COMP5338 – Advanced Data Models

Week 6: Graph Data and Neo4j Introduction

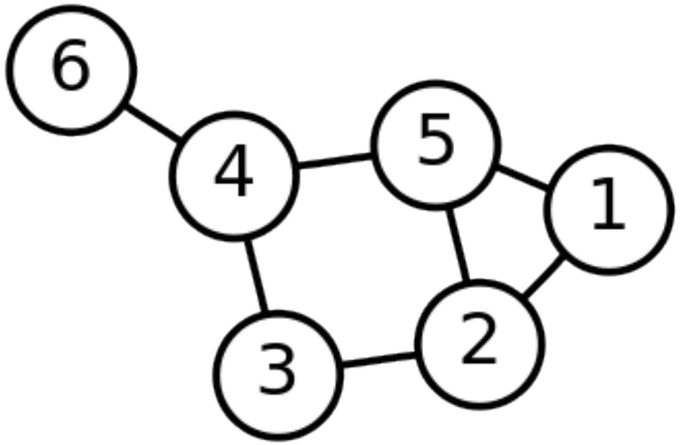
Ying Zhou
School of Computer Science

Outline

- **Brief Review of Graphs**
  - Examples of Graph Data
  - Modelling Graph Data
- Property Graph Model
- Cypher Query

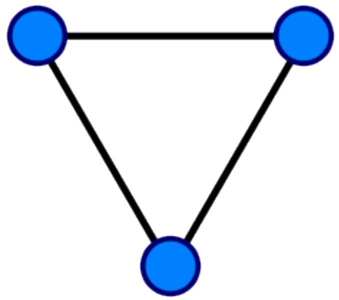
Graphs

- A graph is just a collection of *vertices* and *edges*
  - Vertex is also called Node
  - Edge is also called Arc/Link

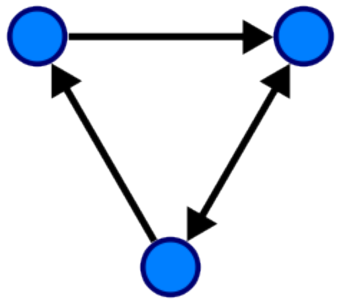


Type of Graphs

- Undirected graphs
  - Edges have no orientation (direction)
  - (a, b) is the same as (b, a)



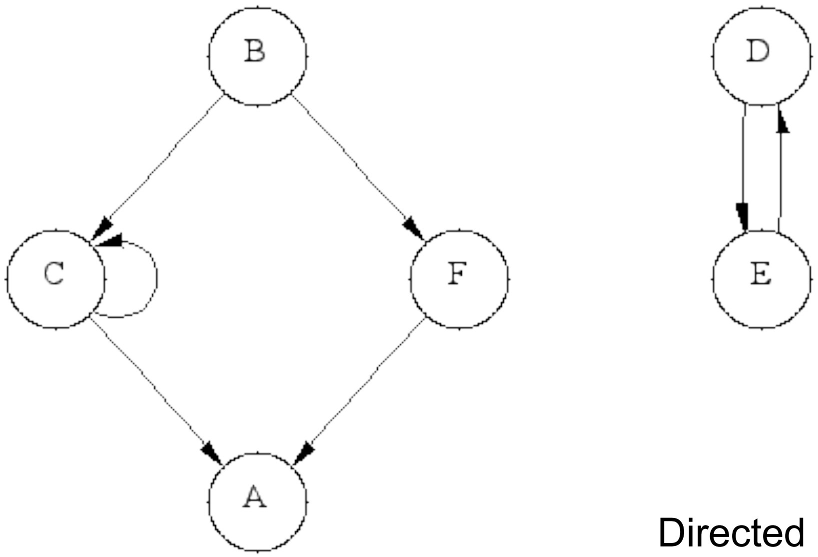
- Directed graphs
  - Edges have orientation (direction)
  - (a, b) is not the same as (b, a)



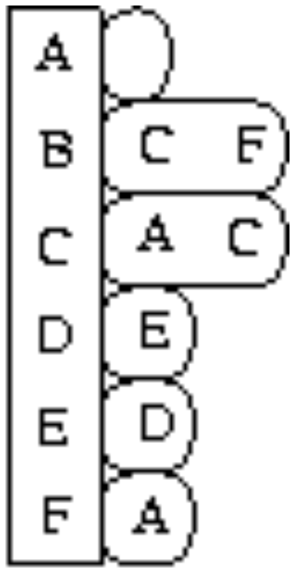
Representing Graph Data

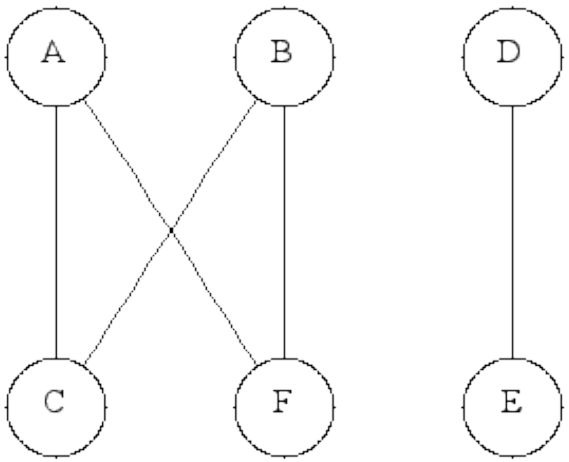
- Data structures used to store graphs in programs
  - Adjacency list
  - Adjacency matrix

Adjacency List

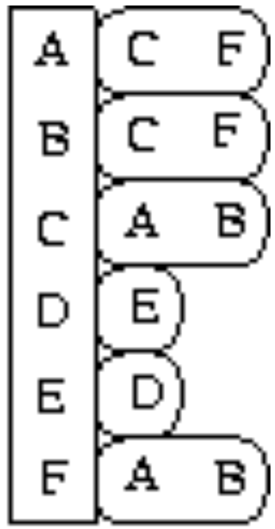


Directed Graph

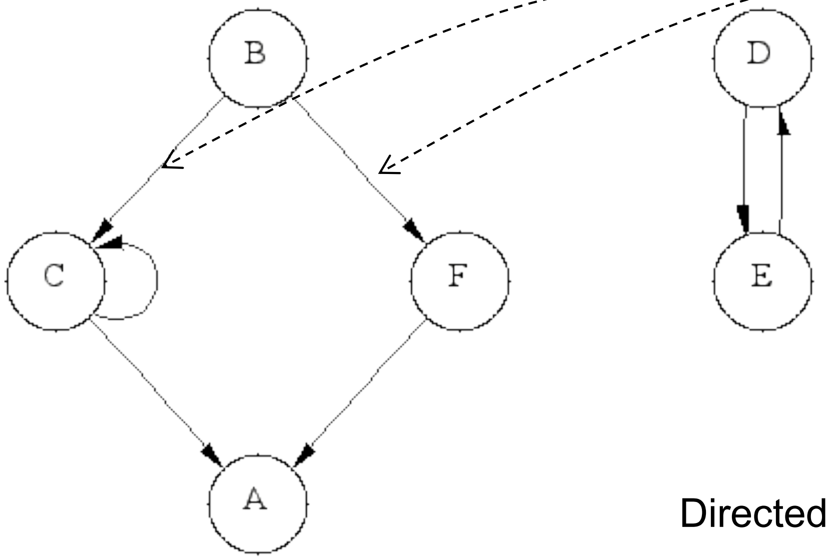




Undirected Graph

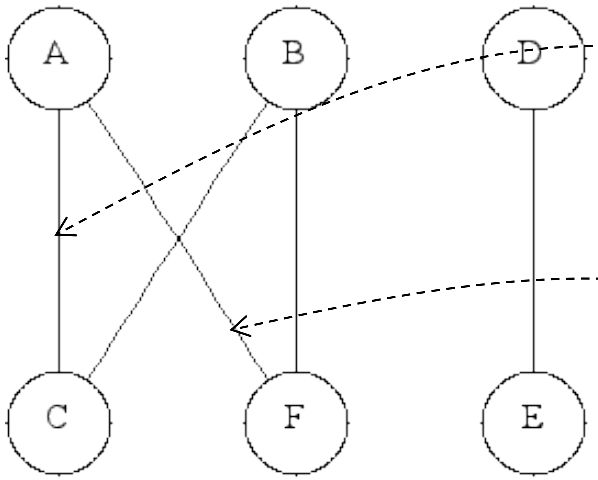


Adjacency matrix



Directed Graph

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	1	0	0	1
C	1	0	1	0	0	0
D	0	0	0	0	1	0
E	0	0	0	1	0	0
F	1	0	0	0	0	0



Undirected Graph

	A	B	C	D	E	F
A	0					
B	0	0				
C	1	1	0			
D	0	0	0	0		
E	0	0	0	1	0	
F	1	1	0	0	0	0

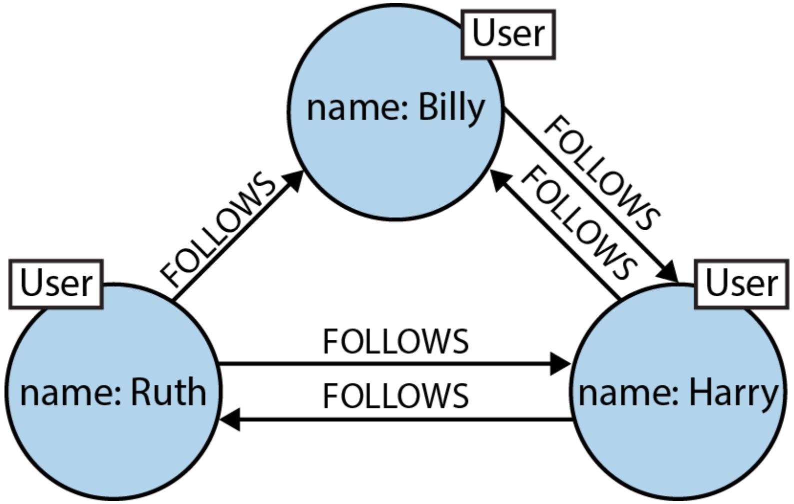
Outline

- **Brief Review of Graphs**
  - **Examples of Graph Data**
  - Modelling Graph Data
- Introduction to Neo4j
- Cypher Query

