Introduction to Neo4j Graph Data Science course notes and links-

# Module 1: Neo4j GDS Overview

## Module Overview

In this module we will provide a high-level technical overview of the Neo4j Graph Data Science (GDS) library. This will serve as a foundation for upcoming courses that will go over graph, algorithm, and machine learning usage in more detail.

In this module, we will cover two core areas:

* **Product Overview** - How GDS is packaged, how to install it, and licensing considerations
* **How GDS Works and Configuration GDS** - GDS general workflow along with concurrency and memory configuration

# GDS Product Overview

## **Introduction**

In this first lesson, you will learn how Neo4j Graph Data Science (GDS) is packaged, how to install it, and some licensing considerations. It is not strictly necessary to install GDS to take data science courses on graph academy. The interactive portions of these courses integrate with a sandbox that is automatically prepared for you with GDS on the backend. Nevertheless, we wanted to start here so you understand GDS as a product.

### **GDS Plugin and Compatibility**

GDS is delivered as library and a plugin to the Neo4j Graph Database. This means that it needs to be installed as an extension in conjunction with configuration updates.

GDS also comes in both a free Community and paid Enterprise license which have important differences in regard to performance and enterprise capabilities. However, all analytics functionality, including graph algorithms and machine learning methods, are the same between both licenses.

The compatibility matrix for The GDS library vs Neo4j can be found [**here**](https://neo4j.com/docs/graph-data-science/current/installation/supported-neo4j-versions/). In general, you can count on the latest version of GDS supporting the latest version of Neo4j and vice versa, and we recommend you always upgrade to that combination.

Below we will go over the installation process and licensing. Of course, if you are using **[AuraDS](https://neo4j.com/docs/aura/aurads/" \t "_blank)**, GDS Enterprise comes prepackaged and ready to use out-of-the-box. You need not worry about installation, setup, and choosing between licenses.

### **Installation**

Of all the on-prem installations, Neo4j Desktop has the simplest process for GDS installation. We will go over how to install GDS there first. Overall, if you plan on testing GDS locally on your desktop, Neo4j Desktop is usually the easiest place to start.

Once you install and open Neo4j Desktop, you will find GDS in the **Plugins** tab of a database:

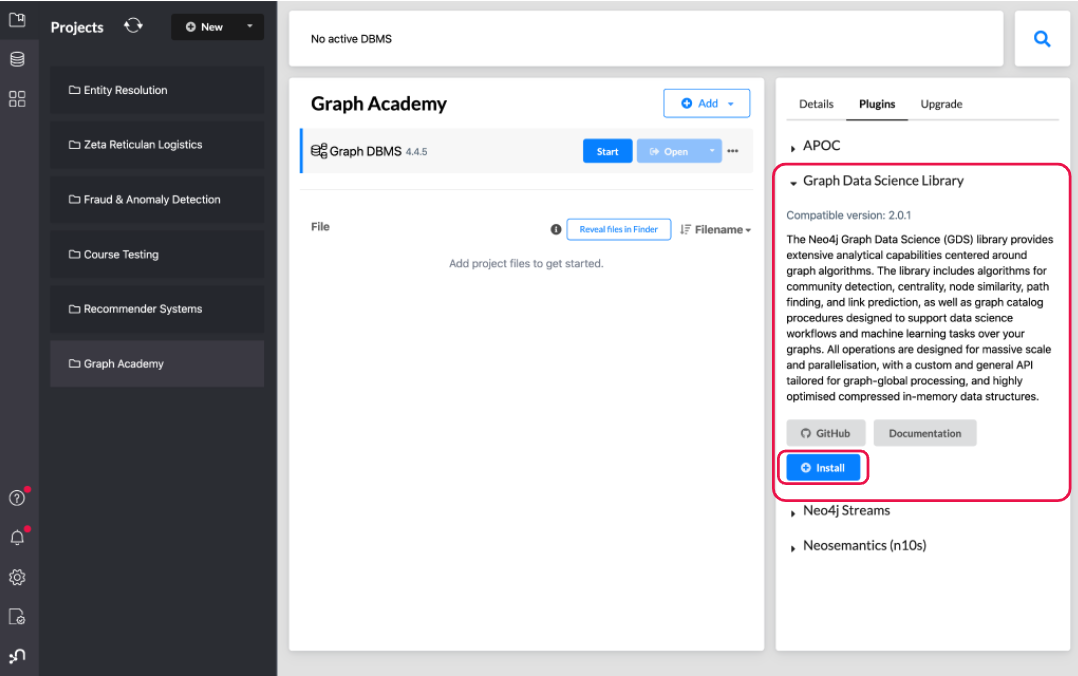


Figure 1. GDS installation on Neo4j Desktop

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:

Copy to Clipboard

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

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

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

Copy to Clipboard

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

In Neo4j Desktop, at least in recent versions, this configuration should be disabled and/or included by default.

For GDS installation on other Neo4j deployment types, including standalone server, docker, and causal cluster, please see the [**Installation documentation**](https://neo4j.com/docs/graph-data-science/current/installation/). The steps are roughly the same as desktop though they include some other considerations and certain aspects may not be fully automated. For example, in Neo4j server, you need to get the plugin from the download center, put it the correct directory location, and update the configuration manually.

### **Licensing**

GDS has both a community and enterprise license. Both have access to all the algorithms and machine learning methods, but the enterprise version has additional features that enable production use cases:

* **Enterprise features for increased performance:** unlimited concurrency to speed up compute time and access to a low-memory analytics graph format enabling the application of data science to very large graphs
* **Enterprise features for security and workflow in production:** fine-grained security, the ability to persist and publish machine learning models, in-memory graph back-up and restore, and causal cluster compatibility via read replica

You can find more information on how to obtain and install an enterprise license [**in our Enterprise Edition Configuration documentation**](https://neo4j.com/docs/graph-data-science/current/installation/installation-enterprise-edition/).

## In summary-

1. GDS (Graph Data Science) is a **plugin to the Neo4j Database**.

You can install the GDS plugin for Neo4j by downloading the plugin JAR file and placing it in the plugins directory of your Neo4j installation.

1. Community and Enterprise both have access to all the GDS algorithms and machine learning methods.

Enterprise-only features for increased performance include unlimited concurrency and a low-memory analytics graph format.

# How GDS Works

## **Introduction**

At a high-level, GDS works by transforming and loading data into an in-memory format that is optimized for high-performance graph analytics. GDS provides graph algorithms, feature engineering, and machine learning methods to execute on this in-memory graph format. This enables the efficient and scalable application of data science to large graphs including representations of entire graph databases or large portions of them.

In this lesson we will cover the high-level workflow in GDS, as well as CPU and memory configuration to support that workflow.

### **General Workflow**

Below is diagram illustrating the general workflow in GDS, which breaks out into 3 high-level steps

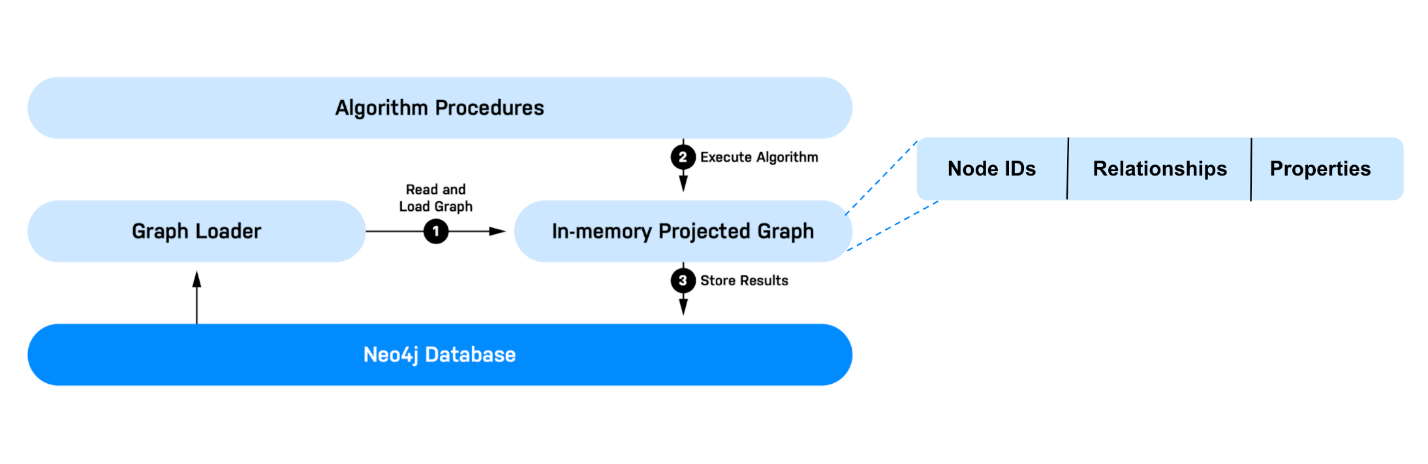


Figure 1. GDS High-Level Workflow

1. **Read and Load the Graph**: GDS needs to read data from the Neo4j database, transform it, and load it into an in-memory graph. In GDS we refer to this process as projecting a graph and refer to the in-memory graph as a graph projection. GDS can hold multiple graph projections at once and they are managed by a component called the Graph Catalog. We will go over the graph Catalog and graph projection management in more detail in the next module.
2. **Execute Algorithms**: This includes classic graph algorithms such as centrality, community detection, path finding, etc. It also includes embeddings, a form of robust graph feature engineering, as well as machine learning pipelines.
3. **Store Results**: There are a few things you may want to do with the output/result of graph algorithms. GDS enables you to write results back to the database, export to disk in csv format, or stream results into another application or downstream workflow.

### **GDS Configuration**

GDS runs greedily in respect to system resources which means it will use as much memory and CPU cores as it needs - not exceeding limits configured by the user.

If you are running in AuraDS, the GDS configuration is fully managed out-the-box, so the below information won’t be relevant to getting started. For other Neo4j deployments, however, configuring workloads and memory allocation to make best use of the available system resources is important to maximize performance.

#### **CPU and Concurrency**

GDS uses multiple CPU cores for graph projections, algorithms, and writing results. This allows GDS to parallelize its computations and significantly speed up processing time. The level of parallelization is configured per execution via the concurrency parameter in the projection, algorithm, or other operation method.

