float	The maximum percentage of the configured max heap (-Xmx) to be consumed.

Name	Type	Description
treeView	Map	Human readable version of memory estimation.
mapView	Map	Detailed information on memory consumption.

## Examples

Consider the graph created by the following Cypher statement:

```

CREATE (alice:User {name: 'Alice', seed_label: 52})
CREATE (bridget:User {name: 'Bridget', seed_label: 21})
CREATE (charles:User {name: 'Charles', seed_label: 43})
CREATE (doug:User {name: 'Doug', seed_label: 21})
CREATE (mark:User {name: 'Mark', seed_label: 19})
CREATE (michael:User {name: 'Michael', seed_label: 52})

CREATE (alice)-[:FOLLOW {weight: 1}]->(bridget)
CREATE (alice)-[:FOLLOW {weight: 10}]->(charles)
CREATE (mark)-[:FOLLOW {weight: 1}]->(doug)
CREATE (brIDGET)-[:FOLLOW {weight: 1}]->(michael)
CREATE (doug)-[:FOLLOW {weight: 1}]->(mark)
CREATE (michael)-[:FOLLOW {weight: 1}]->(alice)
CREATE (alice)-[:FOLLOW {weight: 1}]->(michael)
CREATE (brIDGET)-[:FOLLOW {weight: 1}]->(alice)
CREATE (michael)-[:FOLLOW {weight: 1}]->(brIDGET)
CREATE (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 standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could also be used.

*The following statement will create the graph and store it in the graph catalog.*

```

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.

### Unweighted

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 132. Results

Name	Community
"Alice"	1
"Bridget"	1
"Michael"	1
"Charles"	4
"Doug"	4
"Mark"	4

The following will run the algorithm and write back results:

```
CALL gds.labelPropagation.write('myGraph', { writeProperty: 'community' })
YIELD ranIterations, communityCount
```

Table 133. Results

ranIterations	communityCount
3	2

Our algorithm found two communities, with 3 members each.

It appears that Michael, Bridget, and Alice belong together, as do Doug and Mark. Only Charles doesn't strongly fit into either side, but ends up with Doug and Mark.

### Weighted

The Label-Propagation algorithm can also take node and relationship weights into account. 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 134. Results

Name	Community
"Bridget"	2
"Michael"	2
"Alice"	4
"Charles"	4
"Doug"	4
"Mark"	4

Using the weighted relationships, **Alice** and **Charles** are now in the same community as there is a strong link between them.

The following will run the algorithm on a weighted graph and write back results:

```
CALL gds.labelPropagation.write('myGraph', {
  writeProperty: 'community',
  relationshipWeightProperty: 'weight'
})
YIELD ranIterations, communityCount
```

Table 135. Results

ranIterations	communityCount
4	2

As we can see, the weighted example takes 4 iterations to converge, instead of 3 for the unweighted case.

Additionally 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.

### Seeded

At the beginning of the algorithm, 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, a new unique label is assigned 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. Note that the `consecutiveIds` configuration option cannot be used in combination with seeding 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 136. 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 user **Mark** and **21** from **Doug**.

*The following will run the algorithm and write back results:*

```
CALL gds.labelPropagation.write('myGraph', {
  writeProperty: 'community',
  seedProperty: 'seed_label'
})
YIELD ranIterations, communityCount
```

*Table 137. Results*

ranIterations	communityCount
3	2

## Weakly Connected Components

***This section describes the Weakly Connected Components (WCC) algorithm in the Neo4j Graph Data Science library.***

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Unweighted](#)
  - [Weighted](#)
  - [Seeded components](#)
  - [Memory Estimation](#)
  - [Stats](#)

## 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.

WCC has previously been known as Union Find or Connected Components in this User Guide.

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

### Write mode

*Run WCC in write mode on a graph stored in the catalog.*

```
CALL gds.wcc.write(
    graphName: String,
    configuration: Map
)
YIELD
    // general write return columns
    componentCount: Integer,
    componentDistribution: Map
```

*Table 138. 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 139. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 component ID is written to.

Run WCC in write mode on an anonymous graph.

```
CALL gds.wcc.write(configuration: Map)
YIELD
    // general write return columns
    componentCount: Integer,
    componentDistribution: Map
```

Table 140. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 141. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, String[] or Map	null	yes	The relationship projection used for anonymous graph creation a Native projection.

Name	Type	Default	Optional	Description
nodeQuery	String	null	yes	The Cypher query used to select the nodes for anonymous graph creation via a Cypher projection.
relationshipsQuery	String	null	yes	The Cypher query used to select the relationships for anonymous graph creation via a Cypher projection.
nodeProperties	String, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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 component ID is written to.

Table 142. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	The relationship property that contains the weight. If null, the graph is treated as unweighted. Must be numeric.
defaultValue	Float	null	yes	The default value of the relationship weight in case it is missing or invalid.
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 143. 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 component count and distribution statistics.
nodePropertiesWritten	Integer	The number of node properties written.
relationshipPropertiesWritten	Integer	The number of relationship properties written.
componentCount	Integer	The number of computed components.
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.

## Mutate mode

Run WCC in mutate mode on a graph stored in the catalog.

```
CALL gds.wcc.mutate(
    graphName: String,
    configuration: Map
)
YIELD
    // general mutate return columns
    componentCount: Integer,
    componentDistribution: 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.

The following will run the algorithm and store the results in `myGraph`:

```
CALL gds.wcc.mutate('myGraph', { mutateProperty: 'componentId' })
```

## Stream mode

Run WCC in stream mode on a graph stored in the catalog.

```
CALL gds.wcc.stream(  
    graphName: String,  
    configuration: Map  
)  
YIELD  
    nodeId: Integer,  
    componentId: Integer
```

Table 144. 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 145. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	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 component ID is written to.

Run WCC in stream mode on an anonymous graph.

```
CALL gds.wcc.stream(configuration: Map)  
YIELD  
    nodeId: Integer,  
    componentId: Integer
```

Table 146. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 147. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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 component ID is written to.

Table 148. Algorithm specific configuration

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	The relationship property that contains the weight. If null, the graph is treated as unweighted. Must be numeric.
defaultValue	Float	null	yes	The default value of the relationship weight in case it is missing or invalid.
seedProperty	String	n/a	yes	Used to set the initial component for a node. The property value needs to be a number.

Name	Type	Default	Optional	Description
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 149. Results

Name	Type	Description
nodeId	Integer	The Neo4j node ID.
componentId	Integer	The component ID.

### Stats mode

Run WCC in stats mode on a named graph.

```
CALL gds.wcc.stats(
    graphName: String,
    configuration: Map
)
YIELD
    computeMillis: Integer,
    postProcessingMillis: Integer,
    componentCount: Integer,
    componentDistribution: Map,
```

Table 150. 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.

Run WCC in stats mode on an anonymous graph.

```
CALL gds.wcc.stats(configuration: Map)
YIELD
    createMillis: Integer,
    computeMillis: Integer,
    postProcessingMillis: Integer,
    componentCount: Integer,
    componentDistribution: Map,
```

Table 151. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

The configuration is the same as for the `write` mode.

Table 152. Results

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.
componentCount	Integer	The number of computed components.
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.

### Estimate mode

The following will estimate the memory requirements for running the algorithm. The `mode` can be substituted with the available modes (`stream`, `write` and `stats`).

Run WCC in estimate mode on a named graph.

```
CALL gds.wcc.<mode>.estimate(
    graphName: String,
    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.

Run WCC in estimate mode on an anonymous graph.

```
CALL gds.wcc.<mode>.estimate(configuration: Map)
```

Table 154. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

For memory estimation on a named graph the configuration for creating that graph is used. For the anonymous graph, configuration is used as for [Native projection](#) or [Cypher projection](#). The graph estimation on an anonymous graph is based on the configuration pertaining to anonymous creation or so-called fictive estimation controlled by the options in the table below.

Table 155. Configuration

Name	Type	Default	Optional	Description
nodeCount	Integer	-1	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	-1	yes	The number of relationships in a fictive graph.

Setting the `nodeCount` and `relationshipCount` parameters results in fictive graph estimation which allows a memory estimation without loading the graph. Additionally algorithm specific parameters can also be provided as config which influence the estimation of memory usage specific to the algorithm.

Table 156. Memory estimation results.

Name	Type	Description
nodeCount	Integer	Node count of the graph used in the estimation.
relationshipCount	Integer	Relationship count of the graph used in the estimation.
requiredMemory	Integer	Human readable version for required memory.
bytesMin	Integer	Minimum number of bytes to be consumed.
bytesMax	Integer	Maximum number of bytes to be consumed.
heapPercentageMin	Float	The minimum percentage of the configured max heap (-Xmx) to be consumed.
heapPercentageMax	Float	The maximum percentage of the configured max heap (-Xmx) to be consumed.
treeView	Map	Human readable version of memory estimation.
mapView	Map	Detailed information on memory consumption.

## Examples

Consider the graph created by the following Cypher statement:

```

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)-[:LINK {weight: 0.5}]->(nBridget)
CREATE (nAlice)-[:LINK {weight: 4}]->(nCharles)
CREATE (nMark)-[:LINK {weight: 1.1}]->(nDoug)
CREATE (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 following examples we will demonstrate using the Weakly Connected Components algorithm on this graph.

We can load this graph into the in-memory catalog.



In the examples below we will use named graphs and standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could also be used.

*The following statement will create the graph and store it in the graph catalog.*

```

CALL gds.graph.create(
  'myGraph',
  'User',
  'LINK',
  {
    relationshipProperties: 'weight'
  }
)

```

In the following examples we will demonstrate using the WCC algorithm on this graph.

### Unweighted

*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 157. Results*

name	componentId
"Alice"	0
"Bridget"	0

<b>name</b>	<b>componentId</b>
"Charles"	0
"Doug"	3
"Mark"	3
"Michael"	3

To instead write the component ID to a node property in the Neo4j graph, use this query:

*The following will run the algorithm and write back results:*

```
CALL gds.wcc.write('myGraph', { writeProperty: 'componentId' })
YIELD nodePropertiesWritten, componentCount;
```

*Table 158. Results*

<b>nodePropertiesWritten</b>	<b>componentCount</b>
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 a weight property, a default weight is used. The default is zero, and can be configured to another value using the `defaultValue` configuration parameter.

*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 159. Results*

<b>Name</b>	<b>ComponentId</b>
"Alice"	0
"Charles"	0
"Bridget"	1
"Doug"	3
"Mark"	3

Name	ComponentId
"Michael"	3

The following will run the algorithm and write back results:

```
CALL gds.wcc.write('myGraph', {
    writeProperty: 'componentId',
    relationshipWeightProperty: 'weight',
    threshold: 1.0
})
YIELD nodePropertiesWritten, componentCount;
```

Table 160. Results

nodePropertiesWritten	componentCount
6	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.

### 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 to run WCC without seeds.

To show this in practice, we will run the algorithm, then add another node to our graph, then run the algorithm again with the `seedProperty` configuration parameter. We will use the weighted variant of WCC.

The following will run the algorithm and write back results:

```
CALL gds.wcc.write('myGraph', {  
    writeProperty: 'componentId',  
    relationshipWeightProperty: 'weight',  
    threshold: 1.0  
})  
YIELD nodePropertiesWritten, componentCount;
```

Table 161. Results

nodePropertiesWritten	componentCount
6	3

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'})
```

Note, that we can not use our already created graph as it does not contain the component id. We will therefore create a second 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'  
    }  
)
```

The following will run the algorithm and stream results:

```
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 162. Results

name	componentId
"Alice"	0
"Charles"	0

<b>name</b>	<b>componentId</b>
"Bridget"	1
"Mats"	1
"Doug"	3
"Mark"	3
"Michael"	3

The following will run the algorithm and write back results:

```
CALL gds.wcc.write('myGraph-seeded', {
    seedProperty: 'componentId',
    writeProperty: 'componentId',
    relationshipWeightProperty: 'weight',
    threshold: 1.0
})
YIELD nodePropertiesWritten, componentCount;
```

Table 163. Results

<b>nodePropertiesWritten</b>	<b>componentCount</b>
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.

## Memory Estimation

The following will estimate the memory requirements for running the algorithm:

```
CALL gds.wcc.write.estimate('myGraph', {
    writeProperty: 'communityId'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Table 164. Results

<b>nodeCount</b>	<b>relationshipCoun t</b>	<b>bytesMin</b>	<b>bytesMax</b>	<b>requiredMemory</b>
6	4	176	176	"176 Bytes"

## Stats

The following will run the algorithm and returns the result in form of statistical and measurement values

```
CALL gds.wcc.stats('myGraph')
YIELD componentCount
```

Table 165. Results

componentCount
2

## Triangle Count

**This section describes the Triangle Count algorithm in the Neo4j Graph Data Science library.**

This topic includes:

- [Introduction](#)
- [Syntax](#)
  - [Stream](#)
  - [Stats](#)
  - [Mutate](#)
  - [Write](#)
  - [Triangles Listing](#)
- [Examples](#)
  - [Memory Estimation](#)
  - [Stream](#)
  - [Stats](#)
  - [Mutate](#)
  - [Write](#)
  - [Maximum degree](#)
  - [Triangles Listing](#)

### 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. The named graph variant of the syntax is described. 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.

### Stream

*Run Triangle Count in stream mode on a named graph:*

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

Table 166. 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 167. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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 168. 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 <b>-1</b> .

Table 169. Results

Name	Type	Description
nodeId	Integer	Node ID.
triangleCount	Integer	Number of triangles the node is part of. Is <b>-1</b> if the node has been excluded from computation using the <code>maxDegree</code> configuration parameter.

## Stats

Run Triangle Count in stream mode on a named graph:

```
CALL gds.triangleCount.stats(
    graphName: String,
    configuration: Map
)
YIELD
    triangleCount: Integer,
    nodeCount: Integer,
    createMillis: Integer,
    computeMillis: Integer,
    configuration: Map
```

Table 170. 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 171. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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 172. 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 <b>-1</b> .

Table 173. Results

Name	Type	Description
triangleCount	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.
configuration	Map	The configuration used for running the algorithm.

## Mutate

Run Triangle Count in mutate mode on a named graph:

```
CALL gds.triangleCount.mutate(
    graphName: String,
    configuration: Map
)
YIELD
    triangleCount: Integer,
    nodeCount: Integer,
    nodePropertiesWritten: Integer,
    createMillis: Integer,
    computeMillis: Integer,
    mutateMillis: Integer,
    configuration: Map
```

Table 174. 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 175. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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.
mutateProperty	String	n/a	no	The node property in the GDS graph to which the triangle count is written.

Table 176. 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 177. Results

Name	Type	Description
triangleCount	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.
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.

## Write

Run Triangle Count in write mode on a named graph:

```
CALL gds.triangleCount.write(  
    graphName: String,  
    configuration: Map  
)  
YIELD  
    triangleCount: Integer,  
    nodeCount: Integer,  
    nodePropertiesWritten: Integer,  
    createMillis: Integer,  
    computeMillis: Integer,  
    writeMillis: Integer,  
    configuration: Map
```

Table 178. 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 179. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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.
mutateProperty	String	n/a	no	The node property in the GDS graph to which the triangle count is written.

Table 180. 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 181. Results

Name	Type	Description
triangleCount	Integer	Total number of triangles in the graph.

Name	Type	Description
nodeCount	Integer	Number of nodes in the graph.
nodePropertiesWritten	Integer	Number of properties written to Neo4j.
createMillis	Integer	Milliseconds for creating the graph.
computeMillis	Integer	Milliseconds for running the algorithm.
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
    triangleCount: Integer,
    nodeCount: Integer,
    nodePropertiesWritten: Integer,
    createMillis: Integer,
    computeMillis: Integer,
    writeMillis: Integer,
    configuration: Map
```

*Table 182. General configuration for algorithm execution on an anonymous graph.*

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	<code>null</code>	yes	The node projection used for anonymous graph creation via a Native projection.

Name	Type	Default	Optional	Description
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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.
writeProperty	String	n/a	no	The node property in the Neo4j database to which the triangle count is written.

Table 183. 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, [specified 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 this tier of algorithm, see [here](#).

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 184. 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 185. General configuration for algorithm execution on a named graph.

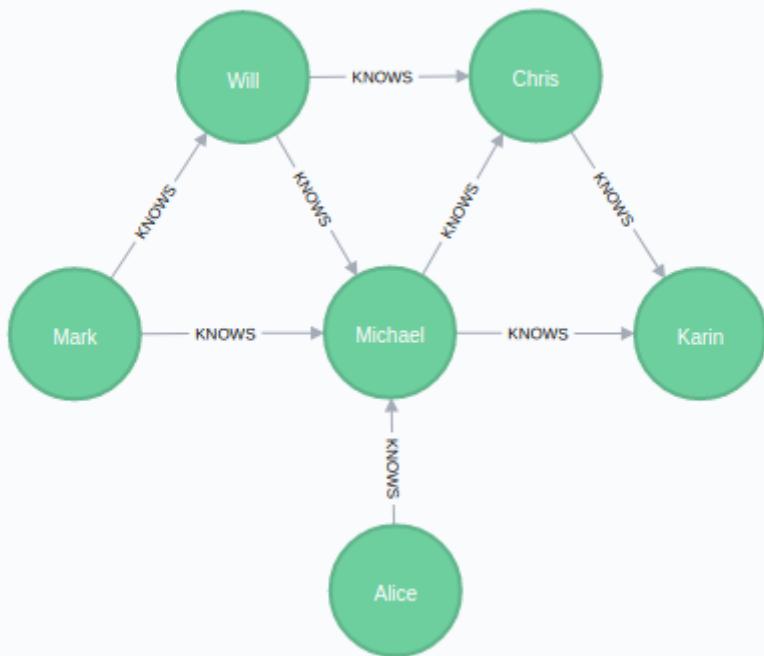
Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	String[]	[ '*' ]	yes	Filter the named graph using the given relationship types.
concurrency	Integer	4	yes	The number of concurrent threads used for running the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

Table 186. 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 executing 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 187. Results

nodeCount	relationshipCoun t	bytesMin	bytesMax	requiredMemory
6	16	144	144	"144 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 example,

we can order the results to find the nodes with the highest triangle count.

