Graph Databases

Data is more connected:

- Text
- HyperText
- RSS
- Blogs
- Tagging
- RDF

Data is more Semi-Structured:

- If you tried to collect all the data of every movie ever made, how would you model it?
- Actors, Characters, Locations, Dates, Costs, Ratings, Showings, Ticket Sales, etc.

Document Databases

- MongoDB, CouchDB

- Pros:
  - Simple, powerful data model
  - Scalable
- Cons
  - Poor for interconnected data
  - Query model limited to keys and indexes
  - Map reduce for larger queries

Graph Databases

- Data Model:
  - Nodes and Relationships
- Examples:
  - Neo4j, OrientDB, InfiniteGraph, AllegroGraph
- Cypher
  - Graph database query language

Graph Databases: Pros and Cons

- Pros:
  - Powerful data model, as general as RDBMS
  - Connected data locally indexed
  - Easy to query
- Cons
  - Sharding (lots of people working on this)
    - Scales UP reasonably well
  - Requires rewiring your brain

What are graphs good for?

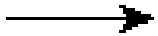
- Real Time Recommendations
- Master Data Management
- Fraud Detection
- Social computing
- Systems management
- Web of things
- Genealogy
- Time series data
- Product catalogue
- Web analytics
- Scientific computing (especially bioinformatics)
- And much more!

What is a Graph?

What is a Graph?

- An abstract representation of a set of objects where some pairs are connected by links.

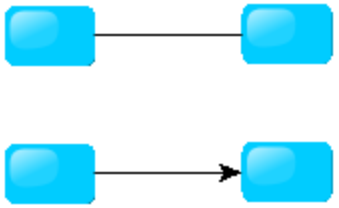
Object (Vertex, Node)



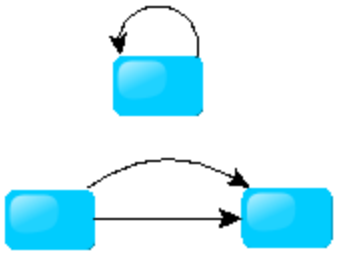
Link (Edge, Arc, Relationship)

Different Kinds of Graphs

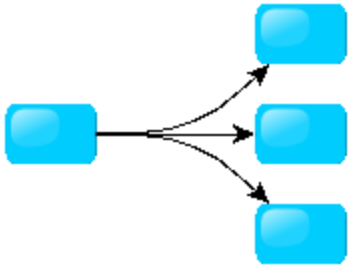
- Undirected Graph
- Directed Graph



- Pseudo Graph
- Multi Graph



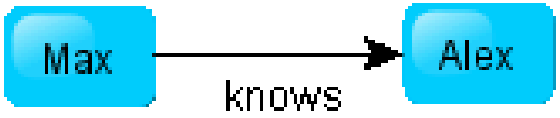
- Hyper Graph



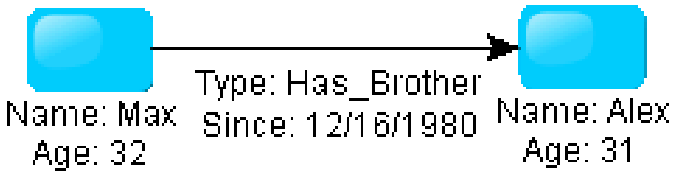
More Kinds of Graphs

- Weighted Graph

- Labeled Graph



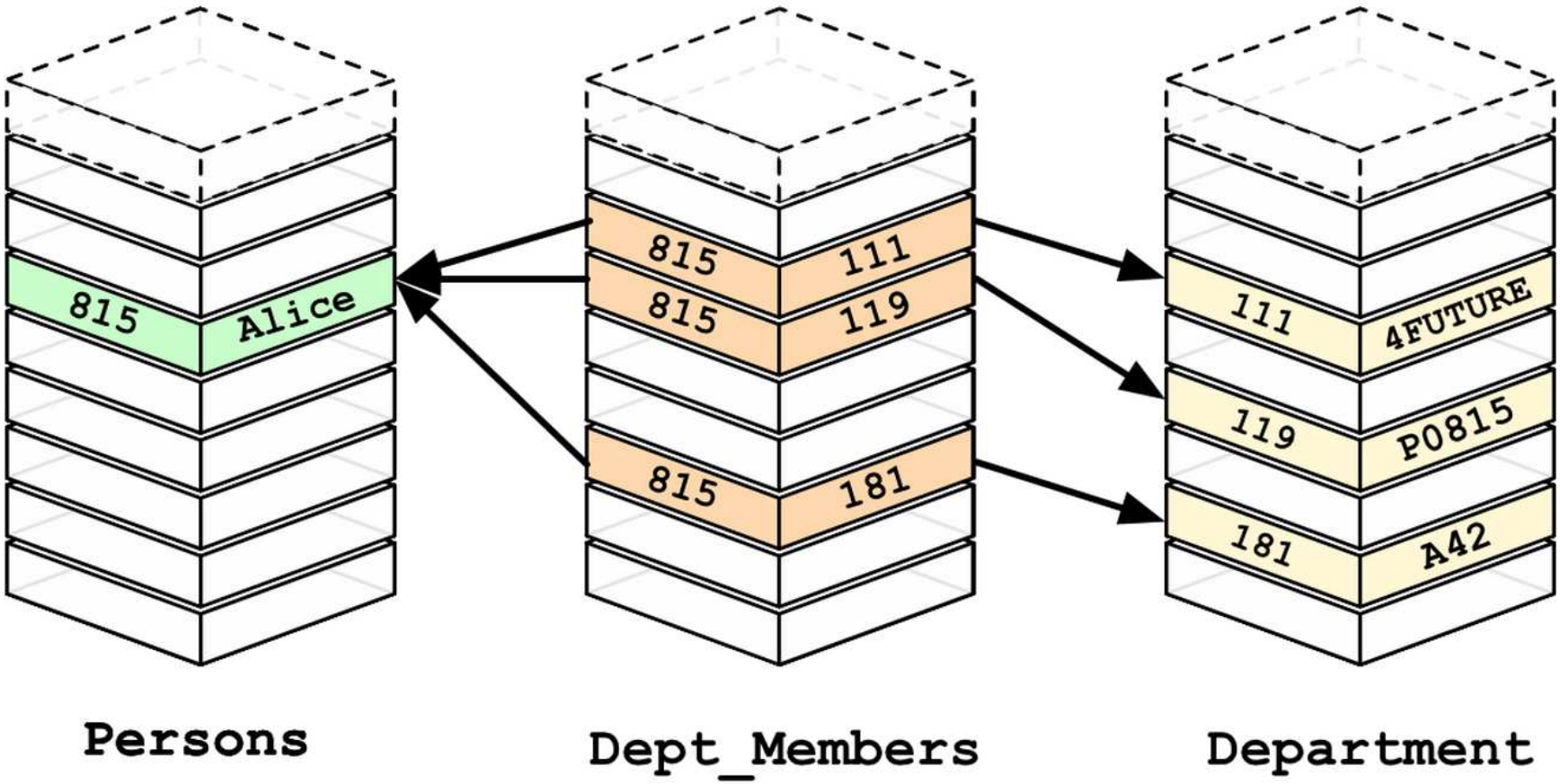
- Property Graph



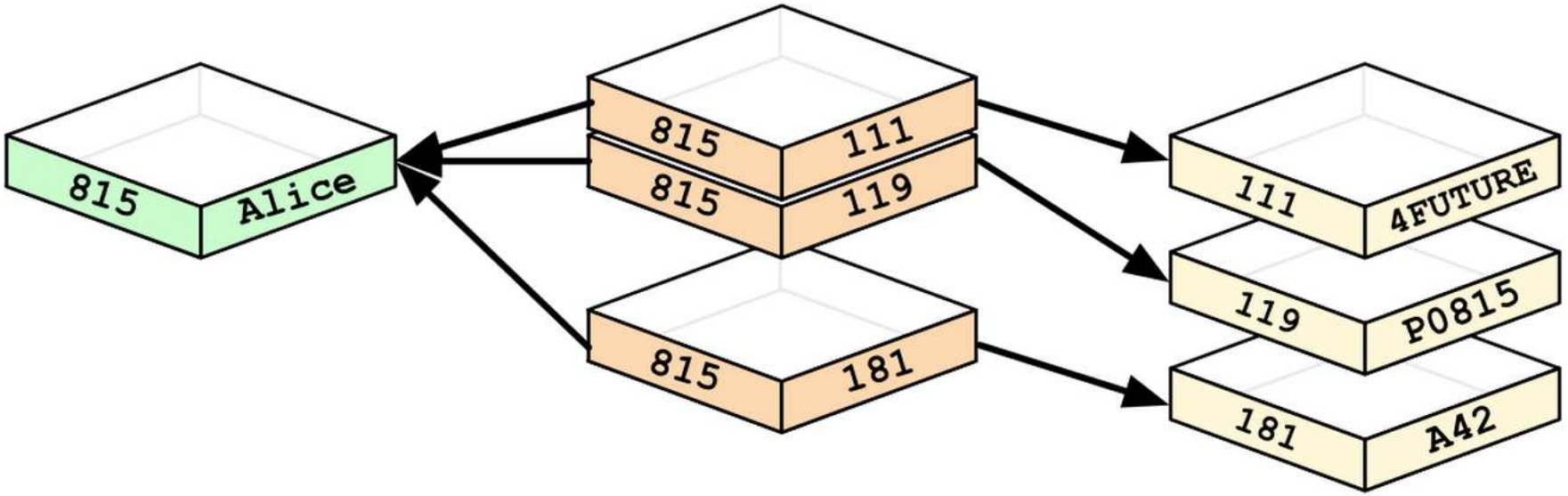
What is a Graph Database?

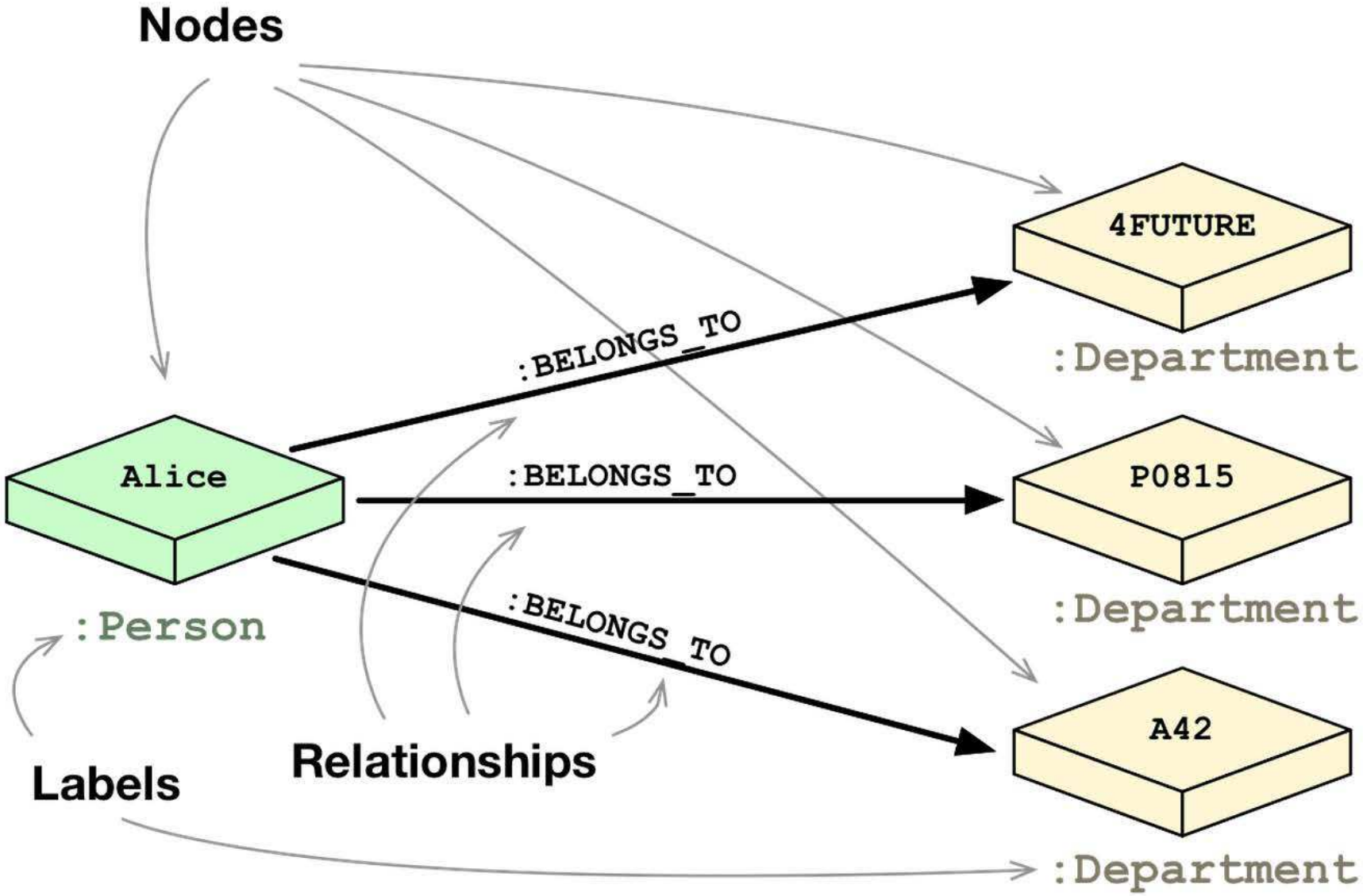
- A database with an explicit graph structure
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Plus an Index for lookups

Relational Databases

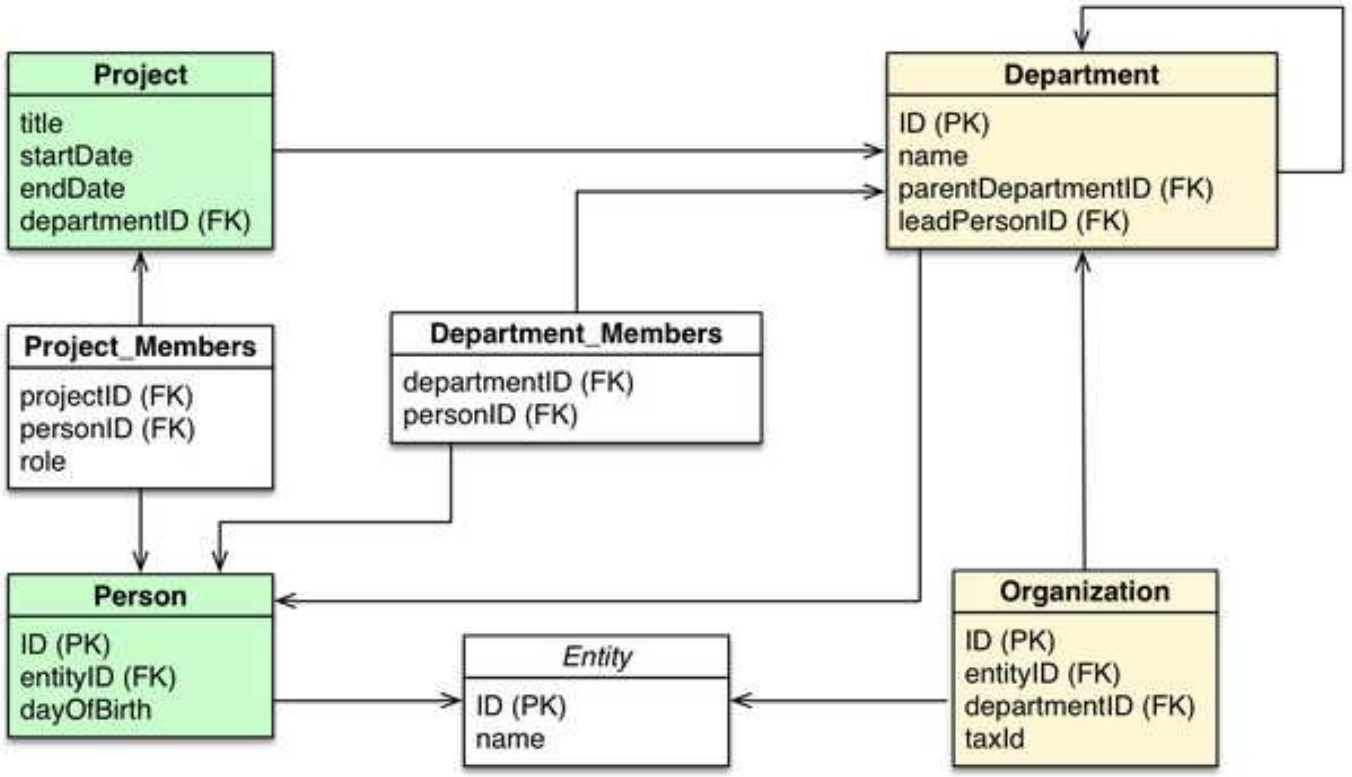


Graph Databases

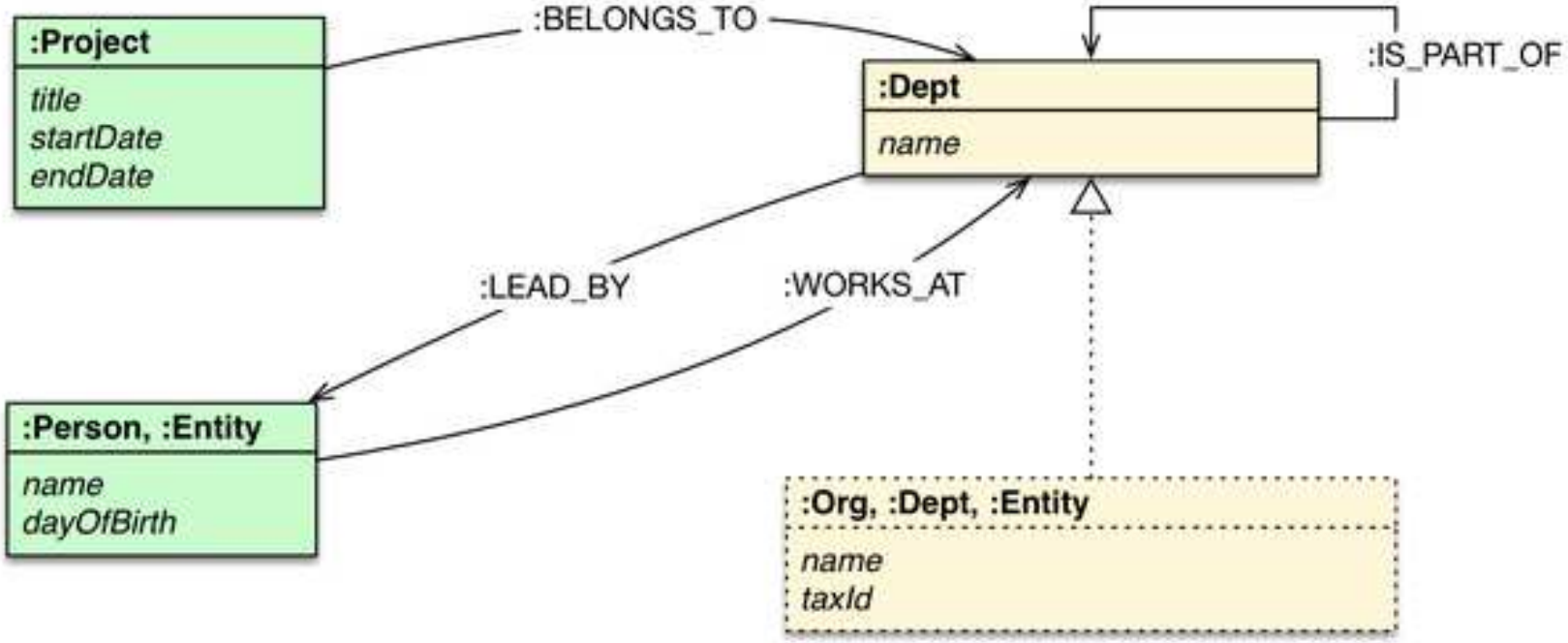



Nodes
4FUTURE
:Department
:BELONGS_TO
Alice
:BELONGS_TO
P0815
:Department
:Person
:BELONGS_TO
A42
Relationships
Labels
:Department