Examples of graphs

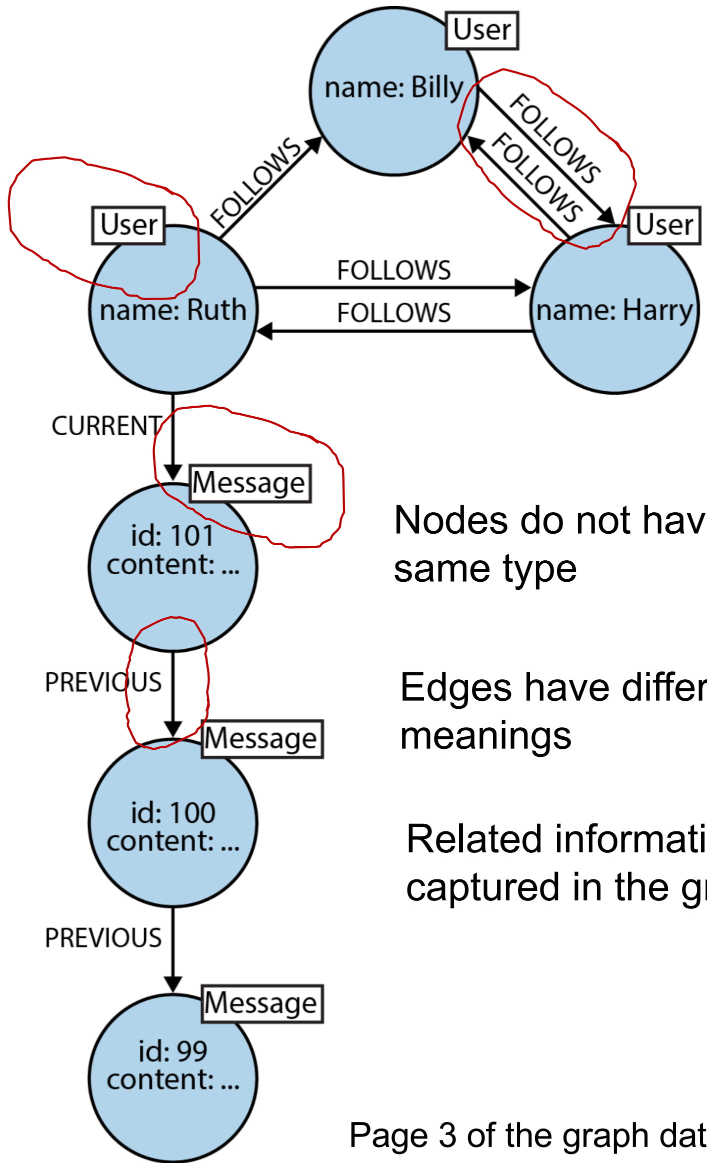
- Social graphs
  - Organization structure
  - Facebook, LinkedIn, etc.
- Computer Network topologies
  - Data centre layout
  - Network routing tables
- Road, Rail and Airline networks

