

A close-up photograph of a person's hand holding a smartphone. The screen of the phone displays a network graph with several nodes (represented by circles of varying sizes) and connecting edges. In the background, through the phone's screen, another person's hands are visible, also interacting with a similar network graph interface. The overall theme is data connectivity and digital interaction.

# Introduction to Neo4j

# SetUp

- First go to [https://s3.amazonaws.com/neo4j-sandbox-usecase-datastores/v3\\_5/yelp.db.zip](https://s3.amazonaws.com/neo4j-sandbox-usecase-datastores/v3_5/yelp.db.zip)
- Unzip the file
- Create a new Local Graph
- Hit the Manage button followed by the Open Folder icon at the top
- Click on the data folder, followed by databases
- Copy and paste the yelp.db folder into /data/databases
- Change the name from yelp.db to graph.db
- Start the graph
- Click the Graph Algorithms Playground application at the top
- Click the lower left yellow icon
- Load the Game of Thrones and European Roads data into your yelp database which should be active currently

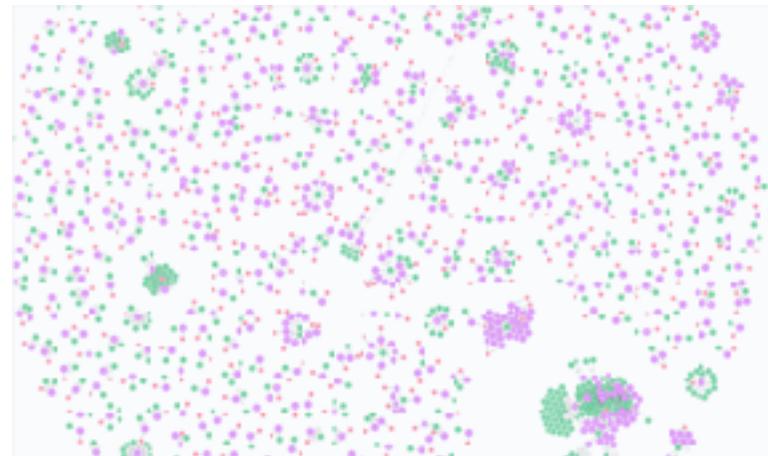
# What is Neo4j?



## *Graph Platform*

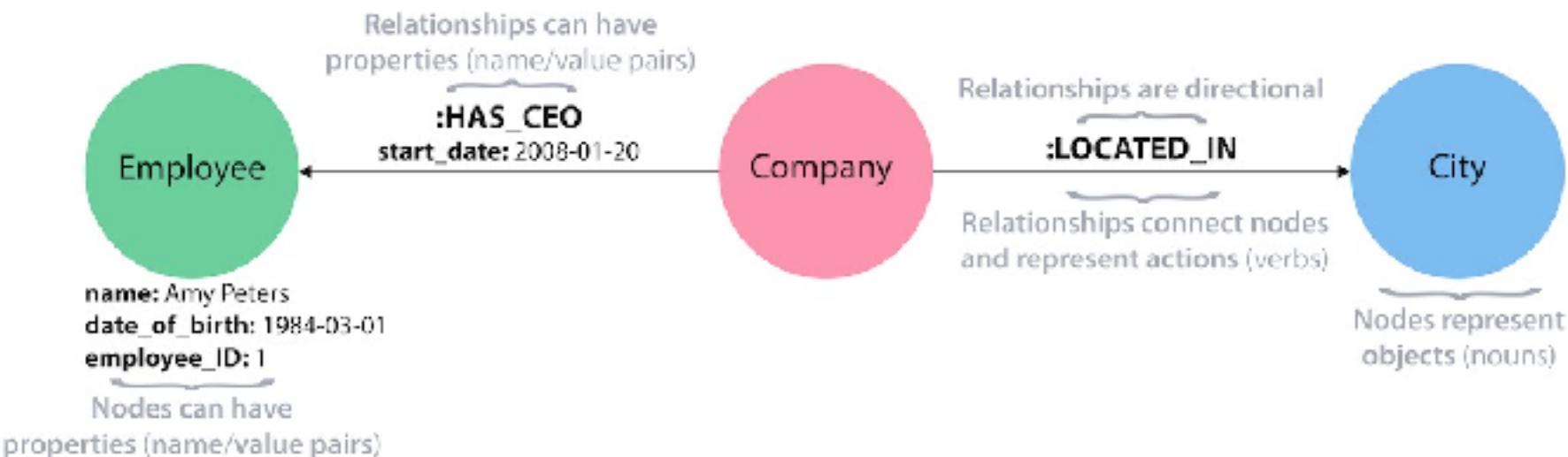
- Database management system (DBMS)
- Property Graph data model
- Cypher query language
- Graph analytics
- Data visualization
- Developer tool for building applications

```
1 MATCH (a:Address)<-[r:REGISTERED_ADDRESS]-(o:Officer)-->(e:Entity)
2 WHERE a.address CONTAINS "New York"
3 RETURN *
```



[neo4j.com/](http://neo4j.com/)

# Labeled Property Graph



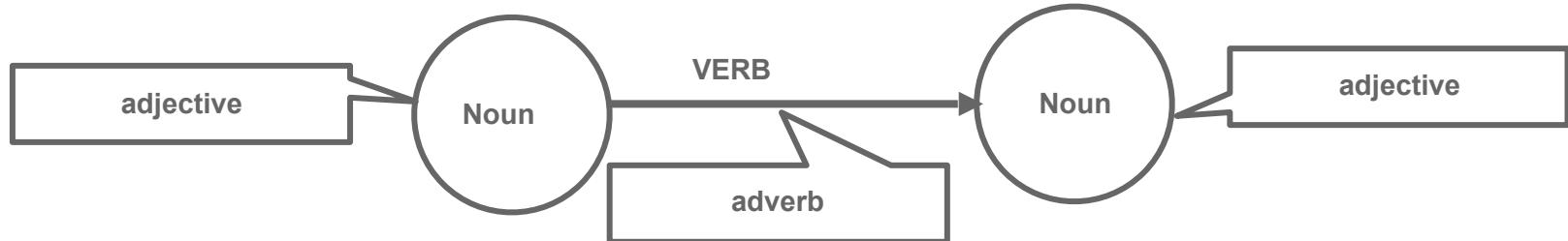
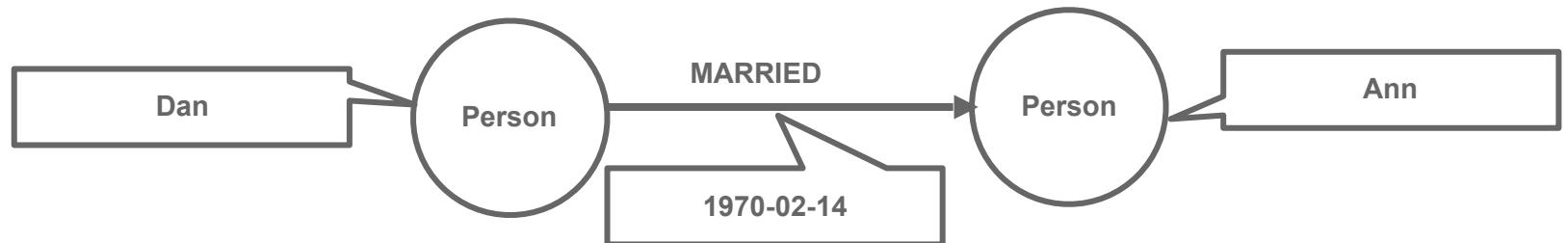
# What is Cypher?

- Declarative
- Focuses on **what**, not how
- ASCII art syntax

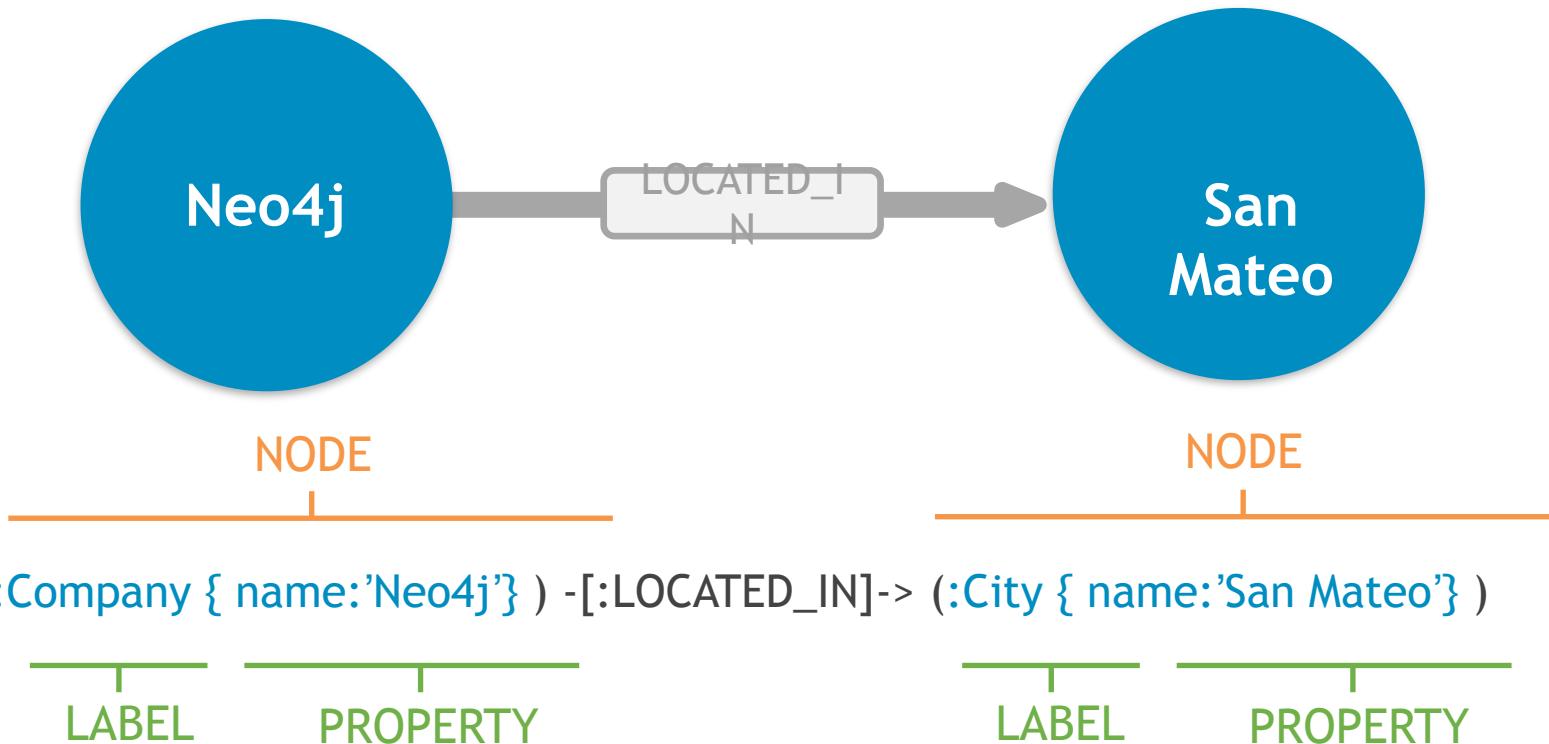


**MATCH (A) - [:LIKES] -> (B)**

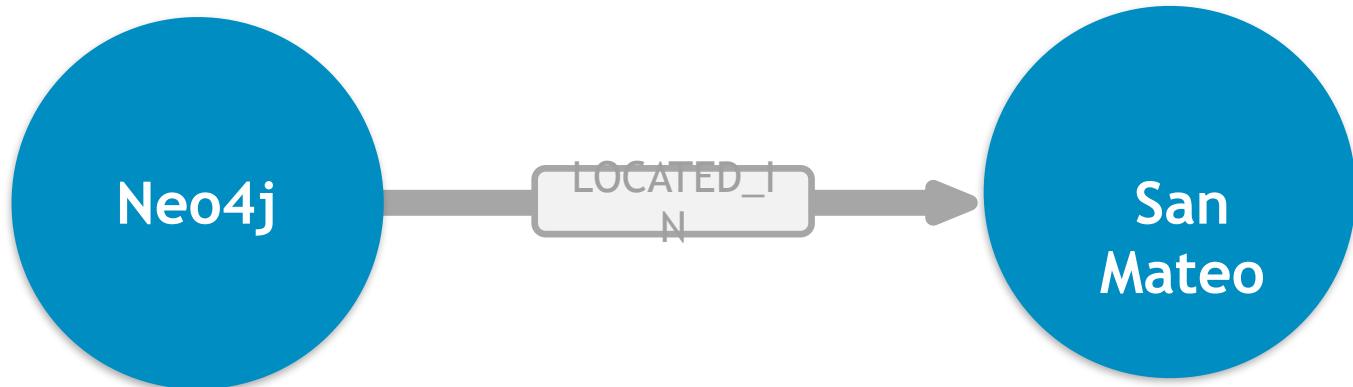
# Cypher is readable



# Cypher Query Language



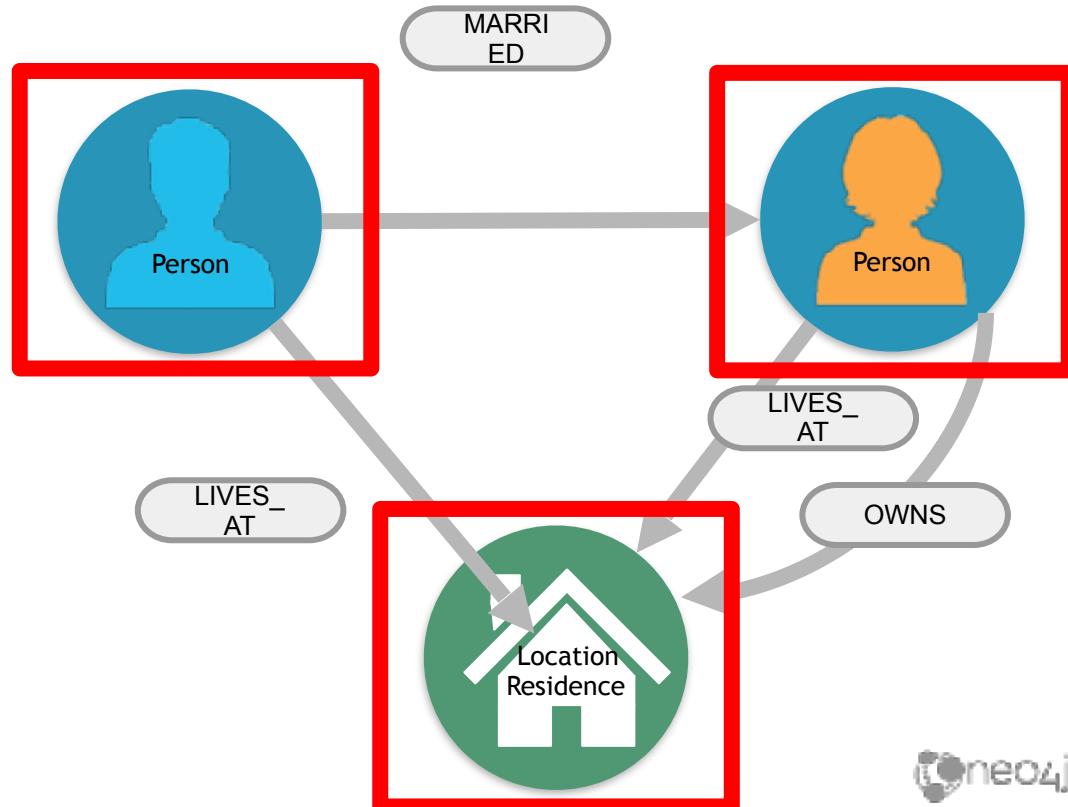
# Cypher Query Language



```
MATCH (:Company { name:'Neo4j' }) -[:LOCATED_IN]->  
(place)  
RETURN place
```

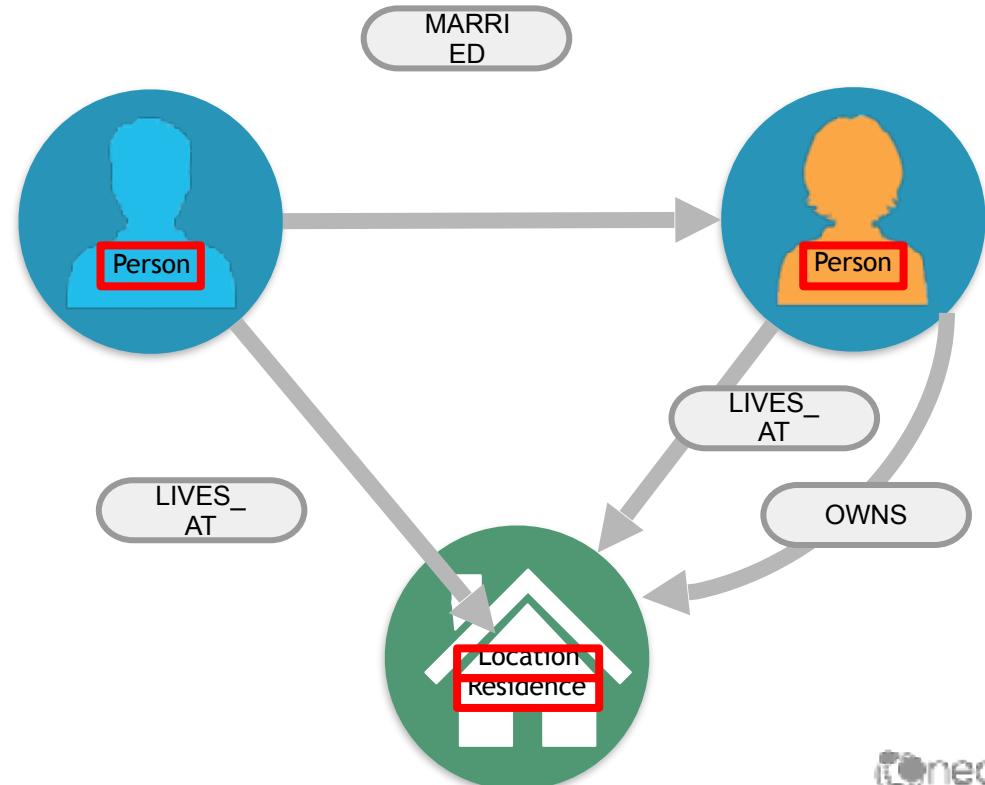
# Nodes

( )  
(p)  
(l)  
(n)



# Labels

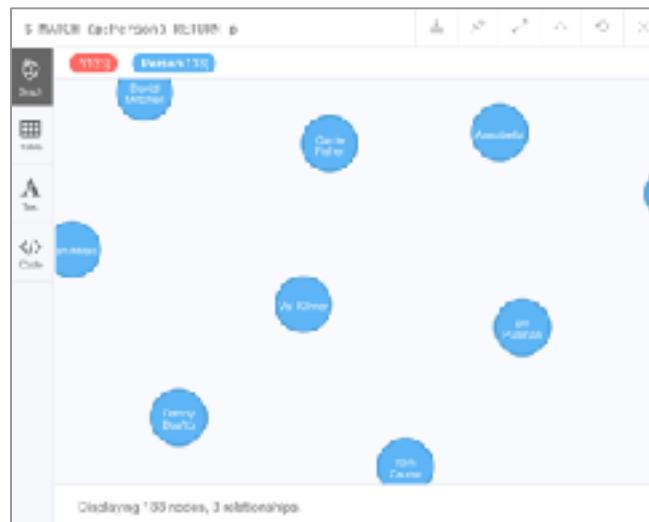
```
(:Person)  
(p:Person)  
(:Location)  
(l:Location)  
(n:Residence)  
(x:Location:Residence)
```



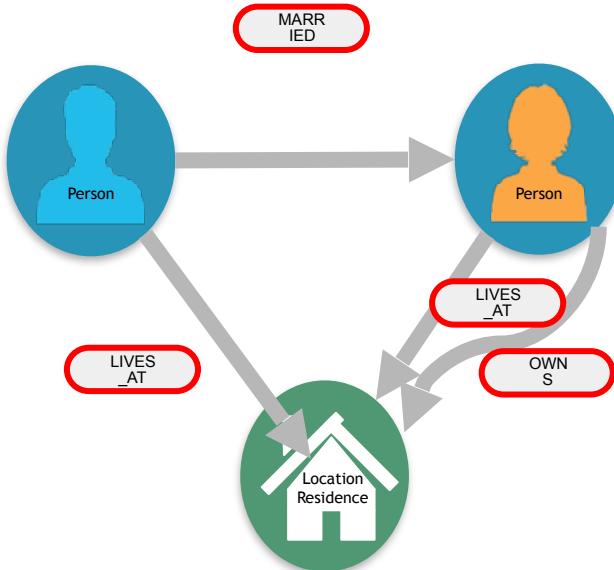
# Using MATCH to retrieve nodes

```
MATCH (n) // returns all nodes in the graph  
RETURN n
```

```
MATCH (p:Person) // returns all Person nodes in the graph  
RETURN p
```



# Querying using relationships

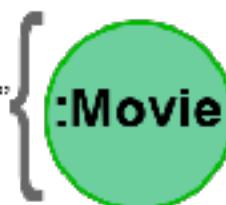


```
MATCH (p:Person) -[:LIVES_AT]-> (h:Residence)  
RETURN p.name, h.address
```

```
MATCH (p:Person) -[:LIVES_AT]-(h:Residence) //undirected  
RETURN p.name, h.address
```

# Properties

**title:** "Something's Gotta Give"  
**released:** 2003



**title:** 'V for Vendetta'  
**released:** 2006  
**tagline:** 'Freedom! Forever!'



**title:** 'The Matrix Reloaded'  
**released:** 2003  
**tagline:** 'Free your mind'

# Retrieving nodes filtered by a property value

Find all *people* born in 1970, returning the nodes:



```
MATCH (p:Person {born: 1970})  
RETURN p
```

```
MATCH (p:Person)  
WHERE p.born = 1970  
RETURN p
```

# Using patterns for queries



Find all people who *Angela Scope* follows, returning the nodes:

```
MATCH (:Person {name:'Angela Scope'}) - [:FOLLOWS] -> (p:Person)  
RETURN p
```

