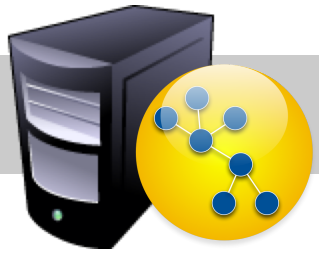
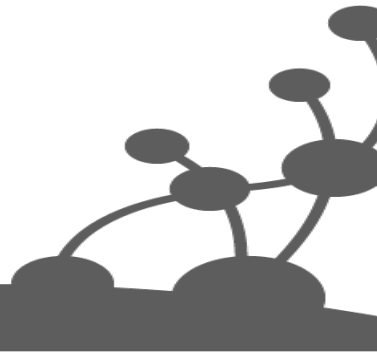
Data Modeling With Neo4j

Topics

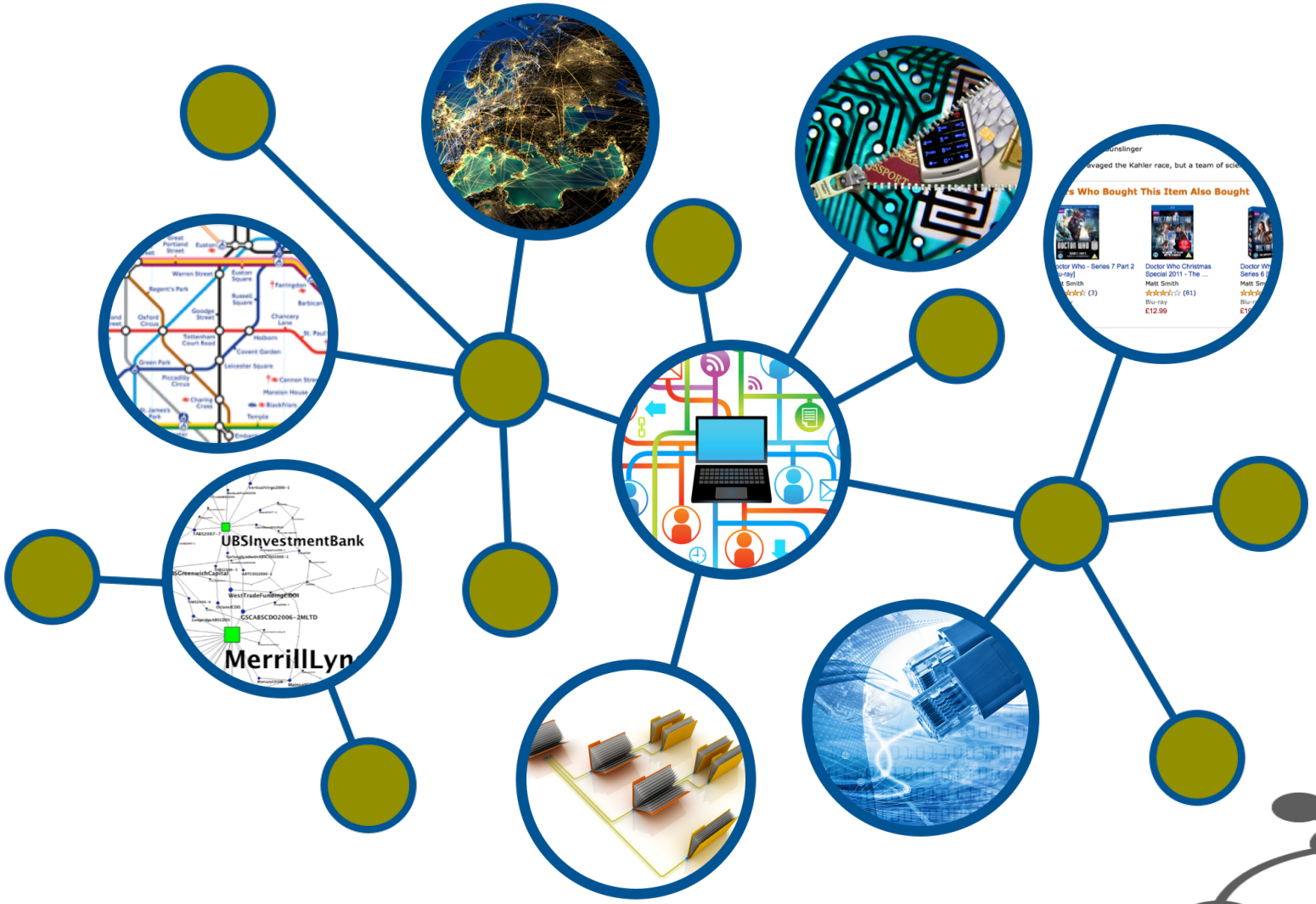


- Data complexity
- Graph model building blocks
- Quick intro to Cypher
- Modeling guidelines
- Common graph structures
- Evolving a graph model

