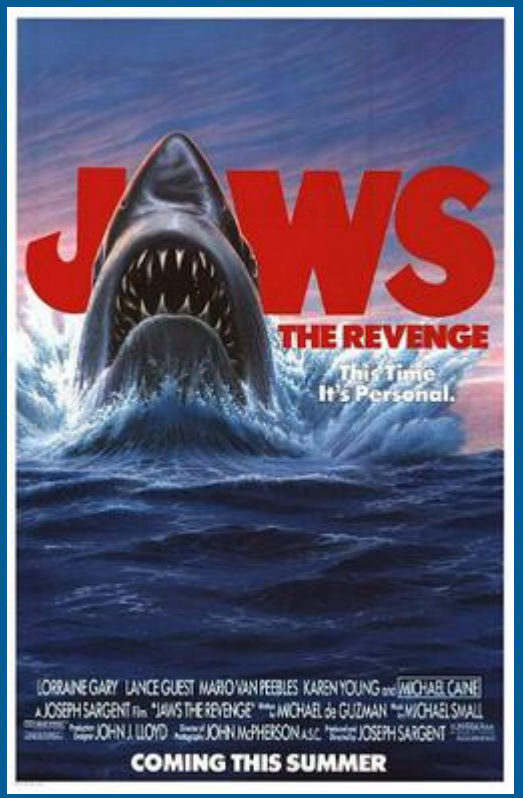
Thursday, May 15, Practical Session IV: This Time, It's Personal

Ryan Horne
Advanced Research Computing, UCLA
ryan.matthew.horne@gmail.com

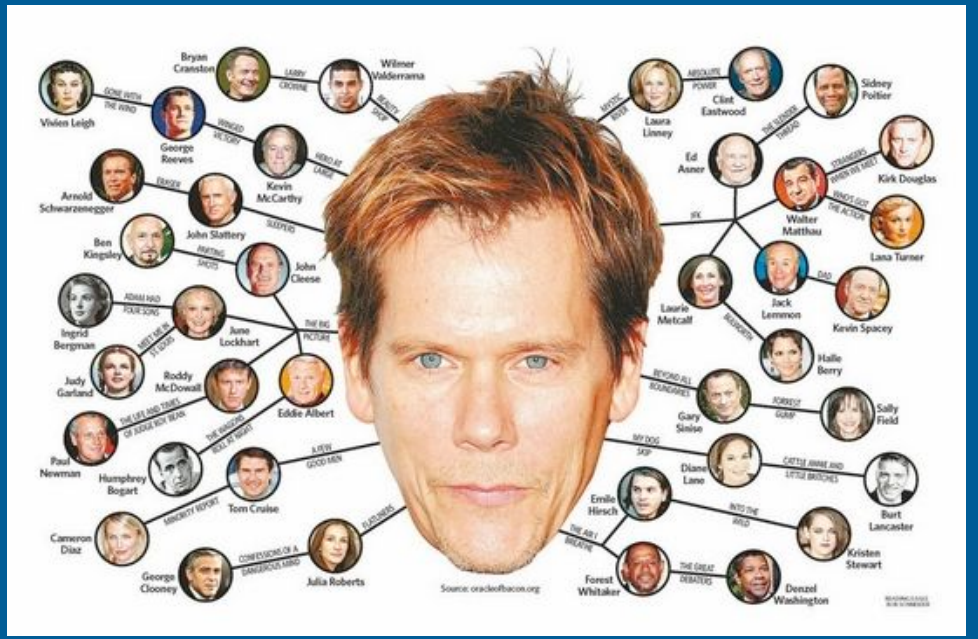
Agenda

- More network stats and measurements
- Visualizing with Neo4j



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?



