Wednesday, May 14, Practical Session III: Querying Your Data

Ryan Horne

Advanced Research Computing, UCLA

ryan.matthew.horne@gmail.com

CC BY SA

Agenda

- More complicated data examples (with error handling)
- Some basic network queries
- Where are the limits of free Neo4j?

CC BY SA

A More complicated example

We will now look at Pleiades Data

- Extensive data set
- Actually *not* in a relational database, but not really in a graph either
- Output CSVs are well-suited for network analysis
- Show various code

CC BY SA

Pleiades

```
1
2  // Import archaeological_remains.csv
3  LOAD CSV WITH HEADERS FROM 'file:///archaeological_remains.csv' AS
4  MERGE (n:ArchaeologicalRemains {key: row.key})
5  SET n.term = row.term, n.definition = row.definition;
6
7  // Import languages_and_scripts.csv
8  LOAD CSV WITH HEADERS FROM 'file:///languages_and_scripts.csv' AS r
9  MERGE (n:LanguagesAndScripts {key: row.key})
10 SET n.term = row.term, n.definition = row.definition;
11
12 // Import place_types.csv
13 LOAD CSV WITH HEADERS FROM 'file:///place_types.csv' AS row
14 MERGE (n:PlaceTypes {key: row.key})
15 SET n.term = row.term, n.definition = row.definition;
16
17 // Import association_certainty.csv
18 LOAD CSV WITH HEADERS FROM 'file:///association_certainty.csv' AS r
```

CC BY SA

For Speed

```
1
2 CREATE INDEX FOR (p:Places) ON (p.id)
3
```

This is optional but *VERY recommended* for large data sets or tables; it essentially creates a method for the computer to efficently and effectively sort and operate on your data.

CC BY SA

Load Edges

Create relationships between places and names nodes (case-insensitive, excluding null/empty strings, with explicit string casting)

```
MATCH (p:Places), (l:Names)
WHERE
  toLower(COALESCE(toString(p.id), '')) <> '' AND
  toLower(COALESCE(toString(l.place_id), '')) <> '' AND
  toLower(COALESCE(toString(p.id), '')) = toLower(COALESCE(toString(l.place_id), ''))
CREATE (p)-[:HAS_NAME]->(l)
RETURN p.id AS PlaceID, l.place_id AS LocationLinestringPlaceID, 'has name' AS RelationshipType
LIMIT 10;

```

```
MATCH (p:Places), (l:Names)
WHERE
  toLower(COALESCE(toString(p.id), '')) <> '' AND
  toLower(COALESCE(toString(l.place_id), '')) <> '' AND
  toLower(COALESCE(toString(p.id), '')) = toLower(COALESCE(toString(l.place_id), ''))
MERGE (p)-[:HAS_NAME]->(l)
RETURN p.id AS PlaceID, l.place_id AS LocationLinestringPlaceID, 'has name' AS RelationshipType
LIMIT 10;
```

CC BY SA

Breakdown

CC BY SA

Breakdown

`toLower(COALESCE(toString(p.id), '')) <> ''`

- Purpose: Check that p.id exists and is not empty or null.
  - `p.id`: The id property of a node labeled :Places.
  - `toString(...)`: Converts it to a string (in case it's a number or null).
  - `COALESCE(..., '')`: If p.id is null, substitute it with an empty string.
  - `toLower(...)`: Normalizes to lowercase for case-insensitive comparison.
  - `<> ''`: Ensures the result is not an empty string.
- This filters out *:Places* nodes that don't have a usable id.
- Same idea for `toLower(COALESCE(toString(l.place_id), '')) <> ''`

CC BY SA

Breakdown

```
toLower(COALESCE(toString(p.id), '')) =
toLower(COALESCE(toString(l.place_id), ''))
```

- Purpose: Match the two values case-insensitively, safely handling nulls and type mismatches.
- Both sides are:
  - Coalesced to ’’ if null
  - Converted to strings
  - Lowercased
- If they match, the relationship is created.