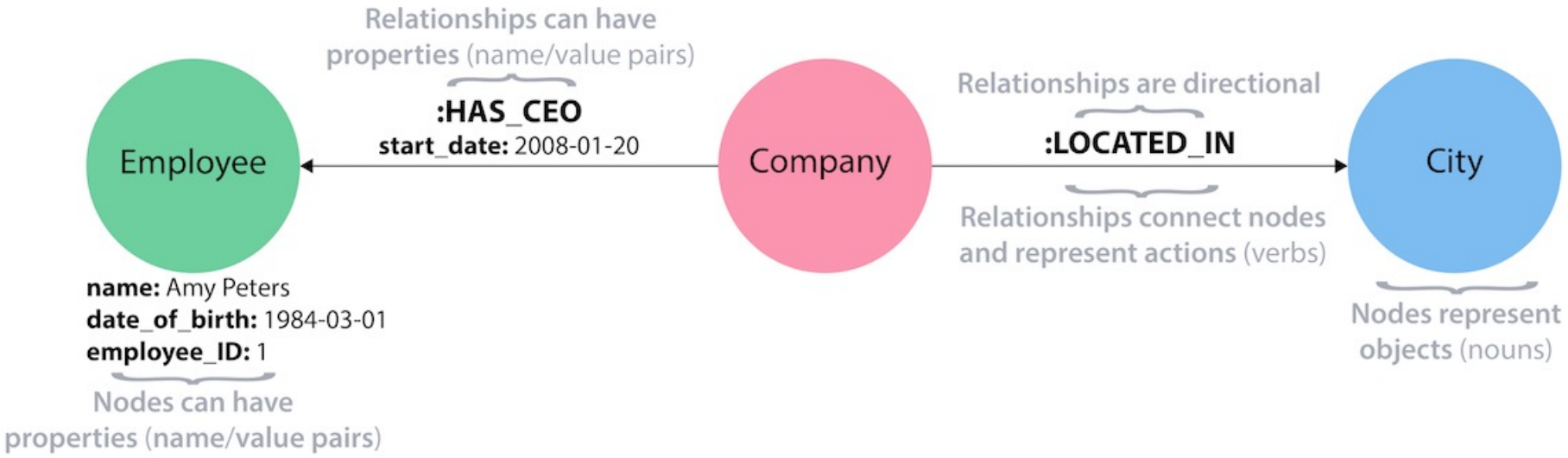
Relational Databases



Graph Databases



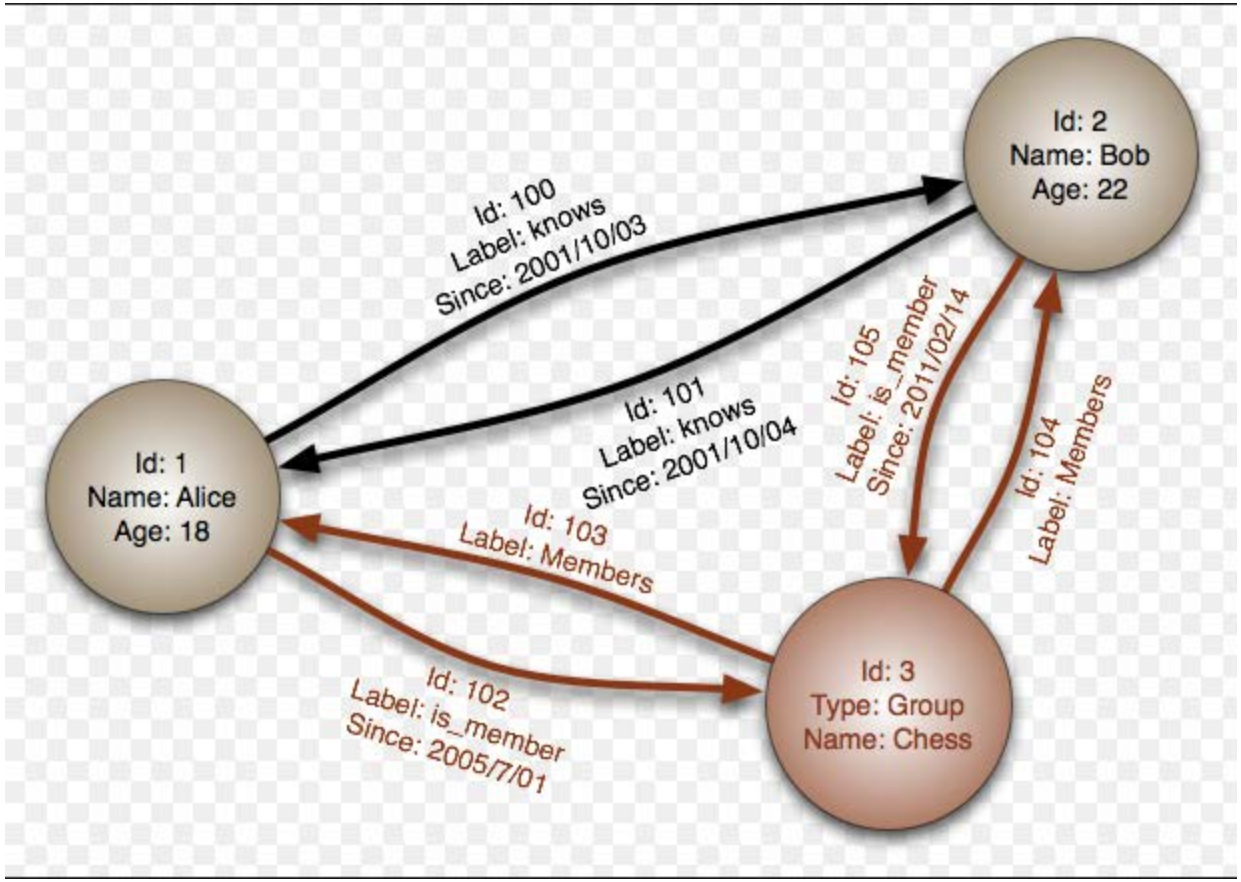
Property graph model



Graph Databases

- Database that uses graph structures with nodes, edges and properties to store data
- Provides index-free adjacency
  - Every node is a pointer to its adjacent element
- Edges hold most of the important information and connect
  - nodes to other nodes
  - nodes to properties

Graph Databases



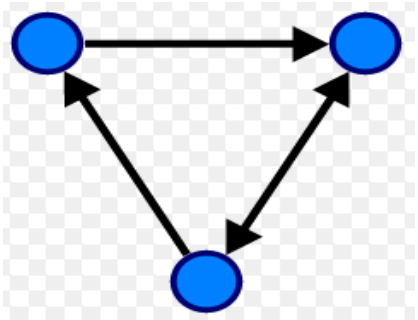
(Wikipedia)

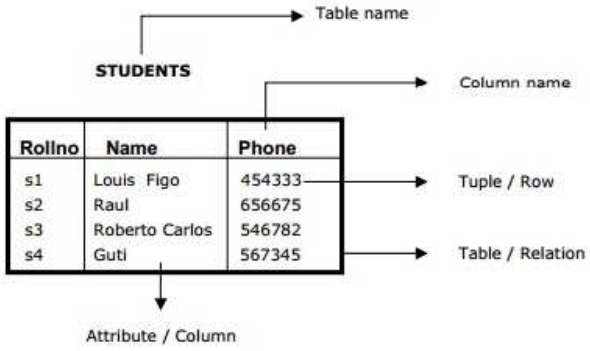
Advantage of Graph Databases

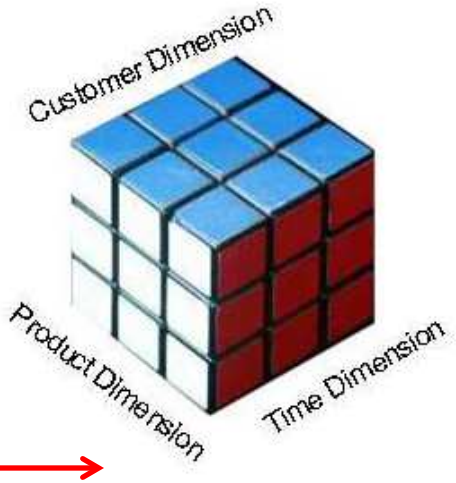
- When there are relationships that you want to analyze Graph databases become a very nice fit because of the data structure
- Graph databases are very fast for associative data sets
  - Like social networks
- Map more directly to object oriented applications
  - Object classification and Parent->Child relationships

Disadvantages

- If data is just tabular with not much relationship between the data, graph databases do not fare well
- OLAP support for graph databases is not well developed
  - Lots of research happening in this area



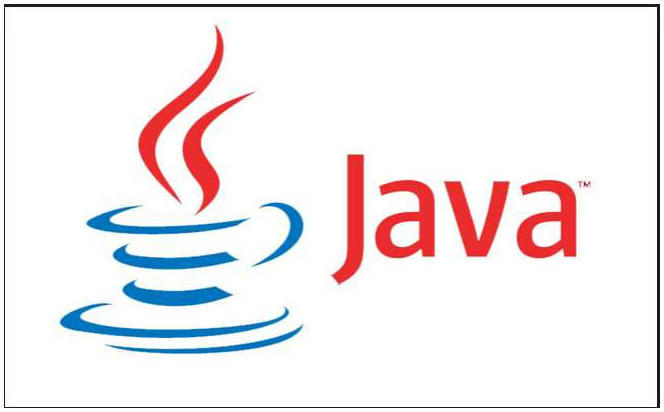


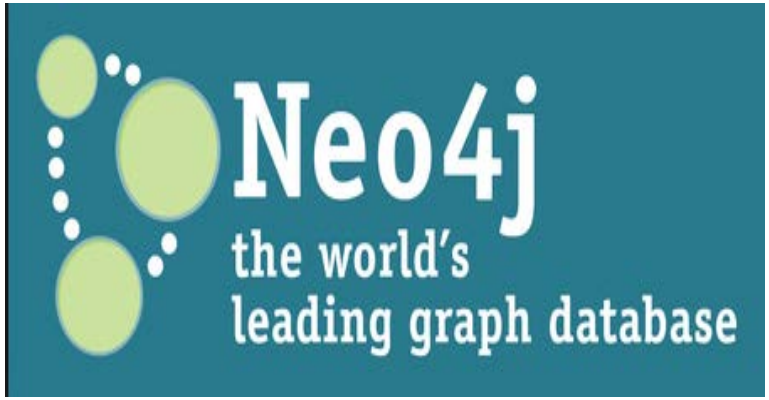


Ease of aggregation

What is Neo4j

- Developed by Neo Technologies
- Most Popular Graph Database
- Implemented in Java
- Open Source





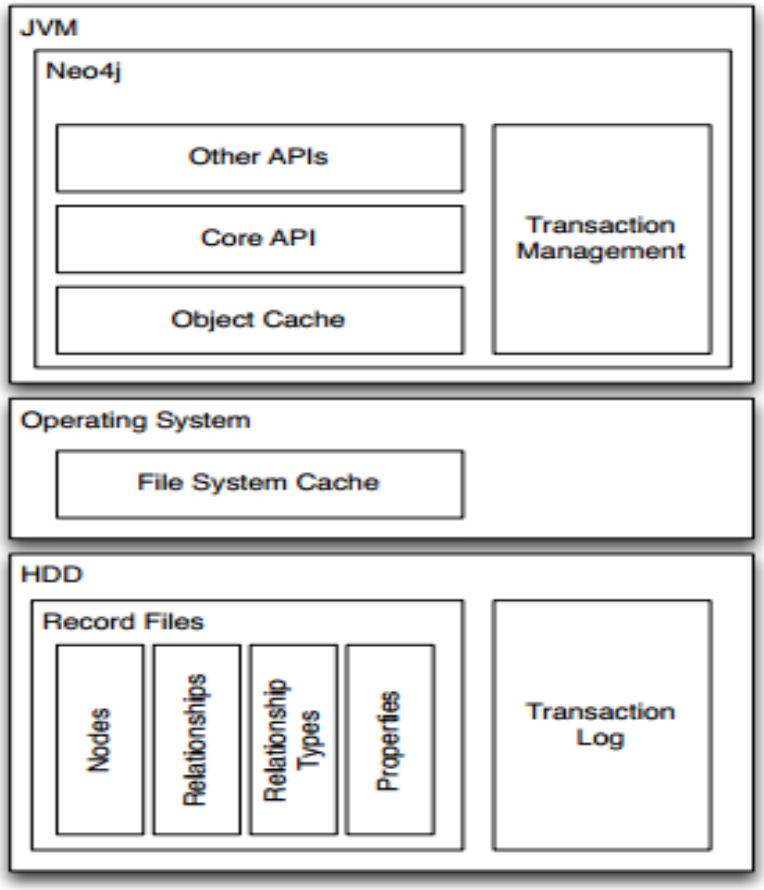
(www.neo4j.org)

Salient features of Neo4j

- Neo4j is schema free – Data does not have to adhere to any convention
- ACID – atomic, consistent, isolated and durable for logical units of work
- Easy to get started and use
- Well documented and large developer community
- Support for wide variety of languages
  - Java, Python, Perl, Scala, Cypher, etc

(Mistry, Deep, 2013)

Neo4j Software Architecture

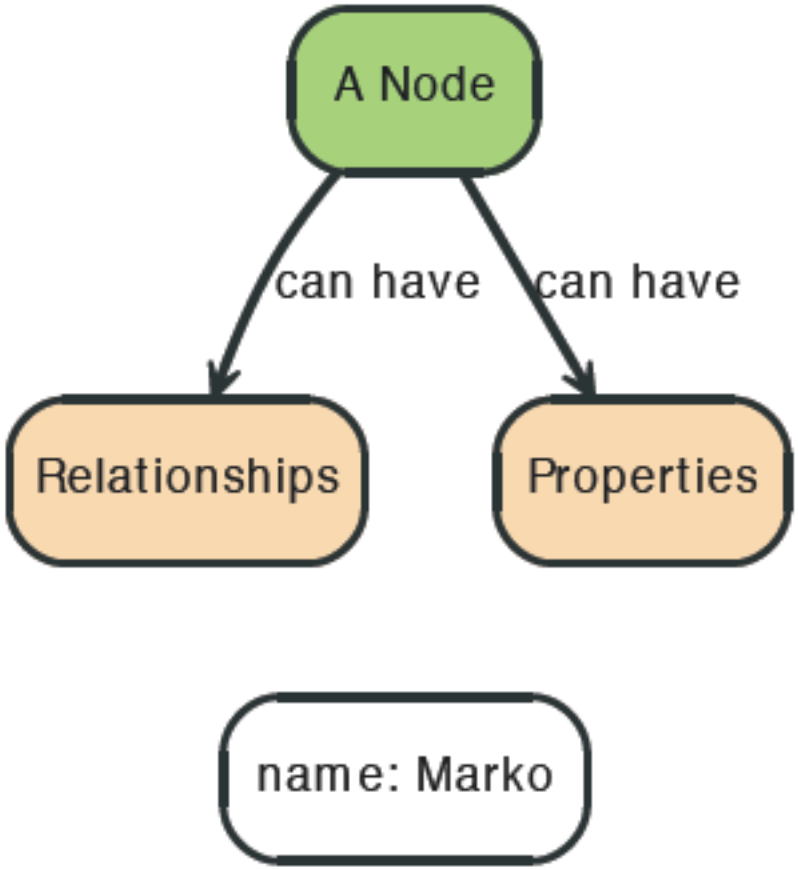


(Bachman, 2013, p.11)

Neo4j Tips

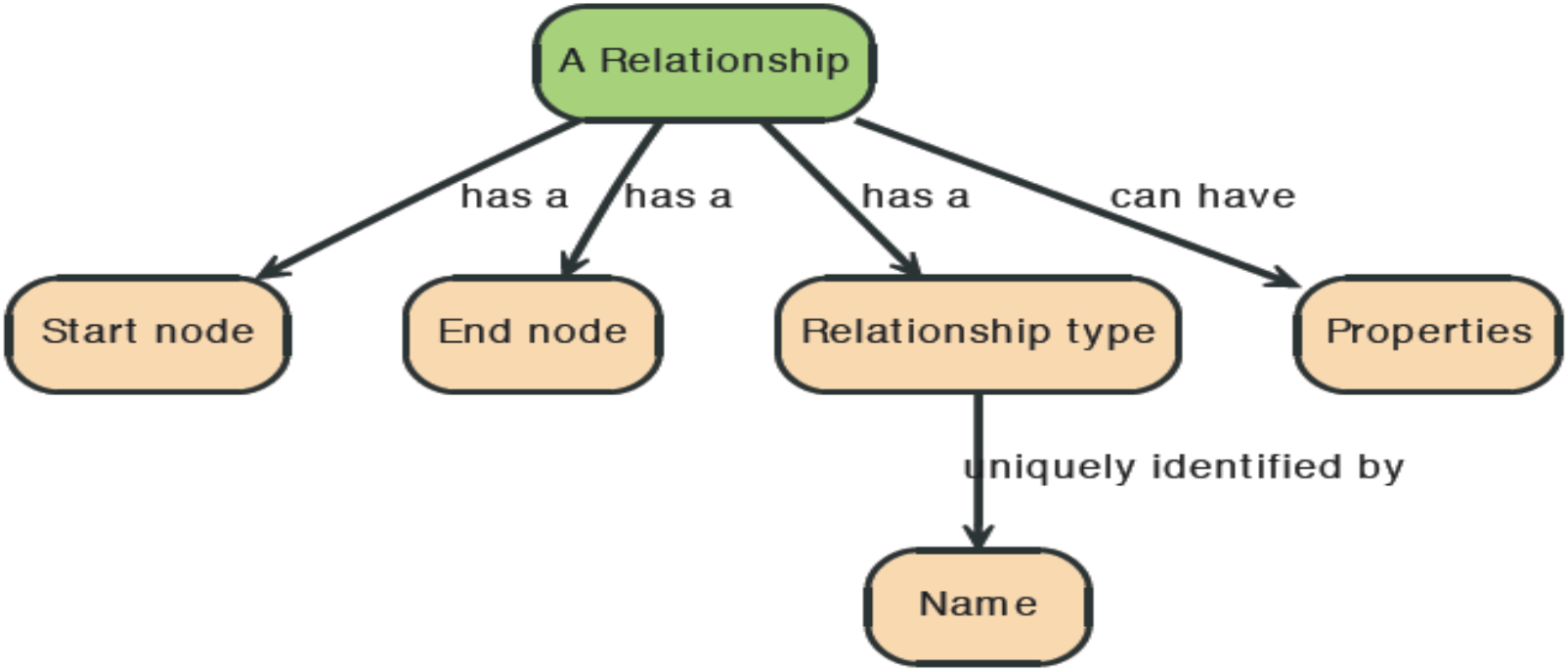
- Each entity table is represented by a label on nodes
- Each row in a entity table is a node
- Columns on those tables become node properties.
- Join tables are transformed into relationships, columns on those tables become relationship properties

Node in Neo4j

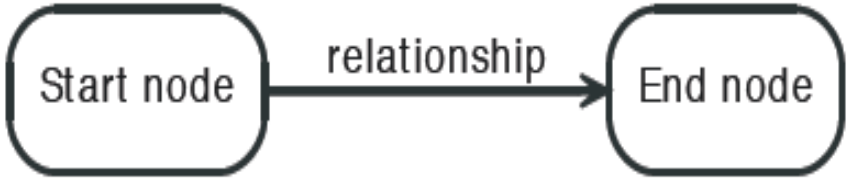


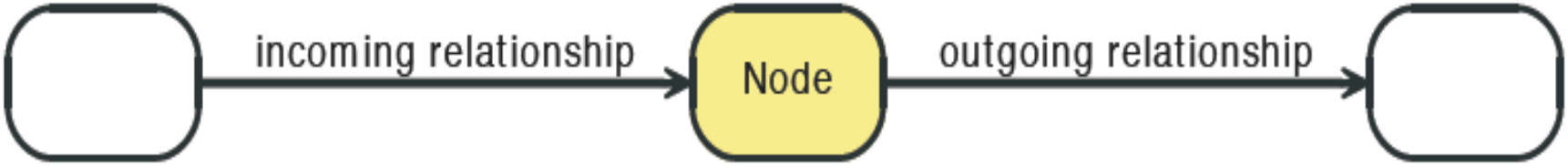
Relationships in Neo4j

- Relationships between nodes are a key part of Neo4j.

