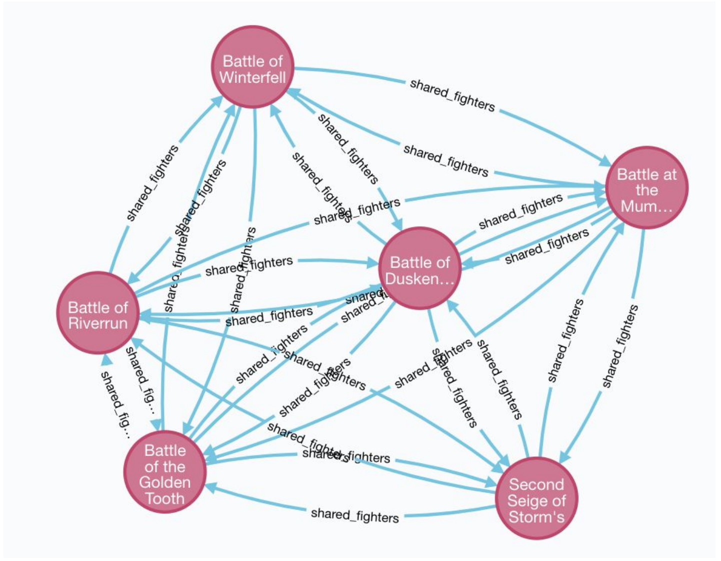
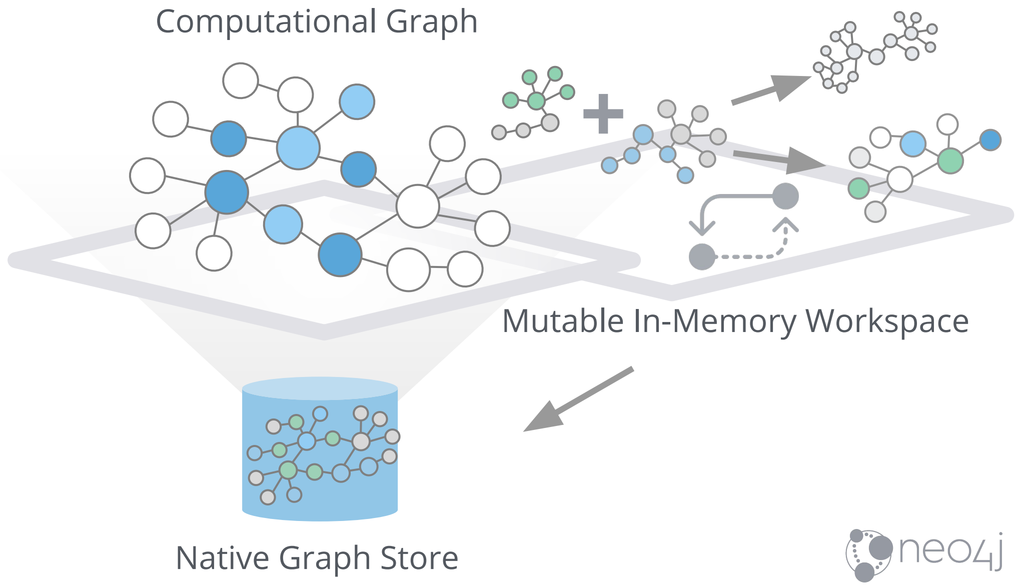
Neo4j Graph Data Science

Graph Data Science (GDS) Library Overview

Neo4j's Graph catalog

Procedures to let you reshape and subset your transactional graph so you have the right data in the right shape.