CC BY SA

Breakdown

```
CREATE (p)-[:HAS_NAME]->(l)
```

- Creates a directed relationship from the Places node to the Names node.
- Relationship type: :HAS_NAME

CC BY SA

Hands-On Neo4j: The Sample Database

CC BY SA

Hands-On Neo4j: The Sample Database

- Switch to the Example Project
- Click on the database icon
- Look at the sections; we have:
  - node labels
  - relationship types
  - property keys
- Click on the `Person` button

CC BY SA

Create Some Data

```
1 CREATE (p1:Person {name: "Keanu Reeves", born: 1964})
2 CREATE (p2:Person {name: "Carrie-Anne Moss", born: 1967})
3 CREATE (m1:Movie {title: "The Matrix", released: 1999})
4 CREATE (m2:Movie {title: "The Matrix Reloaded", released: 2003})
```

CC BY SA

Create Some Data

```
1 MATCH (p:Person {name: "Keanu Reeves"}), (m:Movie {title: "The Matr
2 CREATE (p)-[:ACTED_IN]->(m)
3
4 MATCH (p:Person {name: "Carrie-Anne Moss"}), (m:Movie {title: "The
5 CREATE (p)-[:ACTED_IN]->(m)
6
7 MATCH (p:Person {name: "Keanu Reeves"}), (m:Movie {title: "The Matr
8 CREATE (p)-[:ACTED_IN]->(m)
```

CC BY SA

Finding Mr. Gump

```
1 MATCH (p:Person {name: "Tom Hanks"})
2 RETURN p
3 LIMIT 1;
```

CC BY SA

Finding Mr. Gump

```
1 MATCH (p:Person {name: "Tom Hanks"})-[:ACTED_IN]->(m:Movie)
2 RETURN p.name AS actor, m.title AS movie
3 ORDER BY m.title;
```

CC BY SA

MATCH vs MERGE

- MATCH assumes that a node with the specified query exsists
- MERGE adds the node if it does not

CC BY SA

Match

```
1 MATCH (p:Person)
2 WHERE p.name = "John Doe"
3 SET p.personstatus = 'found'
4 RETURN p
```

CC BY SA

Merge

```
1
2 MERGE (p:Person {name: "John Doe"})
3 ON MATCH SET p.personstatus = 'found'
4 RETURN p
```

CC BY SA

Create a Relationship Between Nodes

```
1  MATCH (p:Person), (m:Movie)
2  WHERE p.name = "Tom Hanks" AND m.title = "Cloud Atlas"
3
4  CREATE (p)-[w:WATCHED]->(m)
5  RETURN type(w)
```

CC BY SA

Wait, what is this?

- Did you see the “->” above?
- What is going on?

CC BY SA

Directed vs. Undirected Graph

- We can specify the *direction* of the relationship
  - or just leave out the arrow for undirected connections
- Why might this be important?
- When do we not care (or that direction does not make sense?)

CC BY SA

Hands-On Neo4j

- In writing a Cypher query, a node is enclosed between a parenthesis
  - `(p:Person)`
  - p is a variable (what we are calling it) and *Person* is the node
  - Also needs a `RETURN` statement
  - We are using `MATCH` that specifies the patterns we will use in the data
    - Often with WHERE clause

CC BY SA

Find all actors

```
1 // Find all actors
2 MATCH (a:Person)-[:ACTED_IN]->(m:Movie) RETURN a.name, m.title LIMI
```

CC BY SA

Count nodes

```
// Count nodes
MATCH (n) RETURN labels(n), count(*);
```

CC BY SA

Write something!

Can you think of a way to write the following query:

- Load movies into variable `m`
- Where the *released* property > 1999
- Now just return the *count*

CC BY SA

One Way

```
1  MATCH (m:Movie)
2  WHERE m.released > 1999
3  RETURN count(m) AS movieCount;
```

CC BY SA

Relationships

- Enclosed in square brackets
  - `[w:WORKS_FOR]`
    - w is a variable
    - WORKS_FOR is the type of relationship

CC BY SA

Count how many movies each person directed

```
1
2 MATCH (p:Person)-[:DIRECTED]->(m:Movie)
3 RETURN p.name, count(m) AS movies_directed
4 ORDER BY movies_directed DESC;
```

CC BY SA

Count How Many Movies?

```
1  // Count how many movies each person directed
2  MATCH (p:Person)-[:DIRECTED]->(m:Movie)
3  RETURN p.name, count(m) AS movies_directed
4  ORDER BY movies_directed DESC;
```

CC BY SA

What do you think this does?

```
1  MATCH (p:Person)-[d:DIRECTED]-(m:Movie)
2  WHERE m.released > 2010
3  RETURN p,d,m
```

CC BY SA

Your Turn

Query to get all the people who acted in a movie that was released after 2010.