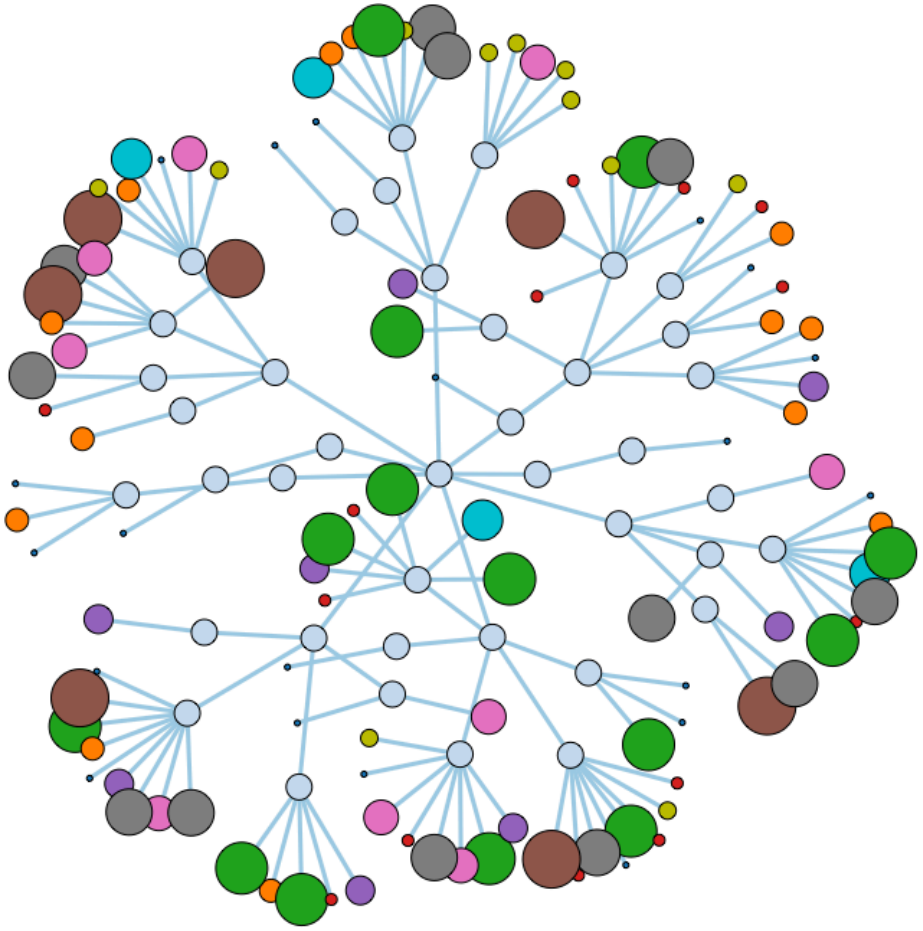


Addressing Data Complexity With Graphs

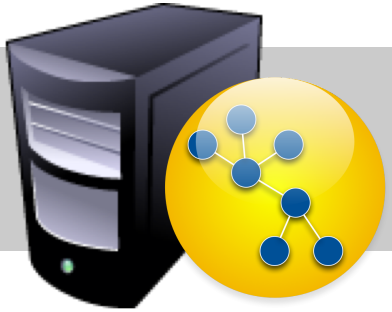
complexity = f(size, variable structure, connectedness)