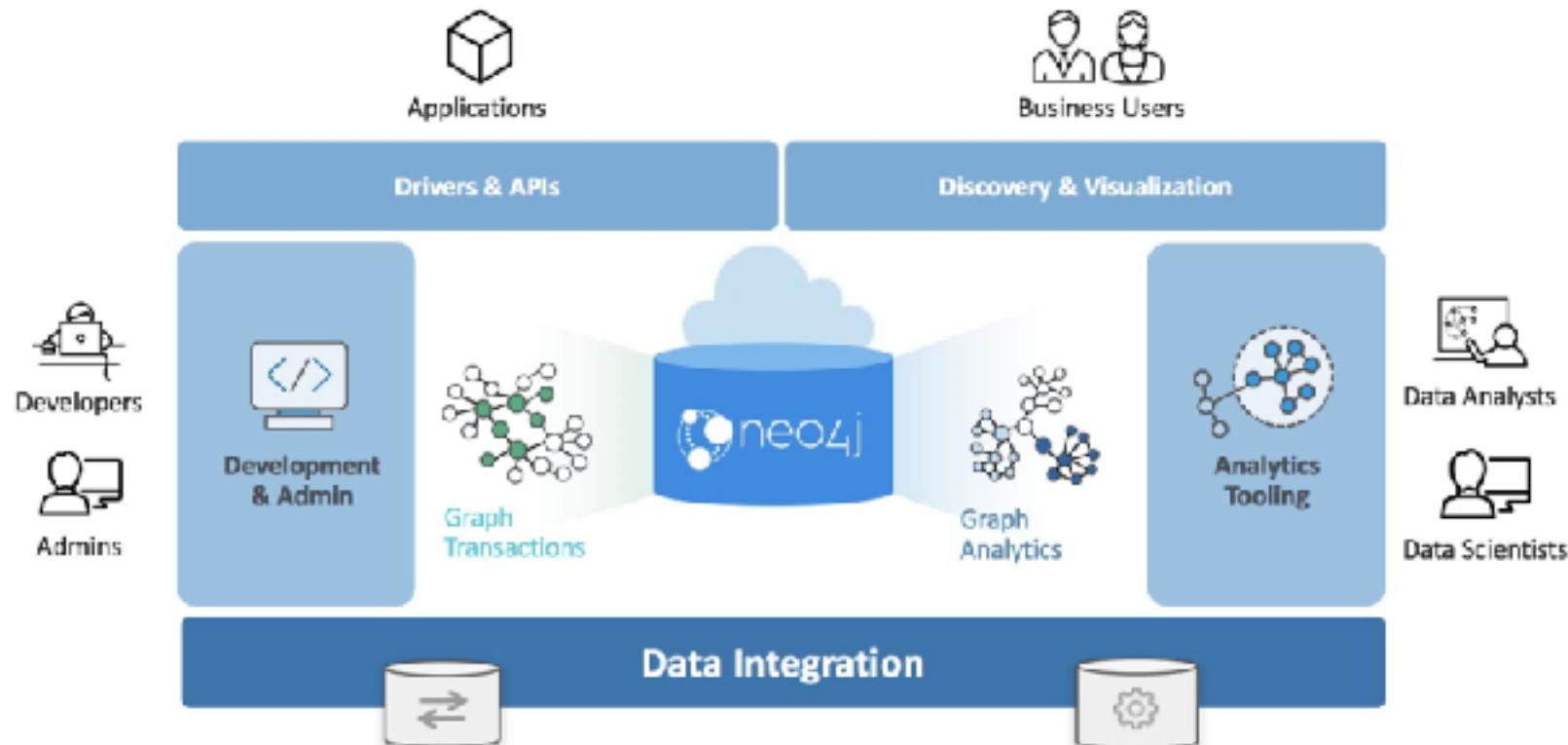
# Traversing relationships

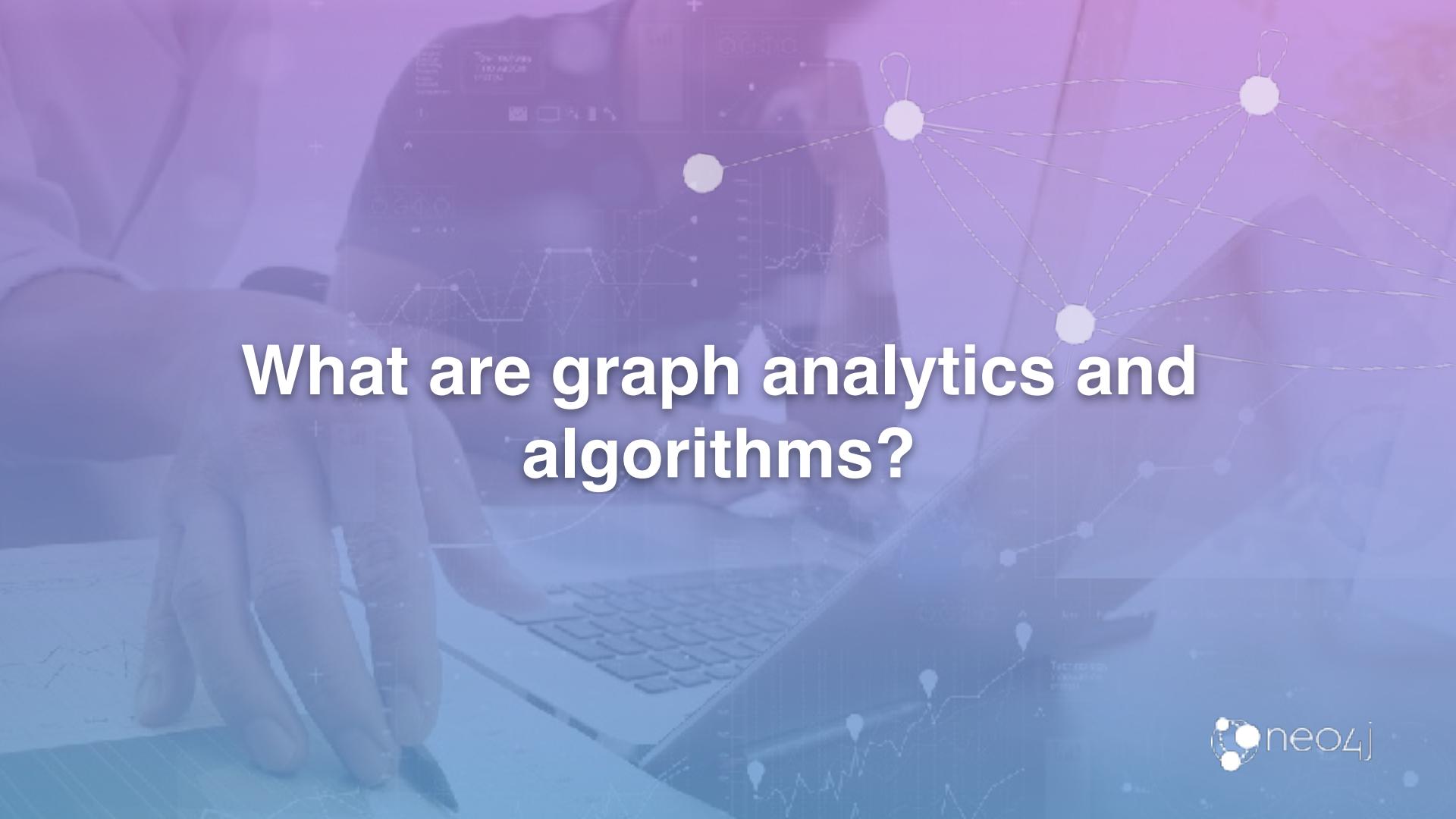


Find persons who follow persons who follow  
*Jessica Thompson* returning the people as nodes:

```
MATCH (p:Person) -[:FOLLOWS]->(:Person)-[:FOLLOWS]->(:Person {name:'Jessica Thompson'})  
RETURN p
```

# Neo4j Graph Platform architecture





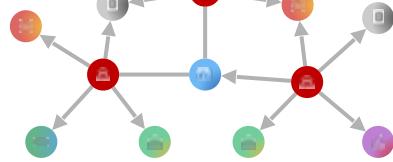
# What are graph analytics and algorithms?

# Graph analytics

## Query (e.g. Cypher/Python)

Real-time, local decisioning  
and pattern matching

Local  
patterns

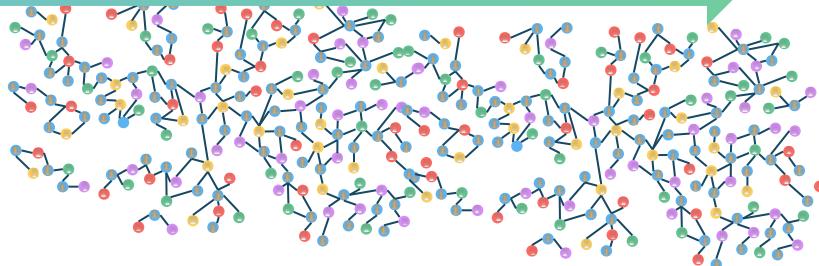


You know what you're looking  
for and making a decision

## Graph Algorithms libraries

Global analysis  
and iterations

Global  
computation



You're learning the overall structure of a  
network, updating data, and predicting

# Don't need graph algorithms to answer . . .

- Questions with just a few connections or flat (not nested).
- Questions solved with specific, well-crafted queries.
- Simple statistical results (sums, averages, ratios).
- Example:
  - Regular reporting based on defined criteria and well-organized data

# Common types of graph algorithms



Pathfinding  
& Search



Centrality /  
Importance



Community  
Detection+

Decomposition,  
covering &  
coloring

Network flow  
& percolation

Subgraph &  
isomorphism

Basic stats



Similarity



Link  
Prediction



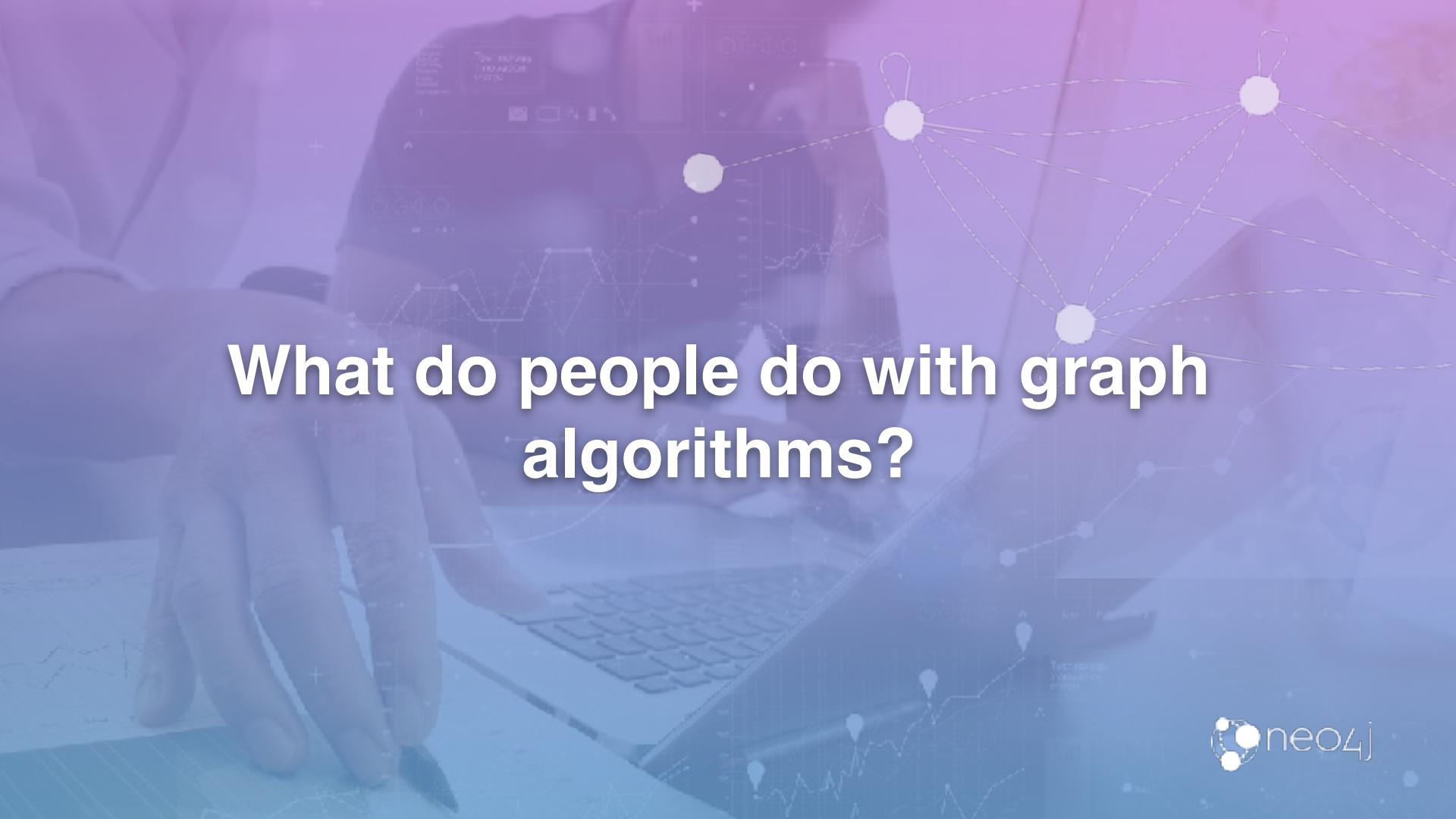
ML  
Workflow

Assortative  
mixing

Classic graph algorithms categories

Other common categories

So many others!

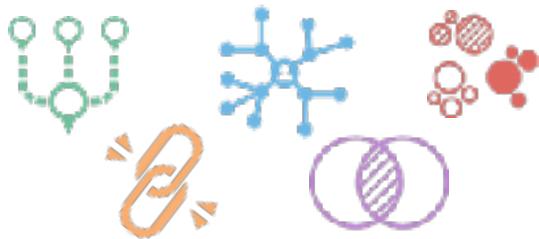


# What do people do with graph algorithms?

# Using graph algorithms

## Explore, plan, measure

Find significant patterns and plan for optimal structures



Score outcomes and set a threshold value for a prediction

## Machine learning

Use the measures as features to train an ML model

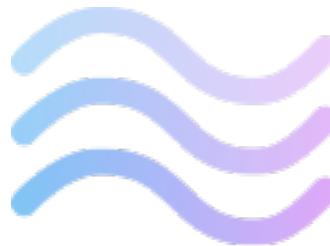
1st node	2nd node	Common neighbors	Preferential attachment	Label
1	2	4	15	1
3	4	7	12	1
5	6	1	1	0

# Understand and predict complex behavior

Propagation  
pathways



Flow &  
dynamics



Interactions &  
resiliency



Requires understanding relationships and structures

# Common graph data science applications

Financial crimes



Drug discovery



Recommendations



Customer segmentation



Cybersecurity



Churn prediction



Predictive maintenance

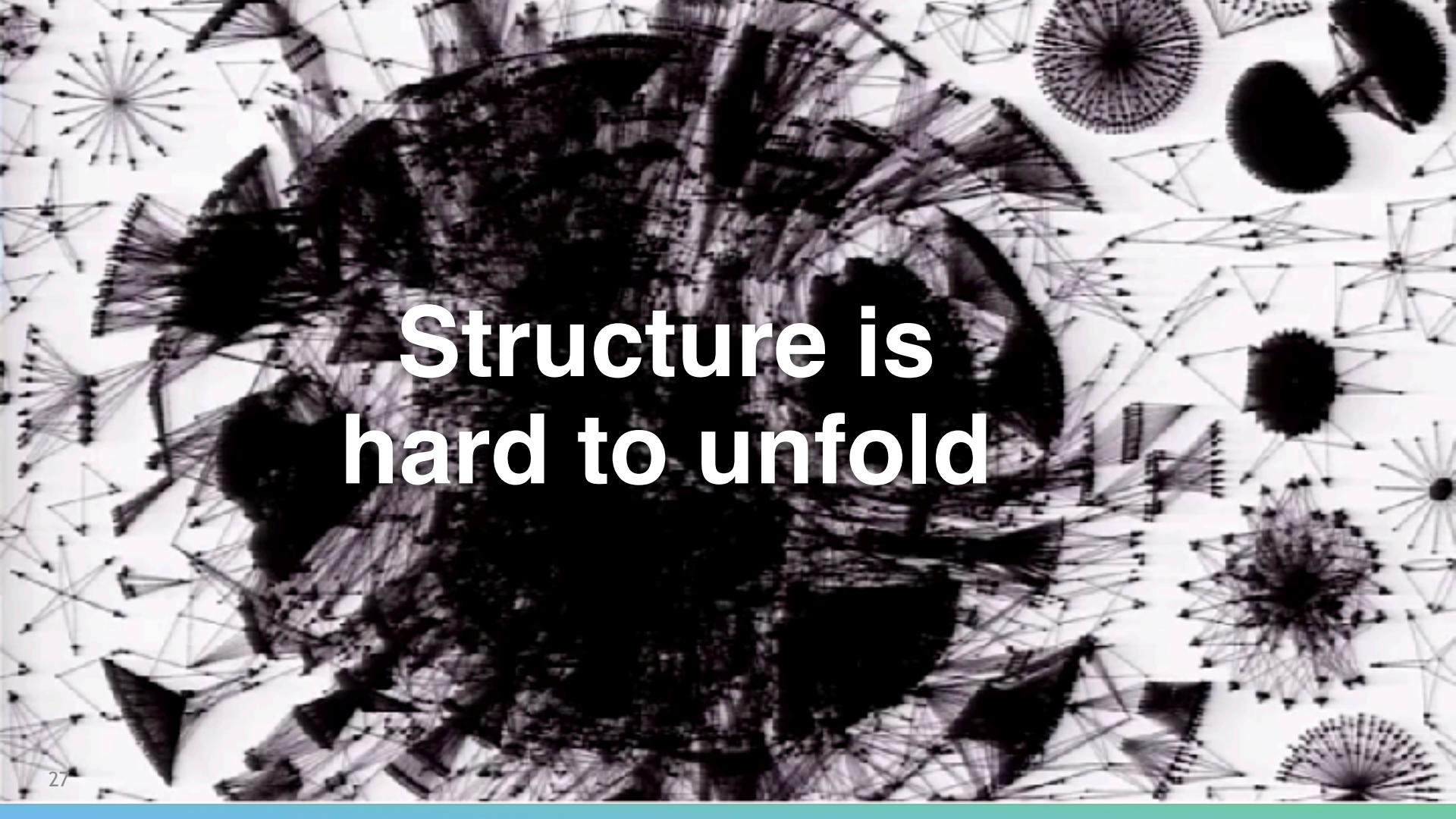


Search & MDM





# Why not use other types of analytics?



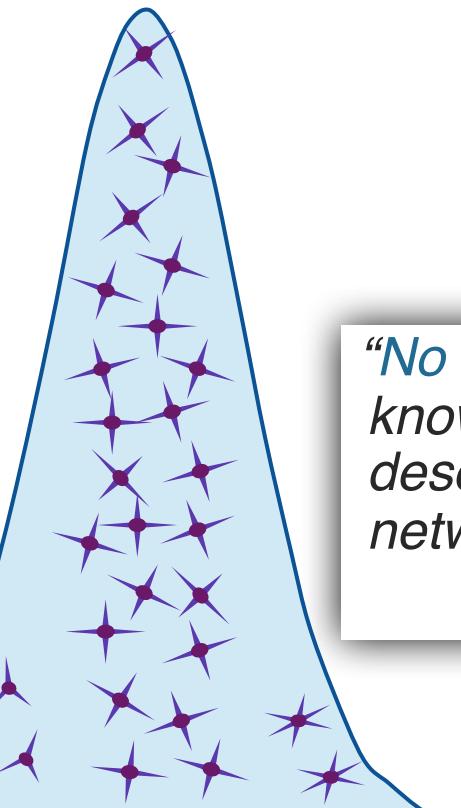
Structure is  
hard to unfold

# Random network

Number of NODES

## Average distribution

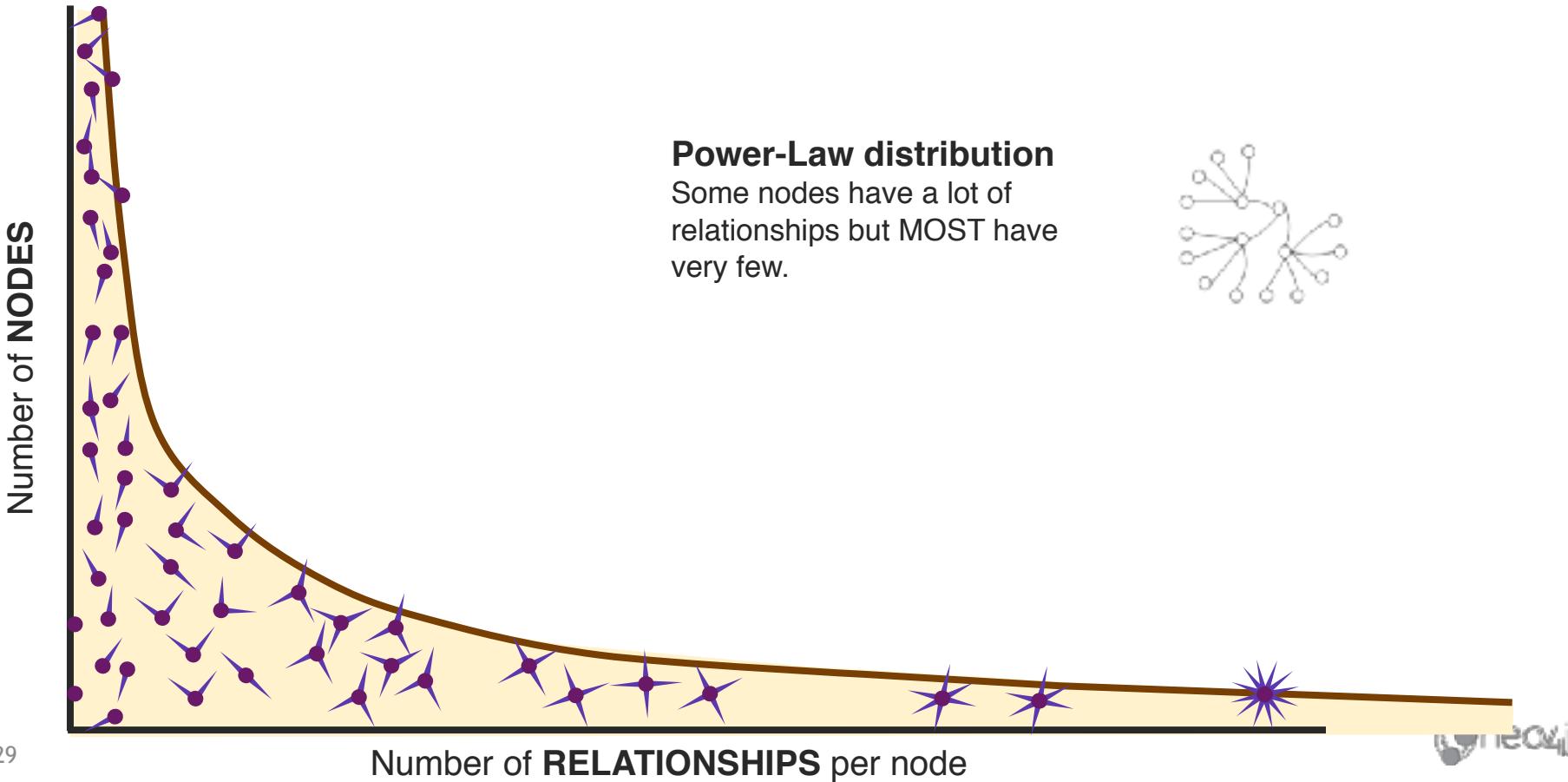
Most nodes have the same number of relationships



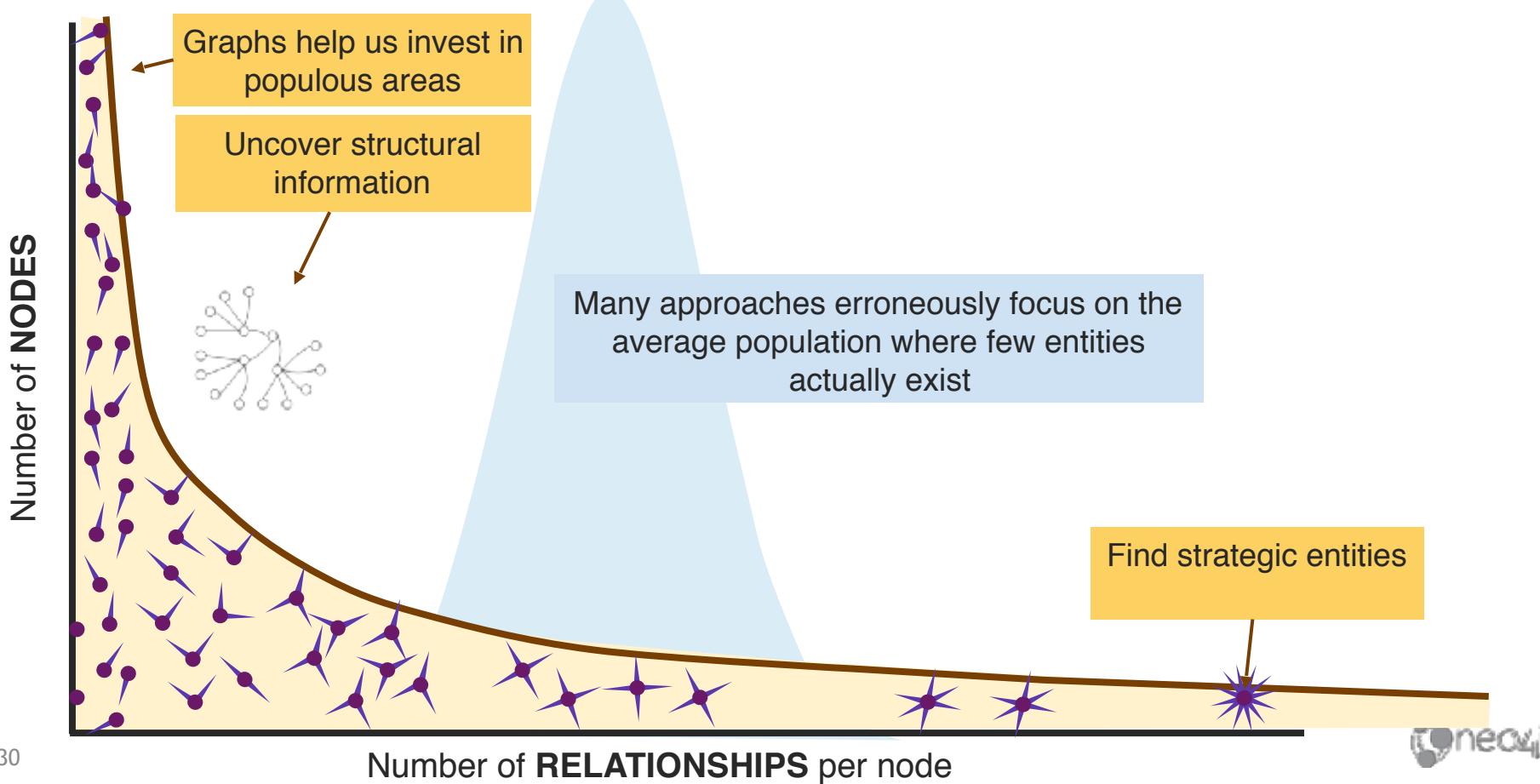
*"No network in nature that we know of that would be described by the random network model."*

—Albert-László Barabási

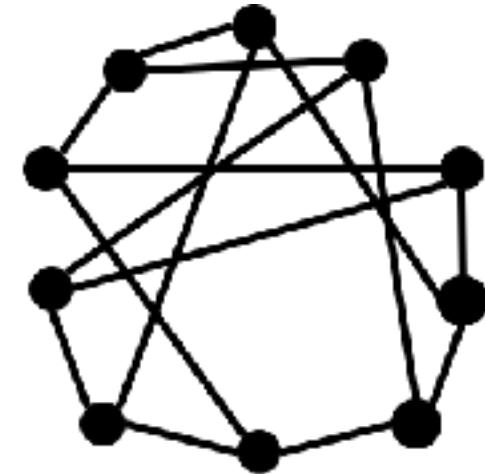
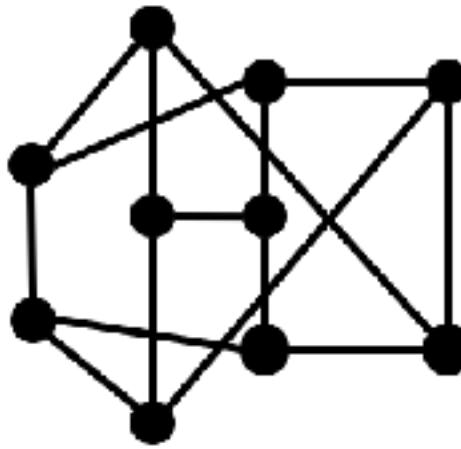
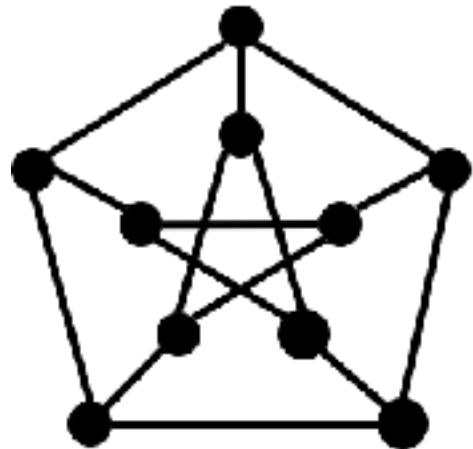
# Structured network



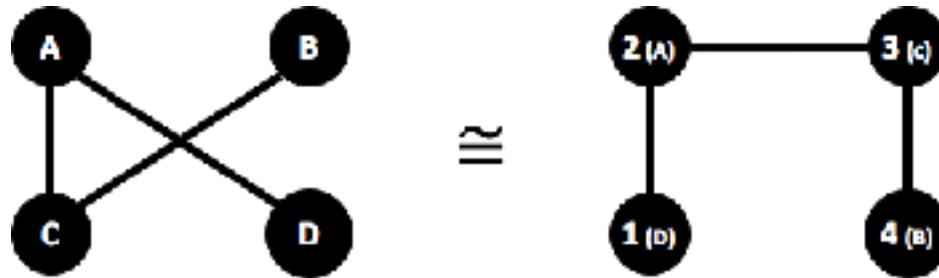
# Structure and density are important



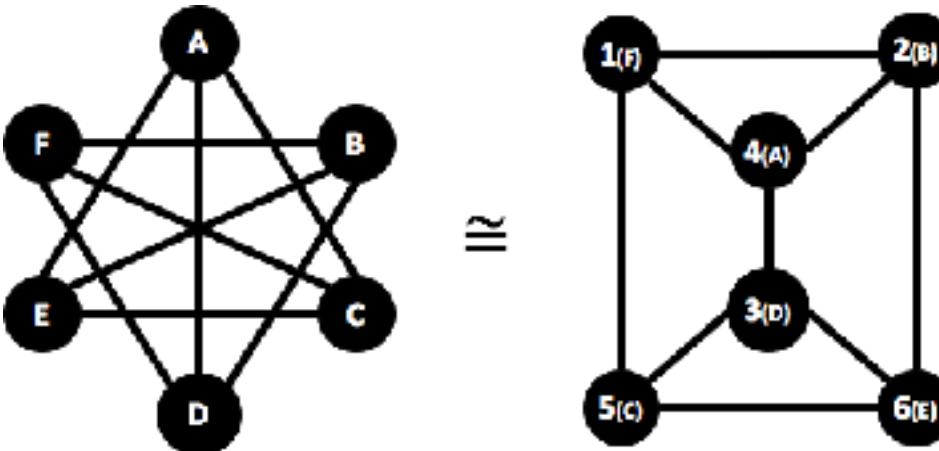
# What do these graphs have in common?