CC BY SA

Possible Answer

```
1  MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
2  WHERE m.released > 2010
3  RETURN DISTINCT p.name AS actor
4  ORDER BY actor;
5
```

CC BY SA

Another question

What is the difference between these two queries?

```
1  MATCH (p:Person)
2  RETURN p
3  LIMIT 20
```

```
1  MATCH (n)
2  RETURN n
3  LIMIT 20
```

CC BY SA

Find all People With any Relation to a Node

```
1
2 MATCH (p:Person)-[relatedTo]-(m:Movie {title: "Cloud Atlas"})
3 RETURN p.name, type(relatedTo)
4
```

CC BY SA

Exploring our Graph: Paths

- Movement in a graph via edges
- Sequence of nodes connected via edges
- *Simple path*: A path that does not repeat nodes

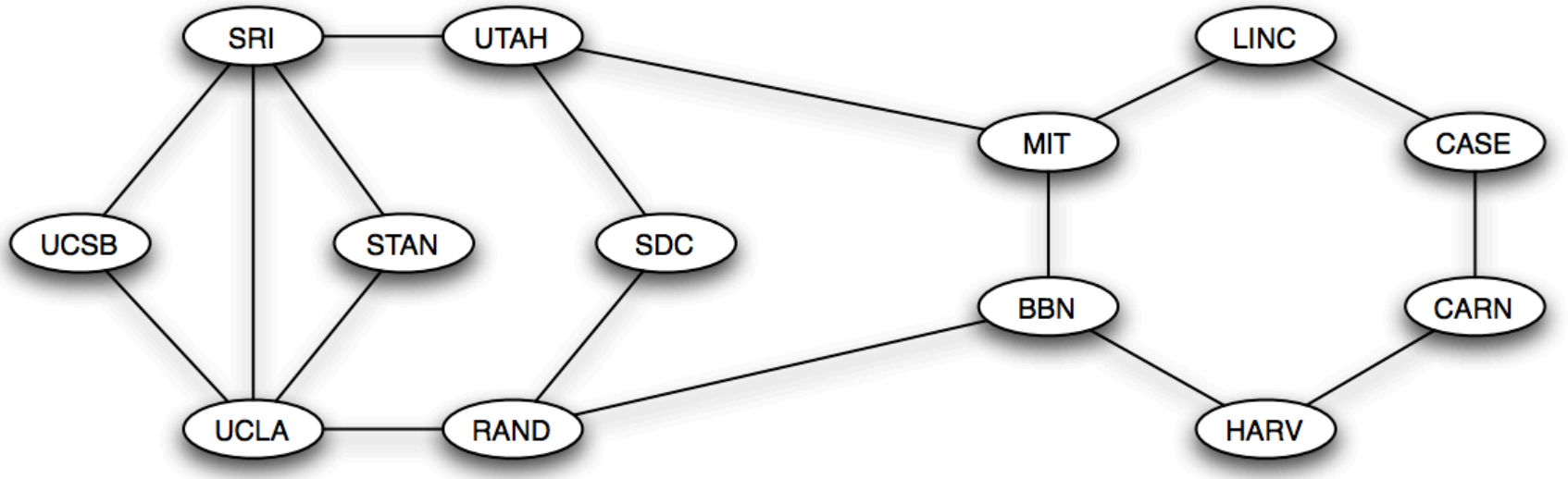
CC BY SA

Connectivity

- Path between every pair of nodes
- Goal of most designed networks
- NOT a necessary feature of graphs though!
- There are social networks with disconnected features

CC BY SA

Example – Anyone know what this is?



CC BY SA

Graph Distance

- Not geographic (mostly!)
- Distance = length of the shortest path between two nodes
  - Number of edges
  - Sometimes we can simply look at this
  - Other times…we need computers!

CC BY SA

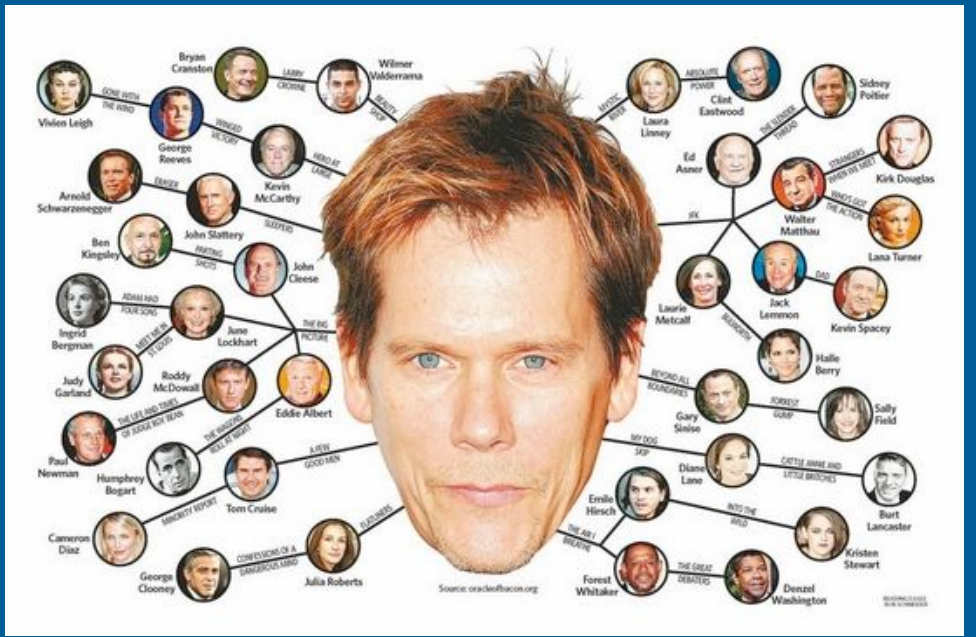
In Cypher

```
1 MATCH (a:Person {name: 'Tom Hanks'}), (b:Person {name: 'Meg Ryan'})
2 MATCH p = shortestPath((a)-[*]-(b))
3 RETURN p, length(p) AS hops
```