The default concurrency used for most operations in GDS is 4. 4 is also the maximum concurrency that can be used with the Community license. In GDS Enterprise, concurrency is unlimited.

#### **Memory**

GDS runs within a Neo4j instance and is therefore subject to the general Neo4j memory configuration. Below is an illustration of Neo4j memory management. Neo4j uses the Java Virtual Machine (JVM) and, as such, memory management is divided into heap and off-heap usage.

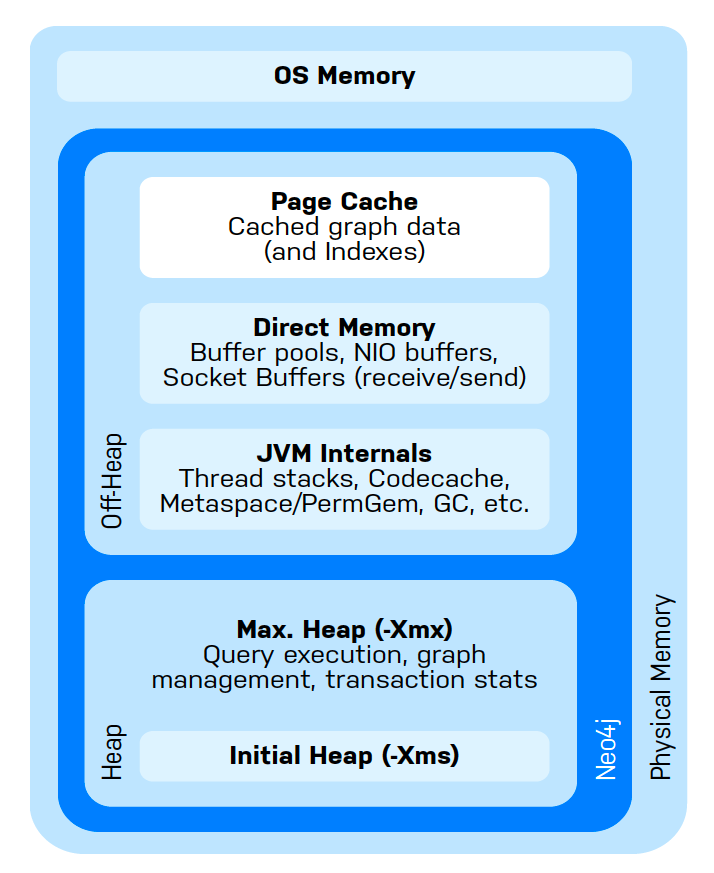


Figure 2. Neo4j Memory Management

Of the above, two main types of memory can be allocated in configuration:

* **Heap Space:** Used for storing in-memory graphs, executing GDS algorithms, query execution, and transaction state
* **Page Cache:** Used for indexes and to cache the Neo4j data stored on disk. Improves performance for querying the database and projecting graphs.

#### Recommendations for Memory Configuration

Data Science computing has a tendency to be memory intensive and GDS is no exception. In general, we recommend being generous when configuring the heap size, allocating as much heap as possible while still providing sufficient page cache to load your data and support Cypher queries. This can be done via the dbms.memory.heap.initial\_size and dbms.memory.heap.max\_size in the Neo4j configuration.

You can also use **Memory Estimation** to gauge heap size requirements early on. Memory estimation is a procedure in GDS which allows you to estimate the memory needed for running a projection, algorithm, or other operation on your data BEFORE actually executing it. We will go through the exact commands for memory estimation in our Neo4j Graph Data Science Fundamentals Course.

As far as page cache is concerned, for purely analytical workloads it is recommended to decrease page cache in favor of an increased heap size. However, setting a minimum page cache size is still important when projecting graphs. This minimum can be estimated at approximately 8KB \* 100 \* readConcurrency for standard, native, projections. Page cache size can be set via dbms.memory.pagecache.size in the Neo4j configuration.

For more information and detailed guidance on tuning these configurations please see the [**systems requirements documentation**](https://neo4j.com/docs/graph-data-science/current/installation/system-requirements/).

# Module 2: Graph Management

## Module Overview

In the previous module you learned that GDS algorithms run on a projected in-memory graph data model. In this module, you will learn how to create and manage these graph projections.

In this module, you will learn:

* how to manage projections with the Graph Catalog
* the different types of projections available in GDS
* how and when to use native projections
* how and when to use cypher projections

# Graph Catalog

## What is the Graph Catalog?

The graph catalog is a concept that allows you to manage graph projections in GDS. This includes

* creating (a.k.a projecting) graphs
* viewing details about graphs
* dropping graph projections
* exporting graph projections
* writing graph projection properties back to the database

### How the Graph Catalog Works

You can call graph catalog operations with commands of the form

**Partial**

CALL gds.graph.<command>

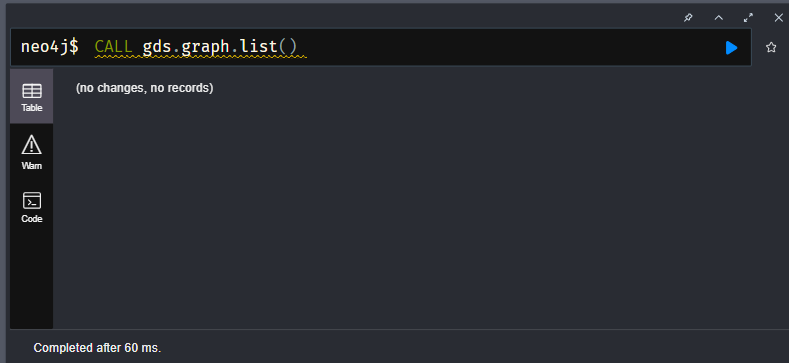


For example, we can list the graph projections that currently exist in our database with the below command.

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.list()



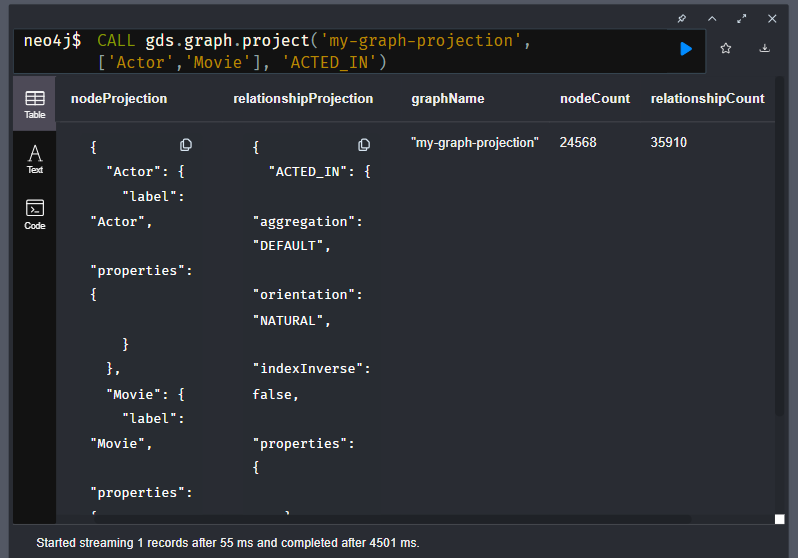
This will return an empty list since we haven’t created any projections yet.

In the recommendations graph, we can create a projection from the Actor and Movie nodes and the ACTED\_IN relationship with the below command.

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.project('my-graph-projection', ['Actor','Movie'], 'ACTED\_IN')

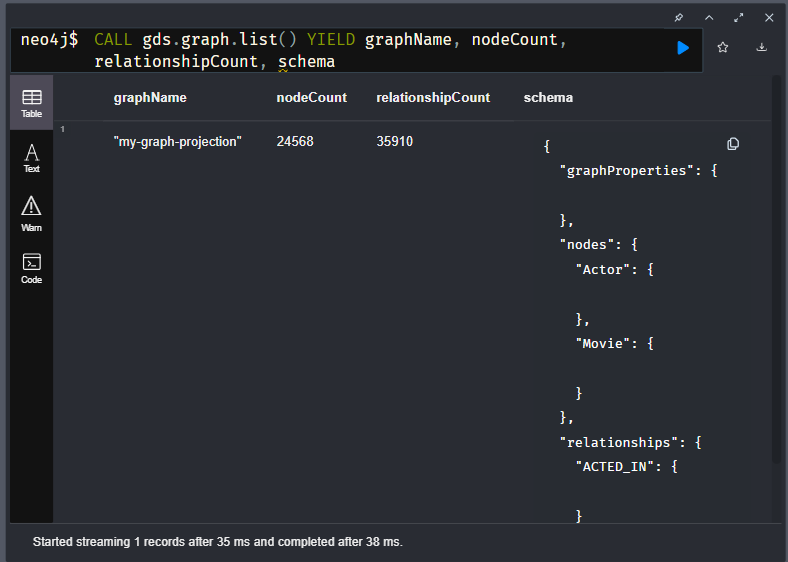


If we now list graphs again we should see information on the graph we just made

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.list() YIELD graphName, nodeCount, relationshipCount, schema



| **"graphName"** | **"nodeCount"** | **"relationshipCount"** | **"schema"** |
| --- | --- | --- | --- |
| "my-graph-projection" | 24568 | 35910 | {"relationships":{"ACTED\_IN":{}},"nodes":{"Movie":{},"Actor":{}}} |
|  |  |  |  |

### **Running Algorithms**

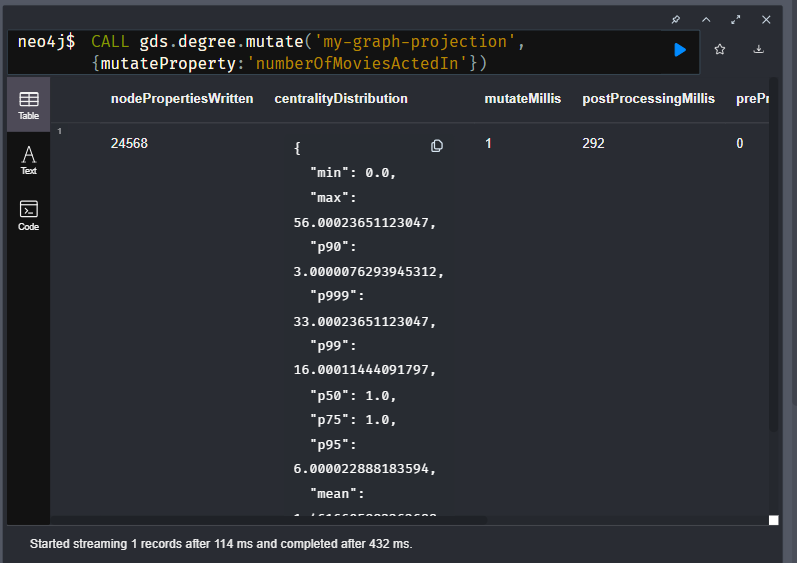
As mentioned in previous lessons, the purpose of creating a projection is to provide a space for running graph algorithms and doing graph data science efficiently.