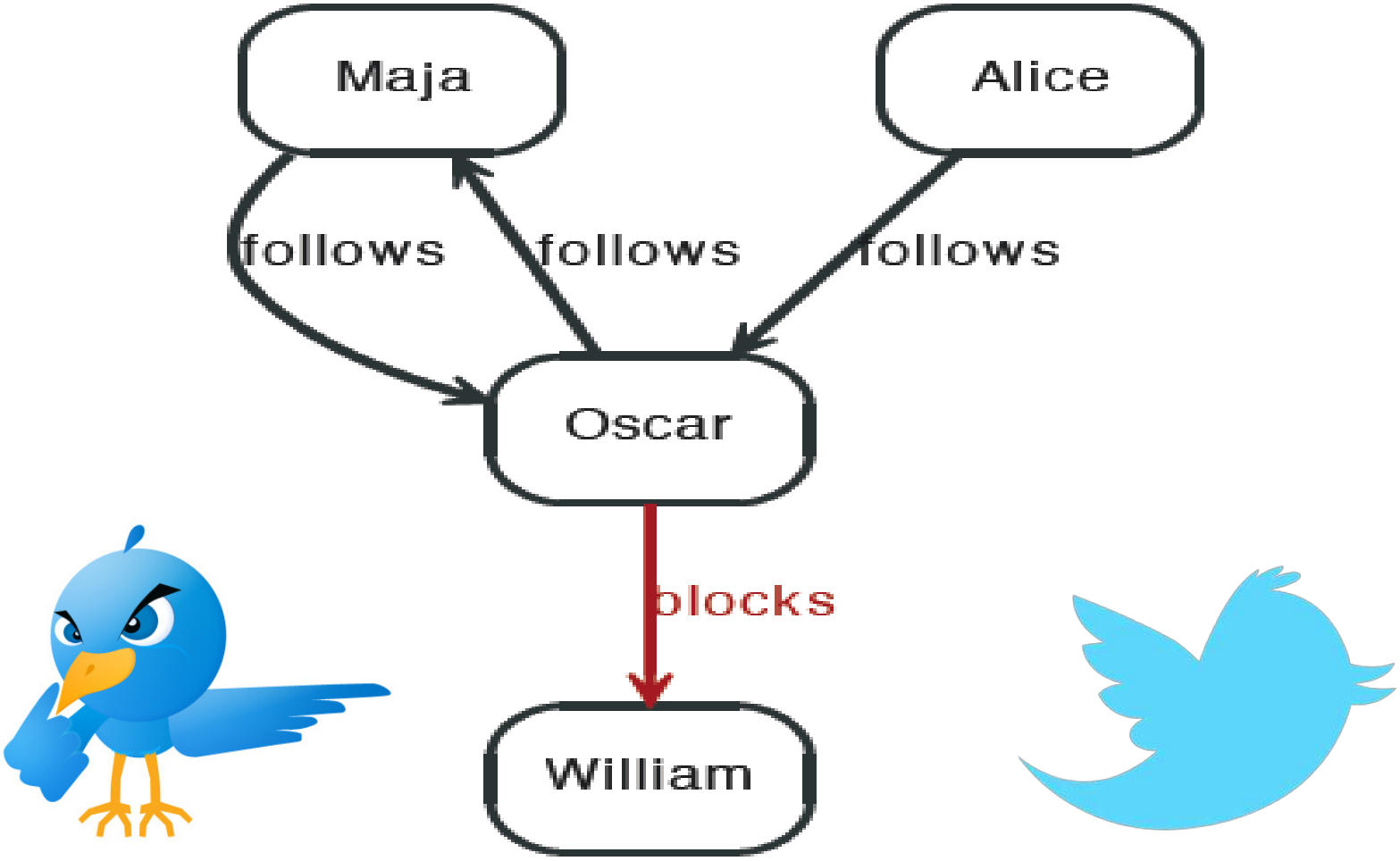


Relationships in Neo4j





Twitter and relationships

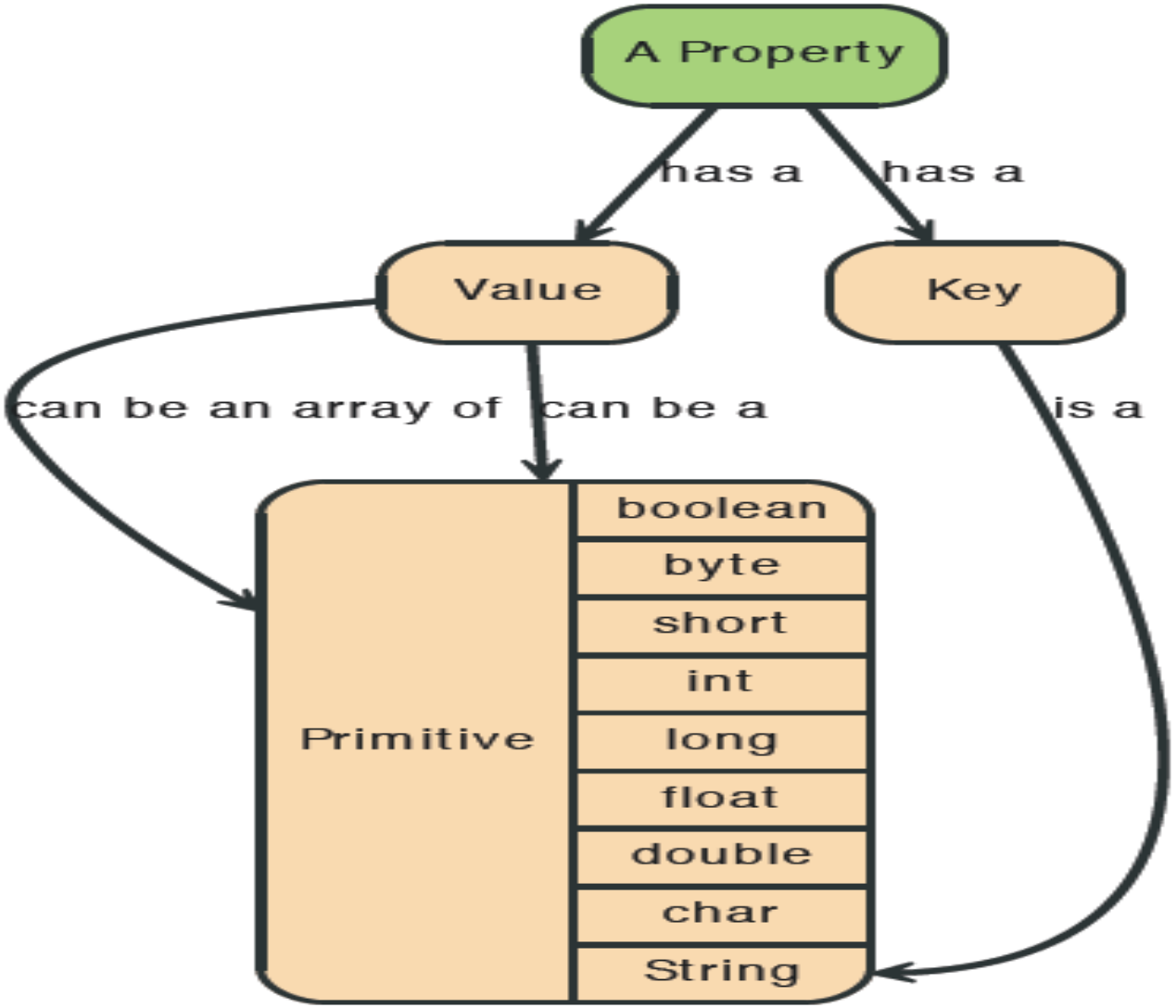


Properties

- Both nodes and relationships can have properties.
- Properties are key-value pairs where the key is a string.
- Property values can be either a primitive or an array of one primitive type.

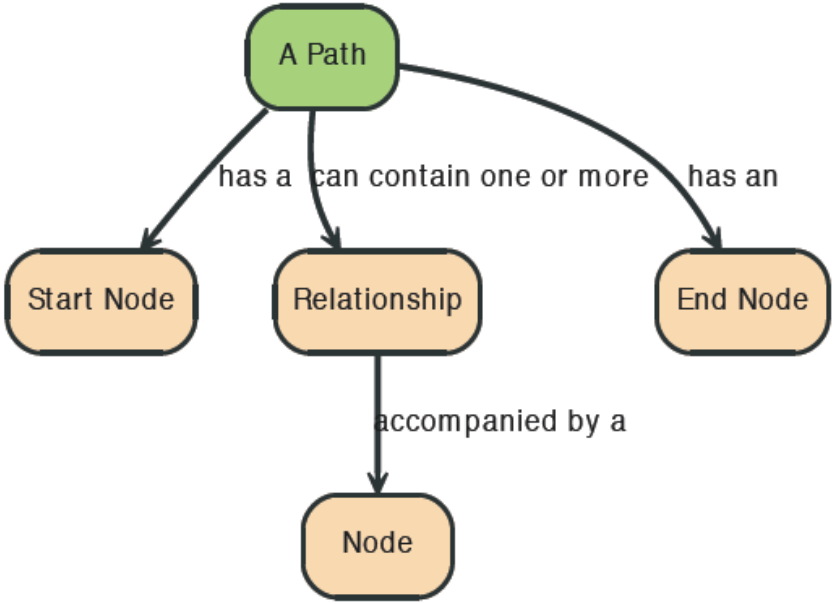
  For example String, int and int[] values are valid for properties.

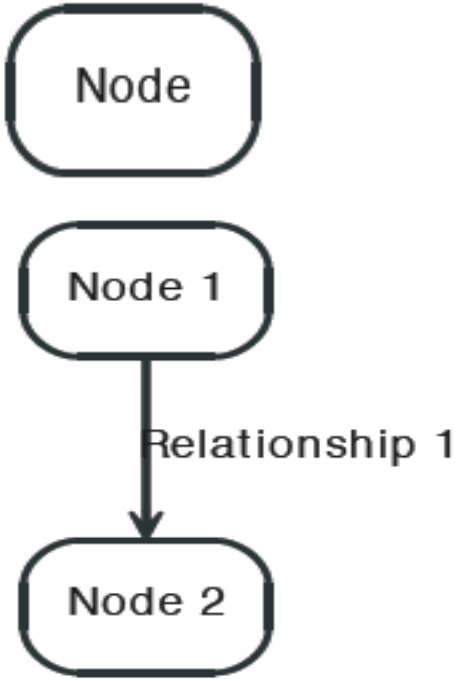
Properties


A Property
has a
has a
Value
Key
can be an array of
can be a
is a
Primitive
boolean
byte
short
int
long
float
double
char
String

Paths in Neo4j

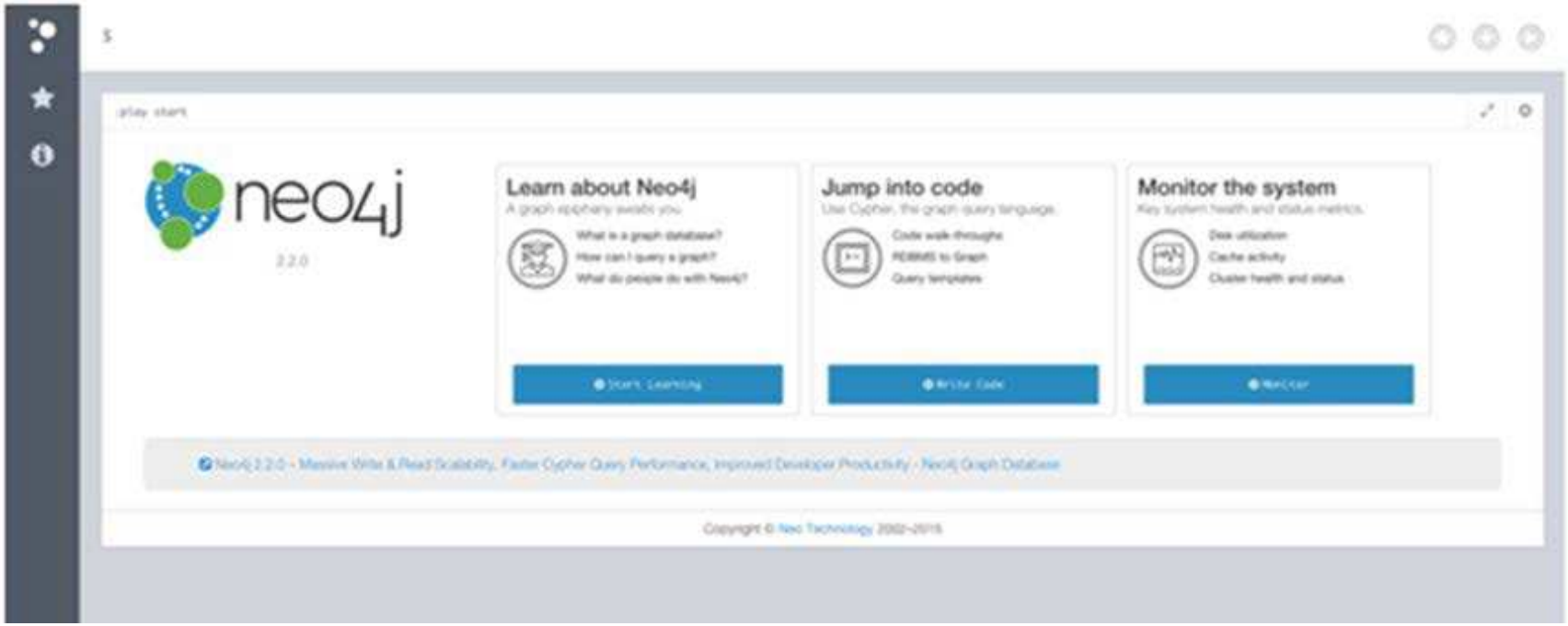
- A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result.





Neo4j browser

After you've downloaded and unzipped the Neo4j package, cd into the Neo4j directory and start up the server like this:
$ bin/neo4j start
To make sure you're up and running, try curling this URL:
$ curl http://localhost:7474/db/data/



Cypher

- Query Language for Neo4j
- Easy to formulate queries based on relationships
- Many features stem from improving on pain points with SQL such as join tables

(Hunger, Michael 2013)

Cypher basic syntax

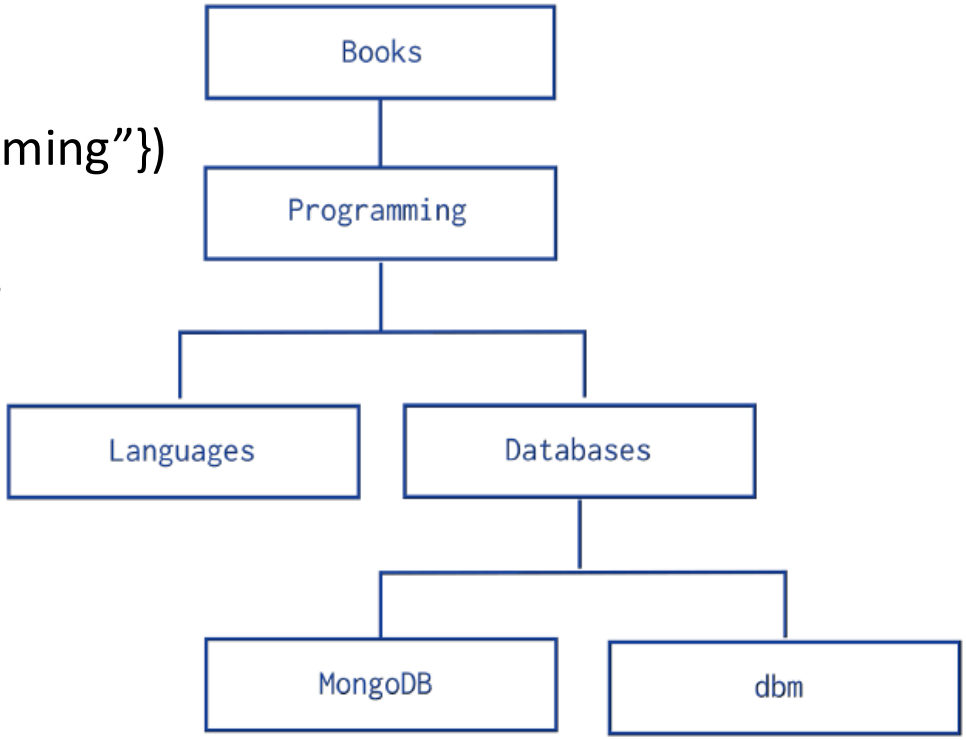
- () Node
- {} Properties
- [] Relationships

- MATCH (n) RETURN n;
- Returns every node in the database

- MATCH (a)-[:CONNECTED]->(b) RETURN a, b;
- Returns all nodes related with the CONNECTED relationship

Collections with Tree-Like Relationships

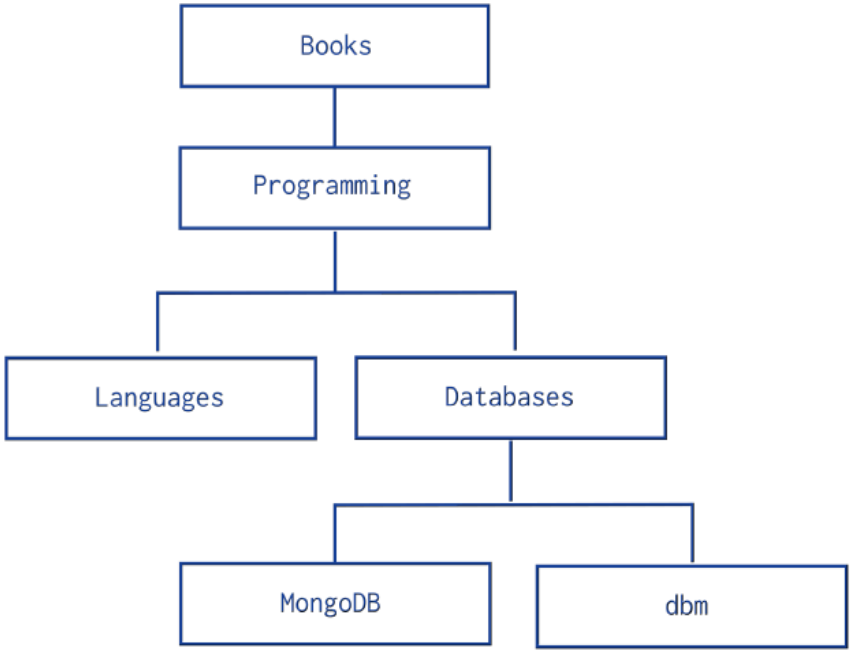
```
Create (c1:Category {name: “Books”})
Create (c2:Category {name: “Programming”})

MATCH (c:Category {name: “Books"}),
(d:Category {name: “Programming"})
CREATE (c)-[p:PARENT_OF]->(d)
```



Collections with Tree-Like Relationships

Given one node, answer queries:

- Report the parent node
  Parent of "Programming"
- Report the children nodes
  Children of "Programming"
- Report the ancestors
  Ancestors of "MongoDB"
- Report the descendants
  Descendants of "Programming"
- Report the siblings
  Siblings of "Databases"



Categories example

Children of “Programming”

```
MATCH (p:Category {name: “Programming”)-[:PARENT_OF]->(c)
RETURN p,c
```

Parent of “Programming”

```
MATCH (p:Category {name: “Programming”)<-[:PARENT_OF]- (c)
RETURN p,c
```

Ancestors of “MongoDB”

```
MATCH (p:Category)-[:PARENT_OF*]->(c:Category{name: “MongoDB”})
RETURN p,c
```

Descendants of “Programming”

```
MATCH (p:Category {name: “Programming”)-[:PARENT_OF*]->(c)
RETURN p,c
```

Siblings of “Databases”

```
MATCH (p:Category {name: "Databases"})<-[r:PARENT_OF]-(c),
(c:Category)-[q:PARENT_OF]->(d)
RETURN d
```

For today…

- Create three nodes:
  - One for you and two other friends
  - Connect them with a FRIEND relationship
  - Query the three nodes, and submit a png of the graph view to ICON

Neo4j Cypher CRUD

Cypher basic syntax

- () Node
- {} Properties
- [] Relationships

- MATCH [some set of nodes and/or relationships]
  WHERE [some set of properties holds]
  RETURN [some set of results captured by the MATCH and WHERE clauses]

- MATCH (n) RETURN n;
  Returns every node in the database

Create

- CREATE (tt:Chocolate {name:“Trinitario Treasure", cocoa: 0.71, broad: “Ghana”, manufacturer: “Jacque Torres”, country: “USA”})
- CREATE (cem:Publication {name: “Chocolate Expert Monthly”})
- CREATE (tt)-[r:review {rating: 2, year: 2006}]->(cem)
- CREATE (tri:Bean {name:”Trinitario”})
- CREATE (tt)-[b:bean_type]->(tri)

Add more nodes

- CREATE (amedei:Company {name: “Amedei”, country: “Italy”})
- CREATE (tb:Chocolate {name:“Toscano Black", cocoa: 0.70})
- CREATE (md:Chocolate {name:“Madagascar", cocoa: 0.70})
- CREATE (blend:Bean {name:”Blend”})
- CREATE (amedei)-[p:produce]->(tb)
- CREATE (amedei)-[p:produce]->(md)
- CREATE (tb)-[b:bean_type]->(blend)

Schemaless Social

- CREATE (p:Person {name: "Alice"})
- MATCH (p:Person {name: "Alice"}),(c:Chocolate {name: “Trinitario Treasure"})
- CREATE (p)-[r:likes]->(c)

- CREATE (p: Person {name: "Tom"})
- MATCH (p:Person {name: "Tom"}),
- (pub:Publication {name: “Chocolate Expert Monthly"})
- CREATE (p)-[r:trusts]->(pub)

- CREATE (p:Person:Critic {name: "Patty"})
- MATCH (p1:Person {name: "Patty"}),(p2:Person {name: "Tom"})
- CREATE (p1)-[r:friends]->(p2)
- CREATE (:Person {name: “Patty”})-[r:friends]->(:Person {name: “Alice”})

Read

- All nodes stored in the database
- MATCH (n) RETURN n;

- Nodes with label “Person” and property “name” with value “Tom”
- MATCH (n:Person { name: “Tom" }) RETURN n;

- All matching nodes that are related, regardless of the direction (notice the lack of an arrow) and returning the nodes from both sides.
- MATCH (a)--(b) RETURN a, b;

- Same as before but returning the relationship
- MATCH (a)-[r]-(b) RETURN a, r, b;

- Nodes connected with relationship friends
- MATCH (a)-[:friends]->(b) RETURN a, b;

Optional Match

- Return the path (instead of individual nodes).
- MATCH p=(a)-[:friends]->(b) RETURN p;

- You can use multiple MATCH clauses in a query :
- MATCH (a:Person {name: ‘Tom'})
- MATCH (b:Person {name: ‘Hanks'})
- RETURN a, b;

- Optional match returns the match if it’s there, if not it’ll return `null`, so the query still works.
- MATCH (a:Person {name: ‘Tom})
- OPTIONAL MATCH (b:Person {name: ‘Hanks'})
- RETURN a, b;

- You can also use the optional flag to return potential relationships for a node. Like:
- MATCH (a:Person {name: ‘Tom'})
- OPTIONAL MATCH (a)-->(x)
- RETURN a, x;
- For all of the `Person` labeled nodes with the `name` of `Chris` both the nodes that do and don’t have relationships will be returned.

Where clause

- Similar to SQL
- MATCH (n:Chocolate) WHERE n.cocoa > .7 RETURN n;

- You can also combine WHERE with AND, OR, and NOT
- MATCH (n:Person)
- WHERE n.age > 18 AND (n.name = 'Chris' OR n.name = “Tom") AND (n)-[:RELATED {relation:"brother"}]-() RETURN n;

- MATCH (n:Chocolate)
- WHERE n.broad = ‘Ghana' OR n.broad IS NULL
- RETURN n
- ORDER BY n.name

- Regular expressions using `=~` followed by the pattern:
- MATCH (n)
- WHERE n.name =~ '(?i)^[a-d].*'
- RETURN n

Update

- MATCH (n { name: ‘Tom' })
- SET n.age = 35
- RETURN n

- MATCH (n { name: ‘Tom' })
- SET n += { username: ‘tomh'}

Delete

- CREATE (e: EphemeralNode {name: "short lived"})
- MATCH (c:Chocolate {name: “Trinitatio Treasure"}),
- (e:EphemeralNode {name: "short lived"})
- CREATE (c)-[r:short_lived_relationship]->(e)

- MATCH ()-[r:short_lived_relationship]-() DELETE r
- MATCH (e:EphemeralNode) DELETE e

- Delete entire graph
- MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n, r

Indexes

- Can help speed up queries

- You can create an index on that type/property combination like this:
- CREATE INDEX ON :Chocolate(name);

- You can easily remove indexes at any time:
- DROP INDEX ON :Chocolate(name);

Constraints

- *Constraints* help sanitize data inputs by preventing writes that don't satisfy a specified criteria
- To ensure that every Chocholate node in the graph have a unique name, for example, you could create this constraint:
- CREATE CONSTRAINT ON (c:Chocolate) ASSERT c.name IS UNIQUE;

- To remove it include the entire constraint statement:
- DROP CONSTRAINT ON (c:Chocolate) ASSERT c.name IS UNIQUE;

- You cannot apply a constraint to a label that already has an index, and if you do create a constraint on a specific label/property pair, an index will be created automatically. So *usually* you'll only need to explicitly create a constraint *or* an index.