CC BY SA

Small-world phenomenon

- Shorter path then you would think to get from one node to another
- Origin of the term “six degrees of separation”
- Practical terms: Who has a friend from another country?



CC BY SA

Degree Measurements

- Sum of all other nodes with a direct *path* to a node
- Signifies activity or popularity
- Very good for looking at nodes in a local context
- In, Out
  - Weighted and unweighted

CC BY SA

Centrality

- *VERY* important concept!
- What many people want to see in networks: the most prominent nodes
- These are often the “key” players in a network
- Idea of social power
  - Assertion: Power is inherently relational

CC BY SA

Measurements

Degree

Betweenness

Closeness

Eigenvector

CC BY SA

Degree Measurement

```
1  MATCH (p:Person)-[:ACTED_IN]->()
2  RETURN p.name, count(*) AS degree
3  ORDER BY degree DESC;
```

CC BY SA

In-Degree Measurement

```
1 // In-degree of each person (number of incoming relationships)
2 MATCH ()-[r]->(p:Person)
3 RETURN p.name AS person, count(r) AS in_degree
4 ORDER BY in_degree DESC;
```

CC BY SA

Out-Degree Measurement

```
// Out-degree of each person (number of outgoing relationships)
MATCH (p:Person)-[r]->()
RETURN p.name AS person, count(r) AS out_degree
ORDER BY out_degree DESC;
```

CC BY SA

Out-Degree Measurement

```
1  // In-degree of each Movie (how many people connected to each movie)
2  MATCH ()-[r]->(m:Movie)
3  RETURN m.title AS movie, count(r) AS in_degree
4  ORDER BY in_degree DESC;
```

CC BY SA

Plugin Time!

For some of the other measurements and centralities we need to enable the *Graph Data Sciecne Library*

CC BY SA

Project the Graph

- We need to put the graph in memory

```
CALL gds.graph.project(
  'movies-graph',
  ['Person', 'Movie'],
  {
    ALL: {
      type: '*',
      orientation: 'NATURAL'
    }
  }
);
```

CC BY SA

What this is

`'movies-graph'`

- Name of the in-memory graph
  - This is the name GDS will use to reference the graph.
  - You can use this name in later algorithm calls, e.g., gds.pageRank.stream(‘movies-graph’).

CC BY SA

```
['Person', 'Movie']
```

- List of node labels to include
  - This tells GDS to include all nodes labeled Person and Movie.
  - These are the two main node types in the Movies sample database.

CC BY SA

```
ALL: {
  type: '*',
  orientation: 'NATURAL'
}
```

This defines the relationships to include in the in-memory graph.

- `ALL:`
  - A custom name for this relationship type group. You can call it anything (e.g., connections, rels, etc.), but here it's called ALL.
- `type: '*'`
  - The asterisk (*) means include all relationship types, e.g., ACTED_IN, DIRECTED, PRODUCED, etc.
  - This is useful when you want to include all existing edges between the selected nodes.
- `orientation: 'NATURAL'`
  - This keeps the original direction of relationships from the database.
  - For example, if `(:Person)-[:ACTED_IN]->(:Movie)` exists in the database, it will remain directed from person to movie.

CC BY SA

Total Degree all Nodes

```
CALL gds.degree.stream('movies-graph', { orientation: 'UNDIRECTED'
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
RETURN *, score AS totalDegree
ORDER BY totalDegree DESC
LIMIT 10;
```

CC BY SA

Breakdown

```
CALL gds.degree.stream('movies-graph', {
orientation: 'UNDIRECTED' })
```

- Runs the degree centrality algorithm on the *movies-graph* projection.
  - The `orientation: 'UNDIRECTED'` setting means that both incoming and outgoing edges are treated equally — each edge counts once for degree.
  - GDS does not collapse multiple relationships between the same node pairs by default.
- Returns a stream of results for each node in the graph.