As a simple example of a graph algorithm, we will run degree centrality on Actor nodes. We will go over the degree centrality algorithm and execution modes in the Neo4j Graph Data Science Fundamentals Course. For now, just know that this will count the number of movies each actor was in and store it on a node property called numberOfMoviesActedIn inside the projection (it will not be written back to the database yet).

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.degree.mutate('my-graph-projection', {mutateProperty:'numberOfMoviesActedIn'})



### **Streaming and Writing Node Properties**

There will be times when we want to take the results from our algorithm calculations and either stream them into another process or write them back to the database. The graph catalog has methods to stream and write both node properties and relationship properties for these purposes. We will go over this for the case of node properties below.

Using our numberOfMoviesActedIn example, we can stream the top 10 most prolific actors by movie count using the nodeProperty.stream graph catalog operation.

**cypher**

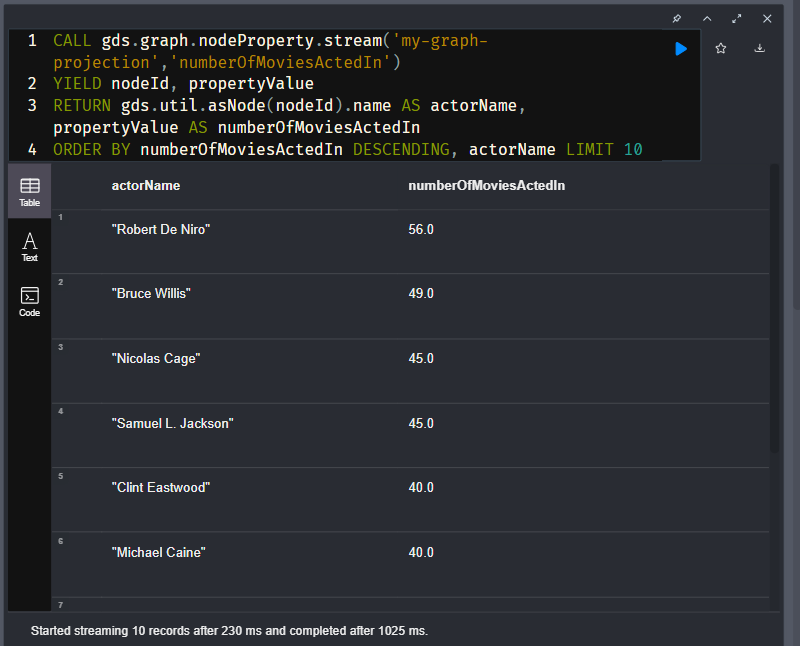
Copy to ClipboardRun in Sandbox

CALL gds.graph.nodeProperty.stream('my-graph-projection','numberOfMoviesActedIn')

YIELD nodeId, propertyValue

RETURN gds.util.asNode(nodeId).name AS actorName, propertyValue AS numberOfMoviesActedIn

ORDER BY numberOfMoviesActedIn DESCENDING, actorName LIMIT 10

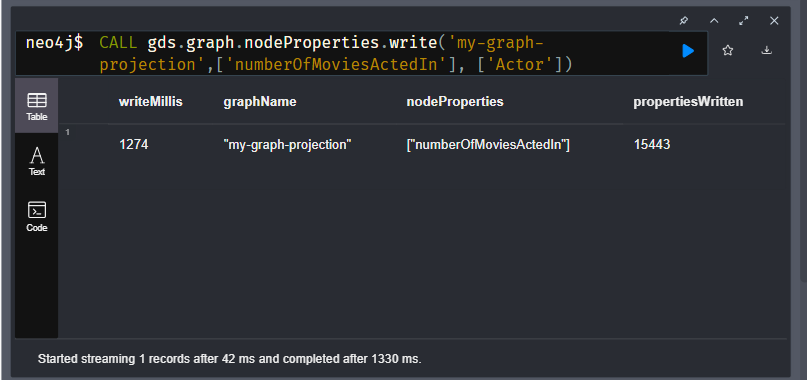


If we instead wanted to write the property back to the database we could use the nodeProperties.write operation.

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.nodeProperties.write('my-graph-projection',['numberOfMoviesActedIn'], ['Actor'])



We could then query the top 10 most prolific actors by movie count with Cypher.

**cypher**

Copy to ClipboardRun in Sandbox

MATCH (a:Actor)

RETURN a.name, a.numberOfMoviesActedIn

ORDER BY a.numberOfMoviesActedIn DESCENDING, a.name LIMIT 10



### **Exporting Graphs**

In a data science workflow, you may encounter situations where you need to bulk export data from a graph projection after performing graph algorithms and other analytics. For example, you may want to:

* export graph features for training a machine learning model in another environment
* create separate analytical views for downstream analytics and/or sharing with colleagues.
* produce snapshots of analytical results and persist to the filesystem

The graph catalog has two methods for export:

1. gds.graph.export to export a graph into a new database - effectively copying the projection into a separate Neo4j database
2. gds.beta.graph.export.csv to export a graph to csv files

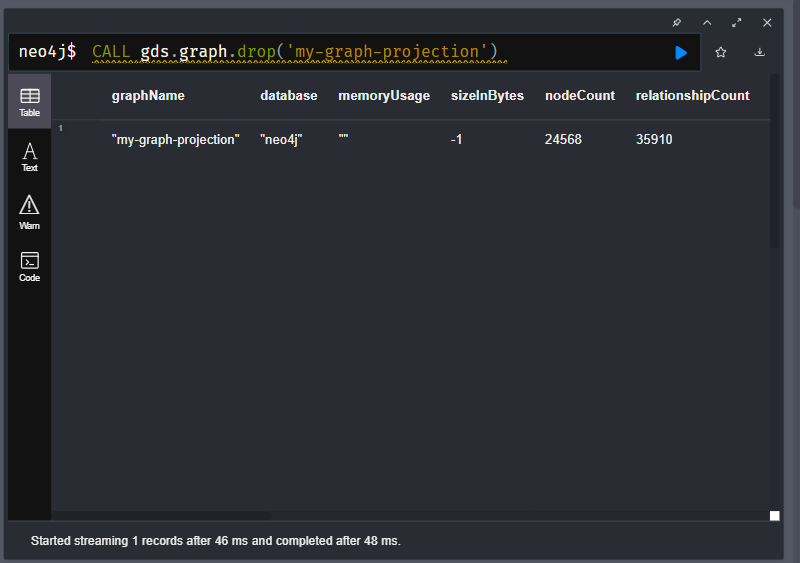
### Dropping Graphs

Projected graphs take up space in memory so once we are done working with a graph projection it is smart to remove it. We can do this with the drop command below:

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.drop('my-graph-projection')

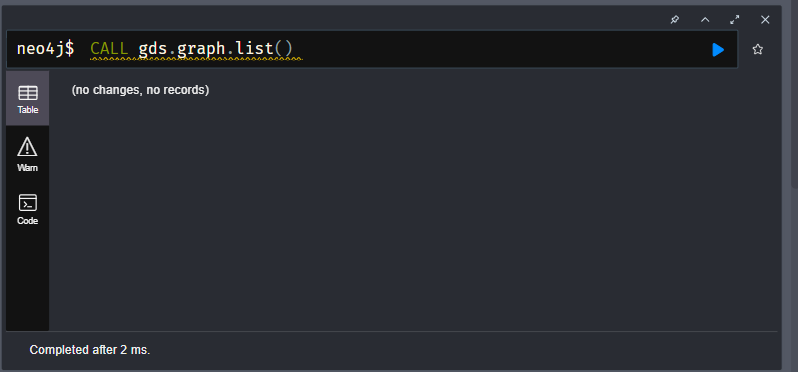


Now when we list graphs it will be empty again.

**cypher**

Copy to ClipboardRun in Sandbox

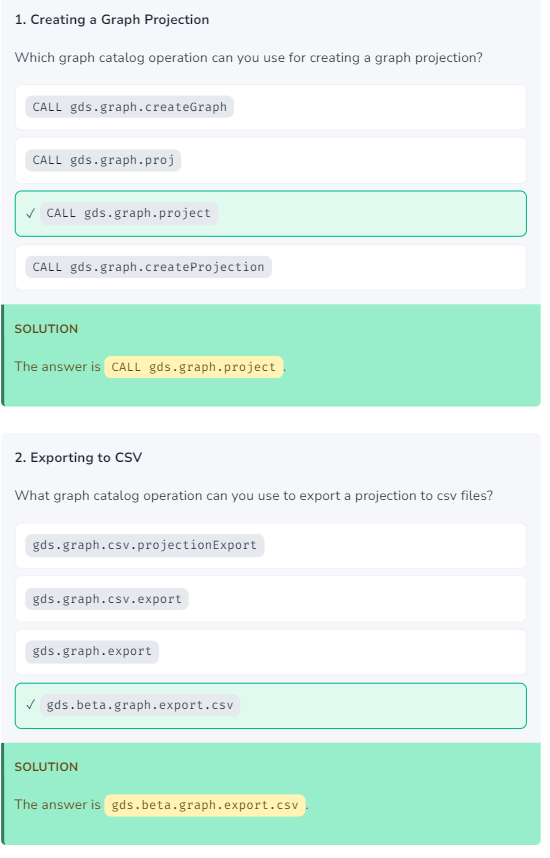
CALL gds.graph.list()