# Understanding graph complexity



Simple isomorphisms can be visualized.

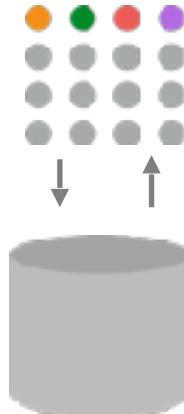


Anything more complicated requires graph algorithms to find.

# Native graph platforms are designed for connected data

## TRADITIONAL PLATFORMS

Store and retrieve data

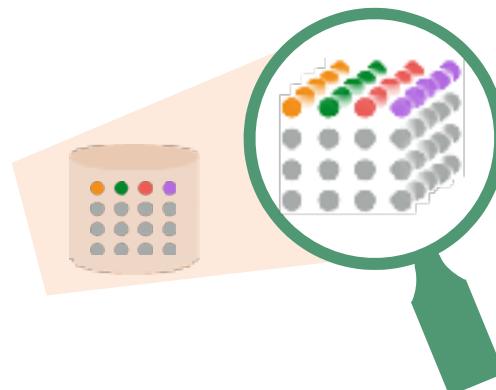


Real time storage & retrieval

*Max # of hops ~3*

## BIG DATA TECHNOLOGY

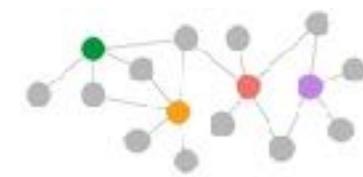
Aggregate and filter data



Long running queries  
aggregation & filtering



Connections in data

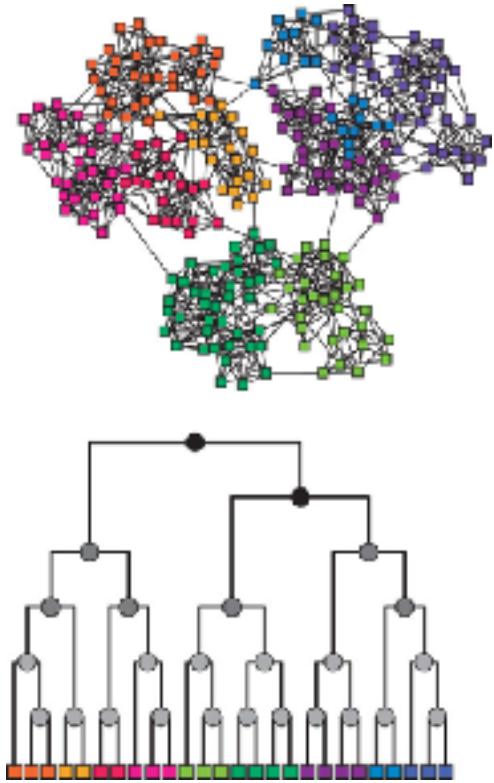
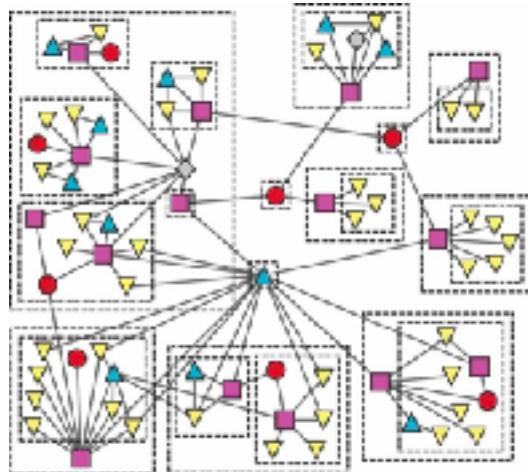
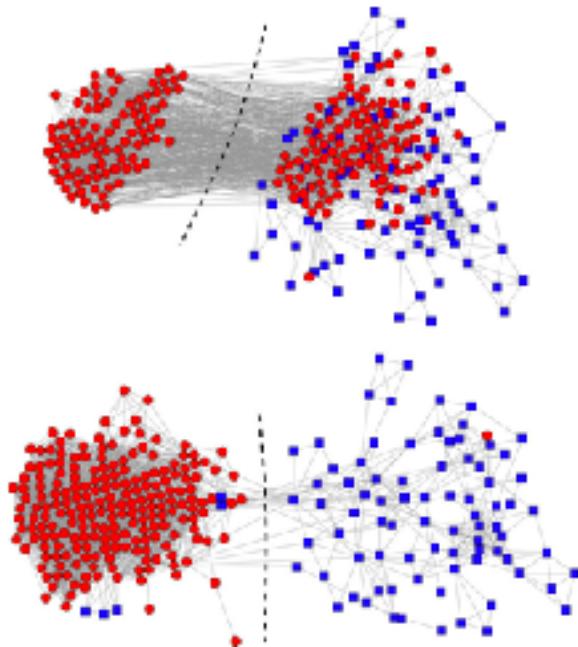


*Millions*

Real-Time Connected Insights



# Graph algorithms: Extract structure and infer behavior



Source: "Communities, modules and large-scale structure in networks" - Mark Newman

Source: "Hierarchical structure and the prediction of missing links in networks"; "Structure and inference in annotated networks" - A. Clauset, C. Moore, and M.E.J. Newman.

# Neo4j Graph Algorithms Library

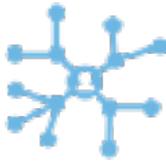


# +35 Graph & ML algorithms in Neo4j



## Pathfinding & Search

Finds optimal paths or evaluates route availability and quality.



## Centrality / Importance

Determines the importance of distinct nodes in the network.



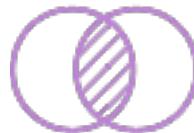
## Community Detection

Detects group clustering or partition options.



Estimates the likelihood of nodes forming a future relationship.

## Link Prediction



## Similarity

Evaluates how alike nodes are.

The book cover features a purple header with the O'Reilly logo and the title 'Graph Algorithms'. Below the title, it says 'Practical Examples in Apache Spark & Neo4j'. The cover art shows a spider in the center of a web, with a grid pattern behind it. The authors' names, 'Marc Needham & Amy E. Hodler', are at the bottom right. The background of the entire block is blue.

[neo4j.com/graph-algorithms-book/](http://neo4j.com/graph-algorithms-book/)

# Graph and ML algorithms in Neo4j



## Pathfinding & Search

- Parallel Breadth First Search & Depth First Search
- Shortest Path
- Single-Source Shortest Path
- All Pairs Shortest Path
- Minimum Spanning Tree
- A\* Shortest Path
- Yen's K Shortest Path
- K-Spanning Tree (MST)
- Random Walk



## Centrality / Importance

- Degree Centrality
- Closeness Centrality
- CC Variations: Harmonic, Dangalchev, Wasserman & Faust
- Betweenness Centrality
- Approximate Betweenness Centrality
- PageRank
- Personalized PageRank
- ArticleRank
- Eigenvector Centrality



## Community Detection

- Triangle Count
- Clustering Coefficients
- Connected Components (Union Find)
- Strongly Connected Components
- Label Propagation
- Louvain Modularity – 1 Step & Multi-Step
- Balanced Triad (identification)



## Similarity

- Euclidean Distance
- Cosine Similarity
- Jaccard Similarity
- Overlap Similarity
- Pearson Similarity
- Approximate Nearest Neighbors



## Link Prediction

- Adamic Adar
- Common Neighbors
- Preferential Attachment
- Resource Allocations
- Same Community
- Total Neighbors

# How to use graph algorithms

1. Pass in specification (Node Label, Relationship, or Cypher Queries) and configuration.

2. Return results:

- Stream procedure returns a lot of results on large graphs - usually one record per node in your graph

```
CALL algo.<name>.stream('Label', 'TYPE',  
{conf})  
  
YIELD nodeId, result
```

- Non-stream procedure can write results to graph and returns statistics.

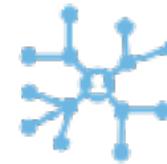
```
CALL algo.<name>('Label', 'TYPE', {conf})
```



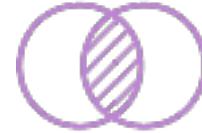
Pathfinding  
& Search



Community  
Detection



Centrality /  
Importance



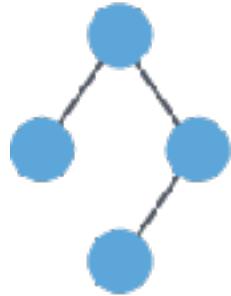
Similarity



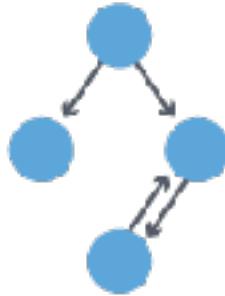
Link Prediction

# Types of graphs considerations

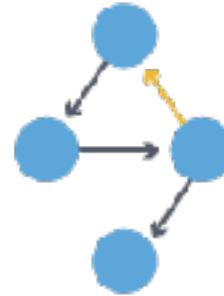
Undirected



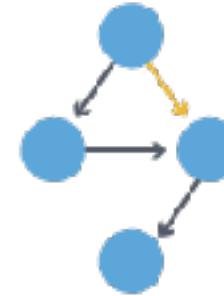
Directed



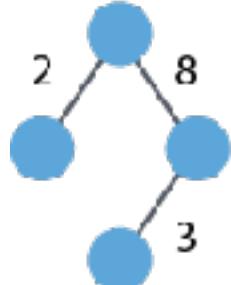
Cyclic



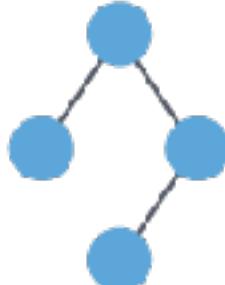
Acyclic



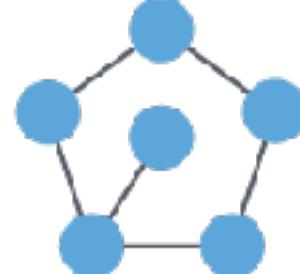
Weighted



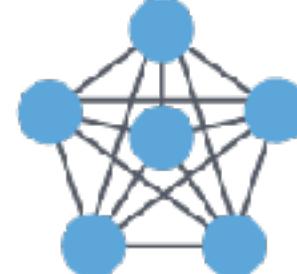
Unweighted



Sparse



Dense

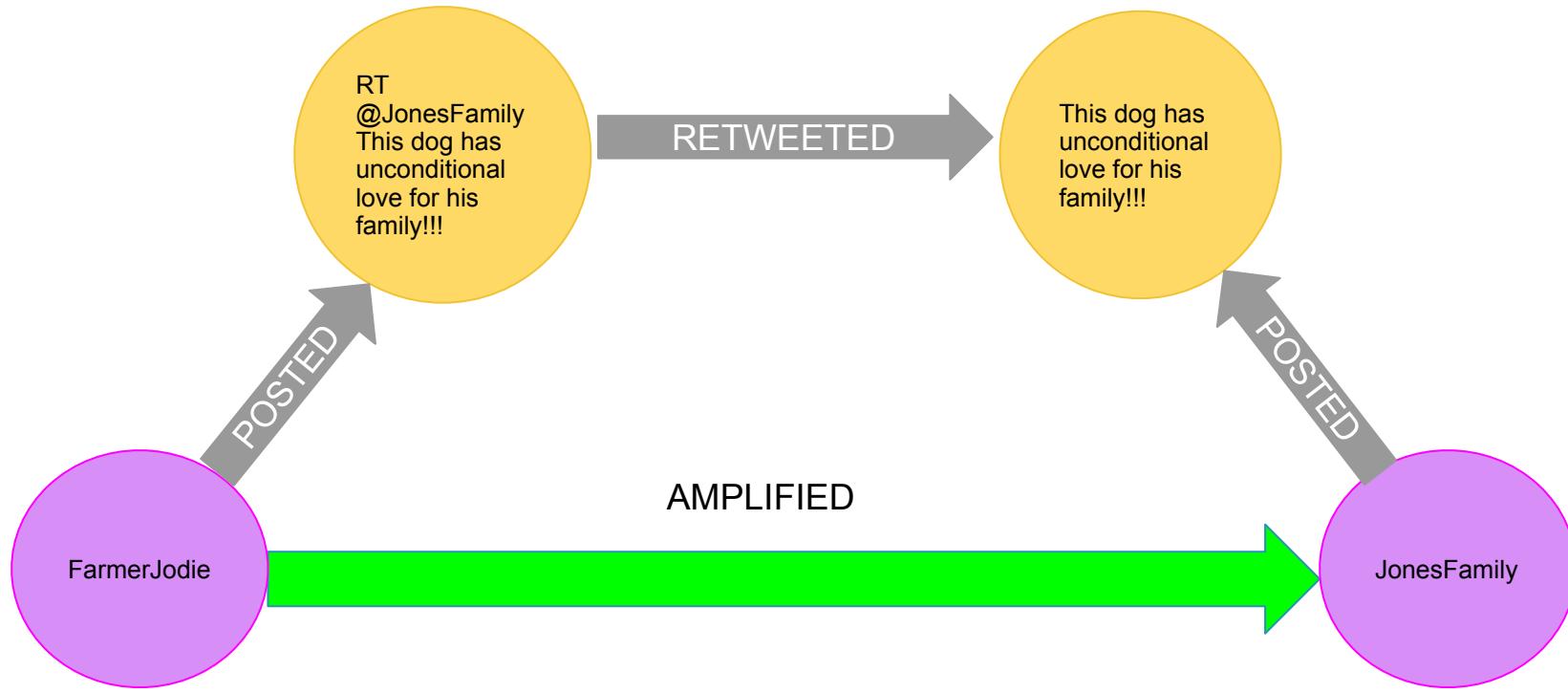


# Cypher projection example: Twitter Trolls - 1

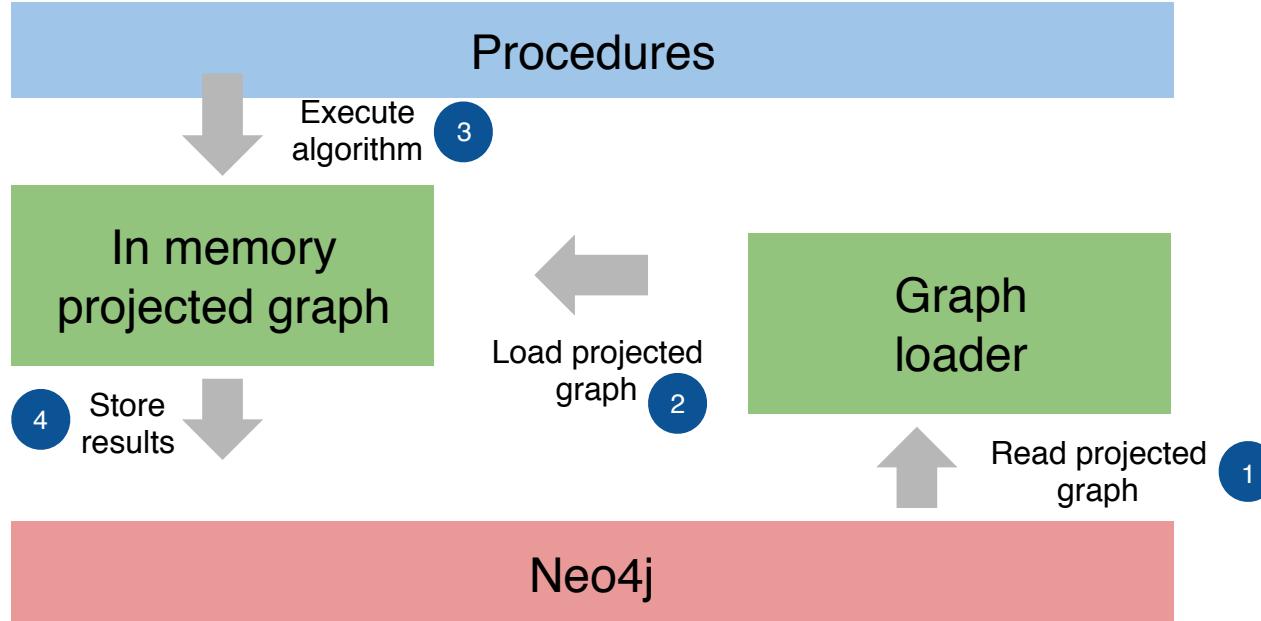


# Inferred relationships

```
MATCH (r1:Troll)-[:POSTED]->(t1:Tweet)<-[ :RETWEETED ]-(t2:Tweet)<-[ :POSTED ]-(r2:Troll)
```



# How do graph algorithms work?

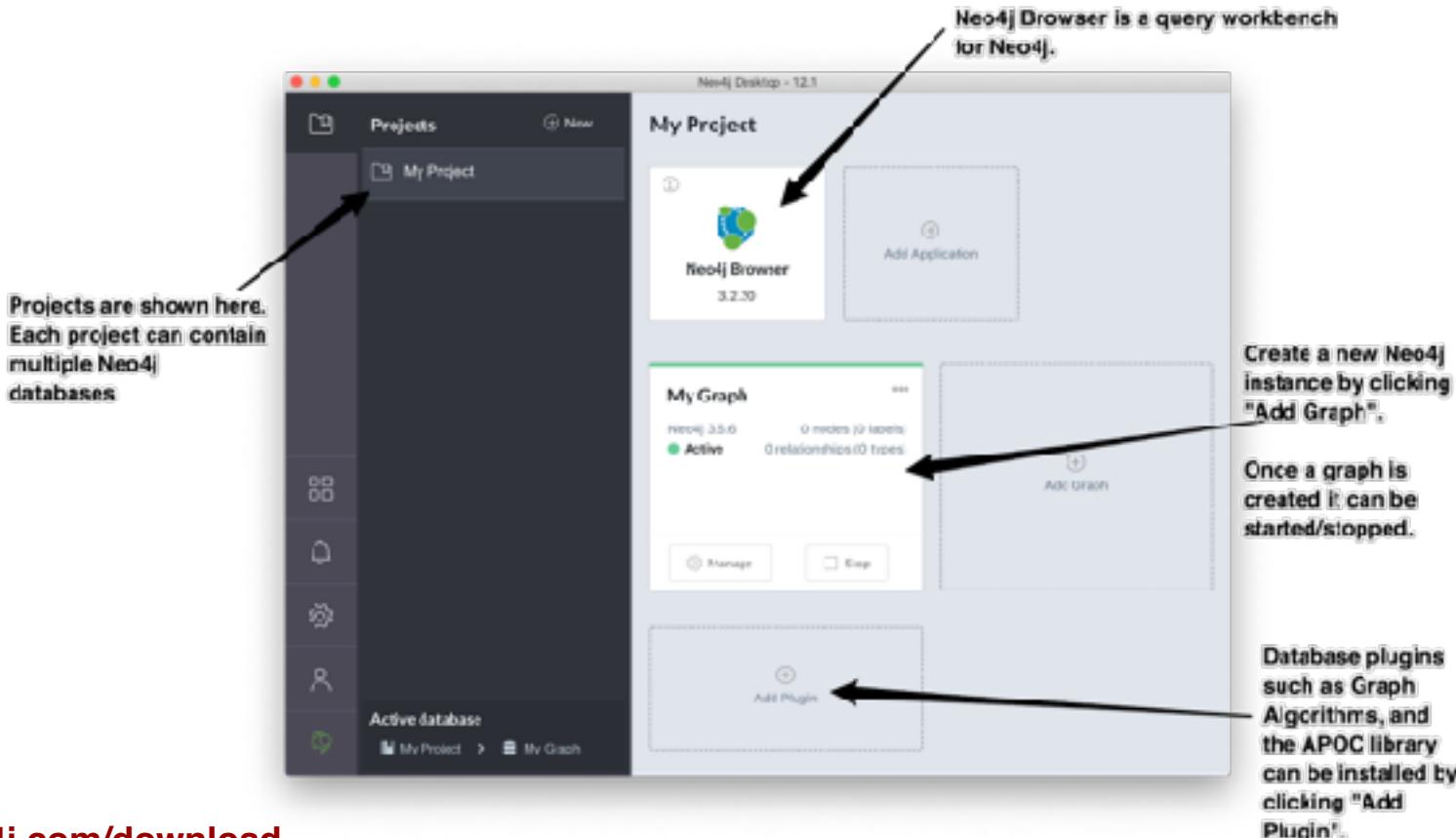


**Everything is concurrent**

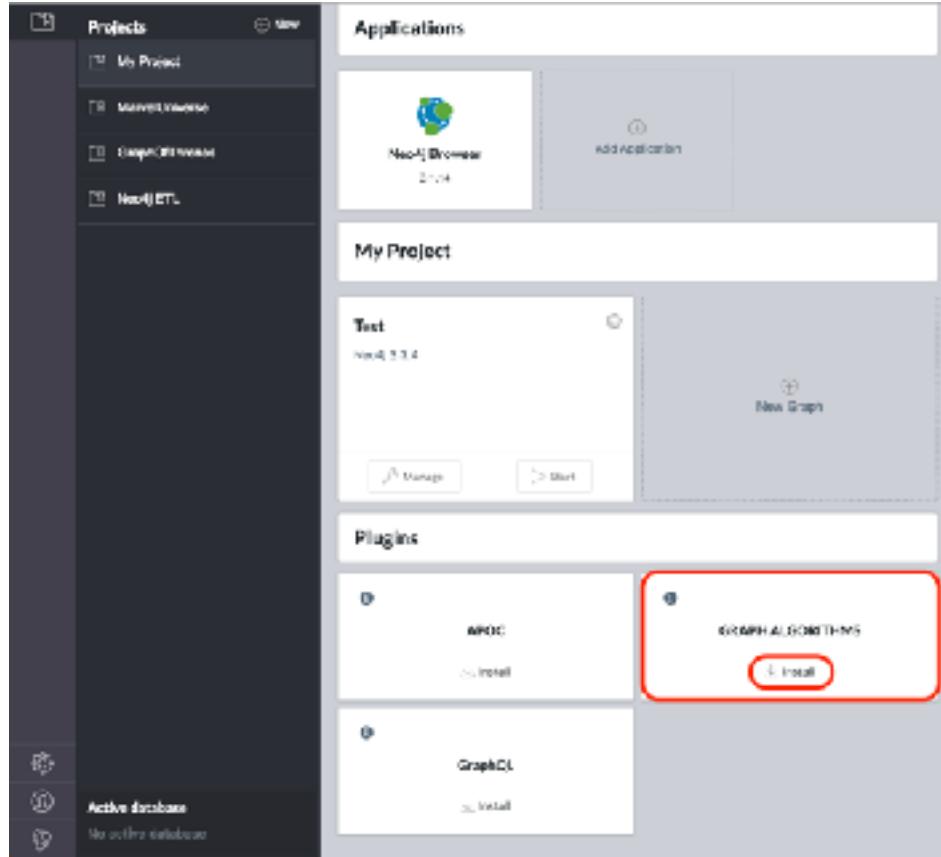


# **BREAK and Installation**

# Neo4j Desktop - “Mission Control”



# One-click install in Neo4j Desktop



# Neo4j Browser - “Query Workbench”

The database information drawer shows the node labels, relationship types, and property keys stored in the currently active database as well as other information about the database.