CC BY SA

Breakdown

`YIELD nodeId, score`

- Extracts:
  - `nodeId`: the internal Neo4j ID for each node in the GDS projection.
  - `score`: the degree value (i.e., number of undirected edges for that node).

CC BY SA

Breakdown

```
WITH gds.util.asNode(nodeId) AS node, score
```

- Converts the *GDS node ID (nodeId)* into the actual Neo4j node object using *gds.util.asNode()*.
  - This allows you to access the node's labels and properties (like title or name).

CC BY SA

Breakdown

```
RETURN *, score AS totalDegree
```

- Returns all variables (node, score) using *.
  - Also aliases *score* as totalDegree for clarity.

CC BY SA

Breakdown

`ORDER BY totalDegree DESC LIMIT 10`

- Sorts the result by degree in descending order.
  - Returns only the top 10 nodes with the highest number of connections.

CC BY SA

Total degree (ignores direction) for just movies:

```
1
2 CALL gds.degree.stream('movies-graph', { orientation: 'UNDIRECTED'
3 YIELD nodeId, score
4 WITH gds.util.asNode(nodeId) AS node, score
5 WHERE 'Movie' IN labels(node)
6 RETURN node.title AS title, score AS totalDegree
7 ORDER BY totalDegree DESC
8 LIMIT 10;
```

CC BY SA

Can you do this for people?

CC BY SA

In-Degree

```
CALL gds.degree.stream('movies-graph', { orientation: 'REVERSE' })
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
RETURN *, score AS inDegree
ORDER BY inDegree DESC
LIMIT 10;

```

CC BY SA

Out-Degree

```
CALL gds.degree.stream('movies-graph', { orientation: 'NATURAL' })
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
RETURN *, score AS inDegree
ORDER BY inDegree DESC
LIMIT 10;
```

CC BY SA

Betweeness Centrality

- Measures how often a node appears on shortest paths between nodes in the network
- Often better to change visualization to identify them
- A node with a high measure here could be important…or at the periphery of multiple networks

CC BY SA

Betweeness Centrality for All Nodes

```
CALL gds.betweenness.stream('movies-undirected')
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
RETURN *, score AS betweenness
ORDER BY betweenness DESC
LIMIT 10;
```

CC BY SA

Betweeness Centrality for All Nodes

```
1 CALL gds.betweenness.stream('movies-graph')
2 YIELD nodeId, score
3 WITH gds.util.asNode(nodeId) AS node, score
4 RETURN *, score AS betweenness
5 ORDER BY betweenness DESC
6 LIMIT 10;
```

CC BY SA

Betweeness Centrality

Why the massive difference? This is the same data, right?

CC BY SA

Closeness Centrality

- Closeness centrality focuses on the shortest distances between a node and all other nodes in the network.
- It calculates the average length of these shortest paths
- A higher closeness centrality number means the node is closer to all other nodes on average, and vice versa.
- Useful to find out who spreads information quickly; might not be useful in a highly connected network
- Easy access to the rest of the network

CC BY SA

Closeness Centrality

```
1
2 CALL gds.closeness.stream('movies-undirected')
3 YIELD nodeId, score
4 RETURN gds.util.asNode(nodeId) AS node, labels(gds.util.asNode(node
5 ORDER BY score DESC
6 LIMIT 10;
7
```

CC BY SA

Eigenvector Centrality

- Basic idea: A node is important if it is linked to by other important nodes
  - I have almost no friends - but my friends are
    - The President
    - The Pope
    - Santa Claus
    - *Insert influencer / Reality TV star here*
- Global vs. local importance
- Mathematical modeling of this stretches back to the 1940s
  - Software does this for us!

CC BY SA

Eigenvector Centrality

```
1
2 CALL gds.eigenvector.stream('movies-undirected')
3 YIELD nodeId, score
4 RETURN gds.util.asNode(nodeId) AS node, labels(gds.util.asNode(node
5 ORDER BY score DESC
6 LIMIT 10;
7
```

CC BY SA

Another Ranking….

