Neo4j Graph Database

Installing and Running Neo4j

- Go to http://neo4j.com/download
- Download Community Edition for your OS
- For Windows run the exe file to install, then use the installed application to manage neo4j server
- For Linux/Mac un-compress the downloaded file and run the "*./neo4j start*" command from within the included bin directory

How to use Neo4j

- Cypher
  - command line (neo4j-shell)
  - web interface (defaults at http://localhost:7474)
- Neo4j Language Drivers
  - java
  - .NET
  - JavaScript
  - Python
  - Ruby
  - PHP

  and more!

Cypher Query Language

- Declarative, SQL-inspired language
- Used to describe patterns in graphs
- User describes **what** she wants to
  - select
  - insert
  - update
  - delete
- Without describing **how** to do it
- Cypher Documentation: http://neo4j.com/docs/stable/cypher-query-lang.html
- Cypher Reference Card: http://neo4j.com/docs/stable/cypher-refcard/

Cypher Nodes Representation

- Cypher uses ASCII-Art to represent patterns
- Surround nodes with parentheses so it looks like a circle
  - e.g. (person), (movie)
- A node can have properties
  - e.g. (bob {age: 28, name: 'Bob'})
- In the above examples *bob, person, movie* are variables names
- A relationship among nodes is represented with an arrow as:
- e.g. (bob) --> (mary), or (bob)--(mary) bidirectional

Cypher Relationships Representation

- A relationship has a type, e.g. :LIKES
- Surround relationships with square brackets
  - e.g. [:LIKES]
  - :LIKES is the type of the relationship
- Relationships are declared as:
  - (bob)-[:LIKES]->(mary)
- Relationships can also have properties:
  - (bob)-[:GRADUATED {year: 2015}]->(aueb)

Cypher Labels Representation

- Labels allow us to assign roles or types to nodes
  - e.g. (bob:Person)
- Can have more than one label per node
  - e.g. (bob:Person:Student:Actor)
- In the relational world the label would most probably be the name of a table

MATCH & RETURN

- **MATCH:** used to match patterns of nodes and relationships in the graph
- **RETURN:** declare what information you want returned from the query
- Describe a pattern and ask the database to return the desired info
- A very basic example is:

```
MATCH (p1:Person)-[:Friend]->(p2:Person)
RETURN p1.name, p2.name
```

WHERE, ORDER BY, LIMIT

- **WHERE:** filter results by properties values
- **ORDER BY:** ask for a specific order of results
- **LIMIT:** how many results to show

```
MATCH (p:Person)-[r:Acted]->(m:Movie)
WHERE m.year = 1995
RETURN m.title AS title, p.name, r.role
ORDER BY title ASC LIMIT 10;
```

Describing Paths

- **(a)-[*2]->(b)**
  - all paths of length 2
- **(a)-[*3..5]->(b)**
  - all paths of length 3 to 5
- **(a)-[*]->(b)**
  - all paths of any length
- **shortestPath((a)-[*..5]->(b))**
  - shortest path of max length 5

Aggregation

- MATCH (n:Person) RETURN count(n)
- MATCH (n:Person) RETURN collect(n.name)
- MATCH (p:Person{name:'bob'})-[:OWNS]->(n:BankAccount) RETURN sum (n.amount)
- Other available aggregate functions:
  - avg
  - min
  - max
  - percentileDisc
  - stdev

Mathematical Functions

- abs
- rand
- round
- sqrt
- sign
- sin
- log
- log10

and more!

CREATE

CREATE: create new nodes and relationships

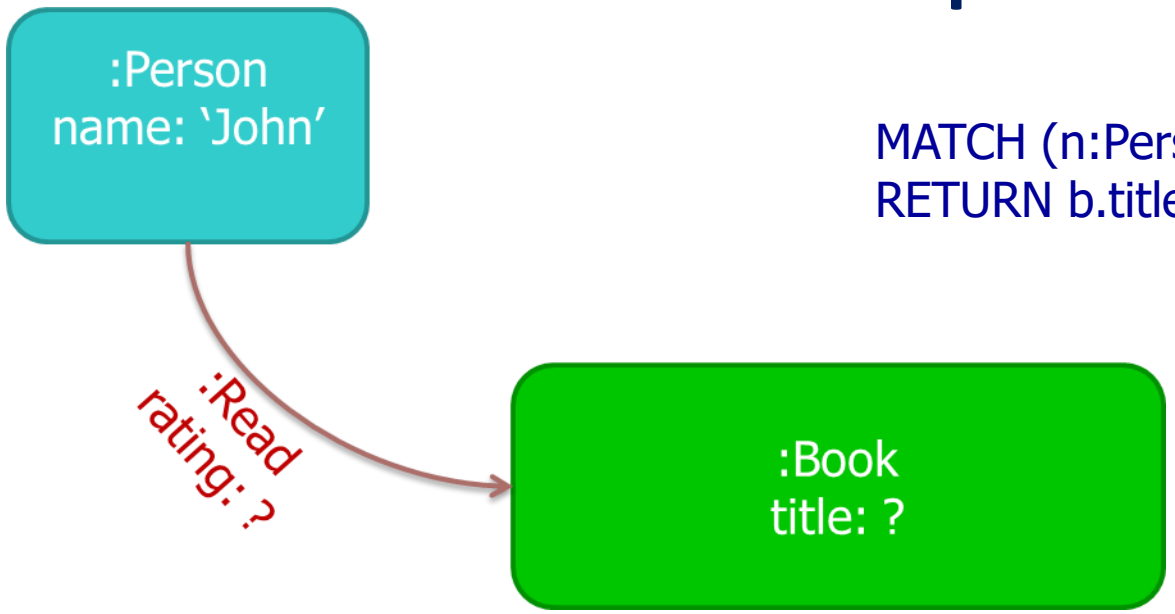
```
CREATE (a:Person{name:'Bob'})-[:Likes]->(b:Person{name:'Mary'})
```

```
MATCH (x:Person {name:'Bob'})
CREATE (x)-[:WorksAt]->(c:Company{name:'1B Dollars'})
```

Querying the graph database

- Queries are also graphs!