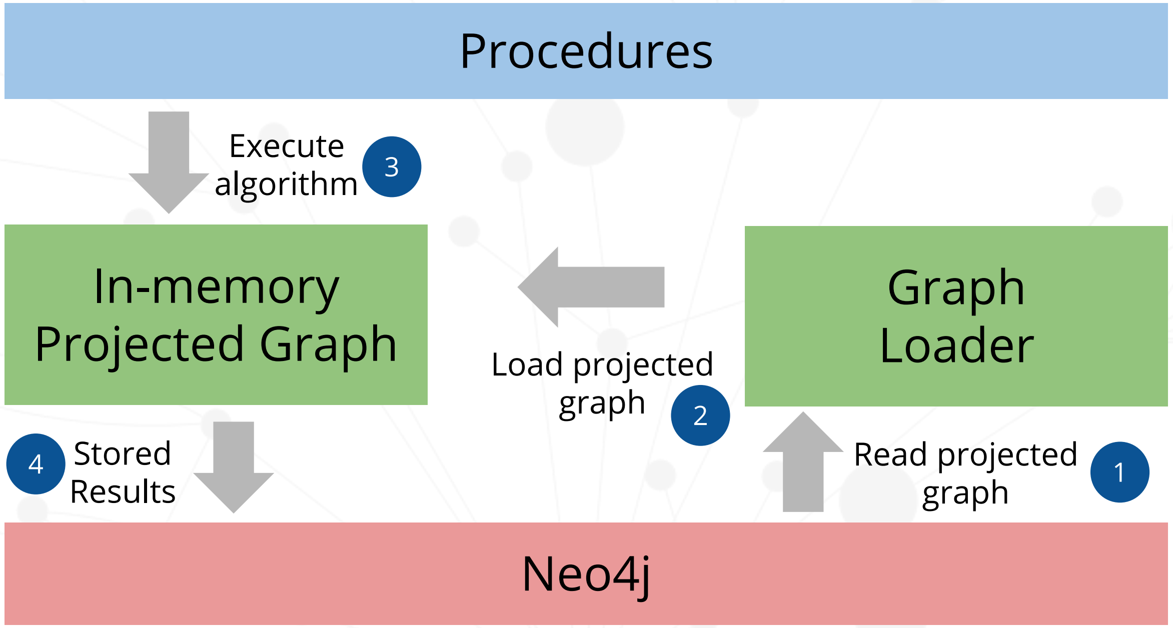


Objective

Understand the GDS Graph Catalog.

- Named Graphs vs Anonymous Graphs
- Native Projection vs Cypher Projection
- Flexibility vs Performance Trade Off
- In-memory graph mutability syntax and benefits

Execution of Graph Algorithms



Graph Loader

- **Named graphs**
  - are loaded once, with a name, and can be used by multiple algorithms

```
CALL gds.graph.create(
     'got-interactions',
     'Person',
     'INTERACTS_1',
     {
          nodeProperties: 'birth_year',
          relationshipProperties:
'weight'
     }
)
```

- Projects a monopartite graph with Person nodes and “INTERACTS_1” relationships
- Loads properties “birth_year” and “weight”

Graph Loader

- **Anonymous graphs**
  - are created on-demand
  - are deleted after its execution is completed

```
CALL gds.<algo>.<mode>(
     {
    nodeProjection: 'Person',
    relationshipProjection:'INTERACTS_1',
    nodeProperties: 'birth_year',
    relationshipProperties: 'weight'
  }
)
YIELD <results>
```

- Projects a monopartite graph with Person nodes and “INTERACTS_1” relationships
- Loads properties “birth_year” and “weight”

Graph Loader

- **Native Projection**
  - loaded directly from the data store using node labels and relationship types

```
CALL gds.graph.create(
     'got-interactions',
     'Person',
     'INTERACTS_1',
     {
          nodeProperties: 'birth_year',
          relationshipProperties:
'weight'
     }
)
```

- Projects a monopartite graph with Person nodes and “INTERACTS_1” relationships
- Loads properties “birth_year” and “weight”

