

# Graph Data Management

---

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS DEPARTMENT OF INFORMATICS

# Outline

---

- Graph Databases
- What is Neo4j
- Neo4j Property Graph Model
- Cypher Query Language
- Complaint Database example
- Centrality Metrics Examples

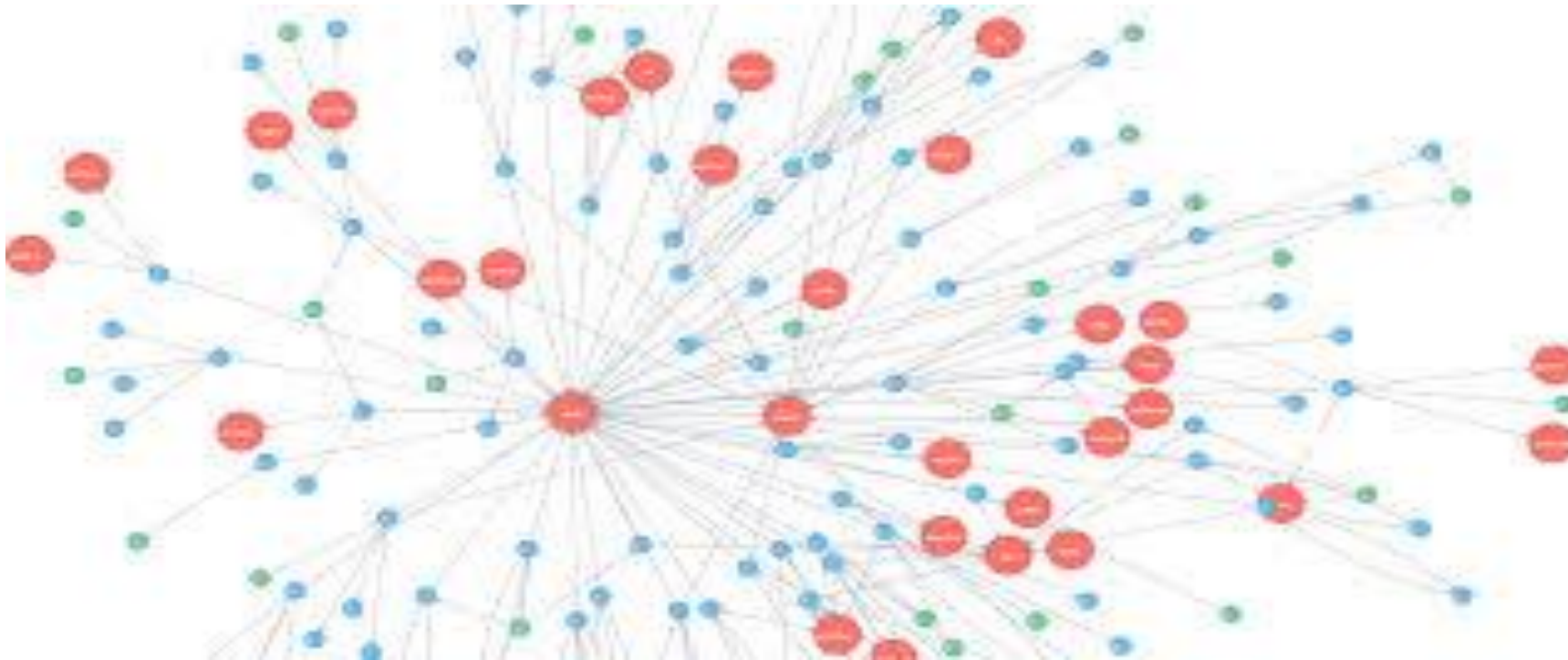
# Graph Databases

---

# What is a Graph

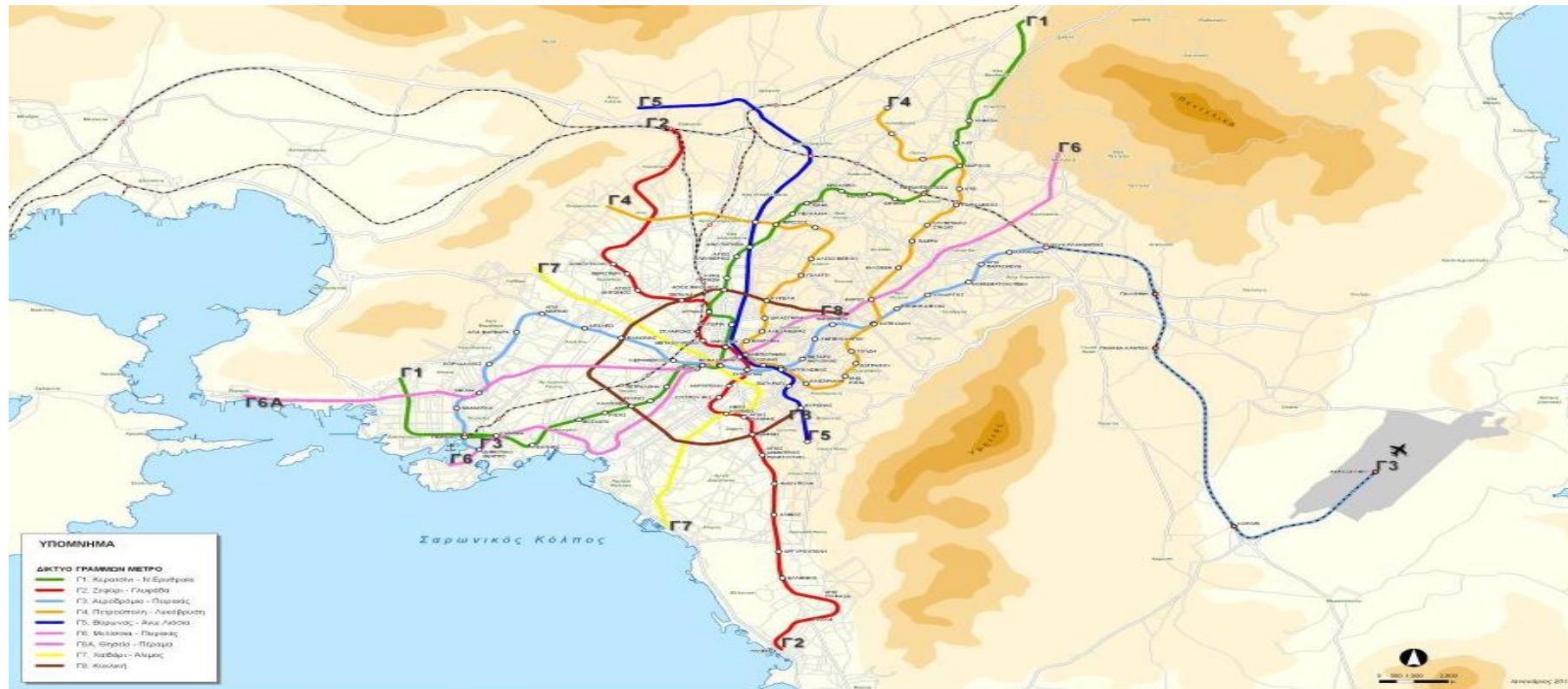
---

A graph is **connected** data



# What is a Graph

## Transport Networks



# What is a Graph

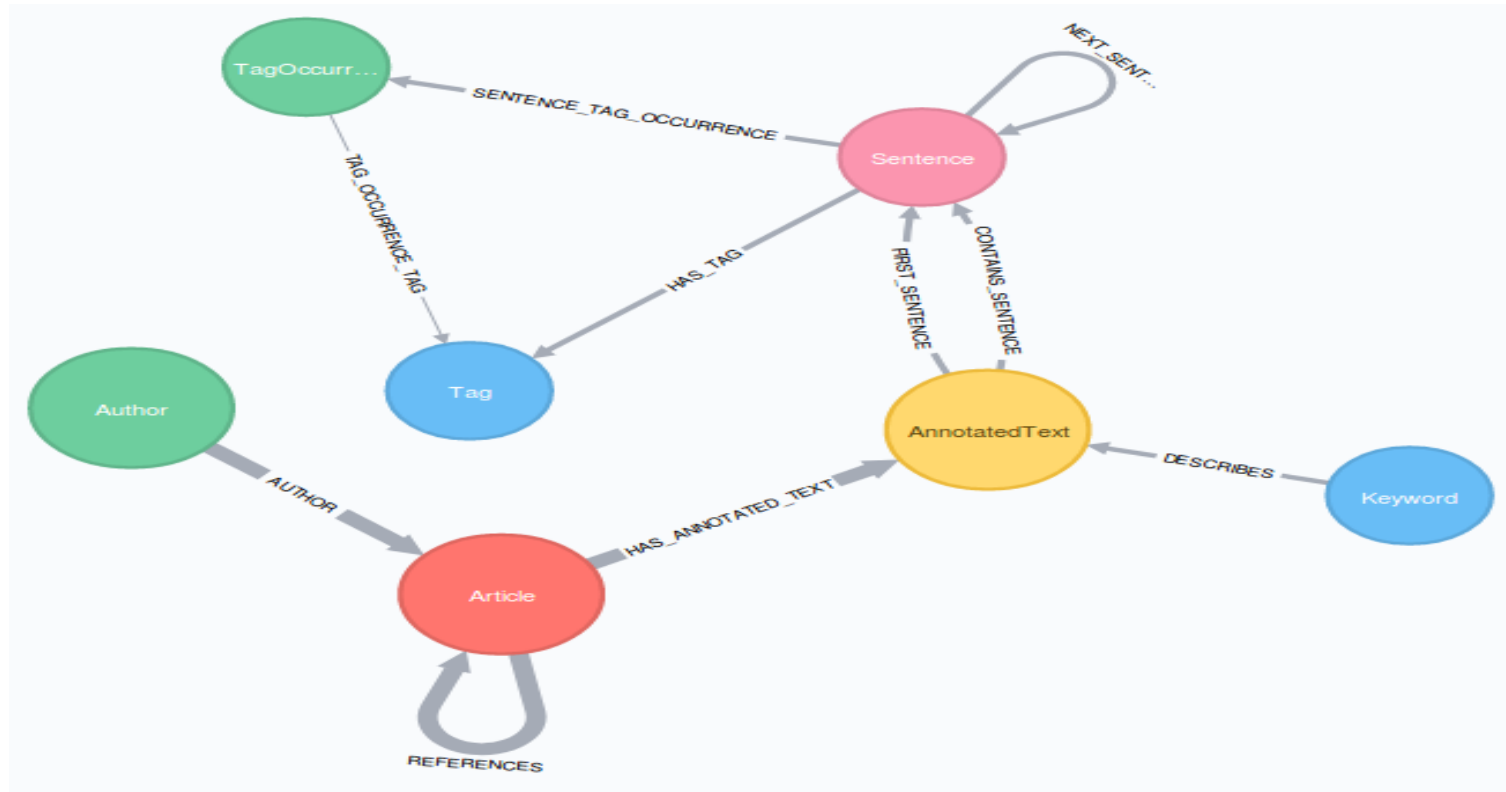
---

## Social Networks



# What is a Graph

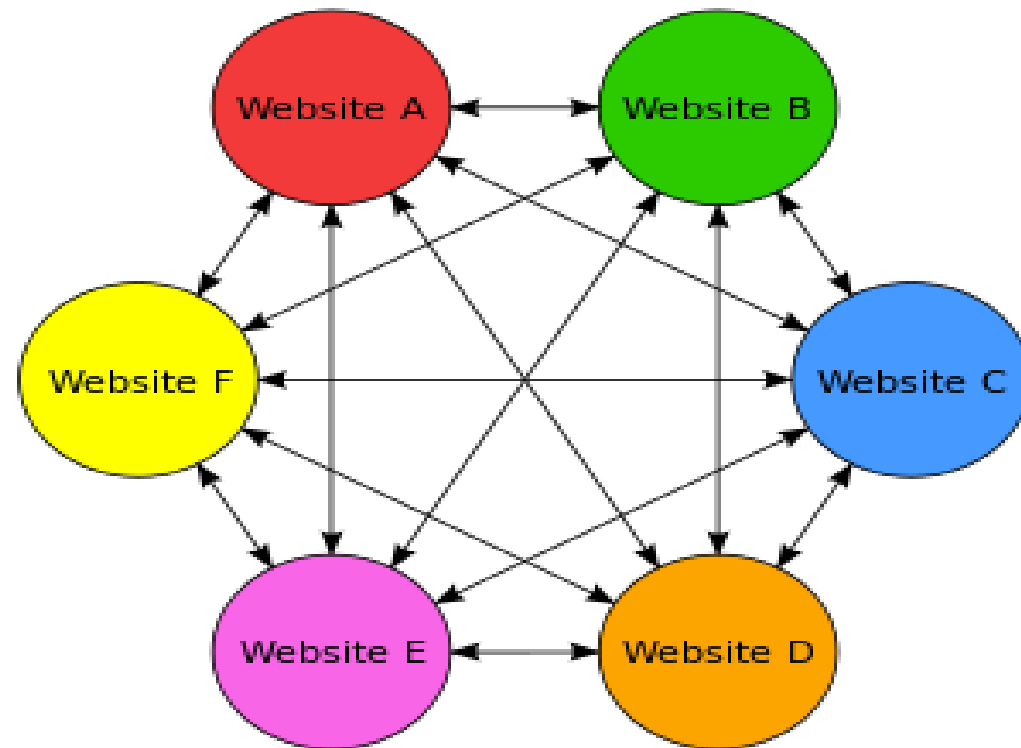
## Citation Networks



# What is a Graph

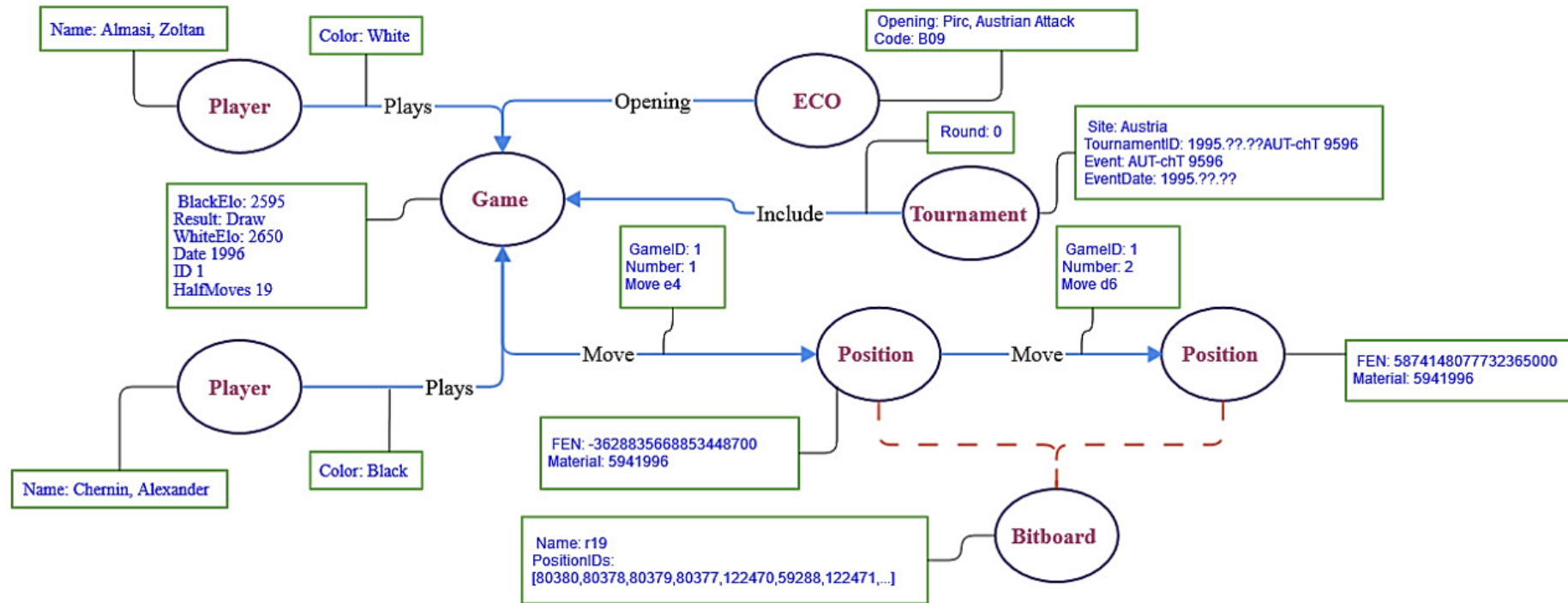
---

## Web Graph



# What is a Graph

## Chess Game Graph



# What is a Graph

---

- Data structures that model **structural** relationships among objects.
- Widely used in application domains for which identifying and exploring relationship patterns, rules, and anomalies is useful.

Today we see graph-projects in virtually every industry



Finance



Social networks



HR &  
Recruiting



Manufacturing  
& Logistics



Health Care



Telco

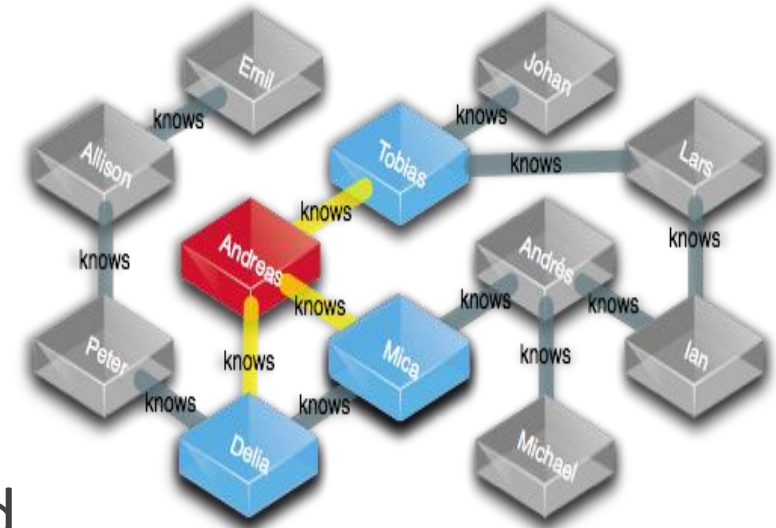


Retail

# Graph Databases

---

- Data Model:
  - Nodes with properties
  - Named Relationships with properties
- Manage:
  - Highly connected data
  - Efficiently explore a node's neighborhood
- Examples:
  - Neo4J, InfiniteGraph, OrientDB, AllegroGraph