Degree Measurements

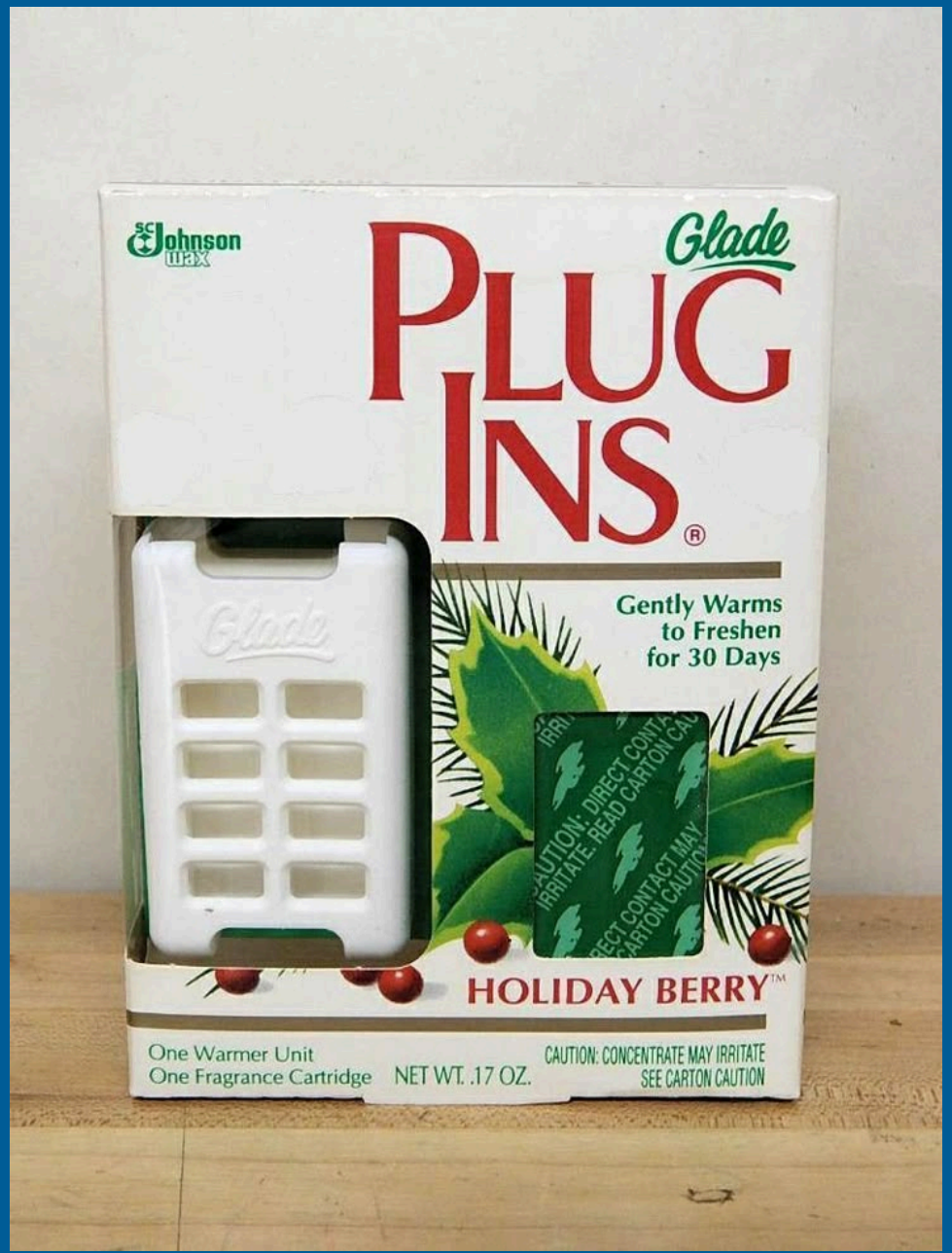
- Sum of all other nodes with a direct *path* to a node
- Signifies activity or popularity
- Very good for looking at nodes in a local context
- In & Out
  - Weighted and unweighted
  - Hmmm….In-N-Out Burgers…..

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

Plugin Time!