Limit and Skip

- MATCH (n)
- RETURN n
- ORDER BY n.name LIMIT 3

- MATCH (n)
- RETURN n
- ORDER BY n.name SKIP 1

- MATCH (n)
- RETURN n
- ORDER BY n.name
- SKIP 1 LIMIT 5

Other functions

- count(*), count(n), count(DISTINCT n)
- length(p)
- type (r)
- id(n), labels(n)
- nodes(p)
- collect – create array with return values
- timestamp()

Unwind

- UNWIND takes collections of nodes, or arrays of data and splits them into individual rows.
- Data must be aliased for the query to work.

- UNWIND ['Chris', 'Kyle', 'Andy', 'Dave', “Tom”] AS x
- RETURN x

- This would return all of the names in individual rows, rather than as a collection as they were passed

For today…

- After following the movies example (excluding the clean up)
  - https://neo4j.com/developer/cypher/guide-cypher-basics/
  - :play movie-graph
- Write a query that shows the movies that Tom Hanks has acted in and the directors that have directed them
- Upload the graph result to ICON
- Clean up

Importing data into Neo4j APOC library

Neo4j conf file

- Stop neo4j
- neo4j.conf file is in the conf directory of your $NEO4J_HOME.

- You can create a new database called transport by editing the option

  `dbms.default_database`
- Uncomment the line and replace neo4j with transport, and save the file.
- Start neo4j again.

- Lots of other configuration options

Many ways of importing data to Neo4j

1.LOAD CSV Cypher command: this command is a great starting point and handles small- to medium-sized data sets (up to 10 million records).

2.neo4j-admin bulk import tool: command line tool useful for straightforward loading of large data sets.

3.Kettle import tool

- Import from CSV using cypher
  - Files can be placed in the import folder within $NEO4J_HOME or hosted on the web
    - On OSX/UNIX, you would need to use the following for a local file “file:///path/to/data.csv” whereas the same url on Windows would be “file:c:/path/to/data.csv”.
  - Load csv with headers from (path) as alias

The transport Graph

- Graph containing a subset of the European road network

Nodes

id	latitude	longitude	population
Utrecht	52.092876	5.104480	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	-0.22219	9642
Doncaster	53.52285	-1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	-0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

Relationships

src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

Merge and With clauses

https://neo4j.com/docs/cypher-manual/current/clauses/merge/

The MERGE clause ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

- Like a combination of MATCH and CREATE

https://neo4j.com/docs/cypher-manual/current/clauses/with/#query-with

The WITH clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

Import data using cypher

- We'll start by loading the nodes:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
place.longitude = toFloat(row.latitude),
place.population = toInteger(row.population)
```

- And now the relationships:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination)
```

Some queries

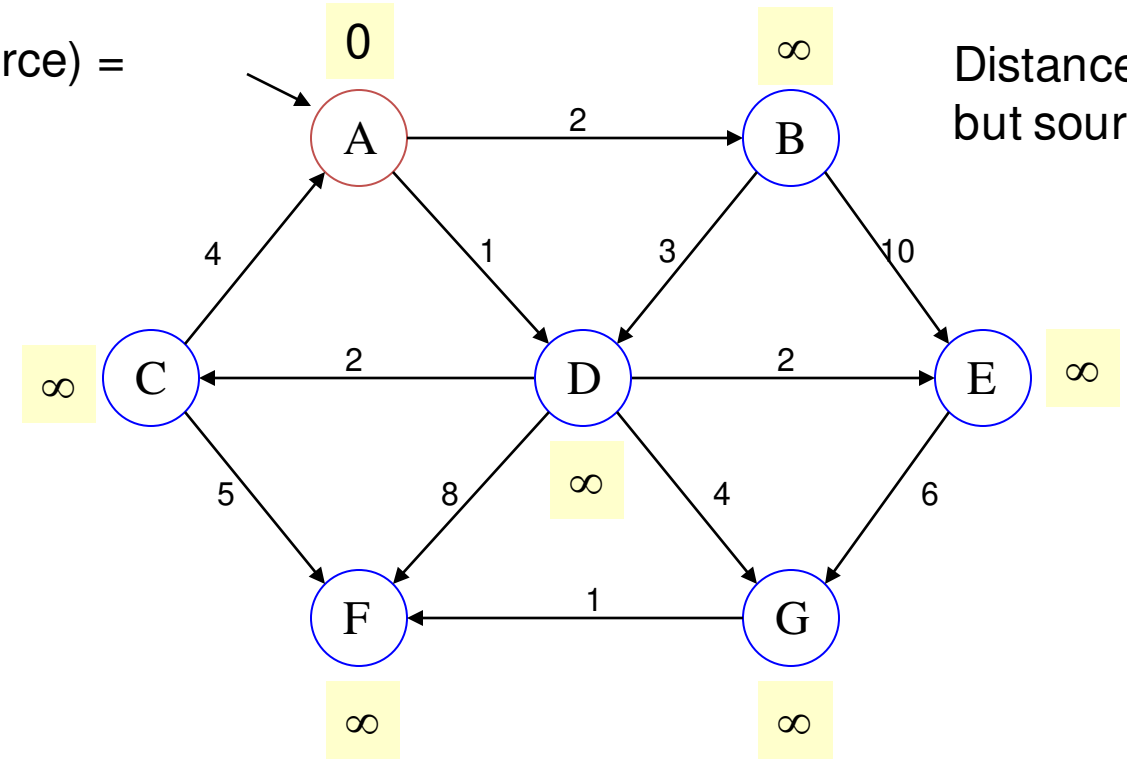
```
MATCH (p:Place)
WITH max(p.population)  as highestPop
MATCH (p2:Place)
where p2.population=highestPop
return p2;

MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"}),
p=shortestPath((source)-[*]-(destination))
RETURN p;
```

Dijkstra pseudocode

```
Dijkstra(v1, v2):
   for each vertex v:                    // Initialization
       v's distance := infinity.
       v's previous := none.
   v1's distance := 0.
   List := {all vertices}.

   while List is not empty:
       v := remove List vertex with minimum distance.
       mark v as known.
       for each unknown neighbor n of v:
           dist := v's distance + edge (v, n)'s weight.

           if dist is smaller than n's distance:
               n's distance := dist.
               n's previous := v.

   reconstruct path from v2 back to v1,
   following previous pointers.
```

Example: Initialization

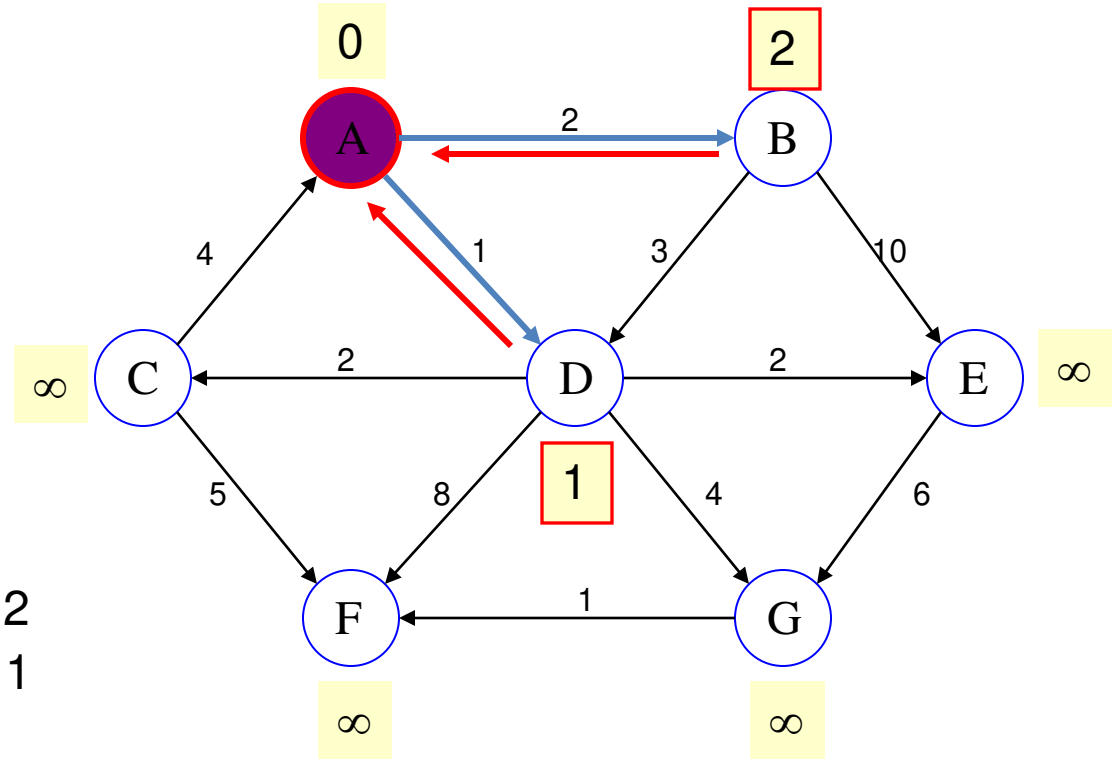
Distance(source) = 0



Distance (all vertices but source) = ∞

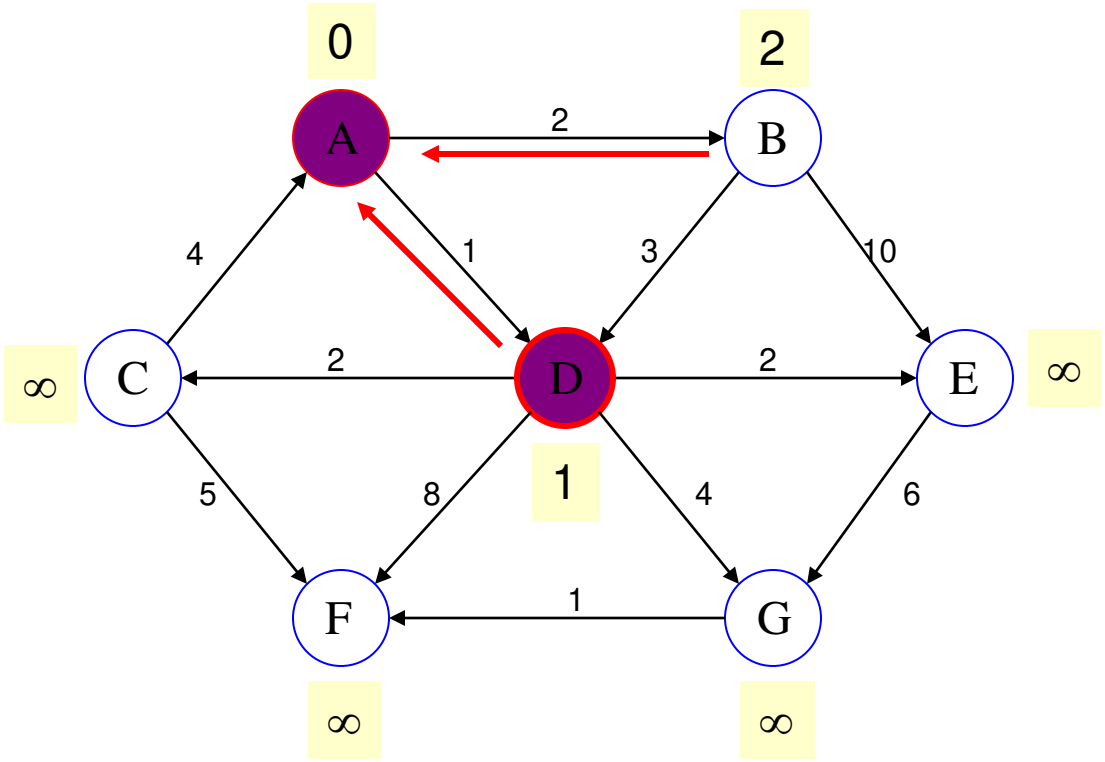
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



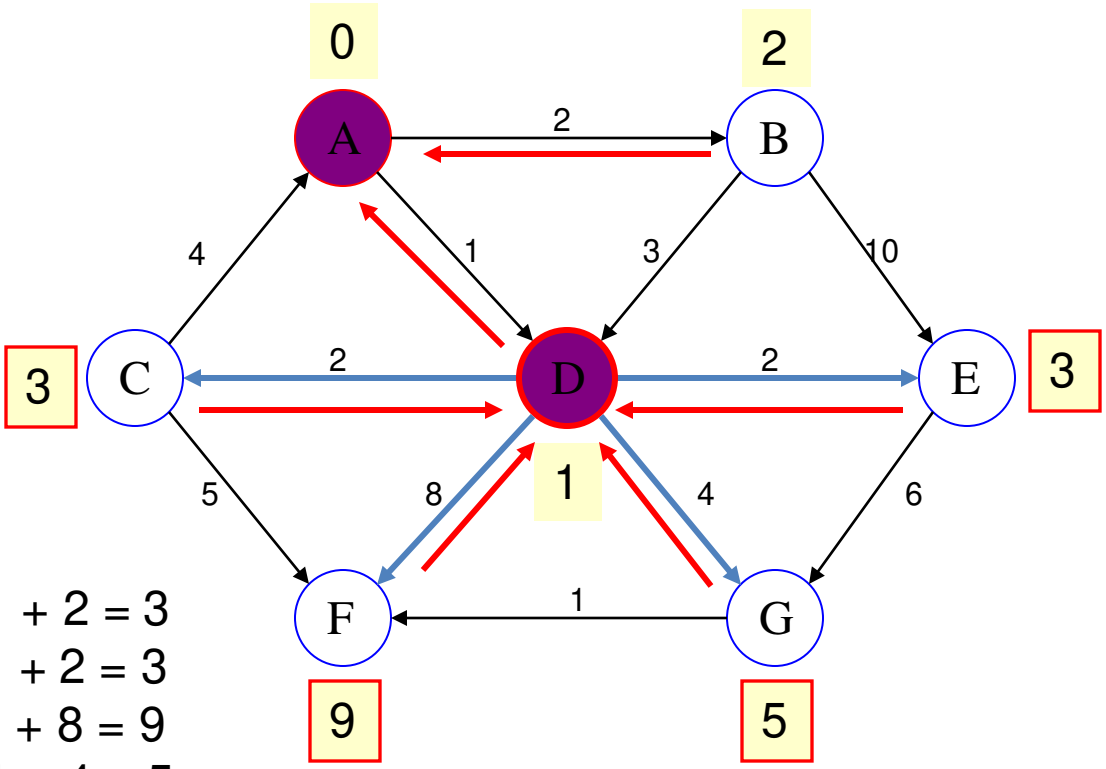
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

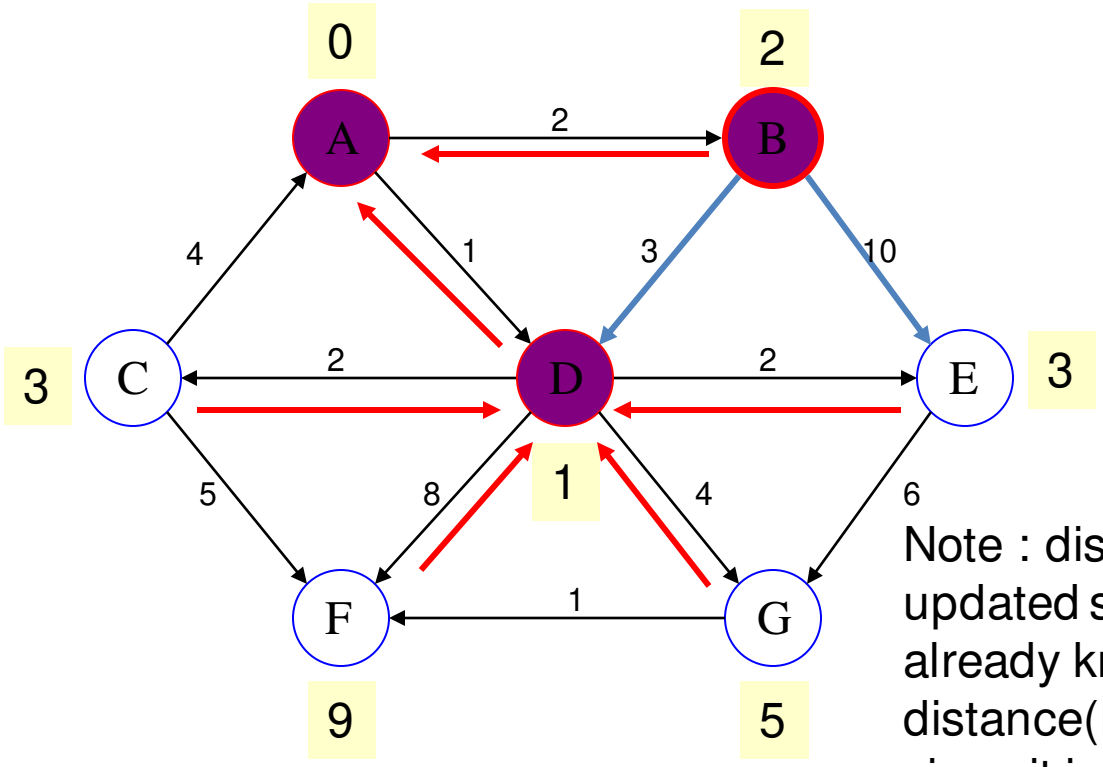
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

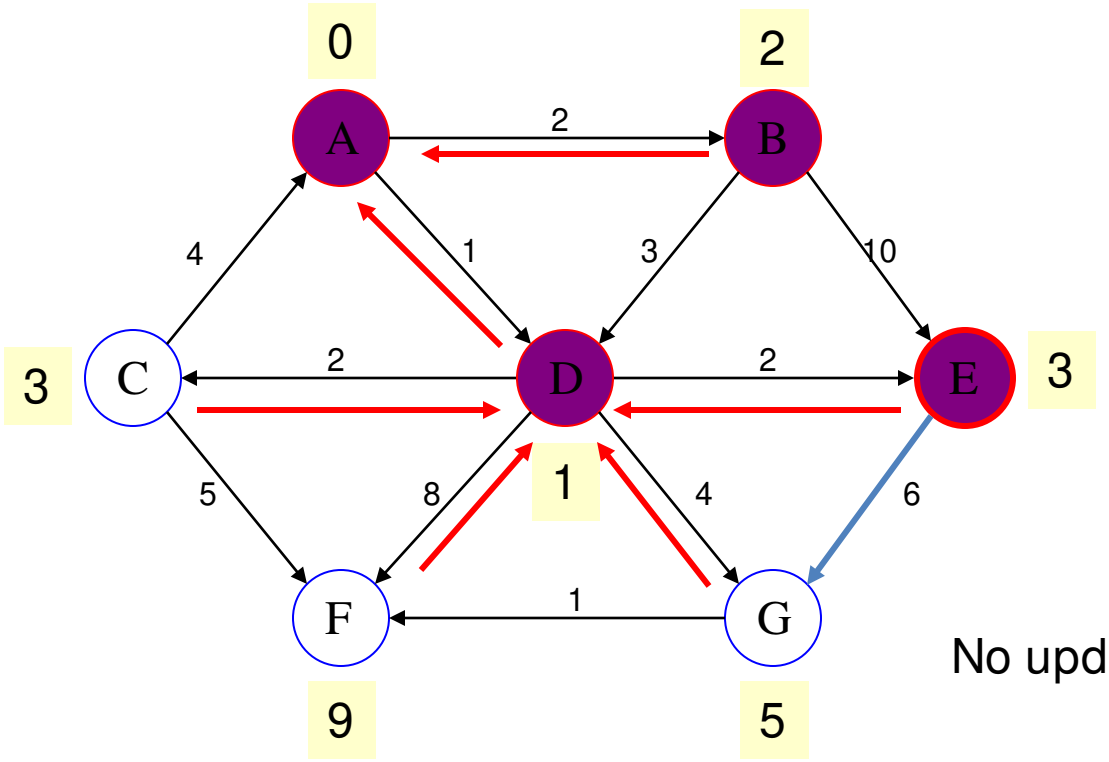
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