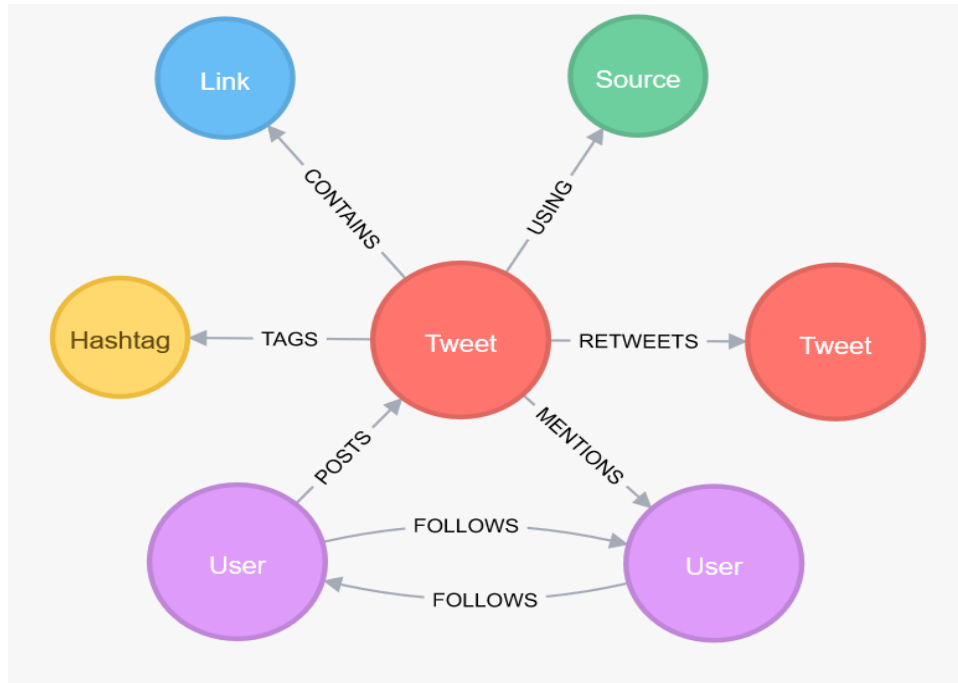


# Graph Database Use cases

---

## Social Media and Social Network Graphs

- Leverage social connections or infer relationships based on activity



### Queries:

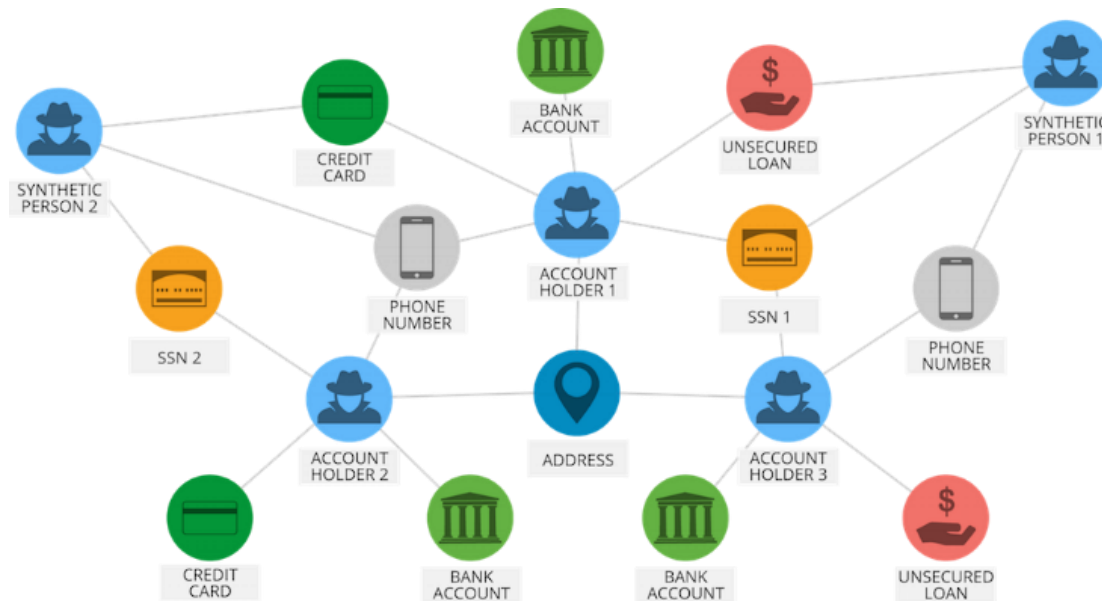
Community Cluster Analysis  
Friend-of-Friend Recommendations  
Influencer Analysis  
Sharing & Collaboration  
Social Recommendations

# Graph Database Use cases

---

## Fraud Detection

- Real-time analysis of data relationships to uncovering fraud rings and scams



### Queries:

Anti Money Laundering (AML)

Ecommerce Fraud

First-Party Bank Fraud

Insurance Fraud

Link Analysis

# Graph Database Use cases

## Knowledge Graph

- Graph-based search tools for better digital asset management



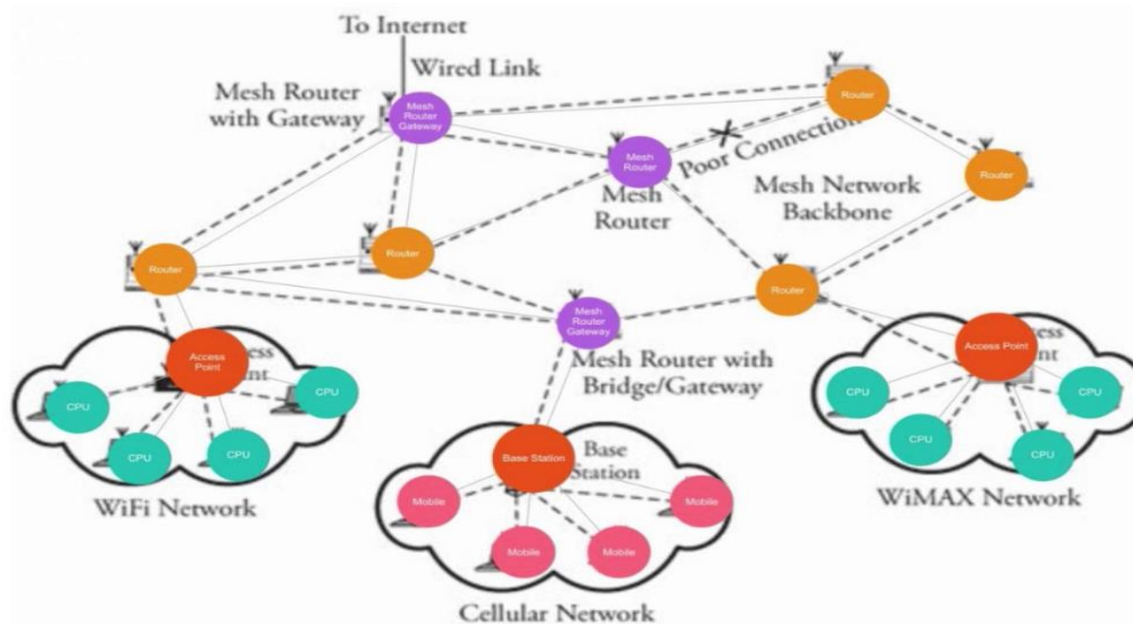
### Queries:

Asset Management  
Cataloging  
Content Management  
Inventory  
Work Flow Processes

# Graph Database Use cases

## Network and Database Monitoring

- Graph databases are more suitable for making sense of complex interdependencies central to managing networks and IT infrastructure



### Queries:

Asset Management

Cybersecurity

Impact Analysis

Quality-of-Service Mapping

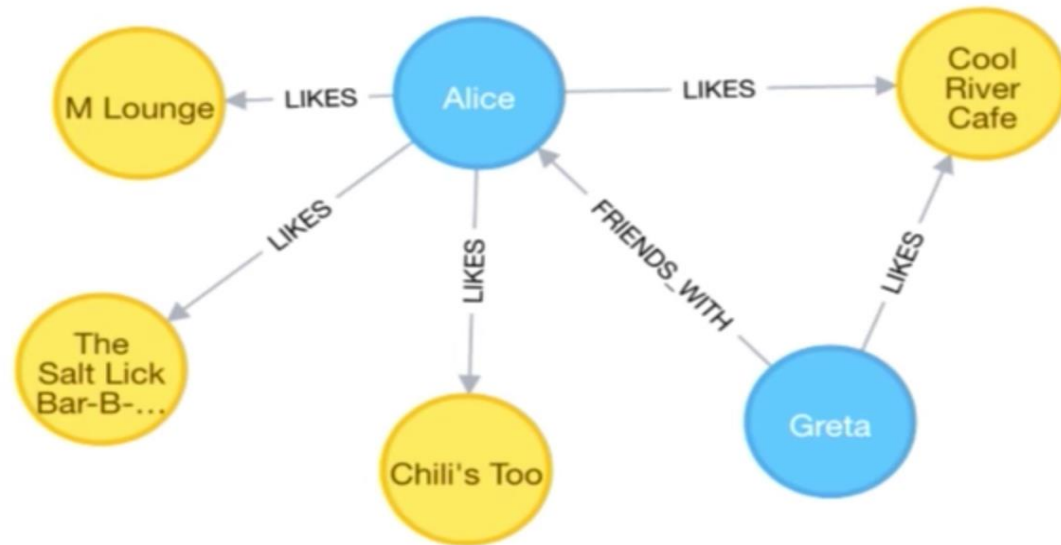
Root Cause Analysis

# Graph Database Use cases

---

## Recommendation Engines

- Graph-powered recommendation engines help companies personalize products, content and services by leveraging a multitude of connections in real time