PageRank is a centrality algorithm originally developed by Google to rank web pages. In graph terms:

- A node's PageRank score measures how important or influential it is based on
  - How many connections it has
  - How important the nodes linking to it are

CC BY SA

Another Ranking....

```
1 CALL gds.pageRank.stream('movies-undirected')
2 YIELD nodeId, score
3 RETURN gds.util.asNode(nodeId) AS node, labels(gds.util.asNode(node
4 ORDER BY score DESC
5 LIMIT 10;
```

CC BY SA

Pagenvector

When to use…	Use PageRank when:	Use Eigenvector Centrality when:
You want to model flow or attention	You want a **more robust** ranking with teleportation	You want **pure influence based on structure**
You have **web-like** data (e.g., citations, followers)	You're analyzing **networks with strong mutual influence**	
You care about **importance from being linked to**	You care about **importance from who you're linked to**	

CC BY SA

Bridges

- Edge joining two nodes is a bridge if deleting that edge would cause the nodes to be in two different components of the graph
  - Very rare in social networks – think of the small world principle
- Local bridge if the endpoints have no friends in common
  - There can still be a path between the nodes if the bridge is deleted
- Best statistical approximation: Betweenness Centrality

CC BY SA

Bridges

If you want to list or find beriges…you will have to use another tool

CC BY SA

Python: Bridges

Python Code | Result

```
1  import networkx as nx
2  import csv
3  import pandas as pd
4  from community import community_louvain
5  import matplotlib.cm as cm
6  import matplotlib.pyplot as plt
7
8  # First, get the edge data
9  url = 'https://raw.githubusercontent.com/mathbeveridge/gameofthrones/master/data/got-s1-edges.csv'
10 dfedges = pd.read_csv(url)
11 # Dataset is now stored in a Pandas Dataframe
12
13 # Now we create the graph from the edge list. We need to specify the column names as they are in mix
14 G = nx.from_pandas_edgelist(dfedges, source="Source", target = "Target", edge_attr=True)
15 bridges = list(nx.bridges(G))
16 list(nx.bridges(G))
```

CC BY SA

Python: Highlight Bridges

Python Code | Result

```python
# Identify the bridges
bridges = list(nx.bridges(G))

# Set positions for nodes using a layout algorithm
pos = nx.spring_layout(G, k=.01, iterations=20)

# Draw entire network
nx.draw(G, pos, with_labels=False, node_size=5, width=.2)

# Draw bridges in red
for bridge in bridges:
    nx.draw_networkx_edges(G, pos, edgelist=[bridge], edge_color='red', width=2)
    nx.draw_networkx_nodes(G, pos, nodelist=bridge, node_color='red', node_size=700)

# Create a dictionary of labels for nodes involved in bridges
bridge_labels = {node: node for bridge in bridges for node in bridge}

# Draw labels for nodes involved in bridges
nx.draw_networkx_labels(G, pos, labels=bridge_labels, font_size=8, font_weight='bold')

plt.show()
```

CC BY SA

Network Evolution: Triadic Closure

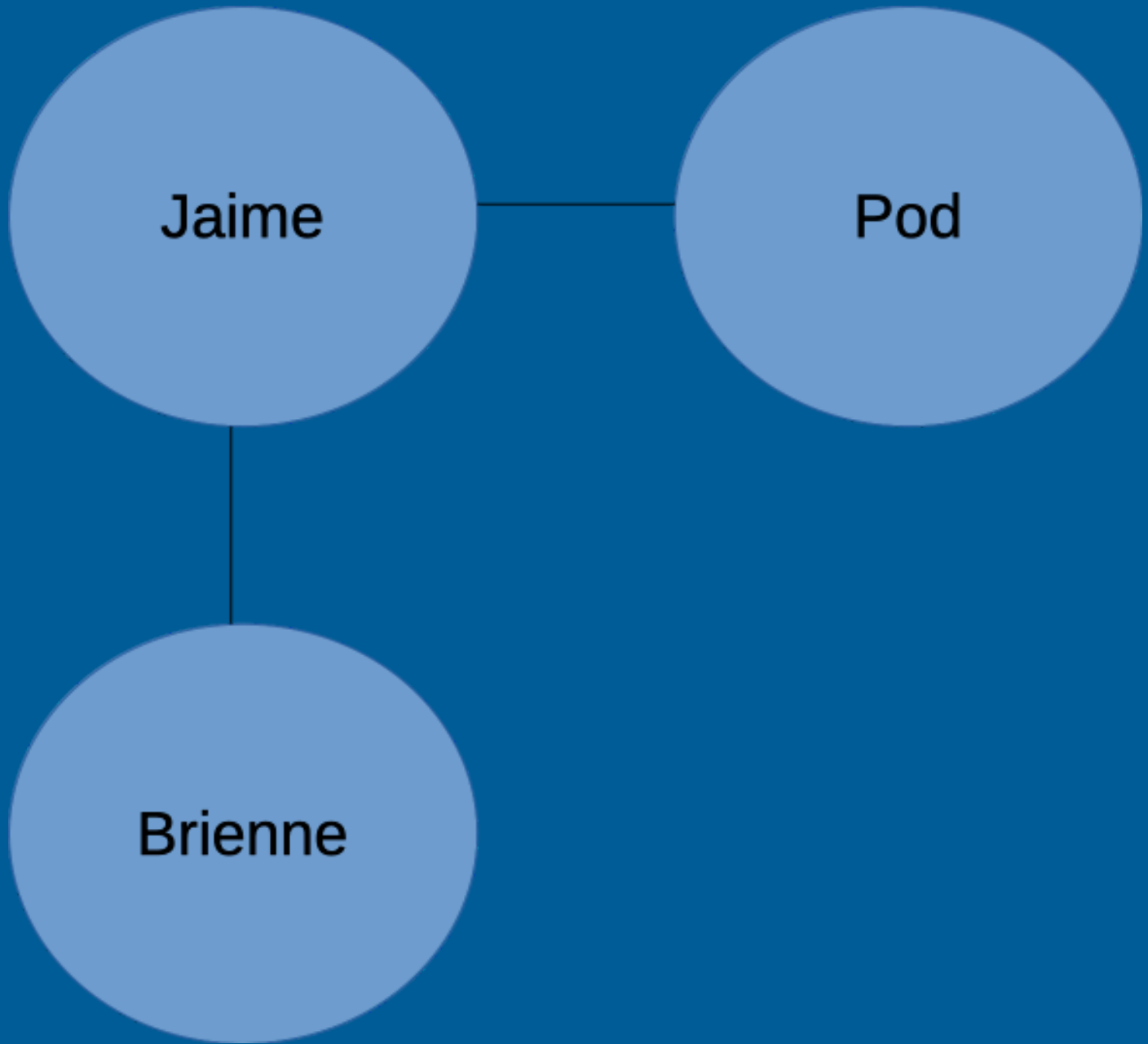
- Concept: If two people in a social network have a friend in common, then there is an increased likelihood that they will become friends themselves at some point in the future
- Example: If Jaime and Pod are friends, and Jaime and Brienne of Tarth are friends, then it is likely that Brienne and Pod will become friends