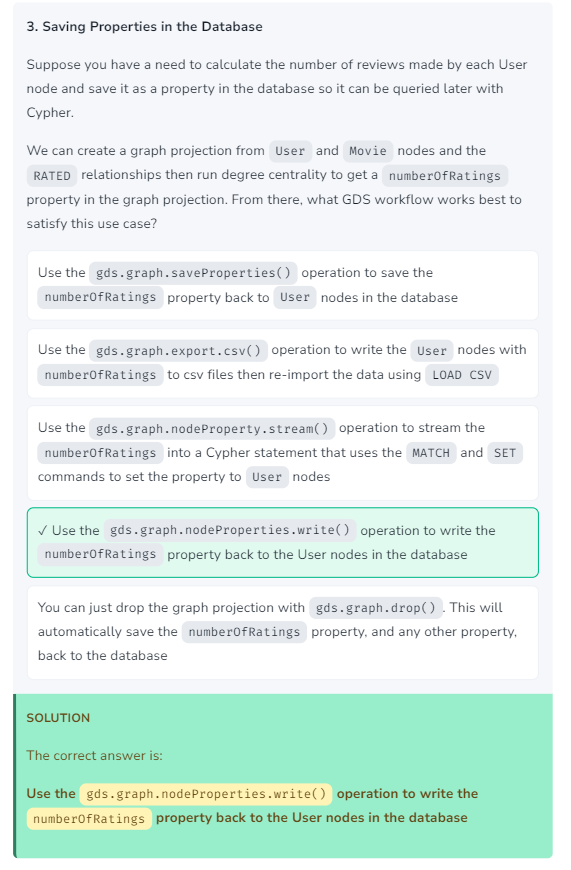
****

### **Other Graph Catalog Operations**

There are a few other management operations in the graph catalog that we will not be going over in detail inside this module such as subsetting projections (a.k.a subgraph projections), delete, and remove operations. You can read about all of them in our [**Graph Catalog documentation**](https://neo4j.com/docs/graph-data-science/current/management-ops/graph-catalog-ops/).

### Questions and Answers-





# Native Projections

## Introduction

In the last lesson you learned about the graph catalog. We briefly introduced graph projections there, but we didn’t go into much depth. Understanding Graph projections is foundational to success in GDS, so we will spend the next two lessons covering them in more detail.

There are 2 primary types of projections in GDS, native projections and cypher projections. In summary, native projections are optimized for efficiency and performance to support graph data science at scale. Cypher projections are optimized for flexibility and customization to support exploratory analysis, experimentation, and smaller graph projections.

In this lesson we will cover native projections specifically - what they are and how to use them. In the next lesson we will do the same for Cypher projections.

### **About Native Projections**

We actually used a native projection in our last lesson. When you call gds.graph.project() you are using a native projection. Native projections provide the best performance by reading from the Neo4j store files directly. We recommend them for both development and production phases.

In addition to just projecting node and relationship elements as-is from the database, native projections offer a variety of other features. Below are a few of the big ones:

* the inclusion of numeric node and relationship properties
* altering relationship direction or "orientation"
* aggregating parallel relationships

These options help prepare the projection for different types of analytical workflows and algorithms.

Below we cover the basic syntax for native projections and walk through a couple common configurations.

### **Basic Syntax**

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

| **Name** | **Type** | **Optional** | **Description** |
| --- | --- | --- | --- |
| graphName | String | no | The name under which the graph is stored in the catalog. |
| nodeProjection | String, List or Map | no | The configuration for projecting nodes. |
| relationshipProjection | String, List or Map | no | The configuration for projecting relationships. |
| configuration | Map | yes | Additional parameters to configure the native projection. |

There are multiple different options for the nodeProjection and relationshipProjection. To introduce the basics it is helpful to walk through by use case.

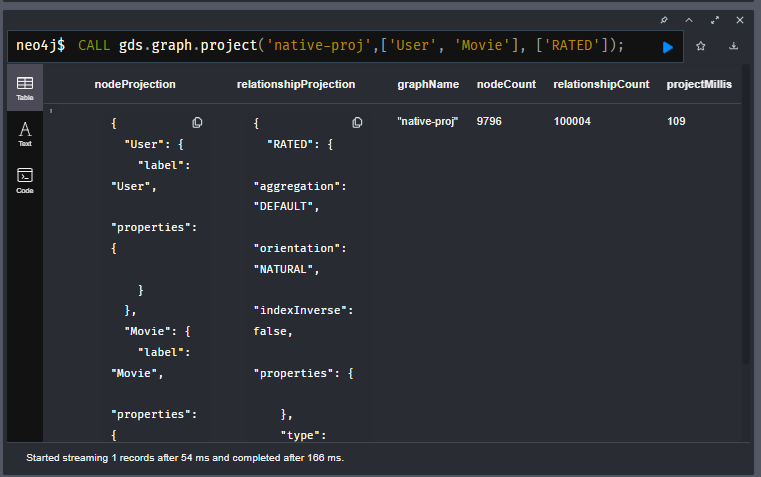
### Basic Native Projections

Let’s first consider the very basic scenario where we want to project nodes and relationships as-is without any properties. You can use a list-like syntax for both the node labels and relationships you want to include. Take the below example where we project the User and Movie nodes with the RATED relationship. This type of projection is very common for graph data science based Recommendation Systems as it supports variations of Implicit Collaborative Filtering - a memory based approach to recommendation.

**cypher**

Run in Sandbox

CALL gds.graph.project('native-proj',['User', 'Movie'], ['RATED']);

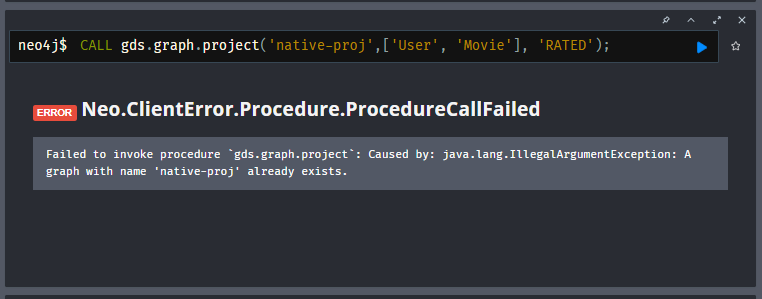


There are various forms of shorthand syntax too. For example, if you plan to include only one node label or relationship type you can just use a single string value. We could for example just enter the value RATED for the relationshipProjection and get an equivalent projection.

**cypher**

Run in Sandbox

CALL gds.graph.project('native-proj',['User', 'Movie'], 'RATED');



**A GRAPH WITH NAME 'NATIVE-PROJ' ALREADY EXISTS.**

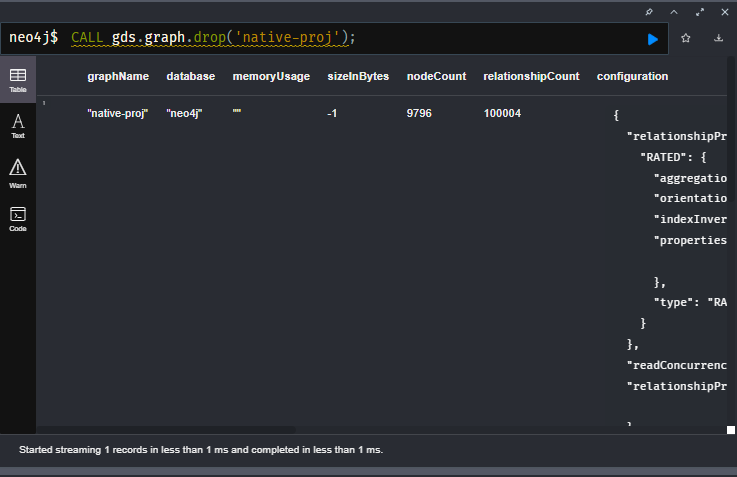
If you attempt to create a new graph projection with a name that already exists, you will receive an error. To continue you will first have to run the gds.graph.drop() procedure to drop the existing graph projection.

**cypher**

**Dropping a Graph**

Copy to ClipboardRun in Sandbox

CALL gds.graph.drop('native-proj');

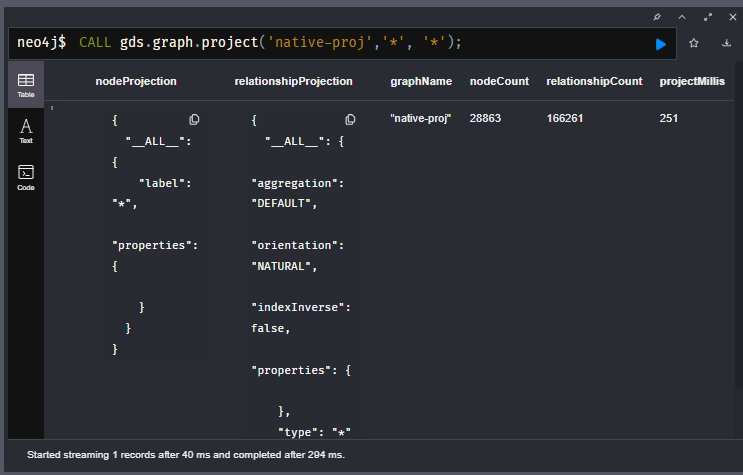


The wildcard character '\*' can be used to include all nodes and/or relationships in the database. The below projections all nodes and relationships.

**cypher**

Run in Sandbox

CALL gds.graph.project('native-proj','\*', '\*');



### **Changing Relationship Orientation**

Native projections allow you to change the relationship orientation as well. To best describe the concept of orientation and why we would want to change it, we need to cover the difference between a directed and an undirected relationship.

A directed relationship is non-symmetrical. It goes from a source node to a target node, illustrated by the image below. This type of relationship may contain additional qualifying properties, for example a weighting or strength indicator.