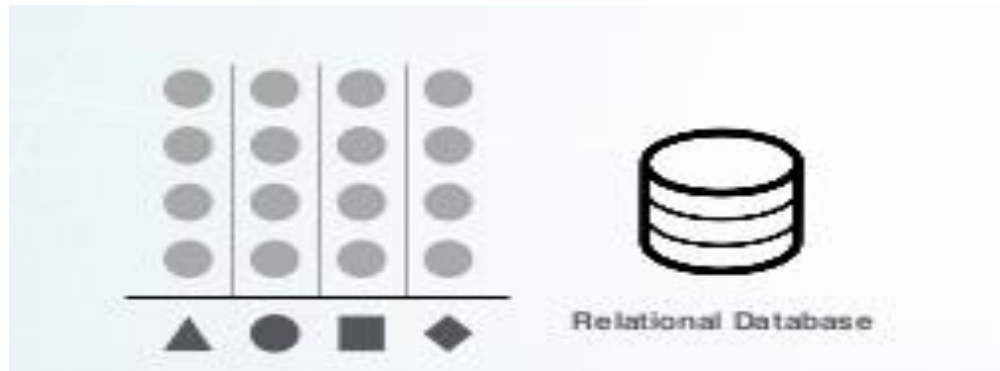


### Queries:

Content & Media Recommendations  
Graph-Aided Search Engine  
Product Recommendations  
Professional Networks  
Social Recommendations

# Why Graph Databases?

---



Good for:

- Well understood data structure that don't change frequently
- Known problems involving discrete parts of the data, or minimal connectivity



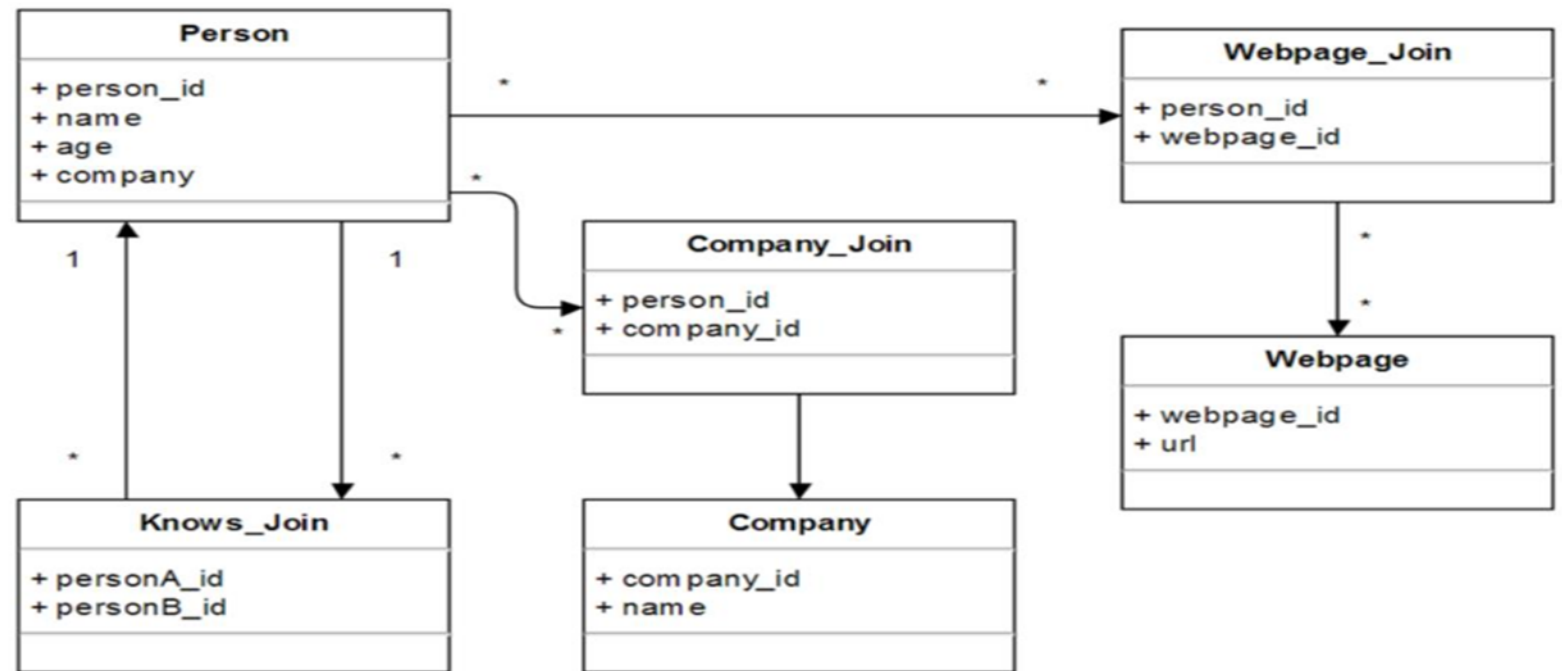
Good for:

- Dynamic systems where the data topology is difficult to predict
- Dynamic requirements: evolving data
- Problems where the relationships in data contribute meaning & value

# Why Graph Databases?

## Schema Flexibility

- Relationships – join tables
- Join tables represent edges
- A lot of table joins (join bomb)-reduced query performance

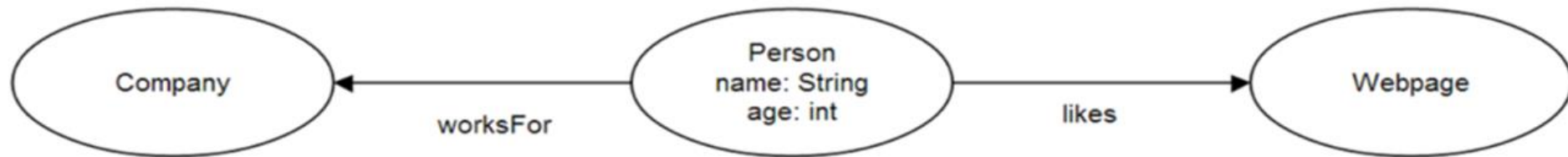


# Why Graph Databases?

---

## Schema Flexibility

- Named Nodes and Relationships
- Instead of table joins - traversals
- Can add any kind of nodes and relations without schema change
- Can add any number of different relationships between nodes (multigraph)



# Why Graph Databases?

---

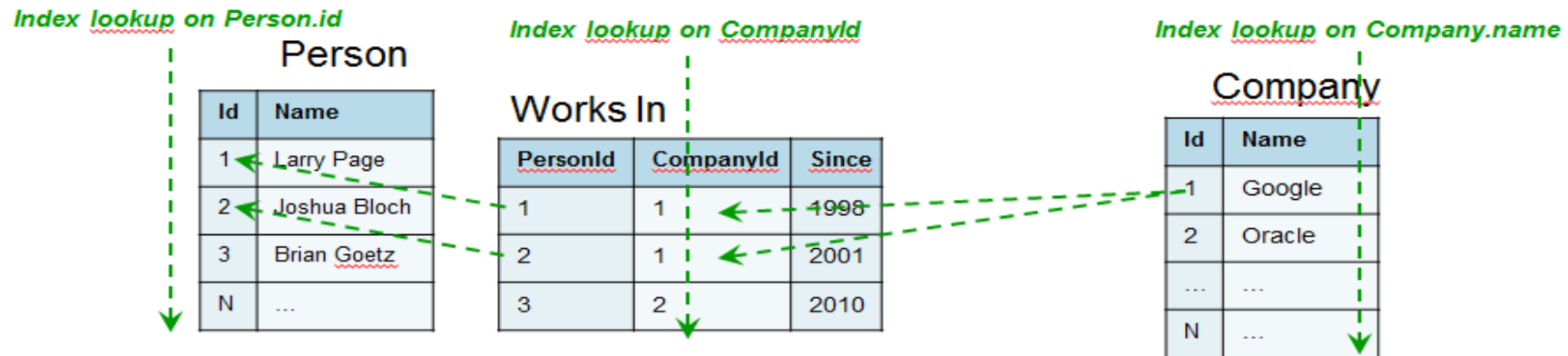
## **Whiteboard friendly**

- Easily describe the domain with nodes and relationships
- Consider if the domain is appropriate for graph representation:
  - Draw the domain on a whiteboard
  - If your domain entities have relationships to other entities
  - If your queries rely on exploring relationships
  - Graph Database is a great fit

# Why Graph Databases?

## Express Queries as Traversals

- We store: companies, employers and employment period
- Query: find all people that work at Google
- 3 index lookups in relational DB

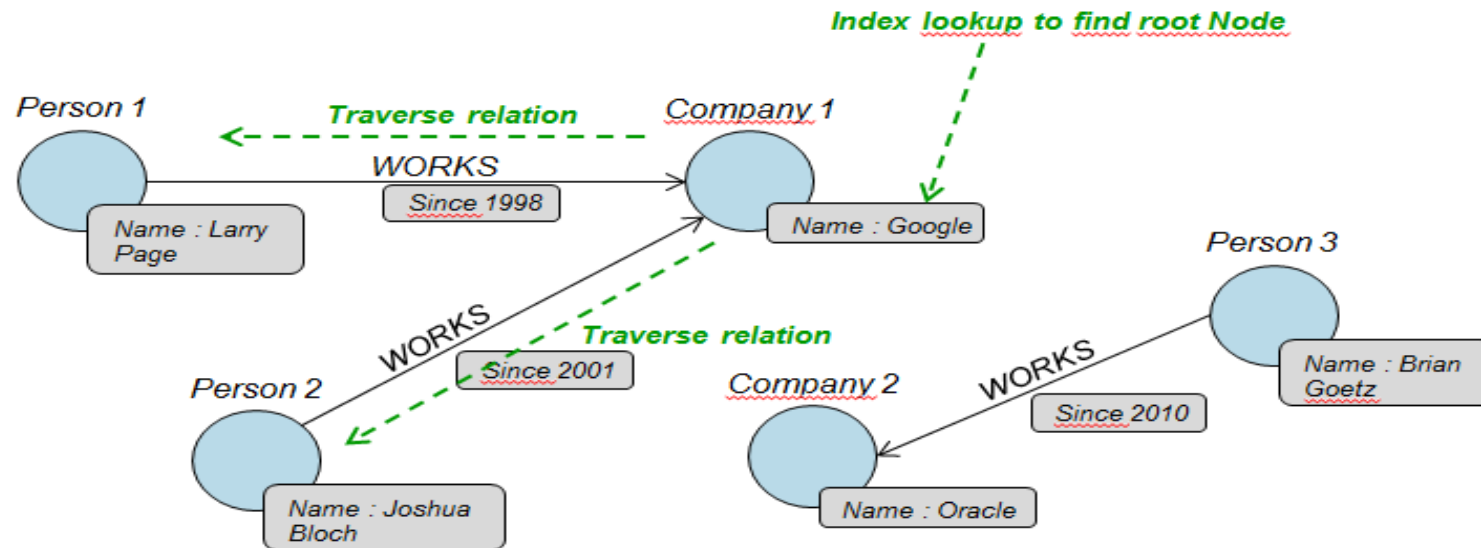


Select Person Name  
from Person, Company, WorksIn  
where Company.name='Google'  
and WorksIn.CompanyId = Company.Id  
and WorksIn.PersonId = Person.Id

# Why Graph Databases?

## Express Queries as Traversals

- We store: companies, employers and employment period
- Query: find all people that work at Google
- 1 index lookup, traverse relationships

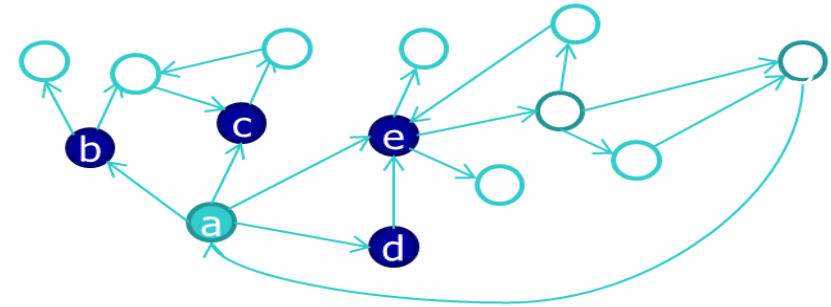


# Why Graph Databases?

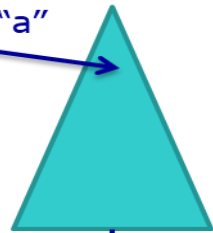
Very natural to express graph related problems with traversals

- Find shortest path, centrality, node degree...