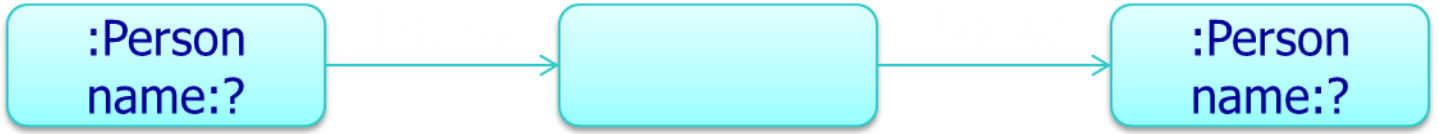
“Find the titles of all books that a person named John has read and report his ratings”




```
MATCH (n:Person {name:'John'})-[r:Read]->(b:Book)
RETURN b.title, r.rating
```

Querying the graph database

- Friend-of-friend pairs in a social network



- MATCH (x:Person)-[:Knows]->(someone),(someone)-[:Knows]->(y:Person)

  RETURN x.name, y.name

OR (simpler)

- MATCH (x:Person)-[:Knows]->()-[:Knows]->(y:Person)

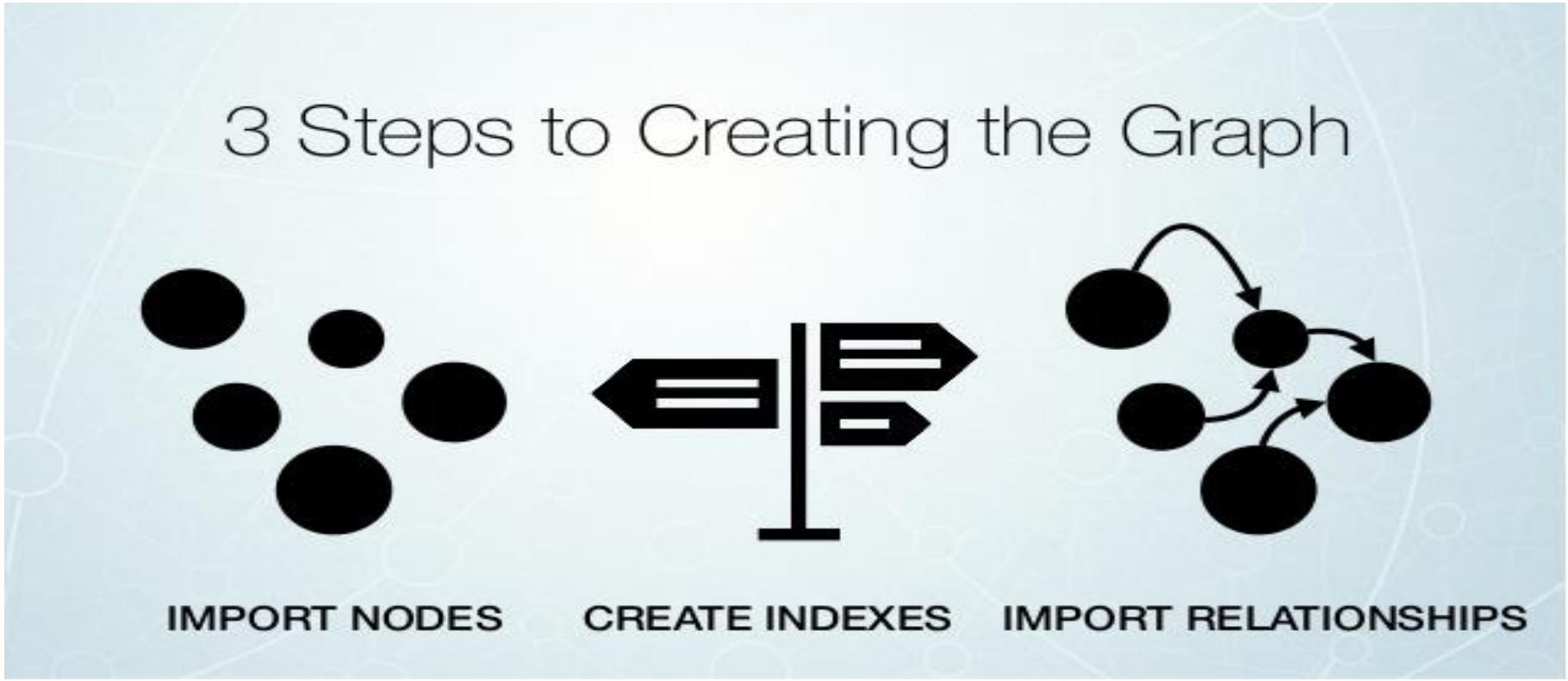
  RETURN x.name, y.name

Import Data

Can use a number of methods:

- Multiple CREATE statements
  - http://neo4j.com/docs/stable/query-create.html
- LOAD CSV FROM ‘path_to_file’ command
  - http://neo4j.com/docs/stable/cypherdoc-importing-csv-files-with-cypher.html
- Neo4j Import Tool
  - http://neo4j.com/docs/stable/import-tool.html

Import Data


3 Steps to Creating the Graph
IMPORT NODES
CREATE INDEXES
IMPORT RELATIONSHIPS

Load CSV From path

- Direct mapping of input data into complex graph/domain structure
- Create or merge data, relationships and structure
- All data from CSV is read as a string, use (toInteger, toFloat, split)
- Separate node creation from relationship creation into different statements
- Create indexes after insertion for the required properties