Social Graphs and extension



A small social graph

Page 2 of the graph database book



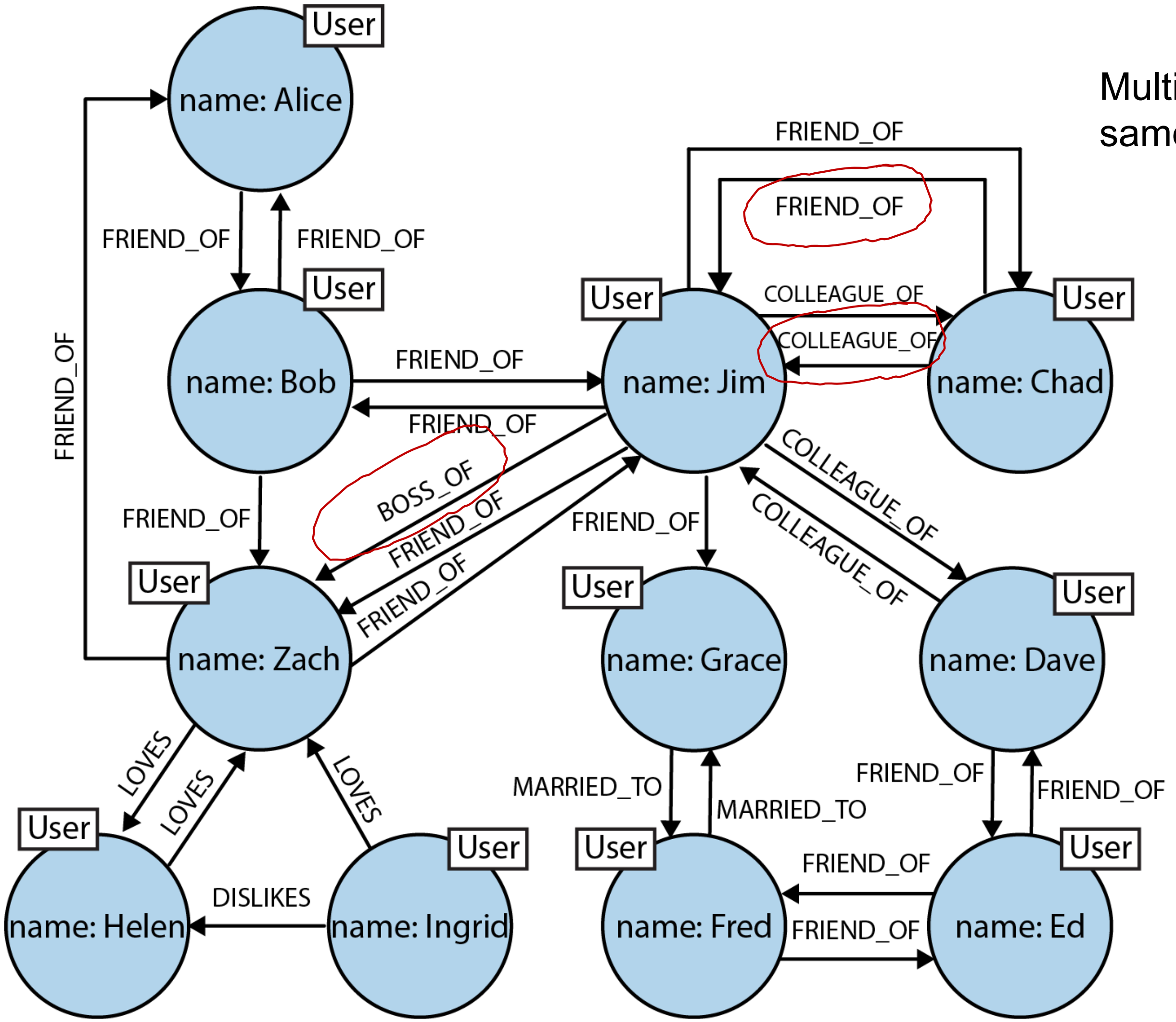
Nodes do not have to be of same type

Edges have different meanings

Related information can be captured in the graph

Page 3 of the graph database book

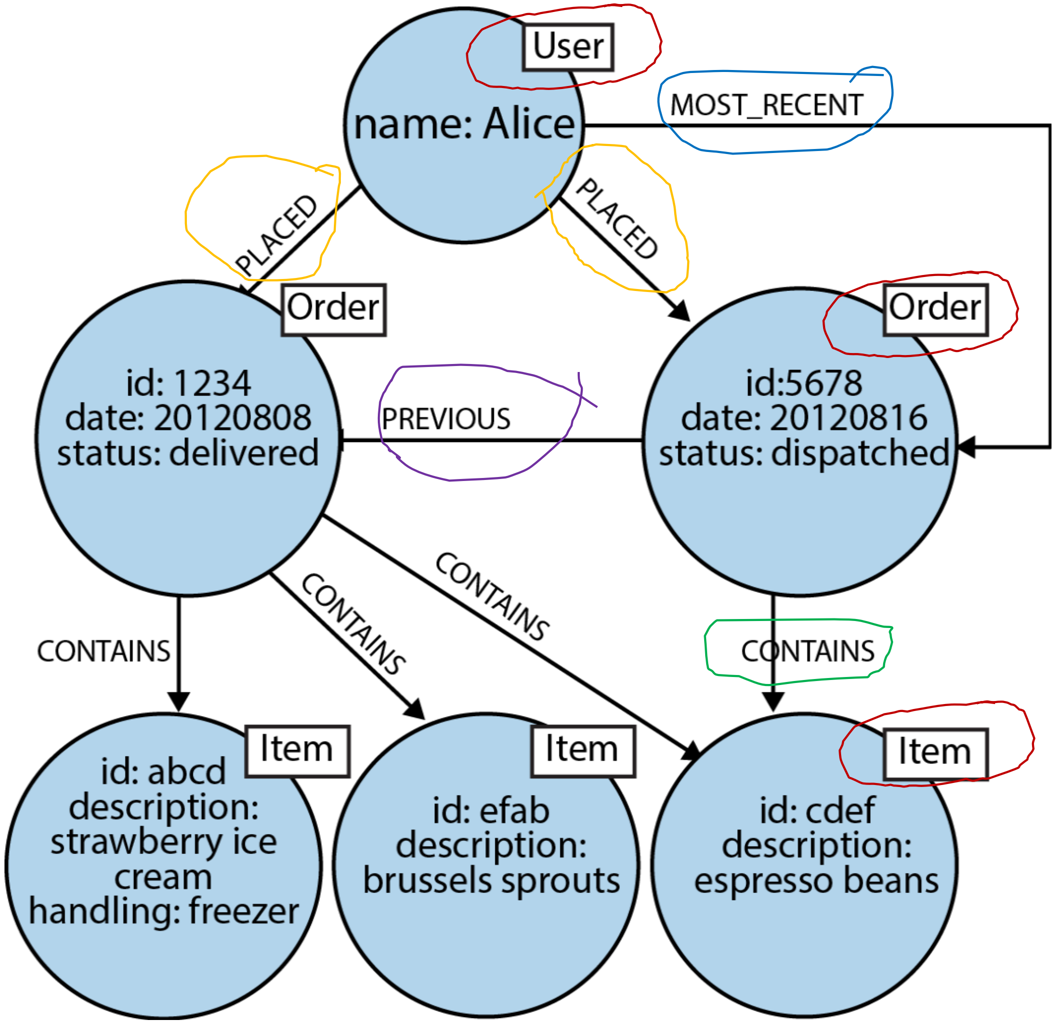
Social Graph with Various Relationships



Multiple edges between the same pair of nodes

Page 19 of the graph database book

Transaction information



This graph shows three types of entities

And various types of relationships among those entities

User PLACED order

User MOST_RECENT order

Order CONTAINS Item

Order PREVIOUS Order

Page 23 of the graph database book

Outline

- **Brief Review of Graphs**
  - Examples of Graph Data
  - **Modelling Graph Data**
- Property Graph Model
- Cypher Query

RDBMS to store graph

Person

ID	Person
1	Alice
2	Bob
…	…
99	Zach

nodes

PersonFriend

PersonID	FriendID
1	2
2	1
2	99
…	…
99	1

edges

- Who are Bob's friends?

```
SELECT  p1.Person
FROM    Person p1 JOIN PersonFriend pf ON pf.FriendID = p1.ID
        JOIN Person p2 ON pf.PersonID = p2.ID
WHERE    p2.Person = "Bob"
```

Page 13 of the graph database book

RDBMS to store Graphs

- Who are friends with Bob?

Person

ID	Person
1	Alice
2	Bob
...	...
99	Zach

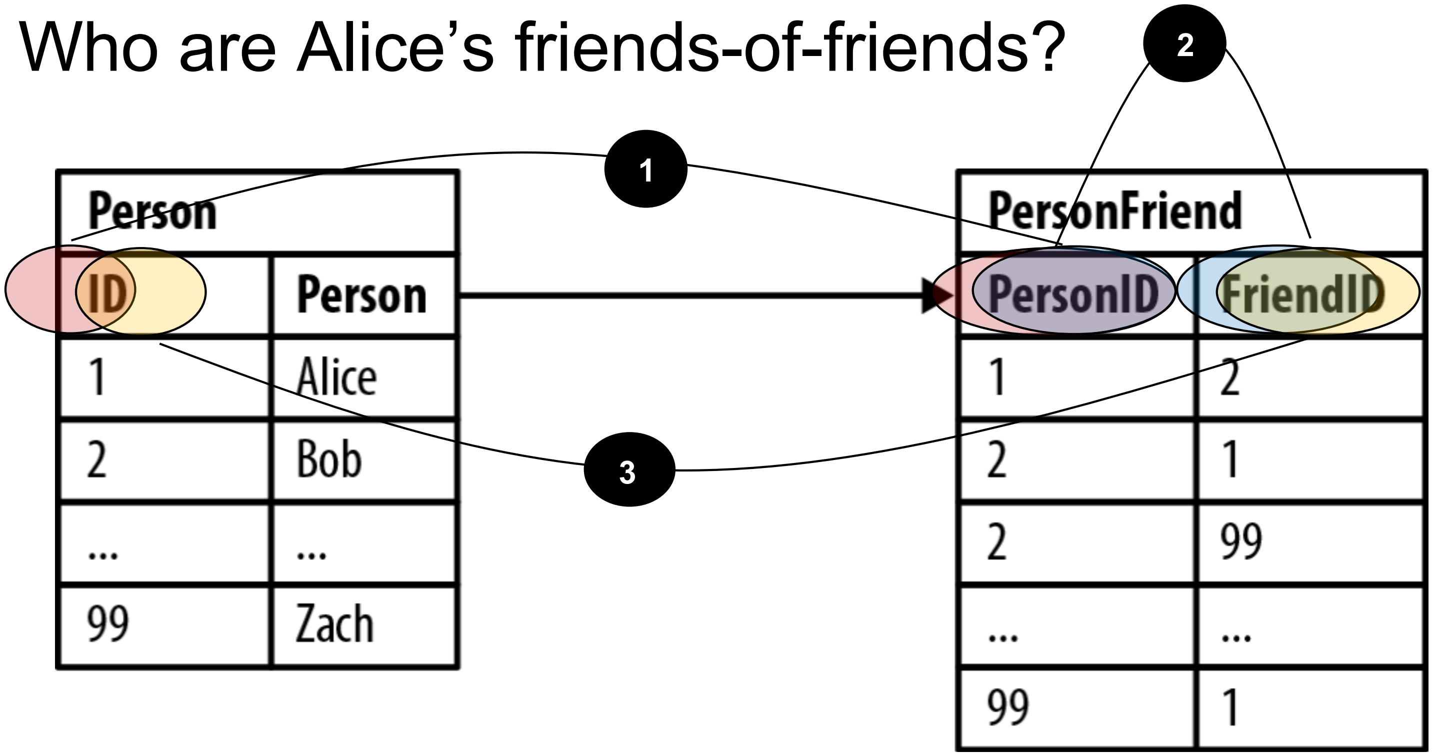
PersonFriend

PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

```
SELECT   p1.Person
FROM     Person p1 JOIN PersonFriend pf ON pf.PersonID = p1.ID
         JOIN Person p2 ON pf.FriendID = p2.ID
WHERE
         p2.Person = "Bob"
```

RDBMS to store Graphs

- Who are Alice's friends-of-friends?




```sql
SELECT   p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM     PersonFriend pf1 JOIN Person p1 ON pf1.PersonID = p1.ID
         JOIN PersonFriend pf2 ON pf2.PersonID = pf1.FriendID
         JOIN Person p2 ON pf2.FriendID = p2.ID
WHERE    p1.Person = "Alice" AND pf2.FriendID <> p1.ID
```

MongoDB to store Graph

`persons` collection

```
{ _id: 1,
  person: "Alice",
  friends:[2]
}

{ _id: 2,
  person: "Bob",
  friends:[1,99]
}

{ _id: 99,
  person: "Zach",
  friends:[1]
}
```

- Who are Bob's friends?
  - Find out Bob's friends' ID
    - `db.persons.find({person:"Bob"},{friends:1})`
  - For each id, find out the actual person
    - `db.persons.find({_id: 1},{person:1}),`
      `db.persons.find({_id: 99},{person:1}),`
    - `db.persons.find({_id:{$in:[1,99]}}, {person:1})`
- Who are friends with Bob?
  - Find out Bob's id
    - db.persons.find({person:"Bob"})
  - Find out the persons that are friends with Bob
    - db.persons.find({friends: **2**}, {person:1})
- Who are Alice's friends-of-friends?
  - Find out Alice's friends ID
    - `db.persons.find({person:"Alice"},{friends:1})`
  - For each id, find out the friends ID again
    - `db.persons.find({_id:{$in:[2]}}, {friends:1}`
  - For each id, find out the actual person
    - `db.persons.find({_id:{$in:[1,99]}}, {person:1})`
- The MongDB 3.4 and later has a new aggregation stage called `$graphLookup`

$graphLookup

```
db.persons.aggregate([
  {$match:{person:"Alice"}},
  {$graphLookup:{
     from: "persons",
     startWith: "$friends",
     connectFromField:"friends",
     connectToFirld:"_id",
     maxDepth: 1,
     as: "friendsnetwork"}}
  ])
```

```
{"_id" : 1,
  "person" : "Alice",
  "friends" : [ 2],
  "friendsnetwork" : [
    {"_id" : 99.0,
      "name" : "Zach",
      "friends" : [ 1, 3],
      "depth" : 1 },
    {"_id" : 1,
      "name" : "Alice",
      "friends" : [ 2],
      "depth" : 1},
    {"_id" : 2,
      "name" : "Bob",
      "friends" : [1, 99],
      "depth" : 0}
  ] }
```

```
{ _id: 1,
  person: "Alice",
  friends:[2]
}

{ _id: 2,
  person: "Bob",
  friends:[1,99]
}

{ _id: 99,
  person: "Zach",
  friends:[1]
}
```

In Summary

- It is possible to store graph data in various storage systems
  - Shallow traversal
    - Relatively easy to implement
    - Performance OK
  - Deep traversal or traversal in other direction
    - Complicated to implement
      - Multiple joins or multiple queries or full table scan
    - Less efficient
    - Error prone