Find the friends of “John”  
(node a) traversal  
Easy: index scan



PersonID="a"



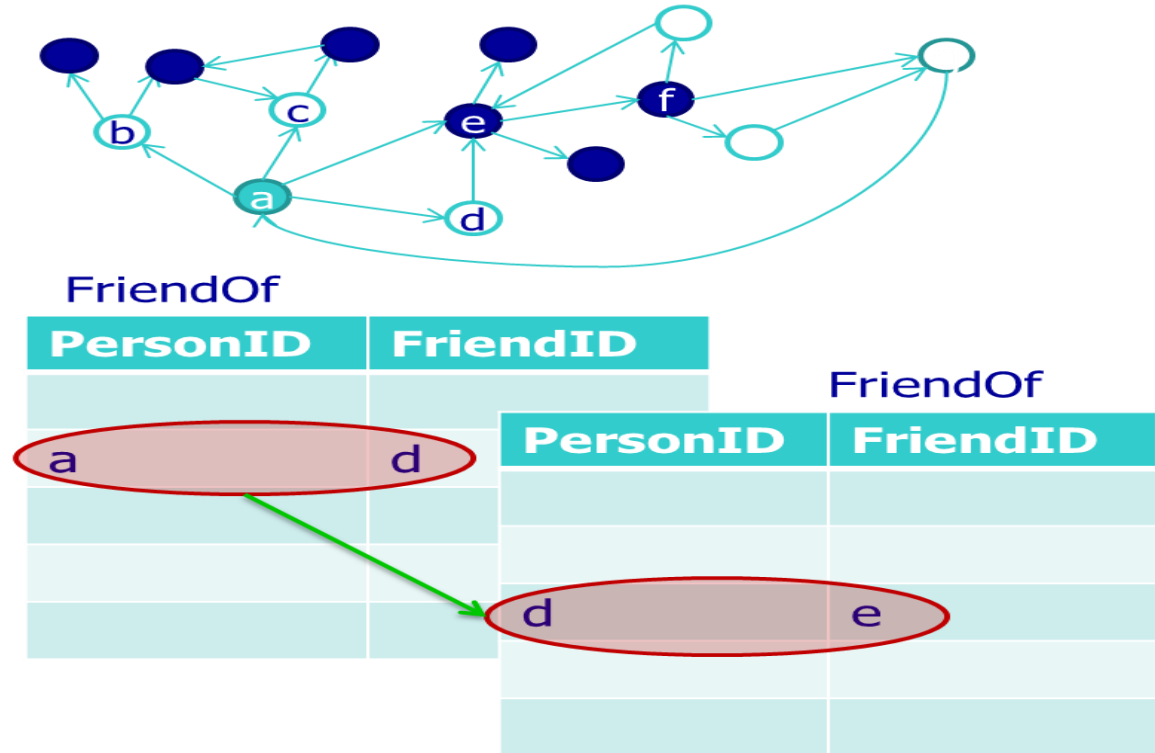
FriendOf

PersonID	FriendID
a	b
a	c
a	e
..	..
e	f

# Why Graph Databases?

Very natural to express graph related problems with traversals

- Find the friends-of-friends of John
- Harder to compute: self-join
- How do we find the k-hop neighbors of John?

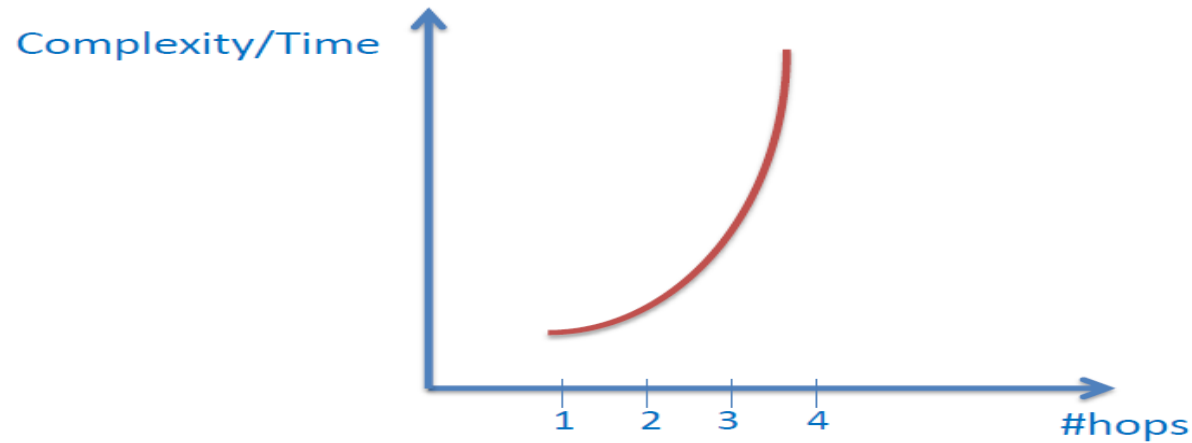


# Why Graph Databases?

---

Very natural to express graph related problems with traversals

## Performance of RDBMs on path queries



# When are Graph Databases NOT a Good Fit

---

- Where data is disconnected and relationships do not matter
- Where data model stay consistent and data structure is fixed and tabular
- Where queries execute bulk data scans or do not start from a known data point
- Where you will use it as a key-value store
- Where large amounts of text need to be stored as properties

# Neo4j Graph Database

---

# What is Neo4j?

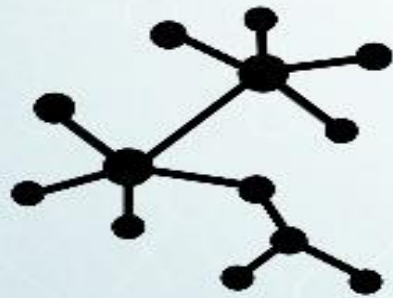
---

- Open source NoSQL graph database
- Implemented in Java and Scala
- Most popular Graph Database
- Implements the Property Graph Model down to the storage model
  - constant time traversal for relationships
- ACID transaction compliance
  - atomicity, consistency, isolation, durability
  - guarantee: database transactions are processed reliably

# Neo4j Usage

---

How do you use Neo4j?



CREATE MODEL



LOAD DATA

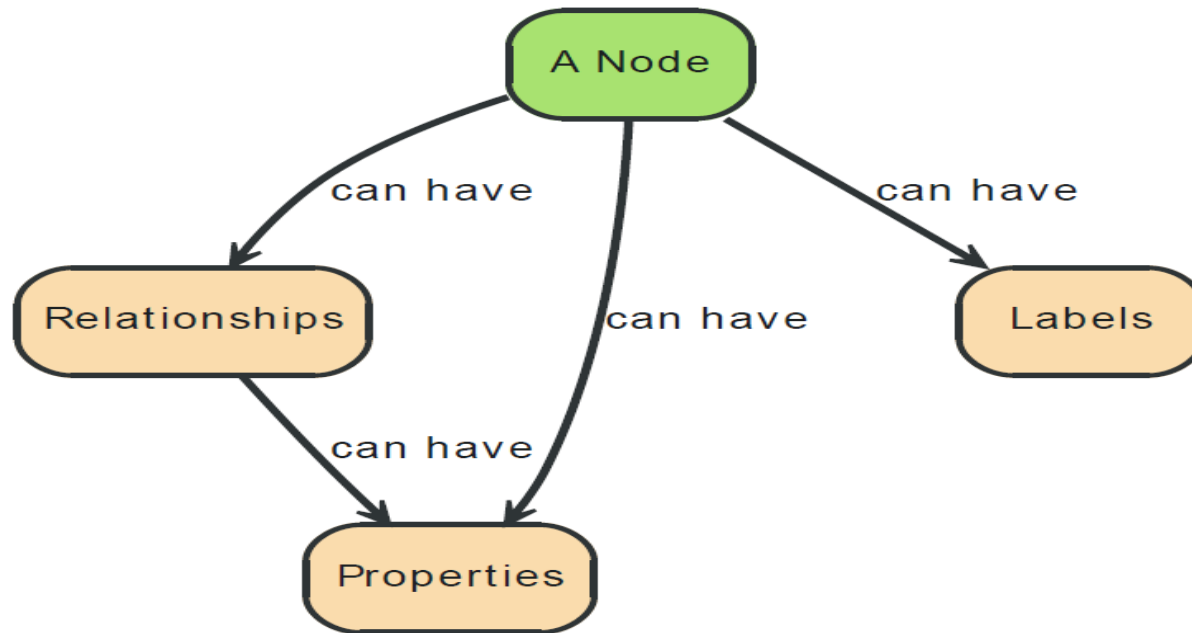


QUERY DATA

# Neo4j Property Graph Model

---


**Nodes:** can have properties and labels



# Creating Nodes

---

CREATE (john:Person {name: 'John', age: 35})



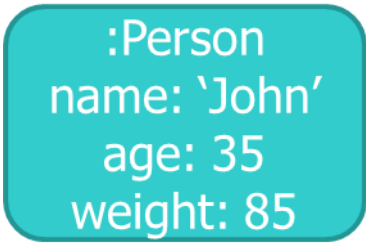
:Person  
name: 'John'  
age: 35

- General syntax CREATE (n:Label<sub>1</sub>:...:Label<sub>n</sub> { attr<sub>1</sub>:val<sub>1</sub>, attr<sub>2</sub>:val<sub>2</sub>, ...attr<sub>k</sub>:val<sub>k</sub> })
  - n is a variable that you can use to refer to that node in the same script

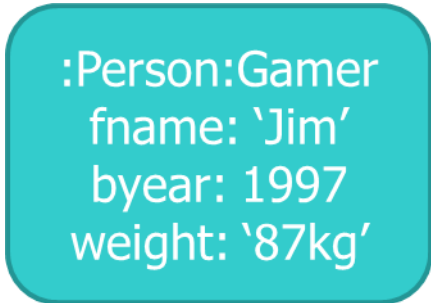
# Creating Nodes

---

- Unlike relational databases, there is no restriction on the number and type of properties on a node
  - E.g. nodes may have different properties, or same properties of different types
  - Recall **Person** is just a label. It does not restrict the schema of the corresponding nodes



```
:Person
name: 'John'
age: 35
weight: 85
```

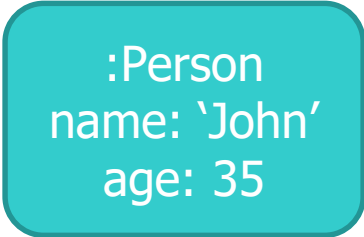


```
:Person:Gamer
fname: 'Jim'
byear: 1997
weight: '87kg'
```

# Creating Nodes

---

- Assert that each Person has a name (Existential constraints)



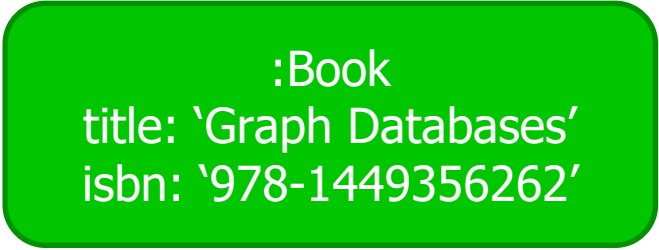
```
:Person  
name: 'John'  
age: 35
```

- `CREATE CONSTRAINT ON (person:Person) ASSERT exists(person.name)`

# Creating Nodes

---

- Assert that no two books in the database can have the same isbn (Unique constraints)



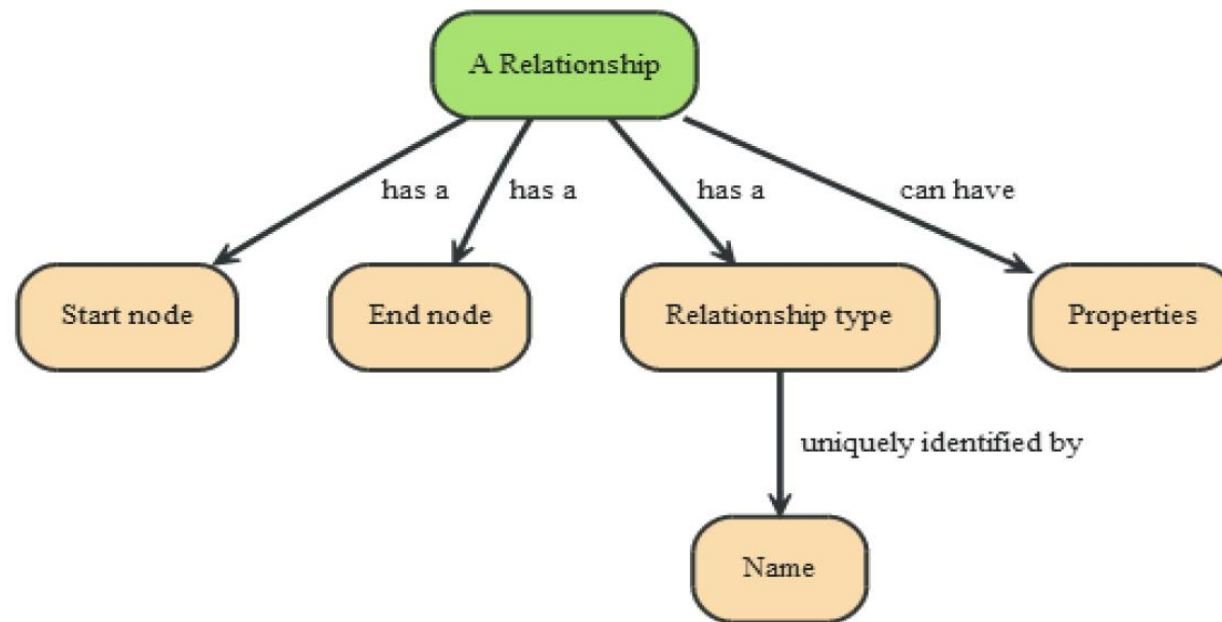
```
:Book  
title: 'Graph Databases'  
isbn: '978-1449356262'
```

- CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE

# Neo4j Property Graph Model

---

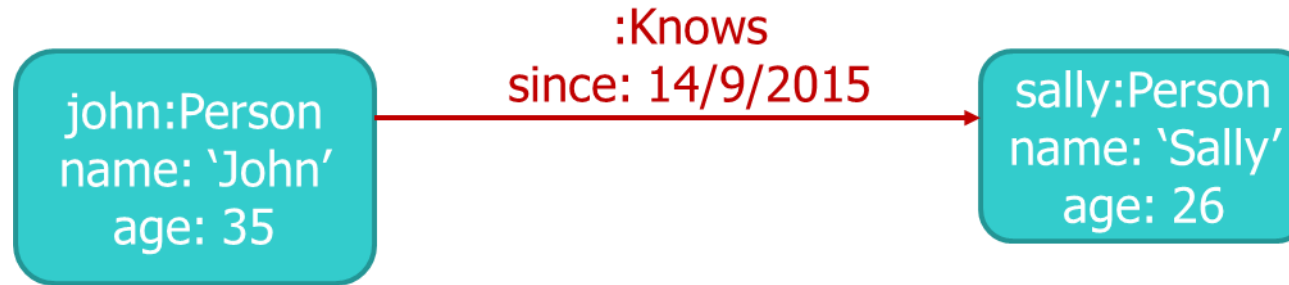
**Relationships:** connect two nodes, have direction, have properties, have relationship type