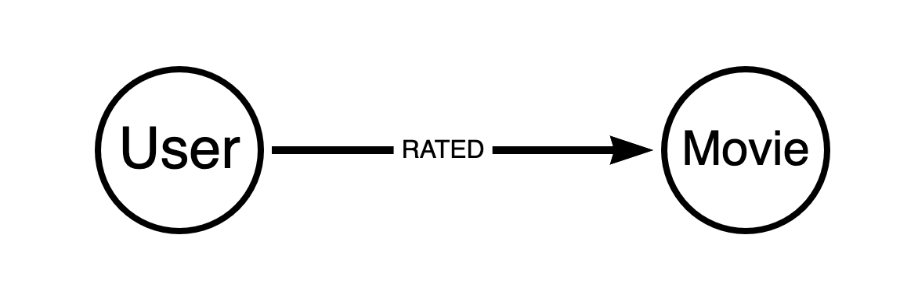


Figure 1. Directed Relationship

An undirected relationship is symmetric with no directional character, it is simply between two nodes instead of having a source and target.

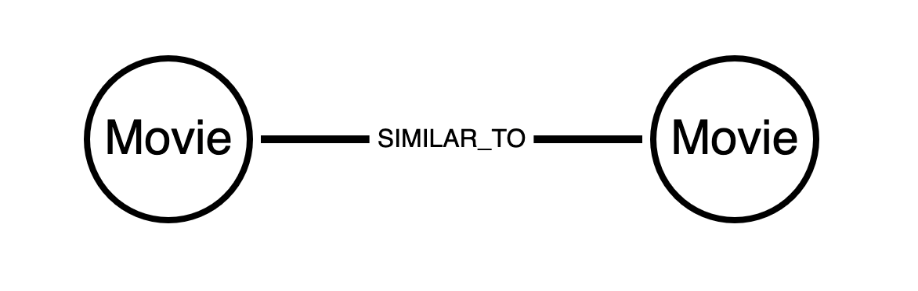


Figure 2. Undirected Relationship

Every relationship in the neo4j database is directed by design. However, some graph algorithms are designed to work on undirected relationships. Other algorithms are directed, but we may need to reverse the direction of the relationship in the database to get the analytic we want.

To accommodate this there are three orientation options we can apply to relationship types in the relationshipProjection:

* NATURAL: same direction as in the database (default)
* REVERSE: opposite direction as in the database
* UNDIRECTED: undirected

Take the graph we just projected as an example. Say we want to count the number of user ratings each movie has. If we try to use the degree call like we did last lesson we will get all zeros.

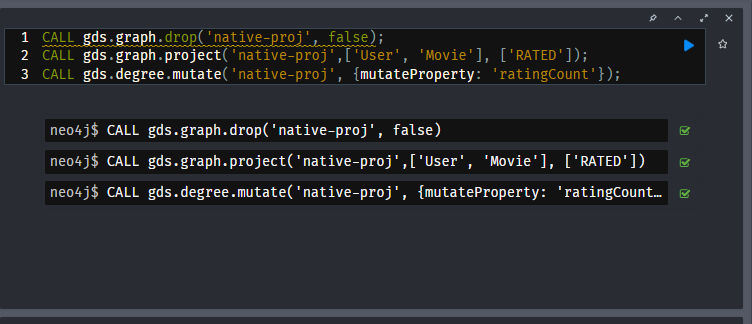
**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.drop('native-proj', false);

CALL gds.graph.project('native-proj',['User', 'Movie'], ['RATED']);

CALL gds.degree.mutate('native-proj', {mutateProperty: 'ratingCount'});



**cypher**

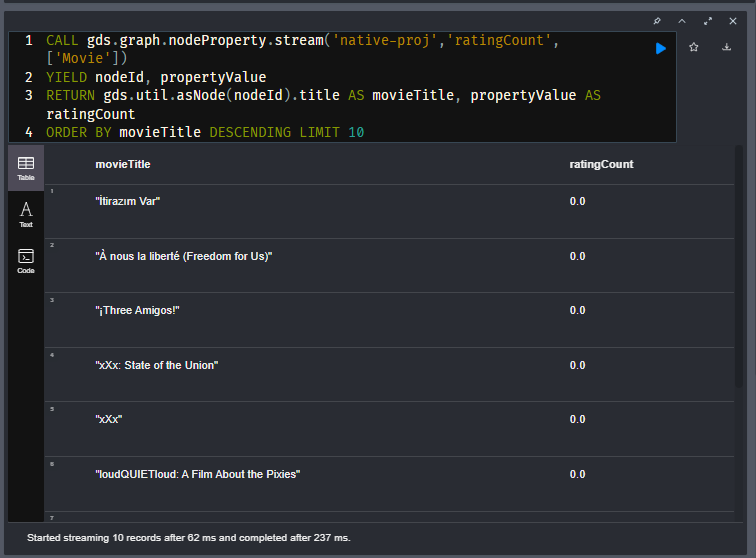
Copy to ClipboardRun in Sandbox

CALL gds.graph.nodeProperty.stream('native-proj','ratingCount', ['Movie'])

YIELD nodeId, propertyValue

RETURN gds.util.asNode(nodeId).title AS movieTitle, propertyValue AS ratingCount

ORDER BY movieTitle DESCENDING LIMIT 10



| **movieTitle** | **ratingCount** |
| --- | --- |
| İtirazım Var | 0.0 |
| À nous la liberté (Freedom for Us) | 0.0 |
| ¡Three Amigos! | 0.0 |
| xXx: State of the Union | 0.0 |
| xXx | 0.0 |

This has to do with the direction of the relationships. Let’s delete that graph and project a new one where we reverse the RATED relationship direction.

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.drop('native-proj', false);

*//replace with a project that has reversed relationship orientation*

CALL gds.graph.project(

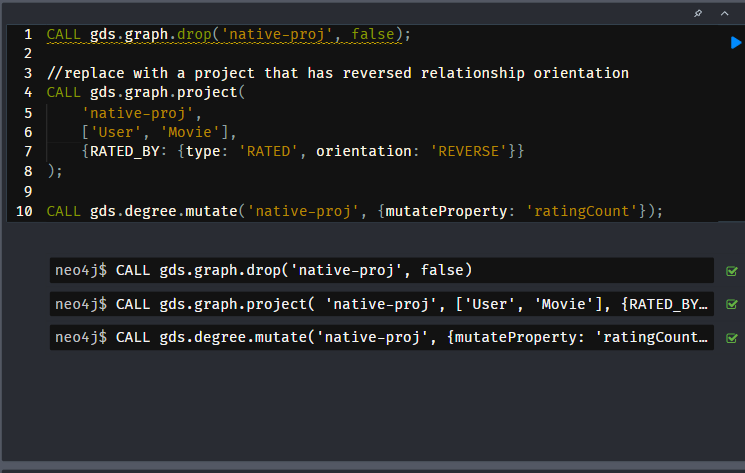
'native-proj',

['User', 'Movie'],

{RATED\_BY: {type: 'RATED', orientation: 'REVERSE'}}

);

CALL gds.degree.mutate('native-proj', {mutateProperty: 'ratingCount'});



Now when we use the degree algorithm we will get the rating counts we need.

**cypher**

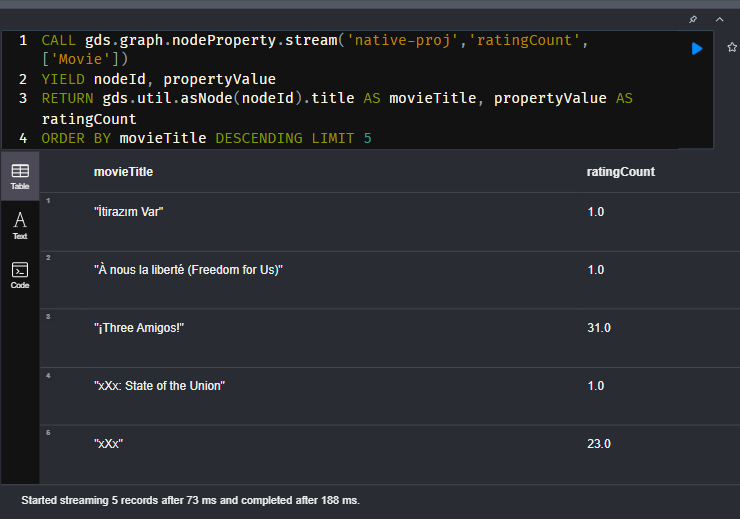
Copy to ClipboardRun in Sandbox

CALL gds.graph.nodeProperty.stream('native-proj','ratingCount', ['Movie'])

YIELD nodeId, propertyValue

RETURN gds.util.asNode(nodeId).title AS movieTitle, propertyValue AS ratingCount

ORDER BY movieTitle DESCENDING LIMIT 5



| **movieTitle** | **ratingCount** |
| --- | --- |
| İtirazım Var | 1.0 |
| À nous la liberté (Freedom for Us) | 1.0 |
| ¡Three Amigos! | 31.0 |
| xXx: State of the Union | 1.0 |
| xXx | 23.0 |
|  |  |

### **Including Node and Relationship Properties**

Node and relationship properties may be useful to consider in graph analytics. They can be used as weights in graph algorithms and features for machine learning.

Below is an example of including multiple movie node properties and the rating relationship property.

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.drop('native-proj', false);

CALL gds.graph.project(

'native-proj',

['User', 'Movie'],

{RATED: {orientation: 'UNDIRECTED'}},

{

nodeProperties:{

revenue: {defaultValue: 0}, *//* ***(1)***

budget: {defaultValue: 0},

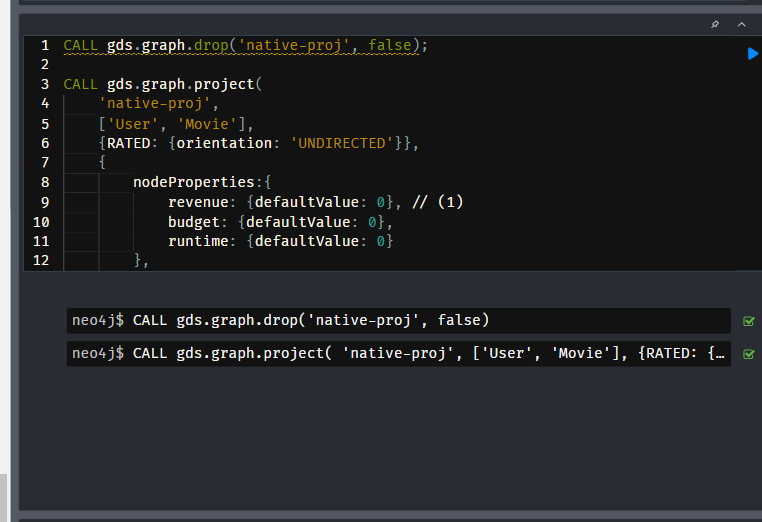
runtime: {defaultValue: 0}

},

relationshipProperties: ['rating'] *//* ***(2)***

}

);



**Notes:**

1. the defaultValue parameter allows us to fill in missing values with a default. In this case we use 0.
2. simpler syntax with no default values as these should not be missing according to the data model.