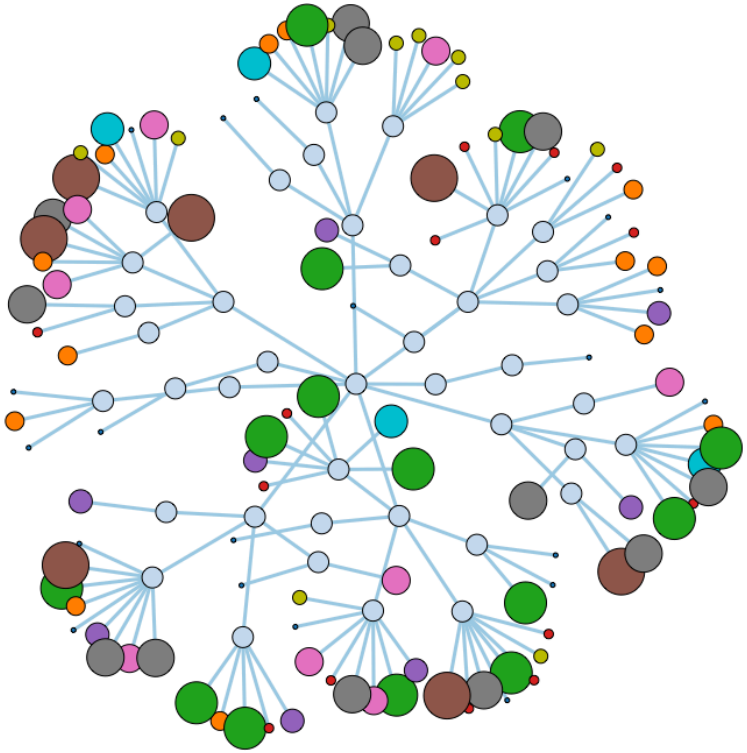
Graphs Are Everywhere



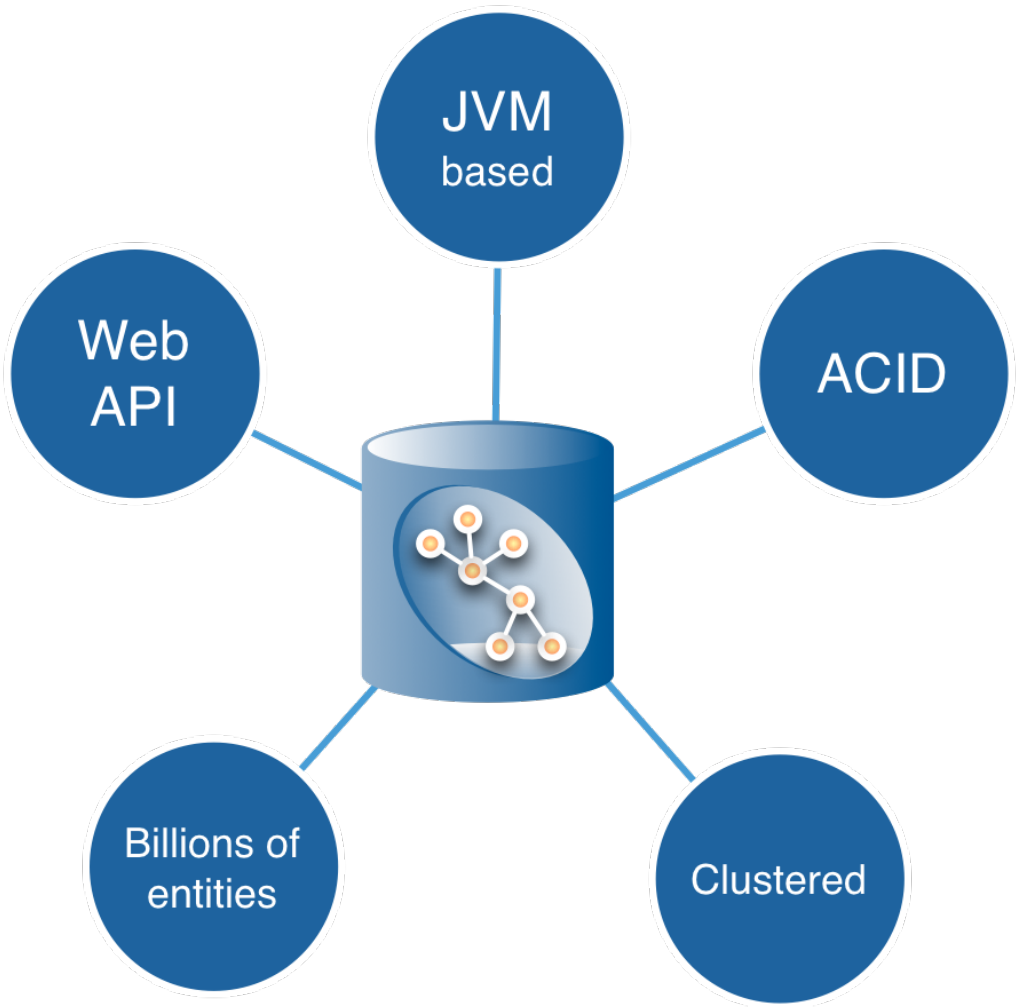
Graph Databases



- Store
- Manage
- Query