# Create Relationships

---

- CREATE (john)-[:Knows {since: '14/9/2015'}]->(sally)

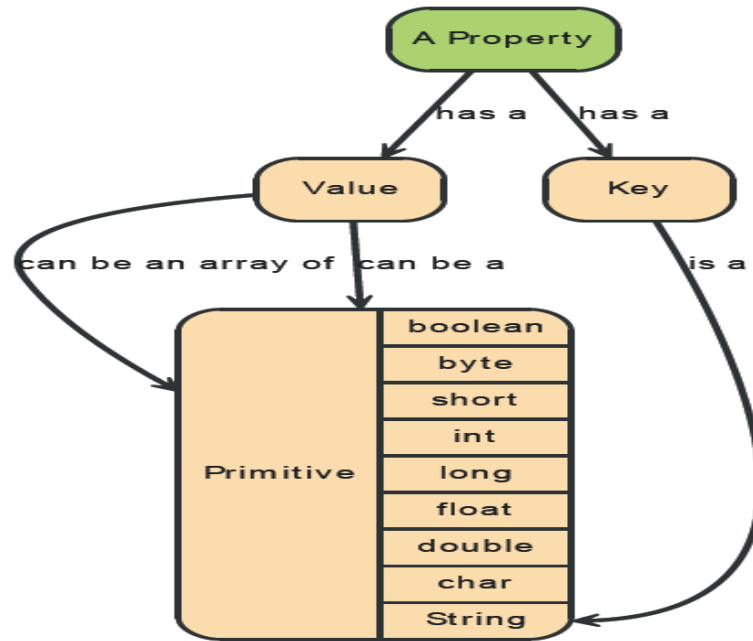


- In this example **Knows** is a relationship type, **since** is an attribute for that particular instance, **john** & **sally** are variables that refer to previously created nodes

# Neo4j Property Graph Model

---

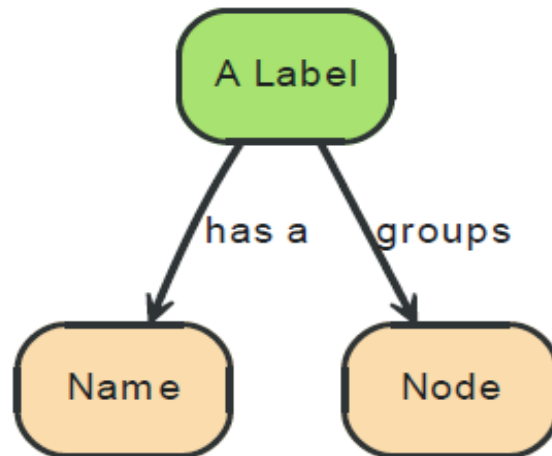
**Properties:** key-value pairs, key is a String, values can be primitives or an array of primitives