CC BY SA

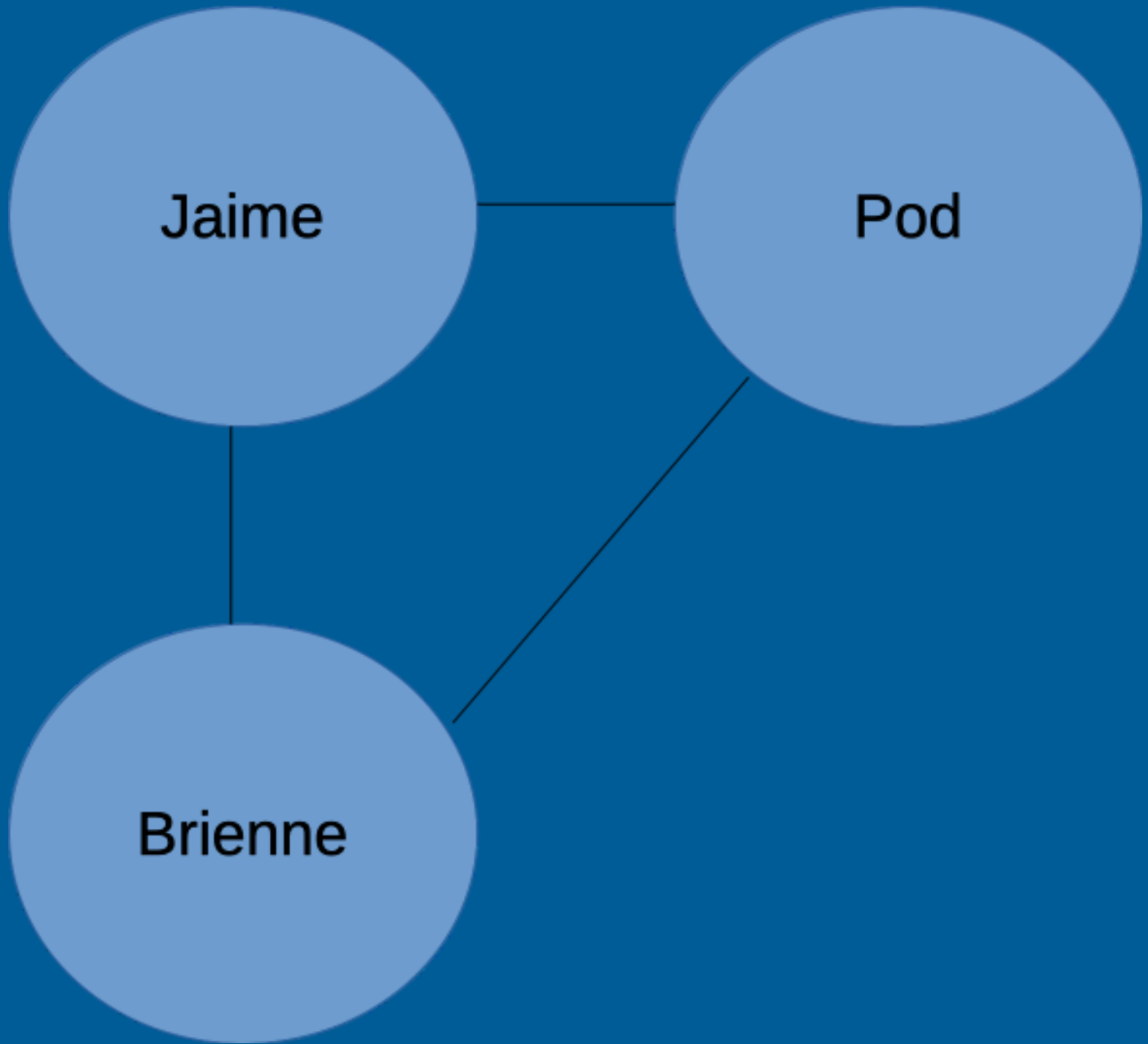
Triadic Closure

- This forms a triangle in the graph
- Generally form in social networks given enough time
- Think facebook friends feature
- Basic functionality of a social network

CC BY SA



CC BY SA



CC BY SA

Community Detection

- The number of connections between nodes is more dense in a certain grouping than outside it
- GOOD IDEA / common practice to make this the color of your nodes!

CC BY SA

Louvian Method

```
1
2 CALL gds.louvain.write(
3   'movies-graph',           // Graph name
4   {
5     writeProperty: 'community',   // The property where the communi
6     concurrency: 4              // Number of concurrent threads
7   }
8 ) YIELD communityCount, modularity;
```

CC BY SA

Breakdown

```
CALL gds.louvain.write()
```

- This invokes the *Louvain algorithm* from the Neo4j Graph Data Science (GDS) library.
  - The write variant of the algorithm means the community detection results will be written back to the graph (not just returned as a result).

CC BY SA

Breakdown

`'movies-graph'`

::: incremental

- The first argument specifies the graph projection you want to use. In this case, the algorithm will work on the ‘movies-graph’ projection.
  - This projection must already exist!

CC BY SA

Breakdown

`writeProperty: 'community'`

- *writeProperty* specifies the property where the community ID will be written.
- The algorithm will assign a community ID to each node (in this case, each node will receive a community label).
- The community ID will be stored in the property community for each node.

CC BY SA

Breakdown

`concurrency: 4`

- This parameter controls the number of threads used to run the algorithm.
- It specifies the number of concurrent threads (or processes) that will be used to parallelize the algorithm, making it faster on large graphs.