Graph Loader

- **Cypher Projection**
  - execute cypher queries to populate nodes and relationships which are then loaded into the analytics graph

```
CALL gds.graph.create.cypher(
'got-interactions',
'MATCH (p:Person) RETURN id(p) as id, p.birth_year as birth_year',
'MATCH (p1:Person)-[r:INTERACTS_1]->(p2:Person) RETURN id(p1) as source,
id(p2) as target, r.weight as weight')
```

- Projects a monopartite graph with Person nodes and “INTERACTS_1” relationships
- Loads properties “birth_year” and “weight”

Native Projection

neo4j

Native Projection

```
CALL gds.graph.create(
    graphName: STRING,
    nodeProjection: STRING, LIST, or MAP,
    relationshipProjection: STRING, LIST, or MAP,
    configuration: MAP
);
```

Syntax

```
CALL gds.graph.create(
    'got-interactions',
    'Person',
    'INTERACTS_1',
    {
        nodeProperties: 'birth_year',
        relationshipProperties: 'weight',
        readConcurrency: 4
    }
)
```

Example

Native Projections: Nodes

Shorthand:

```
CALL gds.graph.create('my-graph', 'nodeLabel','relationshipType');
```

Long form:

```
CALL gds.graph.create('my-graph', {
<node-label>:  {
        label: <neo4j-label>,
        properties: <node-property-mappings>}
    },
    relationshipProjection: STRING, LIST, or MAP);
```

<node-label>: specify the node label name you want to use in your analytics graph

label: specify the node label(s) *from the Neo4j database*

properties: one or mode node properties to map from Neo4j

Native Projections: Nodes

Node Labels:

```
CALL gds.graph.create('my-graph', {
    <node-label>:  {
            label: ['Label1', 'Label2'],
            properties: {
                <property-key-1>: {
                    property: <neo-property-key>
                    defaultValue: <numeric-value>
                },
                <property-key-2>: {
                    property: <neo-property-key>
                    defaultValue: <numeric-value>
                }
            }
        }
    },relationshipProjection)
```

Native Projections: Nodes

Node Labels:

```
CALL gds.graph.create('my-graph', {
    Client:  {
            label: ['ComercialClient', 'CosumerClient'],
            properties: {
                stateId,
                seed: {
                    property: 'prevCommunityId'
                },
            }
        }
    },relationshipProjection)
```

`neo-property-key` will default to the specified property key
`defaultValue` defaults to NaN

Native Projections: Relationships

Shorthand:

```
CALL gds.graph.create('my-graph', 'nodeLabel','relationshipType');
```

Long form:

```
CALLs gds.graph.create('my-graph',
    nodeProjection: STRING, LIST, or MAP, {
        <relationship-type-1>:  {
       type: <neo4j-type>,
       orientation: <projection-type>,
       aggregation: <aggregation-type>,
       properties: <relationship-property-mappings>
    },
         <relationship-type-2>:  {
           type: <neo4j-type>,
           orientation: <projection-type>,
           aggregation: <aggregation-type>,
           properties: <relationship-property-mappings>
    }
});
```

Native Projections: Relationships

Long Form:

```
CALL gds.graph.create('my-graph',
    nodeProjection: STRING, LIST, or MAP, {
    <relationship-type-1>:  {
        type: <neo4j-type>,
        orientation: <projection-type>,
        aggregation: <aggregation-type>,
        properties: <relationship-property-mappings>
    }});
```

Aggregation specifies how parallel (duplicate) Neo4j relationships are represented in the analytics graph:

- **NONE**: No aggregation (*default*)
- **MIN, MAX, SUM, COUNT**: A single relationship is retained with an **aggregate** property
- **SINGLE**: A single, arbitrary relationship is projected