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 188. 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. The summary result contains the global triangle count, which is the total number of triangles in the entire graph. 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 189. 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 example, using the triangle count to compute the [local clustering coefficient](#).

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 190. 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 in the catalog](#).

## 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 191. 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 in the catalog](#).

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 192. 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 this tier of algorithm, see [here](#).

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 193. 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.

## Local Clustering Coefficient

*This section describes the Local Clustering Coefficient algorithm in the Neo4j Graph Data Science library.*

This topic includes:

- [Introduction](#)
- [Syntax](#)
  - [Stream](#)
  - [Stats](#)
  - [Mutate](#)
  - [Write](#)
  - [Anonymous Graphs](#)
- [Examples](#)
  - [Memory Estimation](#)
  - [Stream](#)
  - [Stats](#)
  - [Mutate](#)
  - [Write](#)
  - [Pre-computed counts](#)

## 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. To learn more about general syntax variants, see [Syntax overview](#).

### Stream

*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 194. 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 195. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	['*']	yes	Filter the named graph using the given node labels.

Name	Type	Default	Optional	Description
relationshipTypes	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 196. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 197. Results

Name	Type	Description
nodeId	Integer	Node ID.
localClusteringCoefficient	Double	The local clustering coefficient for the node.

## Stats

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,
    configuration: Map
```

Table 198. 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 199. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.

Name	Type	Default	Optional	Description
relationshipsTypes	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 200. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 201. 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.
configuration	Map	The configuration used for running the algorithm.

## Mutate

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,
    mutateMillis: Integer,
    configuration: Map
```

Table 202. 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 203. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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.
mutateProperty	String	n/a	no	The node property in the GDS graph to which the local clustering coefficient is written.

Table 204. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 205. 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.
mutateMillis	Integer	Milliseconds for adding properties to the in-memory graph.
configuration	Map	The configuration used for running the algorithm.

## Write

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,  
    writeMillis: Integer,  
    configuration: Map
```

Table 206. 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 207. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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.
writeProperty	String	n/a	no	The node property in the Neo4j database to which the local clustering coefficient is written.

Table 208. Algorithm specific configuration

Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

Table 209. 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.
computeMillis	Integer	Milliseconds for running the algorithm.
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 210. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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.
writeProperty	String	n/a	no	The node property in the Neo4j database to which the local clustering coefficient is written.

Table 211. Algorithm specific configuration

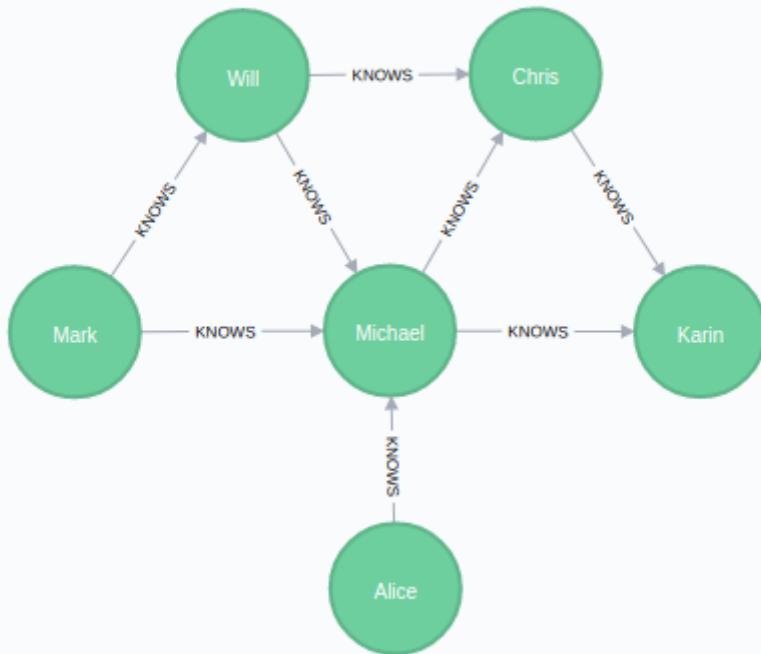
Name	Type	Default	Optional	Description
triangleCountProperty	String	n/a	Yes	Node property that contains pre-computed triangle count.

The results are the same as for running write mode with a named graph, [specified above](#).

## Examples

In this section we will show examples of executing 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 212. Results

nodeCount	relationshipCoun t	bytesMin	bytesMax	requiredMemory
6	16	288	288	"288 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 example, we can order the results to find the nodes with the highest local clustering coefficient.

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 213. 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. The summary result contains the average clustering coefficient of the graph, which is the normalised sum over all local clustering coefficients. 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 214. 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 example, using the triangle count to compute the `local clustering coefficient`.

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 215. 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 in the catalog](#).

## 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 216. 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 217. Results

name	localClusteringCoefficient
"Karin"	1.0

<b>name</b>	<b>localClusteringCoefficient</b>
"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](#).

## K-1 Coloring

***This section describes the K-1 Coloring algorithm in the Neo4j Graph Data Science library.***

This algorithm is in the beta tier. For more information on this tier of algorithm, see [here](#).

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)

### 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

### Write mode

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 nodes, colorCount, ranIterations, didConverge, configuration, createMillis,
computeMillis, writeMillis
```

Table 218. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	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 219. 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.

Name	Type	Default	Optional	Description
writeProperty	String	n/a	no	The node property this procedure writes the color to.

Table 220. 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.
write	Boolean	Specifies if the result was written back as a node property.
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.

## Mutate mode

Run K1-Coloring in mutate mode on a graph stored in the catalog.

```
CALL gds.beta.k1coloring.mutate(graphName: String, configuration: Map)
YIELD nodes, colorCount, ranIterations, didConverge, configuration, createMillis,
computeMillis, mutateMillis
```

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.

The following will run the algorithm and store the results in `myGraph`:

```
CALL gds.beta.k1coloring.mutate('myGraph', { mutateProperty: 'color' })
```

## Stream mode

The following describes the API for running the algorithm and stream results:

```
CALL gds.beta.k1coloring.stream(graphName: String, {  
    // additional configuration  
})  
YIELD nodeId, color
```

Table 221. Parameters

Name	Type	Default	Optional	Description
graphName	String	null	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	{}	yes	Additional configuration, see below.

Table 222. 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 223. Results

Name	Type	Description
nodeId	Integer	The ID of the Node
color	Integer	The color of the Node

## Stats mode

The following describes the API for running the algorithm and returning the computation statistics:

```
CALL gds.beta.k1coloring.stats(  
    graphName: String,  
    configuration: Map  
)  
YIELD  
    nodes,  
    colorCount,  
    ranIterations,  
    didConverge,  
    configuration,  
    createMillis,  
    computeMillis
```

Table 224. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	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 225. 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 226. 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.
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 standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could 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 227. 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 228. 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 229. 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 230. Results

nodeCount	colorCount	ranIterations	didConverge
4	3	1	true

## Modularity Optimization

This section describes the Modularity Optimization algorithm in the Neo4j Graph Data Science library.

This algorithm is in the beta tier. For more information on this tier of algorithm, see [here](#).

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)

### 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

### Write mode

The following describes the API for running the algorithm and writing results back to Neo4j:

```
CALL gds.beta.modularityOptimization.write(graphName: String|Map, configuration: Map)
YIELD nodes, ranIterations, didConverge, modularity, createMillis, computeMillis,
writeMillis, configuration
```

Table 231. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	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 232. 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.

Name	Type	Default	Optional	Description
relationshipsProperties	Map or List	null	yes	The relationship properties to load during implicit graph loading.

Table 233. Algorithm specific configuration

Name	Type	Default	Optional	Description
weightProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
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).

Table 234. 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.

Name	Type	Description
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.

### Mutate mode

Run Modularity Optimization in mutate mode on a graph stored in the catalog.

```
CALL gds.beta.modularityOptimization.mutate(graphName: String|Map, configuration: Map)
YIELD nodes, ranIterations, didConverge, modularity, createMillis, computeMillis,
mutateMillis, configuration
```

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.

The following will run the algorithm and store the results in `myGraph`:

```
CALL gds.beta.modularityOptimization.mutate('myGraph', { mutateProperty: 'modularity' })
```

### Stream mode

The following will run the algorithm and stream back results:

```
CALL gds.modularityOptimization.stream(graphNameOrConfig: String|Map, configuration: Map)
YIELD nodeId, communityId
```

Table 235. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	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 <code>graphNameOrConfig</code> is a Map.

Table 236. 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 237. Algorithm specific configuration

Name	Type	Default	Optional	Description
maxIterations	Integer	10	yes	The maximum number of iterations to run.
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.
relationshipWeightProperty	String	null	yes	The property name of relationship that contain weight. Must be numeric.
seedProperty	String	n/a	yes	Used to define initial set of labels (must be a number).

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).

Table 238. Results

Name	Type	Description
nodeId	Integer	Node ID
communityId	Integer	Community ID

## 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 standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could 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 239. Results

<b>name</b>	<b>communityId</b>
"Alice"	4
"Bridget"	1
"Charles"	1
"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 240. Results

<b>nodes</b>	<b>communityCount</b>	<b>ranIterations</b>	<b>didConverge</b>
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 241. 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.

## Strongly Connected Components

*This section describes the Strongly Connected Components algorithm in the Neo4j Graph Data Science library.*

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Strongly Connected Components algorithm](#)
- [Syntax](#)
- [Strongly Connected Components algorithm example](#)
- [Cypher projection](#)

### 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 242. 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 243. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.

Name	Type	Description
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.
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 244. Parameters

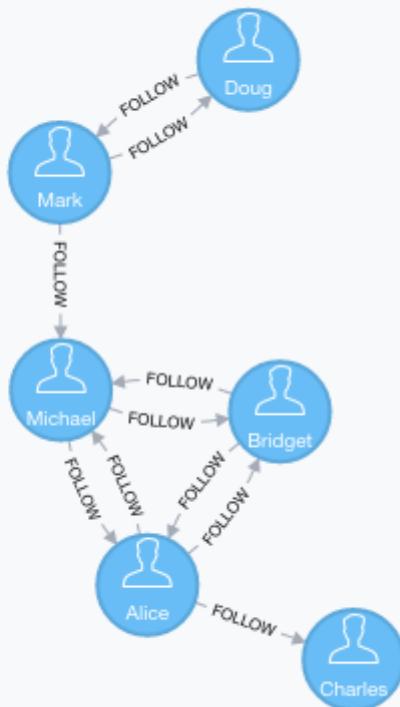
Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

Table 245. Results

Name	Type	Description
nodeId	Integer	Node ID.

Name	Type	Description
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 246. 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 247. 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 248. Results

Component	ComponentSize
0	3

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) 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 249. Results

Name	Component
"Doug"	3
"Mark"	3
"Charles"	2
"Alice"	0
"Bridget"	0
"Michael"	0

## Similarity algorithms

This chapter provides explanations and examples for each of the similarity algorithms in the Neo4j Graph Data Science library.

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](#)
- Alpha
  - [Approximate Nearest Neighbors](#)
  - [Cosine Similarity](#)
  - [Euclidean Similarity](#)
  - [Jaccard Similarity](#)

- Overlap Similarity
- Pearson Similarity

## Node Similarity

*This section describes the Node Similarity algorithm in the Neo4j Graph Data Science library. The algorithm is based on the Jaccard Similarity score.*

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)
  - [Streaming results](#)
  - [Writing results](#)
  - [Limiting results](#)
    - [topK and bottomK](#)
    - [topN and bottomN](#)
  - [Degree cutoff and similarity cutoff](#)
  - [Memory Estimation](#)
  - [Stats](#)

### 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.

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 all nodes from the first node set with each other based on their relationships to nodes in the second set. 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

### Write mode

Run *Node Similarity* in write mode on a graph stored in the catalog.

```
CALL gds.nodeSimilarity.write(  
    graphName: String,  
    configuration: Map  
)  
YIELD  
    // general write return columns  
    nodesCompared: Integer,  
    relationshipsWritten: Integer,  
    writeRelationshipType: String,  
    writeProperty: String
```

Table 250. 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 251. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	WRITE mode only: The number of concurrent threads used for writing the result.
writeProperty	String	n/a	no	WRITE mode only: The relationship property to which the similarity score is written to.

Run Node Similarity in write mode on an anonymous graph.

```
CALL gds.nodeSimilarity.write(configuration: Map)
YIELD
    // general write return columns
    nodesCompared: Integer,
    relationshipsWritten: Integer
```

Table 252. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 253. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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.

Name	Type	Default	Optional	Description
nodeProperties	String, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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 relationship property to which the similarity score is written to.

Table 254. 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 cannot 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
writeRelationshipType	String	SIMILAR	no	The relationship type used to represent a similarity score.

Table 255. 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.

## Mutate mode

Run Node Similarity in write mode on a graph stored in the catalog.

```
CALL gds.nodeSimilarity.mutate(
    graphName: String,
    configuration: Map
)
YIELD
    // general mutate return columns
    nodesCompared: Integer,
    relationshipsWritten: Integer
```

The configuration for the `mutate` mode is similar to the `write` mode. Instead of specifying a `writeRelationshipType` and `writeProperty`, we need to specify a `mutateRelationshipType` and `mutateProperty`. Also, specifying `writeConcurrency` is not possible in `mutate` mode.

The following will run the algorithm and store the results in `myGraph`:

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

### Stream mode

Run Node Similarity in stream mode on a graph stored in the catalog.

```
CALL gds.nodeSimilarity.stream(
    graphName: String,
    configuration: Map
) YIELD
    node1: Integer,
    node2: Integer,
    similarity: Float
```

Table 256. 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 257. General configuration for algorithm execution on a named graph.

Name	Type	Default	Optional	Description
nodeLabels	String[]	[ '*' ]	yes	Filter the named graph using the given node labels.
relationshipTypes	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	WRITE mode only: The number of concurrent threads used for writing the result.
writeProperty	String	n/a	no	WRITE mode only: The relationship property to which the similarity score is written to.

Run Node Similarity in stream mode on an anonymous graph.

```
CALL gds.nodeSimilarity.stream(configuration: Map)
YIELD
  node1: Integer,
  node2: Integer,
  similarity: Float
```

Table 258. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

Table 259. General configuration for algorithm execution on an anonymous graph.

Name	Type	Default	Optional	Description
nodeProjection	String, String[] or Map	null	yes	The node projection used for anonymous graph creation via a Native projection.
relationshipProjection	String, 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, String[] or Map	null	yes	The node properties to project during anonymous graph creation.
relationshipProperties	String, 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.

Name	Type	Default	Optional	Description
writeProperty	String	n/a	no	WRITE mode only: The relationship property to which the similarity score is written to.

Table 260. 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.

Table 261. Results

Name	Type	Description
node1	Integer	The Neo4j ID of the first node.
node2	Integer	The Neo4j ID of the second node.
similarity	Float	The similarity score for the two nodes.

## Stats mode

Run Node Similarity in stats mode on a named graph.

```
CALL gds.nodeSimilarity.stats(
    graphName: String,
    configuration: Map
)
YIELD
    computeMillis: Integer,
    nodesCompared: Integer,
```

Table 262. 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.

Run Node Similarity in stats mode on an anonymous graph.

```
CALL gds.nodeSimilarity.stats(configuration: Map)
YIELD
  createMillis: Integer,
  computeMillis: Integer,
  nodesCompared: Integer,
```

Table 263. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

The configuration is the same as for the `write` mode.

Table 264. Results

Name	Type	Description
createMillis	Integer	Milliseconds for loading data.
computeMillis	Integer	Milliseconds for running the algorithm.
postProcessingMillis	Integer	Milliseconds for computing percentiles.
nodesCompared	Integer	The number of nodes compared.
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.

### Estimate mode

The following will estimate the memory requirements for running the algorithm. The `mode` can be substituted with the available modes (`stream`, `write` and `stats`).

Run Node Similarity in estimate mode on a named graph.

```
CALL gds.nodeSimilarity.<mode>.estimate(  
    graphName: String,  
    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.

Run Node Similarity in estimate mode on an anonymous graph.

```
CALL gds.nodeSimilarity.<mode>.estimate(configuration: Map)
```

Table 266. Parameters

Name	Type	Default	Optional	Description
configuration	Map	null	no	Configuration for anonymous graph creation and algorithm-specifics.

For memory estimation on a named graph the configuration for creating that graph is used. For the anonymous graph, configuration is used as for [Native projection](#) or [Cypher projection](#). The graph estimation on an anonymous graph is based on the configuration pertaining to anonymous creation or so-called fictive estimation controlled by the options in the table below.

Table 267. Configuration

Name	Type	Default	Optional	Description
nodeCount	Integer	-1	yes	The number of nodes in a fictive graph.
relationshipCount	Integer	-1	yes	The number of relationships in a fictive graph.

Setting the `nodeCount` and `relationshipCount` parameters results in fictive graph estimation which allows a memory estimation without loading the graph. Additionally algorithm specific parameters can also be provided as config which influence the estimation of memory usage specific to the algorithm.

Table 268. Memory estimation results.

Name	Type	Description
nodeCount	Integer	Node count of the graph used in the estimation.
relationshipCount	Integer	Relationship count of the graph used in the estimation.

Name	Type	Description
requiredMemory	Integer	Human readable version for required memory.
bytesMin	Integer	Minimum number of bytes to be consumed.
bytesMax	Integer	Maximum number of bytes to be consumed.
heapPercentageMin	Float	The minimum percentage of the configured max heap (-Xmx) to be consumed.
heapPercentageMax	Float	The maximum percentage of the configured max heap (-Xmx) to be consumed.
treeView	Map	Human readable version of memory estimation.
mapView	Map	Detailed information on memory consumption.

## 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);

```

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 standard projections as the norm. However, [Cypher projection](#) and anonymous graphs could 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');
```

Similarly, loading a graph with Cypher also requires to load the whole pool of nodes of the bipartite graph as well as the relationships that link them.

The following statement will create a graph with Cypher and store it in the catalog.