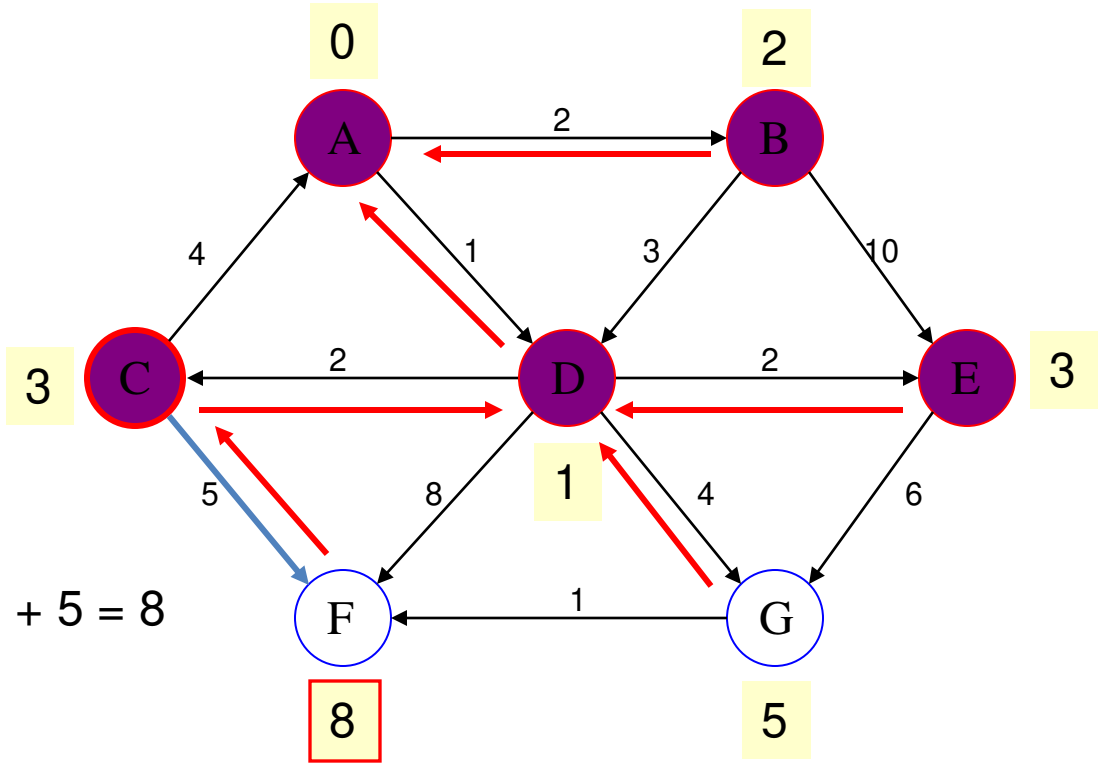
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

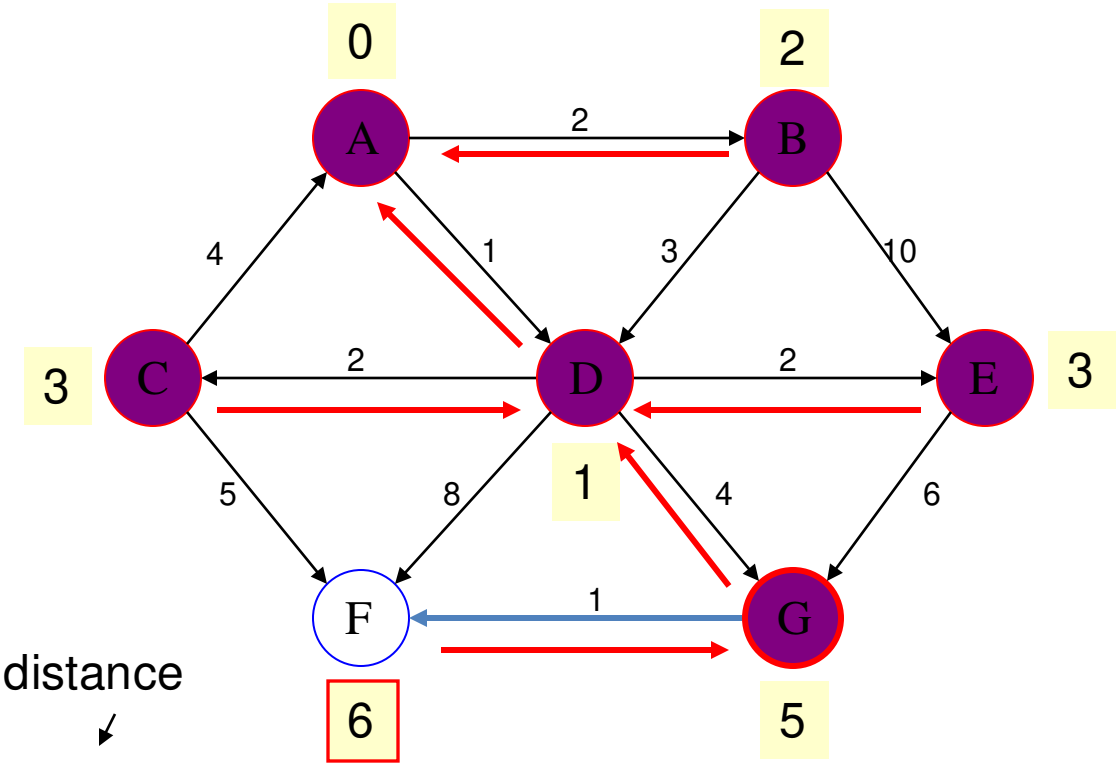
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

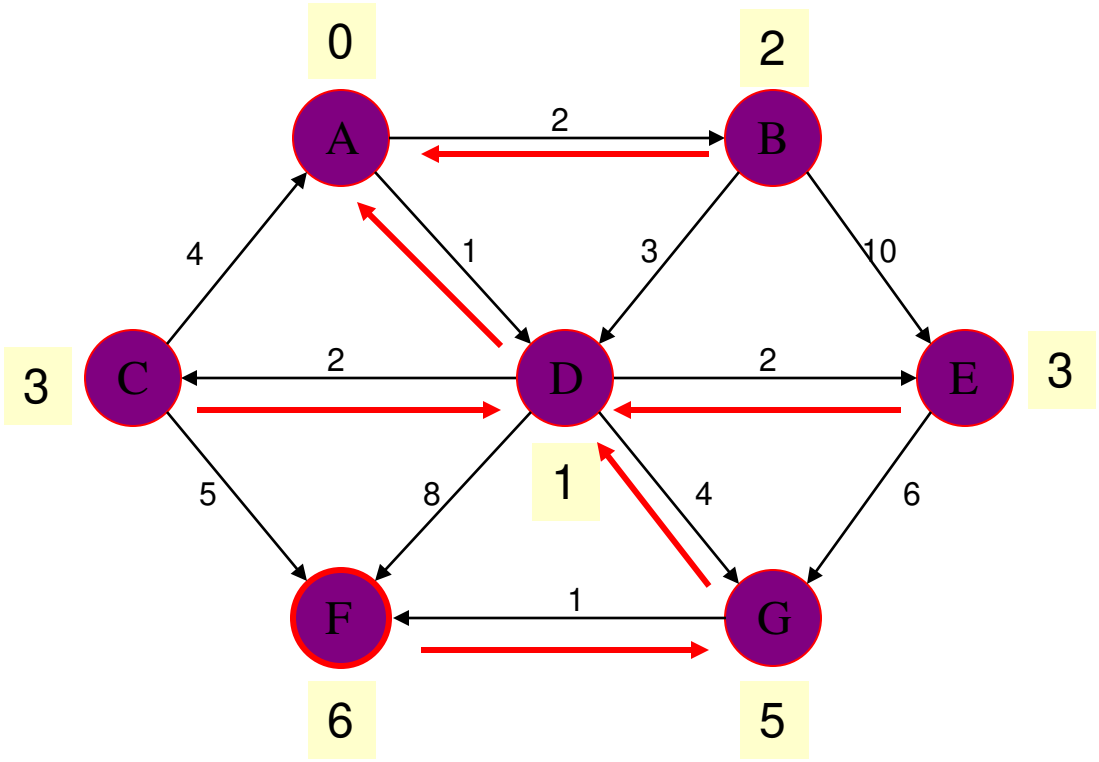
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

Distance(F) = min (8, 5+1) = 6

Example (end)



Pick vertex not in S with lowest cost (F) and update neighbors

APOC library

- https://neo4j.com/labs/apoc/4.1/

  APOC Core can be installed by moving the APOC jar file from the $NEO4J_HOME/labs directory to the $NEO4J_HOME/plugins directory and restarting Neo4j.

- https://neo4j.com/labs/apoc/4.1/import/
- WITH 'https://raw.githubusercontent.com/neo4j-contrib/neo4j-apoc-procedures/4.1/src/test/resources/person.json' AS url CALL apoc.load.json(url) YIELD value as person MERGE (p:Person {name:person.name}) ON CREATE SET p.age = person.age, p.children = size(person.children)

Dijkstra's shortest path

- MATCH (source:Place {id: "Amsterdam"}),
  (destination:Place {id: "London"})
  CALL apoc.algo.dijkstra(source, destination, 'EROAD',
  'distance')
  YIELD path, weight as cost
  RETURN path, cost

- MATCH (source:Place {id: "Amsterdam"}),
  (destination:Place {id: "London"})
  CALL apoc.algo.dijkstraWithDefaultWeight(source,
  destination, 'EROAD', 'distance',10)
  YIELD path, weight as cost
  RETURN path, cost

Install and configure the Graph Data Science Library

On a standalone Neo4j Server, the library will need to be installed and configured manually.

1.Download neo4j-graph-data-science-[version]-standalone.jar from the Neo4j Download Center and copy it into the $NEO4J_HOME/plugins directory.

2.Add the following to your $NEO4J_HOME/conf/neo4j.conf file:
dbms.security.procedures.unrestricted=gds.*

This configuration entry is necessary because the GDS library accesses low-level components of Neo4j to maximize performance.

3.Check if the procedure whitelist is enabled in the $NEO4J_HOME/conf/neo4j.conf file and add the GDS library if necessary:
dbms.security.procedures.whitelist=gds.*

4.Restart Neo4j

For today…

After configuring gds, load the result of this query in the transport database to ICON:

```
MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"})
CALL gds.algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
RETURN gds.algo.getNodeById(nodeId).id AS place, cost
```

Graph algorithms in Neo4j

Examples and data from OReilly's Graph Algorithms Book

(Posted in ICON)

AND https://neo4j.com/docs/graph-data-science/

What is the Graph Data Science Library?

- Open Source Neo4j Add-On for graph analytics
- Provides a set of high performance graph algorithms
  - Community Detection / Clustering (e.g. Label Propagation)
  - Similarity Calculation (e.g. NodeSimilarity)
  - Centrality Algorithms (e.g. PageRank)
  - PathFinding (e.g. Dijkstra)
  - Link Prediction (e.g. AdamicAdar)
  - and more

Local Patterns to Global Computation

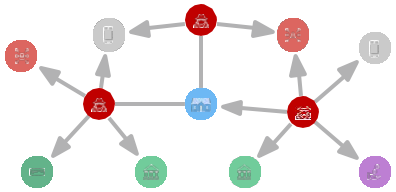
Query (e.g. Cypher/SQL)

Real-time, local decisioning and pattern matching

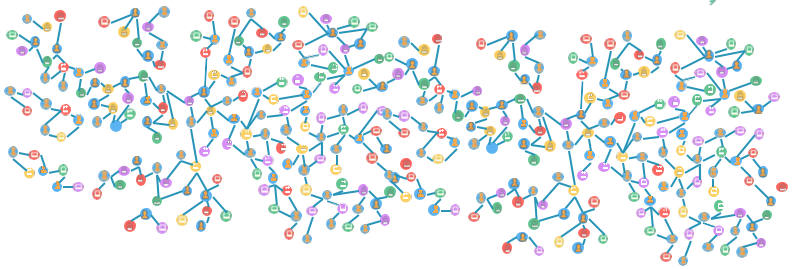
Graph Algorithms Libraries

Global analysis and iterations

Local Patterns → **Global Computation**

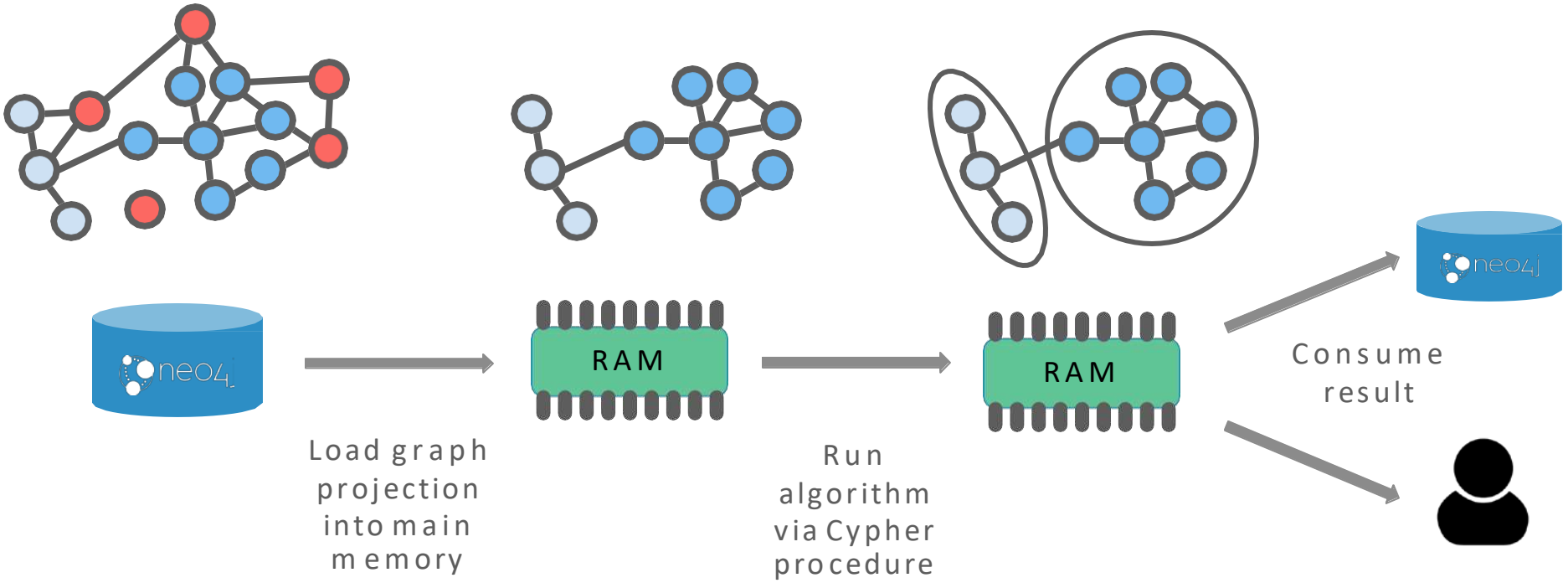


You know what you're looking for and making a decision



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

Workflow


RAM
RAM
Load graph projection into main memory
Run algorithm via Cypher procedure
Consume result

Available Algorithms

Community Detection

- **Label Propagation**
- **Louvain**
- **Weakly Connected Components**
- Triangle Count
- Clustering Coefficients
- Strongly Connected Components
- Balanced Triad (identification)

Centrality / Importance

- **PageRank**
- **Personalized PageRank**
- Degree Centrality
- Closeness Centrality
- Betweenness Centrality
- ArticleRank
- Eigenvector Centrality

Similarity

- **Node Similarity**
- Euclidean Distance
- Cosine Similarity
- Overlap Similarity
- Pearson Similarity

Link Prediction

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

Pathfinding & Search

- Parallel Breadth First Search
- Parallel Depth First Search
- Shortest Path
- Minimum Spanning Tree
- A* Shortest Path
- Yen's K Shortest Path
- K-Spanning Tree (MST)
- Random Walk

GDS - Algo Syntax

```
CALL gds.<algo-name>.<mode>(
  graphName: STRING,
  configuration: MAP
)
```

Available Modes:

- `stream`: streams results back to the user.
- `write`: writes results to the Neo4j database and returns a summary of the results.
- `stats`: runs the algorithm and only reports statistics.
- `mutate:` writes results to the in-memory graph and returns a summary of the results.

```
CALL gds.wcc.write(  "got-
 interactions",
 {
  writeProperty: "component",
  consecutiveIds: true
 }
) YIELDS writeMillis, componentCount
```

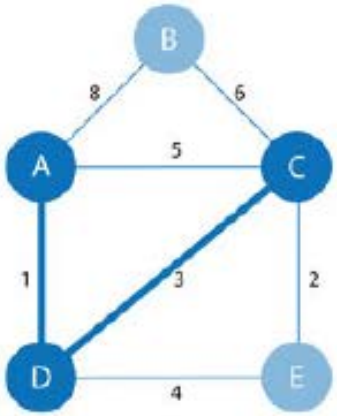
```
CALL gds.wcc.stream("got-
 interactions",
 {}
) YIELDS nodeId, componentId
```

Graph algorithms

- Implemented in the Neo4j Graphs library
- Three main categories:
  - Pathfinding and search
    - Shortest path, minimum spanning trees
  - Centrality computation
    - Node importance
  - Community detection
    - Connectness

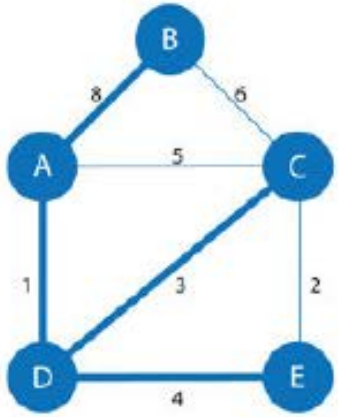
Pathfinding algorithms

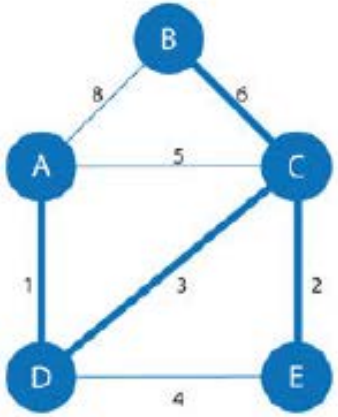
Pathfinding Algorithms



Calculated by relationship weights

(A, B) = 8
(A, C) = 4 via D
(A, D) = 1
(A, E) = 5 via D
(B, C) = 6
(B, D) = 9 via A or C
And so on...





Shortest Path

Shortest path between 2 nodes (A to C shown)

All-Pairs Shortest Paths

Optimized calculations for shortest paths from **all** nodes to **all** other nodes

Single Source Shortest Path

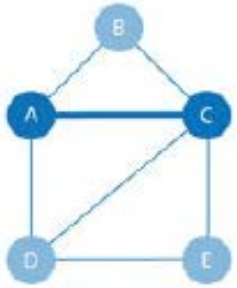
Shortest path from a **root** node (A shown) to **all** other nodes

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

Minimum Spanning Tree

Shortest path **connecting all nodes** (A start shown)

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



Calculated by number of hops

Pathfinding Example: The transport Graph

- Graph containing a subset of the European road network

Nodes

id	latitude	longitude	population
Utrecht	52.092876	5.104480	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	-0.22219	9642
Doncaster	53.52285	-1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	-0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

Relationships

src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

Import data using cypher (last class)

- We'll start by loading the nodes:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
place.longitude = toFloat(row.latitude),
place.population = toInteger(row.population)
```

- And now the relationships:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination)
```

Shortest Path (Dijkstra's alg)

- No weights

```
MATCH (source:Place {id: "Amsterdam"}), (destination:Place {id: "London"})
CALL gds.alpha.shortestPath.stream({
startNode: source,endNode: destination,    nodeProjection: "*",
relationshipProjection: {all: { type: "*", properties: "distance",
orientation: "UNDIRECTED“ }}    })
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).id AS place, cost;
```

- Weighted by distance add relationshipWeightProperty: "distance“ to the relationshipProjection