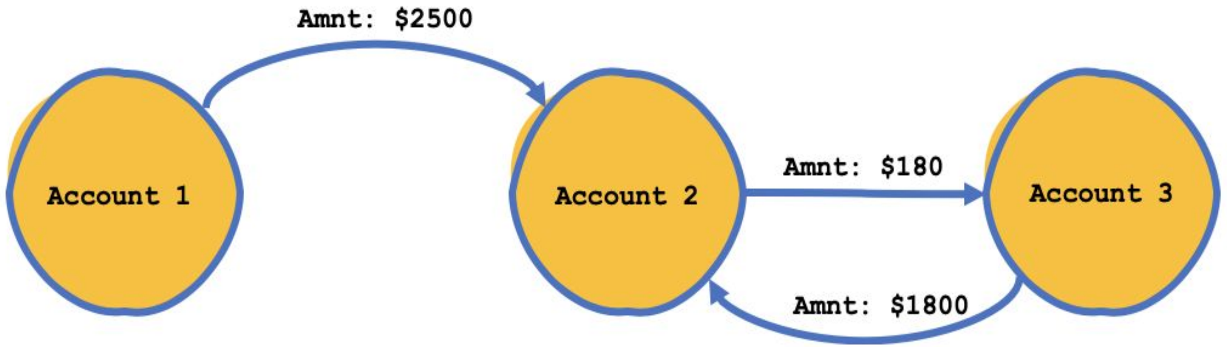
Native Projections: Relationships

Aggregation is pretty important:

- How you deduplicate (or don't) your relationships impacts results
- Much faster way to aggregate properties than via Cypher

using 'single' to pick a single relationship:

using 'sum' to sum up the amounts:



Useful when you aren't using weights and a single relationship is appropriate

Useful when you have many relationships you want to flexibly combine into a weight

Native Projections: Relationships

Long Form:

```
CALL gds.graph.create('my-graph',
    nodeProjection: STRING, LIST, or MAP, {
    <relationship-type-1>:  {
       type: <neo4j-type>,
       orientation: <projection-type>,
       aggregation: <aggregation-type>,
       properties: {
        <property-key1>: {
            property:<neo4j-property-key>,
            defaultValue: <defaultValue>,
            aggregation: <aggregation-type>
            }
        }
    }})
```

properties works the same as with nodes - specifies a mapping of neo4j relationship properties to the in-memory graph.

...putting it all together

```
CALL gds.graph.create('my-graph’,{
      Customer: {
            label: ['commercialClient', 'consumerClient'],
            properties: {
                  seed: {
                        property:'stateId’
                                     }
                        }
             }
      },
      INTERACTS:  {
            type: 'TRANSACTION',
            orientation: 'NATURAL',
            aggregation: 'SUM',
            properties: {
                  weight: {
                        property:'amount',
                        defaultValue: 0.0,
                        aggregation: 'SUM'
                        }
                           }
      }});
```

...putting it all together

```
CALL gds.graph.create('my-graph',{
     Customer: {
          label: ['commercialClient', 'consumerClient'],
          properties: {
               seed: {
                    property:'stateId'
               }
          }
     },{
     INTERACTS:  {
          type: 'TRANSACTION',
          orientation: 'NATURAL',
          aggregation: 'SUM',
          properties: {
               weight: {
                    property:'amount',
                    defaultValue: 0.0,
                    aggregation: 'SUM'
               }
          }
     }})
```

graph name

...putting it all together

```
CALL gds.graph.create('my-graph’,{
     Customer: {
          label: ['commercialClient', 'consumerClient'],
          properties: {
               seed: {
                    property:'stateId'
               }
          }
     },{
     INTERACTS:  {
          type: 'TRANSACTION',
          orientation: 'NATURAL',
          aggregation: 'SUM',
          properties: {
               weight: {
                    property:’amount',
                    defaultValue: 0.0,
                    aggregation: 'SUM'
               }
          }
     }})
```

nodes

…putting it all together

```
CALL gds.graph.create('my-graph’,{
     Customer: {
          label: ['commercialClient', 'consumerClient'],
          properties: {
               seed: {
                    property:'stateId'
               }
          }
     },{
     INTERACTS:  {
          type: 'TRANSACTION',
          orientation: 'NATURAL',
          aggregation: 'SUM',
          properties: {
               weight: {
                    property:’amount',
                    defaultValue: 0.0,
                    aggregation: 'SUM'
               }
          }
     }})
```