```
CALL gds.graph.create.cypher(
    'myCypherGraph',
    'MATCH (n) WHERE n:Person OR n:Instrument RETURN id(n) AS id',
    'MATCH (p:Person)-[:LIKES]->(i:Instrument) RETURN id(p) AS source, id(i) AS target'
)
```

In the following examples we will demonstrate using the Node Similarity algorithm on this graph.

### Streaming results

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 269. 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
"Dave"	"Carol"	0.3333333333333333

Person1	Person2	similarity
10 rows		

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.

### Writing results

To instead write the similarity results back to the graph in Neo4j, use the following query. Each result is written as a new relationship between the compared nodes. The Jaccard similarity score is written as a property on the relationship.

*The following will run the algorithm, and write back results:*

```
CALL gds.nodeSimilarity.write('myGraph', {
    writeRelationshipType: 'SIMILAR',
    writeProperty: 'score'
})
YIELD nodesCompared, relationshipsWritten
```

Table 270. 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.

### Mutate

*The following will run the algorithm, and write back results:*

```
CALL gds.nodeSimilarity.mutate('myGraph', {
    mutateRelationshipType: 'SIMILAR',
    mutateProperty: 'score'
})
YIELD nodesCompared, relationshipsWritten
```

Table 271. Results

nodesCompared	relationshipsWritten
4	10

### Limiting 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 272. Result limits

	<b>total results</b>	<b>results per node</b>
<b>highest score</b>	topN	topK
<b>lowest score</b>	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 273. Results

<b>Person1</b>	<b>Person2</b>	<b>similarity</b>
"Alice"	"Dave"	1.0
"Bob"	"Alice"	0.6666666666666666
"Carol"	"Alice"	0.3333333333333333
"Dave"	"Alice"	1.0
4 rows		

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 274. Results

<b>Person1</b>	<b>Person2</b>	<b>similarity</b>
"Alice"	"Carol"	0.3333333333333333
"Bob"	"Alice"	0.6666666666666666

Person1	Person2	similarity
"Carol"	"Alice"	0.3333333333333333
"Dave"	"Carol"	0.3333333333333333
4 rows		

## 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 275. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Dave"	"Alice"	1.0
"Bob"	"Alice"	0.6666666666666666
3 rows		

## 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 276. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Dave"	"Alice"	1.0

Person1	Person2	similarity
2 rows		

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 277. Results

Person1	Person2	similarity
"Alice"	"Dave"	1.0
"Alice"	"Bob"	0.6666666666666666
"Bob"	"Dave"	0.6666666666666666
"Bob"	"Alice"	0.6666666666666666
"Dave"	"Alice"	1.0
"Dave"	"Bob"	0.6666666666666666
6 rows		

### 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 278. Results

nodeCount	relationshipCoun t	bytesMin	bytesMax	requiredMemory
9	9	2568	2568	"2568 Bytes"

## 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
```

Table 279. Results

nodesCompared
4

## Jaccard Similarity

**This section describes the Jaccard Similarity algorithm in the Neo4j Graph Data Science library.**

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 algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

A related procedure for computing Jaccard similarity is described in [Node Similarity](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Jaccard Similarity algorithm](#)
- [Jaccard Similarity algorithm function sample](#)

### History and explanation

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 280. 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:

$$\begin{aligned} J(A, B) &= |A \cap B| / |A| + |B| - |A \cap B| \\ J(A, B) &= 2 / 3 + 4 - 2 \\ &= 2 / 5 \\ &= 0.4 \end{aligned}$$

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 281. 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 282. Results

from	to	similarity
"Karin"	"Arya"	0.6666666666666666
"Karin"	"Michael"	0.25
"Karin"	"Praveena"	0.0
"Karin"	"Zhen"	0.0

## Cosine Similarity

**This section describes the Cosine Similarity algorithm in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Cosine Similarity algorithm](#)
- [Cosine Similarity function algorithm sample](#)
- [Cosine Similarity procedures algorithm sample](#)
- [Specifying source and target ids](#)
- [Skipping values](#)
- [Cypher projection](#)
- [Syntax](#)

## 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 run the algorithm on a graph in the catalog and write back results:*

```
CALL gds.alpha.similarity.cosine.write(graphName: String, configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean,
stdDev, p25, p50, p75, p90, p95, p99, p999, p100
```

Table 283. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph in the catalog.
configuration	Map	{}	yes	Algorithm-specific configuration.

*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 284. Parameters

Name	Type	Default	Optional	Description
configuration	Map	{}	no	Graph creation and algorithm-specific configuration.

Table 285. Configuration

Name	Type	Default	Optional	Description
data	String[]	null	no	A list of maps of the following structure: {item: nodeId, weights: [double, double, double]} 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 targets list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	gds.util.NaN	yes	Value to skip when executing similarity computation. A value of null 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 '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.
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.
writeProperty	String	score	yes	The property to use when storing results.

Name	Type	Default	Optional	Description
sourceIds	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	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 286. 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 on a graph in the catalog and stream results:

```
CALL gds.alpha.similarity.cosine.stream(graphName: String, configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```

Table 287. Parameters

Name	Type	Default	Optional	Description
graphName	String	n/a	no	The name of a graph in the catalog.
configuration	Map	{}	yes	Algorithm-specific configuration.

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 288. Parameters

Name	Type	Default	Optional	Description
configuration	Map	{}	no	Graph creation and algorithm-specific configuration.

Table 289. Configuration

Name	Type	Default	Optional	Description
data	String[]	null	no	A list of maps of the following structure: {item: nodeId, weights: [double, double, double]} 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 targets 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 null 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 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
graph	String	dense	yes	The graph type ('dense' or 'cypher').

Name	Type	Default	Optional	Description
sourceIds	Integer[]	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <b>data</b> parameter.
targetIds	Integer[]	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <b>data</b> parameter.

Table 290. 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 <b>targets</b> list of one node.
count2	Integer	The size of the <b>targets</b> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <b>targets</b> 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 291. Results

<b>similarity</b>
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:

$$\text{similarity}(A,B) = \frac{3 \cdot 10 + 8 \cdot 8 + 7 \cdot 6 + 5 \cdot 6 + 2 \cdot 4 + 9 \cdot 5}{\sqrt{3^2 + 8^2 + 7^2 + 5^2 + 2^2 + 9^2} \times \sqrt{10^2 + 8^2 + 6^2 + 6^2 + 4^2 + 5^2}} = \frac{219}{15.2315 \times 16.6433} = 0.8639$$

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'})-[:LIKES]->(cuisine)
MATCH (p2:Person {name: "Arya"})-[:LIKES]->(cuisine)
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.cosine(collect(likes1.score), collect(likes2.score)) AS
similarity
```

Table 292. 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 293. 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)
```

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({nodeProjection: '*', relationshipProjection:
'*', 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 294. 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
"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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 295. 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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 296. 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.

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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 297. Results

nodes	similarity Pairs	writeRelat ionshipTyp e	writePrope rty	min	max	mean	p95
5	5	"SIMILAR"	"score"	0.95422363 28125	1.00000381 46972656	0.98240203 85742187	1.00000381 46972656

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 298. Results

cuisine
Italian
Lebanese

## 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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 299. Results

from	to	similarity
Praveena	Karin	1.0
Arya	Michael	0.9788908326303921

## 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:

```
CREATE (french:Cuisine {name:'French'}) SET french.embedding = [0.71, 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, 0.26, 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, 0.81, 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, 0.21, 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.cosine.stream({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 loader 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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
```

## Pearson Similarity

**This section describes the Pearson Similarity algorithm in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Pearson Similarity algorithm](#)
- [Pearson Similarity algorithm function sample](#)
- [Pearson Similarity algorithm procedures sample](#)
- [Specifying source and target ids](#)
- [Skipping values](#)
- [Cypher projection](#)
- [Syntax](#)

### 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 300. 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 301. 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 302. 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 `((# items)^2 / 2) - # 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)
```

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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 303. Results

from	to	similarity
"Zhen"	"Praveena"	0.8865926413116155
"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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 304. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 305. 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.

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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 306. Results

nodes	similarity Pairs	writeRelationshipType	writeProperty	min	max	mean	p95
5	4	"SIMILAR"	"score"	0.81946182 25097656	0.88658905 02929688	0.85617160 79711914	0.88658905 02929688

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)-[r:RATED]->(movie)
WHERE not((p)-[:RATED]->(movie)) and r.score >= 5
RETURN movie.name AS movie
    
```

Table 307. Results

movie
Jerry Maguire

## 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 308. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 309. 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

from	to	similarity
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 loader 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 310. Results

nodes	similarity Pairs	writeRelat ionshipTyp e	writePrope rty	min	max	mean	p95
5	4	"SIMILAR"	"score"	0.81946182 25097656	0.88658905 02929688	0.85617160 79711914	0.88658905 02929688

## Syntax

The following will run the algorithm and write back results:

```
CALL gds.alpha.similarity.pearson.write(graphNameOrConfig: String|Map, configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean,
stdDev, p25, p50, p75, p90, p95, p99, p999, p100
```

Table 311. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	no	Either then name of a loaded graph or directly the config.
configuration	Map	n/a	yes	Additional configuration, if the first parameter was a graph name.

Table 312. 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 <code>0</code> , it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If <code>0</code> , 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	gds.util.NaN()	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 '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.

Name	Type	Default	Optional	Description
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	String[]	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <b>data</b> parameter.
targetIds	String[]	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <b>data</b> parameter.

Table 313. 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(graphNameOrConfig: String|Map, configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```

Table 314. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	no	Either then name of a loaded graph or directly the config.
configuration	Map	n/a	yes	Additional configuration, if the first parameter was a graph name.

Table 315. 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 <code>0</code> , it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If <code>0</code> , 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	gds.util.NaN()	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 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
sourceIds	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.

Name	Type	Default	Optional	Description
targetIds	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 316. 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.

## Euclidean Distance

**This section describes the Euclidean Distance algorithm in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Euclidean Distance algorithm](#)
- [Euclidean Distance algorithm function sample](#)
- [Euclidean Distance algorithm procedures sample](#)
- [Specifying source and target ids](#)
- [Skipping values](#)
- [Cypher projection](#)
- [Syntax](#)

### History and explanation

Euclidean distance is computed using the following formula:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_i - q_i)^2 + \dots + (p_n - q_n)^2}.$$

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 317. 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 318. 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 319. 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)
```

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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 320. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 321. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 322. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 323. 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.

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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 324. Results

nodes	similarity Pairs	writeRelat ionshipTyp e	writePrope rty	min	max	mean	p95
5	5	"SIMILAR"	"score"	3.0	4.35890197 75390625	3.51399841 30859374	4.35890197 75390625

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 325. Results

cuisine
Italian
Lebanese

## 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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 326. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 loader 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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(graphNameOrConfig: String|Map,
configuration: Map)
YIELD nodes, similarityPair, writeRelationshipType, writeProperty, min, max, mean,
stdDev, p25, p50, p75, p90, p95, p99, p999, p100
```

Table 327. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	no	Either then name of a loaded graph or directly the config.
configuration	Map	n/a	yes	Additional configuration, if the first parameter was a graph name.

Table 328. 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 <code>0</code> , it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If <code>0</code> , it will return as many as it finds.
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 <code>targets</code> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	gds.util.NaN()	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 '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.
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	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	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 329. 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.

Name	Type	Description
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(graphNameOrConfig: String|Map,
configuration: Map)
YIELD item1, item2, count1, count2, intersection, similarity
```

Table 330. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	no	Either then name of a loaded graph or directly the config.
configuration	Map	n/a	yes	Additional configuration, if the first parameter was a graph name.

Table 331. 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 <code>0</code> , it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If <code>0</code> , 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 <b>targets</b> list. If the list contains less than this amount, that node will be excluded from the calculation.
skipValue	Float	gds.util.NaN()	yes	Value to skip when executing similarity computation. A value of <b>null</b> 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 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
sourceIds	Integer[]	null	yes	The ids of items from which we need to compute similarities. Defaults to all the items provided in the <b>data</b> parameter.
targetIds	Integer[]	null	yes	The ids of items to which we need to compute similarities. Defaults to all the items provided in the <b>data</b> parameter.

Table 332. 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 <b>targets</b> list of one node.
count2	Integer	The size of the <b>targets</b> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <b>targets</b> lists.
similarity	Integer	The euclidean similarity of the two nodes.

## Overlap Similarity

**This section describes the Overlap Similarity algorithm in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Overlap Similarity algorithm](#)
- [Overlap Similarity algorithm function sample](#)
- [Overlap Similarity algorithm procedures sample](#)
- [Specifying source and target ids](#)
- [Syntax](#)

## 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 333. 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:

$$O(A, B) = (\lvert A \cap B \rvert) / (\min(\lvert A \rvert, \lvert B \rvert))$$

$$O(A, B) = 2 / \min(3, 4)$$

$$= 2 / 3$$

$$= 0.66$$

## 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)
```

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({nodeProjection: '*',
relationshipProjection: '*', 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 334. 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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 335. 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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 336. 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

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({
  nodeProjection: '*',
  relationshipProjection: '*',
  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 337. Results

nodes	similarity Pairs	writeRelat ionshipTyp e	writePrope rty	min	max	mean	p95
4	5	NARROWE R_THAN	score	0.66666412 35351562	1.00000381 46972656	0.88333511 35253906	1.00000381 46972656

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 338. Results

hierarchy
["Fantasy", "Science Fiction"]
["Fantasy", "Classics"]
["Fantasy", "Science Fiction", "Classics"]

## 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({
    nodeProjection: '*',
    relationshipProjection: '*',
    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 339. 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(graphNameOrConfig: String|Map, configuration: Map)
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean,
stdDev, p25, p50, p75, p90, p95, p99, p999, p100
```

Table 340. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	no	Either then name of a loaded graph or directly the config.
configuration	Map	n/a	yes	Additional configuration, if the first parameter was a graph name.

Table 341. 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 <code>0</code> , it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If <code>0</code> , 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	gds.util.NaN()	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 '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.
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	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	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 342. 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.overlap.stream(graphNameOrConfig: String|Map, configuration: Map)
YIELD item1, item2, count1, count2, similarity
```

Table 343. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	no	Either the name of a created graph or directly the config.
configuration	Map	n/a	yes	Additional configuration, if the first parameter was a graph name.

Table 344. 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 <code>0</code> , it will return as many as it finds.
topK	Integer	3	yes	The number of similar values to return per node. If <code>0</code> , 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	gds.util.NaN()	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 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.
graph	String	dense	yes	The graph name ('dense' or 'cypher').
sourceIds	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	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 345. 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 overlap similarity of the two nodes.

## Approximate Nearest Neighbors (ANN)

**This section describes the Approximate Nearest Neighbors algorithm in the Neo4j Graph Data Science library.**

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

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](#).

This section includes:

- [Syntax](#)
- [Use-cases - when to use the Approximate Nearest Neighbors algorithm](#)
- [Approximate Nearest Neighbors algorithm sample](#)
- [Usage](#)

### 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 346. 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 <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 <code>0</code> , 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.

Name	Type	Default	Optional	Description
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 <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 '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.
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 347. 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.

Name	Type	Description
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 348. 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 <code>0</code> , 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) *`
topK	`	similarityCutoff	Integer	-1

Name	Type	Default	Optional	Description
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 <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 'readConcurrency'.	readConcurrency	Integer	value of 'concurrency'

Table 349. 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 <b>targets</b> list of one node.
count2	Integer	The size of the <b>targets</b> list of other node.
intersection	Integer	The number of intersecting values in the two nodes <b>targets</b> 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({
    nodeProjection: '*',
    relationshipProjection: '*',
    data: data,
    algorithm: 'jaccard',
    similarityCutoff: 0.1,
    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 350. 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
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({
    nodeProjection: '*',
    relationshipProjection: '*',
    algorithm: 'jaccard',
    data: data,
    similarityCutoff: 0.1,
    showComputations: true,
    concurrency: 1
})
YIELD nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean,
p95
RETURN nodes, similarityPairs, writeRelationshipType, writeProperty, min, max, mean,
p95
```

Table 351. Results

nodes	similarity Pairs	writeRelat ionshipTy pe	writeProp erty	min	max	mean	p95
5	13	"SIMILAR"	"score"	0.19999980 926513672	0.66666698 45581055	0.35128226 64701022	0.66666698 45581055