- We *can* use Cypher queries to make these measurements
- However there is an easier way!
  - we need to enable the *Graph Data Sciecne Library*



Before We Begin

- We need to put the graph in memory
- This is called *projecting* the graph

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

Breakdown

`'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’).

Breakdown

```
['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.

Breakdown

```
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.

Breakdown

- `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.

What Does This Do?

```
1
2  CALL gds.graph.project(
3    'movies-graph-3',
4    ['Person', 'Movie'],
5    {
6      ACTED_IN: {
7        type: 'ACTED_IN',
8        orientation: 'NATURAL'
9      },
10     DIRECTED: {
11       type: 'DIRECTED',
12       orientation: 'NATURAL'
13     },
14     PRODUCED: {
15       type: 'PRODUCED',
16       orientation: 'NATURAL'
17     }
18   }
```

Now Make an *Undirected* Graph

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

Measurements

Degree

Betweenness

Closeness

Eigenvector

Degree Measurement

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;
```

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.

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).

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).

Breakdown

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

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

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.

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;
```

Some notes:

- Degree counts *outgoing* edges by default.
- `UNDIRECTED`, as we mentioned before, collapses all edge direction

Can you do this for people?

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

In-Degree Measurement

The number of connections going *into* a node.

```
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;
```

Note the use of REVERSE

Out-Degree Measurement

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

Note the use of NATURAL

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

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;
```

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;
```

Betweeness Centrality

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

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

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
```

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!

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
```

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

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;
```

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**	

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

Bridges

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

Python: Bridges

Python Code | Result

```
import networkx as nx
import csv
import pandas as pd
from community import community_louvain
import matplotlib.cm as cm
import matplotlib.pyplot as plt

# First, get the edge data
url = 'https://raw.githubusercontent.com/mathbeveridge/gameofthrones/master/data/got-s1-edges.csv'
dfedges = pd.read_csv(url)
# Dataset is now stored in a Pandas Dataframe