The screenshot displays the Neo4j Browser interface. On the left, a dark sidebar labeled "Database Information" contains sections for "Node Labels", "Relationship Types", "Property Keys", "Constraints", and "Database". The "Node Labels" section lists "User", "Review", "Business", and "Category". The "Relationship Types" section lists "RATED", "REVIEWS", and "CATEGORIES". The "Property Keys" section lists "address", "city", "date", "id", "name", "stars", "state", and "text". The "Database" section shows "Version: 3.0.0", "Status: active", "Active: true", and "Storage: memory". On the right, the main area features a "Cypher query editor" with the text "Use the Cypher query editor to execute Cypher statements." and a sample query: "MATCH (u:User)-[r:RATED]->(n:Review)-[c:CATEGORIES]->(b:Business) RETURN r". Below the editor is a graph visualization showing nodes (User, Review, Business, Category) connected by edges (RATED, REVIEWS, CATEGORIES). A result frame below the graph displays a table with three rows:

Review	avgRating
Movie	3.0
Book	4.0
Music	4.0

A result frame is created for each executed Cypher statement.

Results can be shown as a graph visualization or as a table.



# Cypher & Neo4j 101

:play gotintro.html

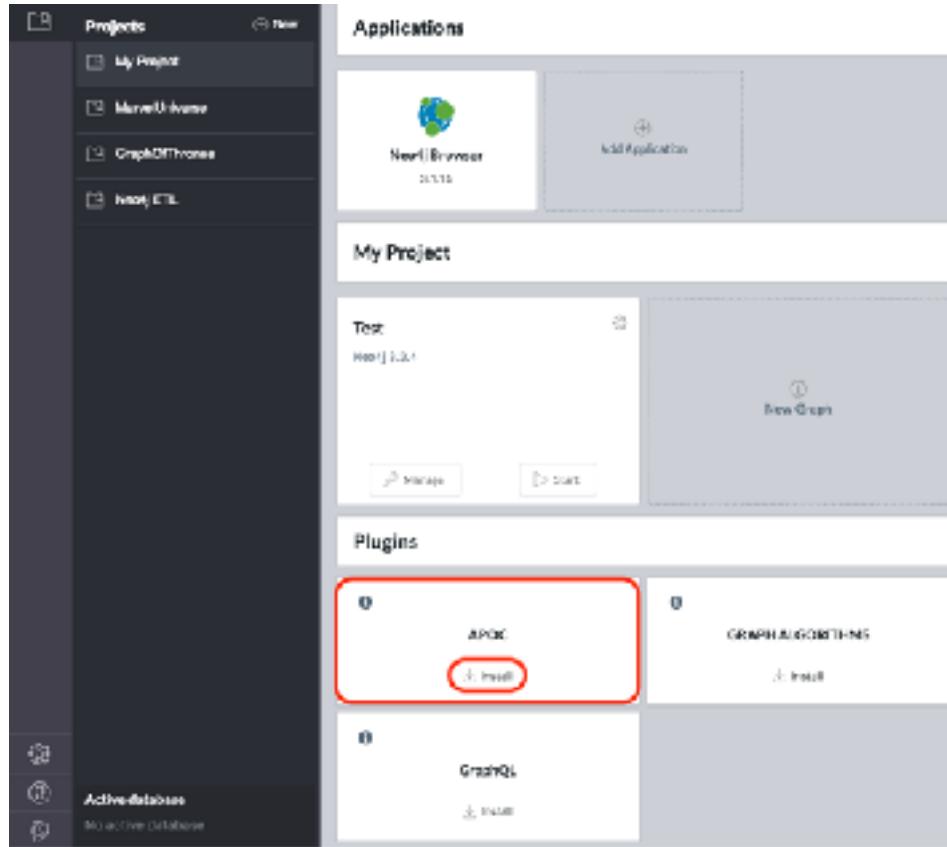
The screenshot shows the Neo4j Community Detection - DataDay interface. On the left, there's a sidebar with icons for Projects, New, Community Detection, USCON, and other database management functions. The main area is titled "Community Detection - DataDay". It displays several cards:

- Neo4j Browser** (version 3.2.22): This card is highlighted with a purple rounded rectangle. It shows a small globe icon and the text "Neo4j Browser 3.2.22".
- Add Application**: A card with a plus sign icon.
- Graph**: A card showing statistics: Neo4j 3.2.22, 479 nodes (1 active), and 2,823 relationships (0 forced). It has "Manage" and "Stop" buttons.
- APOC**: A card with a plus sign icon.
- GRAPH ALGORITHMS**: A card with a plus sign icon.

OR web browser url: **localhost:7474**



# One-click install in Neo4j Desktop



# CALL dbms.procedures()

\$ CALL dbms.procedures()		
name	signature	description
"algo.allShortestPaths.stream"	"algo.allShortestPaths.stream(propertyName :: STRING?, config :: {} :: MAP?) :: (sourceNodeid :: INTEGER?, targetNodeid :: INTEGER?, distance :: FLOAT?)"	"CALL algo.allShortestPaths.stream(weightProperty:String nodeQuery:labelName relationshipQuery:'relationshipName', defaultValue:1.0, concurrency:4) YIELD sourceNodeid, targetNodeid, distance - yields a stream of {sourceNodeid, targetNodeid, distance}"
"algo.articleRank"	"algo.articleRank(label :: STRING?, relationship :: STRING?, config :: {} :: MAP?) :: (nodes :: INTEGER?, iterations :: INTEGER?, loadMills :: INTEGER?, computeMills :: INTEGER?, writeMills :: INTEGER?, dampingFactor :: FLOAT?, write :: BOOLEAN?, writeProperty :: STRING?)"	"CALL algo.articleRank(label:String, relationship:String, [iterations:5, dampingFactor:0.85, weightProperty: null, write:true, writeProperty:'articlerank', concurrency:4]) YIELD nodes, iter, loadMills, computeMills, writeMills, dampingFactor, write, writeProperty - calculates page rank potentially while back"
"algo.articleRank.stream"	"algo.articleRank.stream(label :: STRING?, relationship :: STRING?, config :: {} :: MAP?) :: (nodeid :: INTEGER?, score :: FLOAT?)"	"CALL algo.articleRank.stream(label:String, relationship:String, [iterations:20, dampingFactor:0.85, weightProperty: null, concurrency:4]) YIELD node, score - calculates page rank and streams results"

## Table of Contents

- 1. Introduction
  - 1.1. Algorithms
  - 1.2. Installation
  - 1.3. Usage
- 2. Projected Graph Model
  - 2.1. Label and relationship-type projection
  - 2.2. Cypher projection
  - 2.3. Named graphs
- 3. The Yelp example
  - 3.1. The Yelp Open Dataset
  - 3.2. Data
  - 3.3. Graph model
  - 3.4. Import
  - 3.5. Networks
- 4. Procedures
- 5. Centrality algorithms
  - 5.1. The PageRank algorithm
  - 5.2. The ArticleRank algorithm
  - 5.3. The Betweenness Centrality algorithm
  - 5.4. The Closeness Centrality algorithm
  - 5.5. The Harmonic Centrality algorithm
  - 5.6. The Eigenvector Centrality algorithm
  - 5.7. The Degree Centrality algorithm
- 6. Community detection algorithms
  - 6.1. The Louvain algorithm
  - 6.2. The Label Propagation algorithm
  - 6.3. The Connected Components algorithm
  - 6.4. The Strongly Connected

# The Neo4j Graph Algorithms User Guide v3.5

Copyright © 2019 Neo4j, Inc.

License: [Creative Commons 4.0](#)

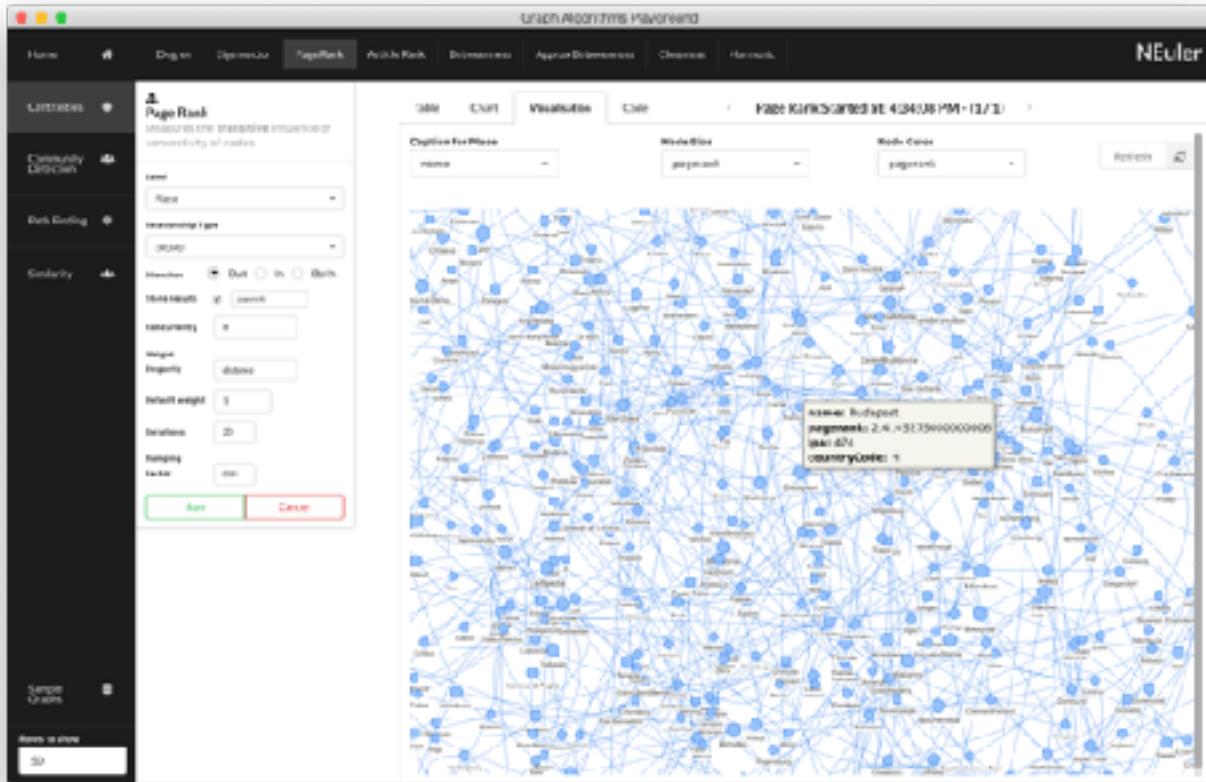
*This is the user guide for Neo4j Graph Algorithms version 3.5, authored by the Neo4j Team.*

The guide covers the following areas:

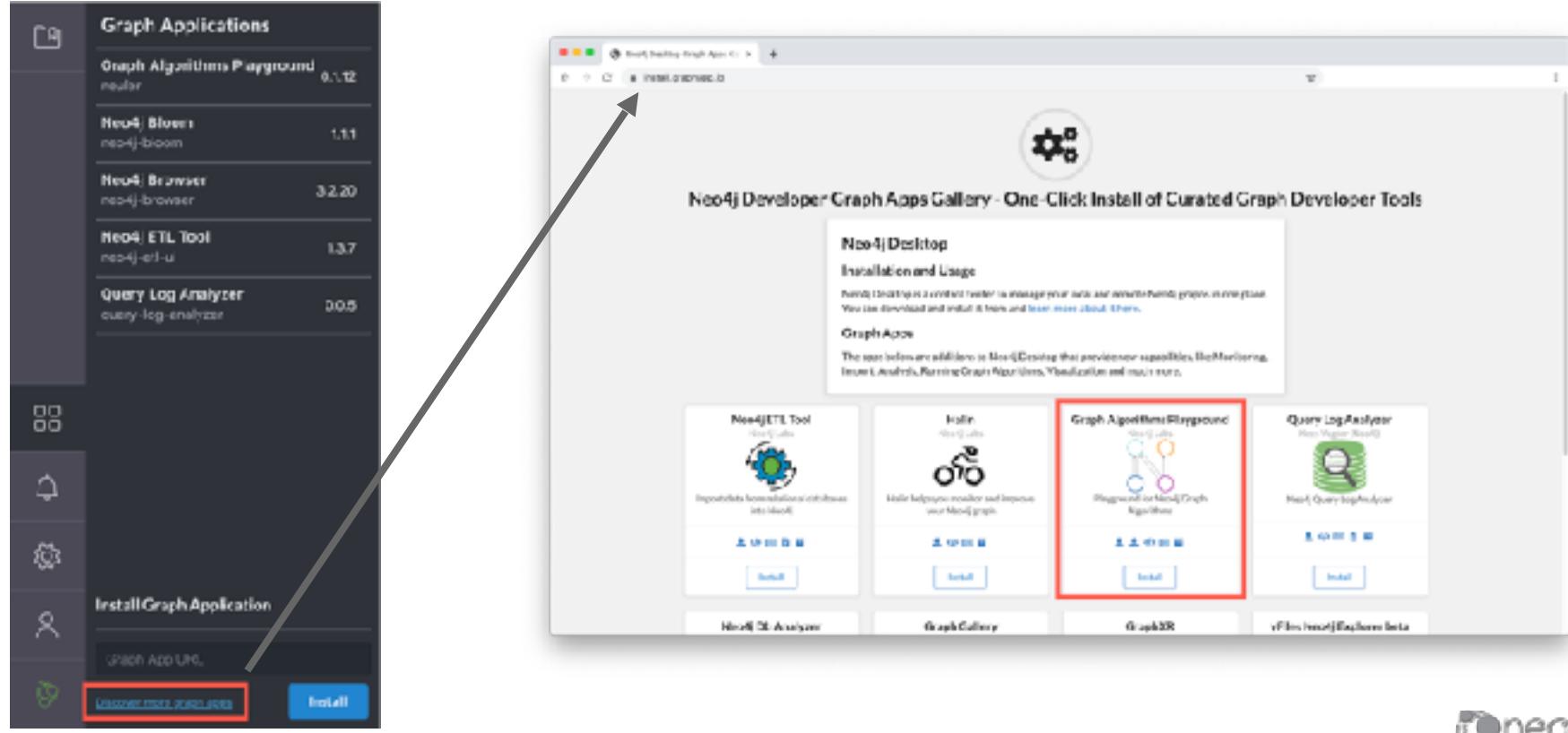
- [Chapter 1, Introduction](#) — An introduction to Neo4j Graph Algorithms.
- [Chapter 2, Projected Graph Model](#) — A detailed guide to the projected graph model.
- [Chapter 3, The Yelp example](#) — An illustration of how to use graph algorithms on a social network of friends.
- [Chapter 4, Procedures](#) — A list of Neo4j Graph Algorithm procedures.
- [Chapter 5, Centrality algorithms](#) — A detailed guide to each of the centrality algorithms, including use cases and examples.
- [Chapter 6, Community detection algorithms](#) — A detailed guide to each of the community detection algorithms, including use cases and examples.
- [Chapter 7, Path finding algorithms](#) — A detailed guide to each of the path finding algorithms, including use cases and examples.
- [Chapter 8, Similarity algorithms](#) — A detailed guide to each of the similarity algorithms, including use cases and examples.
- [Chapter 9, Link Prediction algorithms](#) — A detailed guide to each of the link prediction algorithms, including use cases and examples.
- [Chapter 10, Preprocessing functions and procedures](#) — A detailed guide to each of the preprocessing functions and procedures.

[neo4j.com/docs/graph-algorithms/  
current](https://neo4j.com/docs/graph-algorithms/current)

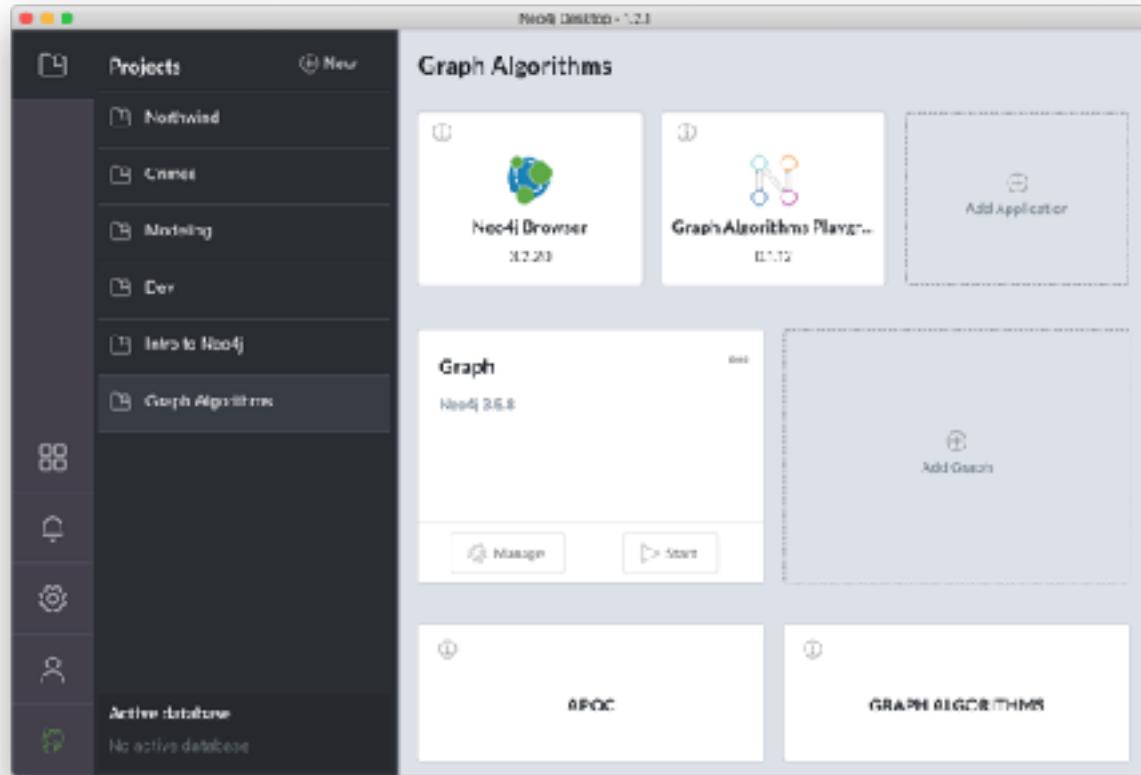
# Enter the NEuler



# Install NEuler (Graph Algorithms Playground): [install.graphapp.io](https://install.graphapp.io)



# Graph Algorithms Playground (NEuler) installed



# Loading datasets from NEuler

The screenshot shows the "Graph Algorithms Playground" interface with a dark theme. At the top, there's a navigation bar with icons for Home, Sample Graphs, and NEuler. Below the navigation bar, there's a section titled "Sample Graphs" with a sub-section header "Below are some sample graphs that are useful for learning how to use the graph algorithms library. Note that clicking on Load will import data into your graph, so don't do this on a production database." Three datasets are listed:

- Game of Thrones** by Andrew Bevington: A dataset containing interactions between the characters across the first 7 seasons of the popular TV show. A red box highlights the "Load" button.
- European Roads** by Lucie Wesslau-Nebauer: A dataset containing European Roads. A red box highlights the "Load" button.
- Twitter** by Mark Newman: A dataset containing Twitter followees of the graph community. A red box highlights the "Load" button.

A red box also highlights the bottom-left icon in the sidebar, which appears to be a stack of coins or a database symbol.

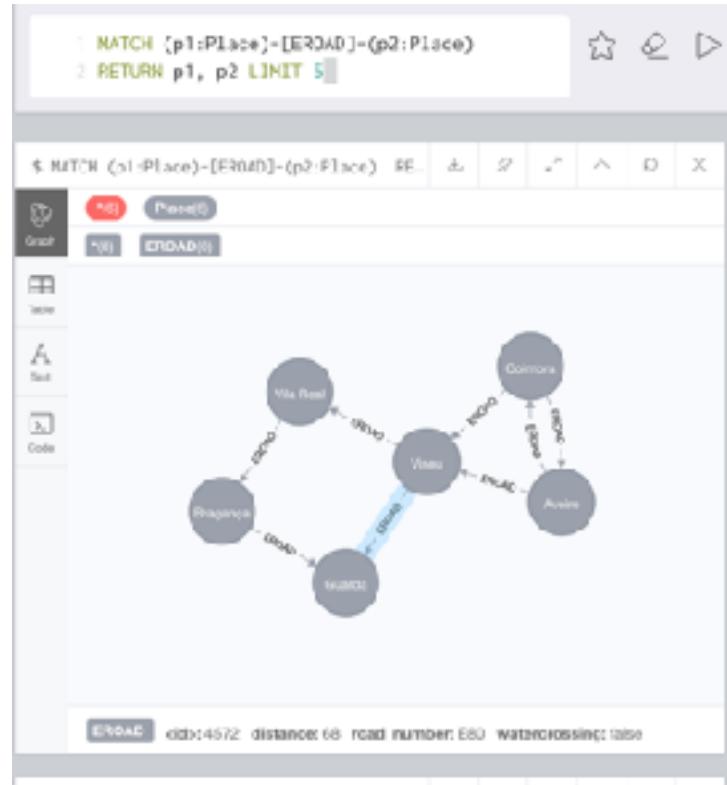
# Game of Thrones (GOT) data model



The **INTERACTS\_SEASONx** relationship has a property, **weight** to indicate the number of interactions.

The **Character** node has two properties, **name** and **id** where **id** is the capitalized value of **name**.

# European Roads data model



The **EROAD** relationship has three properties, **distance**, **road\_number**, and **watercrossing**.

The **Place** node has multiple properties, **name** and **countryCode**.

## Exercise 1: Set up your Development Environment

Before coming to this training, you should have:

1. Installed Neo4j Desktop.
2. Downloaded the Yelp dataset. (Note: This database is 1G and will take a while to download) [https://s3.amazonaws.com/neo4j-sandbox-usecase-datastores/v3\\_5/yelp.db.zip](https://s3.amazonaws.com/neo4j-sandbox-usecase-datastores/v3_5/yelp.db.zip) and unzip.
3. Created a project and a local graph, providing a password you will remember. (Do not start the database!)
4. Installed APOC, Graph Algorithms, and Graph Algorithms Playground in the project.
5. Copied the Yelp database to your local graph:
  - a. Click the Manage button for the local graph you just created.
  - b. Click Open Folder.
  - c. Navigate to data/databases.
  - d. Copy the yelp.db folder that you unzipped to databases.
  - e. Rename the yelp.db folder to graph.db.
6. Started the database.

:play intro-graph-algos-exercises (Set up your Development Environment)

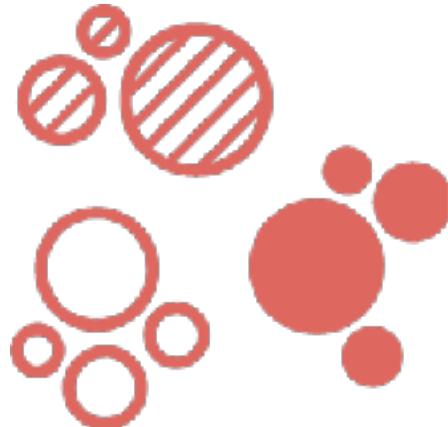
Estimated time to complete this exercise:

10 minutes



# Community Detection algorithms

# Community Detection Algorithms



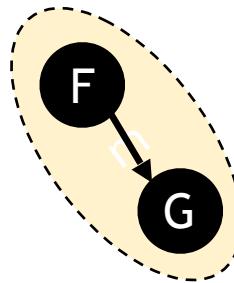
Evaluates how a group is clustered or partitioned

Different approaches to define a community

# Weakly Connected Components



# Weakly Connected Components (UNION FIND)



- All nodes can reach each other when disregarding direction.
- Find disconnected subgraphs or nodes in common and preprocess data.
- Optionally write unionFind value to each node for the analysis.

# Weakly Connected Components (UNION FIND)

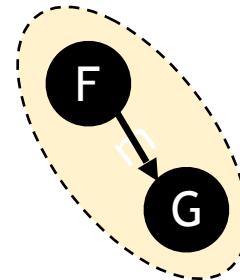
## Uses

Early graph analysis to understand graph structure

Understanding how graph structure has changed with updates (scales well)

Seeing how connected a graph is

Looking for new nodes in common between groups



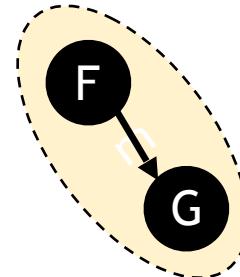
# Weakly Connected Components (UNION FIND)

## Examples

### MDM

- tracking clusters of database records for deduplication process

### Citation networks



# Example: Connected Components - GOT

The screenshot shows the Neo4j Browser interface. On the left, there's a sidebar with various filters and a main panel displaying a table of nodes and their relationships. A red box highlights the 'unionFind' column in the table. On the right, a results window displays a list of character names with their unionFind values. A red box highlights the 'unionFind' column in the results table.

Type	Name	unionFind
Character	Arya	0
Character	Bran	0
Character	Daenerys	0
Character	Drogo	0
Character	Gendry	0
Character	Hot Pie	0
Character	Jaime	0
Character	Joanna	0
Character	Jon	0
Character	Jessie	0
Character	Lancel	0
Character	Randy	0
Character	Renee	0
Character	Ronan	0
Character	Sansa	0
Character	Samwell	0
Character	Tyrion	0
Character	Tommen	0
Character	Walder	0
Character	Yara	0
Character	Zane	0

unionFind	Count
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1

Any node that has a unique value for unionFind will not be connected to any other node in the graph for the season 1 relationship.