CC BY SA

Breakdown

```
YIELD communityCount, modularity
```

CC BY SA

Retrieve Communities

```
1 MATCH (p:Person)
2 RETURN p.name, p.community
3 ORDER BY p.community;
4
```

CC BY SA

Components

- Natural breaks for connected portions of a graph
- Connected component of a graph
  - Every node in the subset has a path to each other

CC BY SA

Giant Component

Informal definition:
Connected component that contains a significant fraction of all the nodes

- Game of Thrones: People who want Joffrey dead. Not **everybody**, but close!
- Most networks only have one giant component

CC BY SA

How to Find the Giant Component in Neo4j GDS

First, run weakly connected components (WCC)

- WCC is used to:
  - Identify islands or clusters of connectivity in a graph.
  - Detect isolated groups of nodes.
  - Understand fragmentation of a network.

CC BY SA

WCC

```
1  CALL gds.wcc.stream('movies-graph-undirected')
2  YIELD nodeId, componentId
3  RETURN *,gds.util.asNode(nodeId).name AS name,componentId
4  ORDER BY componentId
5  LIMIT 50;
```

CC BY SA

Write to node

```
CALL gds.wcc.write('movies-graph', {
  writeProperty: 'componentId'
});
```

CC BY SA

Get the size of each component

```
1 MATCH (n)
2 WITH n.componentId AS component, count(*) AS size
3 RETURN component, size
4 ORDER BY size DESC;
```

CC BY SA

Assuming 0 is the largest

```
1  MATCH (n)
2  WHERE n.componentId = 0
3  RETURN n
```

CC BY SA

Merger of Giant Components

- Only one connection merges giant components into one
  - In history: Sudden, often catastrophic change
  - Think of 1492 C.E.
    - Disease
    - Political change
    - Previous contacts were not sustained
- Issue of time

CC BY SA

Thank You!

Any Questions?

CC BY SA