Minimum Spanning Tree

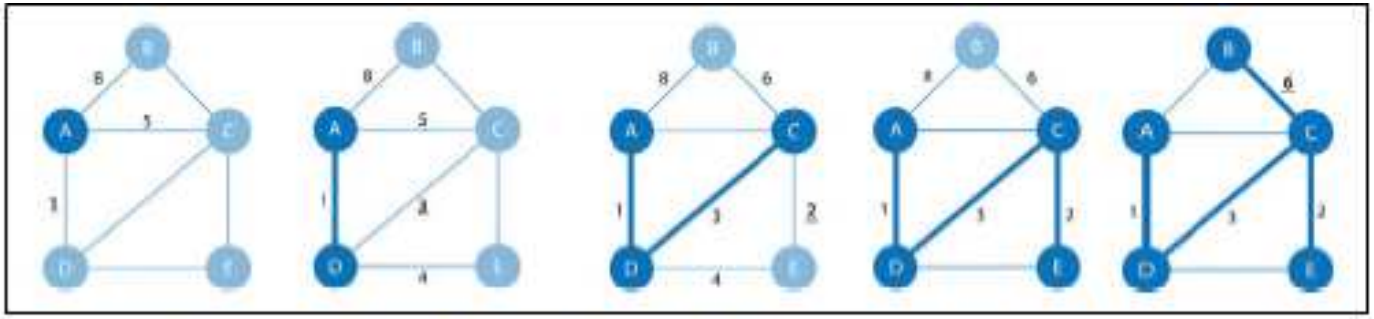


Figure 4-10. The steps of the Minimum Spanning Tree algorithm

```
MATCH (n:Place {id:"Amsterdam"})
CALL gds.alpha.spanningTree.minimum.write({
startNodeId: id(n), nodeProjection: "*",       relationshipProjection: {EROAD: {type: "EROAD",
properties: "distance",
orientation: "UNDIRECTED“   }      },     relationshipWeightProperty: "distance",
writeProperty: 'MINST’,  weightWriteProperty: 'cost’   })
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

Result stored in the graph. Retrieve the minimum weight spanning tree from the graph

Centrality algorithms

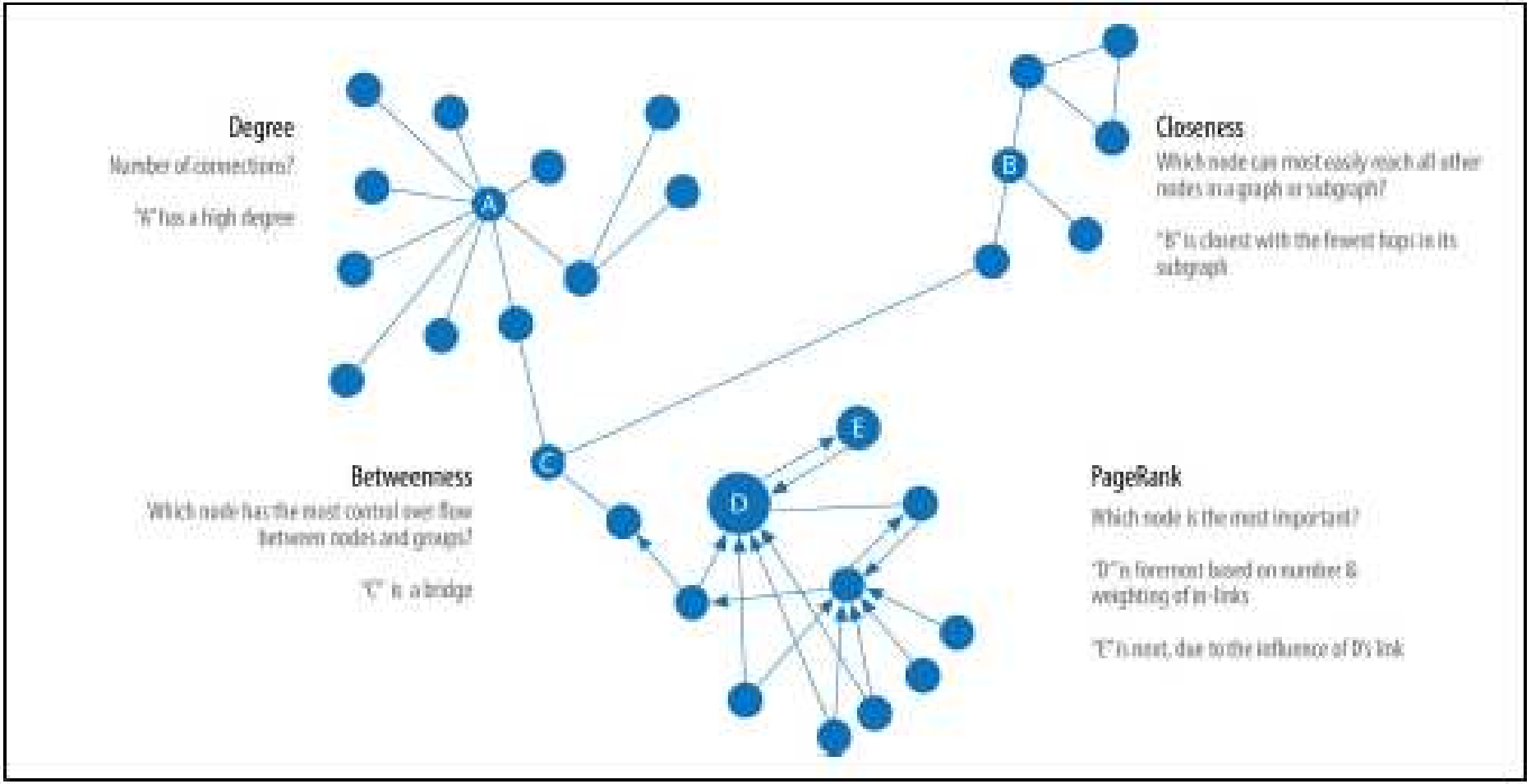


Figure 5-1. Representative centrality algorithms and the types of questions they answer

Centrality Examples: The social graph

- Small Twitter-like graph

id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

src	dst	relationship
Alice	Bridget	FOLLOWS
Alice	Charles	FOLLOWS
Mark	Doug	FOLLOWS
Bridget	Michael	FOLLOWS
Doug	Mark	FOLLOWS
Michael	Alice	FOLLOWS
Alice	Michael	FOLLOWS
Bridget	Alice	FOLLOWS
Michael	Bridget	FOLLOWS

Import social graph

- The following query imports nodes:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/social-nodes.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- MERGE (:User {id: row.id});
- And this query imports relationships:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/social-relationships.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- **MATCH** (source:User {id: row.src})
- **MATCH** (destination:User {id: row.dst})
- MERGE (source)-[:FOLLOWS]->(destination);

Closeness Centrality

- Neo4j's implementation of Closeness Centrality uses the following formula:

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

- where:
- • *u* is a node.
- • *n* is the number of nodes in the same component (subgraph or group) as *u*.
- • *d(u,v)* is the shortest-path distance between another node *v* and *u*.

Betweenness Centrality

- A way of detecting the amount of influence a node has over the flow of information or resources in a graph

$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$$

- *u* is a node.
- *p* is the total number of shortest paths between nodes *s* and *t*.
- *p(u)* is the number of shortest paths between nodes *s* and *t* that pass through node *u*.

PageRank

- PageRank is defined in the original Google paper as follows:

$$PR(u) = (1 - d) + d\left(\frac{PR(T1)}{C(T1)} + \ldots + \frac{PR(Tn)}{C(Tn)}\right)$$

- where:
- • We assume that a page *u* has citations from pages *T1* to *Tn*.
- • *d* is a damping factor which is set between 0 and 1. It is usually set to 0.85. You
- can think of this as the probability that a user will continue clicking. This helps
- minimize rank sink
- • *1-d* is the probability that a node is reached directly without following any relationships.
- • *C(Tn)* is defined as the out-degree of a node *T*.

Social Graph

- MATCH p=(:User)-[:FOLLOWS]->(:User) RETURN p;
- CALL gds.graph.create('nameOfGraph','NodeLabel','RelationshipType’)
- CALL gds.graph.create(‘pagerank_example’, ‘User’, ‘FOLLOWS’);

pageRank Stream

- CALL gds.pageRank.stream('pagerank_example', {maxIterations: 20, dampingFactor: 0.85})
  YIELD nodeId, score
  RETURN gds.util.asNode(nodeId).id AS name, score
  ORDER BY score DESC LIMIT 10

Write back the results

- CALL gds.<algorithm>.write({writeProperty:'pagerank’})
- CALL gds.pageRank.write(‘pagerank_example’, {maxIterations: 20, dampingFactor: 0.85, writeProperty:’pageRank’})

- CALL gds.graph.drop(‘pagerank_example’);

PageRank

- *PageRank is named after Google cofounder Larry Page, who created it to rank websites in Google's search results. The basic assumption is that a page with more incoming and more influential incoming links is more likely a credible source. PageRank measures the number and quality of incoming relationships to a node to determine an estimation of how important that node is. Nodes with more sway over a network are presumed to have more incoming relationships from other influential nodes.*

PageRank

- CALL algo.pageRank.stream('User', 'FOLLOWS', {iterations:20, dampingFactor:0.85})
- YIELD nodeId, score
- **RETURN** algo.getNodeById(nodeId).id **AS** page, score
- **ORDER BY** score **DESC**

Community detection algorithms

Algorithm type	What it does	Example use
Triangle Count and Clustering Coefficient	Measures how many nodes form triangles and the degree to which nodes tend to cluster together	Estimating group stability and whether the network might exhibit "small-world" behaviors seen in graphs with tightly knit clusters
Strongly Connected Components	Finds groups where each node is reachable from every other node in that same group *following the direction* of relationships	Making product recommendations based on group affiliation or similar items
Connected Components	Finds groups where each node is reachable from every other node in that same group, *regardless of the direction* of relationships	Performing fast grouping for other algorithms and identify islands
Label Propagation	Infers clusters by spreading labels based on neighborhood majorities	Understanding consensus in social communities or finding dangerous combinations of possible co-prescribed drugs
Louvain Modularity	Maximizes the presumed accuracy of groupings by comparing relationship weights and densities to a defined estimate or average	In fraud analysis, evaluating whether a group has just a few discrete bad behaviors or is acting as a fraud ring

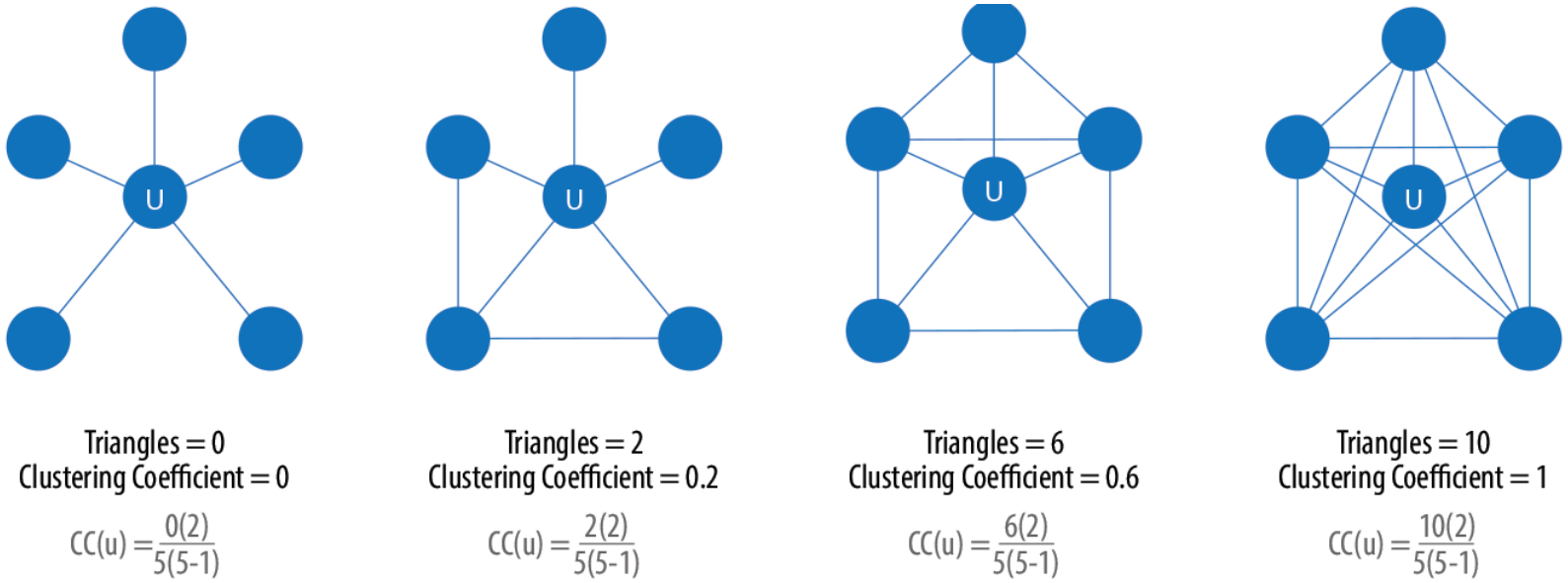
Louvain Example

- https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/

The software graph

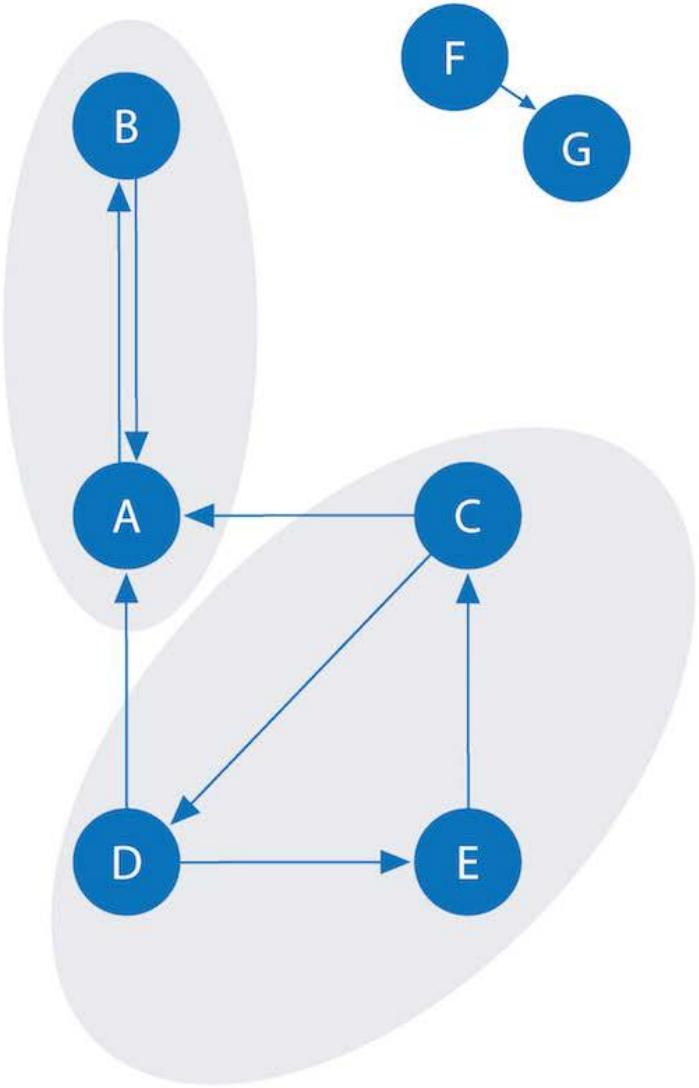
- The following query imports the nodes:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/sw-nodes.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- MERGE (:Library {id: row.id});
- And this imports the relationships:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/sw-relationships.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- **MATCH** (source:Library {id: row.src})
- **MATCH** (destination:Library {id: row.dst})
- MERGE (source)-[:DEPENDS_ON]->(destination);

Local Clustering Coefficient




```
CALL gds.localClusteringCoefficient.stream({
nodeProjection: "Library",
relationshipProjection: {
DEPENDS_ON: {type: "DEPENDS_ON", orientation: "UNDIRECTED“ } } })
YIELD nodeId, localClusteringCoefficient
WHERE localClusteringCoefficient > 0
RETURN gds.util.asNode(nodeId).id AS library, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC;
```

Strongly connected components



Strongly Connected Components

Sets where all nodes can reach all other nodes in both directions, but not necessarily directly.

2 sets of strongly connected components are shown shaded : {A,B,} and {C,D,E}

Note that in {C,D,E} each node can reach the others, but in some cases they must go through another node first.

Connected components

- There exist a path connecting the nodes

```
CALL gds.wcc.stream({
nodeProjection: "Library",
relationshipProjection: "DEPENDS_ON"
})
YIELD nodeId, componentId
RETURN componentId, collect(gds.util.asNode(nodeId).id) AS libraries

ORDER BY size(libraries) DESC;
```

For today…

- To the software dependency graph, add the node naibr with the following dependencies (numpy, scipy, and matplotlib) using the merge command
- Run the label propagation (https://neo4j.com/docs/graph-data-science/current/algorithms/label-propagation/ or Graph Algorithms book page. 139)
- Upload the label propagation results ignoring the direction of the relationship to ICON

More algorithms in Neo4j

Examples and data from OReilly's Graph Algorithms Book

(Posted in ICON)

AND https://neo4j.com/docs/graph-data-science/

Available Algorithms

Community Detection

- **Label Propagation**
- **Louvain**
- **Weakly Connected Components**
- Triangle Count
- Clustering Coefficients
- Strongly Connected Components
- Balanced Triad (identification)

Centrality / Importance

- **PageRank**
- **Personalized PageRank**
- Degree Centrality
- Closeness Centrality
- Betweenness Centrality
- ArticleRank
- Eigenvector Centrality

Similarity

- **Node Similarity**
- Euclidean Distance
- Cosine Similarity
- Overlap Similarity
- Pearson Similarity

Link Prediction

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

Pathfinding & Search

- Parallel Breadth First Search
- Parallel Depth First Search
- Shortest Path
- Minimum Spanning Tree
- A* Shortest Path
- Yen's K Shortest Path
- K-Spanning Tree (MST)
- Random Walk

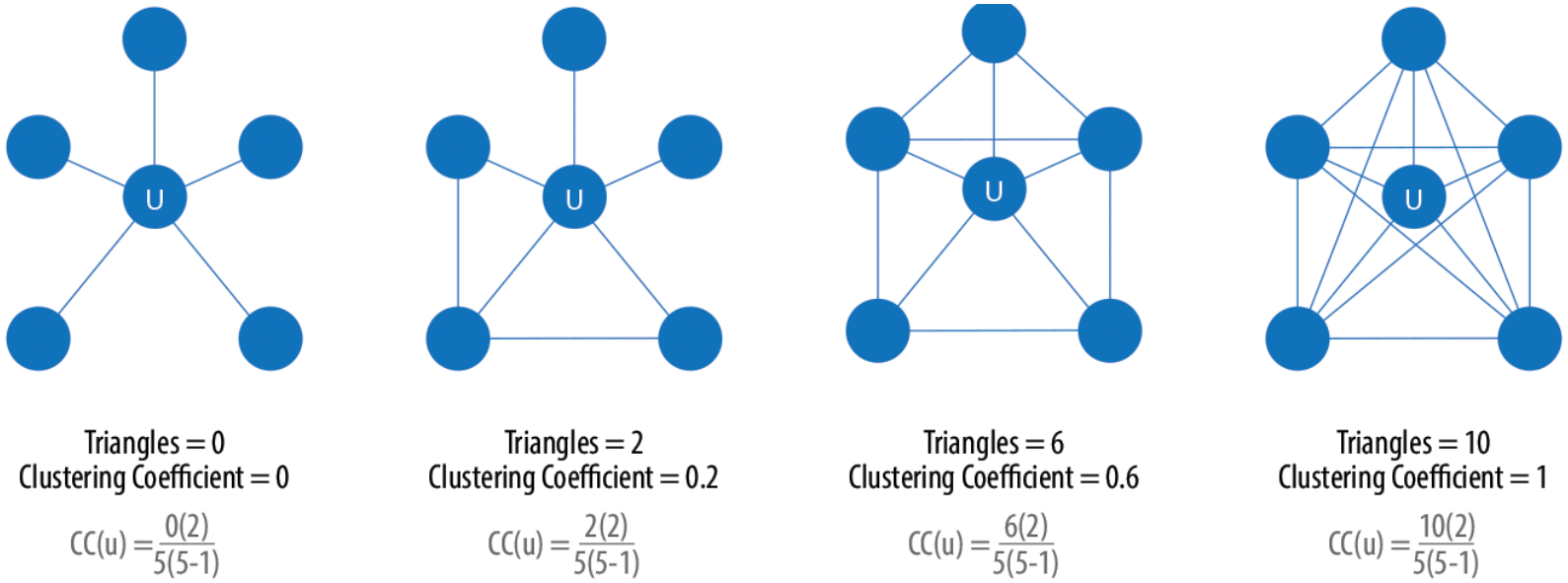
Community detection algorithms

Algorithm type	What it does	Example use
Triangle Count and Clustering Coefficient	Measures how many nodes form triangles and the degree to which nodes tend to cluster together	Estimating group stability and whether the network might exhibit "small-world" behaviors seen in graphs with tightly knit clusters
Strongly Connected Components	Finds groups where each node is reachable from every other node in that same group *following the direction* of relationships	Making product recommendations based on group affiliation or similar items
Connected Components	Finds groups where each node is reachable from every other node in that same group, *regardless of the direction* of relationships	Performing fast grouping for other algorithms and identify islands
Label Propagation	Infers clusters by spreading labels based on neighborhood majorities	Understanding consensus in social communities or finding dangerous combinations of possible co-prescribed drugs
Louvain Modularity	Maximizes the presumed accuracy of groupings by comparing relationship weights and densities to a defined estimate or average	In fraud analysis, evaluating whether a group has just a few discrete bad behaviors or is acting as a fraud ring

The software graph

- The following query imports the nodes:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/sw-nodes.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- MERGE (:Library {id: row.id});
- And this imports the relationships:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/sw-relationships.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- **MATCH** (source:Library {id: row.src})
- **MATCH** (destination:Library {id: row.dst})
- MERGE (source)-[:DEPENDS_ON]->(destination);

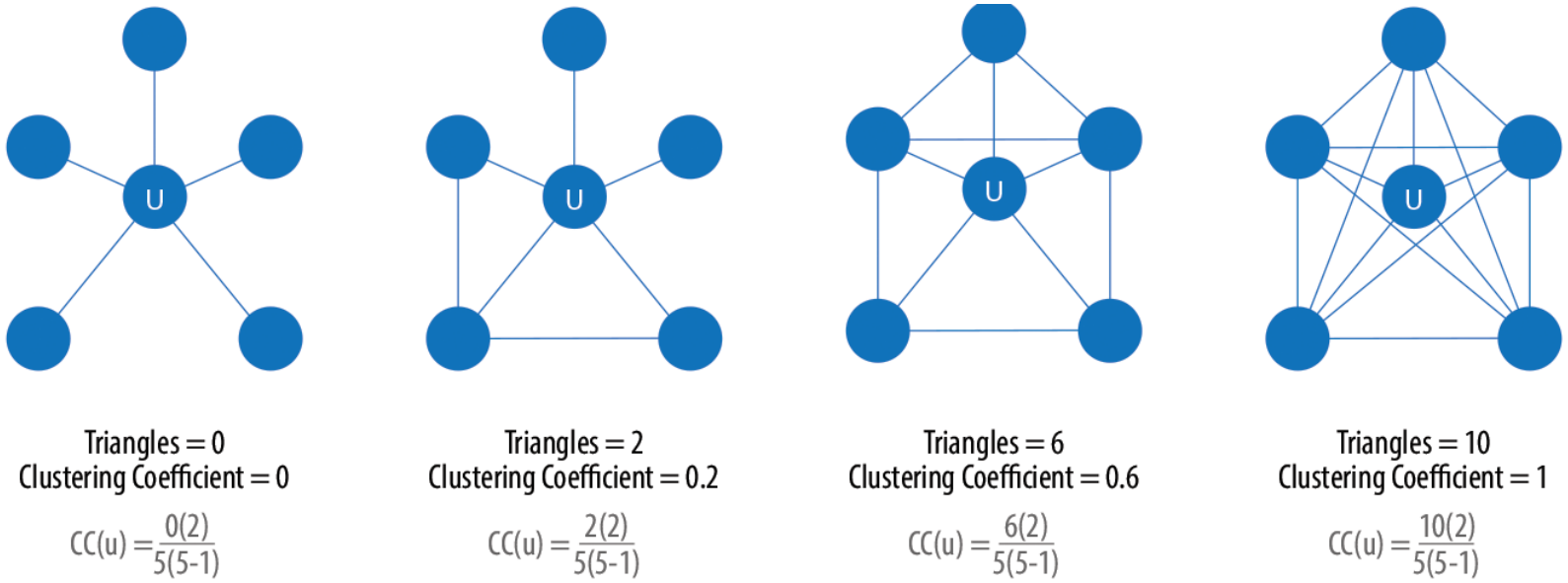
Local Clustering Coefficient



The local clustering coefficient C_n of a node n describes the likelihood that the neighbors of n are also connected. To compute C_n we use the number of triangles a node is a part of T_n, and the degree of the node d_n. The formula to compute the local clustering coefficient is as follows:

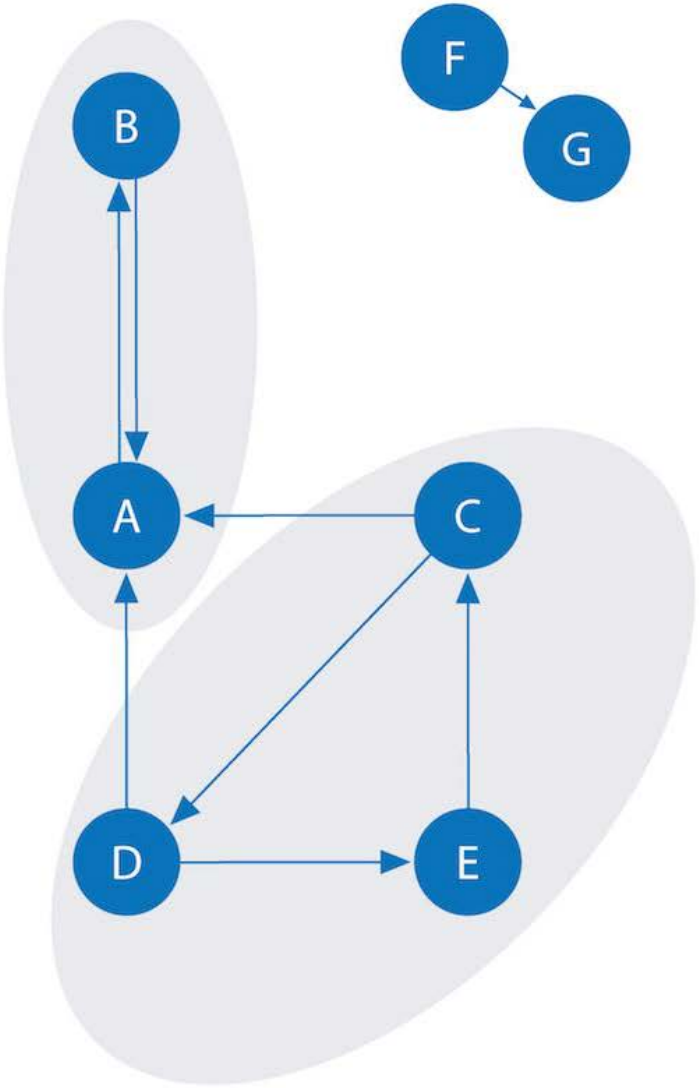
$$C_n = \frac{2T_n}{d_n(d_n - 1)}$$

Local Clustering Coefficient