# Connected Components - GOT data

\$ MATCH (c1:Character)-[:INTERACTS\_SEASON?]->(c2:Character) WHERE c1.unionFind = c2.unionFind ..

181 Character(s)

Graph Table SQL Code

\*(1525) INTERACTS\_SEASON1(193) INTERACTS\_SEASON2(188) INTERACTS\_SEASON3(181) INTERACTS\_SEASON4(228)



Displaying 61 nodes, 1523 relationships.

neo4j

# Connected Components in Cypher

Graph algorithms [unresolved]

Connected Components

ConnectedComponents

ConnectedComponents

Traverser

Index Count

NFuser

Connected Components

This algorithm finds sets of interconnected nodes in an otherwise disconnected graph. It starts at a single node and explores all nodes it can reach via edges.

Table

Instructions

Code

ConnectedComponents Started at: 10.128.191.171:0

This algorithm can be reproduced in the Neo4j Browser, by entering the following parameters:

```
parameters: label: > "Character";  
parameters: relationshipType: > "DIRECTED_RELATIONSHIP";  
parameters: limit: > 100;  
parameters: ordering: > 1;  
parameters: startNode: > "Character";  
parameters: endNode: > "Character";  
parameters: traversalPath: > "";  
parameters: traversalOrder: > "outbound";  
parameters: maxLevel: > 0;
```

Final query using the following cypher:

```
CALL algo.connectedComponents([label], {relationshipType: "DIRECTED_RELATIONSHIP", parameterCount: 96}, {parameterCount: 96})
```

Final Cypher:

```
algo.connectedComponents([label], {relationshipType: "DIRECTED_RELATIONSHIP", parameterCount: 96}, {parameterCount: 96})
```

Run

Cancel

PHOENIX-DESKTOP-1:50000 [neo4j] 2018-01-25 10:10:10

algo.connectedComponents	maxLevel	nodes	connectedComponentsCount	setCount	id	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12	p13	p14	p15	p16	p17	p18	p19	p20	p21	p22	p23	p24	p25	p26	p27	p28	p29	p30	p31	p32	p33	p34	p35	p36	p37	p38	p39	p40	p41	p42	p43	p44	p45	p46	p47	p48	p49	p50	p51	p52	p53	p54	p55	p56	p57	p58	p59	p60	p61	p62	p63	p64	p65	p66	p67	p68	p69	p70	p71	p72	p73	p74	p75	p76	p77	p78	p79	p80	p81	p82	p83	p84	p85	p86	p87	p88	p89	p90	p91	p92	p93	p94	p95	p96	p97	p98	p99	p100	p101	p102	p103	p104	p105	p106	p107	p108	p109	p110	p111	p112	p113	p114	p115	p116	p117	p118	p119	p120	p121	p122	p123	p124	p125	p126	p127	p128	p129	p130	p131	p132	p133	p134	p135	p136	p137	p138	p139	p140	p141	p142	p143	p144	p145	p146	p147	p148	p149	p150	p151	p152	p153	p154	p155	p156	p157	p158	p159	p160	p161	p162	p163	p164	p165	p166	p167	p168	p169	p170	p171	p172	p173	p174	p175	p176	p177	p178	p179	p180	p181	p182	p183	p184	p185	p186	p187	p188	p189	p190	p191	p192	p193	p194	p195	p196	p197	p198	p199	p200	p201	p202	p203	p204	p205	p206	p207	p208	p209	p210	p211	p212	p213	p214	p215	p216	p217	p218	p219	p220	p221	p222	p223	p224	p225	p226	p227	p228	p229	p230	p231	p232	p233	p234	p235	p236	p237	p238	p239	p240	p241	p242	p243	p244	p245	p246	p247	p248	p249	p250	p251	p252	p253	p254	p255	p256	p257	p258	p259	p260	p261	p262	p263	p264	p265	p266	p267	p268	p269	p270	p271	p272	p273	p274	p275	p276	p277	p278	p279	p280	p281	p282	p283	p284	p285	p286	p287	p288	p289	p290	p291	p292	p293	p294	p295	p296	p297	p298	p299	p300	p301	p302	p303	p304	p305	p306	p307	p308	p309	p310	p311	p312	p313	p314	p315	p316	p317	p318	p319	p320	p321	p322	p323	p324	p325	p326	p327	p328	p329	p330	p331	p332	p333	p334	p335	p336	p337	p338	p339	p340	p341	p342	p343	p344	p345	p346	p347	p348	p349	p350	p351	p352	p353	p354	p355	p356	p357	p358	p359	p360	p361	p362	p363	p364	p365	p366	p367	p368	p369	p370	p371	p372	p373	p374	p375	p376	p377	p378	p379	p380	p381	p382	p383	p384	p385	p386	p387	p388	p389	p390	p391	p392	p393	p394	p395	p396	p397	p398	p399	p400	p401	p402	p403	p404	p405	p406	p407	p408	p409	p410	p411	p412	p413	p414	p415	p416	p417	p418	p419	p420	p421	p422	p423	p424	p425	p426	p427	p428	p429	p430	p431	p432	p433	p434	p435	p436	p437	p438	p439	p440	p441	p442	p443	p444	p445	p446	p447	p448	p449	p450	p451	p452	p453	p454	p455	p456	p457	p458	p459	p460	p461	p462	p463	p464	p465	p466	p467	p468	p469	p470	p471	p472	p473	p474	p475	p476	p477	p478	p479	p480	p481	p482	p483	p484	p485	p486	p487	p488	p489	p490	p491	p492	p493	p494	p495	p496	p497	p498	p499	p500	p501	p502	p503	p504	p505	p506	p507	p508	p509	p510	p511	p512	p513	p514	p515	p516	p517	p518	p519	p520	p521	p522	p523	p524	p525	p526	p527	p528	p529	p530	p531	p532	p533	p534	p535	p536	p537	p538	p539	p540	p541	p542	p543	p544	p545	p546	p547	p548	p549	p550	p551	p552	p553	p554	p555	p556	p557	p558	p559	p560	p561	p562	p563	p564	p565	p566	p567	p568	p569	p570	p571	p572	p573	p574	p575	p576	p577	p578	p579	p580	p581	p582	p583	p584	p585	p586	p587	p588	p589	p590	p591	p592	p593	p594	p595	p596	p597	p598	p599	p600	p601	p602	p603	p604	p605	p606	p607	p608	p609	p610	p611	p612	p613	p614	p615	p616	p617	p618	p619	p620	p621	p622	p623	p624	p625	p626	p627	p628	p629	p630	p631	p632	p633	p634	p635	p636	p637	p638	p639	p640	p641	p642	p643	p644	p645	p646	p647	p648	p649	p650	p651	p652	p653	p654	p655	p656	p657	p658	p659	p660	p661	p662	p663	p664	p665	p666	p667	p668	p669	p670	p671	p672	p673	p674	p675	p676	p677	p678	p679	p680	p681	p682	p683	p684	p685	p686	p687	p688	p689	p690	p691	p692	p693	p694	p695	p696	p697	p698	p699	p700	p701	p702	p703	p704	p705	p706	p707	p708	p709	p710	p711	p712	p713	p714	p715	p716	p717	p718	p719	p720	p721	p722	p723	p724	p725	p726	p727	p728	p729	p730	p731	p732	p733	p734	p735	p736	p737	p738	p739	p740	p741	p742	p743	p744	p745	p746	p747	p748	p749	p750	p751	p752	p753	p754	p755	p756	p757	p758	p759	p760	p761	p762	p763	p764	p765	p766	p767	p768	p769	p770	p771	p772	p773	p774	p775	p776	p777	p778	p779	p780	p781	p782	p783	p784	p785	p786	p787	p788	p789	p790	p791	p792	p793	p794	p795	p796	p797	p798	p799	p800	p801	p802	p803	p804	p805	p806	p807	p808	p809	p810	p811	p812	p813	p814	p815	p816	p817	p818	p819	p820	p821	p822	p823	p824	p825	p826	p827	p828	p829	p830	p831	p832	p833	p834	p835	p836	p837	p838	p839	p840	p841	p842	p843	p844	p845	p846	p847	p848	p849	p850	p851	p852	p853	p854	p855	p856	p857	p858	p859	p860	p861	p862	p863	p864	p865	p866	p867	p868	p869	p870	p871	p872	p873	p874	p875	p876	p877	p878	p879	p880	p881	p882	p883	p884	p885	p886	p887	p888	p889	p890	p891	p892	p893	p894	p895	p896	p897	p898	p899	p900	p901	p902	p903	p904	p905	p906	p907	p908	p909	p910	p911	p912	p913	p914	p915	p916	p917	p918	p919	p920	p921	p922	p923	p924	p925	p926	p927	p928	p929	p930	p931	p932	p933	p934	p935	p936	p937	p938	p939	p940	p941	p942	p943	p944	p945	p946	p947	p948	p949	p950	p951	p952	p953	p954	p955	p956	p957	p958	p959	p960	p961	p962	p963	p964	p965	p966	p967	p968	p969	p970	p971	p972	p973	p974	p975	p976	p977	p978	p979	p980	p981	p982	p983	p984	p985	p986	p987	p988	p989	p990	p991	p992	p993	p994	p995	p996	p997	p998	p999	p1000	p1001	p1002	p1003	p1004	p1005	p1006	p1007	p1008	p1009	p10010	p10011	p10012	p10013	p10014	p10015	p10016	p10017	p10018	p10019	p10020	p10021	p10022	p10023	p10024	p10025	p10026	p10027	p10028	p10029	p10030	p10031	p10032	p10033	p10034	p10035	p10036	p10037	p10038	p10039	p10040	p10041	p10042	p10043	p10044	p10045	p10046	p10047	p10048	p10049	p10050	p10051	p10052	p10053	p10054	p10055	p10056	p10057	p10058	p10059	p10060	p10061	p10062	p10063	p10064	p10065	p10066	p10067	p10068	p10069	p10070	p10071	p10072	p10073	p10074	p10075	p10076	p10077	p10078	p10079	p10080	p10081	p10082	p10083	p10084	p10085	p10086	p10087	p10088	p10089	p10090	p10091	p10092	p10093	p10094	p10095	p10096	p10097	p10098	p10099	p100100	p100101	p100102	p100103	p100104	p100105	p100106	p100107	p100108	p100109	p100110	p100111	p100112	p100113	p100114	p100115	p100116	p100117	p100118	p100119	p100120	p100121	p100122	p100123	p100124	p100125	p100126	p100127	p100128	p100129	p100130	p100131	p100132	p100133	p100134	p100135	p100136	p100137	p100138	p100139	p100140	p100141	p100142	p100143	p100144	p100145	p100146	p100147	p100148	p100149	p100150	p100151	p100152	p100153	p100154	p100155	p100156	p100157	p100158	p100159	p100160	p100161	p100162	p100163	p100164	p100165	p100166	p100167	p100168	p100169	p100170	p100171	p100172	p100173	p100174	p100175	p100176	p100177	p100178	p100179	p100180	p100181	p100182	p100183	p100184	p100185	p100186	p100187	p100188	p100189	p100190	p100191	p100192	p100193	p100194	p100195	p100196	p100197	p100198	p100199	p100200	p100201	p100202	p100203	p100204	p100205	p100206	p100207	p100208	p100209	p100210	p100211	p100212	p100213	p100214	p100215	p100216	p100217	p100218	p100219	p100220	p100221	p100222	p100223	p100224	p100225	p100226	p100227	p100228	p100229	p100230	p100231	p100232	p100233	p100234	p100235	p100236	p100237	p100238	p100239	p100240	p100241	p100242	p100243	p100244	p100245	p100246	p100247	p100248	p100249	p100250	p100251	p100252	p100253	p100254	p100255	p100256	p100257	p100258	p100259	p100260	p100261	p100262	p100263	p100264	p100265	p100266	p100267	p100268	p100269	p100270	p100271	p100272	p100273	p100274	p100275	p100276	p100277	p100278	p100279	p100280	p100281	p100282	p100283	p100284	p100285	p100286	p100287	p100288	p100289	p100290	p100291	p100292	p100293	p100294	p100295	p100296	p100297	p100298	p100299	p100300	p100301	p100302	p100303	p100304	p100305	p100306	p100307	p100308	p100309	p100310	p100311	p100312	p100313	p100314	p100315	p100316	p100317	p100318	p100319	p100320	p100321	p100322	p100323	p100324	p100325	p100326	p100327	p100328	p100329	p100330	p100331	p100332	p100333	p100334	p100335	p100336	p100337	p100338	p100339	p100340	p100341	p100342	p100343	p100344	p100345	p100346	p100347	p100348	p100349	p100350	p100351	p100352	p100353	p100354	p100355	p100356	p100357	p100358	p100359	p100360	p100361	p100362	p100363	p100364	p1

# Weakly Connected Components - GOT data, season 1

```
1 MATCH (node:Character)
2 WHERE not(node[$config.writeProperty] is null)
3 RETURN node, node[$config.writeProperty] AS community
4 LIMIT $limit
```

\$ MATCH (node:Character) WHERE not(node[\$config.writeProperty] is null) RETURN node, node[\$co... A. D. ↻ ⌂ ⌄ X

Graph 750 Character[50] (290) INTERACTS\_SEASON1[103] INTERACTS\_SEASON7[10] INTERACTS\_SEASON2[26] INTERACTS\_SEASON4[27] ◀ ▶

Displaying 50 nodes, 290 relationships.

## Exercise 2: Weakly Connected Components

Estimated time to complete this exercise:

20 minutes

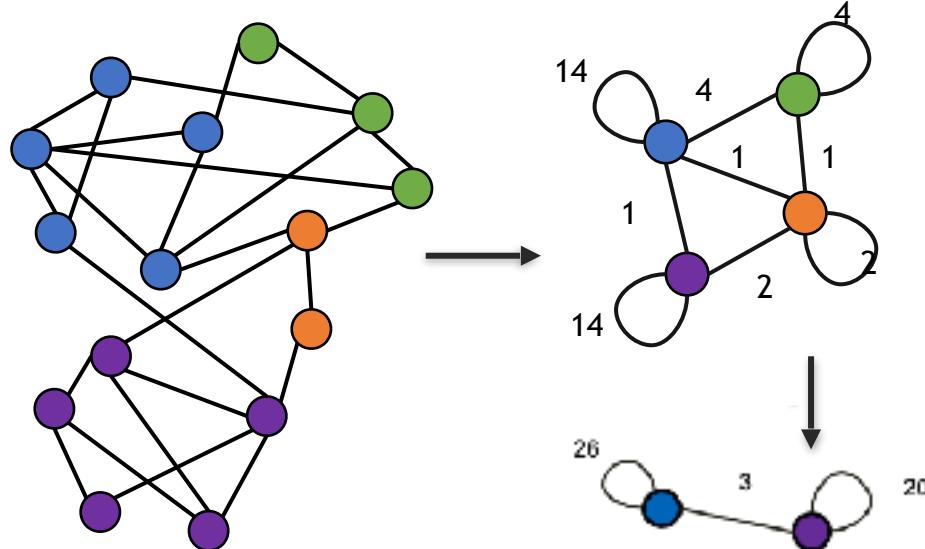
1. Start or restart NEuler so that all loaded data can be seen.
2. In NEuler:
  - a. Find all Connected Person nodes writing the unionFind\_helps property.
  - b. Find all Connected Characters for Season 3 writing the unionFind\_season3 property.
    - i. Advanced: How many nodes are connected for Season 3?
  - c. Do the same for any relationship, writing the unionFind\_any value.
3. In Neo4j Browser:

`:play intro-graph-algos-exercises (Weakly Connected Components)`

# Louvain Modularity



# Louvain Modularity



Continually maximizes the modularity by comparing relationship weights and densities to an estimate /average.

## Tip / Caution

ALL Modularity algorithms:

- Merge smaller communities into larger ones
  - Review intermediates
- Can plateau with similar modularity on several partitions - forming local maxima & stalling progress
  - Treat as a guide and test/validate results

# Louvain Modularity - Uses - 2

## Use When

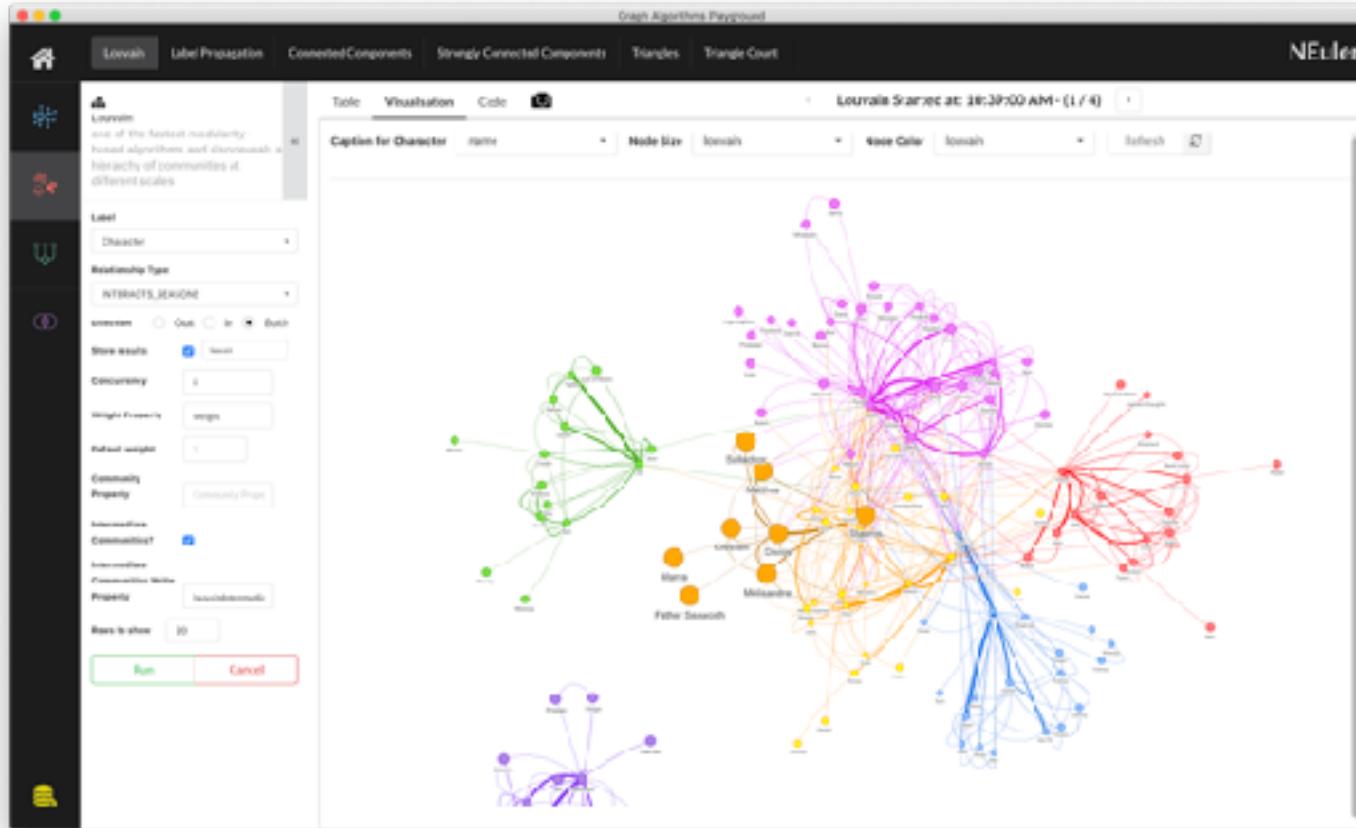
Community detection in large networks

Uncover hierarchical structures in data

Evaluate different grouping thresholds



# Louvain Modularity: GOT



# Result of Louvain Modularity: GOT

The screenshot shows the Neo4j browser interface with a query results table. The table has columns: community, main, size, and characters.

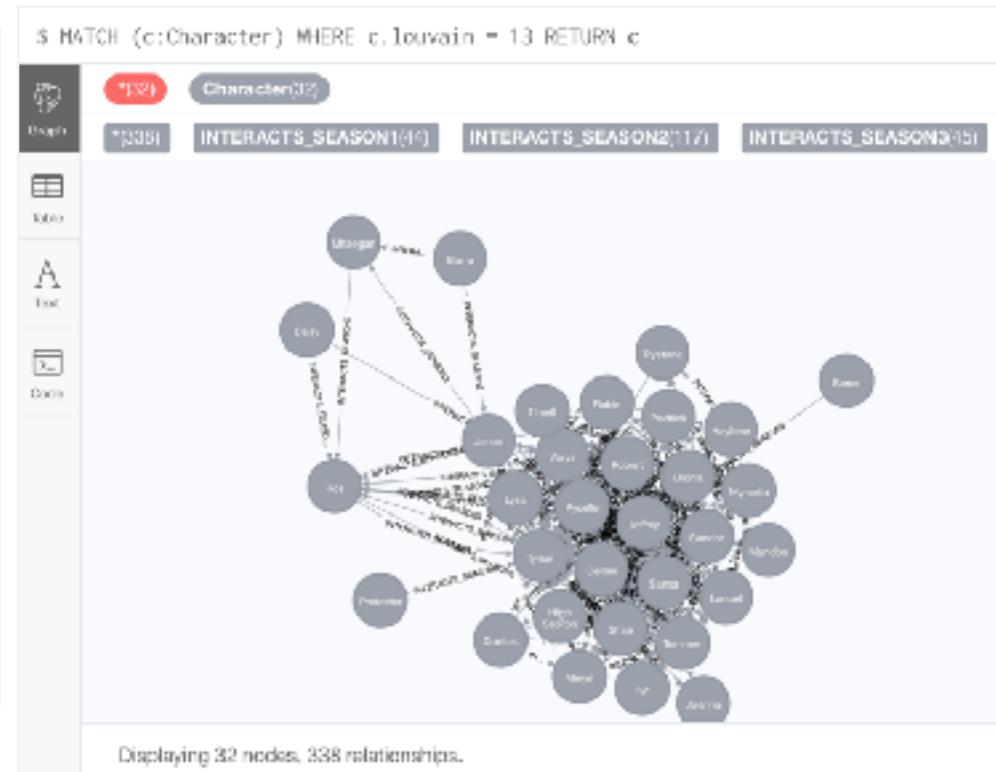
```
1 MATCH (c:Character)
2 WITH c, c.louvain AS community, size( (c)-[:INTERACTS_SEASON2]-() ) AS degree
3 ORDER BY community ASC, degree DESC
4 WITH community, head(collect(c)) name AS main, count(*) AS size,
5      collect(c.name)[0..7] AS characters, collect(c) AS all
6 ORDER BY size DESC
7 RETURN community, main, size, characters
```

community	main	size	characters
13	"Joffrey"	32	["Joffrey", "Tyrion", "Cersei", "Sansa", "Brienne", "Hobbit", "Sandor"]
2	"Robb"	24	["Robb", "Catelyn", "Petyr", "Ned", "Jaime", "Loras", "Renly"]
1	"Arya"	20	["Arya", "Twin", "Gendry", "Hot Pie", "Jaeden", "Yoren", "Amon"]
6	"Theon"	17	["Theon", "Bran", "Maester Luwin", "Rickon", "Daamer", "Oska", "Rocrik"]
8	"Jon"	14	["Jon", "Sam", "Griselda", "Jac", "Eddard", "Quarion", "Gilly"]
15	"Daenerys"	14	["Daenerys", "Drogo", "Yseril", "Kraznys", "Spine King", "Targaryen", "Inigo"]
61	"Stannis"	8	["Stannis", "Davos", "Matthias", "Malsandra", "Cressen", "Saladhor", "Father Saaworth"]
0	"Addam"	1	["Addam"]
3	"Alister"	1	["Alister"]
4	"Acaaslin"	1	["Acaaslin"]
5	"Daeor"	1	["Daeor"]
7	"Barristan"	1	["Barristan"]
9	"Beric"	1	["Beric"]

# Looking at intermediate Louvain values: GOT

\$ MATCH (c:Character) WHERE c.louvain = 13 RETURN c.name, c.louvainIntermediate

"Myrcella"	[14, 13]
"Pycelle"	[14, 13]
"Robert"	[14, 13]
"Rus"	[26, 13]
"Sansa"	[14, 13]
"Shae"	[14, 13]
"Tormund"	[14, 13]
"Tyrion"	[14, 13]
"Varys"	[14, 13]
"Bran"	[79, 13]
"Boros"	[14, 13]
"Daisy"	[26, 13]
"Dontos"	[14, 13]
"Hayne"	[14, 13]
"Janos"	[14, 13]
"Mandon"	[14, 13]
"Meryn"	[14, 13]
"Mhaqqar"	[79, 13]



A background photograph shows two individuals, a man and a woman, sitting at a desk in an office environment. They appear to be focused on work, possibly reviewing documents or using a computer. Overlaid on this image is a large, semi-transparent network graph. The graph consists of numerous dark grey circular nodes connected by thin grey lines, representing relationships or data points. The overall aesthetic is professional and technical.

Estimated time to  
complete this  
exercise:

15 minutes

## Exercise 4: Louvain Modularity

1. In NEuler
  - a. Perform the Louvain Modularity algorithm on different seasons of GOT.
  
1. In Neo4j Browser:
  - a. View the louvain and intermediate louvain values for GOT.
  - b. `:play intro-graph-algos-exercises (Louvain Modularity)`

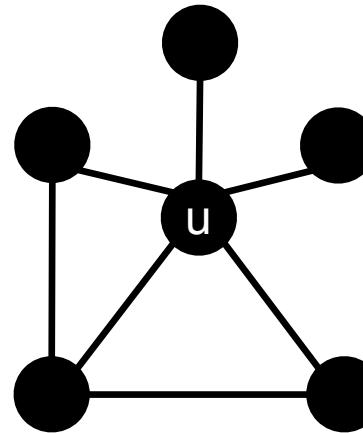
# Triangle count and clustering coefficient



# Triangles and Clustering Coefficient

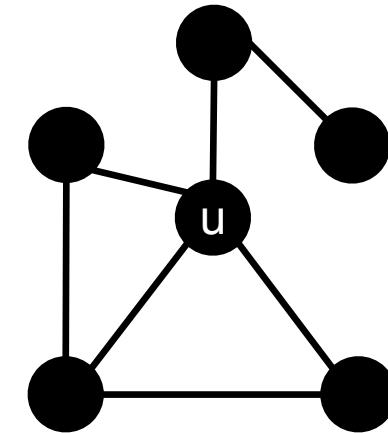
**Triangle Count** determines the number of triangles passing through a node in the graph.