# Neo4j Property Graph Model

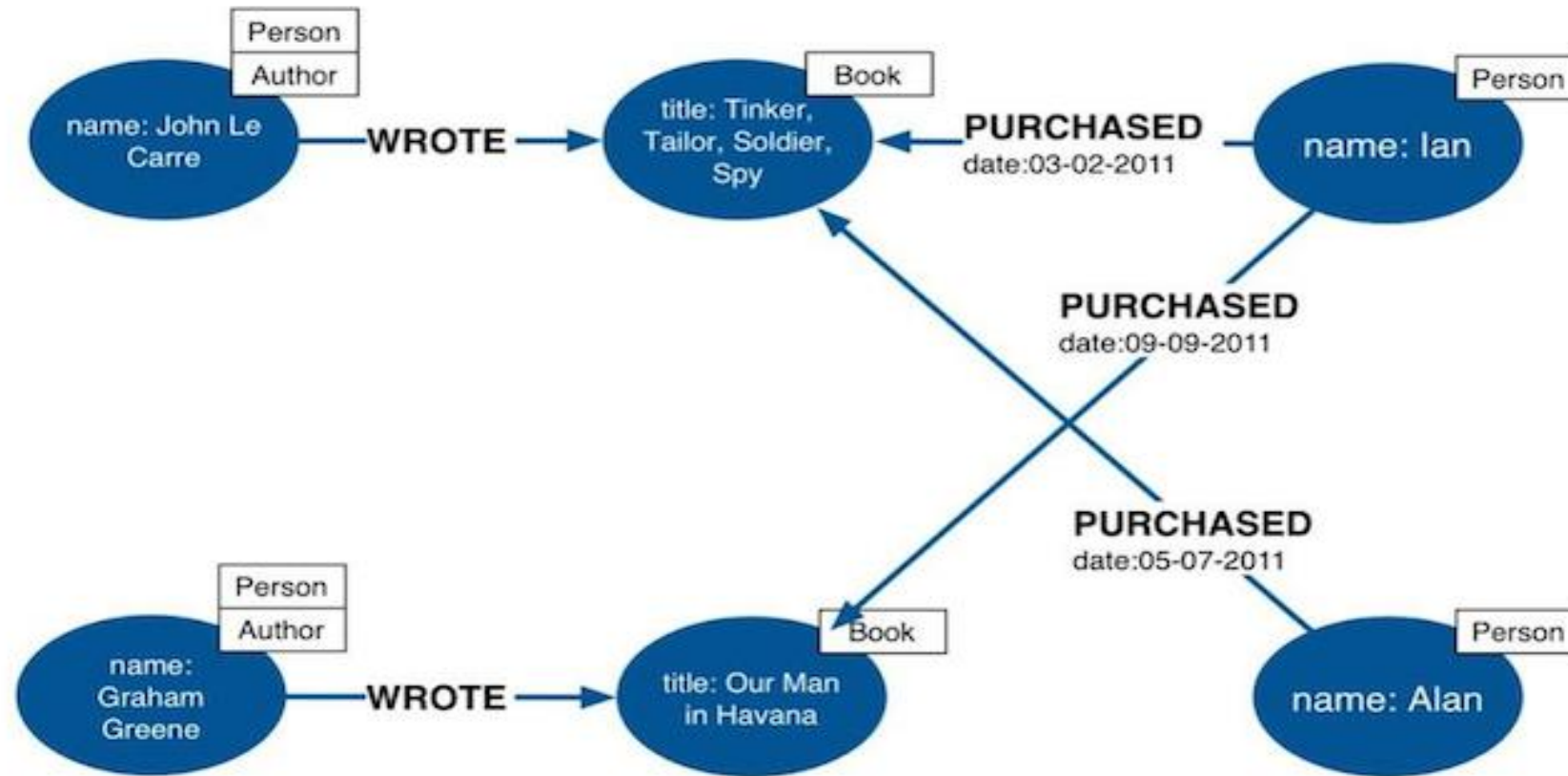
---

**Labels:** allow to assign roles or types to nodes, a node can have any number of labels.



# Neo4j Property Graph Model

---



# 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'})
```

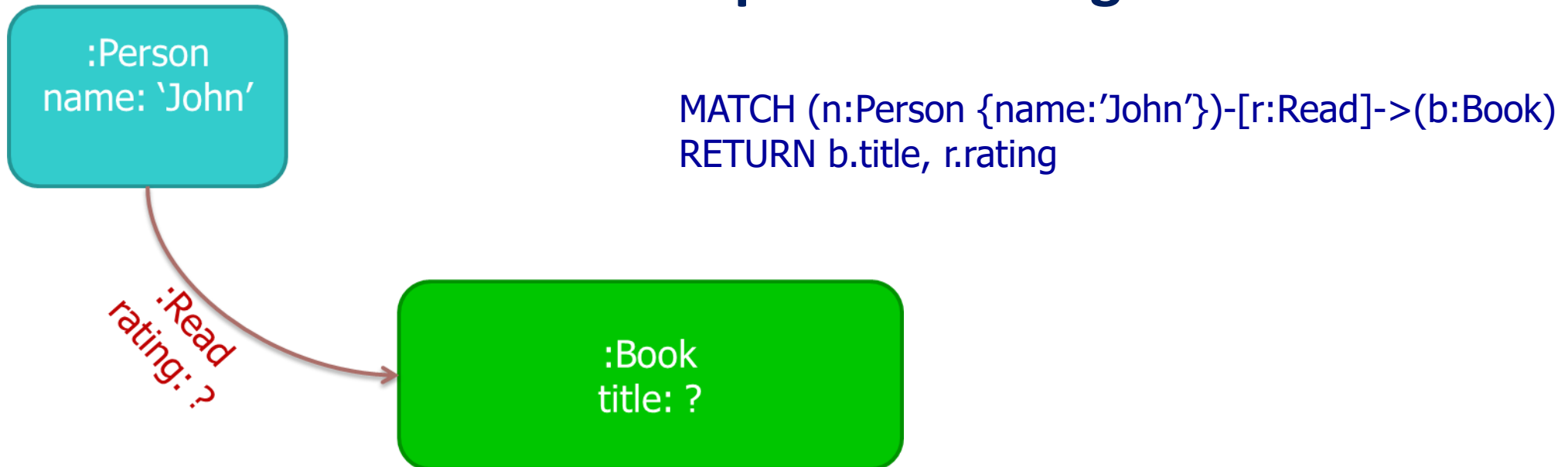
```
CREATE (x)-[:WorksAt]->(c:Company{name:'1B Dollars'})
```

# Querying the graph database

---

- Queries are also graphs!

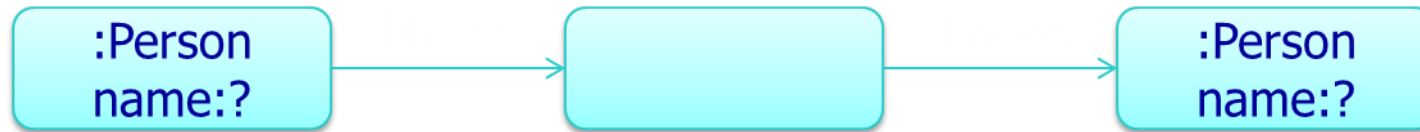
**“Find the titles of all books that a person named John has read and report his ratings”**



# 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

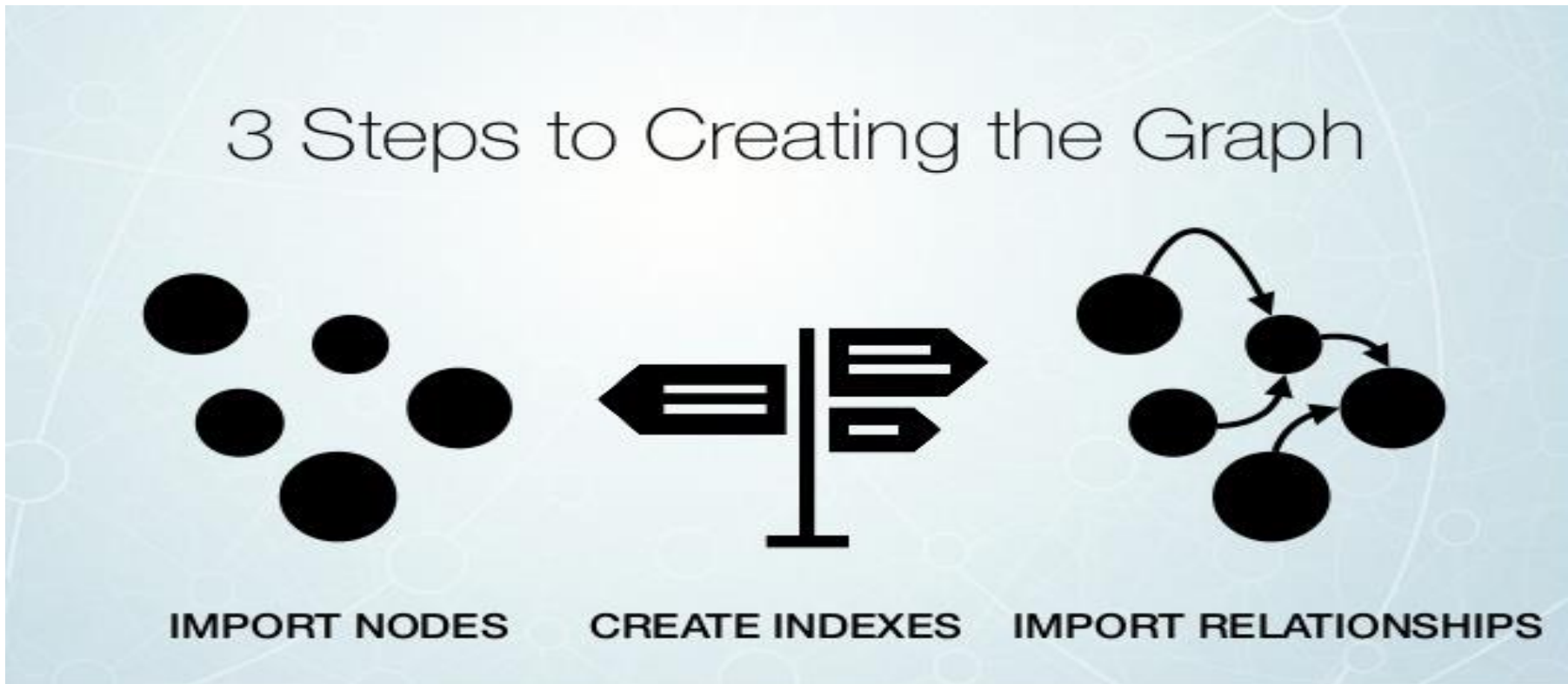
---

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>
- LOAD JSON (apoc.load.json)
  - <https://neo4j.com/docs/labs/apoc/current/import/load-json/>
- Neo4j Import Tool
  - <http://neo4j.com/docs/stable/import-tool.html>

# Import Data

---



# 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

---

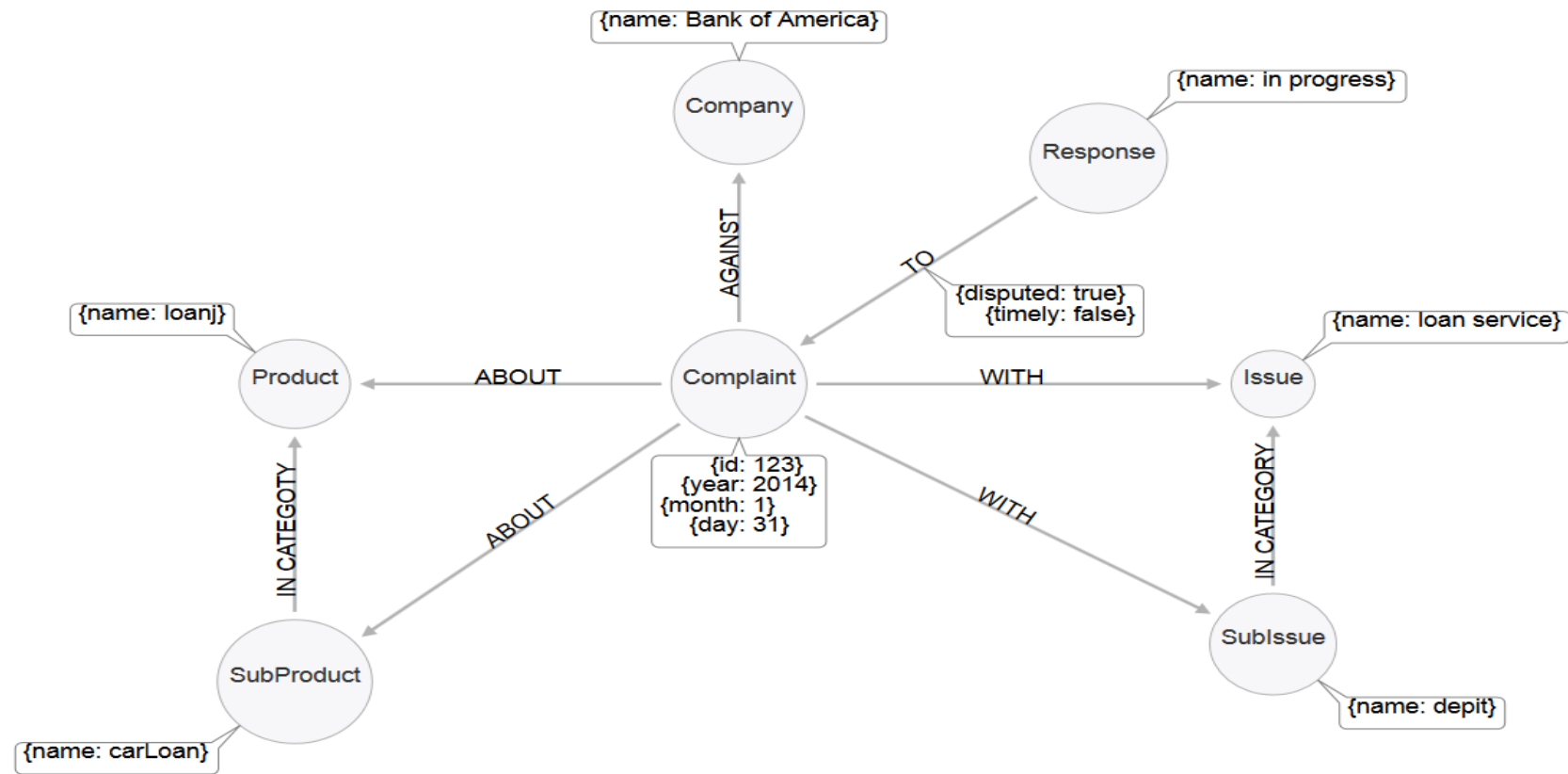
# Consumer Complaints Example

---

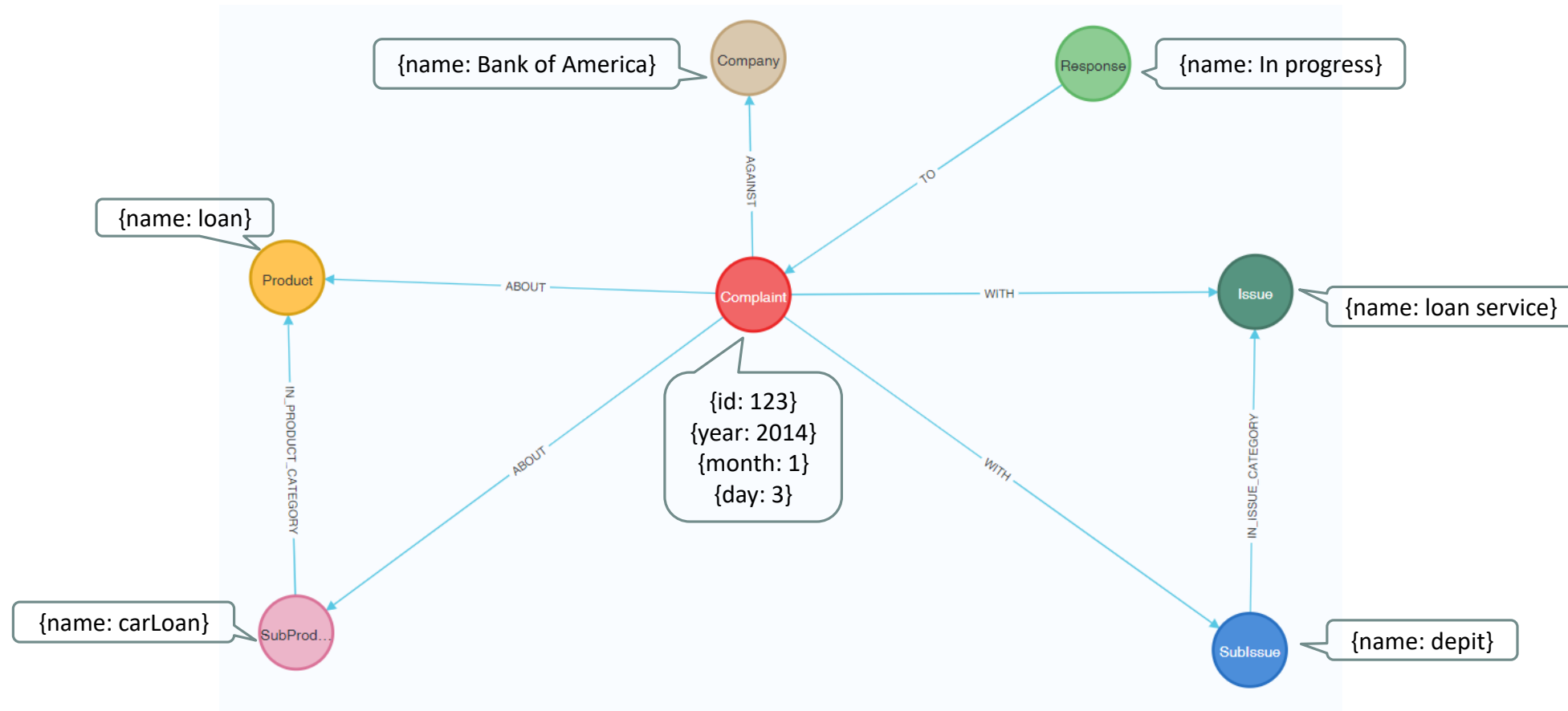
- Model Description:
- **7 nodes:** Company, Response, Product, SubProduct, Issue, SubIssue, Complaint
- **6 relationships:** TO, AGAINST, ABOUT, WITH, IN\_ISSUE\_CATEGORY, IN\_PRODUCT\_CATEGORY
- **1 CSV file:**

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



# 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;
```

# Web Interface Query 1

The screenshot shows the Neo4j web interface in a browser. The address bar indicates the URL is `localhost:7474/browser/`. The interface is divided into a left sidebar with navigation icons (database, star, folder) and a main content area. The main content area has a query editor at the top and a results table below.

**Query Editor:**

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

**Results Table:**

response	count
CLOSED WITH EXPLANATION	96929
CLOSED WITH NON-MONETARY RELIEF	9476
CLOSED WITHOUT RELIEF	4851
CLOSED WITH MONETARY RELIEF	4426
CLOSED	3061
CLOSED WITH RELIEF	714
UNTIMELY RESPONSE	2

Returned 7 rows in 965 ms.

# Web Interface Query 7

Neo4j

localhost:7474/browser/

```
1 MATCH (sub:SubProduct)-[:IN_CATEGORY]->(p:Product)
2 WITH sub, COLLECT(p) AS products
3 WHERE LENGTH(products) > 1
4 RETURN sub, products;
```

\$ MATCH (sub:SubProduct)-[:IN\_CATEGORY]->(p:Product) WITH sub, COLLECT(p) AS products WHERE LENGTH(products) > 1 RETURN sub, products;

\*(5) Product(3) SubProduct(2)

\*(4) IN\_CATEGORY(4)

Graph

Rows

Text

Code

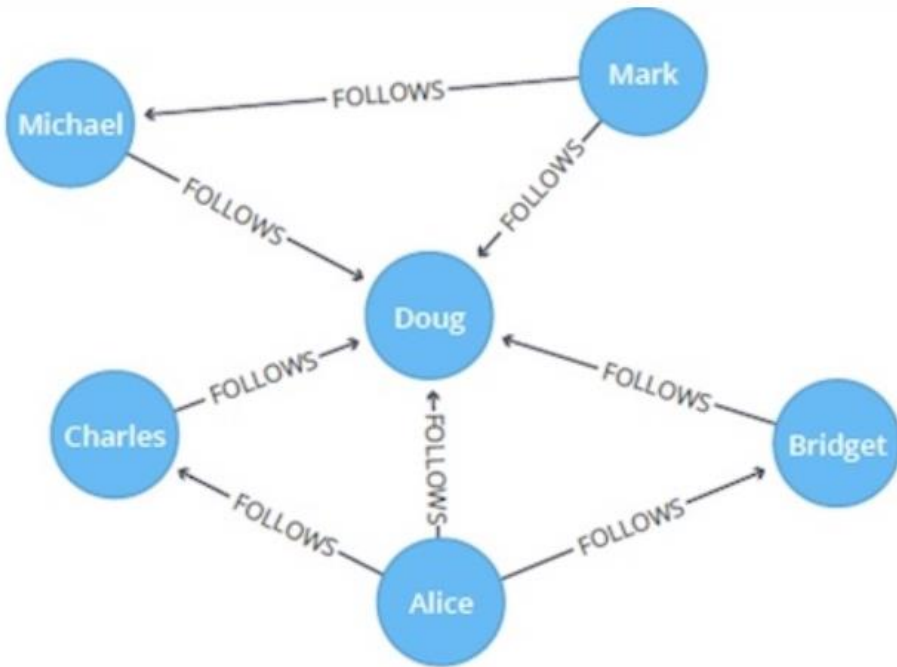
The image shows the Neo4j web interface. At the top, a Cypher query is entered in a text area: `1 MATCH (sub:SubProduct)-[:IN_CATEGORY]->(p:Product)`, `2 WITH sub, COLLECT(p) AS products`, `3 WHERE LENGTH(products) > 1`, and `4 RETURN sub, products;`. Below the query area, the same query is displayed in a read-only format. Underneath, the results are shown in a table-like view with columns: `*(5)`, `Product(3)`, and `SubProduct(2)`. A fourth column, `*(4) IN_CATEGORY(4)`, is also present. The main area displays a graph visualization of the query results. The graph consists of five nodes: `STUDE...`, `NON-FE...`, `DEBT COLLE...`, `PAYDAY LOAN`, and `PAYDAY LOAN`. The nodes are connected by directed edges labeled `IN_CATEGORY`. The connections are: `STUDE...` to `NON-FE...`, `NON-FE...` to `DEBT COLLE...`, `DEBT COLLE...` to `PAYDAY LOAN`, and `PAYDAY LOAN` to `PAYDAY LOAN`.

# 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)

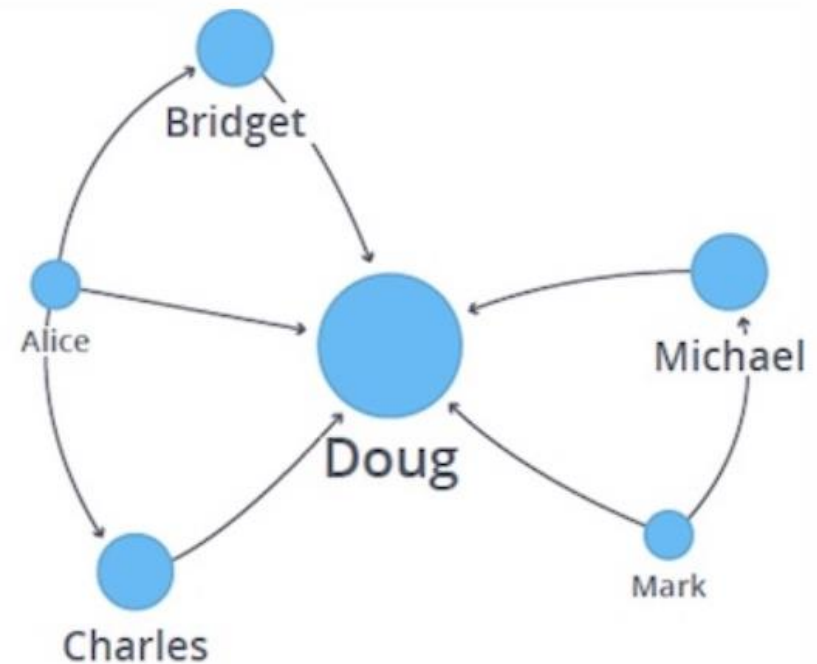
```
MATCH (u:User)  
OPTIONAL MATCH (u)-[:FOLLOWS]->(f:User)  
OPTIONAL MATCH (u)<-[:FOLLOWS]-(follower:User)  
RETURN u.name AS name,  
      COUNT(DISTINCT f) AS follows,  
      COUNT(DISTINCT follower) AS followers
```