# Now we create the graph from the edge list. We need to specify the column names as they are in mix
G = nx.from_pandas_edgelist(dfedges, source="Source", target = "Target", edge_attr=True)
bridges = list(nx.bridges(G))
list(nx.bridges(G))
```

Python: Highlight Bridges

Python Code | Result

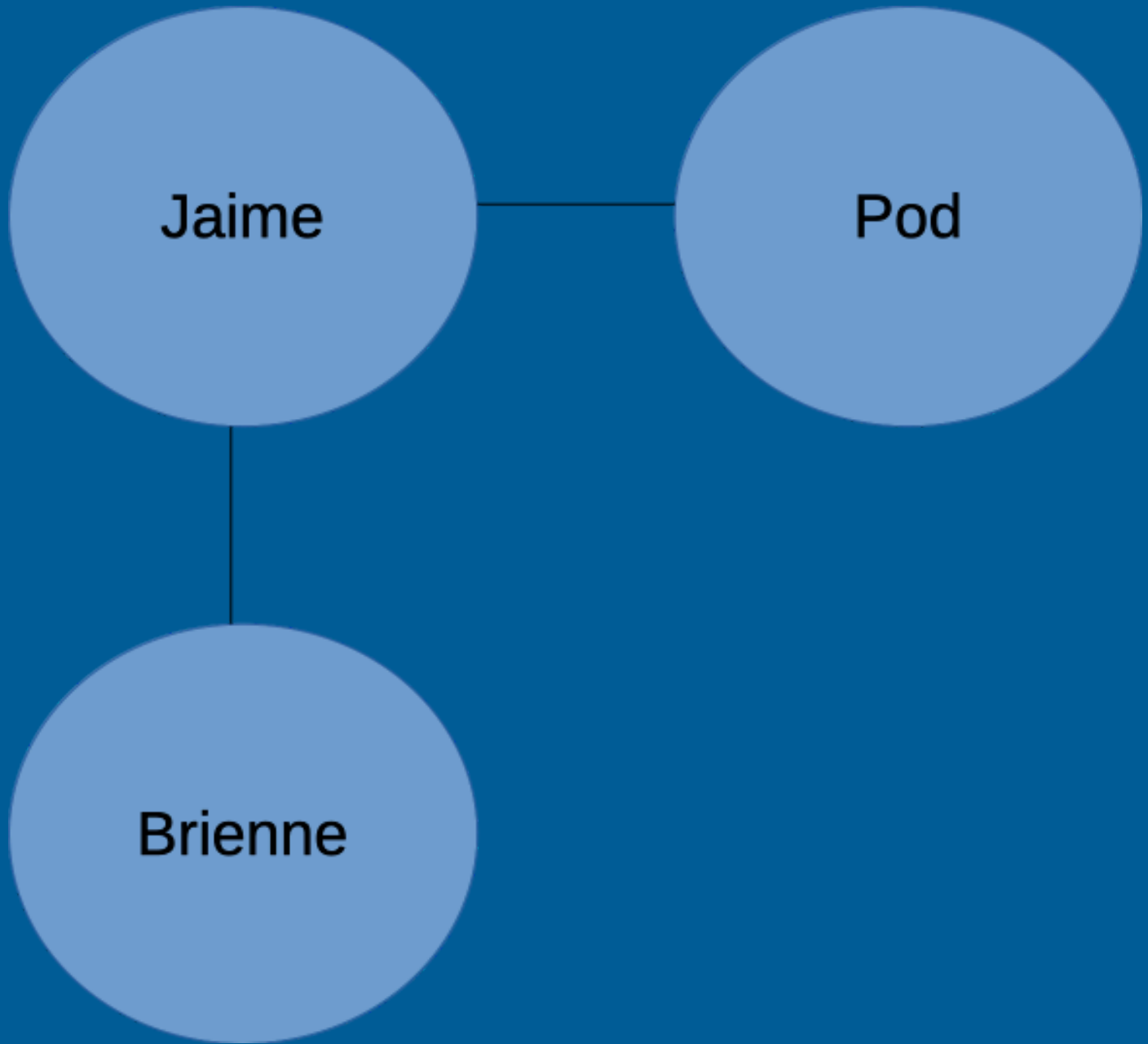
```python
# Identify the bridges
bridges = list(nx.bridges(G))