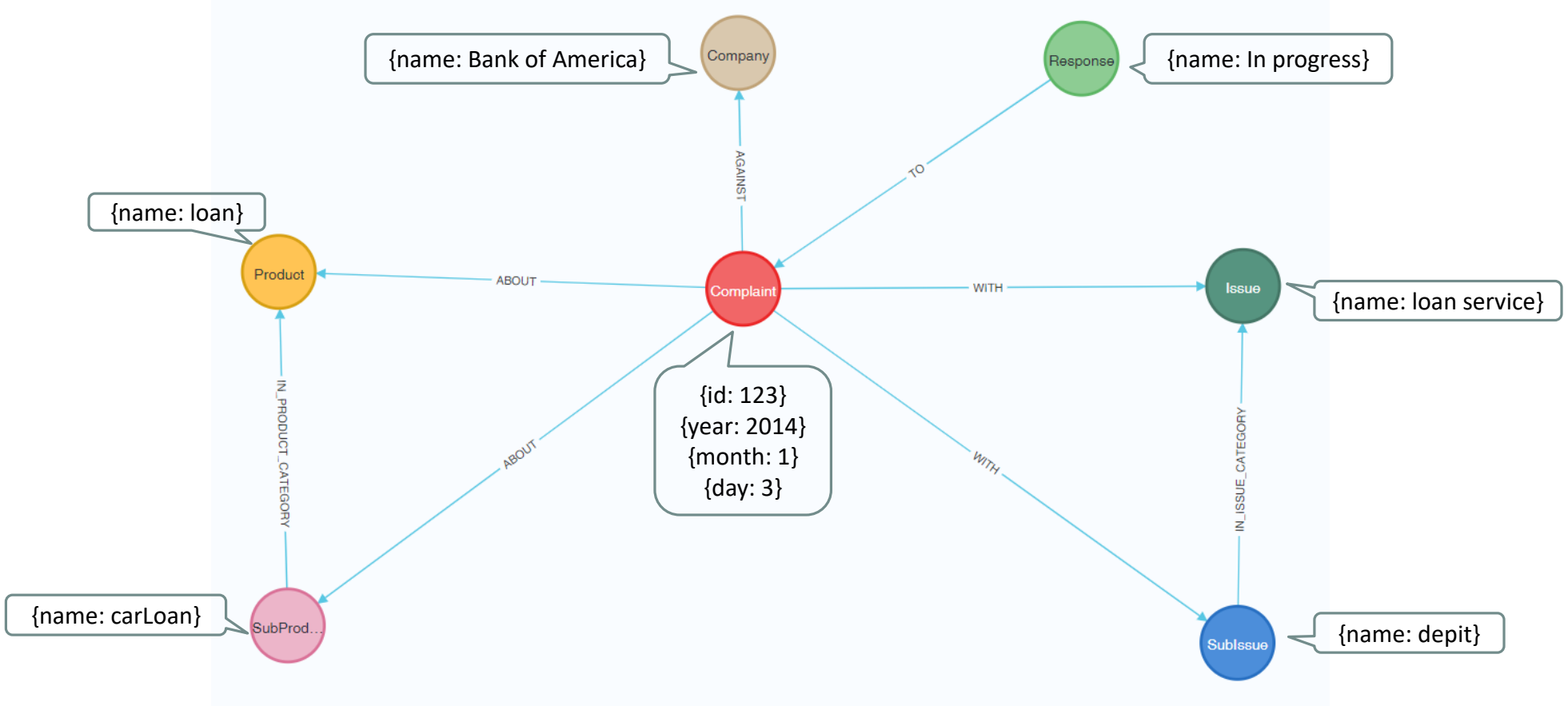
Consumer Complaints Example

Consumer Complaints Example

- Model Description:
- **7 nodes:** Company, Response, Product, SubProduct, Issue, SubIssue, Complaint
- **6 relationships:** TO, AGAINST, ABOUT, WITH, IN_ISSUE_CATEGORY, IN_PRODUCT_CATEGORY
- **1 CSV file:**

Date received	Product	Sub-product	Issue	Sub-issue	Consumer complain	Company	Company response to consumer	Timely response?	Consumer disputed?	Complaint ID
7/29/2013	Consumer Loan	Vehicle loan	Managing the loan or lease			Wells Fargo & Company	Closed with explanation	Yes	No	468882
7/29/2013	Bank account or s	Checking acco	Using a debit or ATM card			Wells Fargo & Company	Closed with explanation	Yes	No	468889
7/29/2013	Bank account or s	Checking acco	Account opening, closing, or management			Santander Bank US	Closed	Yes	No	468879
7/29/2013	Bank account or s	Checking acco	Deposits and withdrawals			Wells Fargo & Company	Closed with explanation	Yes	No	468949
7/29/2013	Mortgage	Conventional	Loan servicing, payments, escrow account			Franklin Credit Managemer	Closed with explanation	Yes	No	475823
7/29/2013	Bank account or s	Checking acco	Deposits and withdrawals			Bank of America	Closed with explanation	Yes	No	468981

Consumer Complaints Example



Consumer Complaints Example

- Read the first line of the CSV-Cypher (check for required properties)

```
LOAD CSV WITH HEADERS FROM
"file:///Consumer_Complaints.csv" AS LINE
RETURN LINE
limit 1
```

Consumer Complaints Load CSV

- **Create**: All Nodes Indexes (unique constraint)

```
// Uniqueness constraints.
CREATE CONSTRAINT FOR (c:Complaint) REQUIRE c.id IS UNIQUE;
CREATE CONSTRAINT FOR (c:Company) REQUIRE c.name IS UNIQUE;
CREATE CONSTRAINT FOR (r:Response) REQUIRE r.name IS UNIQUE;
CREATE CONSTRAINT FOR (p:Product) REQUIRE p.name IS UNIQUE;
CREATE CONSTRAINT FOR (i:Issue) REQUIRE i.name IS UNIQUE;
CREATE CONSTRAINT FOR (s:SubProduct) REQUIRE s.name IS UNIQUE;
CREATE CONSTRAINT FOR (s:SubIssue) REQUIRE s.name IS UNIQUE;
```

Consumer Complaints Load CSV

- **Create**: Complaint nodes with properties (split date)

```
// Load Complaint Nodes.
LOAD CSV WITH HEADERS
FROM "file:///Consumer_Complaints.csv"  AS line
WITH DISTINCT line, SPLIT(line.`Date received`, '/') AS date

CREATE (complaint:Complaint { id: TOINTEGER(line.`Complaint ID`) })
SET complaint.year = TOINTEGER(date[2]),
   complaint.month = TOINTEGER(date[0]),
   complaint.day = TOINTEGER(date[1])
```

Consumer Complaints Load CSV

- **Create**: Company, Response nodes with MERGE (find or create)

```
// Load Company, Response Nodes.
LOAD CSV WITH HEADERS
FROM "file:///Consumer_Complaints.csv"  AS line

MERGE (company:Company { name: TOUPPER(line.Company) })
MERGE (response:Response { name: TOUPPER(line.`Company response to consumer`) })
```