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 count DESC
```

Table 352. Results

cuisine	count
"Italian"	2
"French"	1
"Lebanese"	1

## Usage

When executing ApproximateNearestNeighbors in parallel, it is possible that results are flaky

because of the asynchronous execution fashion of the algorithm.

## Path finding algorithms

*This chapter provides explanations and examples for each of the path finding algorithms in the Neo4j Graph Data Science library.*

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:

- Alpha
  - Minimum Weight Spanning Tree
  - Shortest Path
  - Single Source Shortest Path
  - All Pairs Shortest Path
  - A\*
  - Yen's K-shortest paths
  - Random Walk
  - Breadth First Search
  - Depth First Search

### Minimum Weight Spanning Tree

*This section describes the Minimum Weight Spanning Tree algorithm in the Neo4j Graph Data Science library.*

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Minimum Weight Spanning Tree algorithm](#)
- [Constraints - when not to use the Minimum Weight Spanning Tree algorithm](#)
- [Syntax](#)
- [Minimum Weight Spanning Tree algorithm sample](#)
  - [K-Spanning tree](#)

## 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 353. Configuration

Name	Type	Default	Optional	Description
startNodeId	Integer	null	no	The start node ID
relationshipWeightProperty	String	null	no	The property name that contains weight. Must be numeric.
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 354. 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
```

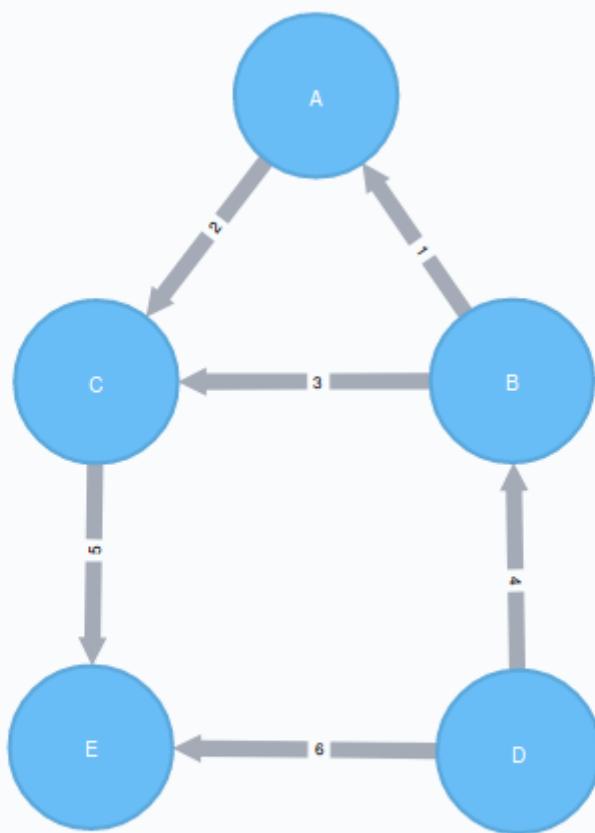
Table 355. Configuration

Name	Type	Default	Optional	Description
k	Integer	null	no	The result is a tree with <code>k</code> nodes and <code>k - 1</code> relationships
startNodeId	Integer	null	no	The start node ID
relationshipWeightProperty	String	null	no	The property name that contains weight. Must be numeric.
writeProperty	String	'MST'	yes	The relationship type written back as result
weightWriteProperty	String	n/a	no	The weight property of the <code>writeProperty</code> relationship type written back

Table 356. Results

Name	Type	Description
effectiveNodesCount	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);

```

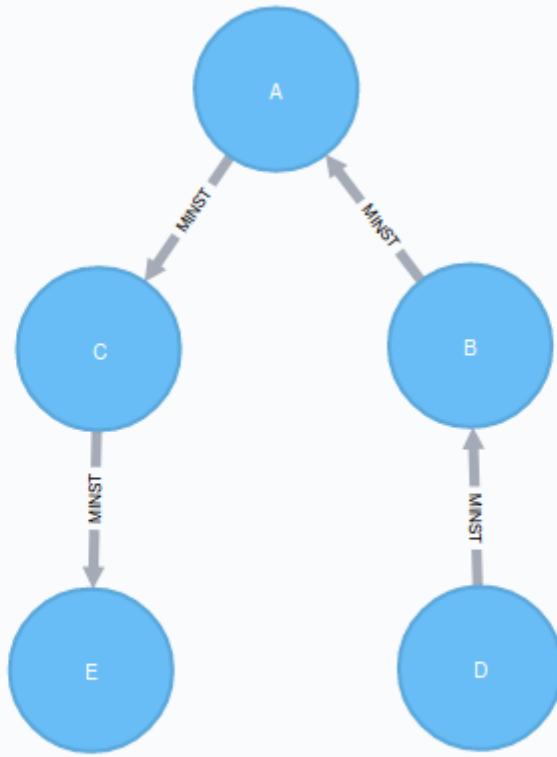
Minimum weight spanning tree visits all nodes that are in the same connected component as the starting node, and returns a spanning tree of all nodes in the component where the total weight of the relationships is minimized.

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 357. 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;
```

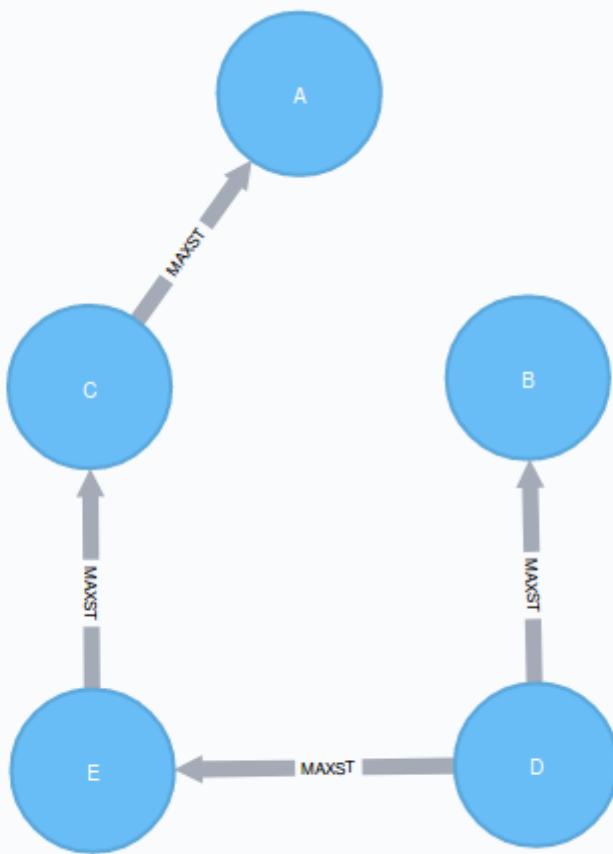


Figure 3. Results

## 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 358. 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 359. 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.

## Shortest Path

**This section describes the Shortest Path algorithm in the Neo4j Graph Data Science library.**

The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. In this category, Dijkstra's algorithm is the most well known. It is a real time graph algorithm, and can be used as part of the normal user flow in a web or mobile application.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- History and explanation
- Use-cases - when to use the Shortest Path algorithm
- Constraints - when not to use the Shortest Path algorithm
- Syntax
- Shortest Path algorithm sample
  - The Dijkstra Shortest Path algorithm
    - Cypher projection

## History and explanation

Path finding has a long history, and is considered to be one of the classical graph problems; it has been researched as far back as the 19th century. It gained prominence in the early 1950s in the context of ‘alternate routing’, i.e. finding a second shortest route if the shortest route is blocked.

Dijkstra came up with his algorithm in 1956 while trying to come up with something to show off the new ARMAC computers. He needed to find a problem and solution that people not familiar with computing would be able to understand, and designed what is now known as Dijkstra’s algorithm. He later implemented it for a slightly simplified transportation map of 64 cities in the Netherlands.

## Use-cases - when to use the Shortest Path algorithm

- Finding directions between physical locations. This is the most common usage, and web mapping tools such as Google Maps use the shortest path algorithm, or a variant of it, to provide driving directions.
- Social networks can use the algorithm to find the degrees of separation between people. For example, when you view someone’s profile on LinkedIn, it will indicate how many people separate you in the connections graph, as well as listing your mutual connections.

## Constraints - when not to use the Shortest Path algorithm

Dijkstra 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.write(configuration: Map)
YIELD nodeCount, totalCost, createMillis, computeMillis, writeMillis
```

Table 360. Configuration

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node

Name	Type	Default	Optional	Description
endNode	Node	null	no	The end node
relationshipWeightProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
writeProperty	String	'sssp'	yes	The property name written back to the node sequence of the node in the path

Table 361. Results

Name	Type	Description
nodeCount	Integer	The number of nodes considered
totalCost	Float	The sum of all weights along the path
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 algorithm and stream results:

```
CALL gds.alpha.shortestPath.stream(configuration: Map)
YIELD nodeId, cost
```

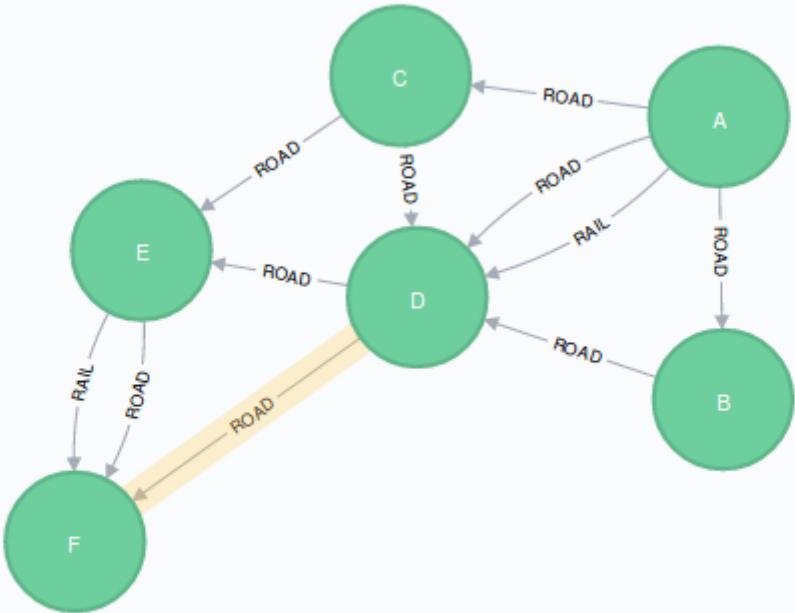
Table 362. Configuration

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node
endNode	Node	null	no	The end node
relationshipWeightProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.

Table 363. Results

Name	Type	Description
nodeId	Integer	Node ID
cost	Integer	The cost it takes to get from start node to specific node.

## 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);
    
```

### Dijkstra Shortest Path

The following will run the algorithm and stream results:

```
MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'F'})
CALL gds.alpha.shortestPath.stream({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost',
      orientation: 'UNDIRECTED'
    }
  },
  startNode: start,
  endNode: end,
  weightProperty: 'cost'
})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).name AS name, cost
```

Table 364. Results

Name	Cost
A	0
C	50
D	90
E	120
F	160

The quickest route takes us from A to F, via C, D, and E, at a total cost of 160:

- First, we go from A to C, at a cost of 50.
- Then, we go from C to D, for an additional 40.
- Then, from D to E, for an additional 30.
- Finally, from E to F, for a further 40.

The following will run the algorithm and write back results:

```
MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'F'})
CALL gds.alpha.shortestPath.write({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost',
      orientation: 'UNDIRECTED'
    }
  },
  startNode: start,
  endNode: end,
  weightProperty: 'cost',
  writeProperty: 'sssp'
})
YIELD nodeCount, totalCost
RETURN nodeCount, totalCost
```

Table 365. Results

nodeCount	totalCost
5	160

### Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

Set `graph: 'cypher'` in the config:

```
MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'F'})
CALL gds.alpha.shortestPath.write({
  nodeQuery:'MATCH(n:Loc) WHERE NOT n.name = "c" RETURN id(n) AS id',
  relationshipQuery:'MATCH(n:Loc)-[r:ROAD]->(m:Loc) RETURN id(n) AS source, id(m) AS target, r.cost AS weight',
  startNode: start,
  endNode: end,
  weightProperty: 'weight',
  writeProperty: 'sssp'
})
YIELD nodeCount, totalCost
RETURN nodeCount, totalCost
```

## Single Source Shortest Path

**This section describes the Single Source Shortest Path algorithm in the**

## *Neo4j Graph Data Science library.*

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Single Source Shortest Path algorithm](#)
- [Constraints - when not to use the Single Source Shortest Path algorithm](#)
- [Syntax](#)
- [Single Source Shortest Path algorithm sample](#)
  - [Delta stepping algorithm](#)
    - [Cypher projection](#)

### **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 366. Configuration*

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node

Name	Type	Default	Optional	Description
relationshipProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
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 367. 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.shortestPath.deltaStepping.stream(configuration: Map)
YIELD nodeId, distance
```

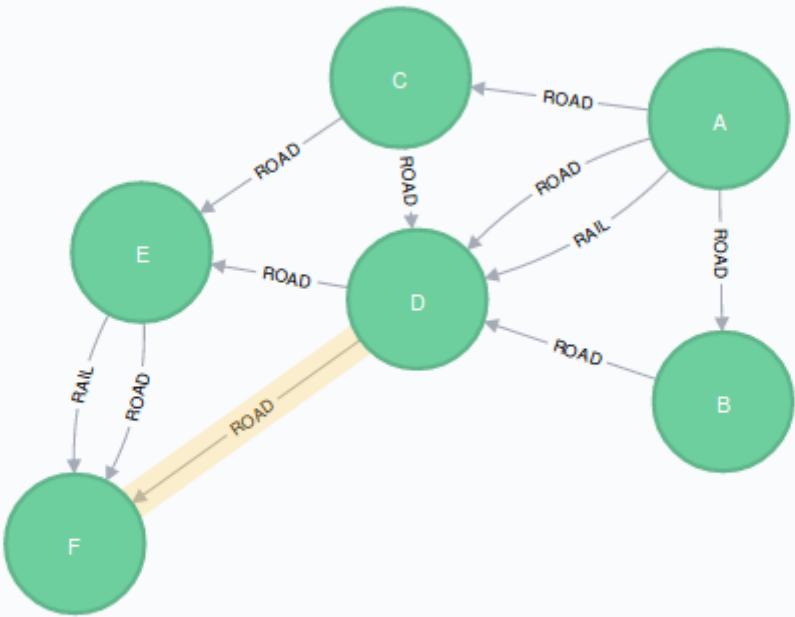
Table 368. Parameters

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node
relationshipProperty	String	null	yes	The property name that contains weight. If null, treats the graph as unweighted. Must be numeric.
delta	Float	null	yes	The grade of concurrency to use.

Table 369. 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



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 destination, distance
```

Table 370. Results

Name	Cost
A	0
B	50
C	50
D	90
E	120
F	160

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, loadDuration, evalDuration, writeDuration
RETURN nodeCount, loadDuration, evalDuration, writeDuration
```

Table 371. Results

nodeCount
6

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) 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) 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
```

## All Pairs Shortest Path

**This section describes the All Pairs Shortest Path algorithm in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the All Pairs Shortest Path algorithm](#)
- [Syntax](#)
- [All Pairs Shortest Path algorithm sample](#)

## 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:*

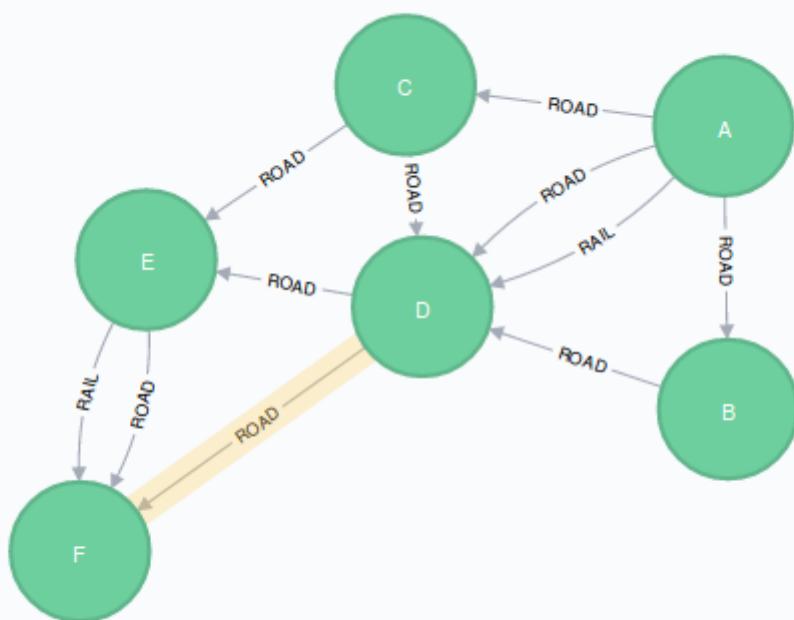
```
CALL gds.alpha.allShortestPaths.stream(configuration: Map)
YIELD startNodeId, targetNodeId, distance
```

Table 372. Parameters

Name	Type	Default	Optional	Description
relationshipWeightProperty	String	null	yes	The name of the relationship property that represents weight. If null, treats the graph as unweighted. Must be numeric.

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'. 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',  
            defaultValue: 1.0  
        }  
    },  
    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 373. Results

Source	Target	Cost
A	F	160
A	E	120
B	F	110

Source	Target	Cost
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 374. 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

## A\*

**This section describes the A\* algorithm in the Neo4j Graph Data Science library.**

The A\* (pronounced “A-star”) algorithm improves on the classic Dijkstra algorithm. It is based upon the observation that some searches are informed, and that by being informed we can make better choices over which paths to take through the graph.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the A\\* algorithm](#)
- [Syntax](#)
- [A\\* algorithm sample](#)
- [Cypher projection](#)

### History and explanation

The A\* algorithm was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. For more information, see [A Formal Basis for the Heuristic Determination of Minimum Cost Paths](#).

In A\*, we split the path cost into two parts:

#### g(n)

This is the cost of the path from the starting point to some node n.

#### h(n)

This represents the estimated cost of the path from the node n to the destination node, as computed by a heuristic (an intelligent guess).