# 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()
```

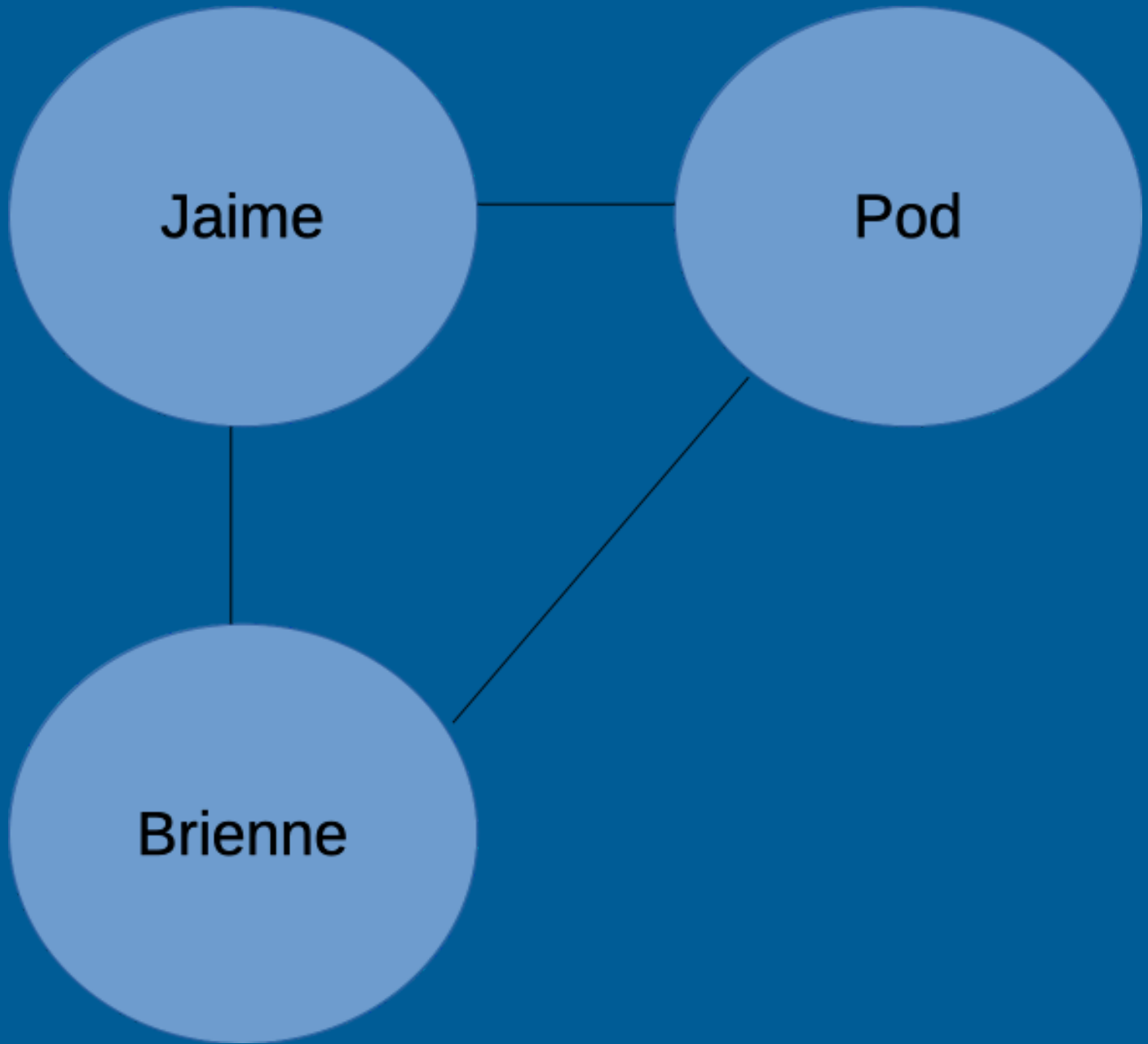
Network Evolution: Triadic Closure

- 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

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





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!

Louvian Method

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

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).

Breakdown

`'movies-graph'`

- 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!

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.

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.

Breakdown

```
YIELD communityCount, modularity
```

Retrieve Communities

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

Remember Degree Measurement?

Total Degree all Nodes

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

How Do We Write to the Node?

```
CALL gds.degree.write('movies-graph', {
  orientation: 'UNDIRECTED',
  writeProperty: 'totalDegree'
})
YIELD nodePropertiesWritten;
```

- `nodePropertiesWritten`
  - Number of nodes that had the totalDegree property written to them.

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

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

How to Find the Giant Component in Neo4j GDS

First, run weakly connected components (WCC)

WCC is an algorithm that identifies clusters (also called components) in a directed graph, where every node in a cluster is connected to every other node if you ignore the direction of relationships.

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

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;
```