Consumer Complaints Load CSV

- **Create**: AGAINST, TO relationships between nodes (with properties)

```
// Load AGAINST, TO relationships.
LOAD CSV WITH HEADERS
FROM "file:///Consumer_Complaints.csv"  AS line
MATCH (complaint:Complaint { id: TOINTEGER(line.`Complaint ID`) })
MATCH (response:Response { name: TOUPPER(line.`Company response to consumer`) })
MATCH(company:Company { name: TOUPPER(line.Company) })
CREATE (complaint)-[:AGAINST]->(company)
CREATE (response)-[r:TO]->(complaint)
SET r.timely = CASE line.`Timely response?` WHEN 'Yes' THEN true ELSE false END,
   r.disputed = CASE line.`Consumer disputed?` WHEN 'Yes' THEN true ELSE false END;
```

Consumer Complaints Load CSV

- **Create**: Product, Issue nodes and ABOUT, WITH relationships (MATCH on Complaint ID)

```
// Load Product, Issue nodes, ABOUT, WITH relations.
LOAD CSV WITH HEADERS
FROM "file:///Consumer_Complaints.csv"  AS line
MATCH (complaint:Complaint { id: TOINTEGER(line.`Complaint ID`) })
MERGE (product:Product { name: TOUPPER(line.Product) })
MERGE (issue:Issue {name: TOUPPER(line.Issue) })
CREATE (complaint)-[:ABOUT]->(product)
CREATE (complaint)-[:WITH]->(issue);
```

Consumer Complaints Load CSV

- **Create**: Sub-issue node and its relationships (remove empty nodes)

```
// Load Sub-issue nodes and relations.
LOAD CSV WITH HEADERS
FROM "file:///Consumer_Complaints.csv"  AS line WITH line
WHERE line.`Sub-issue` <> '' AND line.`Sub-issue` IS NOT NULL
MATCH (complaint:Complaint { id: TOINTEGER(line.`Complaint ID`) })
MATCH (complaint)-[:WITH]->(issue:Issue)
MERGE (subIssue:SubIssue { name: TOUPPER(line.`Sub-issue`) })
MERGE (subIssue)-[:IN_ISSUE_CATEGORY]->(issue)
CREATE (complaint)-[:WITH]->(subIssue);
```

Consumer Complaints Load CSV

- **Create**: Sub-product node and its relationships (remove empty nodes)

```
// Load Sub-product nodes and relations.
LOAD CSV WITH HEADERS
FROM "file:///Consumer_Complaints.csv"  AS line WITH line
WHERE line.`Sub-product` <> '' AND line.`Sub-product` IS NOT NULL
MATCH (complaint:Complaint { id: TOINTEGER(line.`Complaint ID`) })
MATCH (complaint)-[:ABOUT]->(product:Product)
MERGE (subProduct:SubProduct { name: TOUPPER(line.`Sub-product`) })
MERGE (subProduct)-[:IN_PRODUCT_CATEGORY]->(product)
CREATE (complaint)-[:ABOUT]->(subProduct);
```

Querying the Database

1. Top types of responses that are disputed

```
MATCH (r:Response)-[:TO {disputed:true}]->(:Complaint)
RETURN r.name AS response, COUNT(*) AS count
ORDER BY count DESC;
```

2. Companies with the most disputed responses

```
MATCH (:Response)-[:TO {disputed:true}]->(complaint:Complaint)
MATCH (complaint)-[:AGAINST]->(company:Company)
RETURN company.name AS company, COUNT(*) AS count
ORDER BY count DESC
LIMIT 10;
```

Querying the Database

3. All issues

```
MATCH (i:Issue)
RETURN i.name AS issue
ORDER BY issue;
```

4. All sub-issues within the 'communication tactics' issue

```
MATCH (i:Issue {name:'COMMUNICATION TACTICS'})
MATCH (sub:SubIssue)-[:IN_ISSUE_CATEGORY]->(i)
RETURN sub.name AS subissue
ORDER BY subissue;
```

Querying the Database

5. Top products and sub-products associated with the obscene / abusive language sub-issue

```
MATCH (subIssue:SubIssue {name:'USED OBSCENE/PROFANE/ABUSIVE LANGUAGE'})
MATCH (complaint:Complaint)-[:WITH]->(subIssue)
MATCH (complaint)-[:ABOUT]->(p:Product)
OPTIONAL MATCH (complaint)-[:ABOUT]->(sub:SubProduct)
RETURN p.name AS product, sub.name AS subproduct, COUNT(*) AS count
ORDER BY count DESC;
```

Querying the Database

6. Top company associated with the obscene / abusive language sub-issue

```
MATCH (subIssue:SubIssue {name:'USED OBSCENE/PROFANE/ABUSIVE LANGUAGE'})
MATCH (complaint:Complaint)-[:WITH]->(subIssue)
MATCH (complaint)-[:AGAINST]->(company:Company)
RETURN company.name AS company, COUNT(*) AS count
ORDER BY count DESC
LIMIT 10;
```