The A\* algorithm balances  $g(n)$  and  $h(n)$  as it iterates the graph, thereby ensuring that at each iteration it chooses the node with the lowest overall cost  $f(n) = g(n) + h(n)$ .

In our implementation, geospatial distance is used as heuristic.

### Use-cases - when to use the A\* algorithm

- The A\* algorithm can be used to find shortest paths between single pairs of locations, where GPS coordinates are known.

### Syntax

The following will run the algorithm and stream results:

```
CALL gds.alpha.shortestPath.astar.stream(configuration:Map)
YIELD nodeId, cost
```

Table 375. Configuration

Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node
endNode	Node	null	no	The end node
relationships	String	null	yes	The property name that contains weight
propertyKeyLat	String	null	no	The property name that contains latitude coordinate
propertyKeyLon	String	null	no	The property name that contains longitude coordinate
nodeQuery	String	null	yes	The label to load from the graph. If null, load all nodes
relationshipsQuery	String	null	yes	The relationship type to load from the graph. If null, load all nodes
defaultValue	Float	null	yes	The default value of the weight in case it is missing or invalid

Table 376. Results

Name	Type	Description
nodeId	Integer	Node ID
cost	Integer	The cost it takes to get from start node to specific node

### A\* algorithm sample

The following will create a sample graph:

```
CREATE (a:Station {name: 'King\'s Cross St. Pancras', 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 {time: 2}]->(b),
       (b)-[:CONNECTION {time: 3}]->(c),
       (b)-[:CONNECTION {time: 2}]->(d),
       (d)-[:CONNECTION {time: 2}]->(c),
       (c)-[:CONNECTION {time: 2}]->(e)
```

The following will run the algorithm and stream results:

```
MATCH (start:Station {name: 'King\'s Cross St. Pancras'}), (end:Station {name: 'Kentish Town'})
CALL gds.alpha.shortestPath.astar.stream({
  nodeProjection: '*',
  relationshipProjection: {
    CONNECTION: {
      type: 'CONNECTION',
      orientation: 'UNDIRECTED',
      properties: 'time'
    }
  },
  startNode: start,
  endNode: end,
  propertyKeyLat: 'latitude',
  propertyKeyLon: 'longitude'
})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).name AS station, cost
```

Table 377. Results

station	cost
"King's Cross St. Pancras"	0
"Euston"	2
"Camden Town"	5
"Kentish Town"	7

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

```
MATCH (start:Station {name: "King's Cross St. Pancras"}), (end:Station {name: "Kentish Town"})
CALL gds.alpha.shortestPath.astar.stream({
    nodeQuery: 'MATCH (p:Station) RETURN id(p) AS id',
    relationshipQuery: 'MATCH (p1:Station)-[r:CONNECTION]->(p2:Station) RETURN id(p1) AS source, id(p2) AS target, r.time AS weight',
    startNode: start,
    endNode: end,
    relationshipWeightProperty: 'time',
    propertyKeyLat: 'latitude',
    propertyKeyLon: 'longitude'
})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).name AS station, cost
```

Table 378. Results

station	cost
"King's Cross St. Pancras"	0
"Euston"	2
"Camden Town"	5
"Kentish Town"	7

## Yen's K-Shortest Paths

*This section describes the Yen's K-shortest paths algorithm in the Neo4j Graph Data Science library.*

Yen's K-shortest paths algorithm computes single-source K-shortest loopless paths for a graph with non-negative relationship weights.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Yen's K-shortest paths algorithm](#)
- [Constraints - when not to use the Yen's K-shortest paths algorithm](#)
- [Syntax](#)
- [Yen's K-shortest paths algorithm sample](#)

- Cypher projection

## History and explanation

Algorithm was defined in 1971 by Jin Y. Yen in the research paper [Finding the K Shortest Loopless Paths in a Network](#). Our implementation uses Dijkstra algorithm to find the shortest path and then proceeds to find k-1 deviations of the shortest paths.

## Use-cases - when to use the Yen's K-shortest paths algorithm

- K-shortest paths algorithm has been used to optimize multiple object tracking by formalizing the motions of targets as flows along the relationships of the spatial graph. Find more in [Multiple Object Tracking using K-Shortest Paths Optimization](#)
- K-shortest paths algorithm is used to study alternative routing on road networks and to recommend top k-paths to the user. Find this study in [Alternative Routing: k-Shortest Paths with Limited Overlap](#)
- K-shortest paths algorithm has been used as part of [Finding Diverse High-Quality Plans for Hypothesis Generation](#) process.

## Constraints - when not to use the Yen's K-shortest paths algorithm

Yen's K-Shortest paths algorithm 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.kShortestPaths.write(configuration: Map)
YIELD resultCount, createMillis, computeMillis, writeMillis
```

Table 379. Configuration

Name	Type	Default	Optional	Description
startNode	Node	n/a	no	The start node
endNode	Node	n/a	no	The end node
k	Integer	n/a	no	The number of paths to return.
relationshipWeightProperty	String	null	yes	The property name that contains a relationship weight. If null, treats the graph as unweighted. Must be of numeric type.
maxDepth	Integer	Integer.MAX_VALUE	yes	The depth of the shortest paths traversal.
writePropertyPrefix	String	'PATH_'	yes	The relationship-type prefix written back to the graph.

Name	Type	Default	Optional	Description
relationshipProperty	String	'weight'	yes	The relationship property written back to the graph.

Table 380. Results

Name	Type	Description
resultSet	Integer	The number of shortest paths results
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 algorithm and stream the results:

```
CALL gds.alpha.kShortestPaths.stream(configuration: Map)
YIELD index, sourceNodeId, targetNodeId, nodeIds, costs, path
```

Table 381. Configuration

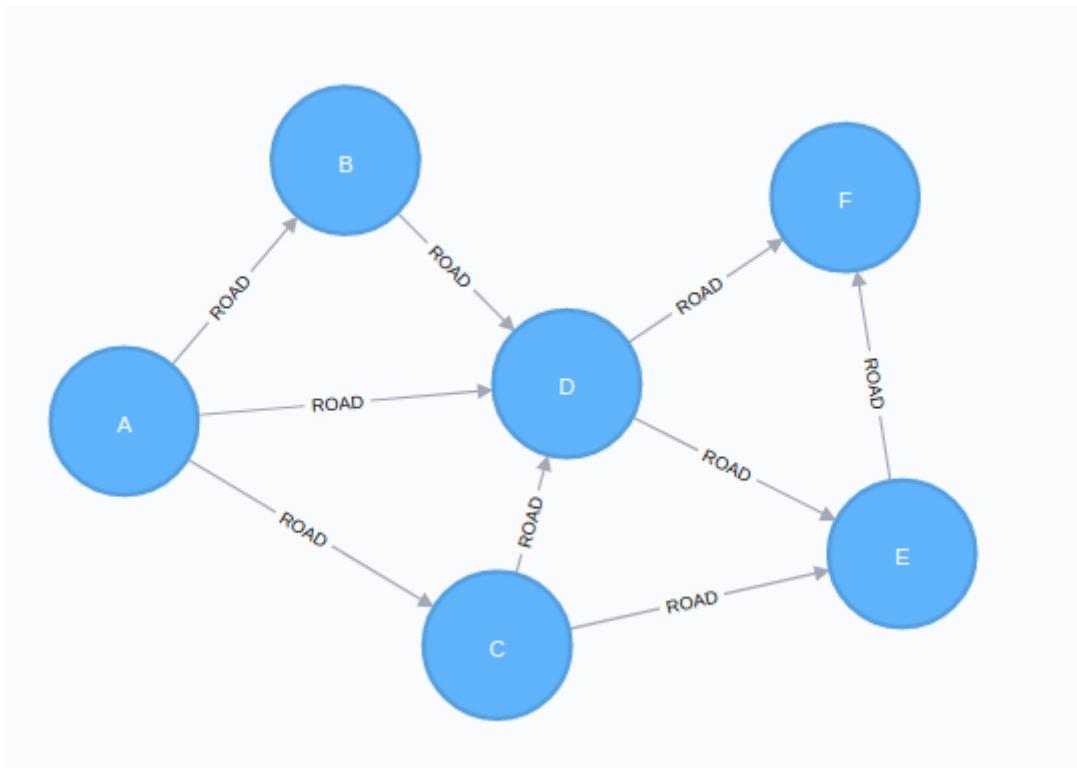
Name	Type	Default	Optional	Description
startNode	Node	null	no	The start node
endNode	Node	null	no	The end node
k	Integer	N/A	no	The number of paths to return.
relationshipProperty	String	null	yes	The relationship property name that contains weight. If null, treats the graph as unweighted. Must be of numeric type.
maxDepth	Integer	Integer.MAX_VALUE	yes	The depth of the shortest paths traversal.
path	Boolean	false	yes	Whether or not to include string representation of the path with the result.

Table 382. Results

Name	Type	Description
index	Integer	Index of the result
startNodeId	Integer	The start node id
targetNodeId	Integer	The end node id
nodeIds	Integer[]	List containing the node ids that form the path.

Name	Type	Description
costs	Float[]	List containing the costs.
path	String	Optional string representation of the path.

### Yen's K-shortest paths 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:

```
MATCH (start:Loc{name: 'A'}), (end:Loc{name: 'F'})  
CALL gds.alpha.kShortestPaths.stream({  
    nodeProjection: 'Loc',  
    relationshipProjection: {  
        ROAD: {  
            type: 'ROAD',  
            properties: 'cost'  
        }  
    },  
    startNode: start,  
    endNode: end,  
    k: 3,  
    relationshipWeightProperty: 'cost'  
})  
YIELD index, nodeIds, costs  
RETURN [node IN gds.util.asNodes(nodeIds) | node.name] AS places,  
       costs,  
       reduce(acc = 0.0, cost IN costs | acc + cost) AS totalCost
```

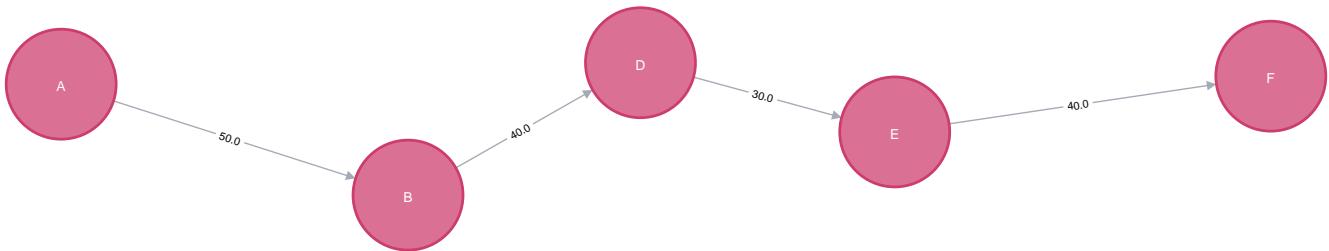
Table 383. Results

places	costs	totalCost
["A", "B", "D", "E", "F"]	[50.0, 40.0, 30.0, 40.0]	160.0
["A", "C", "D", "E", "F"]	[50.0, 40.0, 30.0, 40.0]	160.0
["A", "B", "D", "F"]	[50.0, 40.0, 80.0]	170.0

This procedure doesn't return paths by default, but we can have it return those by passing in the config `path: true`.

The following will run the algorithm and stream results, including paths:

```
MATCH (start:Loc{name: 'A'}), (end:Loc{name: 'F'})
CALL gds.alpha.kShortestPaths.stream({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost'
    }
  },
  startNode: start,
  endNode: end,
  k: 3,
  relationshipWeightProperty: 'cost',
  path: true
})
YIELD path
RETURN path
LIMIT 1
```



The following will run the algorithm and write back results:

```
MATCH (start:Loc{name: 'A'}), (end:Loc{name: 'F'})
CALL gds.alpha.kShortestPaths.write({
  nodeProjection: 'Loc',
  relationshipProjection: {
    ROAD: {
      type: 'ROAD',
      properties: 'cost'
    }
  },
  startNode: start,
  endNode: end,
  k: 3,
  relationshipWeightProperty: 'cost'
})
YIELD resultCount
RETURN resultCount
```

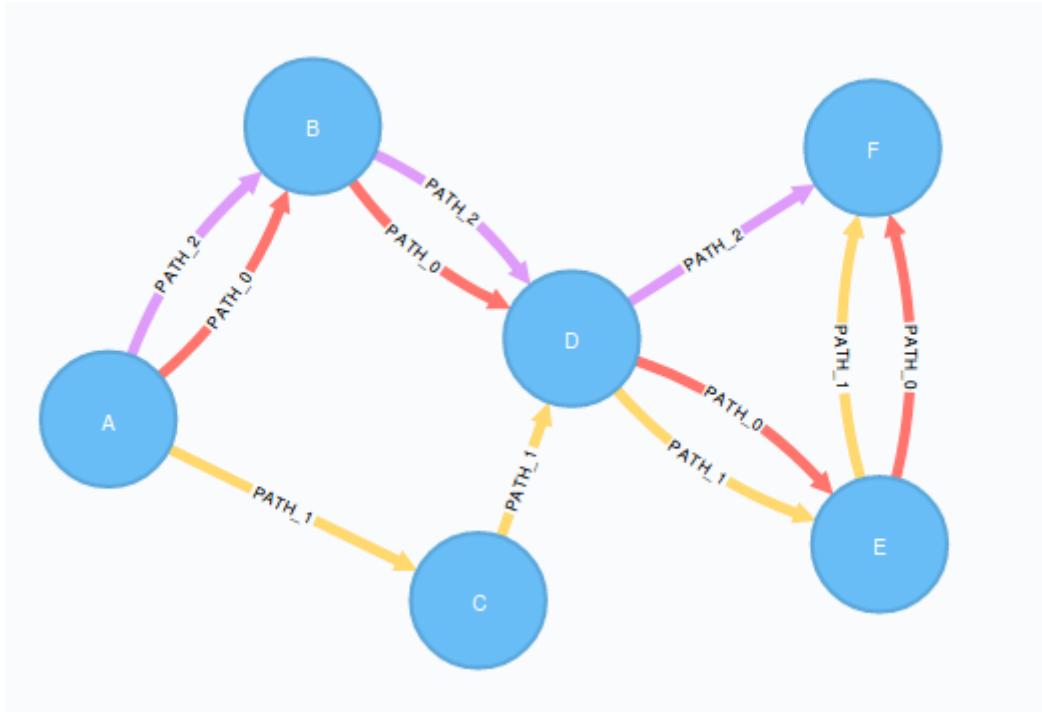
Table 384. Results

resultCount

3

The following will return all 3 of the shortest path:

```
MATCH p=()-[r:PATH_0|:PATH_1|:PATH_2]->() RETURN p LIMIT 25
```



The quickest route takes us from A to B, via D and E and is saved as `PATH_0`. Second quickest path is saved as `PATH_1` and third one is saved as ``PATH_2``

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

```
MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'F'})
CALL gds.alpha.kShortestPaths.write({
    nodeQuery: 'MATCH(n:Loc) WHERE NOT n.name = "C" RETURN id(n) AS id',
    relationshipQuery: 'MATCH (n:Loc)-[r:ROAD]->(m:Loc) RETURN id(n) AS source, id(m) AS target, r.cost AS weight',
    startNode: start,
    endNode: end,
    k: 3,
    relationshipWeightProperty: 'cost',
    writePropertyPrefix: 'cypher_'
})
YIELD resultCount
RETURN resultCount
```

## Random Walk

*This section describes the Random Walk algorithm in the Neo4j Graph Data Science library.*

Random Walk is an algorithm that provides random paths in a graph.

A random walk means that we start at one node, choose a neighbor to navigate to at random or based on a provided probability distribution, and then do the same from that node, keeping the resulting path in a list. It's similar to how a drunk person traverses a city.

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Use-cases - when to use the Random Walk algorithm](#)
- [Constraints - when not to use the Random Walk algorithm](#)
- [Syntax](#)
- [Random Walk algorithm sample](#)
- [Cypher projection](#)

### History and explanation

The term "random walk" was first mentioned by Karl Pearson in 1905 in a letter to Nature magazine titled [The Problem of the Random Walk](#). Study of random walks date back even further to the [Gambler's ruin](#) problem, where it could be used to show that a gambler would eventually go bankrupt against an opponent with infinite wealth.

It's only in the last couple of decades, however, that researchers have studied them with respect to networks.

### Use-cases - when to use the Random Walk algorithm

- It has been shown to relate to Brownian motion and also to the movement and dispersal of animals in the study of [Random walk models in biology](#).
- It has been used to analyse ALSI index of the JSE stock exchange and show that the index followed the random walk hypothesis between years 2000 and 2011. This means the movement of stock prices was random and the ability of investors to perform relied more on luck than anything else. Find this study in [The Random Walk Theory And Stock Prices: Evidence From Johannesburg Stock Exchange](#)

Random Walk is often used as part of other algorithms:

- It can be used as part of the **node2vec** and **graph2vec** algorithms, that create node embeddings.
- It can be used as part of the **Walktrap** and **Infomap community detection** algorithms. If a random walk returns a small set of nodes repeatedly, then it indicates that those set of nodes

may have a community structure.

- It can be used as part of the training process of machine learning model, as described in David Mack's article [Review prediction with Neo4j and TensorFlow](#).

You can read about more use cases in [Random walks and diffusion on networks](#).

### Constraints - when not to use the Random Walk algorithm

The constraints of [Page Rank](#) also apply to Random Walks:

- Dead-ends occur when pages have no out-links. In this case, the random walk will abort and a path containing only the first node will be returned. This problem can be avoided by running on an undirected graph, so that the random walk will traverse relationships in both directions.
- If there are no links from within a group of pages to outside of the group, then the group is considered a spider trap. Random walks starting from any of the nodes in that group will only traverse to the others in the group - our implementation of the algorithm doesn't allow a random walk to jump to non-neighbouring nodes.
- Sinks can occur when a network of links form an infinite cycle.

### Syntax

*The following will run the algorithm and stream results:*