$\left.\vphantom{\begin{matrix}a\\a\\a\\a\\a\\a\\a\\a\\a\\a\\a\end{matrix}}\right\}$ relationships

Shorthand syntax

```
CALL gds.graph.create('my-graph',['commercialClient', 'consumerClient'],
'TRANSACTS',
{
    nodeProperties: {seed: 'stateId'},
    relationshipProperties: { interacts: {
                        property: 'amount',
aggregation:’SUM’,
                        defaultValue: 0.0
                     }
                  }
}) YIELD graphName, nodeCount, relationshipCount
```

Let’s try it out!

```
CALL gds.graph.create(
  'got-interactions-1',
  'Person',
  {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    }
  }
);
```

How many nodes are in your projected graph? How many labels?

```
CALL gds.graph.drop('got-interactions-1');
```

Try it yourself!

Exercise: Can you add in a second relationship type, INTERACTS_2, into your projection?

How many nodes and relationships are in this graph?

```
CALL gds.graph.create(
  'got-interactions-12',
  'Person',
  {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    },
     INTERACTS_2: {
      orientation: 'UNDIRECTED'}});
```

Try it yourself!

Exercise: Can you reverse the direction of INTERACTS_2, and rename it INTERACTS2_BACKWARDS

How many nodes and relationships are in this graph?

```
CALL gds.graph.create(
  'got-interactions-12_reverse',
  'Person',
  {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    },
     INTERACTS_2_BACKWARDS: {
     type: 'INTERACTS_2',
      orientation: 'REVERSE'}});
```

Don’t forget!

```
CALL gds.graph.drop('got-interactions-12');
CALL gds.graph.drop('got-interactions-12_reverse');
```

Native Projection Exercise

1. Use the Native shorthand syntax to load a graph with the following specifications:
   - Name: Season1Interactions
   - Node Label: Character
   - Relationship type: INTERACTS_SEASON1
2. Edit exercise 1 by renaming the graph to Season1InteractionsWeighted and adding the relationship property weight.

Native Projection Exercise Solution

```
CALL gds.graph.create('Season1Interactions','Character', 'INTERACTS_SEASON1');
```

```
CALL gds.graph.create('Season1InteractionsWeighted','Character', 'INTERACTS_SEASON1',
    {relationshipProperties:'weight'});
```

What if I want to project a different structure?

Cypher Projections

Native graphs are fast, but Cypher graphs are ***flexible*** -- use them if:

- you’re in the experimentation phase and trying to decide on a data model
- performance and repeatability aren’t too important

Syntax overview:

```
CALL gds.graph.create.cypher(
     'my-graph',
     'MATCH (p:Person) RETURN id(p) as id',
     'MATCH  (p1:Person)-[:LIVES]->(:Place)<-[:Lives]-(p2:Person)
      RETURN id(p1) as source, id(p2) as target'
);
```

Cypher Projections

Native graphs are fast, but Cypher graphs are ***flexible*** -- use them if:

- you're in the experimentation phase and trying to decide on a data model
- performance and repeatability aren't too important

Syntax overview:

```
CALL gds.graph.create.cypher(
     'my-graph',
     'MATCH (p:Person) RETURN id(p) as id',
     'MATCH  (p1:Person)-[:LIVES]->(:Place)<-[:Lives]-(p2:Person)
      RETURN id(p1) as source, id(p2) as target'
);
```

Cypher Projections

```
CALL gds.graph.create.cypher(
     'my-graph',
     'MATCH (p:Person) RETURN id(p) as id',
     'MATCH  (p1:Person)-[:LIVES]->(:Place)<-[:Lives]-(p2:Person)
      RETURN id(p1) as source, id(p2) as target'
);
```

The **node query** defines the population of nodes to consider, the **relationship query** joins them up