Outline

- Brief Review of Graphs
- **Property Graph Model**
- Cypher Query

Graph Technologies

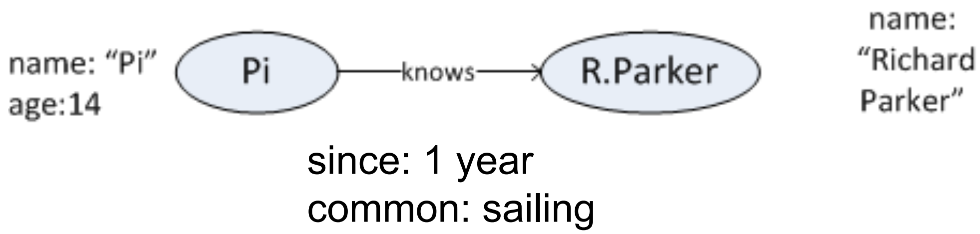
- Graph Processing
  - take data in any input format and perform graph related operations
  - OLAP – OnLine Analysis Processing of graph data
  - Google Pregel, Apache Giraph
- Graph Databases
  - manage, query, process graph data
  - support high-level query language
  - native storage of graph data
  - OLTP – OnLine Transaction Processing possible
  - OLAP – also possible

Graph Data Models

- RDF (Resource Description Framework) Model
  - Express node-edge relation as “subject, predicate, object” triple (RDF statement)
  - SPARQL query language
  - Examples: AllegroGraph, Apache Jena
- Property Graph Model
  - Express node and edge as object like entities, both can have properties
  - Various query language
  - Examples
    - Apache Titan
      - Support various NoSQL storage engine: BerkeleyDB, Cassandra, HBase
      - Structural query language: Gremlin
    - Neo4j
      - Native storage manager for graph data (Index-free Adjacency)
      - Declarative query language: Cypher query language

Property Graph Model

- Proposed by **Neo** technology
- No standard definition or specification
- Both Node and Edges can have property
  - RDF model cannot express edge property in a natural and easy to understand way
- The actual storage varies
- The query language varies



Neo4j

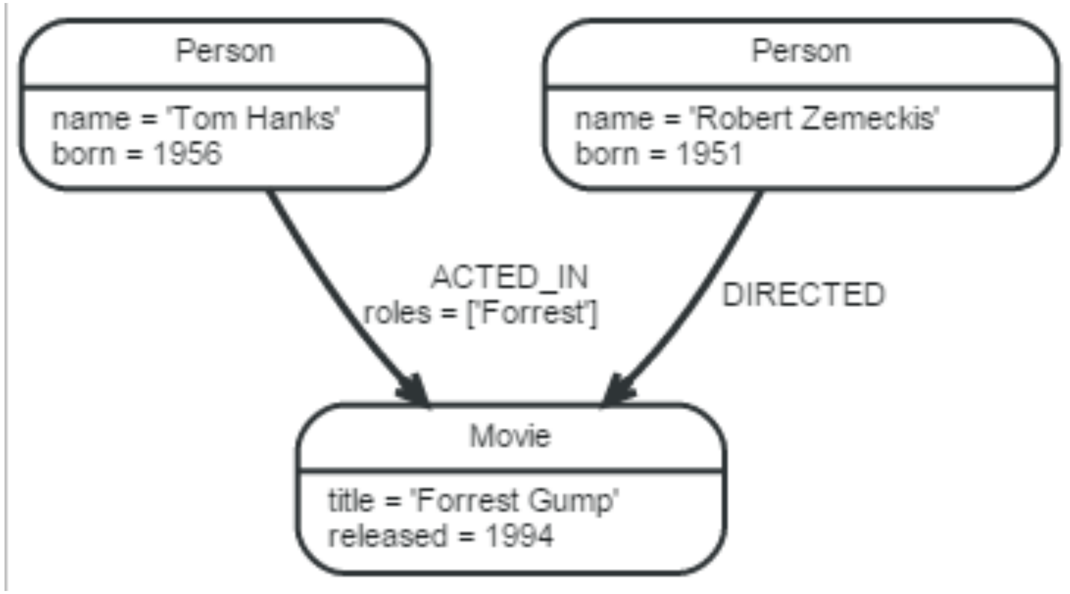
- Native graph storage using **property graph model**
- Index-free Adjacency
  - Nodes and Edges are stored based on graph structure
- Supports indexes
- **Cypher** – query language
- Replication
  - Traditional master/slave replication mechanism
- Neo4j also introduced a sharded graph mechanism since 4.0
  - Neo4j Fabric

Property Graph Model as in Neo4j

- Property graph has the following characteristics
  - It contains <u>nodes</u> and <u>relationships</u>
  - Nodes contain <u>properties</u>
    - Properties are stored in the form of key-value pairs
    - A node can have <u>labels</u> (classes)
  - Relationships connect nodes
    - Has a *direction*, an optional *type*, a *source node* and a *target node*
    - No dangling relationships (can't delete node with a relationship)
  - <u>Properties</u>
    - Both nodes and relationships have properties
    - Useful in modeling and querying based on properties of relationships

https://neo4j.com/developer/guide-data-modeling/

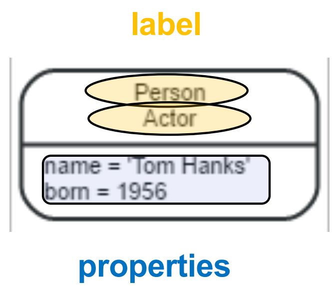
Property Graph Model Example



It models a graph with three entities: two **person** and one **movie**, each with a set of properties;
It also models the relationship among them: one person acted in the movie with a role, another person directed the movie

Property Graph Model: Nodes

- Nodes are often used to represent entities, e.g. objects
  - It has properties
  - It can have labels
- A label is a way to group similar nodes
  - It acts like the 'class' concept in programming world
- Label is a dynamic and flexible feature
  - It can be added or removed during run time
  - It can be used to tag node temporarily
    - E.g. :Suspend, :OnSale, etc

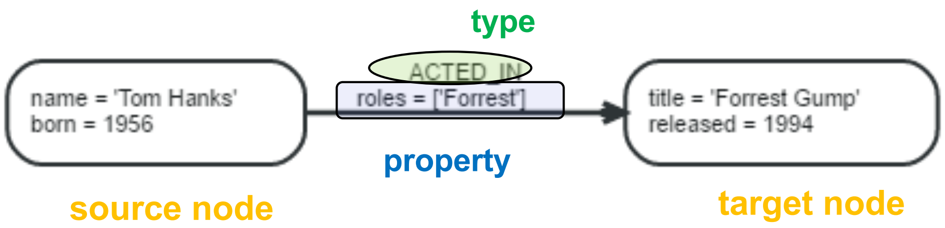


A node with two labels and two properties

Property Graph Model: Relationships

- A relationship connects two nodes: source node and target node
  - The source and the target node can be the same one
- It always has a direction
  - But traversal can happen in either direction

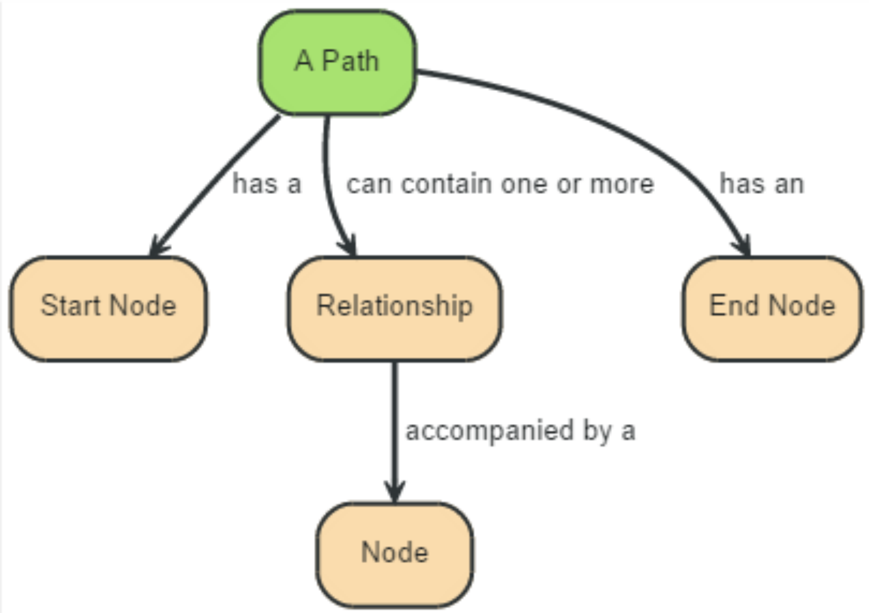
- It can have **<u>a</u>** type
- It can have properties

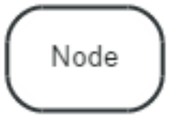


Property Graph Model: Properties

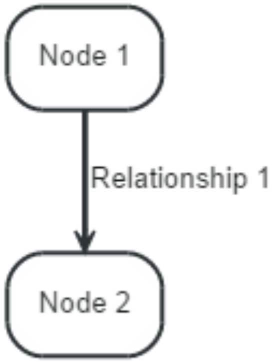
- A property is a pair of property key and property value
- The property value can be of simple type:
  - Number: Integer and Float
  - String
  - Boolean
  - Spatial Type: Point
  - Temporal Type
- The property value can also have homogeneous list of simple types as type
  - e.g. a list of integers or strings
- It cannot have heterogeneous list or other complex types with many levels of embedding