```
CALL gds.alpha.randomWalk.stream(configuration: Map)
YIELD startNodeId, nodeIds, path
```

*Table 385. Configuration*

Name	Type	Default	Optional	Description
start	Object	null	yes	Starting points: null - whole graph, "Label" - nodes with that label, node-id - that node, list of node-ids - these nodes.
steps	Integer	10	yes	Length of paths returned, in case of error only path of length 1 is returned.
walks	Integer	1	yes	Number of paths returned.
mode	String	random	yes	Strategy for choosing the next relationship, modes: random and node2vec.
inOut	Float	1.0	yes	Parameter for node2vec.
return	Float	1.0	yes	Parameter for node2vec.
path	Boolean	false	yes	If the more expensive operation of creating a path from node-ids should be performed and returned in results.

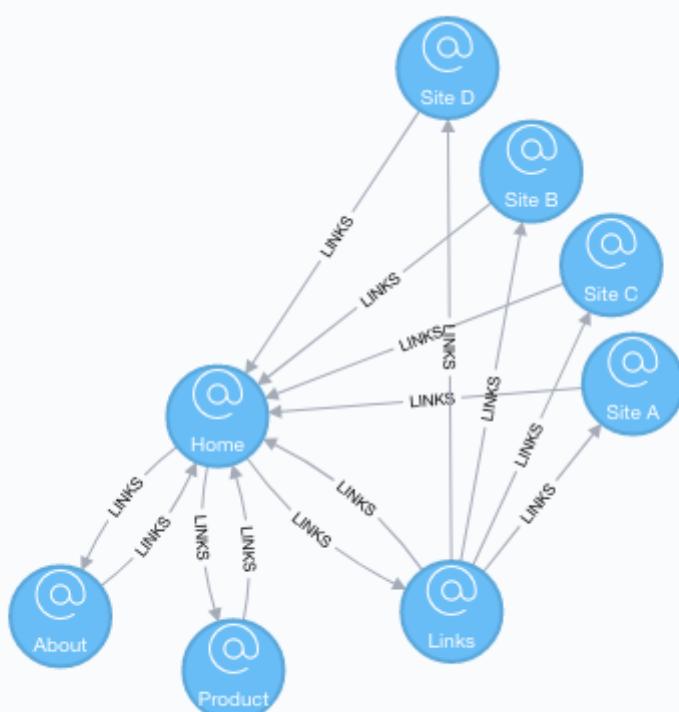
Name	Type	Default	Optional	Description
concurrency	Integer	4	yes	The number of concurrent threads used for running the algorithm. Also provides the default value for 'readConcurrency'.
readConcurrency	Integer	value of 'concurrency'	yes	The number of concurrent threads used for reading the graph.

Table 386. Results

Name	Type	Description
startNodeId	Integer	Node ID starting the path.
nodeIds	Integer[]	List of Node ID forming a path.
path	Path	Optional Path (with virtual relationships).

## Random Walk algorithm sample

This sample will explain the Random Walk algorithm, using a simple graph:



The following will create a sample graph:

```
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)
```

The following will run the algorithm starting from the Home page and returning a 1 random walk, of path length 3:

```
MATCH (home:Page {name: 'Home'})  
CALL gds.alpha.randomWalk.stream({  
    nodeProjection: '*',  
    relationshipProjection: {  
        LINKS: {  
            type: 'LINKS',  
            orientation: 'UNDIRECTED'  
        }  
    },  
    start: id(home),  
    steps: 3,  
    walks: 1  
})  
YIELD nodeIds  
UNWIND nodeIds AS nodeId  
RETURN gds.util.asNode(nodeId).name AS page
```

Table 387. Results

page
"Home"

<b>page</b>
"Site C"
"Links"
"Site A"

## Cypher projection

If node label and relationship type are not selective enough to create the graph projection to run the algorithm on, you can use Cypher queries to project your graph. This can also be used to run algorithms on a virtual graph. You can learn more in the [Cypher projection](#) section of the manual.

```

MATCH (home:Page {name: 'Home'})
CALL gds.alpha.randomWalk.stream({
    nodeQuery: 'MATCH (p:Page) RETURN id(p) AS id',
    relationshipQuery: 'MATCH (p1:Page)-[:LINKS]->(p2:Page) RETURN id(p1) AS source,
    id(p2) AS target',
    start: id(home),
    steps: 5,
    walks: 1
})
YIELD nodeIds
UNWIND nodeIds AS nodeId
RETURN gds.util.asNode(nodeId).name AS page

```

## Breadth First Search

*This section describes the Breadth First Search traversal algorithm in the Neo4j Graph Data Science library.*

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)

### 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 388. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	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 389. 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.

Name	Type	Default	Optional	Description
relationshipsQuery	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 390. Algorithm specific configuration

Name	Type	Default	Optional	Description
startNodeId	Integer	n/a	no	The node id of the node where to start the traversal.
targetNodes	Integer[]	empty list	yes	Ids for target nodes. Traversal terminates when any target node is visited.
maxDepth	Integer	-1	yes	The maximum distance from the start node at which nodes are visited.
maxCost	Integer	NaN	yes	The maximum accumulated cost of any path from start node to a node that should be visited.

Table 391. Results

Name	Type	Description
startNodeId	Integer	The node id of the node where to start the traversal.
nodeIds	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 392. 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 393. 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 394. Results*

tags
"a"
"b"
"c"

In the above example, nodes d and e were not visited since they are at distance 2 from a.

*Running the Breadth First Search algorithm with maxCost:*

```
MATCH (a:Node{tag:'a'})
WITH id(a) AS startNode
CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, maxCost: 10,
relationshipWeightProperty: 'cost'})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags
```

*Table 395. Results*

tags
"a"
"b"
"c"
"e"

Due to the max cost limit of 10, node d cannot be reached since the total cost of the path from a to d is 14.

## Depth First Search

***This section describes the Depth First Search traversal algorithm in the Neo4j Graph Data Science library.***

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This topic includes:

- [Introduction](#)
- [Syntax](#)
- [Examples](#)

## 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 396. Parameters

Name	Type	Default	Optional	Description
graphNameOrConfig	String or Map	null	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 397. 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 398. Algorithm specific configuration

Name	Type	Default	Optional	Description
startNodeId	Integer	n/a	no	The node id of the node where to start the traversal.
targetNodes	Integer[]	empty list	yes	Ids for target nodes. Traversal terminates when any target node is visited.
maxDepth	Integer	-1	yes	The maximum distance from the start node at which nodes are visited.
maxCost	Integer	NaN	yes	The maximum accumulated cost of any path from start node to a node that should be visited.

Table 399. Results

Name	Type	Description
startNodeId	Integer	The node id of the node where to start the traversal.
nodeIds	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 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 400. Results

tags
"a"
"b"
"c"

tags
"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 401. Results*

tags
"a"
"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 402. Results*

tags
"a"
"b"
"c"

In the above case, nodes d and e were not visited since they are at distance 2 from a.

Running the Depth First Search algorithm with maxCost:

```
MATCH (a:Node{tag:'a'})
WITH id(a) AS startNode
CALL gds.alpha.dfs.stream('myGraph', {startNode: startNode, maxCost: 10,
relationshipWeightProperty: 'cost'})
YIELD path
UNWIND [ n in nodes(path) | n.tag ] AS tags
RETURN tags
ORDER BY tags
```

Table 403. Results

tags
"a"
"b"
"c"
"e"

Due to the max cost limit of 10, node d cannot be reached since the total cost of the path from a to d is 14.

## Link Prediction algorithms

*This chapter provides explanations and examples for each of the link prediction algorithms in the Neo4j Graph Data Science library.*

Link prediction algorithms help determine the closeness of a pair of nodes. The computed scores can then be used to predict new relationships between them. 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](#)

### Adamic Adar

*This section describes the Adamic Adar algorithm in the Neo4j Graph Data Science library.*

[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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Syntax](#)
- [Adamic Adar algorithm sample](#)

## 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 404. 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 <code>node1</code> and <code>node2</code>
direction	String	BOTH	yes	The direction of relationship type used to compute similarity between <code>node1</code> and <code>node2</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 405. 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 406. Results

score
0.0

## Common Neighbors

This section describes the Common Neighbors algorithm in the Neo4j Graph Data Science library.

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Syntax](#)
- [Common Neighbors algorithm sample](#)

## 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 407. 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 <code>node1</code> and <code>node2</code>
direction	String	BOTH	yes	The direction of relationship type used to compute similarity between <code>node1</code> and <code>node2</code>

## 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 408. 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 409. Results

score
0.0

## Preferential Attachment

This section describes the Preferential Attachment algorithm in the Neo4j Graph Data Science library.

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Syntax](#)
- [Preferential Attachment algorithm sample](#)

## 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 410. 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 direction of relationship type used to compute similarity between <code>node1</code> and <code>node2</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 411. 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 412. Results

score
1.0

## Resource Allocation

This section describes the Resource Allocation algorithm in the Neo4j Graph Data Science library.

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Syntax](#)
- [Resource Allocation algorithm sample](#)

## 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 413. 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 to use to compute similarity between node1 and node2
direction	String	BOTH	yes	The direction of relationship type to use to compute similarity between node1 and node2

## 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 414. 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 415. Results

score
0.0

## Same Community

**This section describes the Same Community algorithm in the Neo4j Graph Data Science library.**

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 algorithms](#).

This algorithm is in the alpha tier. For more information on this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Syntax](#)
- [Same Community algorithm sample](#)

## 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 416. 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 417. 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 418. 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 419. 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 420. Results

score

1.0

## Total Neighbors

**This section describes the Total Neighbors algorithm in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).

This section includes:

- [History and explanation](#)
- [Syntax](#)
- [Total Neighbors algorithm sample](#)

### 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 421. Parameters

Name	Type	Default	Optional	Description
node1	Node	null	no	A node
node2	Node	null	no	Another node

Name	Type	Default	Optional	Description
<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 direction of relationship type used to compute similarity between <code>node1</code> and <code>node2</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 422. Results

<code>score</code>
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 423. Results

<code>score</code>
2.0

# Auxiliary procedures

**This chapter provides explanations and examples for auxiliary procedures in the Neo4j Graph Data Science library.**

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
  - [One Hot Encoding](#)

## Graph Generation

**This section describes how random graphs can be generated in the Neo4j Graph Data Science library.**

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 this tier of algorithm, see [here](#).



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 424. Parameters

Name	Type	Default	Optional	Description
graphName	String	null	no	The name under which the generated graph is stored.
nodeCount	Integer	null	no	The number of generated nodes.
averageDegree	Integer	null	no	The average out-degree of generated nodes.
configuration	Map	{}	yes	Additional configuration, see below.

Table 425. Configuration

Name	Type	Default	Optional	Description
relationshipDistribution	String	UNIFORM	yes	The probability distribution method used to connect generated nodes. For more information see <a href="#">Relationship Distribution</a> .
relationshipSeed	Integer	null	yes	The seed used for generating relationships.
relationshipProperty	Map	{}	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> .
aggregation	String	NONE	yes	The relationship aggregation method cf. <a href="#">Native Projection</a> .
orientation	String	NATURAL	yes	The method of orienting edges. Allowed values are NATURAL, REVERSE and UNDIRECTED.
allowSelfLoops	Boolean	false	yes	Whether to allow relationships with identical source and target node.

Table 426. Results

Name	Type	Description
name	String	The name under which the stored graph was stored.
nodes	Integer	The number of nodes in the graph.

Name	Type	Description
<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 * 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 427. 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.

Name	Type	Default	Optional	Description
<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.

## One Hot Encoding

*This section describes the One Hot Encoding function in the Neo4j Graph Data Science library.*

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 this tier of algorithm, see [here](#).

### 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 428. 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 429. Results

name	embedding
Arya	[0,0,0]
Michael	[1,0,1]
Praveena	[0,1,0]
Zhen	[1,0,0]

Table 430. Parameters

Name	Type	Default	Optional	Description
availableValues	list	null	yes	The available values. If null, the function will return an empty list.
selectedValues	list	null	yes	The selected values. If null, the function will return a list of all 0's.

Table 431. Results

Type	Description
list	One hot encoding of the selected values.

# Production deployment

*This chapter explains advanced details with regards to common Neo4j components.*

This chapter is divided into the following sections:

- [Transaction Handling](#)

## Transaction Handling

*This section describes the usage of transactions during the execution of an algorithm. When an algorithm procedure is called from Cypher, the procedure call is executed within the same transaction as the Cypher statement.*

This section includes:

- [During graph loading](#)
- [During graph writing](#)
- [Transaction termination](#)

### During graph loading

During loading, 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 loading transactions.

For example, the following statement will not be able to load any nodes:

```
MATCH (n) SET n:MyLabel CALL algo.graph.load('graph-name', 'MyLabel', '')
```

Similarly, the following statement will execute the algorithm over an empty graph:

```
MATCH (n)
SET n:MyLabel
CALL algo.pageRank('MyLabel', null)
```

### During graph writing

Properties are written to the graph in new transactions, batched over a fixed number of nodes. Those 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 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 loading, writing, or the algorithm execution. It is not immediately visible and can take a moment for the transactions to recognize that the Cypher transaction has been terminated.

# Appendix A: Procedures and functions reference

*This chapter contains a reference of all the procedures and functions in the Neo4j Graph Data Science library.*

This chapter contains the following sections:

- [Graph Operations](#)
- [Production-quality tier](#)
- [Beta tier](#)
- [Alpha tier](#)

## Graph Operations

The following table lists all production-quality graph operations in the GDS library:

Operation	Procedure
Create Graph	<code>gds.graph.create</code>
	<code>gds.graph.create.estimate</code>
	<code>gds.graph.create.cypher</code>
	<code>gds.graph.create.cypher.estimate</code>
Check if a named graph exists	<code>gds.graph.exists</code>
List graphs	<code>gds.graph.list</code>
Remove node properties from a named graph	<code>gds.graph.removeNodeProperties</code>
Delete relationships from a named graph	<code>gds.graph.deleteRelationships</code>
Remove a named graph from memory	<code>gds.graph.drop</code>
Write node properties to Neo4j	<code>gds.graph.writeNodeProperties</code>
Write relationships to Neo4j	<code>gds.graph.writeRelationship</code>

## Production-quality tier

The following table lists all production-quality procedures in the GDS library:

Algorithm	Procedure
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	Procedure
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>

## Beta tier

The following table lists all beta graph operations in the GDS library:

Operation	Procedure
Generate Random Graph	<code>gds.beta.graph.generate</code>

The following table lists all beta procedures in the GDS library:

Algorithm	Procedure
K1Coloring	<code>gds.beta.k1coloring.mutate</code> <code>gds.beta.k1coloring.mutate.estimate</code> <code>gds.beta.k1coloring.stats</code> <code>gds.beta.k1coloring.stats.estimate</code> <code>gds.beta.k1coloring.stream</code> <code>gds.beta.k1coloring.stream.estimate</code> <code>gds.beta.k1coloring.write</code> <code>gds.beta.k1coloring.write.estimate</code>
Modularity Optimization	<code>gds.beta.modularityOptimization.mutate</code> <code>gds.beta.modularityOptimization.mutate.estimate</code> <code>gds.beta.modularityOptimization.stream</code> <code>gds.beta.modularityOptimization.stream.estimate</code> <code>gds.beta.modularityOptimization.write</code> <code>gds.beta.modularityOptimization.write.estimate</code>

## Alpha tier

The following table lists all beta graph operations in the GDS library:

Operation	Procedure
Graph Export	<code>gds.beta.graph.export</code>

The following table lists all alpha procedures in the GDS library:

Algorithm	Procedure
All Shortest Paths	<code>gds.alpha.allShortestPaths.stream</code>
Article Rank	<code>gds.alpha.articleRank.stream</code> <code>gds.alpha.articleRank.write</code>
Betweenness Centrality	<code>gds.alpha.betweenness.stream</code> <code>gds.alpha.betweenness.write</code> <code>gds.alpha.betweenness.sampled.stream</code> <code>gds.alpha.betweenness.sampled.write</code>
Breadth First Search	<code>gds.alpha.bfs.stream</code>
Closeness Centrality	<code>gds.alpha.closeness.stream</code> <code>gds.alpha.closeness.write</code>
Degree Centrality	<code>gds.alpha.degree.stream</code> <code>gds.alpha.degree.write</code>
Depth First Search	<code>gds.alpha.dfs.stream</code>

Algorithm	Procedure
Eigenvector Centrality	<code>gds.alpha.eigenvector.stream</code> <code>gds.alpha.eigenvector.write</code>
K-Shortest Paths	<code>gds.alpha.kShortestPaths.stream</code> <code>gds.alpha.kShortestPaths.write</code>
Shortest Paths	<code>gds.alpha.shortestPaths.stream</code> <code>gds.alpha.shortestPaths.write</code>
Random Walk	<code>gds.alpha.randomWalk.stream</code>
Strongly Connected Components	<code>gds.alpha.scc.stream</code> <code>gds.alpha.scc.write</code>
Shortest Path	<code>gds.alpha.shortestPath.stream</code> <code>gds.alpha.shortestPath.write</code>
A-Star	<code>gds.alpha.shortestPath.astar.stream</code>
Single Source Shortest Path	<code>gds.alpha.shortestPath.deltaStepping.write</code> <code>gds.alpha.shortestPath.deltaStepping.stream</code>
Cosine Similarity	<code>gds.alpha.similarity.cosine.stream</code> <code>gds.alpha.similarity.cosine.write</code>
Euclidean Similarity	<code>gds.alpha.similarity.euclidean.stream</code> <code>gds.alpha.similarity.euclidean.write</code>
Overlap Similarity	<code>gds.alpha.similarity.overlap.stream</code> <code>gds.alpha.similarity.overlap.write</code>
Pearson Similarity	<code>gds.alpha.similarity.pearson.write</code> <code>gds.alpha.similarity.pearson.stream</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>
Triangle Finding	<code>gds.alpha.triangles</code>

The following table lists all functions in the GDS library:

Group	Function
Miscellaneous	<code>gds.version</code> <code>gds.list</code>
Graph Operations	<code>gds.graph.exists</code>

Group	Function
Utilities	<code>gds.util.asNode</code>
	<code>gds.util.asNodes</code>
	<code>gds.util.nodeProperty</code>
	<code>gds.util.NaN</code>
	<code>gds.util.infinity</code>
	<code>gds.util.isFinite</code>
Link Prediction	<code>gds.alpha.linkprediction.adamicAdar</code>
	<code>gds.alpha.linkprediction.commonNeighbors</code>
	<code>gds.alpha.linkprediction.preferentialAttachment</code>
	<code>gds.alpha.linkprediction.resourceAllocation</code>
	<code>gds.alpha.linkprediction.sameCommunity</code>
	<code>gds.alpha.linkprediction.totalNeighbors</code>
Encoding	<code>gds.alpha.ml.oneHotEncoding</code>
Similarity Functions	<code>gds.alpha.similarity.cosine</code>
	<code>gds.alpha.similarity.euclidean</code>
	<code>gds.alpha.similarity.jaccard</code>
	<code>gds.alpha.similarity.euclideanDistance</code>
	<code>gds.alpha.similarity.overlap</code>
	<code>gds.alpha.similarity.pearson</code>

# Appendix B: Migration from Graph Algorithms v3.5

If you have previously used Graph Algorithm v3.5, you can find the information you will need to migrate to using the Graph Data Science library in this section.

## 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.

## 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](#)

### Common changes

This section describes changes between Graph Algorithms library and Graph Data Science library that are common to all procedures.

Table 432. Namespace

Graph Algorithms v3.5	Graph Data Science v1.0
algo.*	gds.*

Table 433. Changes in Parameters

<b>Graph Algorithms v3.5</b>	<b>Graph Data Science v1.0 Named Graph</b>	<b>Graph Data Science v1.0 Anonymous Graph</b>
-	graphName	graphConfiguration
node label <sup>[1]</sup>	-	-
relationship type <sup>[2]</sup>	-	-
direction	-	-
config	configuration	-

Table 434. Changes in configuration parameter map

<b>Graph Algorithms v3.5</b>	<b>Graph Data Science v1.0</b>
write: true	Replaced by dedicated write mode
graph: 'cypher'   'huge'	Removed. Always using huge graph <sup>[3]</sup>
direction	Replaced by projection parameter of relationshipProjection
direction: 'OUTGOING'	orientation: 'NATURAL'
direction: 'INCOMING'	orientation: 'REVERSE'
direction: 'BOTH'	Removed <sup>[4]</sup>
undirected: true	Replaced by orientation: 'UNDIRECTED' parameter of relationshipProjection
duplicateRelationships	Replaced by aggregation parameter of relationshipProjection
duplicateRelationships: 'SKIP'	aggregation: 'SINGLE'
iterations	maxIterations

## Memory estimation

Table 435. Changes in the YIELD fields

<b>Graph Algorithms v3.5</b>	<b>Graph Data Science v1.0</b>
requiredMemory	requiredMemory
bytesMin	bytesMin
bytesMax	bytesMax
mapView	mapView
-	treeView
-	nodeCount
-	relationshipCount

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 436. Estimating the memory requirements of loading a named graph:

Native Projections:

```
CALL algo.memrec(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  'graph.load'
)
```

```
CALL gds.graph.create.estimate(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

Cypher Projections:

```
CALL algo.memrec(
  'MATCH (n:MyLabel) RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
    RETURN id(s) AS source, id(t) AS target',
  'graph.load',
  {
    graph: 'cypher'
  }
)
```

```
CALL gds.graph.create.cypher.estimate(
  'MATCH (n:MyLabel) RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
    RETURN id(s) AS source, id(t) AS target'
)
```

## Graph creation - Named Graph

Table 437. 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	-

Graph Algorithms v3.5	Graph Data Science v1.0
loadRelationships	-
-	nodeProjection
-	relationshipProjection

Table 438. 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>
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> :	

## Graph Algorithms v3.5

```
CALL algo.graph.load(  
  'myGraph',  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  {  
    graph: 'huge',  
    direction: 'BOTH'  
  }  
)
```

## Graph Data Science v1.0

```
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'  
    }  
  }  
)
```

Undirected Native Projection:

```
CALL algo.graph.load(  
  'myGraph',  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  {  
    graph: 'huge',  
    undirected: true  
  }  
)
```

```
CALL gds.graph.create(  
  'myGraph',  
  'MyLabel',  
  {  
    MY_RELATIONSHIP_TYPE: {  
      orientation: 'UNDIRECTED'  
    }  
  }  
)
```

## Graph creation - Cypher Queries

Table 439. Loading a named graph using Cypher queries:

### Graph Algorithms v3.5

Basic Cypher queries, defining source and target:

### Graph Data Science v1.0

## Graph Algorithms v3.5

```
CALL algo.graph.load(
  'myGraph',
  'MATCH (n:MyLabel)
    RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
    RETURN id(s) AS source, id(t) AS target',
  {
    graph: 'cypher'
  }
)
```

## Graph Data Science v1.0

```
CALL gds.graph.create.cypher(
  'myGraph',
  'MATCH (n:MyLabel)
    RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
    RETURN id(s) AS source, id(t) AS target'
)
```

With concurrency property and Cypher query with relationship property:

```
CALL algo.graph.load(
  'myGraph',
  'MATCH (n:MyLabel)
    RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
    RETURN
      id(s) AS source,
      id(t) AS target,
      r.myProperty AS weight',
  {
    concurrency: 4,
    graph: 'cypher'
  }
)
```

```
CALL gds.graph.create.cypher(
  'myGraph',
  'MATCH (n:MyLabel)
    RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
    RETURN
      id(s) AS source,
      id(t) AS target,
      r.myProperty AS weight',
  {
    readConcurrency: 4
  }
)
```

Parallel loading:

## Graph Algorithms v3.5

```
CALL algo.graph.load(
  'myGraph',
  'MATCH (n:MyLabel)
  WITH * SKIP $skip LIMIT $limit
  RETURN id(n) AS id',
  'MATCH (s)-[r:MY_RELATIONSHIP_TYPE]->(t)
  WITH * SKIP $skip LIMIT $limit
  RETURN
    id(s) AS source,
    id(t) AS target,
    r.myProperty AS weight',
  {
    concurrency: 4,
    graph: 'cypher'
  }
)
```

## Graph Data Science v1.0

-

## Graph listing

Table 440. Changes in the YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
name	graphName
nodes	nodeCount
relationships	relationshipCount
type	-
direction	-
-	nodeProjection <sup>[5]</sup>
-	relationshipProjection <sup>[5]</sup>
-	nodeQuery <sup>[6]</sup>
-	relationshipQuery <sup>[6]</sup>
-	degreeDistribution <sup>[7]</sup>

Table 441. Listing named graphs:

## Graph Algorithms v3.5

```
CALL algo.graph.list()
```

## Graph Data Science v1.0

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

## Graph info

Table 442. 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>[8]</sup>
-	relationshipProjection <sup>[8]</sup>
-	nodeQuery <sup>[9]</sup>
-	relationshipQuery <sup>[9]</sup>
-	degreeDistribution <sup>[10]</sup>
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[11]</sup>	-

Table 443. Viewing information about a specific named graph:

Graph Algorithms v3.5	Graph Data Science v1.0
View information for a Named graph:	
CALL algo.graph.info('myGraph')	CALL gds.graph.list('myGraph')
Check graph existence:	
CALL algo.graph.info('myGraph') YIELD exists	CALL gds.graph.exists('myGraph') YIELD exists
View graph statistics:	
CALL algo.graph.info('myGraph', true) YIELD min, max, mean, p50	CALL gds.graph.list('myGraph') YIELD degreeDistribution AS dd RETURN dd.min, dd.max, dd.mean, dd.p50

## Removing named graphs

Table 444. 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 <small>[12]</small>
-	relationshipProjection <small>[12]</small>
-	nodeQuery <small>[13]</small>
-	relationshipQuery <small>[13]</small>
-	degreeDistribution

Table 445. Removing a named graph:

Graph Algorithms v3.5	Graph Data Science v1.0
CALL algo.graph.remove('myGraph')	CALL gds.graph.drop('myGraph')

## Production-ready algorithms

- [Label Propagation](#)
- [Louvain](#)
- [Node Similarity](#)
- [PageRank](#)
- [Weakly Connected Components](#)
- [Triangle Count / Clustering Coefficient](#)

## Label Propagation

Table 446. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
iterations	maxIterations
concurrency	concurrency

Graph Algorithms v3.5	Graph Data Science v1.0
readConcurrency	readConcurrency <sup>[14]</sup>
writeConcurrency	writeConcurrency <sup>[15]</sup>
weightProperty <sup>[16]</sup>	-
-	nodeWeightProperty
-	relationshipWeightProperty
seedProperty	seedProperty
partitionProperty	-
writeProperty	writeProperty <sup>[15]</sup>
write	-
graph	-

Table 447. 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
didConverge	didConverge
-	ranIterations
write	-
-	communityDistribution
-	configuration <sup>[17]</sup>
writeProperty <sup>[18]</sup>	-
weightProperty <sup>[19]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[20]</sup>	-

Table 448. Label Propagation Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Streaming over a named graph:	
<pre>CALL algo.labelPropagation.stream(null,     null, {graph: 'myGraph'}) YIELD nodeId, label</pre>	<pre>CALL gds.labelPropagation.stream('myGraph') YIELD nodeId, communityId</pre>

Streaming over a named graph using configuration for iterations and relationship weight property:

```
CALL algo.labelPropagation.stream(
  null,
  null,
  {
    graph: 'myGraph',
    iterations: 15,
    weightProperty: 'myWeightProperty'
  }
)
```

```
CALL gds.labelPropagation.stream(
  'myGraph',
  {
    maxIterations: 15,
    relationshipWeightProperty:
    'myWeightProperty'
  }
)
```

Streaming over anonymous graph:

```
CALL algo.labelPropagation.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

```
CALL gds.labelPropagation.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection:
  'MY_RELATIONSHIP_TYPE'
})
```

Streaming over anonymous graph using relationship with **REVERSE** projection:

```
CALL algo.labelPropagation.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

```
CALL gds.labelPropagation.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

Streaming over anonymous graph using two way relationships [\[22\]](#):

## Graph Algorithms v3.5

```
CALL algo.labelPropagation.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  { direction: 'BOTH' }  
)
```

## Graph Data Science v1.0

```
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'  
    }  
  }  
)
```

Table 449. Label Propagation Write Mode

## Graph Algorithms v3.5

Minimalistic write:

```
CALL algo.labelPropagation(  
  null,  
  null,  
  {  
    graph: 'myGraph',  
    writeProperty: 'myWriteProperty',  
    write: true  
  }  
)  
YIELD  
  writeMillis,  
  iterations,  
  p1,  
  writeProperty
```

## Graph Data Science v1.0

```
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
```

Write using weight properties <sup>[24]</sup>:

## Graph Algorithms v3.5

```
CALL algo.labelPropagation(  
    null,  
    null,  
    {  
        graph: 'myGraph',  
        writeProperty: 'myWriteProperty',  
        weightProperty:  
        'myRelationshipWeightProperty',  
        write: true  
    }  
)
```

## Graph Data Science v1.0

```
CALL gds.labelPropagation.write(  
    'myGraph',  
    {  
        writeProperty: 'myWriteProperty',  
        relationshipWeightProperty:  
        'myRelationshipWeightProperty',  
        nodeWeightProperty:  
        'myNodeWeightProperty'  
    }  
)
```

Memory estimation of the algorithm:

```
CALL algo.memrec(  
    'MyLabel',  
    'MY_RELATIONSHIP_TYPE',  
    'labelPropagation',  
    {  
        writeProperty: 'myWriteProperty',  
        weightProperty:  
        'myRelationshipWeightProperty',  
        write: true  
    }  
)
```

```
CALL  
gds.labelPropagation.write.estimate(  
    {  
        nodeProjection: 'MyLabel',  
        relationshipProjection:  
        'MY_RELATIONSHIP_TYPE',  
        writeProperty: 'myWriteProperty',  
        relationshipWeightProperty:  
        'myRelationshipWeightProperty',  
        nodeWeightProperty:  
        'myNodeWeightProperty'  
    }  
)
```

## Louvain

Table 450. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
levels	maxLevels
concurrency	concurrency
readConcurrency	readConcurrency <small>[25]</small>
writeConcurrency	writeConcurrency <small>[26]</small>
weightProperty	relationshipWeightProperty
seedProperty	seedProperty
innerIterations	maxIterations
includeIntermediateCommunities	includeIntermediateCommunities

Graph Algorithms v3.5	Graph Data Science v1.0
tolerance	tolerance
writeProperty	writeProperty <sup>[26]</sup>
write	-
graph	-

Table 451. 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>[27]</sup>
community	communityId <sup>[27]</sup>
communities	intermediateCommunityIds <sup>[27]</sup>
modularity	modularity <sup>[28]</sup>
modularities	modularities <sup>[28]</sup>
write	-
-	communityDistribution
-	configuration <sup>[29]</sup>
includeIntermediateCommunities <sup>[30]</sup>	-
writeProperty <sup>[30]</sup>	-
weightProperty <sup>[31]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[32]</sup>	-

Table 452. Louvain Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic streaming over named graph:	
<pre>CALL algo.beta.louvain.stream(null,     null, {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 - `maxLevels` and `maxIterations`:

```
CALL algo.beta.louvain.stream(
  null,
  null,
  {
    graph: 'myGraph',
    levels: 15,
    innerIterations: 30
  }
)
```

```
CALL gds.louvain.stream(
  'myGraph',
  {
    maxLevels: 15,
    maxIterations: 30
  }
)
```

Streaming over named graph with weight property:

```
CALL algo.beta.louvain.stream(
  null,
  null,
  {
    graph: 'myGraph',
    weightProperty: 'myWeightProperty'
  }
)
```

```
CALL gds.louvain.stream(
  'myGraph',
  {
    relationshipWeightProperty:
    'myWeightProperty'
  }
)
```

Minimalistic streaming over anonymous graph:

```
CALL algo.beta.louvain.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE'
)
```

```
CALL gds.louvain.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection:
  'MY_RELATIONSHIP_TYPE'
})
```

Streaming over anonymous graph with `REVERSE` relationship projection:

```
CALL algo.beta.louvain.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'INCOMING' }
)
```

```
CALL gds.louvain.stream({
  nodeProjection: 'MyLabel',
  relationshipProjection: {
    MY_RELATIONSHIP_TYPE: {
      orientation: 'REVERSE'
    }
  }
})
```

**Graph Algorithms v3.5****Graph Data Science v1.0**

Streaming over anonymous graph using two way relationships [\[34\]](#):

```
CALL algo.louvain.stream(
  'MyLabel',
  'MY_RELATIONSHIP_TYPE',
  { direction: 'BOTH' }
)
```

```
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'
    }
  }
})
```

Table 453. Louvain Write Mode

**Graph Algorithms v3.5**

Minimalistic write with just `writeProperty`:

```
CALL algo.beta.louvain(
  null,
  null,
  {
    graph: 'myGraph',
    writeProperty: 'myWriteProperty',
    write: true
  }
)
YIELD
  nodes,
  writeMillis,
  levels,
  iterations,
  p1,
  writeProperty
```

```
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
```

Running in `write` mode over weighted graph:

## Graph Algorithms v3.5

```
CALL algo.beta.louvain(  
  null,  
  null,  
  {  
    graph: 'myGraph',  
    writeProperty: 'myWriteProperty',  
    weightProperty: 'myWeightProperty',  
    write: true  
  }  
)
```

## Graph Data Science v1.0

```
CALL gds.louvain.write(  
  'myGraph',  
  {  
    writeProperty: 'myWriteProperty',  
    relationshipWeightProperty:  
    'myWeightProperty'  
  }  
)
```

Memory estimation of the algorithm:

```
CALL algo.memrec(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  'beta.louvain',  
  {  
    writeProperty: 'myWriteProperty',  
    weightProperty:  
    'myRelationshipWeightProperty',  
    write: true  
  }  
)
```