Write to node

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

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;
```

Assuming 0 is the largest

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

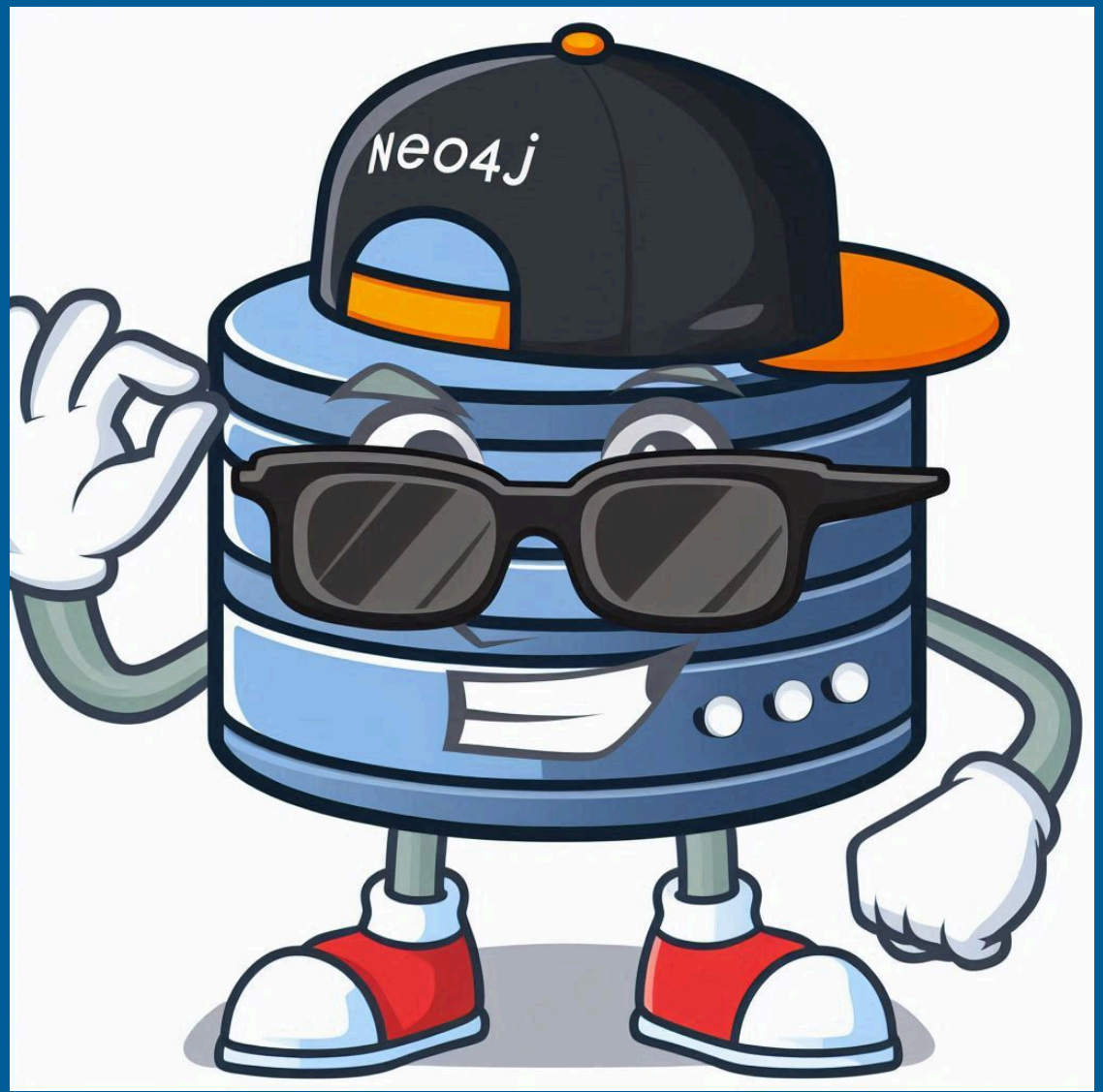
Merger of Giant Components

- 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

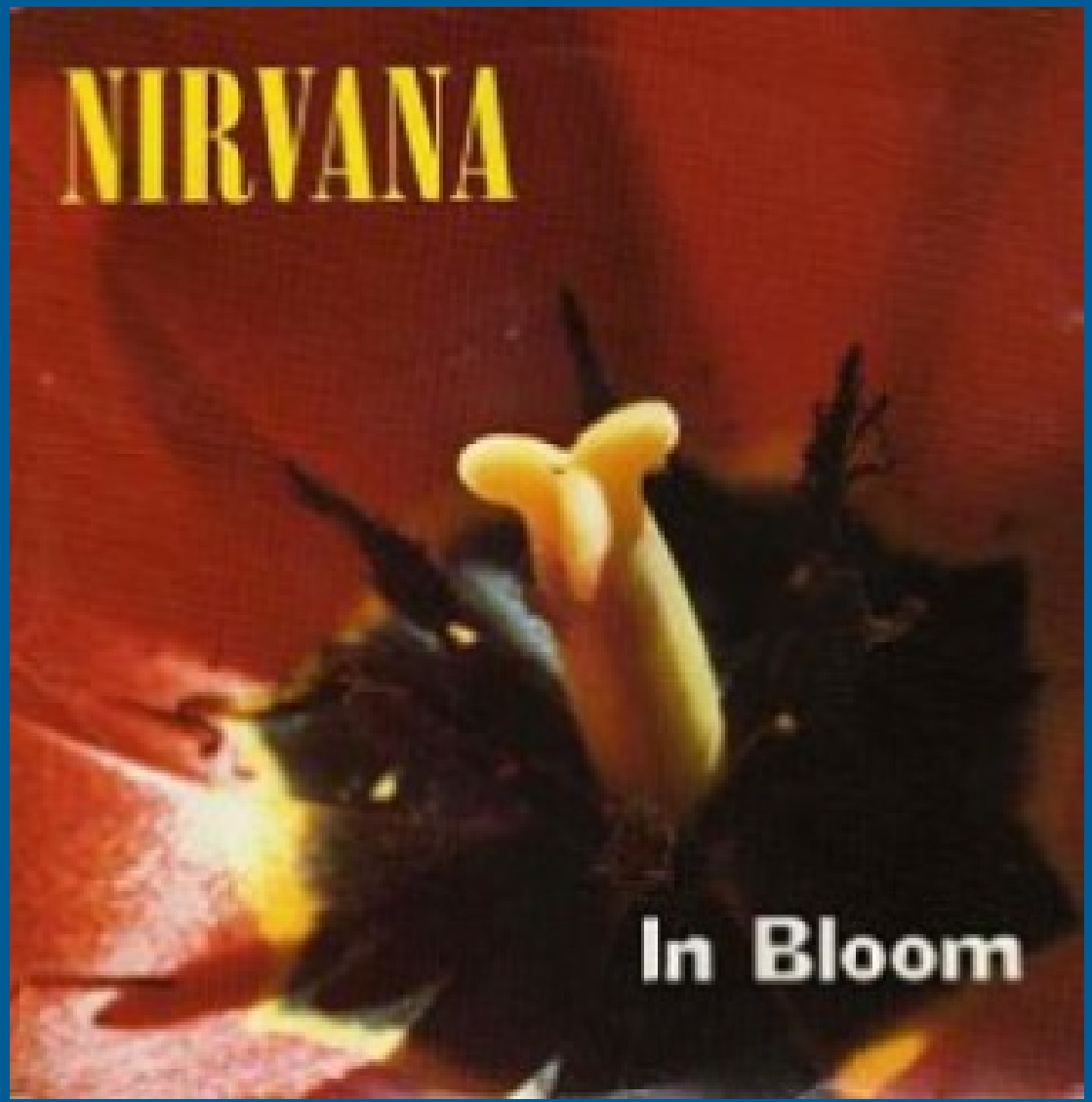
Now That you Know Measurements

We are going to focus on visualizing the data!

We Will Make Neo4J Look Good



Neo4j: Bloom



Neo4j: Bloom

“A beautiful and expressive data visualization tool to quickly explore and freely interact with Neo4j’s graph data platform with no coding required.”

Yeah, sure, kind of.

First Start a New Bloom Session

Open *drop down* and select Neo4J Bloom

- Generate a prespective if you have nothing

Change The Basic Appearance

We are going to add some icons and change the colors

Show Your Graph Data!

- Change node size based on degree
- Change node color based on community

Thank You!

Any Questions?