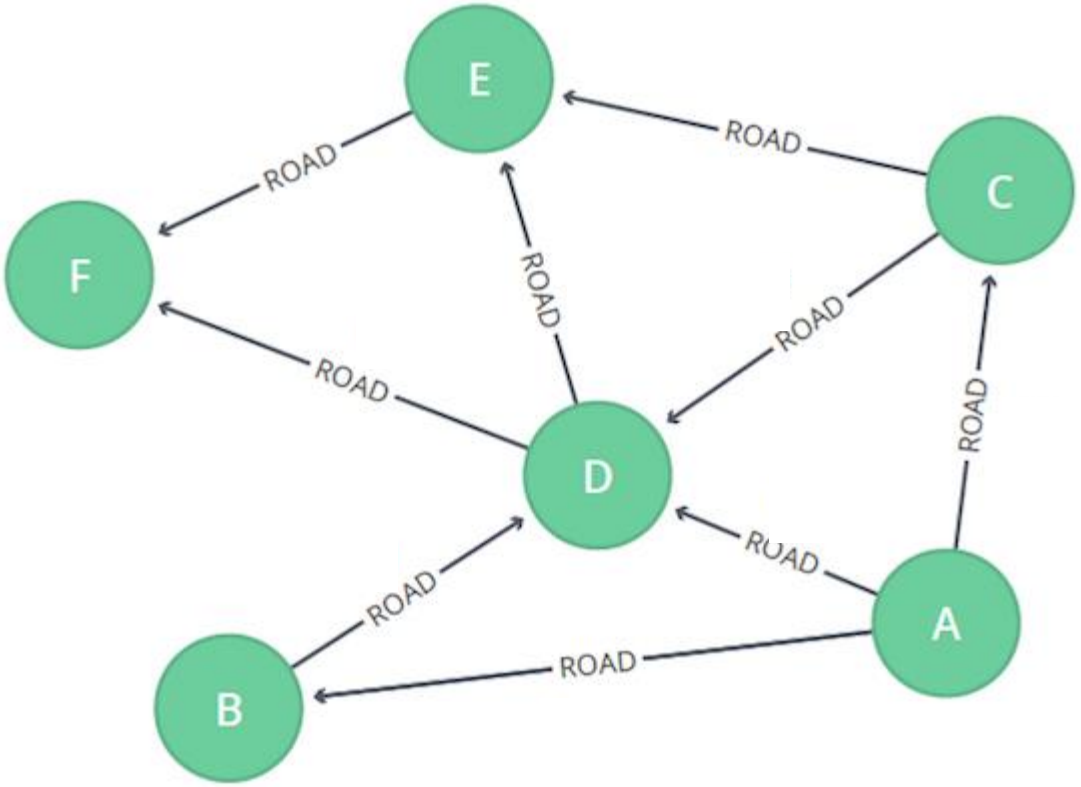
Querying the Database

7. Sub-products that belong to multiple product categories

```
MATCH (sub:SubProduct)-[:IN_PRODUCT_CATEGORY]->(p:Product)
WITH sub, COLLECT(p) AS products
WHERE SIZE(products) > 1
RETURN sub, products;
```

Shortest Paths Examples

Create Graph Unweighted

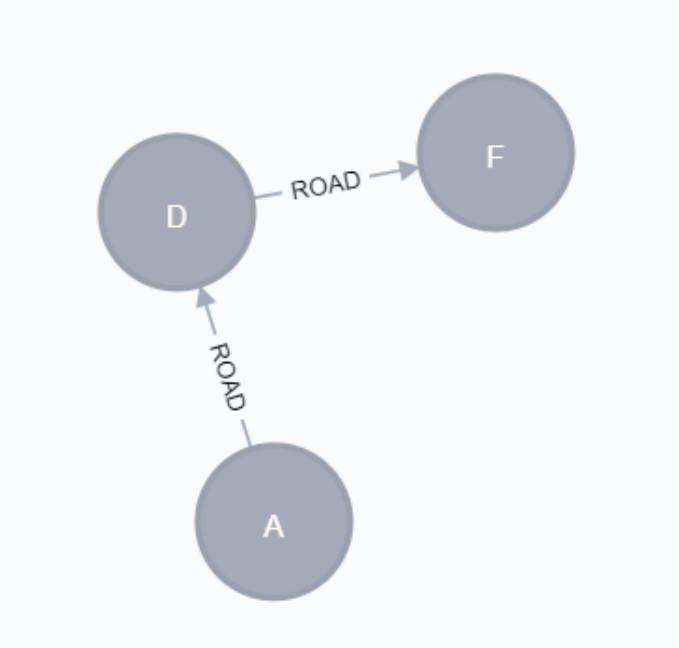



```
MERGE(a:Loc{name:"A"})
MERGE(b:Loc{name:"B"})
MERGE (c:Loc{name:"C"})
MERGE (d:Loc {name:"D"})
MERGE (e:Loc {name:"E"})
MERGE (f:Loc {name:"F"})
MERGE (a)-[:ROAD]->(b)
MERGE (a)-[:ROAD]->(c)
MERGE (a)-[:ROAD]->(d)
MERGE (b)-[:ROAD]->(d)
MERGE (c)-[:ROAD]->(d)
MERGE (c)-[:ROAD]->(e)
MERGE (d)-[:ROAD]->(e)
MERGE (d)-[:ROAD]->(f)
MERGE (e)-[:ROAD]->(f)
```

Shortest Path Unweighted Graphs (BFS)

- The following query calculates **the point to point shortest path** from A to F using BFS (unweighted graph)

```
MATCH (a:Loc{name:'A'}),(f:Loc{name:'F'}),
p = shortestPath((a)-[*]-(f))
RETURN p
```



Shortest Path Unweighted Graphs (BFS)

- The following query calculates the point to point shortest path from C to F and outputs the results

```
MATCH p = shortestPath((c:Loc{name:'C'})-[*]-(f:Loc{name:'F'}))
RETURN [n in nodes(p) | n.name] AS ShortestPath, length(p) as Length
```

ShortestPath	Length
["C", "D", "F"]	2

Shortest Path Unweighted Graphs (BFS)

- The following query finds **all the point to point shortest paths** between node C and F (exist more than 1 shortest path) and outputs the results.

```
MATCH p = allShortestPaths((c:Loc{name:'C'})-[*]-(f:Loc{name:'F'}))
RETURN [n in nodes(p) | n.name] AS AllSortestPaths, length(p) as Length
```

"AllSortestPaths"	"Length"
["C","D","F"]	2
["C","E","F"]	2

Shortest Path Unweighted Graphs (BFS)

- The following query finds **all single source shortest paths** between node A and all other nodes of the graph.

```
MATCH (f:Loc), p = allShortestPaths((c:Loc{name:'A'})-[*]-(f:Loc)
Where f<>c
RETURN c.name as fromNode,
f.name as toNode,[n in nodes(p) | n.name] AS AllSortestPaths,
length(p) as Length
order by c.name
```

"fromNode"	"toNode"	"AllSortestPaths"	"Length"
"A"	"B"	["A","B"]	1
"A"	"C"	["A","C"]	1
"A"	"D"	["A","D"]	1
"A"	"E"	["A","C","E"]	2
"A"	"E"	["A","D","E"]	2
"A"	"F"	["A","D","F"]	2

Shortest Path Unweighted Graphs (BFS)