Property Graph Model: Paths





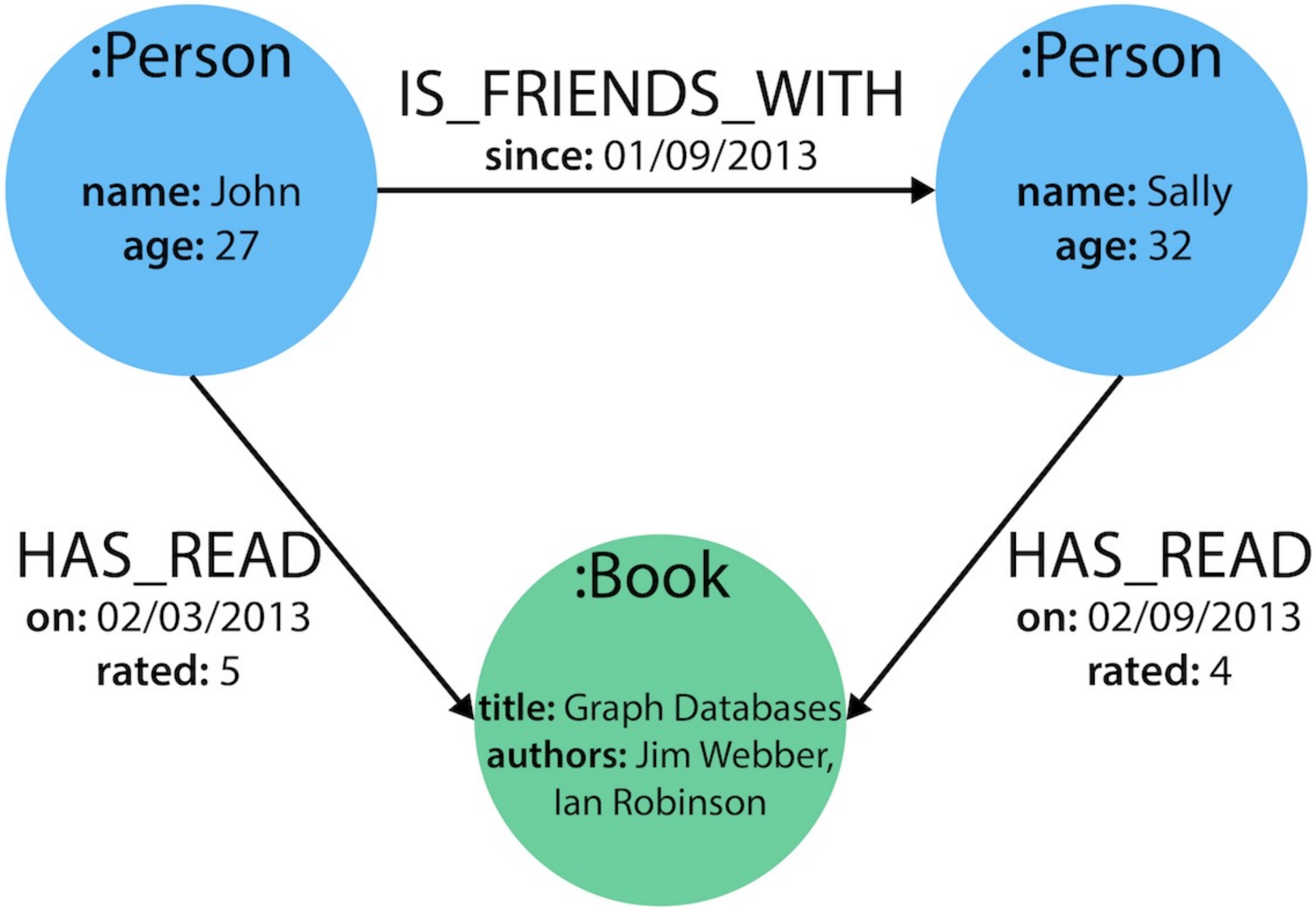
A path of length zero



A path of length one

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

Another Example



Outline

- Brief Review of Graphs
- Property Graph Model
- **Cypher Query**
  - **Patterns and basic clauses**
  - Subclause, subquery and functions

Cypher

- Cypher is a query language specific to Neo4j
- Easy to read and understand
- It uses **patterns** to represent core concepts in the property graph model
  - E.g. a pattern may represent that a user node is having a transaction with the item "*formula*" in it.
  - There are basic pattern representing nodes, relationships and path
- It uses **clauses** to build queries; Certain clauses and keywords are inspired by SQL
  - A query may contain multiple clauses
- **Functions** can be used to perform aggregation and other types of analysis

Cypher patterns: node

- A single node
  - A node is described using a pair of parentheses, and is typically given an identifier (variable)
  - E.g.: **(n)** means a node **n**
  - The variable's scope is restricted in a single query statement
- Labels
  - Label(s) can be attached to a node
  - E.g.: **(a:User)** **or** **(a:User:Admin)**
- Specifying properties
  - Properties are a list of name value pairs enclosed in a curly brackets
  - E.g.: **(a { name: "Andres", sport: "Brazilian Ju-Jitsu" })**

https://neo4j.com/docs/developer-manual/current/cypher/syntax/patterns/

Cypher patterns: relationships

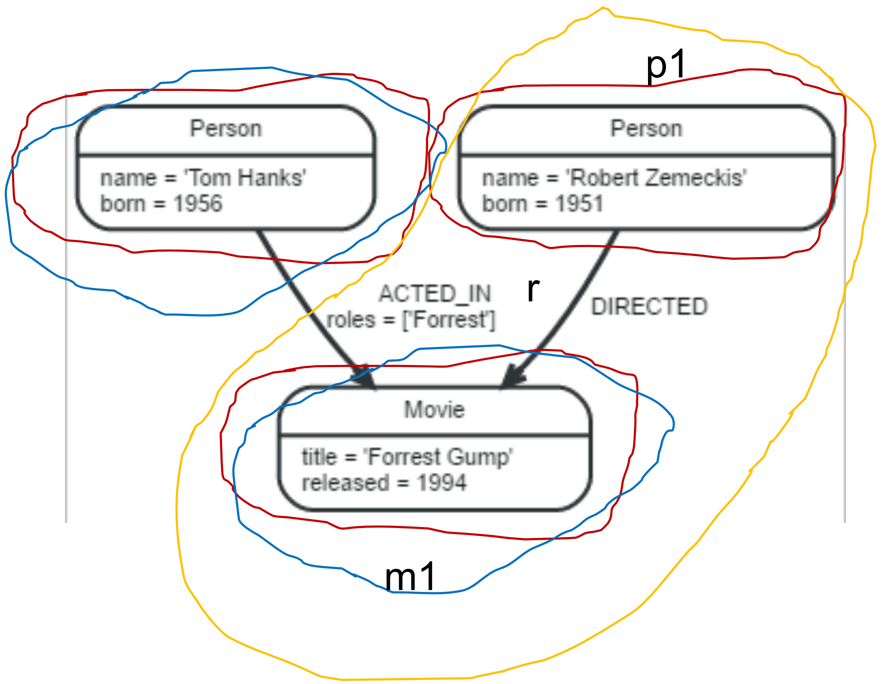
- Relationship is expressed as a pair of dashes (--)
  - Arrowhead can be added to indicate direction
  - Relationship always need a source and a target node.
- Basic Relationships
  - Directions are not important: **(a)--(b)**
  - Named relationship: **(a)-[r]->(b)**
  - Named and typed relationship: **(a)-[r:REL_TYPE]->(b)**
  - Specifying Relationship that may belong to one of a set of types: **(a)-[r:TYPE1|TYPE2]->(b)**
  - Typed but not named relationship: **(a)-[:REL_TYPE]->(b)**
- Whether to not to name a node/relation depends on if we want to refer to them later in the query

Relationship of variable lengths

- (a)-[*2]->(b) describes a path of length 2 between node **a** and node **b**
  - **This is equivalent to (a)-->()-->(b)**
- (a)-[*3..5]->(b) describes a path of minimum length of 3 and maximum length of 5 between node **a** and node **b**
- Either bound can be omitted (a)-[*3..]->(b), (a)-[*..5]->(b)
- Both bounds can be omitted as well (a)-[*]->(b)
- They can be named and typed as well
  - `(a)-[r:KNOWS*1..2]->(b)`