```
CALL gds.louvain.write.estimate(  
  {  
    nodeProjection: 'MyLabel',  
    relationshipProjection:  
    'MY_RELATIONSHIP_TYPE',  
    writeProperty: 'myWriteProperty',  
    relationshipWeightProperty:  
    'myWeightProperty'  
  }  
)
```

## Node Similarity

Table 454. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
concurrency	concurrency
readConcurrency	readConcurrency <sup>[35]</sup>
writeConcurrency	writeConcurrency <sup>[36]</sup>
topK	topK
bottomK	bottomK
topN	topN
bottomN	bottomN
similarityCutoff	similarityCutoff
degreeCutoff	degreeCutoff
writeProperty	writeProperty <sup>[36]</sup>

Graph Algorithms v3.5	Graph Data Science v1.0
writeRelationshipType	writeRelationshipType <sup>[36]</sup>
write	-
graph	-

Table 455. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
node1	node1 <sup>[37]</sup>
node2	node2 <sup>[37]</sup>
similarity	similarity <sup>[37]</sup>
nodesCompared	nodesCompared <sup>[38]</sup>
relationships	relationshipsWritten <sup>[38]</sup>
write	-
-	similarityDistribution
-	configuration <sup>[39]</sup>
writeProperty <sup>[40]</sup>	-
writeRelationshipType <sup>[40]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[41]</sup>	-

Table 456. Node Similarity Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic streaming over named graph:	
<pre>CALL algo.nodeSimilarity.stream(null,     null, {graph: 'myGraph'}) YIELD node1, node2, similarity</pre>	<pre>CALL gds.nodeSimilarity.stream('myGraph') YIELD node1, node2, similarity</pre>
Streaming over named graph using <b>topK</b> and <b>similarityCutoff</b> configuration properties:	

## Graph Algorithms v3.5

```
CALL algo.nodeSimilarity.stream(  
  null,  
  null,  
  {  
    graph: 'myGraph',  
    topK: 1,  
    similarityCutoff: 0.5  
  }  
)
```

## Graph Data Science v1.0

```
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>[43]</sup>:

## Graph Algorithms v3.5

```
CALL algo.nodeSimilarity.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  { direction: 'BOTH' }  
)
```

## Graph Data Science v1.0

```
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'  
    }  
  }  
)
```

Table 457. Node Similarity Write Mode

## Graph Algorithms v3.5

Minimalistic `write` with `writeRelationshipType` and `writeProperty`:

```
CALL algo.nodeSimilarity(  
  null,  
  null,  
  {  
    graph: 'myGraph',  
    writeRelationshipType:  
    'MY_WRITE_REL_TYPE',  
    writeProperty: 'myWriteProperty',  
    write: true  
  }  
)  
YIELD  
  nodesCompared,  
  relationships,  
  writeMillis,  
  iterations,  
  p1,  
  writeProperty
```

## Graph Data Science v1.0

```
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
```

Memory estimation of the algorithm:

## Graph Algorithms v3.5

```
CALL algo.memrec(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  'nodeSimilarity',  
  {  
    writeRelationshipType:  
    'MY_WRITE_REL_TYPE',  
    writeProperty: 'myWriteProperty',  
    write: true  
  }  
)
```

## Graph Data Science v1.0

```
CALL gds.nodeSimilarity.write.estimate(  
  {  
    nodeProjection: 'MyLabel',  
    relationshipProjection:  
    'MY_RELATIONSHIP_TYPE',  
    writeRelationshipType:  
    'MY_WRITE_REL_TYPE',  
    writeProperty: 'myWriteProperty'  
  }  
)
```

## PageRank

Table 458. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
iterations	maxIterations
tolerance	tolerance
dampingFactor	dampingFactor
concurrency	concurrency
readConcurrency	readConcurrency <sup>[44]</sup>
writeConcurrency	writeConcurrency <sup>[45]</sup>
writeProperty	writeProperty <sup>[45]</sup>
weightProperty	relationshipWeightProperty
write	-
graph	-

Table 459. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
node	nodeId <sup>[46]</sup>
score	score <sup>[46]</sup>
nodes	nodePropertiesWritten <sup>[47]</sup>
iterations	ranIterations

Graph Algorithms v3.5	Graph Data Science v1.0
<code>write</code>	-
-	<code>configuration</code> <sup>[48]</sup>
<code>writeProperty</code> <sup>[49]</sup>	-
<code>dampingFactor</code> <sup>[49]</sup>	-
<code>tolerance</code> <sup>[49]</sup>	-
<code>weightProperty</code> <sup>[50]</sup>	-

Table 460. PageRank Stream Mode

Graph Algorithms v3.5	Graph Data Science v1.0
Minimalistic stream over named graph:	
<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 projection:	

## Graph Algorithms v3.5

```
CALL algo.pageRank.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  { direction: 'INCOMING' }  
)
```

## Graph Data Science v1.0

```
CALL gds.pageRank.stream({  
  nodeProjection: 'MyLabel',  
  relationshipProjection: {  
    MY_RELATIONSHIP_TYPE: {  
      orientation: 'REVERSE'  
    }  
  }  
})
```

Streaming over anonymous graph with relationship weight property, assigning it a default value in case the property doesn't have value:

```
CALL algo.pageRank.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  {  
    weightProperty: 'myWeightProperty',  
    defaultValue: 1.5  
  }  
)
```

```
CALL gds.pageRank.stream({  
  nodeProjection: 'MyLabel',  
  relationshipProjection: {  
    MY_RELATIONSHIP_TYPE: {  
      properties: {  
        myWeightProperty: {  
          defaultValue: 1.5  
        }  
      }  
    }  
  }  
})
```

Table 461. PageRank Write Mode

## Graph Algorithms v3.5

Running [write](#) mode on named graph:

## Graph Data Science v1.0

## Graph Algorithms v3.5

```
CALL algo.pageRank(  
    null,  
    null,  
    {  
        graph: 'myGraph',  
        writeProperty: 'myWriteProperty',  
        write: true  
    }  
)  
YIELD  
    nodes,  
    loadMillis,  
    iterations,  
    p1,  
    writeProperty
```

## Graph Data Science v1.0

```
CALL gds.pageRank.write(  
    'myGraph',  
    {  
        writeProperty: 'myWriteProperty'  
    }  
)  
YIELD  
    nodePropertiesWritten,  
    createMillis,  
    ranIterations,  
    configuration AS conf  
RETURN  
    nodePropertiesWritten,  
    writeMillis,  
    ranIterations,  
    conf.writeProperty AS writeProperty
```

Memory estimation of the algorithm:

```
CALL algo.memrec(  
    'MyLabel',  
    'MY_RELATIONSHIP_TYPE',  
    'pageRank',  
    {  
        writeProperty: 'myWriteProperty',  
        write: true  
    }  
)
```

```
CALL gds.pageRank.write.estimate(  
    {  
        nodeProjection: 'MyLabel',  
        relationshipProjection:  
        'MY_RELATIONSHIP_TYPE',  
        writeProperty: 'myWriteProperty'  
    }  
)
```

## Weakly Connected Components

Table 462. Changes in Configuration

Graph Algorithms v3.5	Graph Data Science v1.0
direction	-
concurrency	concurrency
readConcurrency	readConcurrency <small>[51]</small>
writeConcurrency	writeConcurrency <small>[52]</small>
writeProperty	writeProperty <small>[52]</small>
weightProperty	relationshipWeightProperty
defaultValue	defaultValue
seedProperty	seedProperty

Graph Algorithms v3.5	Graph Data Science v1.0
threshold	threshold
consecutiveIds	consecutiveIds
write	-
graph	-

Table 463. Changes in YIELD fields

Graph Algorithms v3.5	Graph Data Science v1.0
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
postProcessingMillis	postProcessingMillis
nodeId	nodeId <sup>[53]</sup>
setId	componentId <sup>[53]</sup>
nodes	nodePropertiesWritten <sup>[54]</sup>
-	relationshipPropertiesWritten <sup>[54]</sup>
write	-
-	componentDistribution
-	configuration <sup>[55]</sup>
writeProperty <sup>[56]</sup>	-
weightProperty <sup>[57]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999 <sup>[58]</sup>	-

Table 464. 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:	

## Graph Algorithms v3.5

```
CALL algo.unionFind.stream(  
  null,  
  null,  
  {  
    graph: 'myGraph',  
    weightProperty: 'myWeightProperty'  
  }  
)
```

## Graph Data Science v1.0

```
CALL gds.wcc.stream(  
  'myGraph',  
  {  
    relationshipWeightProperty:  
    'myWeightProperty'  
  }  
)
```

Minimalistic streaming over anonymous graph:

```
CALL algo.unionFind.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE'  
)
```

```
CALL gds.wcc.stream({  
  nodeProjection: 'MyLabel',  
  relationshipProjection:  
  'MY_RELATIONSHIP_TYPE'  
})
```

Streaming over anonymous graph with **REVERSE** relationship projection:

```
CALL algo.unionFind.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  { direction: 'INCOMING' }  
)
```

```
CALL gds.wcc.stream({  
  nodeProjection: 'MyLabel',  
  relationshipProjection: {  
    MY_RELATIONSHIP_TYPE: {  
      orientation: 'REVERSE'  
    }  
  }  
})
```

Streaming over anonymous graph with relationship specifying default value for the weight property:

## Graph Algorithms v3.5

```
CALL algo.unionFind.stream(  
  'MyLabel',  
  'MY_RELATIONSHIP_TYPE',  
  {  
    graph: 'myGraph',  
    weightProperty: 'myWeightProperty',  
    defaultValue: 2.0  
  }  
)
```

## Graph Data Science v1.0

```
CALL gds.wcc.stream({  
  nodeProjection: 'MyLabel',  
  relationshipProjection: {  
    MY_RELATIONSHIP_TYPE: {  
      properties: {  
        myWeightProperty: {  
          defaultValue: 2  
        }  
      }  
    }  
  }  
})
```

Table 465. Weakly Connected Components Write Mode

## Graph Algorithms v3.5

Minimalistic `write` mode:

```
CALL algo.unionFind(  
  null,  
  null,  
  {  
    graph: 'myGraph',  
    writeProperty: 'myWriteProperty',  
    write: true  
  }  
)  
YIELD  
  nodes,  
  loadMillis,  
  p1,  
  writeProperty
```

## Graph Data Science v1.0

```
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
```

Running `write` mode over weighted named graph:

## Graph Algorithms v3.5

```
CALL algo.unionFind(  
    null,  
    null,  
    {  
        graph: 'myGraph',  
        writeProperty: 'myWriteProperty',  
        weightProperty: 'myWeightProperty',  
        write: true  
    }  
)
```

## Graph Data Science v1.0

```
CALL gds.wcc.write(  
    'myGraph',  
    {  
        writeProperty: 'myWriteProperty',  
        relationshipWeightProperty:  
        'myWeightProperty'  
    }  
)
```

Memory estimation of the algorithm:

```
CALL algo.memrec(  
    'MyLabel',  
    'MY_RELATIONSHIP_TYPE',  
    'unionFind',  
    {  
        writeProperty: 'myWriteProperty',  
        weightProperty:  
        'myRelationshipWeightProperty',  
        write: true  
    }  
)
```

```
CALL gds.wcc.write.estimate(  
    {  
        nodeProjection: 'MyLabel',  
        relationshipProjection:  
        'MY_RELATIONSHIP_TYPE',  
        writeProperty: 'myWriteProperty',  
        relationshipWeightProperty:  
        'myWeightProperty'  
    }  
)
```

## 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 466. Common changes in Configuration*

Graph Algorithms v3.5	Graph Data Science v1.2
<code>direction</code>	-

<b>Graph Algorithms v3.5</b>	<b>Graph Data Science v1.2</b>
concurrency	concurrency
readConcurrency	readConcurrency <sup>[59]</sup>
writeConcurrency	writeConcurrency <sup>[60]</sup>
writeProperty	writeProperty <sup>[60]</sup>
write	-
graph	-

Table 467. Changes in YIELD fields of `algo.triangleCount`

<b>Graph Algorithms v3.5</b>	<b>Graph Data Science v1.2</b>
nodeId	nodeId <sup>[61]</sup>
triangles	triangleCount <sup>[62]</sup>
triangleCount	globalTriangleCount <sup>[63]</sup>
nodeCount	nodeCount <sup>[63]</sup>
averageClusteringCoefficient <sup>[64]</sup>	-
clusteringCoefficientProperty <sup>[65]</sup>	-
loadMillis	createMillis
computeMillis	computeMillis
writeMillis	writeMillis
write	-
-	configuration <sup>[66]</sup>
writeProperty <sup>[67]</sup>	-
min, max, mean, p50, p75, p90, p95, p99, p999	-

Table 468. TriangleCount Stream Mode

<b>Graph Algorithms v3.5</b>	<b>Graph Data Science v1.2</b>
Streaming triangle counts over named graph:	
CALL algo.triangleCount.stream(null, null, {graph: 'myGraph'}) YIELD nodeId, triangles	CALL gds.triangleCount.stream('myGraph') YIELD nodeId, triangleCount
Streaming local clustering coefficients over named graph:	

## Graph Algorithms v3.5

```
CALL algo.triangleCount.stream(null,
null, {graph: 'myGraph'})
YIELD nodeId, coefficient
```

## Graph Data Science v1.2

```
CALL
gds.localClusteringCoefficient.stream('m
yGraph')
YIELD nodeId, localClusteringCoefficient
```

Streaming both triangle counts and local clustering coefficients:

```
CALL algo.triangleCount.stream(null,
null, {graph: 'myGraph'})
YIELD nodeId, triangles, coefficient
```

```
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
```

Streaming triangle counts over anonymous graph:

```
CALL algo.triangleCount.stream(
'MyLabel',
'MY_RELATIONSHIP_TYPE'
)
```

```
CALL gds.triangleCount.stream({
    nodeProjection: 'MyLabel',
    relationshipProjection: {
        MY_RELATIONSHIP_TYPE: {
            orientation: 'UNDIRECTED'
        }
    }
})
```

Table 469. TriangleCount Write Mode

## Graph Algorithms v3.5

Writing triangle counts from named graph:

## Graph Data Science v1.2

## Graph Algorithms v3.5

```
CALL algo.triangleCount(null, null, {  
    graph: 'myGraph',  
    write: true,  
    writeProperty: 'tc'  
}) YIELD nodeCount, triangleCount
```

## Graph Data Science v1.2

```
CALL gds.triangleCount.write('myGraph',  
{  
    writeProperty: 'tc'  
}) YIELD nodeCount, globalTriangleCount
```

Writing local clustering coefficients from named graph:

```
CALL algo.triangleCount(null, null, {  
    graph: 'myGraph',  
    write: true,  
    clusteringCoefficientProperty: 'lcc'  
}) YIELD nodeCount,  
averageClusteringCoefficient
```

```
CALL  
gds.localClusteringCoefficient.write('my  
Graph', {  
    writeProperty: 'lcc'  
}) YIELD nodeCount,  
averageClusteringCoefficient
```

Writing both triangle counts and local clustering coefficients:

```
CALL algo.triangleCount(null, null, {  
    graph: 'myGraph',  
    write: true,  
    writeProperty: 'tc',  
    clusteringCoefficientProperty: 'lcc'  
}) YIELD nodeCount, triangleCount,  
averageClusteringCoefficient
```

```
CALL gds.triangleCount.mutate('myGraph',  
{  
    mutateProperty: 'tc'  
}) YIELD globalTriangleCount  
CALL  
gds.localClusteringCoefficient.write('my  
Graph', {  
    triangleCountProperty: 'tc',  
    writeProperty: 'lcc'  
}) YIELD nodeCount,  
averageClusteringCoefficient  
CALL  
gds.graph.writeNodeProperties('myGraph',  
['tc'])  
YIELD propertiesWritten  
RETURN nodeCount, globalTriangleCount,  
averageClusteringCoefficient
```

[1] Moved to `graphConfiguration` as `nodeProjection`

[2] Moved to `graphConfiguration` as `relationshipProjection`

[3] Graph creation with Cypher queries has dedicated `gds.graph.create.cypher` procedure. There are parameters `nodeQuery` and `relationshipQuery` for anonymous graphs

[4] This behaviour can be achieved by creating two relationship projections - one with `orientation: 'NATURAL'` and one with `orientation: 'REVERSE'`. See [this example](#)

[5] Field will be `null` if a Cypher projection was used

[6] Field will be `null` unless a Cypher projection was used

- [7] Graph statistics map, i.e. min, max, percentiles, etc.
- [8] Field will be `null` if a Cypher projection was used
- [9] Field will be `null` unless a Cypher projection was used
- [10] Graph statistics map, i.e. min, max, percentiles, etc.
- [11] Inlined into `degreeDistribution`
- [12] Field will be `null` if a Cypher projection was used
- [13] Field will be `null` unless a Cypher projection was used
- [14] Only when using anonymous graph
- [15] Only for `write` mode
- [16] Can be configured separately by using `nodeWeightProperty` and `relationshipWeightProperty`
- [17] The configuration used to run the algorithm
- [18] Inlined into `configuration`
- [19] Inlined into `configuration` as `nodeWeightProperty` and/or `relationshipWeightProperty`
- [20] Inlined into `communityDistribution`
- [25] Only when using anonymous graph
- [26] Only for `write` mode
- [27] Only for `stream` mode
- [28] Only for `write` mode
- [29] The configuration used to run the algorithm
- [30] Inlined into `configuration`
- [31] Inlined into `configuration` as `relationshipWeightProperty`
- [32] Inlined into `communityDistribution`
- [35] Only when using anonymous graph
- [36] Only for `write` mode
- [37] Only for `stream` mode
- [38] Only for `write` mode
- [39] The configuration used to run the algorithm
- [40] Inlined into `configuration`
- [41] Inlined into `similarityDistribution`
- [44] Only when using anonymous graph
- [45] Only for `write` mode
- [46] Only for `stream` mode
- [47] Only for `write` mode
- [48] The configuration used to run the algorithm
- [49] Inlined into `configuration`
- [50] Inlined into `configuration` as `relationshipWeightProperty`
- [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] Inlined into `componentDistribution`
- [59] Only when using anonymous graph
- [60] Only for `write` mode
- [61] Only for `stream` mode
- [62] Only for `stream` mode
- [63] Not present in `stream` mode
- [64] Moved to `gds.localClusteringCoefficient`

[65] Moved as `writeProperty` to `gds.localClusteringCoefficient`

[66] The configuration used to run the algorithm

[67] Inlined into `configuration`