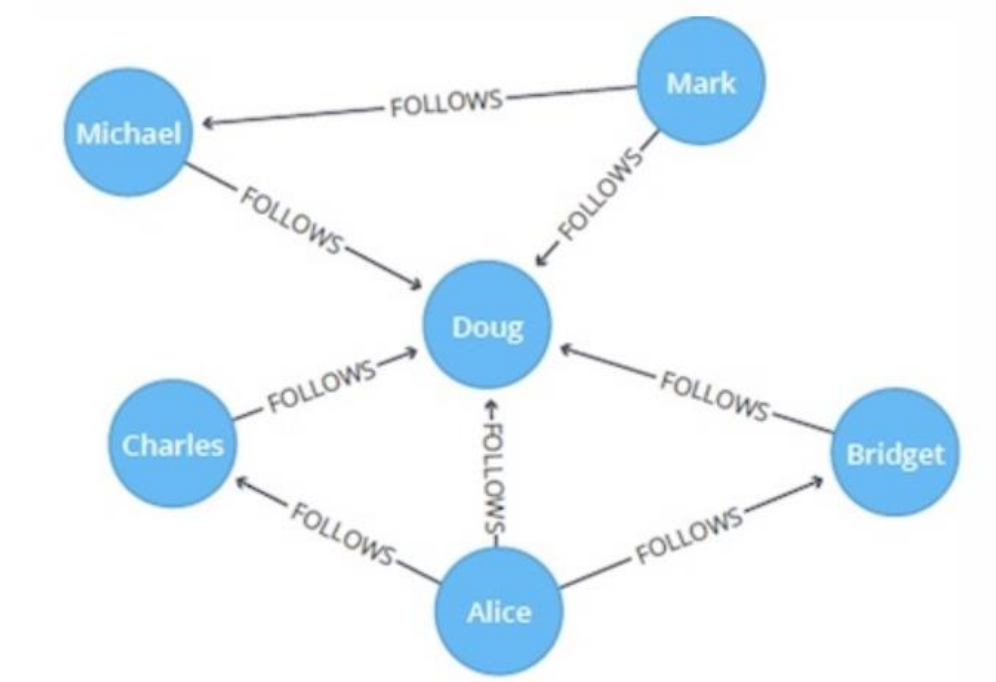
- The following query finds **all pair shortest paths** between all nodes of the graph.

```
MATCH (f:Loc),(c:Loc), p = allShortestPaths((c:Loc)-[*]-(f:Loc))
Where f<>c
RETURN c.name as fromNode,
f.name as toNode,[n in nodes(p) | n.name] AS AllSortestPaths,
length(p) as Length
order by c.name
```

"fromNode"	"toNode"	"AllSortestPaths"	"Length"
"A"	"B"	["A","B"]	1
"A"	"C"	["A","C"]	1
"A"	"D"	["A","D"]	1
"A"	"E"	["A","C","E"]	2
"A"	"E"	["A","D","E"]	2
"A"	"F"	["A","D","F"]	2
"B"	"A"	["B","A"]	1
"B"	"C"	["B","A","C"]	2
"B"	"C"	["B","D","C"]	2
"B"	"D"	["B","D"]	1
"B"	"E"	["B","D","E"]	2

Centrality Metrics Examples

Create Graph




```
CREATE (alice:User {name: 'Alice'}),
       (bridget:User {name: 'Bridget'}),
       (charles:User {name: 'Charles'}),
       (doug:User {name: 'Doug'}),
       (mark:User {name: 'Mark'}),
       (michael:User {name: 'Michael'}),
       (alice)-[:FOLLOWS]->(doug),
       (alice)-[:FOLLOWS]->(bridget),
       (alice)-[:FOLLOWS]->(charles),
       (mark)-[:FOLLOWS]->(doug),
       (mark)-[:FOLLOWS]->(michael),
       (bridget)-[:FOLLOWS]->(doug),
       (charles)-[:FOLLOWS]->(doug),
       (michael)-[:FOLLOWS]->(doug)
```

Degree Centrality Directed Graphs

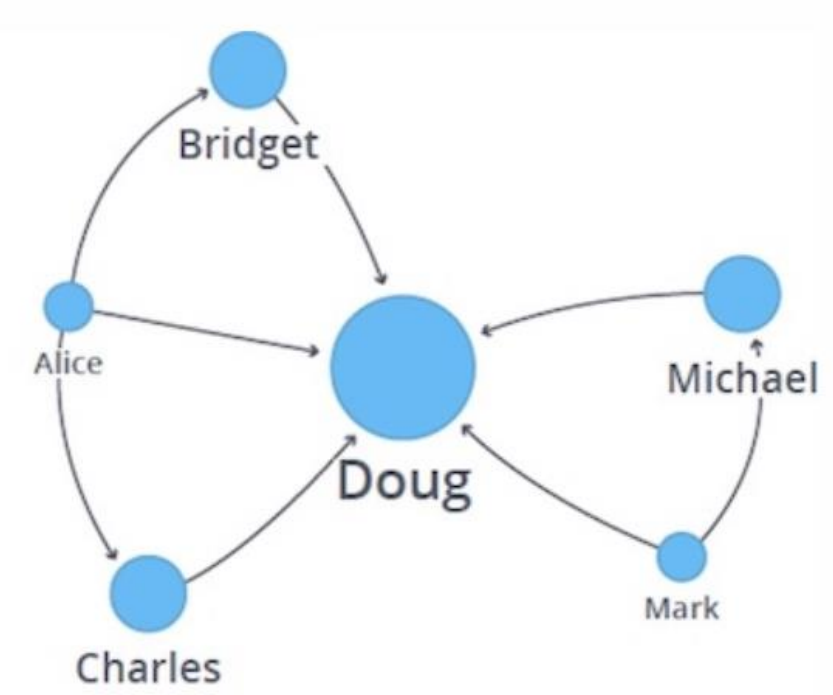
- The following query calculates the number of people that each user follows and is followed by (in-out degree)

```
MATCH (u:User)
OPTIONAL MATCH (u)-[:FOLLOWS]->(f:User)
OPTIONAL MATCH (u)<-[:FOLLOWS]-(follower:User)
RETURN u.name AS name,
    COUNT(DISTINCT f) AS follows,
    COUNT(DISTINCT follower) AS followers
```

name	follows	followers
"Alice"	3	0
"Bridget"	1	1
"Charles"	1	1
"Doug"	0	5
"Mark"	2	0
"Michael"	1	1