We will go over how to leverage properties like these in more detail in the Neo4j Graph Data Science Fundamentals course. There are a variety of different options for setting defaults and for alternative configurations, such as setting properties for all node labels and relationship types instead for doing so for each one separately. Please refer to the [**documentation**](https://neo4j.com/docs/graph-data-science/current/management-ops/projections/graph-project/#graph-project-native-syntax) if you want more details on these.

### Parallel Relationship Aggregations

The Neo4j database allows you to store multiple relationships of the same type and direction between two nodes. These are colloquially known as parallel relationships. For example, consider a graph of financial transaction data where users send money to one another. If a user sends money to the same user multiple times this can form multiple parallel relationships.

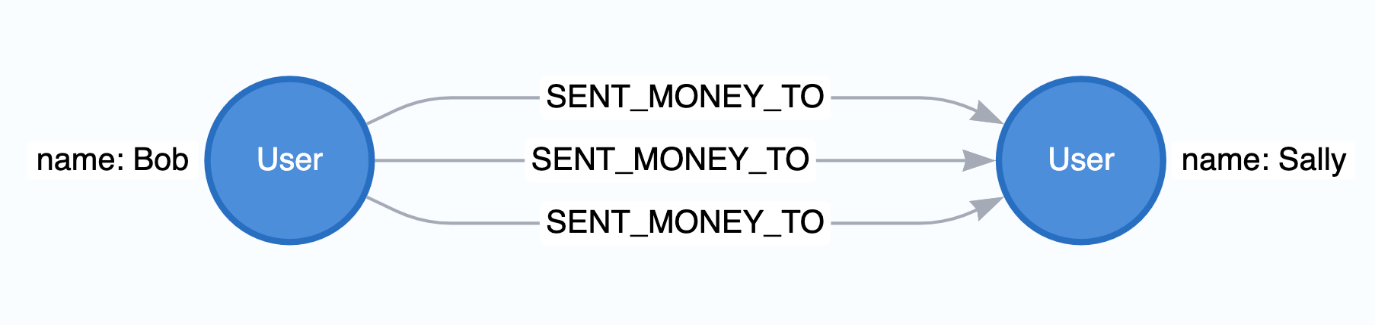


Figure 3. Nodes with Parallel Relationships

Sometimes you will want to aggregate these parallel relationships into a single relationship in preparation for running graph algorithms or machine learning. This is because graph algorithms may count each relationship between two nodes separately when all we need to consider is whether a single relationship exists between them. Other times we may want to weight the connection between two nodes higher if more parallel relationships exists, but it’s not always easy to do so without aggregating the relationships first depending on which algorithm you use.

Native projections allow for this aggregation. When you conduct relationship aggregation you can generate aggregate statistics too, such as parallel relationship counts or sums or averages of relationship properties which can then be used as weights. Below is an example of aggregating relationships without any properties

**cypher**

Copy to Clipboard

CALL gds.graph.project(

'user-proj',

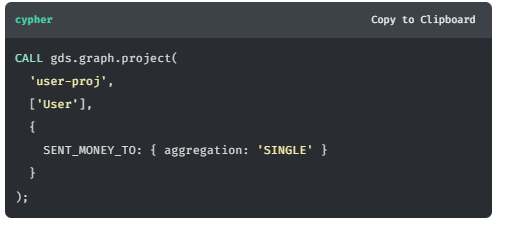
['User'],

{

SENT\_MONEY\_TO: { aggregation: 'SINGLE' }

}

);



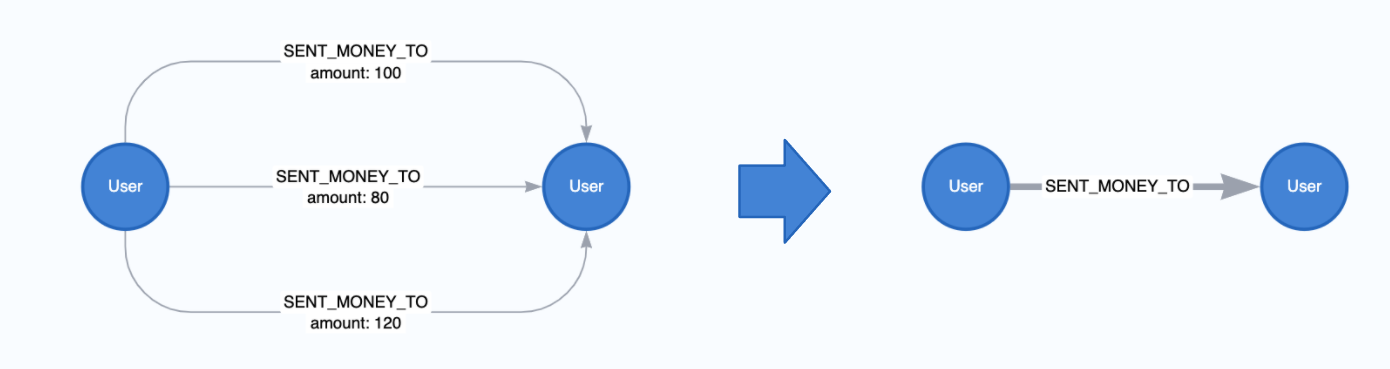


Figure 4. Aggregate the SENT\_MONEY\_TO Realtionship With no Properties

We can create a property with the count of the relationships as well - like so:

**cypher**

Copy to Clipboard

CALL gds.graph.project(

'user-proj',

['User'],

{

SENT\_MONEY\_TO: {

properties: {

numberOfTransactions: {

*// the wildcard '\*' is a placeholder, signaling that*

*// the value of the relationship property is derived*

*// and not based on Neo4j property.*

property: '\*',

aggregation: 'COUNT'

}

}

}

}

);



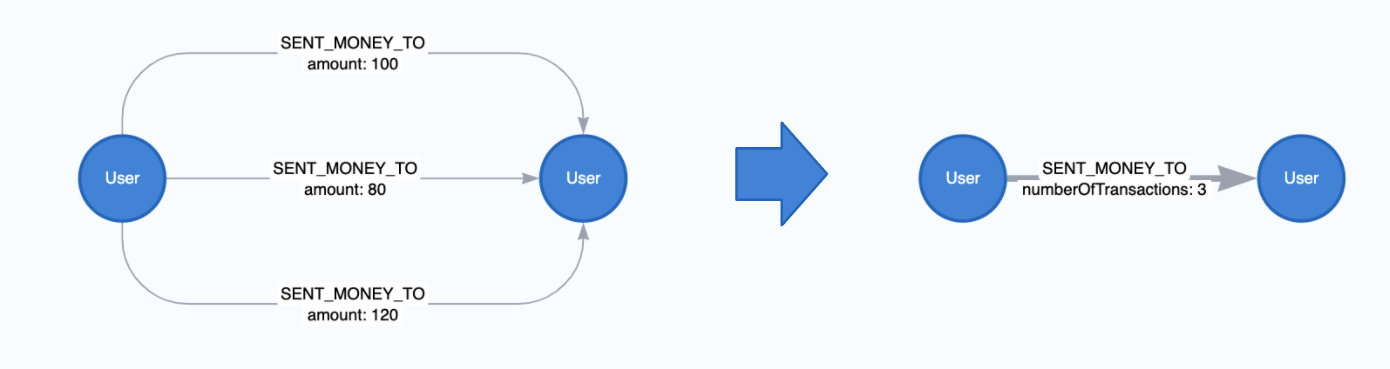


Figure 5. Aggregate the SENT\_MONEY\_TO Realtionship With a Trancastion Count

We can also take the sum, min or max of relationship properties during aggregation. Below is an example with sum.

**cypher**

Copy to Clipboard

CALL gds.graph.project(

'user-proj',

['User'],

{

SENT\_MONEY\_TO: {

properties: {

totalAmount: {

property: 'amount',

aggregation: 'SUM'

}

}

}

}

);



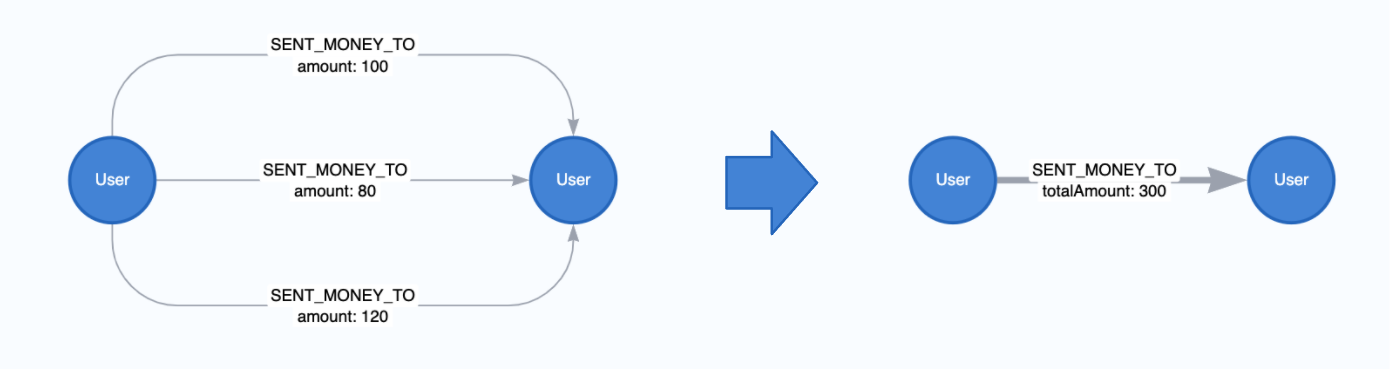


Figure 6. Aggregate the SENT\_MONEY\_TO Realtionship With a Property Sum

### **Other Native Projection Configuration and Features**

We covered the basics here but there are actually many extended syntax and configuration options available for native projections which are detailed in the [**documentation**](https://neo4j.com/docs/graph-data-science/current/graph-project/#graph-project-native-syntax). In general, if you are trying to do something in native projection and can’t quite express the thing you want with the current syntax, check the docs to see if there are additional configurations or syntax extensions to support.