variable type Path length

Pattern Examples



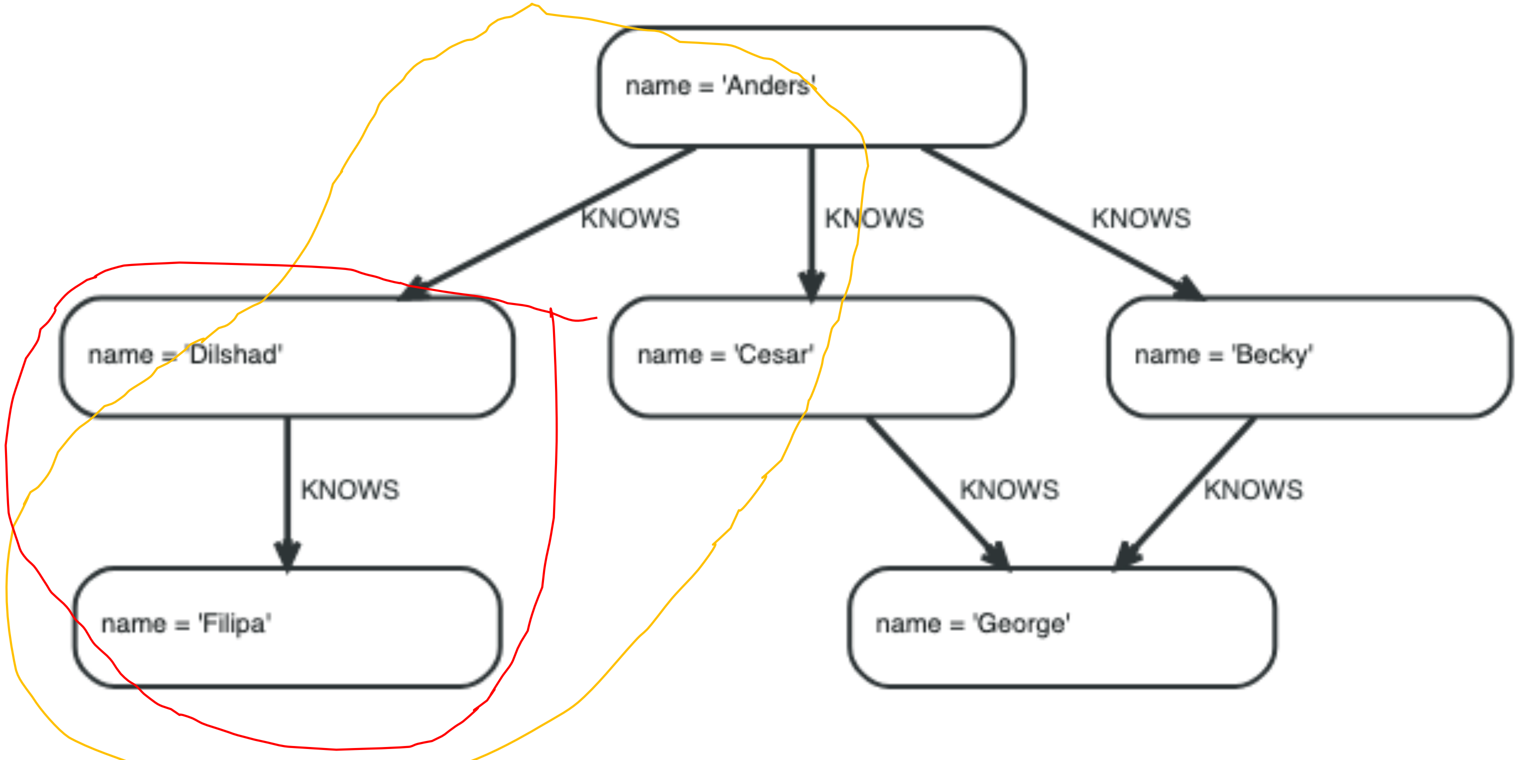
- Pattern: **(n)**
- Matches all nodes in the graph

- Pattern: **(m:Movie)**
- Matches the movie node in the graph

- Pattern: **(p:{name: ’Tom Hanks’})**
- Matches the person node with name ‘Tom Hanks’ in the graph

- Pattern: **(p1)-[r:DIRECTED]->(m1)**
- Matches the path from person Robert Zemeckis to movie “Forrest Gump”

Pattern Examples



- Pattern: `(p1{name:‘Filipa’})<-[r:KNOWS*1..2]-()`
- Matches
  - the path from Dilshad to Filipa (length 1)
  - The path from Anders to Filipa (length 2)

https://neo4j.com/docs/cypher-manual/4.1/syntax/patterns/

Create Clause

- **CREATE** *pattern*
  - Create nodes or relationships with properties

- Create a node **matrix1** with the label **Movie**

```
CREATE (matrix1:Movie {title:'The Matrix', released:1999,
                tagline:'Welcome to the Real World'})
```

We give the node an identifier so we can refer to the particular node later in the same query

- Create a node **keanu** with the label **Actor**

```
CREATE (keanu:Actor {name:'Keanu Reeves', born:1964})
```

- Create a relationship **ACTS_IN**

```
CREATE (keanu)-[:ACTS_IN {roles:'Neo'}]->(matrix1)
```

The identifier “Keanu” and “matrix1” are used in the this create clause.
We did not give the relationship a name/identifier.
We need to write the three clauses in a single query statement to be able to use those variables

Read Clause

- **`MATCH`** *`pattern`*
  `RETURN` *`var-expression`*
  - MATCH is the main reading clause
  - RETURN is a projecting clause
  - They are chained to make a query
- Return all nodes:

```
MATCH (n) RETURN n
```

- Return all nodes with a given label: select * from movie

```
MATCH (movie:Movie) RETURN movie
```

- Return all actors' name in the movie "The Matrix"

We give the Actor node an identifier "a" so we can use refer to in the RETURN sub-clause

```
MATCH (a:Actor)-[:ACTS_IN]->(:Movie{title:"The Matrix"})
RETURN a.name
```

We do not need to return the relationship so we did not give an identifier to it

We do not need to give an identifier to the Movie node too,

Update Clause

- **`MATCH`** *`pattern`*
  `SET/REMOVE` *`properties/labels`*

- Set the age property for all actor nodes

```
MATCH (n:Actor)
SET n.age = 2014 - n.born
RETURN n
```

- Remove a property

```
MATCH (n:Actor)
REMOVE n.age
RETURN n
```

- Remove a label

```
MATCH (n:Actor{name:"Keanu Reeves"})
REMOVE n:Actor
RETURN n
```

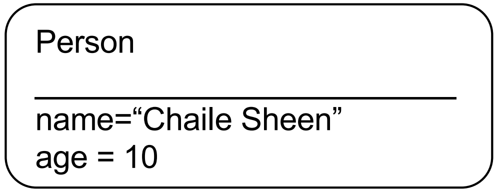
MERGE Clause: basic form

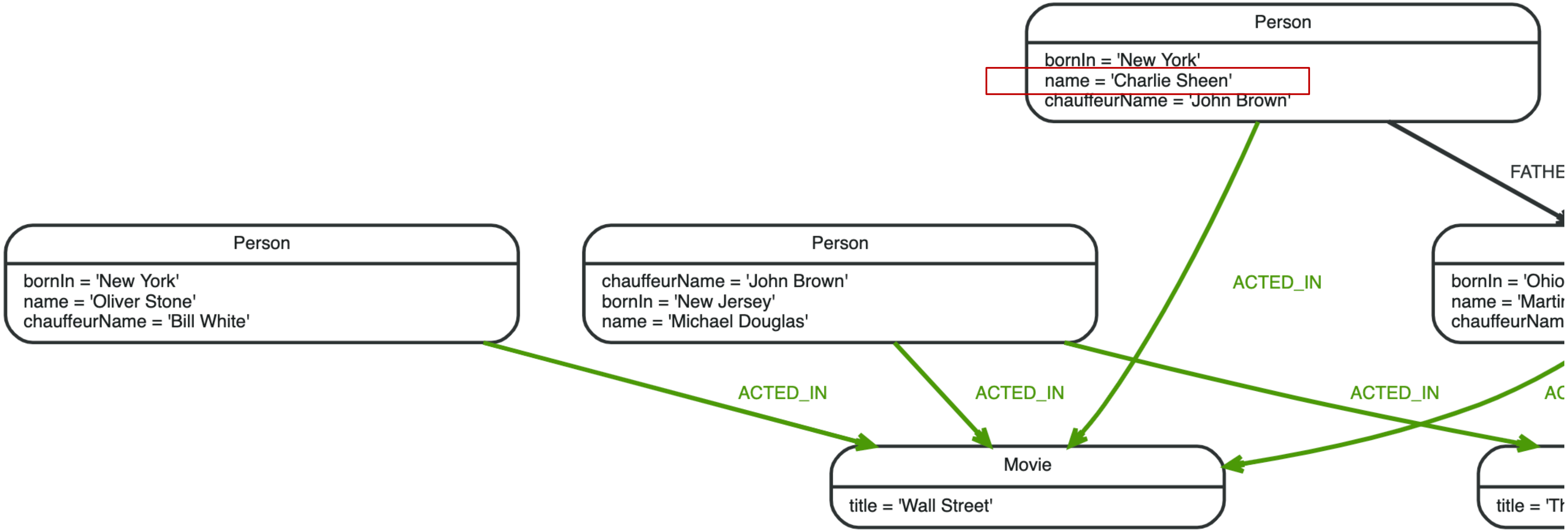
- MERGE clause acts like an upsert:
  - updating an existing pattern when there is a match or create a new one when there is no match
- Simplest form is
  - **MERGE** *pattern*
  - Example:
  - **MERGE (charlie { name: 'Charlie Sheen', age: 10 })**
    RETURN Charlie
  - Create a new node if we could not find a node with all matching properties in the current graph

https://neo4j.com/docs/cypher-manual/4.1/clauses/merge/

Example Graph

```
MERGE (charlie { name: 'Charlie Sheen', age: 10 })
RETURN Charlie
```