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

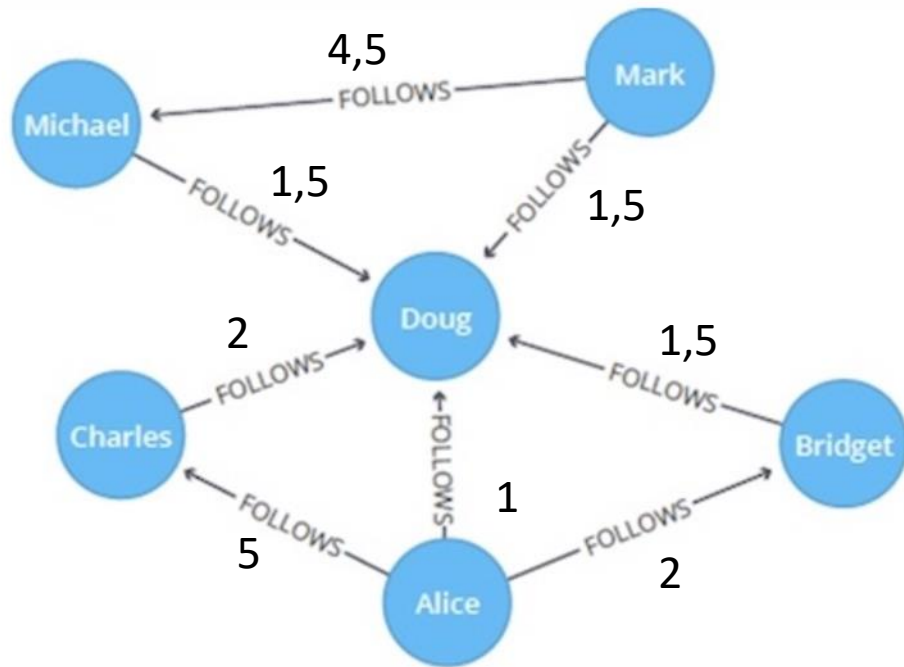
---

- 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

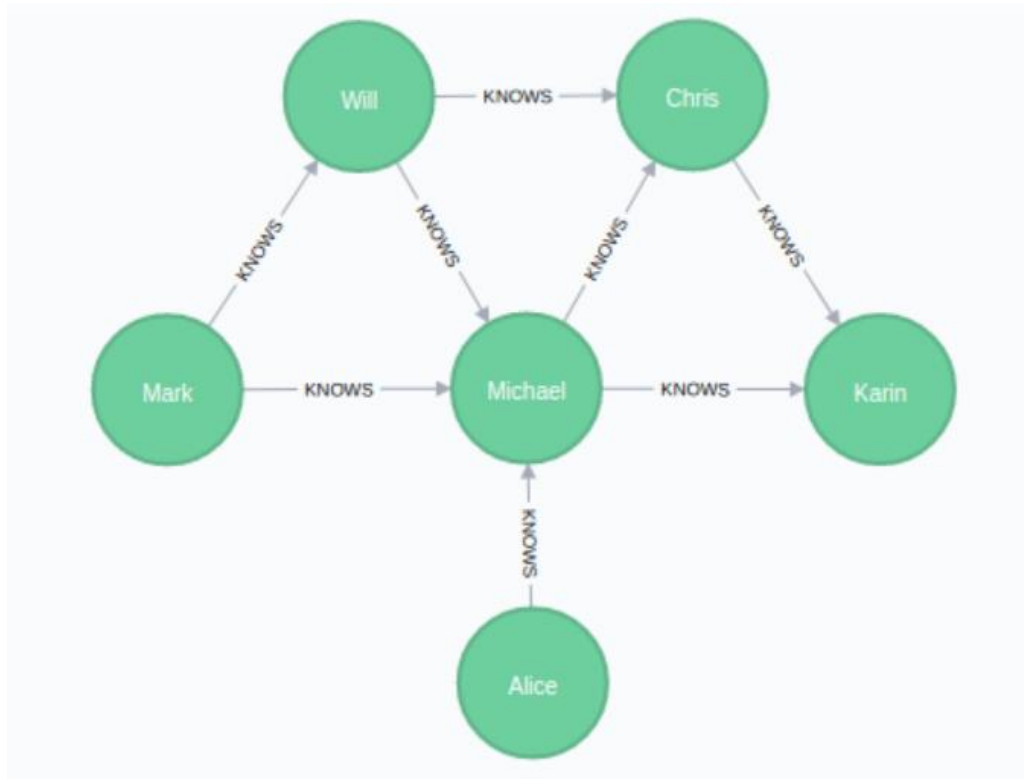
- The following will run the algorithm and stream results, showing which users have the most weighted follows (out degree):

```
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

# 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

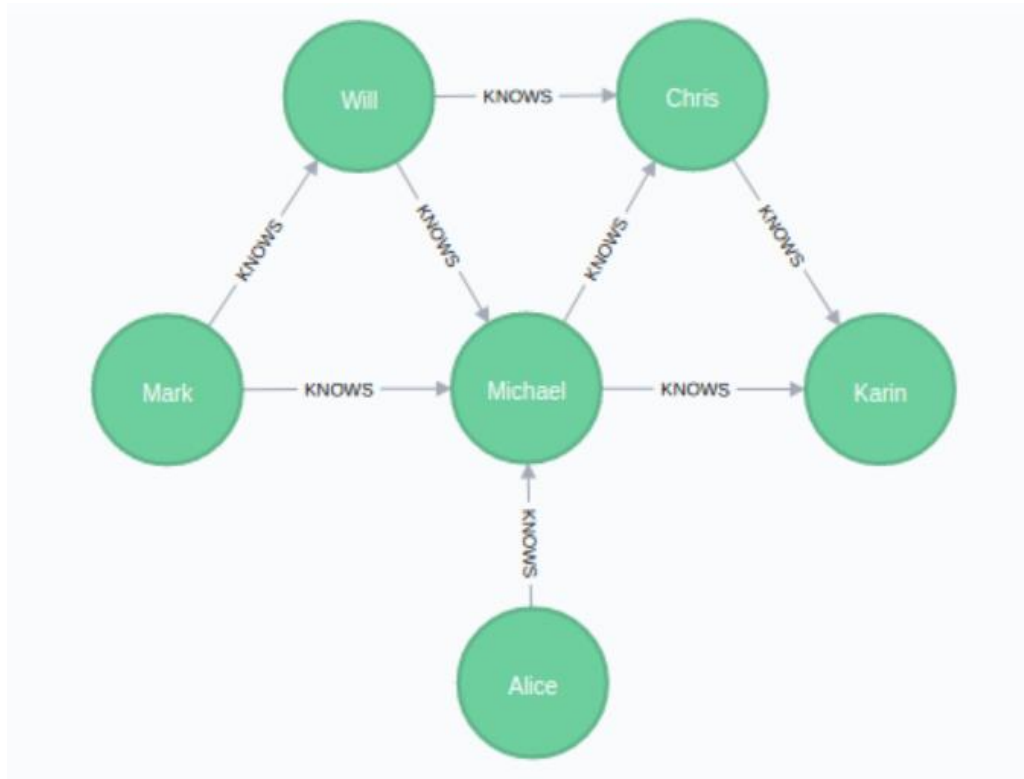
---

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

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

averageClusteringCoefficient	nodeCount
0.6055555555555555	6

# 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;
```

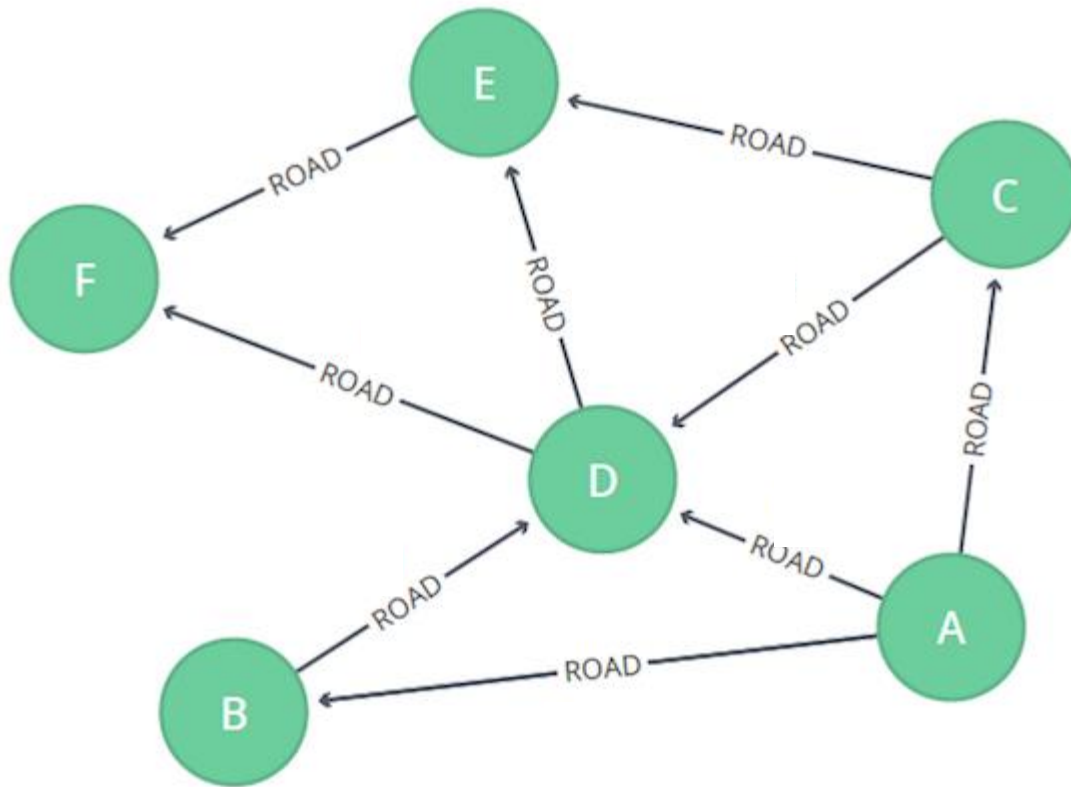
"user"	"centrality"
"Michael"	4.0
"Chris"	0.5
"Will"	0.5
"Alice"	0.0
"Karin"	0.0
"Mark"	0.0

# Shortest Paths Examples

---

# Create Graph Unweighted

---



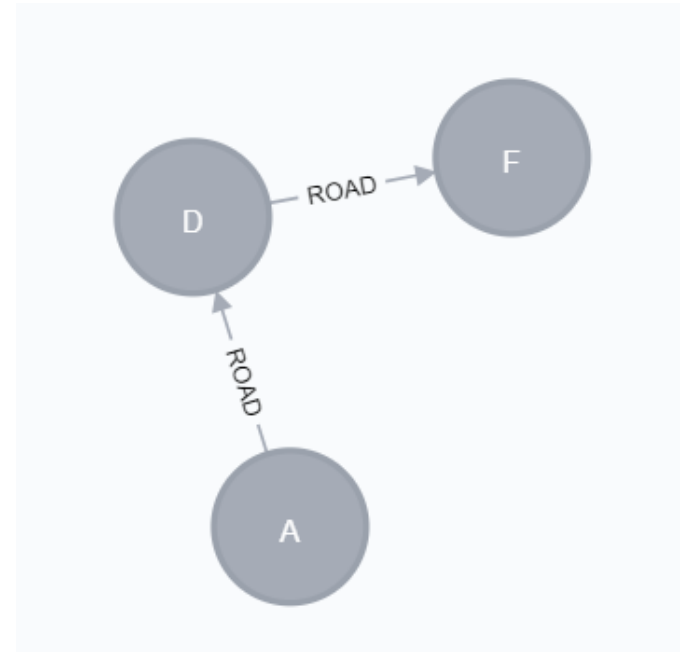
```
MERGE(a:Loc{name:"A"})
MERGE(b:Loc{name:"B"})
MERGE(c:Loc{name:"C"})
MERGE(d:Loc{name:"D"})
MERGE(e:Loc{name:"E"})
MERGE(f:Loc{name:"F"})
MERGE(a)-[:ROAD]->(b)
MERGE(a)-[:ROAD]->(c)
MERGE(a)-[:ROAD]->(d)
MERGE(b)-[:ROAD]->(d)
MERGE(c)-[:ROAD]->(d)
MERGE(c)-[:ROAD]->(e)
MERGE(d)-[:ROAD]->(e)
MERGE(d)-[:ROAD]->(f)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

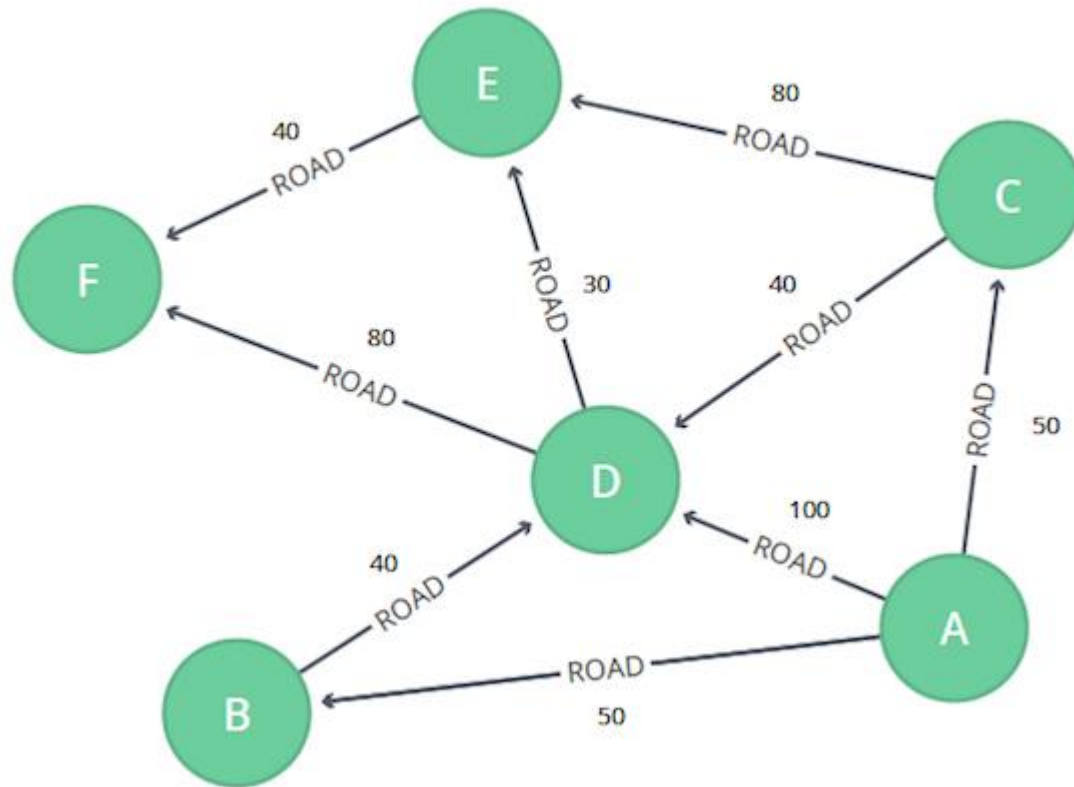
- The following query finds **all pair shortest paths** between all nodes of the graph.