**Clustering Coefficient** is the probability that neighbors of a particular node are connected to each other.



Triangles = 2

CC = 0.2



Triangles = 2

CC = 0.33

Measures can be counted/normalized globally

# Triangles and Clustering Coefficient

## Use When

Basic network analysis

- small world structures

Estimating stability

Finding structural holes

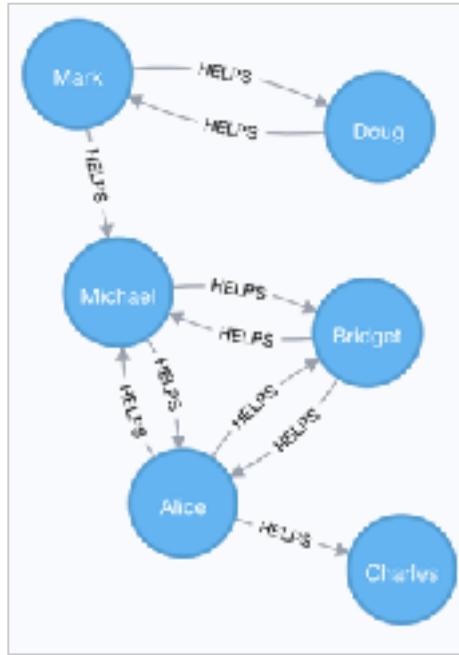
Scoring for ML



## Spam Classification

Semi-streaming web page analysis  
(local triangle and CC)

# Triangles for the Person nodes



Graph Algorithms Playground

NEuler

Labels: Label Propagation, Connected Components, Strongly Connected Components, Triangles, Triangle Count

Table: Visualization, Code,  Triangles, Triangle Count

Triangles Started at: 1:53:40 PM (1 / 1)

Node A Labels: Person

Node A Properties: name: Michael

Relationship Type: HELPS

Direction: OUT

Decomposing:  Grouping

Depth limit: 30

Run Canceled

Node A Labels	Node A Properties	Node B Labels	Node B Properties	Node C Labels	Node C Properties
Person	name: Michael	Person	name: Alice	Person	name: Bridget
Person	name: Bridget	Person	name: Mike	Person	name: Michael
Person	name: Alice	Person	name: Ingrid	Person	name: Michael

# Triangle Counts and Coefficients for the Person nodes

Graph Algorithms Playground

Labels: Person

Relationship Type: HELPS

Direction: Both

Results: 6

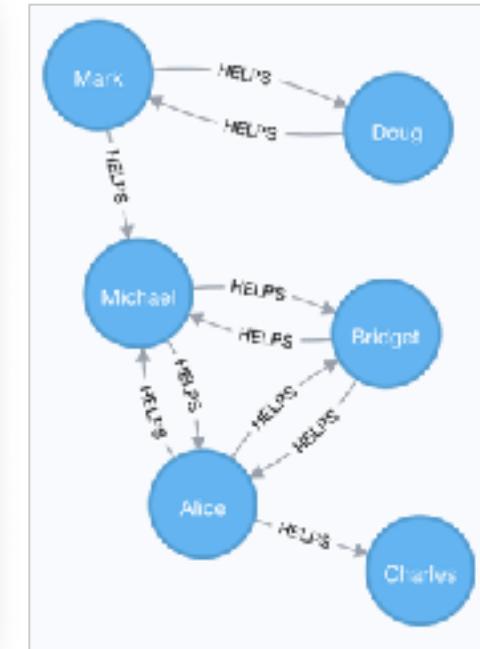
Counting nodes: running

Rows to show: 20

Run Cancel

Triangle Count Started at: 2:00:27 PM - (1 / 12)

Labels	Queries	Triangles	Coefficient
Person	name = michael	1	0.3333333333333333
Person	name = mike	1	0.3333333333333333
Person	name = aroger	1	1
Person	name = mark	0	0
Person	name = doug	0	0
Person	name = charles	0	0



# NEuler: Setting Triangle Counts and Coefficients for the Person nodes

Graph algorithms Playground

Learn ListPreparation ConnectedComponents StronglyConnectedComponents Triangles Triangle Count

Triangle Count  
Each set of three nodes, where each node has a relationship to all other nodes.

Labels Properties Triangles Coefficient

Labels	Properties	Triangles	Coefficient
Person	name: Michael, clusteringCoefficient: 0.3333333333333333	6	0.0000000000000002
Person	name: Alice, clusteringCoefficient: 0.3333333333333333	6	0.0000000000000002
Person	name: Bridget, clusteringCoefficient: 1	6	1
Person	name: Mark, clusteringCoefficient: 0	6	0
Person	name: Doug, clusteringCoefficient: 0	6	0
Person	name: Charles, clusteringCoefficient: 0	6	0

Relationship Type: HELPS  
Direction: Out In Both  
Granularity:  individual  
Degree: 6  
Clustering Coefficient:  clusteringCoeff  
Run

Triangle Count Started at: 2:25:09 PM (1 / 15)

```
{  
  "name": "Alice",  
  "Date_09-21-2010": 0,  
  "CLUSTERINGCOEFFICIENT":  
  0.3333333333333333,  
  "TRIANGLECOUNT": 1  
}
```



# Triangle counts and coefficients for GOT Characters

The screenshot shows the Neo4j Graph Algorithms Playground interface with the 'Triangle Count' tab selected. On the left, there's a sidebar with various graph analysis tools like Label Propagation, Connected Components, Strong Connected Components, Triangles, and Triangle Count. The 'Triangle Count' section is active, showing a table of results.

**Table Headers:** Labels, Properties, Triangles, Coefficient.

**Table Data:**

Labels	Properties	Triangles	Coefficient
Character	name: Ned, clusteringCoefficient_season1: 0.175485964912896, id: NED	280	0.175485964912896
Character	name: Robert, clusteringCoefficient_season1: 0.2936507936507936, id: ROBERT	185	0.2936507936507936
Character	name: Cersi, clusteringCoefficient_season1: 0.445453807817798, id: CERSI	181	0.445453807817798
Character	name: Cersei, clusteringCoefficient_season1: 0.445453807817798, id: CERSEI	181	0.445453807817798
Character	name: Arya, clusteringCoefficient_season1: 0.310446041463146, id: ARYA	175	0.310446041463146
Character	name: Joffrey, clusteringCoefficient_season1: 0.43304843304843305, id: JOFFREY	152	0.43304843304843305
Character	name: Robb, clusteringCoefficient_season1: 0.3264317816091954, id: ROBB	143	0.3264317816091954
Character	name: Petyr, clusteringCoefficient_season1: 0.4279823076923077, id: LITTLEFINGER	139	0.4279823076923077
Character	name: Sansa, clusteringCoefficient_season1: 0.4279823076923077, id: SANSA	137	0.4279823076923077
Character	name: Arya, clusteringCoefficient_season1: 0.35978358788235477, id: ARYA	136	0.35978358788235477

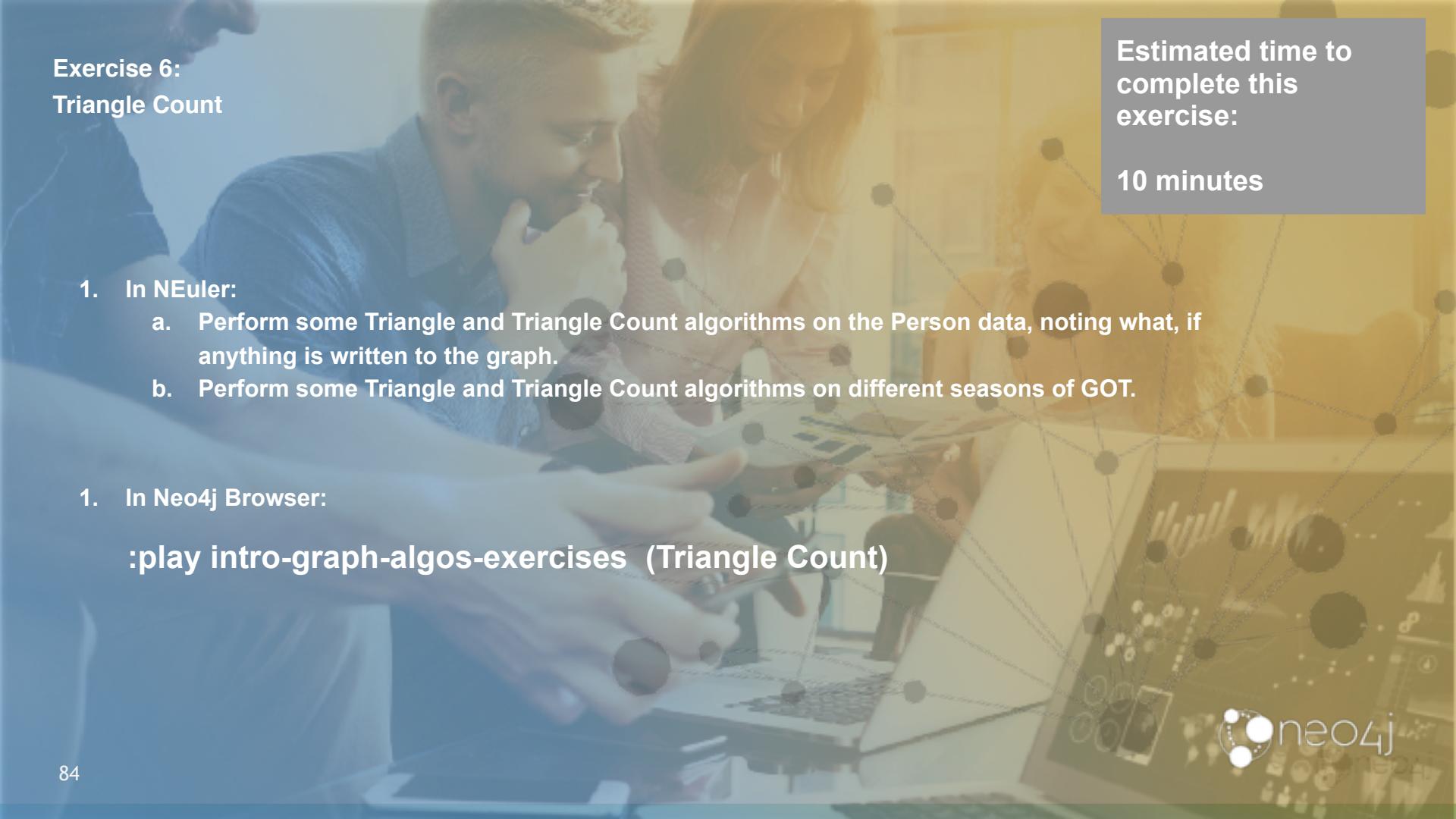
# Ordered clustering coefficients for GOT Characters

\$ MATCH (c:Character) WHERE c.trianglesCount\_season1 > 0 RETURN c.name AS name, c.triangles AS triangles, c.cc AS cc

The screenshot shows the Neo4j browser interface with a table results window. The table has three columns: 'name', 'triangles', and 'cc'. The data is as follows:

name	triangles	cc
"Aegon"	1	1.0
"Assassin"	3	1.0
"Reilon"	1	1.0
"Beric"	1	1.0
"Borcas"	1	1.0
"Bowen"	1	1.0
"Gared"	1	1.0
"Hoster"	1	1.0
"Janos"	10	1.0
"Jaremy"	3	1.0
"Jhiqu"	3	1.0
"Janos"	1	1.0
"Lanod"	10	1.0
"Lommy"	1	1.0
"Luke"	1	1.0
"Lyanna"	3	1.0

Started streaming 110 records after 3 ms and completed after 3 ms.



## Exercise 6: Triangle Count

Estimated time to  
complete this  
exercise:

10 minutes

### 1. In NEuler:

- a. Perform some Triangle and Triangle Count algorithms on the Person data, noting what, if anything is written to the graph.
- b. Perform some Triangle and Triangle Count algorithms on different seasons of GOT.

### 1. In Neo4j Browser:

`:play intro-graph-algos-exercises (Triangle Count)`

# Relationship Directions? Weights?

## Community Detection algorithms

Algorithm	Directed	Undirected	Weighted	Unweighted
Louvain		✓	✓	✓
Label Propagation	✓	✓	✓	✓
Connected Components		✓	✓	✓
Strongly Connected Components	✓			✓
Triangle Counting		✓		✓
Balanced Triads		✓	✓	

weight used as threshold

relationship must exist in both directions

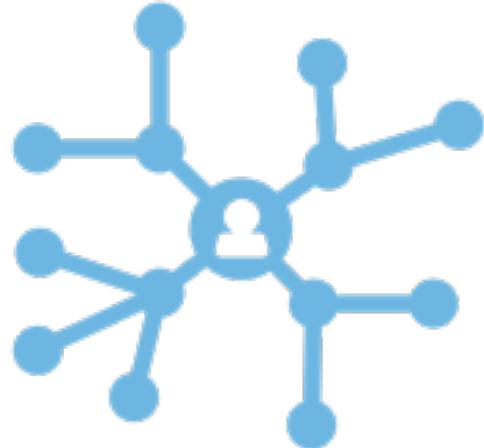
weight indicates a positive or negative relationship (friend or enemy)



**BREAK**

# Centrality algorithms

# Centrality algorithms



Determines the importance of distinct nodes in the network.

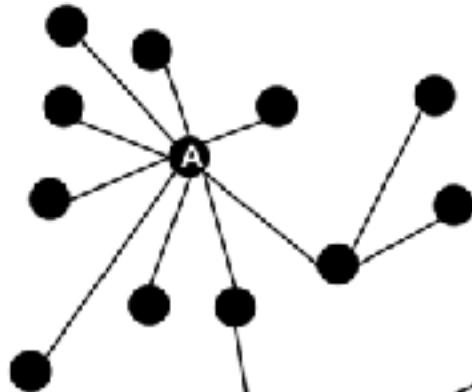
Developed for distinct uses or types of importance.

# Some Centrality algorithms

## Degree

Number of connections?

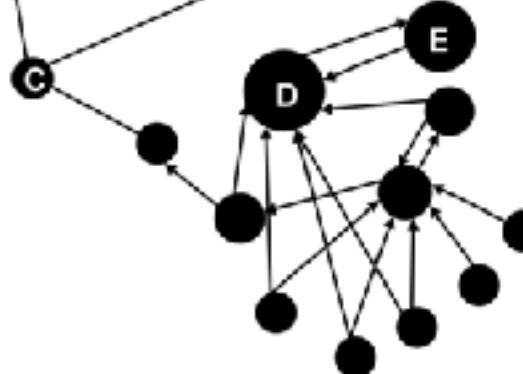
"A" has a high degree



## Betweenness

Which node has the most control over flow between nodes and groups?

"C" is a bridge



## Closeness

Which node can most easily reach all other nodes in a graph or subgraph?

"B" is closest with the fewest hops in its subgraph

## PageRank

Which node is the most important?

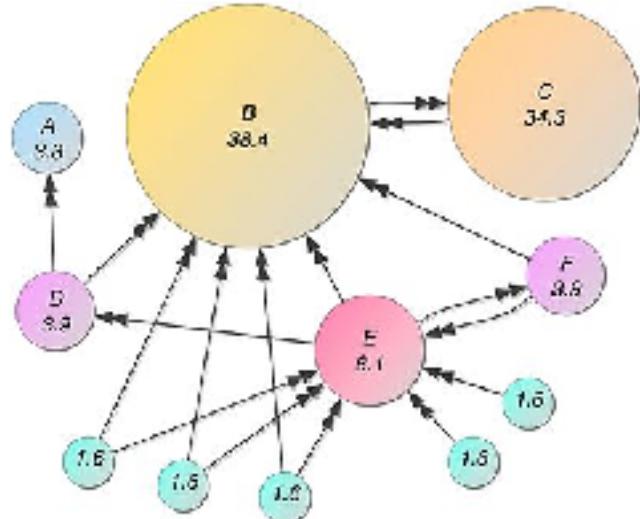
"D" is foremost based on number & weighting of in-links

"E" is next, due to the influence of D's link

# PageRank



# PageRank



Measures the transitive (directional) influence of nodes and considers the influence of neighbors and their neighbors.

- **Dampening factor:** Probability that end user will click through a link. Default value on Neo4j algo is .85.

## Tip / Caution

Test your dampening factor as it will change the outcome.

Note that Spark uses a inverse dampening (0.15)

Careful with mixing node types

# PageRank - Uses

## Use When

Looking for broad influence over a network.

Many domain-specific variations for differing analysis (e.g. Personalized Pagerank for personalized recommendations)

## Recommendations

Who To follow with personalized PR

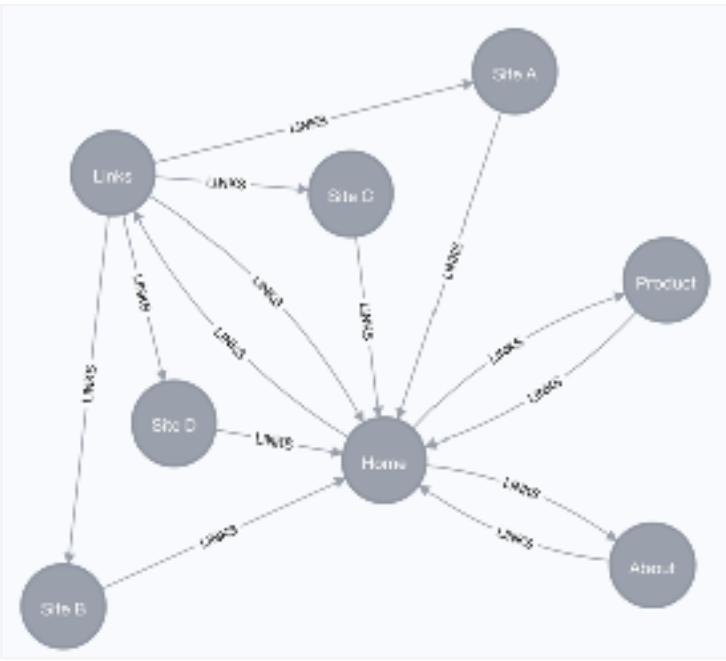


## Fraud Detection

Feature engineering for machine learning



# PageRank example: Using the Page graph



```
CALL algo.pageRank.stream("Page", "LINKS",  
{iterations:20})  
YIELD nodeId, score  
MATCH (node) WHERE id(node) = nodeId  
RETURN node.name AS page, score  
ORDER BY score DESC
```

\$ CALL algo.pageRank.stream("Page", "LINKS", {iterations:20}) YIELD nodeId, score MATCH (node) WHERE id(node) = nodeId RETURN node.name AS page, score ORDER BY score DESC

page	score
"Home"	3.2922017153762284
"About"	1.0611098587023873
"Product"	1.0611098587023873
"Links"	1.0611098587023873
"Site A"	0.3292259009438567
"Site B"	0.3292259009438567
"Site C"	0.3292259009438567
"Site D"	0.3292259009438567

Started streaming 8 records after 49 ms and completed after 49 ms.

# PageRank example in NEuler - 1

Graph Algorithms Playground

NEuler

Page Rank

Measure the transitive influence or connectivity of nodes

Label: Page

Relationship Type: UNIS

Dimension: Out

Show results: sorted

Generativity: 1

Weight property: weight property

Iterations: 10

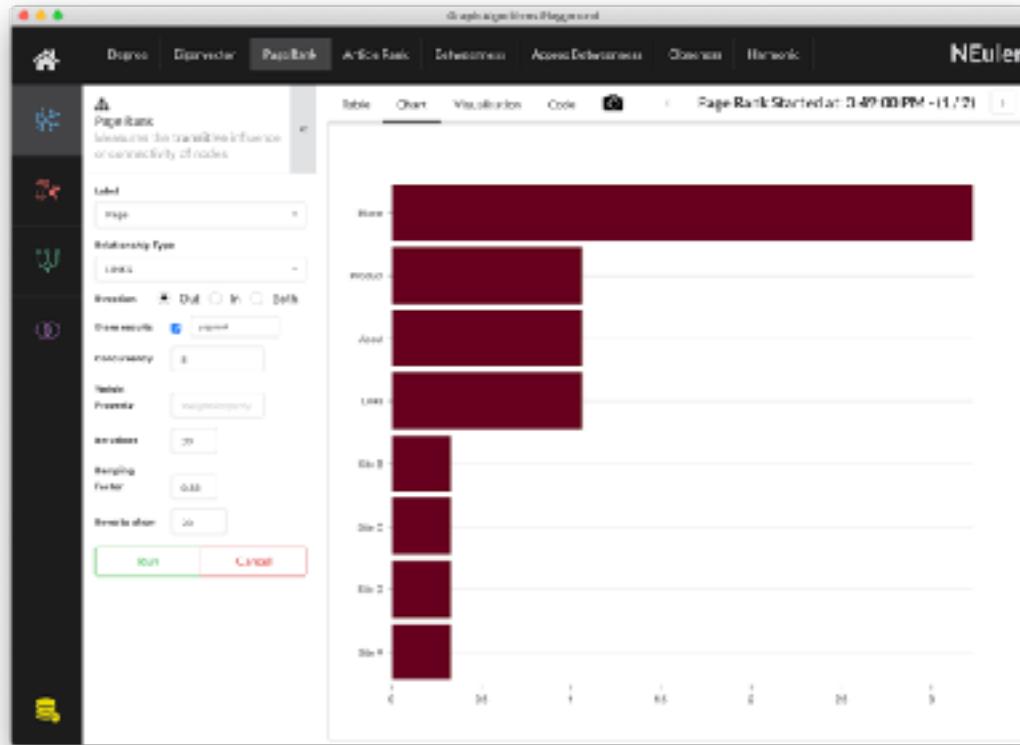
Sampling Factor: 0.05

Run Iterations: 10

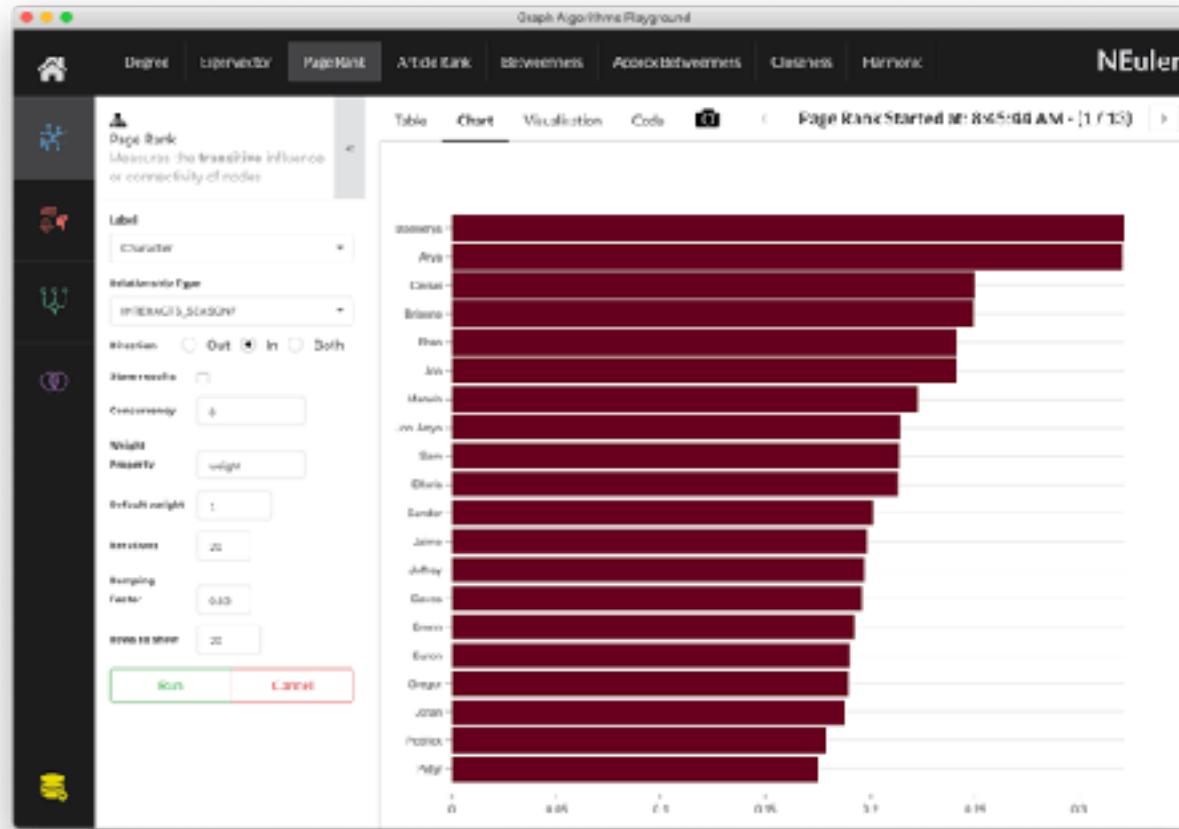
Run Cancel

label	properties	score
Page	same Home	0.2262010763762084
Page	same Product	1.0011886667022070
Page	same About	1.0011886667022073
Page	same Link	1.0011886667022073
Page	same Site B	0.1262250000000000
Page	same Site C	0.1262250000000000
Page	same Site D	0.1262250000000000
Page	same Site A	0.1262250000000000

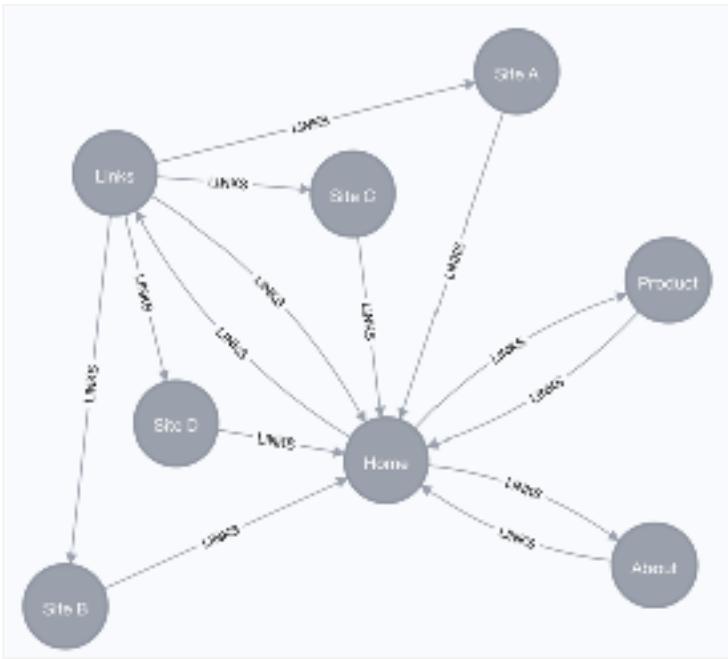
# PageRank example in NEuler - 2



# PageRank: GOT



# Personalized PageRank example: Page graph



```
MATCH (p:Page)
WHERE p.name = 'Site A' OR p.name = 'Site C'
with collect(p) AS sites
CALL algo.pageRank.stream("Page", "LINKS",
{iterations:20, sourceNodes: sites})
YIELD nodeId, score
MATCH (node) WHERE id(node) = nodeId
RETURN node.name AS page, score
ORDER BY score DESC
```

\$ MATCH (p:Page) WHERE p.name = 'Site A' OR p.name = 'Site C' with c	
page	score
'Home'	0.8031758210000007
'About'	0.22011298526170564
'Product'	0.22011298526170564
'Links'	0.22011298526170564
'Site A'	0.18814851720328465
'Site C'	0.18814851720328465
'Site B'	0.03814851720328462
'Site D'	0.03814851720328462

Started streaming 6 records after 21 ms and completed after 22 ms.