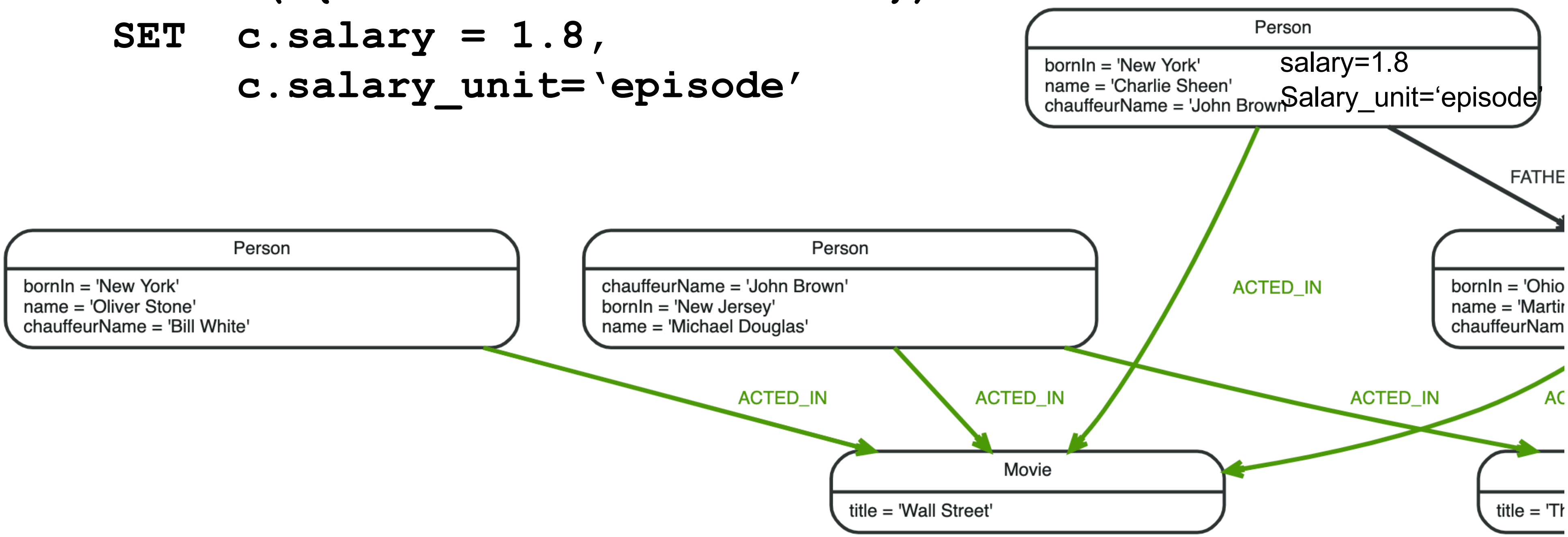


MERGE Clause: property

- `MERGE` *pattern*

  `SET` *properties/labels*

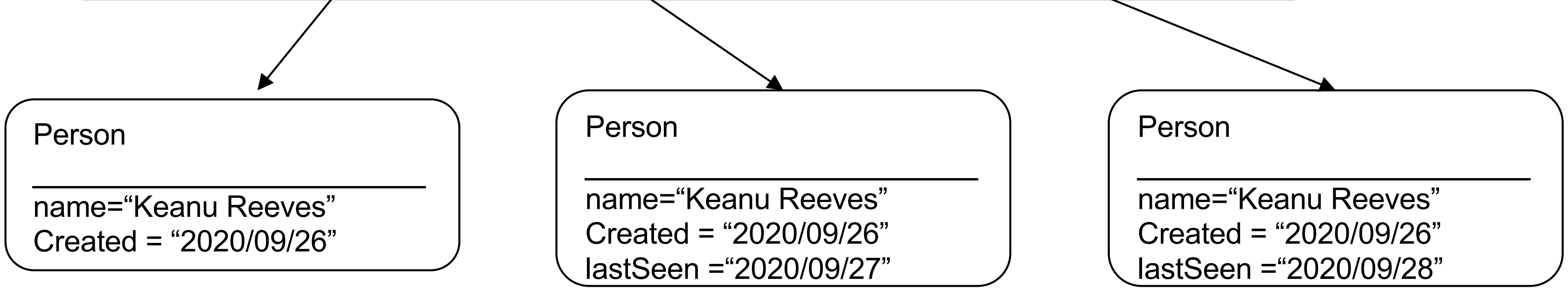
```
MERGE (c{ name: 'Charlie Sheen’})
SET  c.salary = 1.8,
     c.salary_unit=‘episode’
```



MERGE Clause: property

- Specifying different actions on insert and update

```
MERGE (keanu:Person { name: 'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
ON MATCH SET keanu.lastSeen = timestamp()
RETURN keanu.name, keanu.created, keanu.lastSeen
```



MERGE Clause: relationship

- **MATCH** *node_pattern(s)*

  MERGE *relationship_pattern*
- Example:

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(wallStreet:Movie { title: 'Wall Street' })
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
RETURN charlie.name, type(r), wallStreet.title
```

- **MATCH** *node_pattern*

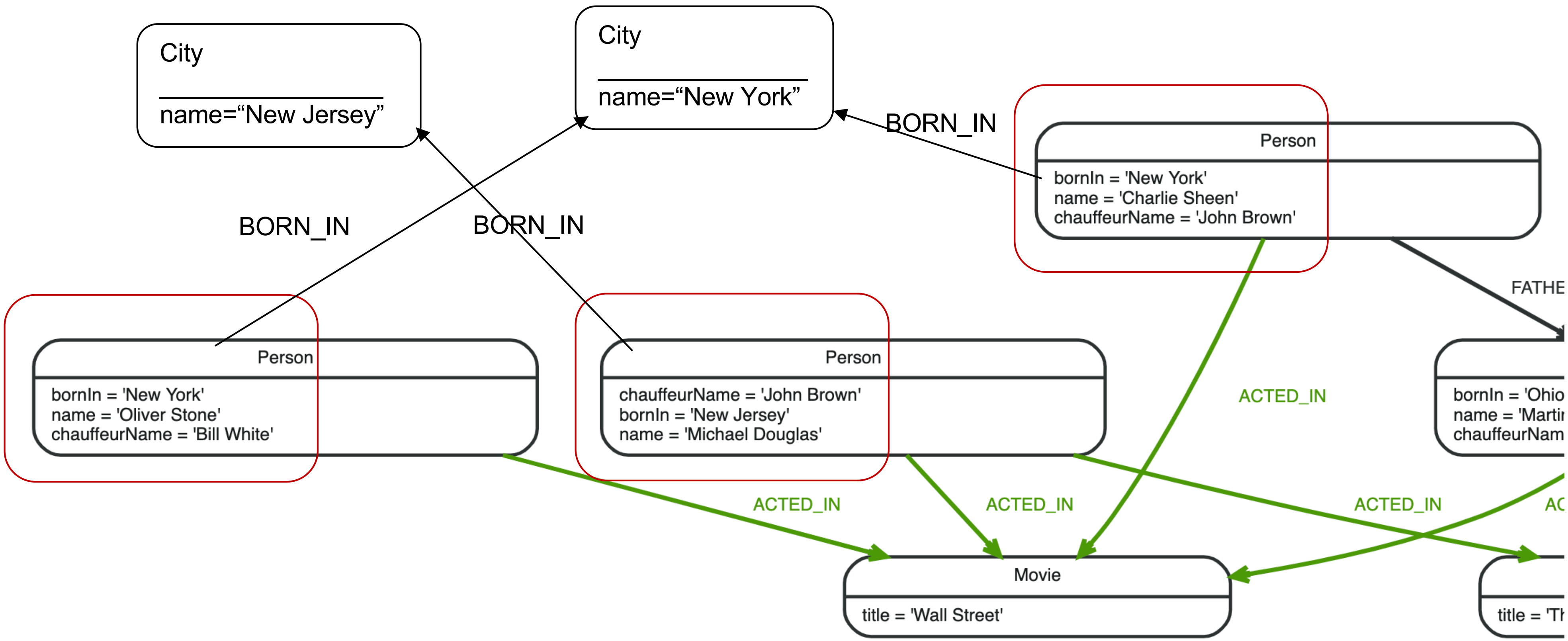
  MERGE *node_pattern*

  MERGE *relationship_pattern*

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city
```

Example Graph

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city
```




```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(wallStreet:Movie { title: 'Wall Street' })
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
RETURN charlie.name, type(r), wallStreet.title
```

MERGE Clause: Usage and Performance

- One major use case of `MERGE` is to create graph model from source data
  - CSV, JSON, XML, ..
- There are always gaps between source data format and the desirable graph model
  - Properties need to be extracted from columns and assigned
  - Relationships need to be built across different lines
- `MERGE` will be called repeatedly in building graph from raw data
  - Call MERGE multiple times per line of source data
- It is very important to build index before bulk loading with `MERGE`

Cypher - Delete

- **MATCH** ***pattern*** **DELETE** ***var-expression***

- Delete relationship

```
MATCH (n{name:"Keanu Reeves"})-[r:ACTS_IN]->()
DELETE r
```

- Delete a node and all possible relationship

```
MATCH (m{title:'The Matrix'})-[r]-()
DELETE m,r
```

Outline

- Brief Review of Graphs
- Property Graph Model
- **Cypher Query**
  - Patterns and basic clauses
  - **Subclause, function and Suqueries**

MATCH: sub-clauses

- The **WHERE** sub clause can be used to specify various query conditions
  - Boolean operators **AND**, **OR**, **NOT**, **XOR** can be used

```
MATCH (n)
WHERE n.age <30 AND n.employ>=3
RETURN n.name
```

  - It can be used to chain an existential sub queries, but you may find an easier way of writing the same query

```
MATCH (person:Person)
WHERE EXISTS {
  MATCH (person)-[:HAS_DOG]->(dog :Dog)
  WHERE person.name = dog.name
}
RETURN person.name as name
```

```
MATCH (person:Person)-[:HAS_DOG]→(dog :Dog)
WHERE person.name = dog.name
RETURN person.name as name
```

Functions

- Functions may appear in various clauses
  - Build-in and user-defined functions
- Build-in functions
  - **Predicate functions**
  - Scalar functions
  - **Aggregation functions**
  - List functions
  - Mathematical functions
  - String functions
  - Temporal functions
  - Spatial Functions

Predicate Functions

- They are boolean functions that return true or false for a given set of non-null input. They are most commonly used to filter out subgraphs in the WHERE part of a query.
  - **`all(), any(), exists(),none(),single()`**
- **`all()`** usage
  - **`all(variable IN list WHERE predicate)`**

Assign a variable to the entire path

```
MATCH p =(a)-[*1..3]->(b)
WHERE a.name = 'Alice' AND b.name = 'Daniel' AND ALL (x IN nodes(p) WHERE x.age > 30)
RETURN p
```

  - All nodes in the returned paths should have an age property of at least **'30'**.

A function returns all nodes of a path

- **`any(),single(),`** and **`none()`** have similar signature but different meanings

Predicate Functions

- **`exists()`** usage
  - `exists(pattern-or-property)`