```
CALL gds.localClusteringCoefficient.stream({
nodeProjection: "Library",
relationshipProjection: {
DEPENDS_ON: {type: "DEPENDS_ON", orientation: "UNDIRECTED“ } } })
YIELD nodeId, localClusteringCoefficient
WHERE localClusteringCoefficient > 0
RETURN gds.util.asNode(nodeId).id AS library, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC;
```

Strongly connected components



Strongly Connected Components

Sets where all nodes can reach all other nodes in both directions, but not necessarily directly.

2 sets of strongly connected components are shown shaded : {A,B,} and {C,D,E}

Note that in {C,D,E} each node can reach the others, but in some cases they must go through another node first.

Weakly Connected components

- There exist a path connecting the nodes

```
CALL gds.wcc.stream({
nodeProjection: "Library",
relationshipProjection: "DEPENDS_ON"
}) YIELD nodeId, componentId
RETURN componentId, collect(gds.util.asNode(nodeId).id) AS libraries
ORDER BY size(libraries) DESC;
```

Louvain Example

- https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/

Recommendations

Search

Recommendations

Items

Products, web sites, blogs, news items, …

Examples:

amazon.com

PANDORA

StumbleUpon

del.icio.us

NETFLIX

movielens
helping you find the *right* movies

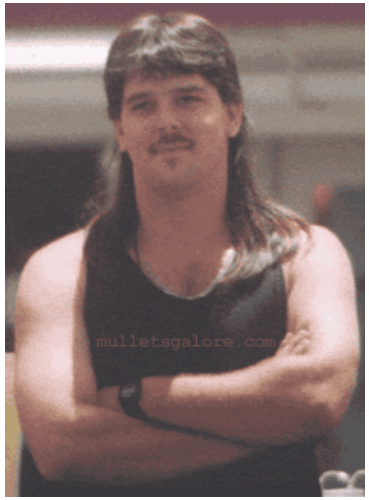
last·fm
the social music revolution

Google News

YouTube

XBOX LIVE

Example: Recommender Systems



- **Customer X**
  - Buys Metallica CD
  - Buys Megadeth CD
- **Customer Y**
  - Does search on Metallica
  - Recommender system suggests Megadeth from data collected about customer **X**

How **Into Thin Air** made **Touching the Void** a bestseller: http://www.wired.com/wired/archive/12.10/tail.html

Formal Model

- ***X*** = set of **Customers**
- ***S*** = set of **Items**

- **Utility function *u*: *X* × *S* → *R***
  - ***R*** = set of ratings
  - ***R*** is a totally ordered set
  - e.g., **0-5** stars, real number in **[0,1]**

Utility Matrix

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2		1	
David				0.4

Three approaches

- **Three approaches to recommender systems:**
  - **1)** Content-based
  - **2)** Collaborative
  - **3)** Latent factor based

Content-based Recommendations

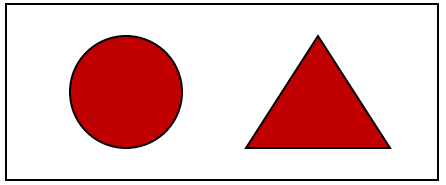
- **Main idea:** Recommend items to customer ***x*** similar to previous items rated highly by ***x***

Example:

- **Movie recommendations**
  - Recommend movies with same actor(s), director, genre, …
- **Websites, blogs, news**
  - Recommend other sites with "similar" content

Plan of Action

likes



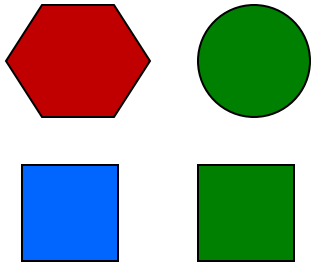
Item profiles

build

Red
Circles
Triangles

User profile

match



recommend

Item Profiles

- For each item, create an **item profile**
- **Profile is a set (vector) of features**
  - **Movies:** author, title, actor, director,…
  - **Text:** Set of "important" words in document
- **How to pick important features?**
  - Usual heuristic from text mining is **TF-IDF** (Term frequency * Inverse Doc Frequency)

  Doc profile = set of words with highest **TF-IDF** scores, together with their scores

User Profiles and Prediction

- **User profile possibilities:**
  - Weighted average of rated item profiles
  - **Variation:** weight by difference from average rating for item
  - …
- **Prediction heuristic:**
  - Given user profile $\boldsymbol{x}$ and item profile $\boldsymbol{i}$, estimate

$$u(\boldsymbol{x}, \boldsymbol{i}) = \cos(\boldsymbol{x}, \boldsymbol{i}) = \frac{\boldsymbol{x} \cdot \boldsymbol{i}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{i}||}$$

Pros: Content-based Approach

- **+: No need for data on other users**
  - No cold-start or sparsity problems
- **+: Able to recommend to users with unique tastes**
- **+: Able to recommend new & unpopular items**
  - No first-rater problem
- **+: Able to provide explanations**
  - Can provide explanations of recommended items by listing content-features that caused an item to be recommended

Cons: Content-based Approach

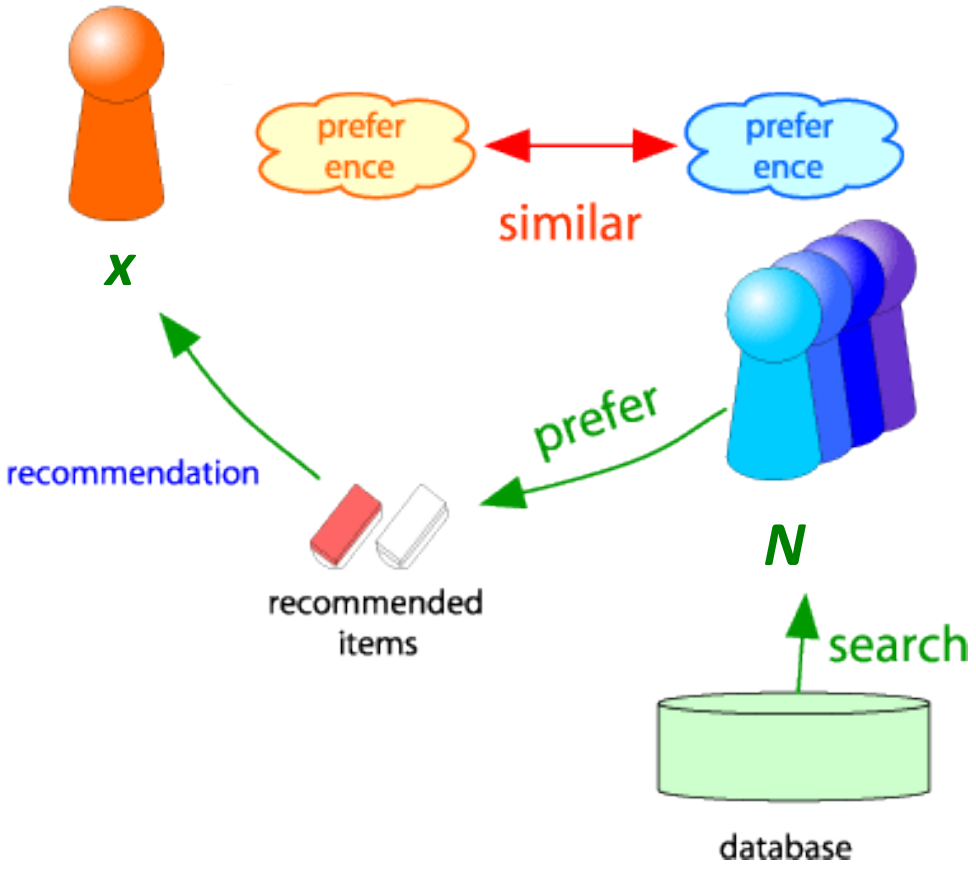
- **–: Finding the appropriate features is hard**
  - E.g., images, movies, music
- **–: Recommendations for new users**
  - **How to build a user profile?**
- **–: Overspecialization**
  - Never recommends items outside user's content profile
  - People might have multiple interests
  - **Unable to exploit quality judgments of other users**

Collaborative Filtering

Harnessing quality judgments of other users

Collaborative Filtering

- Consider user ***x***
- Find set ***N*** of other users whose ratings are "**similar**" to ***x***'s ratings
- Estimate ***x***'s ratings based on ratings of users in ***N***



Finding "Similar" Users

$r_x = [*, \_, \_, *, ***]$
$r_y = [*, \_, **, **, \_]$

- Let r_x be the vector of user x's ratings
- **Jaccard similarity measure**
  - **Problem:** Ignores the value of the rating

r_x, r_y as sets:
$r_x = \{1, 4, 5\}$
$r_y = \{1, 3, 4\}$

Finding "Similar" Users

$r_x = [*, \_, \_, *, ***]$
$r_y = [*, \_, **, **, \_]$

- Let r_x be the vector of user x's ratings
- **Jaccard similarity measure**
  - **Problem:** Ignores the value of the rating
- **Cosine similarity measure**
  - $\text{sim}(x, y) = \cos(r_x, r_y) = \frac{r_x \cdot r_y}{||r_x|| \cdot ||r_y||}$
  - **Problem:** Treats missing ratings as "negative"
    - Solution: only consider items rated by both x and y
    - Solution: subtract the (row) mean (cosine == correlation)

r_x, r_y as sets:
$r_x = \{1, 4, 5\}$
$r_y = \{1, 3, 4\}$

r_x, r_y as points:
$r_x = \{1, 0, 0, 1, 3\}$
$r_y = \{1, 0, 2, 2, 0\}$

Finding "Similar" Users

$r_x = [*, \_, \_, *, ***]$
$r_y = [*, \_, **, **, \_]$

- Let r_x be the vector of user x's ratings
- **Jaccard similarity measure**
  - **Problem:** Ignores the value of the rating
- **Cosine similarity measure**
  - $\text{sim}(x, y) = \cos(r_x, r_y) = \frac{r_x \cdot r_y}{||r_x|| \cdot ||r_y||}$
  - **Problem:** Treats missing ratings as "negative"
- **Pearson correlation coefficient**
  - S_{xy} = items rated by both users x and y

$$sim(x, y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \overline{r_x})(r_{ys} - \overline{r_y})}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \overline{r_x})^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \overline{r_y})^2}}$$

r_x, r_y as sets:
$r_x = \{1, 4, 5\}$
$r_y = \{1, 3, 4\}$

r_x, r_y as points:
$r_x = \{1, 0, 0, 1, 3\}$
$r_y = \{1, 0, 2, 2, 0\}$

$\overline{r_x}$, $\overline{r_y}$ … avg. rating of **x**, **y**

Rating Predictions

From similarity metric to recommendations:

- Let r_x be the vector of user x's ratings
- Let N be the set of k users most similar to x who have rated item i
- **Prediction for item *s* of *user x*:**
  - $r_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$
  - $r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}}$
  - Other options?
- **Many other tricks possible…**

Shorthand:
$s_{xy} = sim(x, y)$

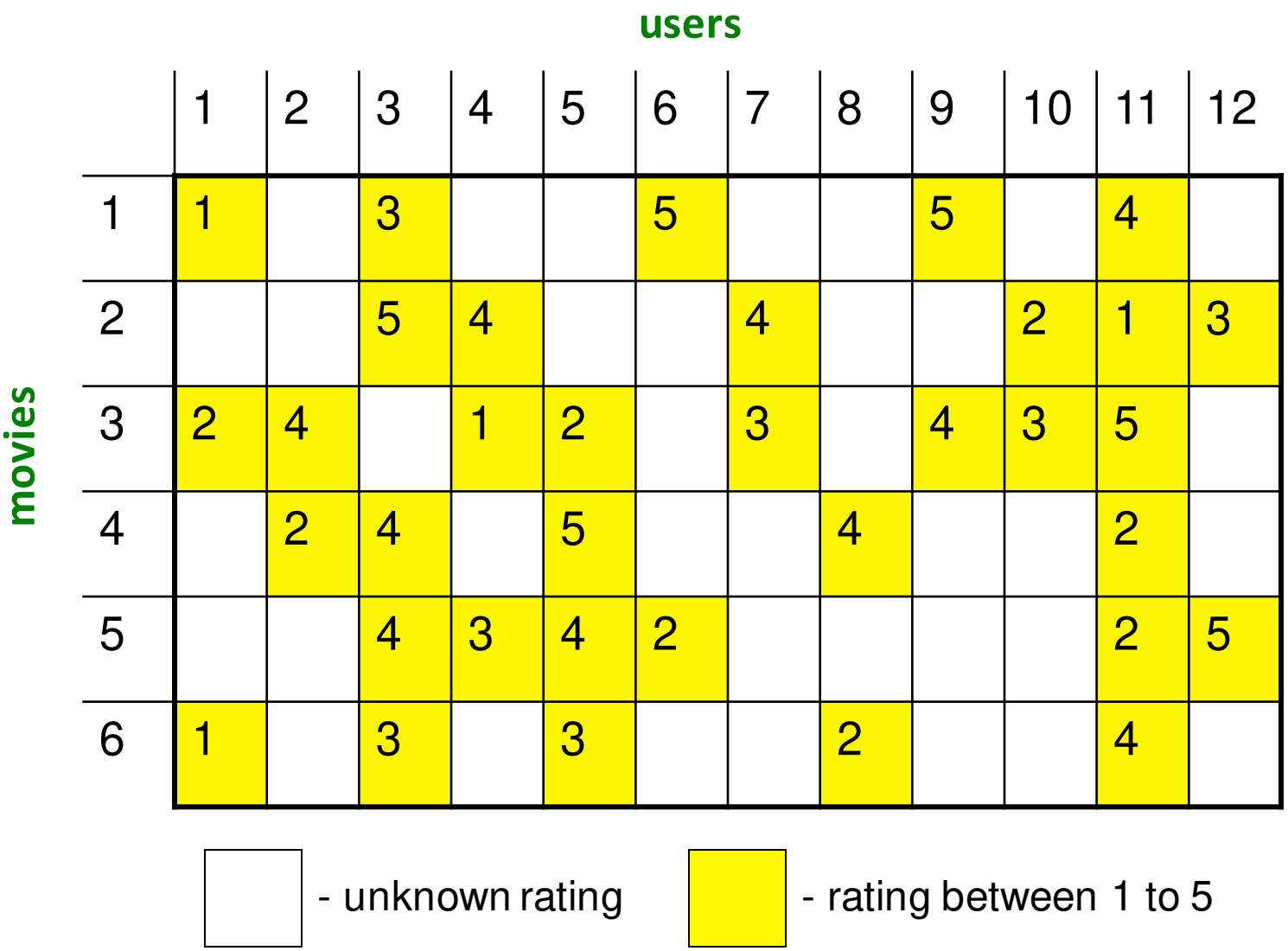
Item-Item Collaborative Filtering

- **So far: User-user collaborative filtering**
- **Another view: Item-item**
  - For item ***i***, find other similar items
  - Estimate rating for item ***i*** based on ratings for similar items
  - Can use same similarity metrics and prediction functions as in user-user model

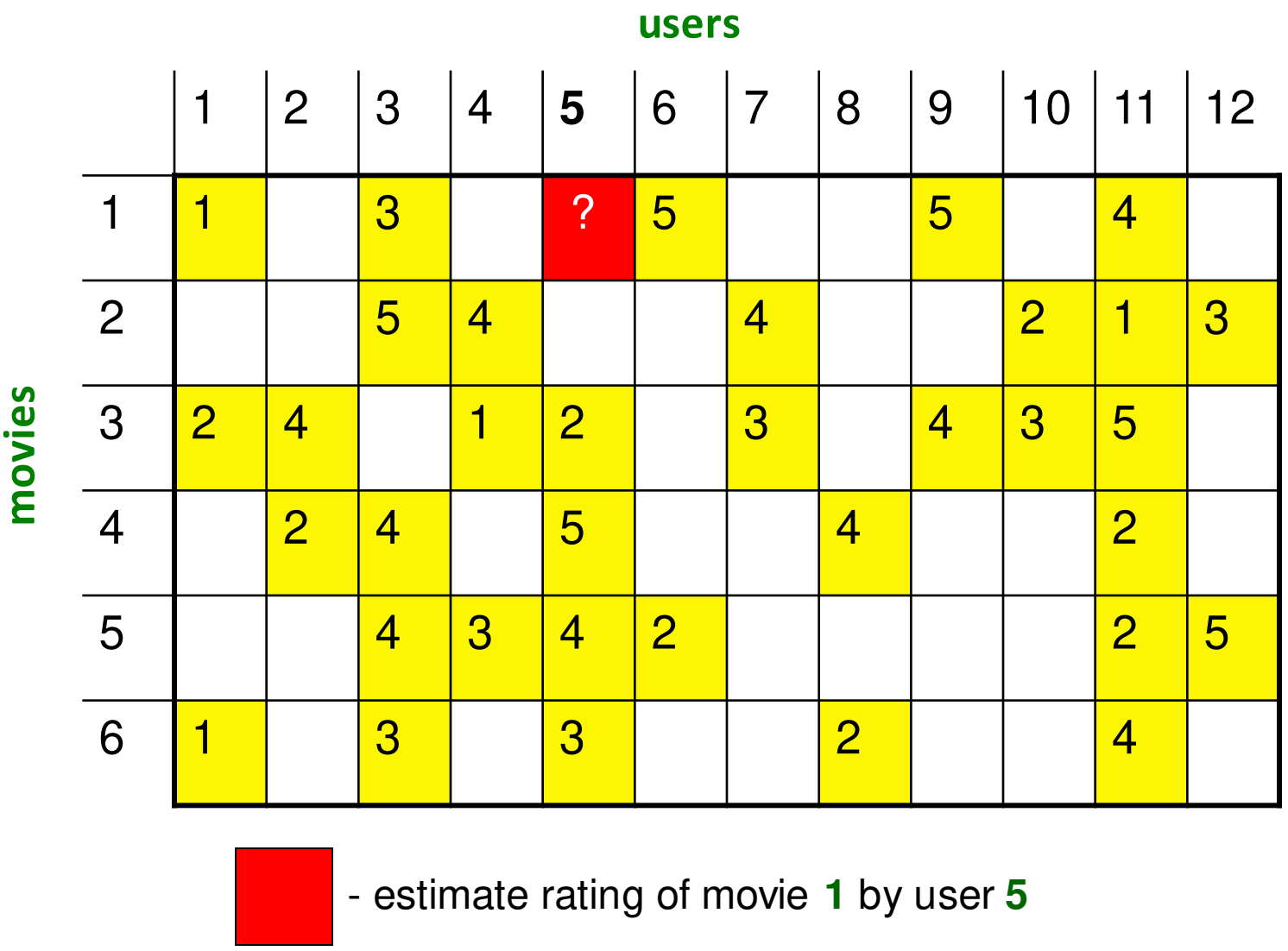
$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

s_{ij}… similarity of items ***i*** and ***j***
r_{xj}…rating of user ***u*** on item ***j***
$N(i;x)$… set items rated by ***x*** similar to ***i***

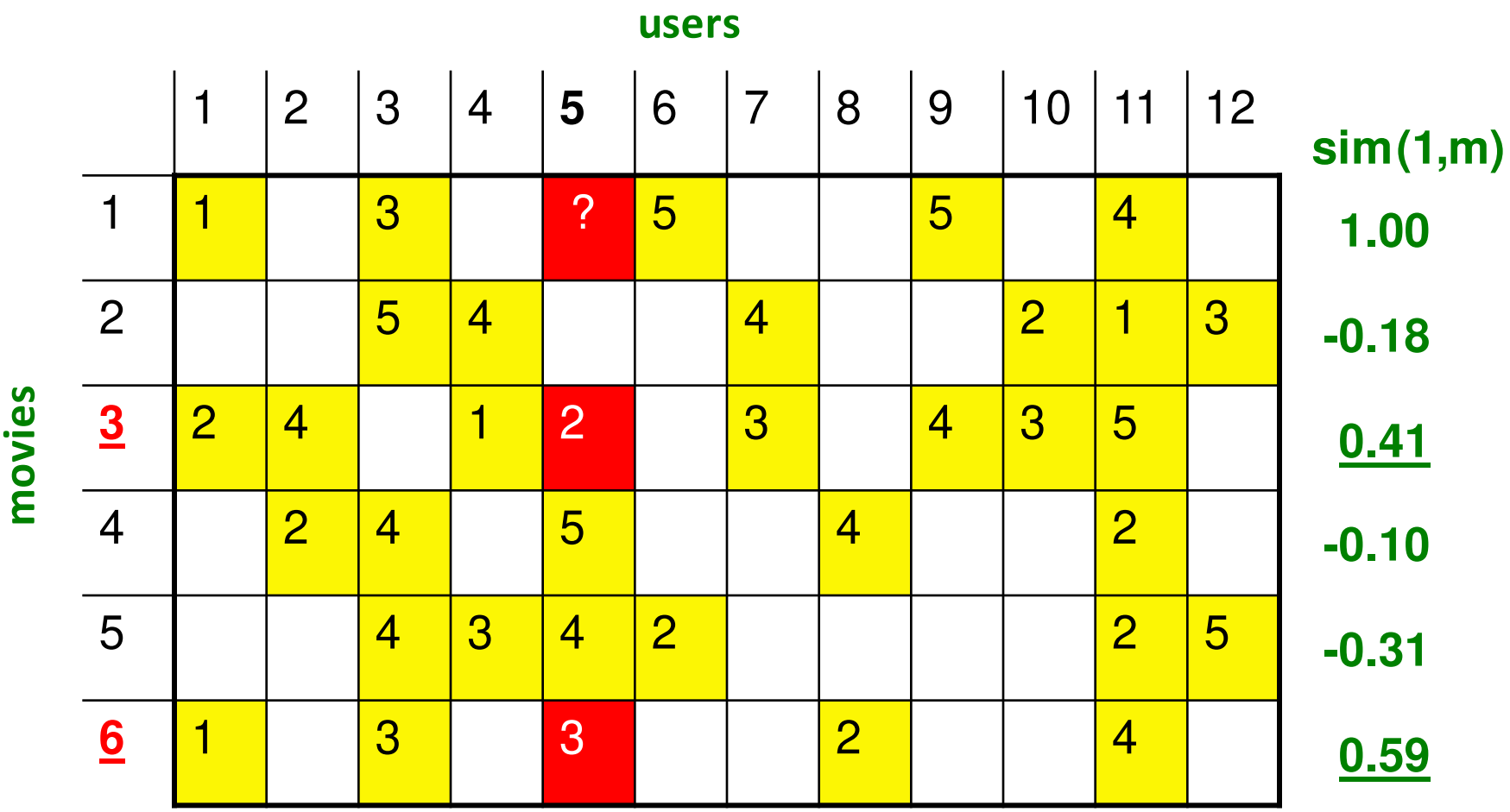
Item-Item CF (|N|=2)



Item-Item CF (|N|=2)



Item-Item CF (|N|=2)



Neighbor selection:
Identify movies similar to movie **1**, **rated by user 5**

Here we use Pearson correlation as similarity:

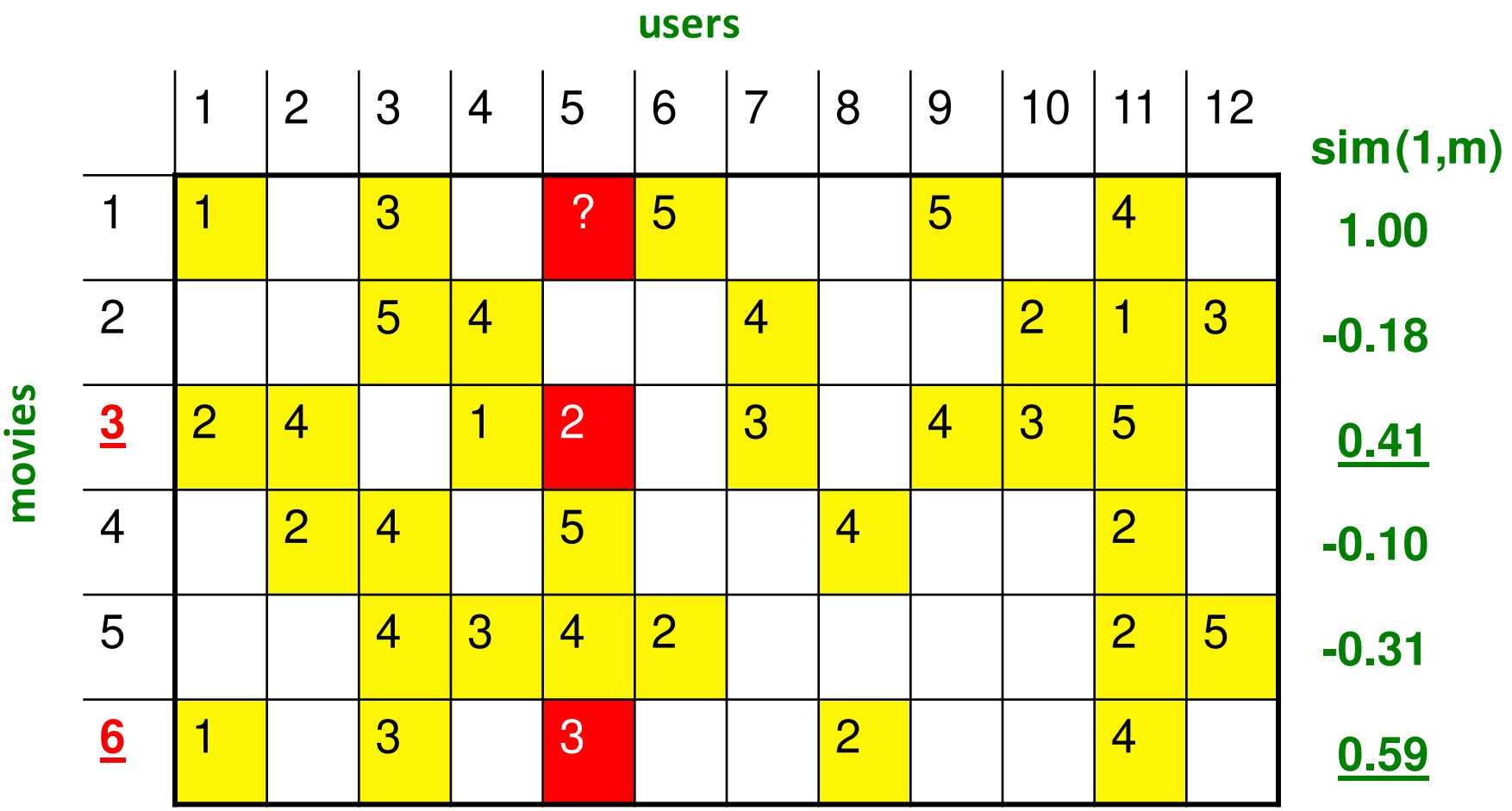
1) Subtract mean rating m_i from each movie i

$m_1 = (1+3+5+5+4)/5 = 3.6$

row 1: *[-2.6, 0, -0.6, 0, 0, 1.4, 0, 0, 1.4, 0, 0.4, 0]*

2) Compute cosine similarities between rows

Item-Item CF (|N|=2)



Compute similarity weights:

$s_{1,3}$=0.41, $s_{1,6}$=0.59

Item-Item CF (|N|=2)

users

movies

	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3		2.6	5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	

Predict by taking weighted average:

$$r_{ix} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{jx}}{\sum s_{ij}}$$

$r_{1.5}$ = **(0.41*2 + 0.59*3) / (0.41+0.59) = 2.6**

Before:

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

CF: Common Practice

- Define **similarity s_{ij}** of items ***i*** and ***j***
- Select ***k*** nearest neighbors ***N(i; x)***
  - Items most similar to ***i***, that were rated by ***x***
- Estimate rating r_{xi} as the weighted average:

$$r_{xi} = b_{xi} + \frac{\sum_{j \in N(i;x)} s_{ij} \cdot (r_{xj} - b_{xj})}{\sum_{j \in N(i;x)} s_{ij}}$$

baseline estimate for r_{xi}

$$b_{xi} = \mu + b_x + b_i$$

- μ = overall mean movie rating
- b_x = rating deviation of user ***x***
  = (*avg. rating of user* ***x***) $- \mu$
- b_i = rating deviation of movie ***i***

Baseline predictor

$$r_{xi} = \mu + b_x + b_i$$

Overall mean rating

Bias for user ***x***

Bias for movie ***i***

- **Example:**
  - Mean rating: $\mu = 3.7$
  - You are a critical reviewer: your ratings are 1 star lower than the mean: $b_x = -1$
  - Star Wars gets a mean rating of *0.5* higher than average movie: $b_i = +0.5$
  - Predicted rating for you on Star Wars:
    $= 3.7 - 1 + 0.5 = 3.2$

Item-Item vs. User-User

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.8	
Bob		0.5		0.3
Carol	0.9		1	0.8
David			1	0.4

- **In practice, it has been observed that item-item often works better than user-user**
- **Why?** Items are simpler, users have multiple tastes

Pros/Cons of Collaborative Filtering

- **+ Works for any kind of item**
  - No feature selection needed
- **- Cold Start:**
  - Need enough users in the system to find a match
- **- Sparsity:**
  - The user/ratings matrix is sparse
  - Hard to find users that have rated the same items
- **- First rater:**
  - Cannot recommend an item that has not been previously rated
  - New items, Esoteric items
- **- Popularity bias:**
  - Cannot recommend items to someone with unique taste
  - Tends to recommend popular items

For today…

- Follow the cosine similarity example: https://neo4j.com/docs/graph-data-science/current/alpha-algorithms/cosine/
- Add yourself as a node and include your cuisine preferences.
- Write a query that returns the 3 most similar persons to yourself, sorted by similarity
- (optionally) Write a query that would connect the 3 most similar persons to you with a relationship MYSIMILARITY and the similarity as the weight
- Upload the result to ICON

Neo4j Rest API

https://neo4j.com/docs/rest-docs/current/

What is REST?

- REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000.
- Key abstraction of information in REST is a resource
- A resource identifier is used to identify the particular resource involved in an interaction between components
- Typically used in conjunction with HTTP protocol
  - GET — retrieve a specific resource (by id) or a collection of resources
  - POST — create a new resource
  - PUT — update a specific resource (by id)
  - DELETE — remove a specific resource by id

curl

- curl is a command-line utility that lets you execute HTTP requests with different parameters and methods.
- https://curl.se/download.html

- **curl http://localhost:7474/db/data/**

Using curl with Windows

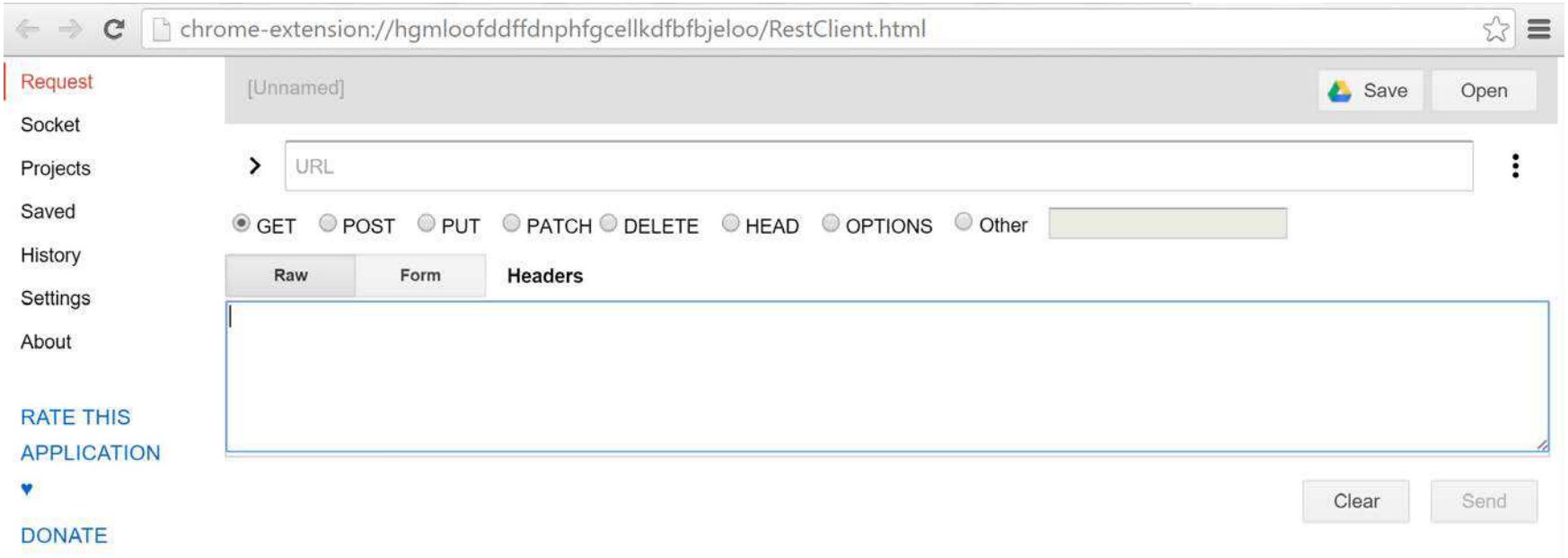
If you're using Windows, note the following formatting requirements when using curl:

- Use double quotes in the Windows command line. (Windows doesn't support single quotes.)
- Don't use backslashes (\) to separate lines. (This is for readability only and doesn't affect the call on Macs.)
- By adding -k in the curl command, you can bypass curl's security certificate, which may or may not be necessary.

Alternative

Google Chrome Advanced REST Client

After installing and launching the Google Chrome Advanced REST Client application, your browser should appear as follows:



Accessing Neo4j Data using REST API

- Service root is the starting point to discover the REST API
  - GET http://localhost:7474/db/data/

    Accept: application/json

Creating Nodes Using REST

- Creating a node requires a POST to the /db/data/node path
- with JSON data. As a matter of convention, it pays to give each node a name
- property. This makes viewing any node's information easy: just call name.
- **$ curl -i -X POST http://localhost:7474/db/data/node **
- **-H** *"Content-Type: application/json"* ****
- **-d '{**
- "name": "P.G. Wodehouse"
- "genre": "British Humour"
- }'

- **$ curl http://localhost:7474/db/data/node/1**
- **$ curl http://localhost:7474/db/data/node/1/properties/genre**

- **Add another node with these properties** ["name" : "Jeeves Takes Charge", "style" : "short story"]

Creating Relationships Using REST

- P. G. Wodehouse wrote the short story “Jeeves Takes Charge,” we
- can make a relationship between them:

- **curl -i -XPOST http://localhost:7474/db/data/node/9/relationships **
- **-H** *"Content-Type: application/json"* ****
- **-d '{**
- "to": "http://localhost:7474/db/data/node/1",
- "type": "WROTE",
- "data":{"published":"November 28, 1916"}
- }

- **$ curl http://localhost:7474/db/data/node/2**

Finding a Path

- You can find the path between two nodes by posting the request data to the starting node's /paths URL. The POST request data must be a JSON string denoting the node you want the path to, the type of relationships you want to follow, and the path-finding algorithm to use.