Requirements:

- Node query ***must*** return a column called `id` that uniquely identifies a node
- Relationship query ***must*** return `source` and `target` columns with unique node identifiers

Cypher Projections

Node and relationship properties are specified in the queries

```
CALL gds.graph.create.cypher(
    'my-graph',
    'MATCH (p:Person) RETURN id(p) as id, p.community as community',
    'MATCH  (p1:Person)-[r:WORKS_WITH|FRIEND_OF]->(p2:Person)
     RETURN id(p1) as source, id(p2) as target, type(r) AS type,
           count(r) as weight'
);
```

Multiple relationships are identified in the relationship query

Directionality is interpreted from the ordering of source, target

Cypher Projections: relationship aggregation

The easiest way to aggregate relationships is in the relationship query

Cypher graphs also support aggregation functions like Native graphs:

```
CALL gds.graph.create.cypher(
     graphName: STRING,
     nodeQuery: STRING,
     relationshipQuery:  STRING,
     configuration: MAP
);
```

Cypher Projections: relationship aggregation

```
CALL gds.graph.create.cypher(
     'my-graph',
     'MATCH (p:Person) RETURN id(p) as id',
     'MATCH (p1:Person)-[r:WORKS_WITH|FRIEND_OF]->(p2:Person)
      RETURN id(p1) as source, id(p2) as target, r.duration as time',
     {    relationshipProperties: {
               time_known: {
                    property: 'time',
                    aggregation: 'SUM'
                    defaultValue: 0.0
               }
          }
     }
);
```

Why bother?

This lets you bypass using the Cypher engine to do the aggregation -- it’s much more performant on big data sets!

Let’s try it out!

```
CALL gds.graph.create.cypher(
  'same-house-graph',
  'MATCH (n:Person) RETURN id(n) AS id',
  'MATCH
(p1:Person)-[:BELONGS_TO]-(:House)-[:BELONGS_TO]-(p2:Person
) RETURN id(p1) AS source, id(p2) AS target'
);
```

How many nodes are in your projected graph? How many labels?

```
CALL gds.graph.drop('got-interactions-cypher');
```

Try it yourself!

Exercise: Can you modify the cypher projection to find people who `APPEARED_IN` the same `Book`?

How many nodes and relationships are in this graph?

How could you modify this query to add a weight property with the number of books people `APPEARED_IN` together?

```
`Match (p1:Person)-(:APPEARED_IN)->(b:Book)<-(:APPEARED_IN)-(p2:Person)
RETURN id(p1) AS source, id(p2) AS target, count(distinct b) AS weight`
```

Cypher Projection Exercise

- Use a Cypher projection to load a graph with the following specifications:
    - Name: InteractsWeighted
    - Node labels: Character
    - Relationships: All
    - Relationship property: weight
    - Relationship aggregation: SUM
    - Relationship property default value: 0.0

Cypher Projection Exercise Solution

```
CALL gds.graph.create.cypher('InteractsWeighted','MATCH(c:Character) RETURN id(c) AS id',
    'MATCH(c1:Character)-[r]→(c2:Character) RETURN id(c1) AS source,id(c2) AS target,r.weight AS weight',
    {
        relationshipProperties:{
            weight:{
                aggregation:'SUM',
                defaultValue:0.0
            }
        }
    }
);
```

Graph Management

Every Named Graph is stored in memory (the heap)

Management procedures:

- `gds.graph.list: for listing named graphs`
- `gds.graph.exists: check is the graph exists`
- `gds.graph.drop: remove named graph`

Graph access:

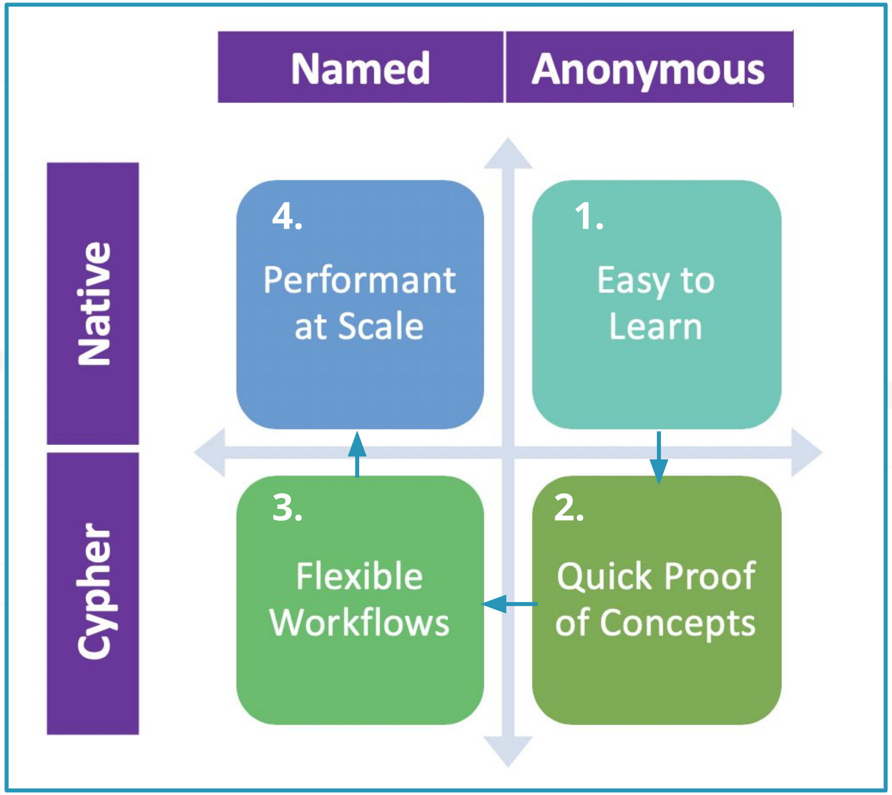
- all graphs are user specific -- another user can’t drop or use my graphs

Graph Management Exercise

- Use graph management procedures to remove the graph InteractsWeighted from the Graph Catalog.
- Solution:

```
CALL gds.graph.drop('InteractsWeighted');
```

Flexibility vs. Performance Trade Offs



Mutating the in-memory graph

```
CALL gds[.<tier>].<algorithm>.mutate(
    graphName: STRING,
    configuration: MAP
);
```

Mutate is similar to write but instead adds a new property to the in-memory graph, specified by the mutateProperty parameter (note: this must be a *new* property)

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

Writing your results back to Neo4j

After you’ve finished your workflow, you can write your results back to Neo4j:

```
CALL gds.graph.writeNodeProperties(
    graphName, [<node_properties>])
```

```
CALL gds.graph.writeRelationship(
    graphName, <RELATIONSHIP>, [<relationship_property>] )
```

Writing your results back to Neo4j

After you’ve finished your workflow, you can write your results back to Neo4j:

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

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

Note: you can also export your in-memory graph with

```
    gds.beta.graph.export('graph-name',
    {storeDir:'/some/dir',dbName:'persisted-graph'})
```

Why bother?

Writing back to Neo4j is often the slowest step:

- When chaining algorithms, skip writing back the results you don't need
- Write optimization for writing multiple properties more efficiently

Example: Similarity + Louvain

- Run node similarity, update the in-memory graph with SIMILAR_TO
- Run Louvain on SIMILAR_TO relationship
- Only write back Louvain communities

Next: Graph Algorithms

- What are graph algorithms and how can we use them?
- Neo4j GDS Procedures and Functions Overview
- Algorithms support tiers and execution modes
- Deep Dive on Graph Algorithms Using Game of Thrones Dataset
  - Community Detection
  - Similarity
  - Centrality
  - Path Finding & Search
  - Link Prediction
- Best Practices Using Neo4j GDS