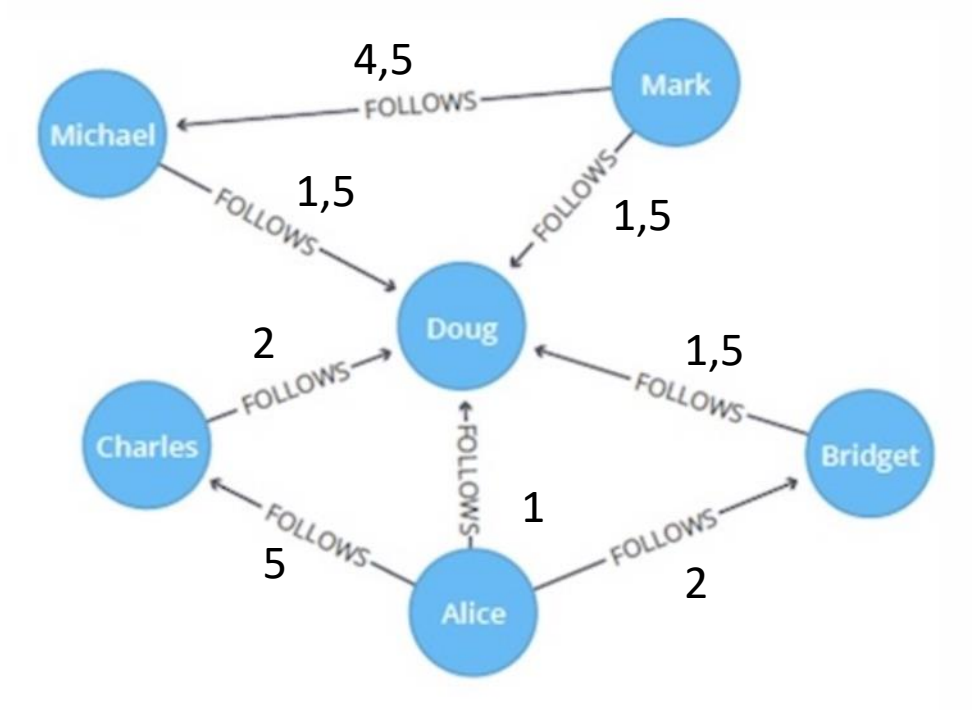
Degree Centrality Directed Graphs

- Doug is the most popular user (in-degree)
- All other users follow Doug but he doesn't follow anybody back
- In real social networks celebrities have high follower counts but tend to follow few people



Degree Centrality Weighted Graphs

- This algorithm is a variant of the Degree Centrality algorithm, that measures the sum of the weights of incoming and outgoing relationships




```
CREATE (alice:User {name:'Alice'}),
       (bridget:User {name:'Bridget'}),
       (charles:User {name:'Charles'}),
       (doug:User {name:'Doug'}),
       (mark:User {name:'Mark'}),
       (michael:User {name:'Michael'}),
       (alice)-[:FOLLOWS {score: 1}]->(doug),
       (alice)-[:FOLLOWS {score: 2}]->(bridget),
       (alice)-[:FOLLOWS {score: 5}]->(charles),
       (mark)-[:FOLLOWS {score: 1.5}]->(doug),
       (mark)-[:FOLLOWS {score: 4.5}]->(michael),
       (bridget)-[:FOLLOWS {score: 1.5}]->(doug),
       (charles)-[:FOLLOWS {score: 2}]->(doug),
       (michael)-[:FOLLOWS {score: 1.5}]->(doug)
```

Degree Centrality Weighted Graphs

- The following will run the algorithm and stream results, showing which users have the most weighted followers (in degree):

```
CALL gds.graph.project(
  'userGraph',
  'User',
  {
    FOLLOWS: {
      properties: 'score'
    }
  }
);

CALL gds.degree.stream('userGraph', {relationshipWeightProperty: 'score', orientation: 'REVERSE'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS weightedFollowers
ORDER BY score DESC
```

user	weightedFollowers
"Doug"	7.5
"Charles"	5.0
"Michael"	4.5
"Bridget"	2.0
"Alice"	0.0
"Mark"	0.0

Degree Centrality Weighted Graphs

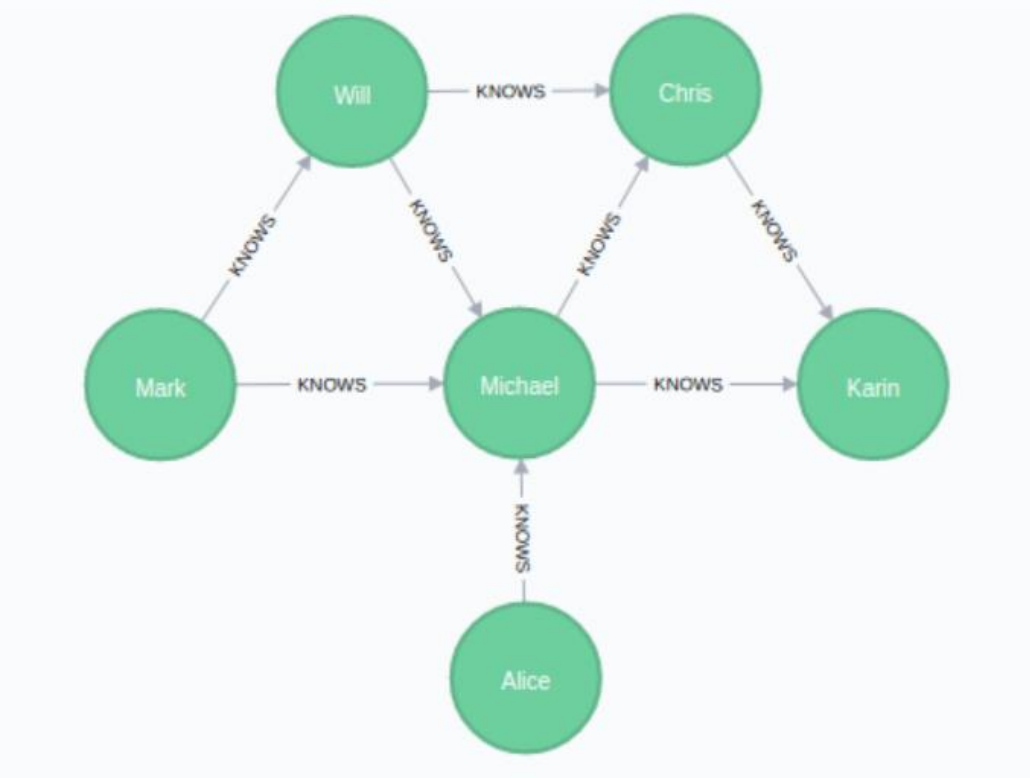
- The following will run the algorithm and stream results, showing which users have the most weighted follows (out degree):

```
CALL gds.graph.project(
  'userGraph',
  'User',
  {
    FOLLOWS: {
      properties: 'score'
    }
  }
);

CALL gds.degree.stream('userGraph', {relationshipWeightProperty: 'score', orientation: 'NATURAL'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS weightedFollows
ORDER BY score DESC
```

user	weightedFollows
"Alice"	8.0
"Mark"	6.0
"Charles"	2.0
"Bridget"	1.5
"Michael"	1.5
"Doug"	0.0