```
MATCH (n)
WHERE EXISTS (n.name)
RETURN n.name AS name, EXISTS ((n)-[:MARRIED]->()) AS is_married
```

  - The names of all nodes with the name property are returned, along with a boolean true / false indicating if they are married.
  - The first **`EXISTS()`** function does filtering because it is used in WHERE clause
  - The second **`EXISTS()`** does not filter anything because it is used in the return clause, it only computes and returns value

Aggregating Functions

- GROUP BY feature in Neo4j is achieved using aggregating functions
  - E.g. **count()**, **sum()**, **avg()**, **max()**, **min()** and so on
- The grouping key is implied in the RETURN clause
  - None aggregate expression in the return clause is the grouping key
  - **RETURN n, count(*)**
    - **n** is a variable declared in a previous clause, and it is the grouping key
  - `MATCH(n:Person) RETURN n.gender, COUNT(*)`
    - Count the number of nodes representing each gender in the graph
    - A person's gender is the grouping key
- A grouping key is not always necessary, the aggregation function can apply to all results returned
  - `MATCH (n:Person) RETURN COUNT(*)`
    - To count the number of Person nodes in the graph

Aggregation Examples

- To find out the earliest year a Person was born in the data set

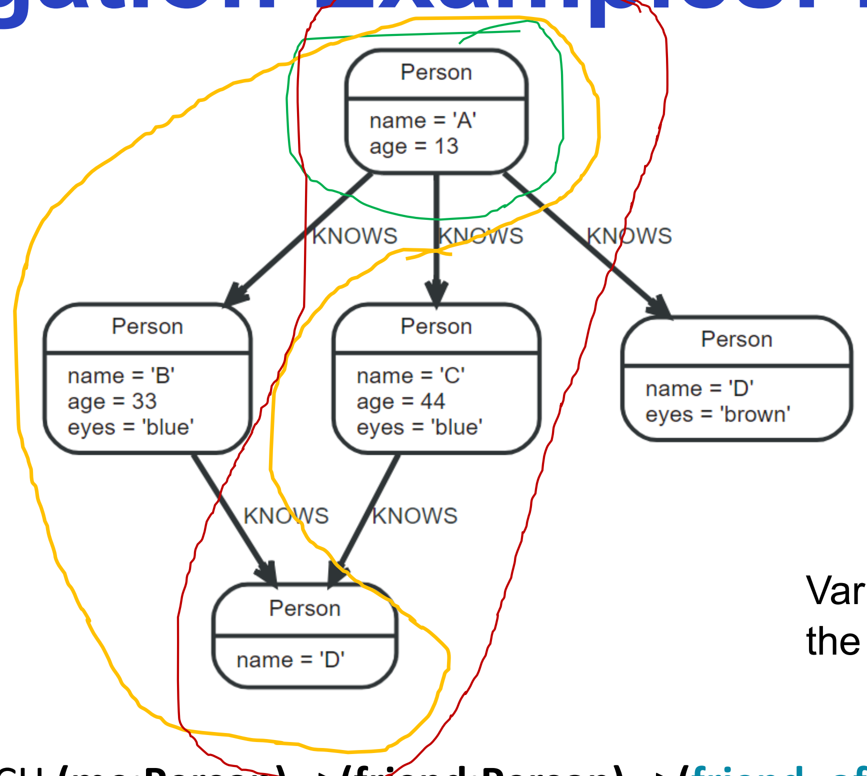
```
MATCH (n:Person) RETURN min (n.born)
```

- To find out the distribution of relationship types belonging to nodes with certain feature

```
MATCH (n { name: 'A' })-[r]->()
RETURN type(r), count(*)
```

```
The grouping key is type(r) which is a scalar function,
returns the type of relationship in the matching results
```

Aggregation Examples: DISTINCT



Variable **friend_of_friend** refers to the same node in both matches

MATCH **(me:Person)-->(friend:Person)-->(friend_of_friend:Person)**

WHERE me.name = 'A'

RETURN count(**DISTINCT** friend_of_friend), count(friend_of_friend)

count(DISTINCT friend_of_friend)	count(friend_of_friend)
1	2
1 row	

MATCH: subqueries

- The **WITH** clause can chain different query parts together in a pipeline style
  - Used to apply conditions on aggregation result
  - Used to modify (order, limiting, etc) the results before collecting them as a list
- Examples
  - Find the person who has directed 3 or more movies

```
MATCH (p:Person)-[r:DIRECTED]->(m:Movie)
WITH p, count(*) as movies
WHERE movies >= 3
RETURN p.name, movies
```

  - Return the oldest 3 person as a list

```
MATCH (n:Person)
WITH n
ORDER by n.age DESC LIMIT 3
RETURN collect(n.name)
```

```
MATCH (n:Person)
RETURN n.name
ORDER by n.age DESC LIMIT
3
```

Dealing with Array type

- Array literal is written in a similar way as it is in most programming languages
  - examples
    - An array of integer: `[1,2,3]`
    - An array of string: `[“Sydney”, “University”]`
- Both node and relationship can have property of array type
  - Example: create an relationship with array property

```
create (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix)
```

  - Example: update an existing node with array property

```
MATCH (n:Person{name: "Tom Hanks"})
set n.phone=["0123456789","93511234"]
```

Dealing with Array type (cont'd)

- Querying array property
  - The IN operator: check if a value is in an array
    - Example: find out who has played 'Neo' in which movie

      MATCH (a:Person) -[r:ACTED_IN]->(m:Movie)

      WHERE 'Neo' IN r.roles

      RETURN a , m

  - The UNWIND operator: flatten an array into multiple rows
    - Example: find all the movies released in 1999 or in 2003

      UNWIND [1999,2003] as year

      MATCH (m: Movie)

      WHERE m.released = year

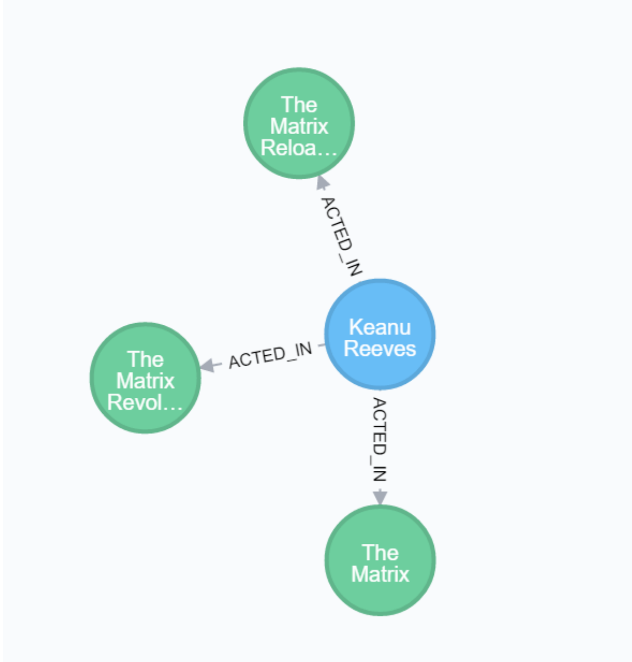
      RETURN m.title, m.released

      This is equivalent to

      MATCH(m: Movie)

      WHERE m.released IN [1999,2003]

      RETURN m.title, m.released



$ UNWIND [1999,2003] as year MATCH (m: Movie) WHERE m.releas...

Rows | Text | Code

m.title	m.released
The Matrix	1999
Snow Falling on Cedars	1999
The Green Mile	1999
Bicentennial Man	1999
The Matrix Reloaded	2003
The Matrix Revolutions	2003
Something's Gotta Give	2003

Returned 7 rows in 37 ms.

Dealing with Array Type (cont'd)

- A relatively complex query
  - Update another node

```
MATCH (n:Person{name: "Meg Ryan"}) set n.phone=["0123456789"]
```

  - Run a query to see who shares any phone number with Tom Hanks

```
MATCH (n:Person{name: "Tom Hanks"})
WITH   n.phone as phones, n
UNWIND phones as phone
MATCH  (m:Person)
WHERE  phone in m.phone and n<>m
RETURN m.name
```

Where to find more about cypher query:
Developer's guide: http://neo4j.com/docs/developer-manual/current/cypher/
Reference card: https://neo4j.com/docs/cypher-refcard/current/

Indexing

- Neo4j supports index on properties of labelled node
- Index has similar behaviour as those in relational systems
- Create Index
  - **CREATE INDEX ON :Person(name)**
- Drop Index
  - **DROP INDEX ON :Person(name)**
- Storage and query execution will be covered in week 7

References

- Ian Robinson, Jim Webber and Emil Eifrem, *Graph Databases*, Second Edition, O'Reilly Media Inc., June 2015
  - You can download this book from the Neo4j site, http://www.neo4j.org/learn will redirect you to http://graphdatabases.com/
- The Neo4j Document
  - The Neo4j Graph Database Concept (http://neo4j.com/docs/stable/graphdb-neo4j.html)
  - Cypher manual (https://neo4j.com/docs/cypher-manual/current/introduction/)
- Noel Yuhanna, *Market Overview: Graph Databases,* Forrester White Paper, May, 2015
- Renzo Angeles, *A Comparison of Current Graph Data Models*, ICDE Workshops 2013 (DOI-10.1109/ICDEW.2012.31)
- Renzo Angeles and Claudio Gutierrez, *Survey of Graph Database Models*, ACM Computing Surveys, Vol. 40, N0. 1, Article 1, February 2008 (DOI-10.1145/1322432.1322433)