```
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

# Create Graph Weighted

---



**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 {cost:50}]->(b)**  
**MERGE (a)-[:ROAD {cost:50}]->(c)**  
**MERGE (a)-[:ROAD {cost:100}]->(d)**  
**MERGE (b)-[:ROAD {cost:40}]->(d)**  
**MERGE (c)-[:ROAD {cost:40}]->(d)**  
**MERGE (c)-[:ROAD {cost:80}]->(e)**  
**MERGE (d)-[:ROAD {cost:30}]->(e)**  
**MERGE (d)-[:ROAD {cost:80}]->(f)**  
**MERGE (e)-[:ROAD {cost:40}]->(f)**

# Shortest Path Weighted Graphs (Dijkstra)

- The following query calculates **the point to point shortest path** from A to F using Dijkstra (weighted graph), using Graph Data Science Library

```
CALL gds.graph.project(
  'myGraph',
  'Loc',
  {
    ROAD: {
      type: 'ROAD',
      properties: 'cost'
    }
  }
)
YIELD graphName, nodeCount, relationshipCount;

MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'F'})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: id(start),
  targetNode: id(end),
  relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs
RETURN
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS name,
  totalCost;
```

name	totalCost
["A", "B", "D", "E", "F"]	160.0

# Shortest Path Weighted Graphs (Dijkstra)

- The following query calculates **the point to point shortest path** from A to F using Dijkstra (weighted graph), using Graph Data Science Library

```
MATCH (source:Loc {name: 'A'}) // Set A as the source node
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: id(source),
  relationshipWeightProperty: 'cost'
})
YIELD targetNode, totalCost
WITH gds.util.asNode(targetNode).name AS target, totalCost
RETURN target, totalCost
ORDER BY totalCost;
```

name	cost
"A"	0.0
"B"	50.0
"D"	90.0
"E"	120.0
"F"	160.0

# Shortest Path Weighted Graphs (Dijkstra)

- The following query calculates **single source shortest paths** from A to all other nodes using Dijkstra

```
MATCH (start:Loc {name: 'A'})
CALL gds.allShortestPaths.dijkstra.stream('myGraph', {
  sourceNode: id(start),
  relationshipWeightProperty: 'cost'
})
YIELD index, sourceNode, targetNode, totalCost
RETURN
  gds.util.asNode(sourceNode).name AS startNode,
  gds.util.asNode(targetNode).name AS endNode,
  totalCost AS distance
ORDER BY distance;
```

startNode	endNode	distance
"A"	"A"	0.0
"A"	"B"	50.0
"A"	"C"	50.0
"A"	"D"	90.0
"A"	"E"	120.0
"A"	"F"	160.0

# Shortest Path Weighted Graphs (Dijkstra)

- The following query calculates **all pair shortest paths** for all node of the graph using Dijkstra

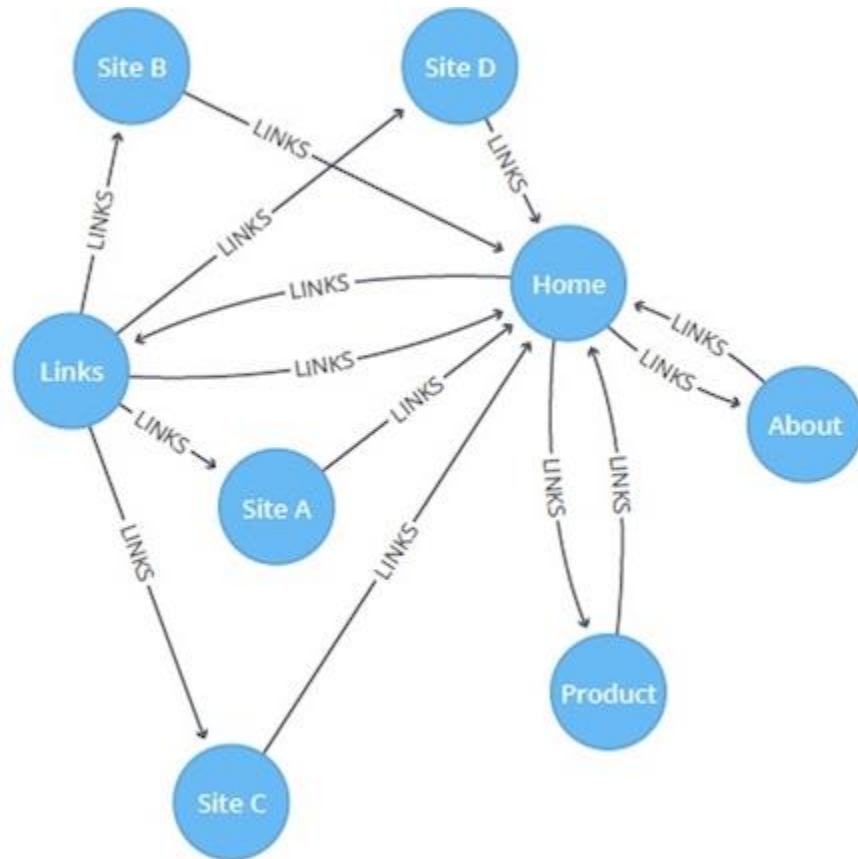
```
MATCH (source:Loc), (target:Loc) // Iterate over all node pairs
WHERE id(source) <> id(target) // Avoid self-loops
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: id(source),
  targetNode: id(target),
  relationshipWeightProperty: 'cost'
})
YIELD totalCost
RETURN
  gds.util.asNode(id(source)).name AS source,
  gds.util.asNode(id(target)).name AS target,
  totalCost AS distance
ORDER BY distance desc ;
```

source	target	distance
"A"	"F"	160.0
"A"	"E"	120.0
"B"	"F"	110.0
"C"	"F"	110.0
"A"	"D"	90.0
"B"	"E"	70.0

# Page Rank Examples

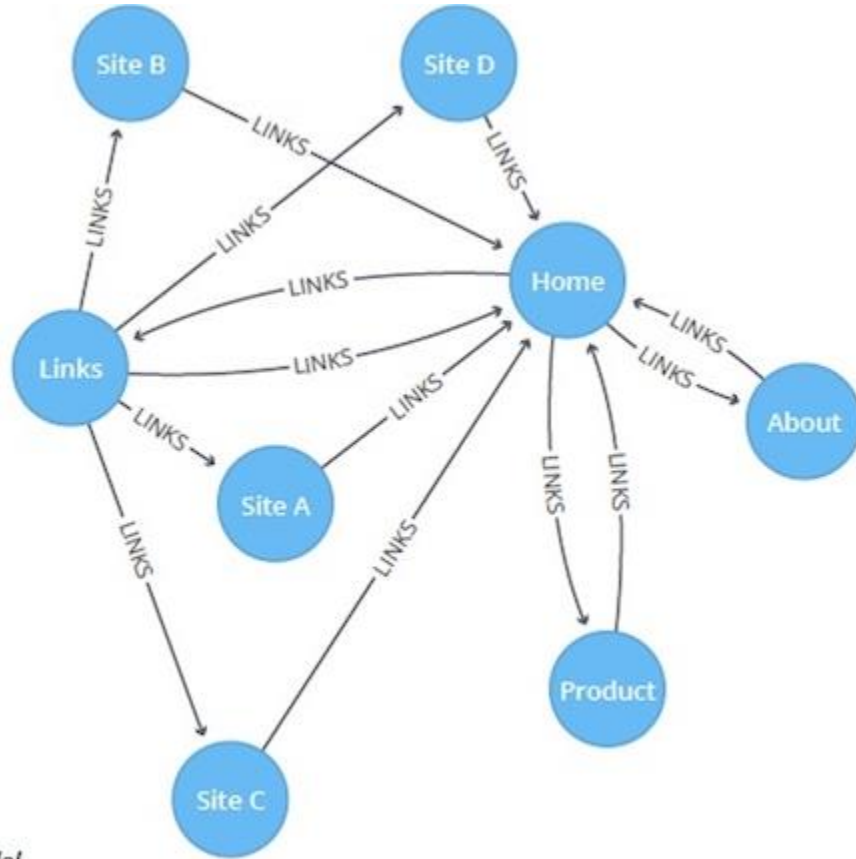
---

# Create Graph Weighted



```
CREATE (home:Page {name:'Home'})
CREATE (about:Page {name:'About'})
CREATE (product:Page {name:'Product'})
CREATE (links:Page {name:'Links'})
CREATE (a:Page {name:'Site A'})
CREATE (b:Page {name:'Site B'})
CREATE (c:Page {name:'Site C'})
CREATE (d:Page {name:'Site D'})
```

# Create Graph Weighted



el

```
MATCH (home:Page {name: 'Home'}), (about:Page {name: 'About'})
CREATE (home)-[:LINKS {weight: 0.2}]->(about);
MATCH (home:Page {name: 'Home'}), (links:Page {name: 'Links'})
CREATE (home)-[:LINKS {weight: 0.2}]->(links);
MATCH (home:Page {name: 'Home'}), (product:Page {name: 'Product'})
CREATE (home)-[:LINKS {weight: 0.6}]->(product);
MATCH (about:Page {name: 'About'}), (home:Page {name: 'Home'})
CREATE (about)-[:LINKS {weight: 1.0}]->(home);
MATCH (product:Page {name: 'Product'}), (home:Page {name: 'Home'})
CREATE (product)-[:LINKS {weight: 1.0}]->(home);
MATCH (a:Page {name: 'Site A'}), (home:Page {name: 'Home'})
CREATE (a)-[:LINKS {weight: 1.0}]->(home);
MATCH (b:Page {name: 'Site B'}), (home:Page {name: 'Home'})
CREATE (b)-[:LINKS {weight: 1.0}]->(home);
MATCH (c:Page {name: 'Site C'}), (home:Page {name: 'Home'})
CREATE (c)-[:LINKS {weight: 1.0}]->(home);
MATCH (d:Page {name: 'Site D'}), (home:Page {name: 'Home'})
CREATE (d)-[:LINKS {weight: 1.0}]->(home);
MATCH (links:Page {name: 'Links'}), (home:Page {name: 'Home'})
CREATE (links)-[:LINKS {weight: 0.8}]->(home);
MATCH (links:Page {name: 'Links'}), (a:Page {name: 'Site A'})
CREATE (links)-[:LINKS {weight: 0.05}]->(a);
MATCH (links:Page {name: 'Links'}), (b:Page {name: 'Site B'})
CREATE (links)-[:LINKS {weight: 0.05}]->(b);
MATCH (links:Page {name: 'Links'}), (c:Page {name: 'Site C'})
CREATE (links)-[:LINKS {weight: 0.05}]->(c);
MATCH (links:Page {name: 'Links'}), (d:Page {name: 'Site D'})
CREATE (links)-[:LINKS {weight: 0.05}]->(d);
```

# Page Rank Unweighted

---

- The following statement will project the graph and store it in the graph catalog.

```
CALL gds.graph.project(  
  'myGraph',  
  'Page',  
  {  
    LINKS: {  
      properties: 'weight'  
    }  
  }  
)
```

# Page Rank Unweighted

---

- The following will run PageRank algorithm and stream results on the projected unweighted graph:

```
CALL gds.pageRank.stream(  
  'myGraph',  
  { maxIterations: 20, dampingFactor: 0.85 }  
)  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score  
ORDER BY score DESC, name ASC;
```

name	score
"Home"	3.215681999884452
"About"	1.0542700552146724
"Links"	1.0542700552146724
"Product"	1.0542700552146724
"Site A"	0.3278578964488539
"Site B"	0.3278578964488539
"Site C"	0.3278578964488539
"Site D"	0.3278578964488539

# Page Rank Weighted

---

- The following will run PageRank algorithm and stream results on the projected weighted graph:

```
CALL gds.pageRank.stream('myGraph', {  
  maxIterations: 20,  
  dampingFactor: 0.85,  
  relationshipWeightProperty: 'weight'  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score  
ORDER BY score DESC, name ASC
```

name	score
"Home"	3.5375102839633907
"Product"	1.93578382916511
"About"	0.7452612763883699
"Links"	0.7452612763883699
"Site A"	0.18152677135466103
"Site B"	0.18152677135466103
"Site C"	0.18152677135466103
"Site D"	0.18152677135466103

# Personalized Page Rank

---

- Personalized Page Rank is a variation of Page Rank which is biased towards a set of sourceNodes. The following show how to run Page Rank centered around 'Site A'

```
MATCH (siteA:Page {name: 'Site A'})  
CALL gds.pageRank.stream('myGraph', {  
  maxIterations: 20,  
  dampingFactor: 0.85,  
  sourceNodes: [siteA]  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score  
ORDER BY score DESC, name ASC
```

name	score
"Home"	0.39902290442518784
"Site A"	0.16890325301726694
"About"	0.11220151747374331
"Links"	0.11220151747374331
"Product"	0.11220151747374331
"Site B"	0.01890325301726691
"Site C"	0.01890325301726691
"Site D"	0.01890325301726691