## **Summary**

In this lesson we went over native projections, the primary graph projection mechanism in GDS. Native Projections are optimized for efficiency and performance to support graph data science at scale. Native projections have a rich syntax and set of configuration options that allow you to

1. filter the graph by node label and relationship types
2. include node and relationship properties
3. alter relationship orientation
4. aggregate parallel relationships

In the next lesson you will be challenged to run a Native Projection against your Neo4j Sandbox.

# Challenge: Native Projection

## Create a native graph projection

Create a native graph projection representing Users rating Movies and ensure the RATED relationship is undirected. What is the relationship count of the native projection?

Solution-

CALL gds.graph.project(

  // Name of the projection

  'user-rated-movie',

  // Labels of nodes to include in the projection

  ['User', 'Movie'],

  // Relationship types to include in the projection

  {

    RATED: {

        type: 'RATED',

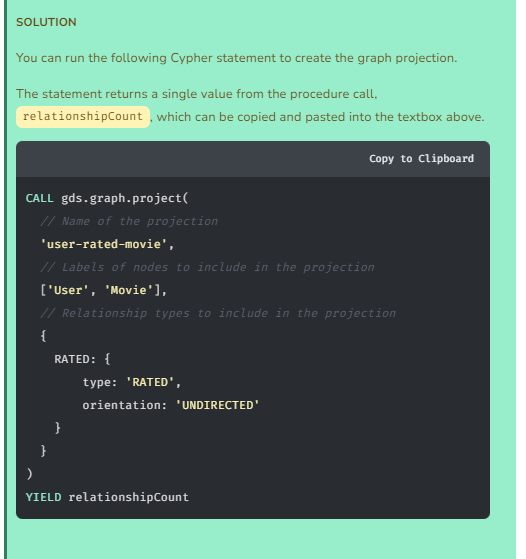
        orientation: 'UNDIRECTED'

    }

  }

)

YIELD relationshipCount



# Cypher Projections

## Introduction

While the native projection is scalable and fast, its filtering and aggregation capabilities aren’t as flexible as Cypher. The Cypher projection, as its name implies, uses Cypher to define the projection pattern, and as such, enables more flexibility.

Cypher projections are intended to be used in exploratory analysis and developmental phases where additional flexibility and/or customization is needed. They can also work in production settings where you plan to subset only a small portion of the graph, such as a relatively small community or neighborhood of nodes.

While Cypher projections offer more flexibility and customization, they have a diminished focus on performance relative to native projections and as a result won’t perform as quickly or as well on larger graphs. This is a key trade-off to keep in mind whenever you consider using Cypher projections.

In this lesson, we will go over the cypher projection syntax, an applied example, where cypher projections are useful, and common strategies for transition from Cypher to native projections as workflows mature.

### 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.

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

### Applied Example

In the last lesson we answered which actors were most prolific based on the number of movies they acted in. Suppose instead we wanted to know which actors are the most influential in terms of the number of other actors they have been in recent, high grossing, movies with.

For the sake of this example, we will call a movie “recent” if it was released on or after 1990, and high-grossing if it had revenue >= $1M.

The graph is not set up to answer this question well with a direct native projection. However, we can use a cypher projection to filter to the appropriate nodes and perform an aggregation to create an ACTED\_WITH relationship that has a actedWithCount property going directly between actor nodes.

**cypher**

Copy to ClipboardRun in Sandbox

CALL gds.graph.project.cypher(

'proj-cypher',

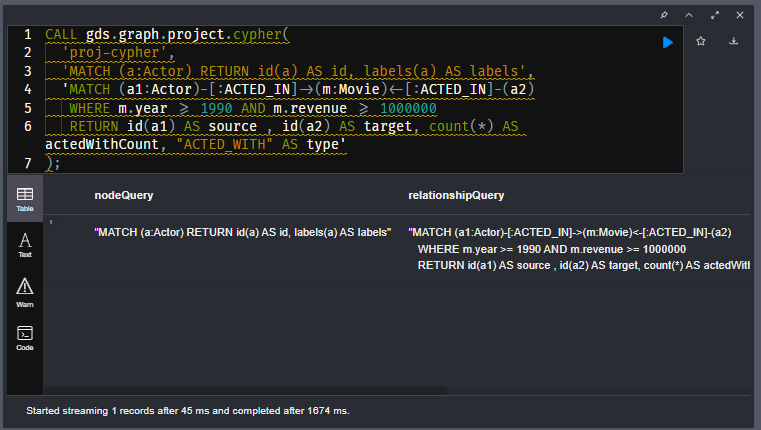
'MATCH (a:Actor) RETURN id(a) AS id, labels(a) AS labels',

'MATCH (a1:Actor)-[:ACTED\_IN]->(m:Movie)<-[:ACTED\_IN]-(a2)

WHERE m.year >= 1990 AND m.revenue >= 1000000

RETURN id(a1) AS source , id(a2) AS target, count(\*) AS actedWithCount, "ACTED\_WITH" AS type'

);



Once that is done we can apply degree centrality like we did last lesson. Except we will weight the degree centrality by actedWithCount property and also directly stream the top 10 results back. This counts how many times the actor has acted with other actors in recent, high grossing movies.

**cypher**

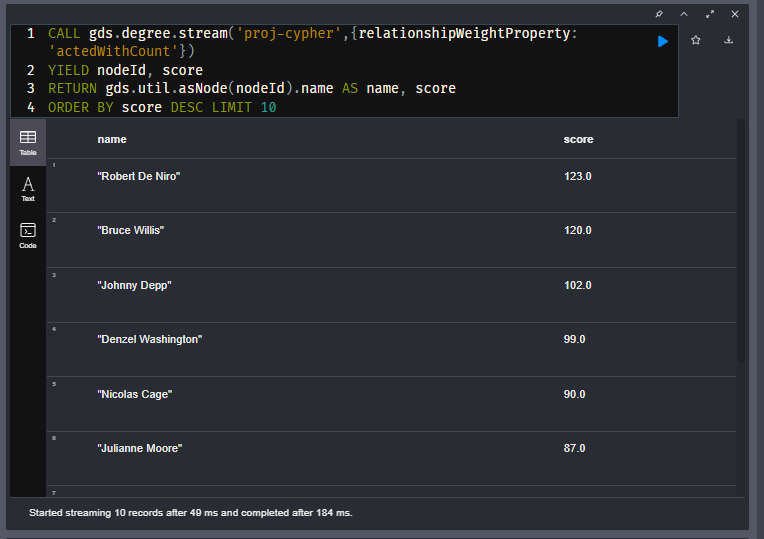
Copy to ClipboardRun in Sandbox

CALL gds.degree.stream('proj-cypher',{relationshipWeightProperty: 'actedWithCount'})

YIELD nodeId, score

RETURN gds.util.asNode(nodeId).name AS name, score

ORDER BY score DESC LIMIT 10



The results include some big name actors as we would expect.

| **name** | **score** |
| --- | --- |
| Robert De Niro | 123.0 |
| Bruce Willis | 120.0 |
| Johnny Depp | 102.0 |
| Denzel Washington | 99.0 |
| Nicolas Cage | 90.0 |
| Julianne Moore | 87.0 |
| Brad Pitt | 87.0 |
| Samuel L. Jackson | 85.0 |
| George Clooney | 84.0 |
| Morgan Freeman | 84.0 |

### When To Use Cypher Projections

In the above example, there were two things that prevented us from directly using a native projection. They also happen to be two of the most common cases for using Cypher Projections.

1. **Complex Filtering:** Using node and/or relationship property conditions or other more complex MATCH/WHERE conditions to filter the graph, rather than just node label and relationship types.
2. **Aggregating Multi-Hop Paths with Weights:** The relationship projection required aggregating the (Actor)-[ACTED\_IN]-(Movie)-[ACTED\_IN]-(Actor) pattern to a (Actor)-[ACTED\_WITH {actedWithCount}]-(Actor) pattern where the actedWithCount is a relationship weight property. This type of projection, where we need to transform multi-hop paths into an aggregated relationship that connects the source and target node, is a commonly occurring pattern in graph analytics.

There are a few other special use cases for Cypher projections too, including merging different node labels and relationship types and defining virtual relationships between nodes based on property conditions or other query logic.

### Transitioning to Native Projections

While Cypher projections are great for experimenting with these patterns and for small subsets of the graph, we encourage you to transition to native projections as workflows mature, graph projections become larger, and fast performance becomes more important.

For example, with the calculations we made above, we can instead use the following workflow which takes advantage of collapse path in a native projection. This technique does not weight the resulting relationships, so while the ranking of top actors is not exactly the same it is still very similar.