Closeness Centrality




```
CREATE
  (alice:Person {name: 'Alice'}),
  (michael:Person {name: 'Michael'}),
  (karin:Person {name: 'Karin'}),
  (chris:Person {name: 'Chris'}),
  (will:Person {name: 'Will'}),
  (mark:Person {name: 'Mark'}),
  (michael)-[:KNOWS]->(karin),
  (michael)-[:KNOWS]->(chris),
  (will)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(will),
  (alice)-[:KNOWS]->(michael),
  (will)-[:KNOWS]->(chris),
  (chris)-[:KNOWS]->(karin)
```

Closeness Centrality

- The following will run closeness centrality for each node(treat edges as undirected)

```
CALL gds.graph.project(
   'personGraph',
   'Person',
   {
      KNOWS: {
         orientation: 'UNDIRECTED'
      }
   }
)

CALL gds.closeness.stream('personGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS centrality
ORDER BY score DESC;
```

"user"	"centrality"
"Michael"	1.0
"Chris"	0.7142857142857143
"Will"	0.7142857142857143
"Karin"	0.625
"Mark"	0.625
"Alice"	0.5555555555555556

Betweenness Centrality

- The following will run betweenness centrality for each node(directed edges)

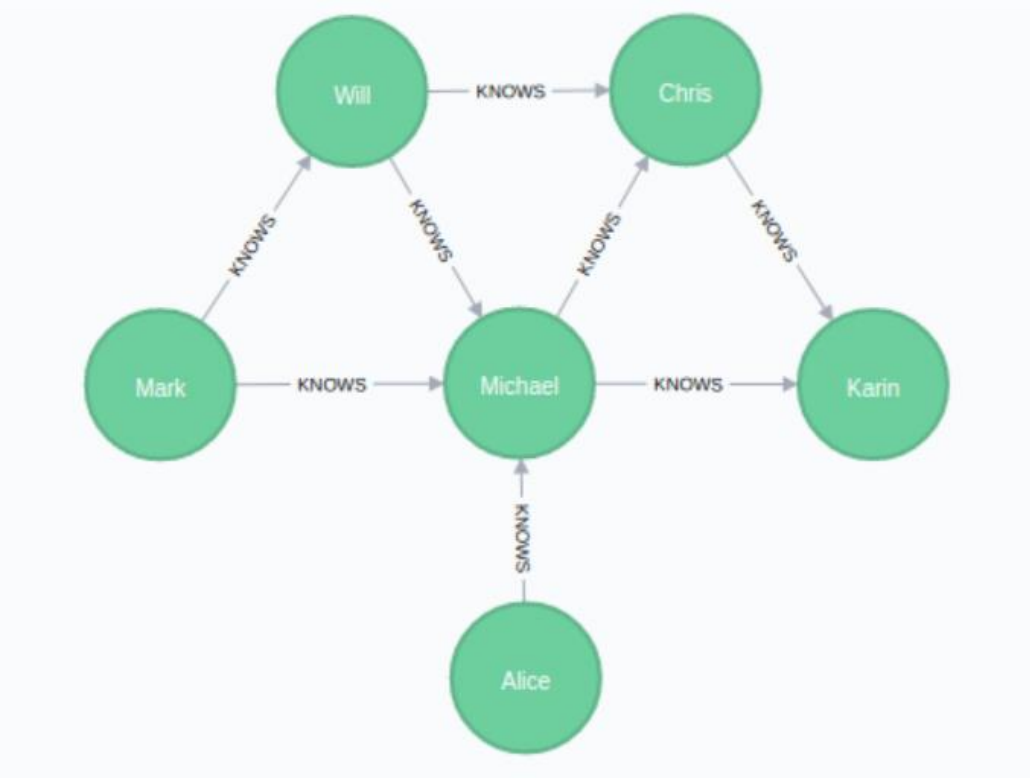
```
CALL gds.graph.project(
   'personGraph1',
   'Person',
   {
      KNOWS: { orientation: 'NATURAL' }

    }
)

CALL gds.betweenness.stream('personGraph1')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS  centrality
ORDER BY centrality DESC;
```

"user"	"centrality"
"Michael"	4.0
"Chris"	0.5
"Will"	0.5
"Alice"	0.0
"Karin"	0.0
"Mark"	0.0

Local Clustering Coefficient




```
CREATE
  (alice:Person {name: 'Alice'}),
  (michael:Person {name: 'Michael'}),
  (karin:Person {name: 'Karin'}),
  (chris:Person {name: 'Chris'}),
  (will:Person {name: 'Will'}),
  (mark:Person {name: 'Mark'}),
  (michael)-[:KNOWS]->(karin),
  (michael)-[:KNOWS]->(chris),
  (will)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(michael),
  (mark)-[:KNOWS]->(will),
  (alice)-[:KNOWS]->(michael),
  (will)-[:KNOWS]->(chris),
  (chris)-[:KNOWS]->(karin)
```

Local Clustering Coefficient

- The following statement will project the graph to undirected and store it in the graph catalog under the name 'myGraph‘
- Neo4j computes local clustering coefficient only for undirected graphs

```
CALL gds.graph.project(
  'myGraph',
  'Person',
  {
    KNOWS: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

Local Clustering Coefficient

- The following will run the local clustering coefficient for each node

```
CALL gds.localClusteringCoefficient.stream('myGraph')
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).name AS name,
localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

name	localClusteringCoefficient
"Karin"	1.0
"Mark"	1.0
"Chris"	0.6666666666666666
"Will"	0.6666666666666666
"Michael"	0.3
"Alice"	0.0

Global Clustering Coefficient

- The following will calculate the global clustering coefficient of the graph

CALL gds.localClusteringCoefficient.stats('myGraph')
YIELD averageClusteringCoefficient, nodeCount

averageClusteringCoefficient	nodeCount
0.6055555555555555	6