```
curl -X POST http://localhost:7474/db/data/node/2/paths \
-H "Content-Type: application/json" \
-d '{
    "to": "http://localhost:7474/db/data/node/2",
    "relationships": {"type": "WROTE"}, "algorithm":
    "shortestPath",
    "max_depth": 10
}'
```

Indexing

- Neo4j indexes have a different path because the indexing service is a separate service.
- To create a key-value or hash style index:
- **$ curl -X POST http://localhost:7474/db/data/index/node/authors **
- **-H** *"Content-Type: application/json"* ****
- **-d '{**
- "uri": "http://localhost:7474/db/data/node/9",
- "key": "name",
- "value":"P.G.+Wodehouse"
- }’

- **curl http://localhost:7474/db/data/index/node/authors/name/P.G.+Wodehouse**

Full-text indexing

- Neo4j incorporates Lucene to build an inverted index over the entire dataset.

```
curl -X POST http://localhost:7474/db/data/index/node \
-H "Content-Type: application/json" \
-d '{
"name": "fulltext",
"config": {"type": "fulltext", "provider": "lucene"}
}'
```

- Add Wodehouse to the full-text index, you get this:

```
$ curl -X POST http://localhost:7474/db/data/index/node/fulltext \
-H "Content-Type: application/json" \
-d '{
"uri": "http://localhost:7474/db/data/node/9",
"key": "name",
"value" : "P.G.+Wodehouse"
}'
```

- Then you can query using the Lucene syntax on the index URL

```
$ curl http://localhost:7474/db/data/index/node/fulltext?query=name:P*
```

REST and Cypher

- Neo4j REST interface has a Cypher plugin

```
$ curl -X POST \
http://localhost:7474/db/data/cypher \
-H "Content-Type: application/json" \
-d '{
"query": "MATCH ()-[r]-() RETURN r;"
}'
{
"columns" : [ "n.name" ],
"data" : [ [ "Prancing Wolf" ], [ "P.G. Wodehouse" ] ]
}
```

Using Transactional Cypher HTTP Endpoint

- Allows you to execute a series of Cypher statements within the scope of a transaction
- The transaction may be kept open across multiple HTTP requests, until the client chooses to commit or roll back
- Each HTTP request can include a list of statements
- https://neo4j.com/docs/http-api/3.5/actions/

Using Drivers to Access Neo4j

- https://neo4j.com/developer/language-guides/
- Binary Bolt protocol (starting with Neo4j 3.0)
- Binary protocol is enabled in Neo4j by default and can be used in any language driver that supports it
- Drivers implement all low level connection and communication tasks

```
import org.neo4j.driver.v1.*;

public class Neo4j
{
 public static void javaDriverDemo() {
   Driver driver = GraphDatabase.driver("bolt://ganxis.nest.rpi.edu", "neo4j", "neo4j"));
   Session session = driver.session();

   StatementResult result = session.run("MATCH (a)-[]-(b)-[]-(c)-[]-(a) WHERE a.id < b.id AND b.id < c.id
RETURN DISTINCT a,b,c");
   int counter = 0;
   while (result.hasNext())
   {
     counter++;
     Record record = result.next();
     System.out.println(record.get("a").get("id") + " \t" + record.get("b").get("id") + " \t" +
record.get("c").get("id"));
   }
   System.out.println("Count: " + counter);
   session.close();
   driver.close();
 }
 public static void main(String [] args)
 {
   javaDriverDemo();
 }
}
```

Using Core Java API

- Native Java API performs database operations directly with Neo4j core

```
import java.io.*;
import java.util.*;
import org.neo4j.graphdb.*

public class Neo4j
{
  public enum NodeLabels implements Label { NODE; }
  public enum EdgeLabels implements RelationshipType{ CONNECTED; }
  public static void javaNativeDemo(int nodes, double p) {
    Node node1, node2; Random randomgen = new Random();
    GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
    GraphDatabaseService db = dbFactory.newEmbeddedDatabase(new File("TestNeo4jDB"));
    try (Transaction tx = db.beginTx()) {
      for (int i = 1; i <= nodes; i++) {
        Node node = db.createNode(NodeLabels.NODE);
        node.setProperty("id", i);
      }
      for (int i = 1; i <= nodes; i++)
        for (int j = i + 1; j <= nodes; j++) {
          if (randomgen.nextDouble() < p) {
            node1 = db.findNode(NodeLabels.NODE, "id", i);
            node2 = db.findNode(NodeLabels.NODE, "id", j);
            Relationship relationship =
node1.createRelationshipTo(node2,EdgeLabels.CONNECTED);
            relationship = node2.createRelationshipTo(node1,EdgeLabels.CONNECTED);
          }
        }
      tx.success();
    }
    db.shutdown();
  }
  public static void main(String [] args) {
    javaNativeDemo(100, 0.2); }
```

Phyton

- https://neo4j.com/developer/python/

- Py2neo is a client library and comprehensive toolkit for working with Neo4j from within Python applications.

```
pip install py2neo
from py2neo import Graph
graph = Graph("bolt://localhost:7687", auth=("neo4j", "asdfgh123"))
query = "MATCH (n) return n"
graph.run(query).data()
```

For today…

- Download and install redis http://redis.io
- From the command line start the server by calling:
- **$ redis-server**
- It won't run in the background by default, but you can append &, or just open another terminal.
- Next, run the command line tool, which should connect to the default port 6379 automatically.
- **$ redis-cli**
- After you connect, ping the server (it should reply PONG):

  redis 127.0.0.1:6379> PING
- Submit to ICON a screenshot of the redis-cli running