## Exercise 7 PageRank

Estimated time to complete this exercise:

10 minutes

1. In NEuler:

Perform the PageRank analysis on different seasons of GOT.

1. In Neo4j Browser:

`:play intro-graph-algos-exercises (PageRank)`

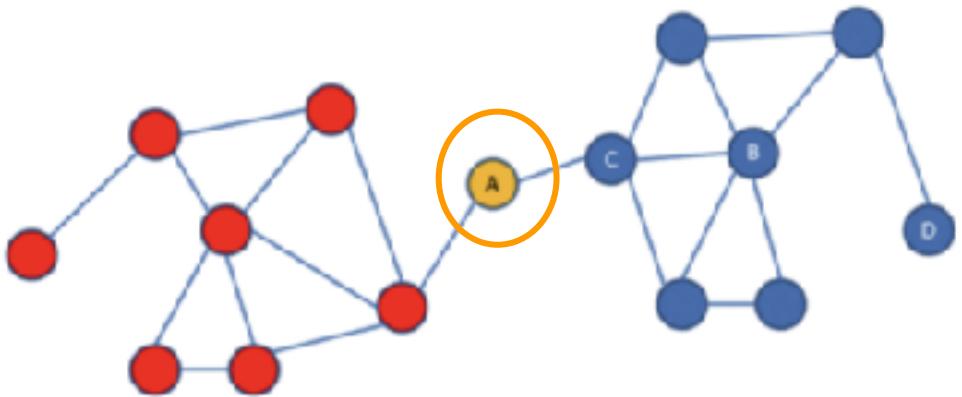
# Betweenness Centrality





# Betweenness Centrality

The sum of the % shortest paths that pass through a node, calculated by pairs.



## Tip / Caution

Computationally intensive: use RA Brandes approximation on large graphs.

Assumes all communication between nodes happens along the shortest path and with the same frequency (not always the case in real life)



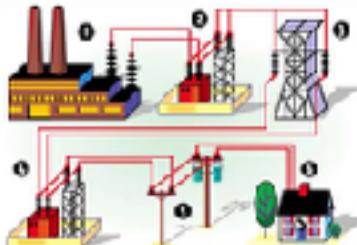
# Betweenness Centrality - Uses

Use When

Identify bridges

Uncover control points

Find bottlenecks and  
vulnerabilities

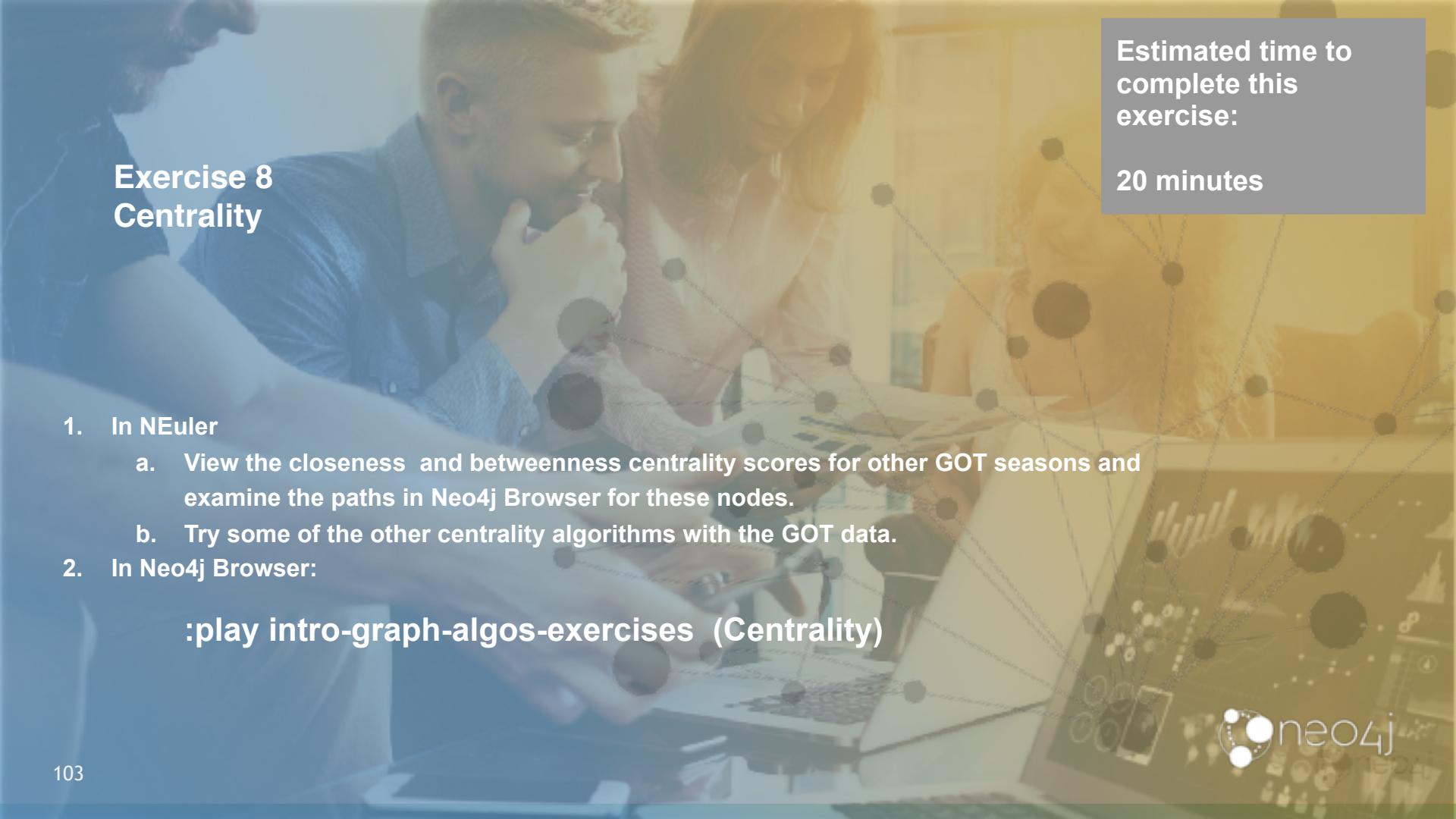


Network Resilience  
Key points of cascading failure

# Relationship Directions? Weights?

## Centrality algorithms

Algorithm	Directed	Undirected	Weighted	Unweighted
PageRank	✓	✓	✓	✓
ArticleRank	✓	✓		✓
Betweenness Centrality	✓	✓		✓
Closeness Centrality	✓	✓		✓
Harmonic Centrality		✓		✓
Eigenvector Centrality	✓	✓		✓
Degree Centrality	✓	✓	✓	✓



## Exercise 8 Centrality

Estimated time to complete this exercise:

20 minutes

1. In NEuler
  - a. View the closeness and betweenness centrality scores for other GOT seasons and examine the paths in Neo4j Browser for these nodes.
  - b. Try some of the other centrality algorithms with the GOT data.

2. In Neo4j Browser:

`:play intro-graph-algos-exercises (Centrality)`

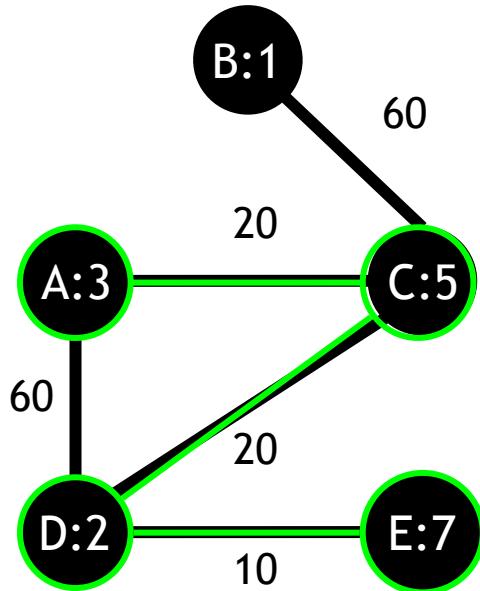
# Pathfinding algorithms

# Pathfinding and Graph Search algorithms

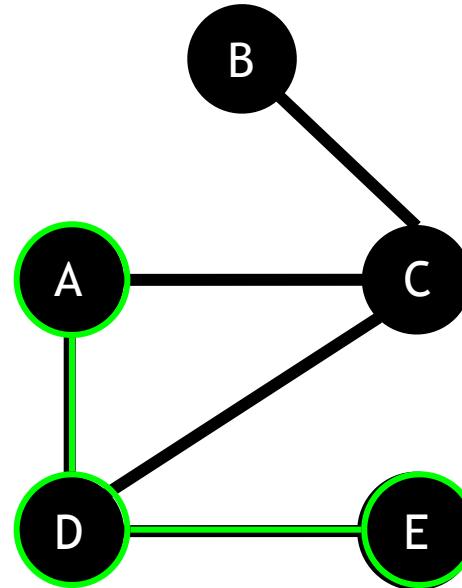


Pathfinding and Graph Search algorithms are used to identify optimal routes, and they are often a required first step for many other types of analysis.

# Shortest Path

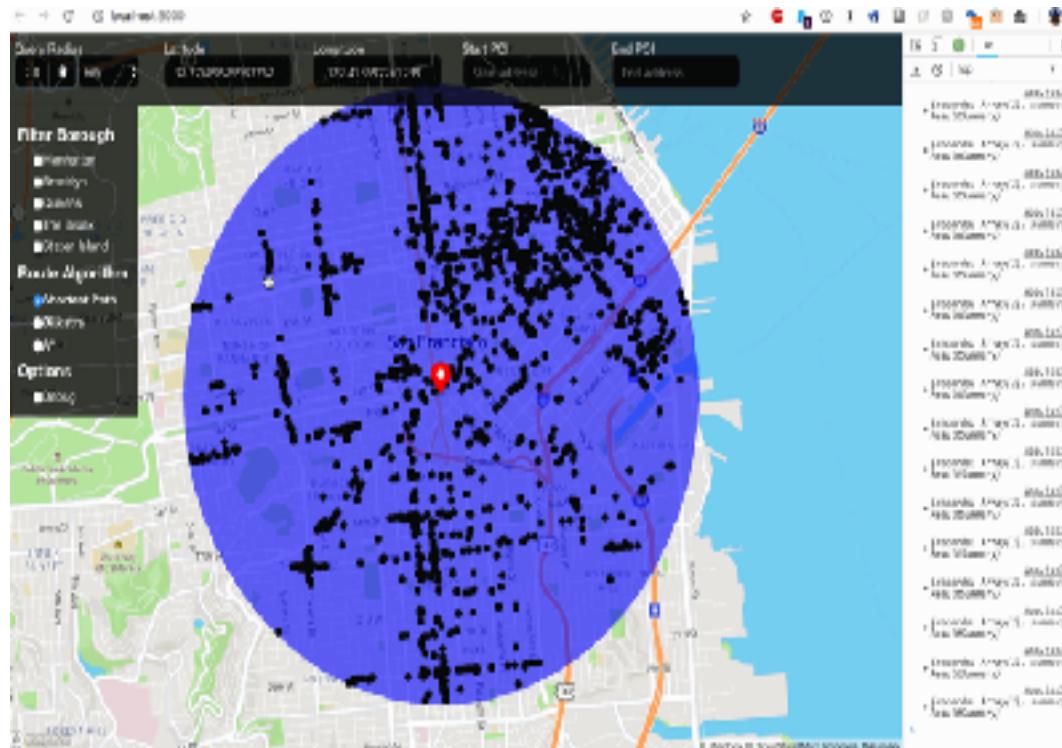


Weighted



Unweighted

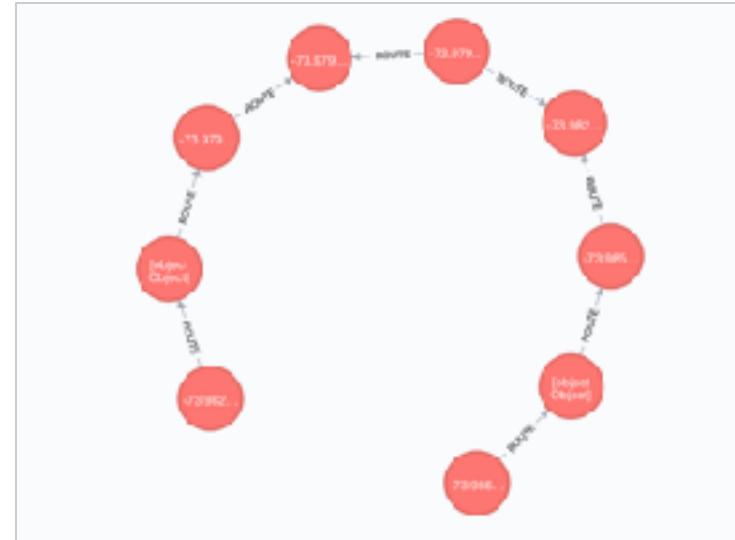
# Shortest Path example



<https://github.com/johnymontana/osm-routing-app>

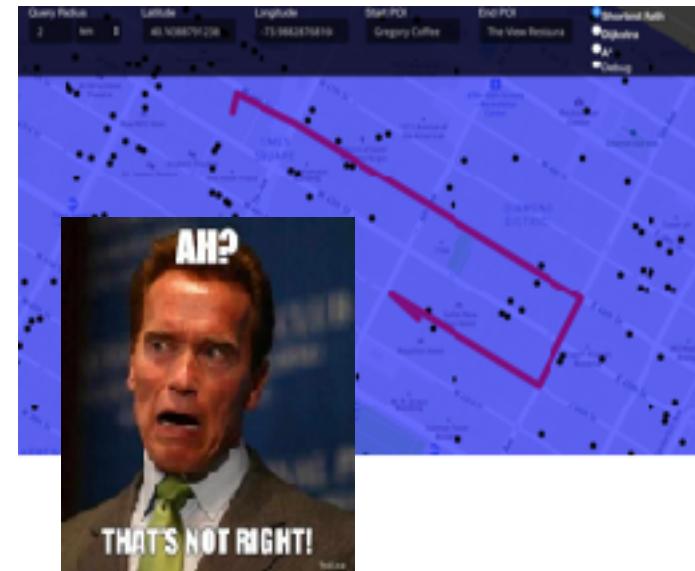
# Routing with Cypher's shortestPath() function

```
MATCH (a:PointOfInterest)
WHERE a.name = 'The View Restaurant & Lounge'
MATCH (b:PointOfInterest)
WHERE b.name = 'Gregory Coffee'
MATCH p=shortestPath((a)-[:ROUTE*..100]-(b))
RETURN p
```

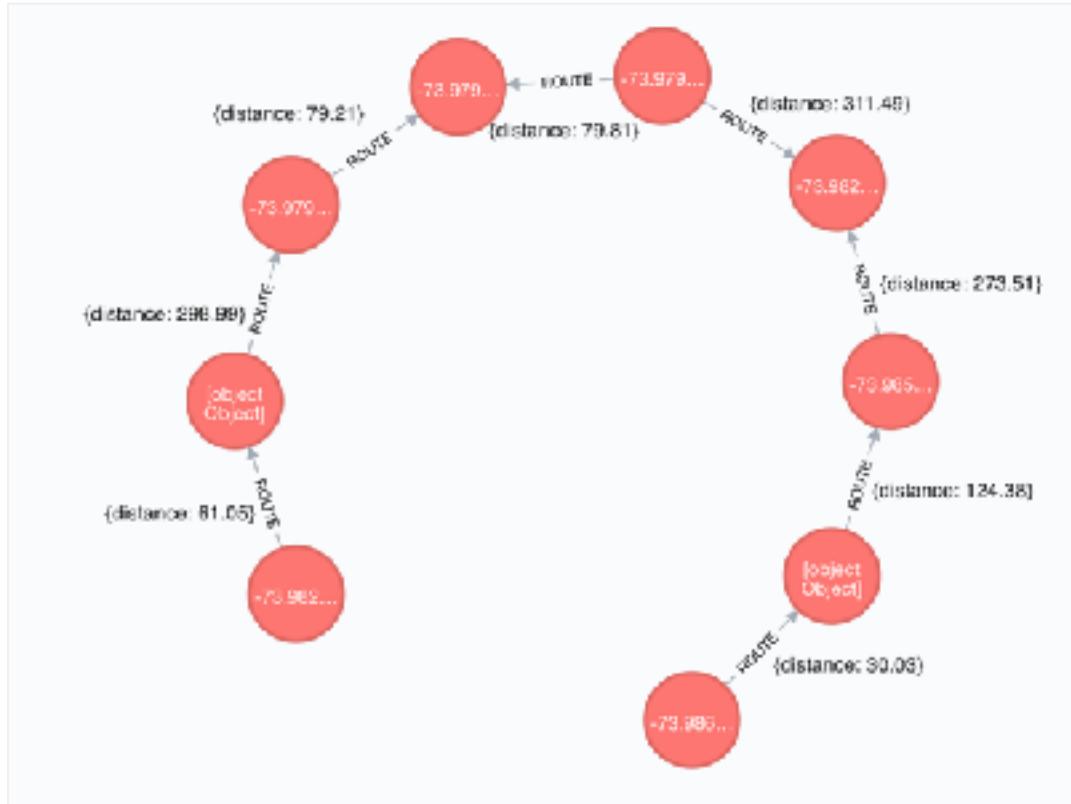


# Cypher's shortestPath() not always best?

```
MATCH (a:PointOfInterest)  
WHERE a.name = 'The View Restaurant & Lounge'  
  
MATCH (b:PointOfInterest)  
WHERE b.name = 'Gregory Coffee'  
  
MATCH p=shortestPath((a)-[:ROUTE*..100]-(b))  
RETURN p
```



# Weighted Paths



**Total weight:  
1258.48m**

# Shortest Weighted Path algorithms

## Dijkstra

- Similar to BFS.
- Uses priority queue to **explore shorter paths first**.
- No concept of direction.
- No intermediate evaluation.

## A\*

- Extends Dijkstra with heuristic.
- Priority queue is ordered by sum of weights plus **heuristic** (how much further).
- Explore paths in the right direction first!

# Shortest Weighted Path algorithms

*Dijkstra*



*A\**



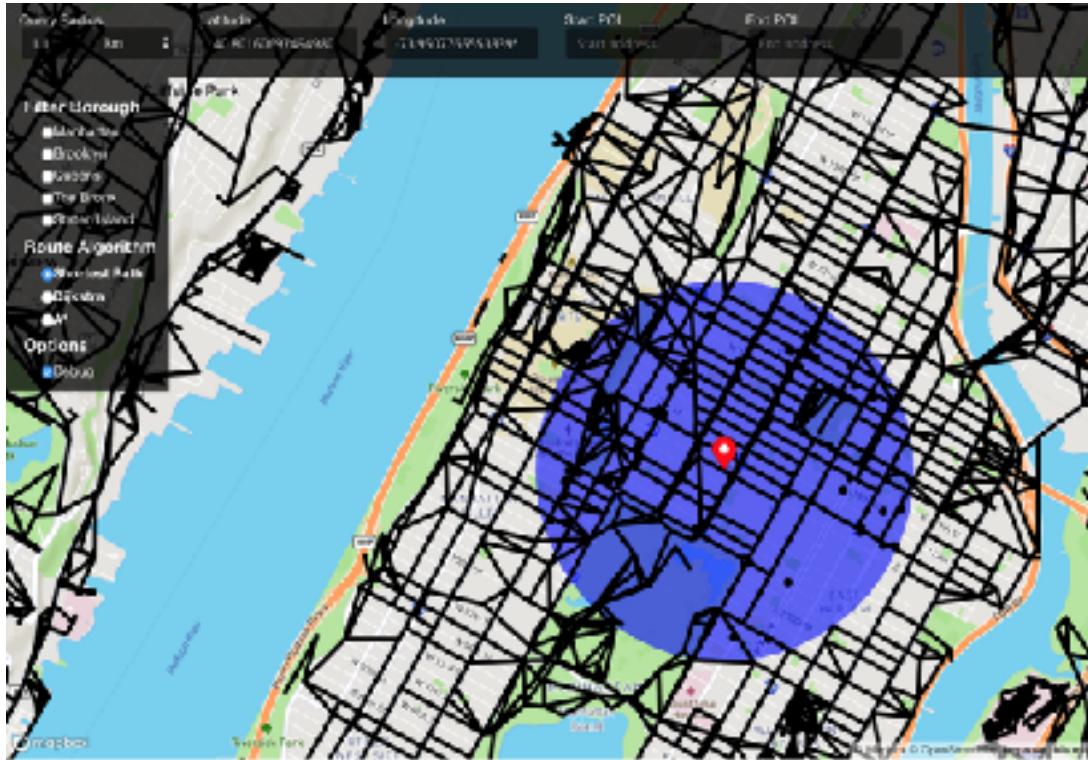
# Example: Routing with Dijkstra algorithm

```
MATCH (a:PointOfInterest) WHERE a.name = 'The View Restaurant & Lounge'  
MATCH (b:PointOfInterest) WHERE b.name = 'Gregory Coffee'  
  
CALL algo.shortestPath.stream(a, b, 'distance',  
{relationshipQuery: 'ROUTE', nodeQuery:'Routable', direction:'BOTH', defaultValue:1.0})  
YIELD nodeId, cost  
  
MATCH (n) WHERE id(n) = nodeId  
RETURN COLLECT({lat: n.location.latitude, lon: n.location.longitude}) AS route
```

# Example: Routing with A\*

```
MATCH (a:PointOfInterest) WHERE a.name = 'The View Restaurant & Lounge'  
MATCH (b:PointOfInterest) WHERE b.name = 'Gregory Coffee'  
  
CALL algo.shortestPath.astar.stream(a, b, distance, 'lat', 'lon',  
{relationshipQuery: 'ROUTE', nodeQuery:'Routable', direction:'BOTH', defaultValue:1.0})  
  
YIELD nodeId, cost  
MATCH (route) WHERE id(route) = nodeId  
RETURN COLLECT({lat: route.location.latitude, lon: route.location.longitude}) AS route
```

# Limit search space with Cypher projections



# Example: Limit search space with Cypher projections

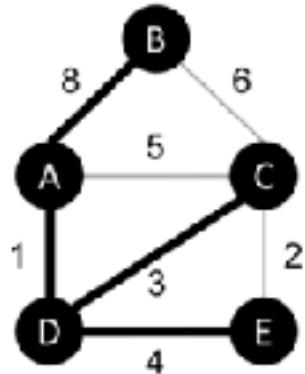
```
MATCH (a:PointOfInterest) WHERE a.poi_id = $startPOI  
MATCH (b:PointOfInterest) WHERE b.poi_id = $endPOI  
CALL algo.shortestPath.astar.stream(a, b, 'weight', 'lat', 'lon',
```

```
{  
    relationshipQuery: "MATCH (a1:Routable)-[r:ROUTE]-(a2:Routable) WHERE distance(a1.location,$center) < $radius  
AND distance(a2.location, $center) < $radius RETURN id(a1) as source, id(a2) as target,r.distance as weight",  
    nodeQuery:"MATCH (a1:Routable) WHERE distance(a1.location, $center) < $radius RETURN id(a1) AS id",  
  
    direction:'both', defaultValue:1.0, graph:'cypher',  
    params: {center: point({latitude: $routeCenterLat, longitude: $routeCenterLon}), radius: $routeRadius}  
})
```

```
FIELD nodeId, cost
```

```
MATCH (route) WHERE id(route) = nodeId  
RETURN COLLECT({lat: route.location.latitude, lon: route.location.longitude}) AS route
```