**cypher**

Copy to ClipboardRun in Sandbox

*//set a node label based on recent release and revenue conditions*

MATCH (m:Movie)

WHERE m.year >= 1990 AND m.revenue >= 1000000

SET m:RecentBigMovie;

*//native projection with reverse relationships*

CALL gds.graph.project('proj-native',

['Actor','RecentBigMovie'],

{

ACTED\_IN:{type:'ACTED\_IN'},

HAS\_ACTOR:{type:'ACTED\_IN', orientation: 'REVERSE'}

}

);

*//collapse path utility for relationship aggregation - no weight property*

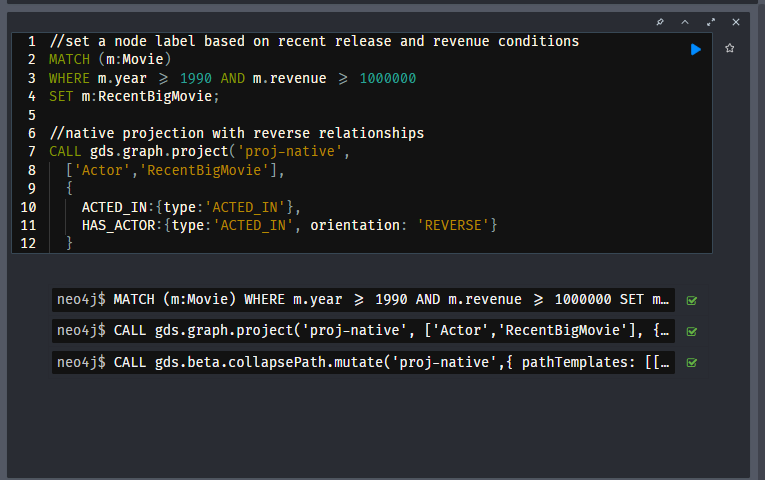
CALL gds.beta.collapsePath.mutate('proj-native',{

pathTemplates: [['ACTED\_IN', 'HAS\_ACTOR']],

allowSelfLoops: false,

mutateRelationshipType: 'ACTED\_WITH'

});



**cypher**

Copy to ClipboardRun in Sandbox

*//count actors that acted with the most other actors in recent high grossing movies and stream the top 15*

CALL gds.degree.stream('proj-native', {nodeLabels:['Actor'], relationshipTypes: ['ACTED\_WITH']})

YIELD nodeId, score

RETURN gds.util.asNode(nodeId).name AS name, score

ORDER BY score DESC LIMIT 15



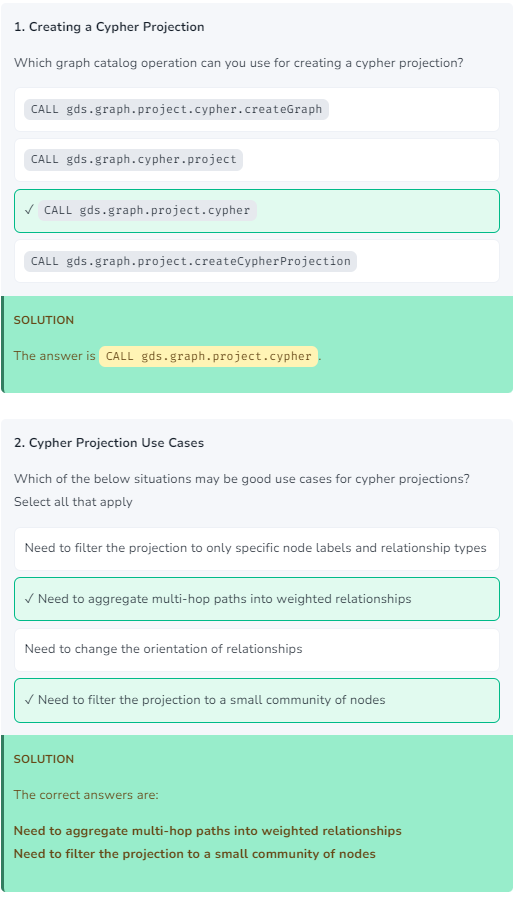
| **name** | **score** |
| --- | --- |
| Bruce Willis | 114.0 |
| Robert De Niro | 109.0 |
| Denzel Washington | 96.0 |
| Johnny Depp | 90.0 |
| Nicolas Cage | 86.0 |
| Julianne Moore | 84.0 |
| Samuel L. Jackson | 82.0 |
| Morgan Freeman | 81.0 |
| Ben Affleck | 81.0 |
| Brad Pitt | 79.0 |
| Sandra Bullock | 77.0 |
| George Clooney | 77.0 |
| Julia Roberts | 75.0 |
| Matt Damon | 75.0 |
| Keanu Reeves | 74.0 |

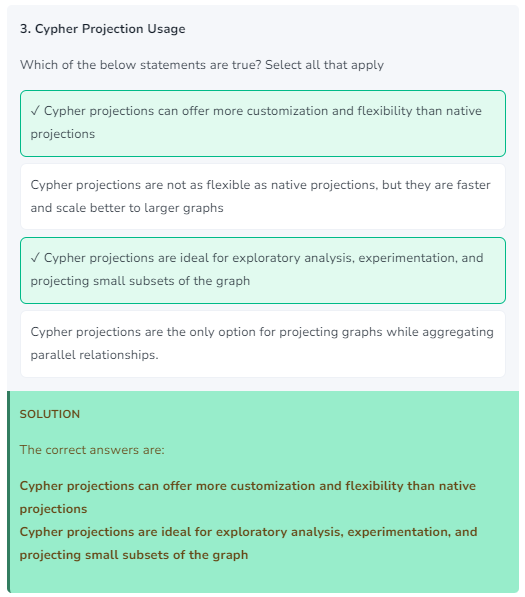
Here are some general tips for transition workflow from cypher to native projections depending on the initial use case:

1. **Filtering with Node Property Conditions**: Add a label to nodes that meet the property condition(s) so you can filter in the native projection
2. **Filtering with Relationship Property Conditions:** If possible, add a label to source and target nodes for each relationship that meets the property condition. Otherwise, consider adding an additional relationship type to your data model to capture the relationships that meet the condition.
3. **Aggregating Multi-Hop Paths:**
   1. See if collapse-path can meet your use case needs. It doesn’t weight relationships, but results can often be very similar to weighted aggregations
   2. For certain types of problems on large projections, similarity and embedding algorithms can be used to approximate the aggregated relationships.

For other complex use cases it will often come back to your data model in the Neo4j database. Is it possible to adjust your data model so node labels and relationship types better distinguish the data you want to filter on for data science application? This may involve aggregating certain paths into single relationships, developing more node labels and or relationship types, or other types of transformations.

### Questions and Answers-





# Challenge: Cypher Projection

## Create a Cypher Projection

Create a cypher projection representing all User nodes that have rated a Movie with a release year **greater than** 2014. Only include RATED relationships with ratings **greater than or equal to** 4 stars.

What is the relationship count of the projection?

## Solution-

CALL gds.graph.project.cypher(

  // Projection name

  'movie-ratings-after-2014',

  // Cypher statement to find all nodes in projection

  '

    MATCH (u:User) RETURN id(u) AS id, labels(u) AS labels

    UNION MATCH (m:Movie) WHERE m.year > 2014 RETURN id(m) AS id, labels(m) AS labels

  ',

  // Cypher statement to find every User that rated a Movie

  // where the rating property is greater than or equal to 4

  // and the movie was released after 2014 (year > 2014)

  '

    MATCH (u:User)-[r:RATED]->(m:Movie)

    WHERE r.rating >= 4 AND m.year > 2014

    RETURN id(u) AS source,

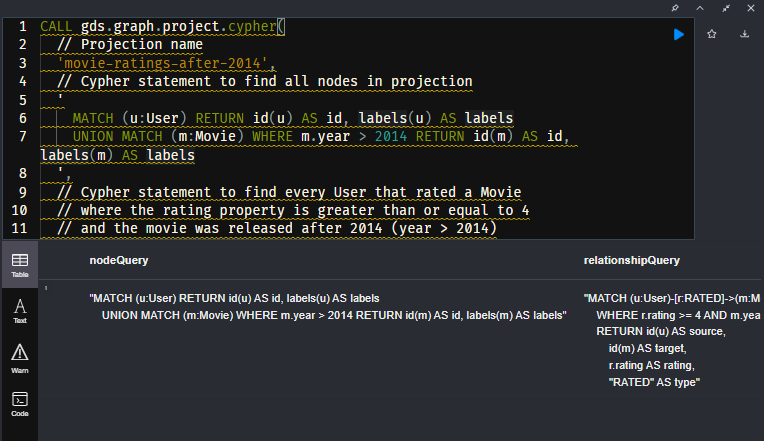
        id(m) AS target,

        r.rating AS rating,

        "RATED" AS type

  '

);



# Course Summary

Congratulations! You should now have a good understanding of the Neo4j Graph Data Science (GDS) library.

In the first module, **Neo4j GDS Overview**, we covered learned a high level technical overview of Neo4j GDS including how the product works.

In the second module, **Graph Management**, you learned how GDS how to create and manage in-memory graph projections, the distinction between native and cypher projections, and when to use them.

## Recommendations Dataset

If you would like an offline copy of the database, you can download a database dump from Github.

[**Recommendations dataset**](https://github.com/neo4j-graph-examples/recommendations)

## Resources

* **There are many resources available to you for learning more about Neo4j**  
  [**https://neo4j.com/developer/resources/**](https://neo4j.com/developer/resources/)
* **Neo4j Community Site where you can ask or answer questions about Neo4j and discuss with other users:**  
  [**https://community.neo4j.com**](https://community.neo4j.com/)
* **Neo4j documentation:**  
  [**https://neo4j.com/docs/**](https://neo4j.com/docs/)
* **Neo4j Sandboxes for experimenting with graphs:**  
  [**https://sandbox.neo4j.com/?ref=graph-academy**](https://sandbox.neo4j.com/?ref=graph-academy)
* **Videos on the Neo4j YouTube channel:**  
  [**https://www.youtube.com/channel/UCvze3hU6OZBkB1vkhH2lH9Q**](https://www.youtube.com/channel/UCvze3hU6OZBkB1vkhH2lH9Q)
* **Become a Neo4j certified developer:**  
  [**https://graphacademy.neo4j.com/categories/certification/**](https://graphacademy.neo4j.com/categories/certification/)
* **GitHub repository:**  
  [**https://github.com/neo4j-contrib**](https://github.com/neo4j-contrib)
* **Neo4j events all over the world:**  
  [**https://neo4j.com/events/world/all/**](https://neo4j.com/events/world/all/)
* **Graph Gists for learning more use cases for Neo4j:**  
  [**https://neo4j.com/graphgists/**](https://neo4j.com/graphgists/)
* **Attend a Neo4j meetup:**  
  [**https://www.meetup.com/topics/neo4j/**](https://www.meetup.com/topics/neo4j/)
* **View questions/answers raised about Neo4j:**  
  [**https://stackoverflow.com/tags/neo4j/hot**](https://stackoverflow.com/tags/neo4j/hot)