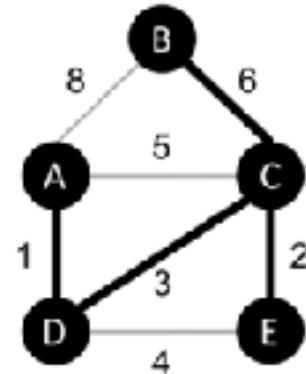
# Other types of Pathfinding algorithms



## Single Source Shortest Path

Shortest path from a **root node** (A shown) to all other nodes

Traverses to the next unvisited node via the lowest cumulative weight from the root



## Minimum Weight Spanning Tree

Shortest path connecting all nodes (A start shown)

Traverses to the next unvisited node via the lowest weight from any visited node

# Relationship Directions? Weights?

## Pathfinding algorithms

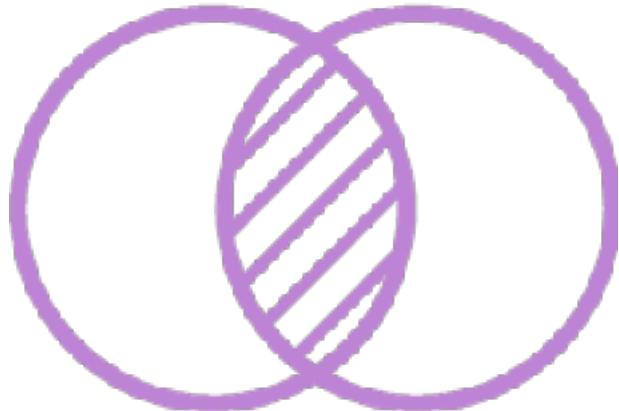
Algorithm	Directed	Undirected	Weighted	Unweighted
Minimum Weight Spanning Tree		✓	✓	
Shortest Path	✓	✓	✓	✓
Single Source Shortest Path	✓	✓	✓	✓
All Pairs Shortest Path		✓	✓	✓
A*	✓	✓	✓	✓
Ken's K-shortest Paths	✓	✓	✓	✓
Random Walk		✓		✓



**BREAK**

# Similarity algorithms

# Similarity Algorithms

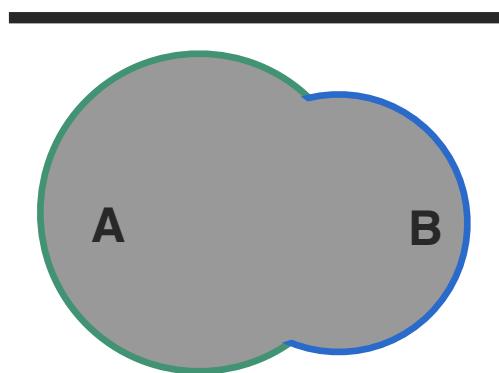
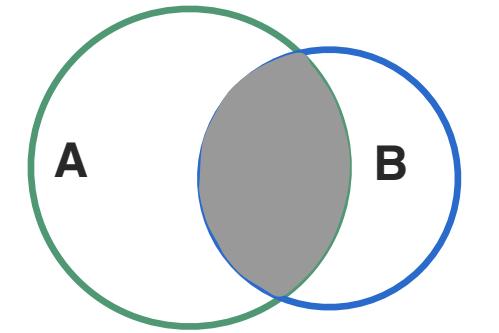


Evaluates how alike nodes are at an individual level either based on node attributes, neighboring nodes, or relationship properties

# Jaccard Similarity coefficient

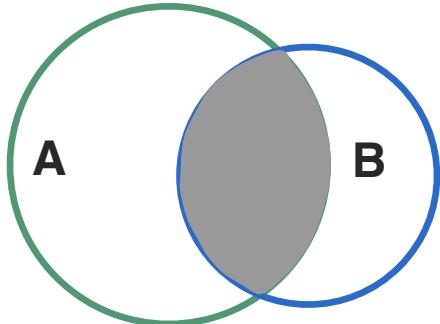
Often used to find recommendations of similar items as well as part of link prediction.

Jaccard Similarity measures the similarity between sets.



$$\frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

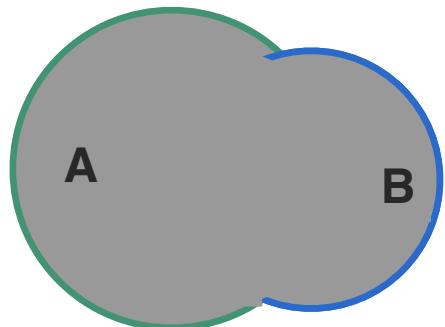
# Jaccard Similarity coefficient



$A = \{\text{Orange, Banana, Cherry, Pineapple}\}$

$B = \{\text{Orange, Banana, Apple}\}$

$$|A \cap B| = |\{\text{Orange, Banana}\}| = 2$$



$$|A \cup B| = |A| + |B| - |A \cap B|$$

$$|A \cup B| = |\{\text{Orange, Banana, Cherry, Pineapple, Apple}\}| = 5$$

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = 2/5 = 0.4$$

# Jaccard Similarity in NEuler

Graph Algorithms Playground

NEuler

Jaccard Started at: 11:27:15 AM - (1 / 12)

Jaccard

measures similarities between sets. It is defined as the size of the intersection divided by the size of the union of two sets.

Item Label: Customer

Relationship Type: LIKES

Category Label: Cuisine

Show results:

Concurrency: 8

Similarity cutoff: 0.1

Degree Cutoff: 0

Rows to show: 20

Run Cancel

From Labels	From Properties	To Labels	To Properties	Similarity
Customer	name: arya	Customer	name: kann	0.35
Customer	name: zhen	Customer	name: Michael	0.4
Customer	name: zhen	Customer	name: Pravenna	0.4
Customer	name: Michael	Customer	name: kann	0.4
Customer	name: Pravenna	Customer	name: Michael	0.000300030003000
Customer	name: Michael	Customer	name: Arya	0.3333333333333333
Customer	name: Pravenna	Customer	name: Arya	0.3333333333333333
Customer	name: Braawaa	Customer	name: kann	0.1666666666666666

# Example: Jaccard Similarity procedure

```
MATCH (c:Customer)-[:LIKES]->(cuis)
WITH {item:id(c), categories:
collect(id(cuis))} as userData
WITH collect(userData) as data
CALL
algo.similarity.jaccard.stream(data)
YIELD item1, item2, count1, count2,
intersection, similarity
RETURN algo.asNode(item1).name AS from,
      algo.asNode(item2).name AS to,
      intersection,
      similarity
ORDER BY similarity DESC
```

from	to	intersection	similarity
"Arya"	"Karin"	3	0.75
"Zhen"	"Praveena"	2	0.4
"Zhen"	"Michael"	2	0.4
"Michael"	"Karin"	2	0.4
"Praveena"	"Michael"	2	0.3333333333333333
"Praveena"	"Arya"	2	0.3333333333333333
"Michael"	"Arya"	2	0.3333333333333333
"Praveena"	"Karin"	1	0.1666666666666666
"Zhen"	"Arya"	0	0.0
"Zhen"	"Karin"	0	0.0

# Warning: Similarity algorithms are computationally intensive

Similarity calculations can be slow:

- create vectors for every node to calculate metrics
- Pair comparison between all nodes in the graph
- Writing back many relationships

What can you do instead?

- **Set your similarity threshold high** and only writeback relationships above threshold
- **Pre-process your data** (instead of unwind) to create vectors ahead of the algorithm call
- **Subset your graph** using an algorithm like UnionFind or LabelPropagation

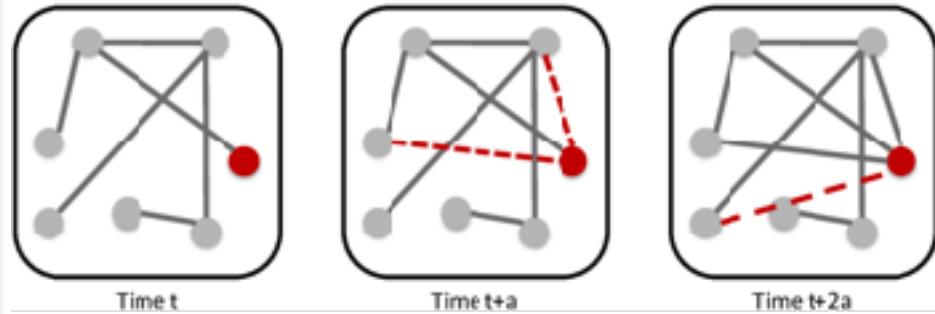
# Link Prediction

# Link Prediction

*Can we infer which new interactions are likely to occur in the future?*

“We formalize this question as the **link prediction problem**, and develop approaches to link prediction based on measures for **analyzing the “proximity” of nodes** in a network.”

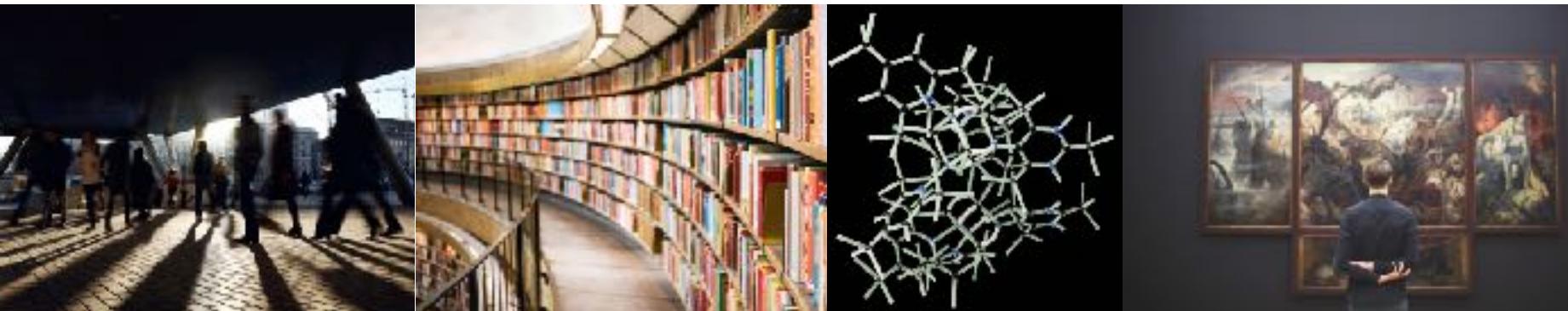
Jon Kleinberg and David Liben-Nowell



A Goal, an Approach  
&  
an Algorithm Category

# What's common across all these use cases?

- future associations in a terrorist network
- co-authorships in a citation network
- associations between molecules in a biology network
- interest in an artist or artwork



# Graph Algorithms used with Link Prediction

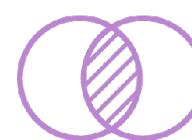


## Link Prediction

These methods compute a score for a pair of nodes, where the score could be considered a **measure of proximity** or “similarity” between those nodes **based on the graph topology**.

## Other Algorithms types

It's common when our goal is link prediction to use a variety of algorithm types to extract features and use them together in a machine learning model



Similarity



Community Detection

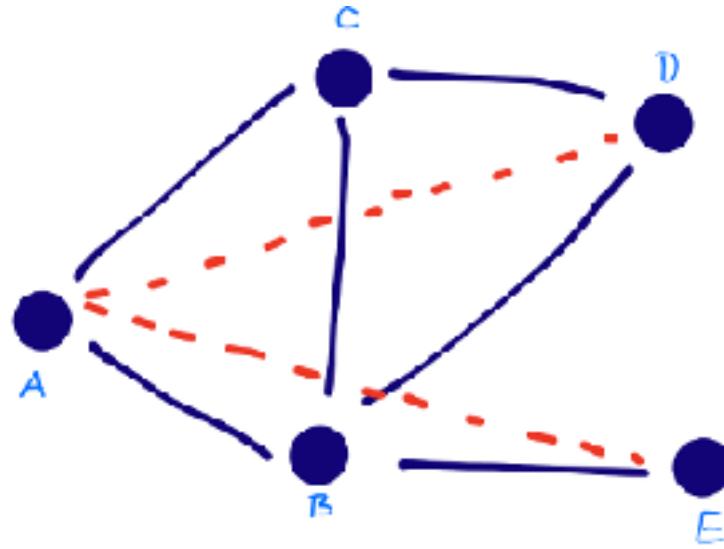


Centrality

# Common Neighbors

Based on number of potential triangles / closing triangles.

Concept is that if 2 strangers have a friend/colleague in common, they are more likely to be introduced.



# Example: Common Neighbors



```
MATCH (e1:Employee {name: 'Larry'})  
MATCH (e2:Employee {name: 'Sophia'})  
RETURN algo.linkprediction.commonNeighbors(e1, e2) AS score
```

score  
2.0

```
MATCH (e1:Employee {name: 'Robert'})  
MATCH (e2:Employee {name: 'Joe'})  
RETURN algo.linkprediction.commonNeighbors(e1, e2) AS score
```

score  
1.0

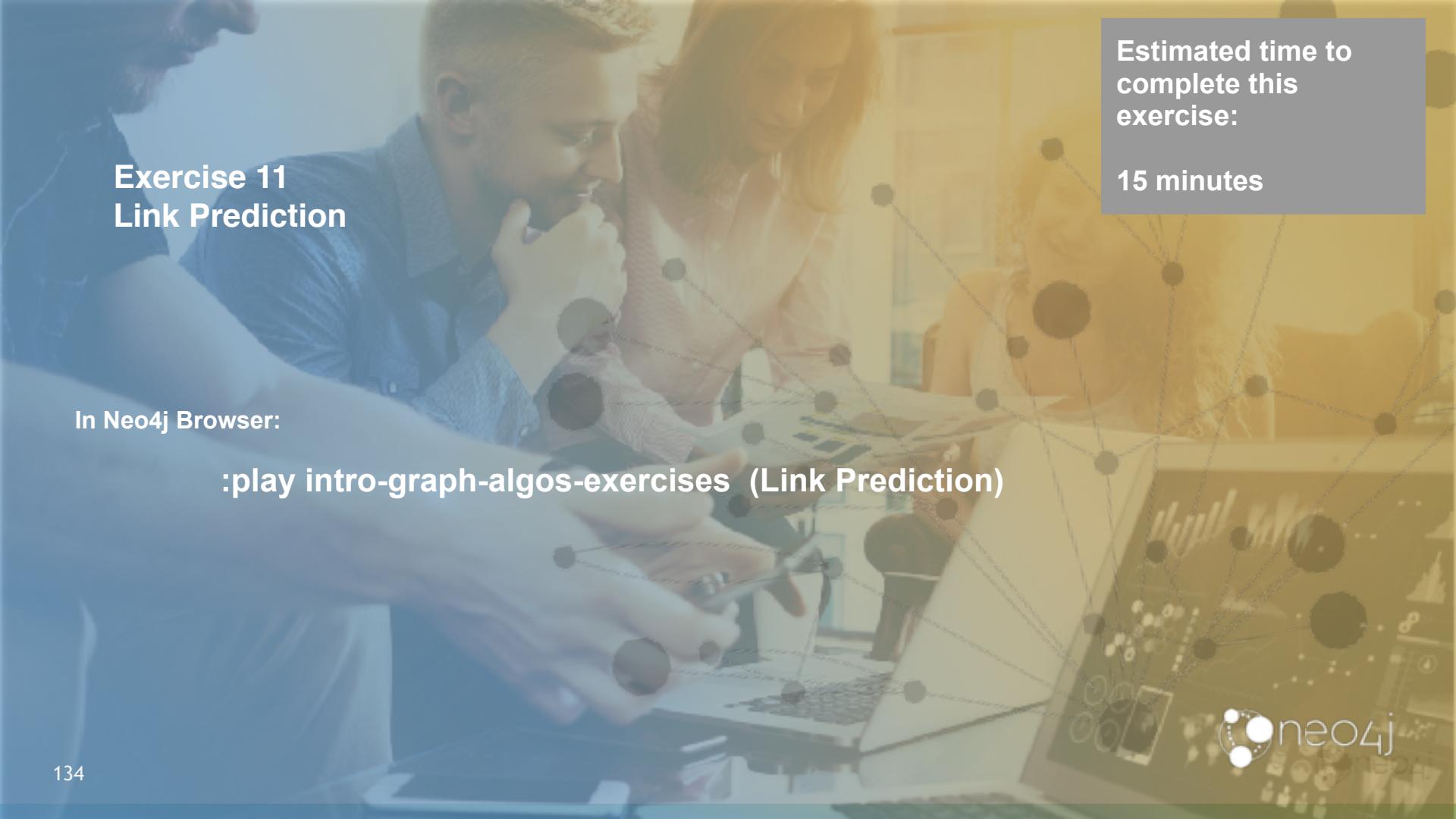
```
MATCH (e1:Employee {name: 'Sophia'})  
MATCH (e2:Employee {name: 'Joe'})  
RETURN algo.linkprediction.commonNeighbors(e1, e2,  
{relationshipQuery: 'WORKS_WITH'}) AS score
```

score  
0.0

# Relationship Directions?

## Link Prediction algorithms

Algorithm	Directed	Undirected	Additional Parameter	Unweighted
Adamic Adar	✓	✓		
Common Neighbors	✓	✓		
Preferential Attachment	✓	✓		
Resource Allocation	✓	✓		
Same Community			community	
Total Neighbors	✓	✓		



## Exercise 11

### Link Prediction

Estimated time to complete this exercise:

15 minutes

In Neo4j Browser:

```
:play intro-graph-algos-exercises (Link Prediction)
```

# Resources



OPERATIONS MANUAL   CYPHER MANUAL   DRIVER MANUAL   OGM MANUAL   GRAPH ALGORITHMS   JAVA REFERENCE   VERSIONS   Search Neo4j docs...

Table of Contents

- 1. Introduction
  - 1.1. Algorithms
  - 1.2. Installation
  - 1.3. Usage
- 2. Projected Graph Model
  - 2.1. Label and relationship-type projection
  - 2.2. Cypher projections
  - 2.3. Named graphs
- 3. The Yelp example
  - 3.1. The Yelp Open Dataset
  - 3.2. Data
  - 3.3. Graph model
  - 3.4. Import
  - 3.5. Neo4j
- 4. Procedures
- 5. Centrality algorithms
  - 5.1. The PageRank algorithm
  - 5.2. The AdamicRank algorithm
  - 5.3. The Betweenness Centrality algorithm
  - 5.4. The Closeness Centrality algorithm
  - 5.5. The harmonic Centrality algorithm
  - 5.6. The eigenvector Centrality algorithm
  - 5.7. The Degree Centrality algorithm
- 6. Community detection algorithms
  - 6.1. The Louvain algorithm
  - 6.2. The Label Propagation algorithm
  - 6.3. The Connected Components algorithm
  - 6.4. The Strongly Connected Components algorithm
  - 6.5. The Triangle Counting / Clustering Coefficient algorithm

introduction

# The Neo4j Graph Algorithms User Guide v3.5

Copyright © 2019 Neo4j, Inc.

Licence: Creative Commons 4.0

This is the user guide for Neo4j Graph Algorithms version 3.5, authored by the Neo4j Team.

The guide covers the following areas:

- [Chapter 1, Introduction](#) — An introduction to Neo4j Graph Algorithms.
- [Chapter 2, Projected Cypher Model](#) — A detailed guide to the projected graph model.
- [Chapter 3, The Yelp example](#) — An illustration of how to use graph algorithms on a social network of friends.
- [Chapter 4, Procedures](#) — A list of Neo4j Graph Algorithm procedures.
- [Chapter 5, Centrality algorithms](#) — A detailed guide to each of the centrality algorithms, including use-cases and examples.
- [Chapter 6, Community detection algorithms](#) — A detailed guide to each of the community detection algorithms, including use-cases and examples.
- [Chapter 7, Path finding algorithms](#) — A detailed guide to each of the path finding algorithms, including use-cases and examples.
- [Chapter 8, Similarity algorithms](#) — A detailed guide to each of the similarity algorithms, including use-cases and examples.
- [Chapter 9, Link Prediction algorithms](#) — A detailed guide to each of the link prediction algorithms, including use-cases and examples.
- [Chapter 10, Preprocessing/functions and procedures](#) — A detailed guide to each of the preprocessing functions and procedures.

# neo4jsandbox.com

\$ play https://guides.neo4j.com/sandbox/graph-algorithms/

## Community Detection

We can detect communities in our data by running an algorithm which traverses the graph structure to find highly connected subgraphs with fewer connections to other, other subgraphs.

Run the following query to calculate the communities that exist based on interactions across all the books.

```
graph TD
    subgraph "All"
        alg1[Label Propagation]
        alg1 --> R1[Match (e)-[:character]->(v1) as $el1]
        alg1 --> R2[Match (e)-[:character]->(v2) BFTB((e)) as source, in(v2) as target, $el1/v1.weight as weight]
        alg1 --> R3[Match (e)->(v1), part1[property: "communities"]]
    end
```

## Russian Twitter Trolls



Explore data released by NBC News from their investigation into Russian Twitter Trolls around the 2016 US election.

[Launch Sandbox](#)

## Recommendations



Generate personalized real-time recommendations using a dataset of movie reviews.

[Launch Sandbox](#)

## Graph Algorithms



Learn graph algorithms with simple examples.

[Launch Sandbox](#)

## Yelp Public Dataset



Explore businesses, reviews, and users from Yelp's public dataset.

[Launch Sandbox](#)

# Online Training!

## Applied Graph Algorithms

The screenshot shows the Neo4j Graph Academy interface for the 'Applied Graph Algorithms' module. On the left, a sidebar lists 'Module: Photo Recommendation' with sections for 'Recommender Systems', 'Check your understanding', 'Dive deeper', 'Summary', and 'Grade Quiz and Continue'. The main content area is titled 'Solution' and contains a video player showing a video titled 'How to Implement a Photo Recommendation System Using Neo4j'. To the right of the video is a 'Photo Recommendations' section with a 'View Details' button and three thumbnail images: 'Vegan Smoothie', 'Jalapeño Queso', and 'Fried Chicken'. At the bottom right of the main content area is a 'Summary' button.

<https://neo4j.com/graphacademy/online-training/applied-graph-algorithms>

## Data Science With Neo4j

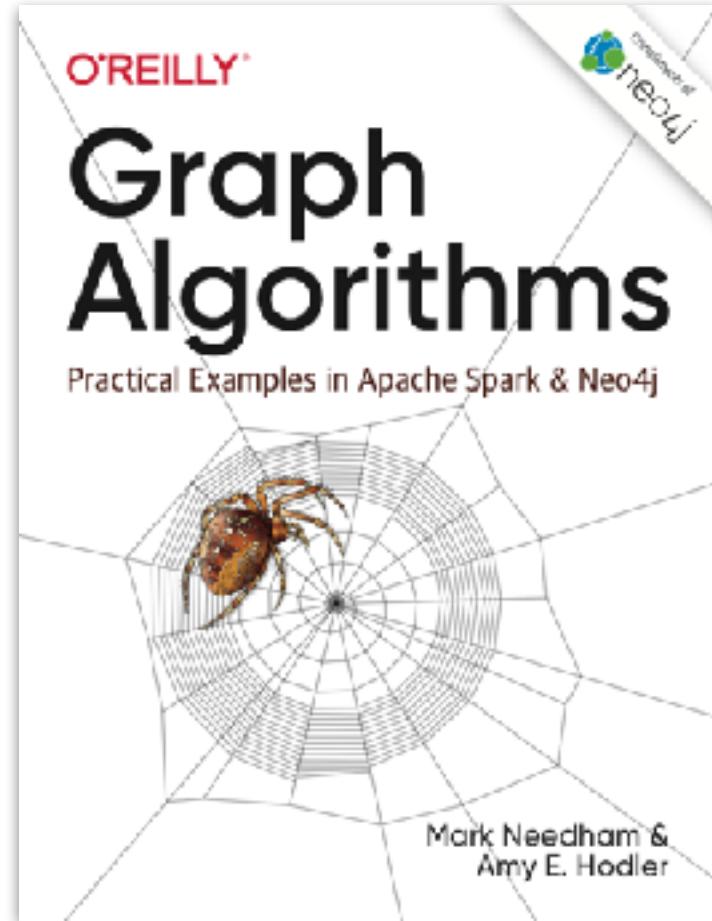
The screenshot shows the Neo4j Graph Academy interface for the 'Data Science With Neo4j' module. On the left, a sidebar lists 'Module: Predictions' with sections for 'About this module', 'The Link Prediction Problem', 'Link Prediction Algorithms', 'Exercise 1: Running the prediction algorithms', 'Applying Link Prediction Algorithms', 'Using the measures directly', 'Supervised learning', 'Exercise 2: Building a binary classifier', 'Check your understanding', 'Question 1', 'Question 2', 'Question 3', 'Summary', and 'Grade Quiz and Continue'. The main content area is titled 'Exploratory Data Analysis' and 'The Link Prediction Problem'. It includes a brief history of link prediction, mentioning its origin in a paper by Jon Kleinberg and David Liben-Nowell in 1999, and a diagram illustrating link prediction over time steps T1, T2, and T3. Below the diagram, it discusses the Kleinberg and Liben-Nowell approach from the perspective of social networks. A note at the bottom explains that link prediction involves predicting future interactions based on existing network data.

<https://neo4j.com/graphacademy/online-training/data-science>

# Free O'Reilly Book

[neo4j.com/  
graph-algorithms-book](http://neo4j.com/graph-algorithms-book)

- Spark & Neo4j Examples
- Machine Learning Chapter



# Questions?

Justin Fine  
[Justin.Fine@neo4j.com](mailto:Justin.Fine@neo4j.com)

**Thank you!**