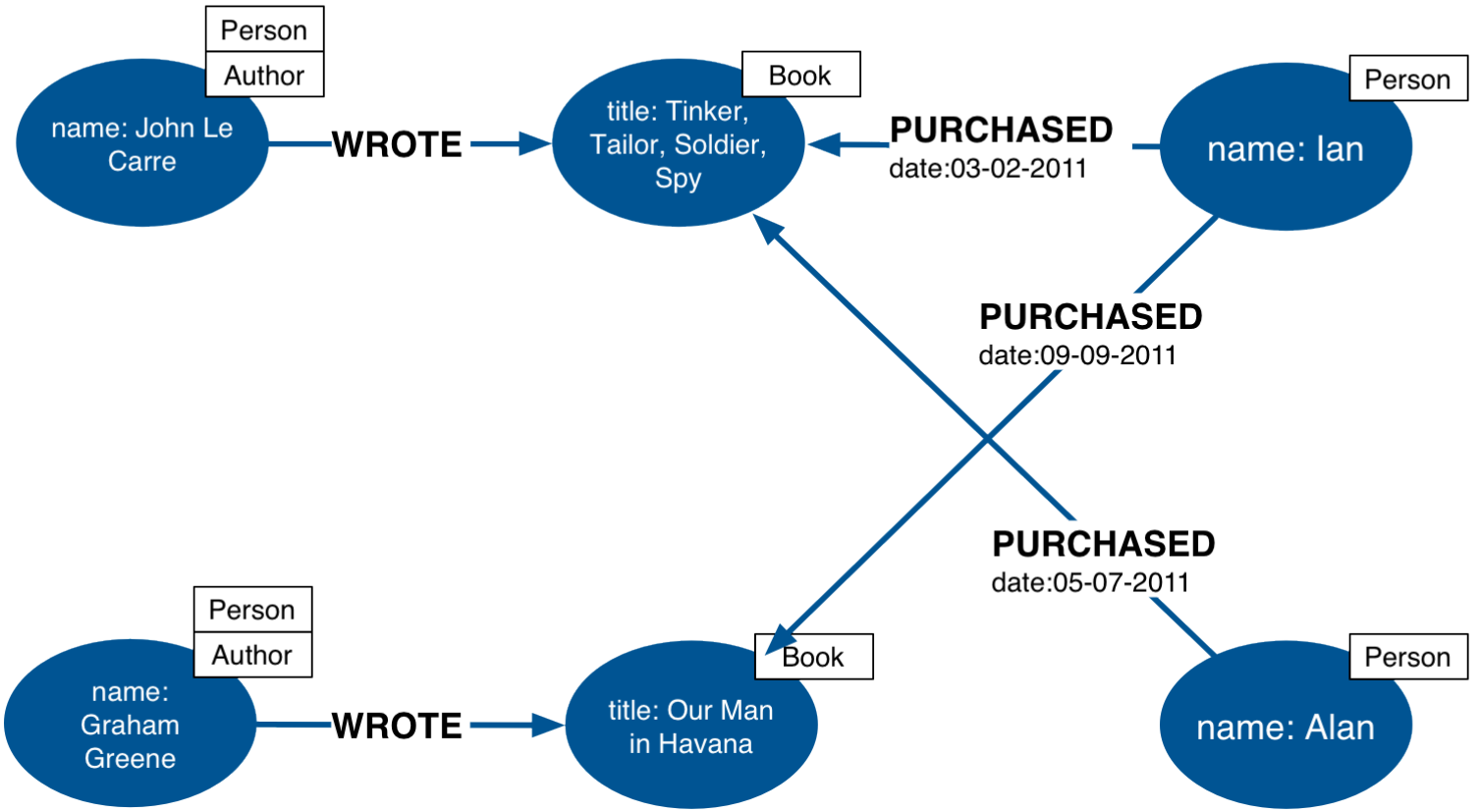


Neo4j is a Graph Database



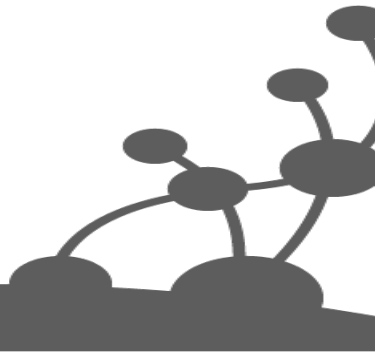
Graph Model Building Blocks

Labeled Property Graph Data Model

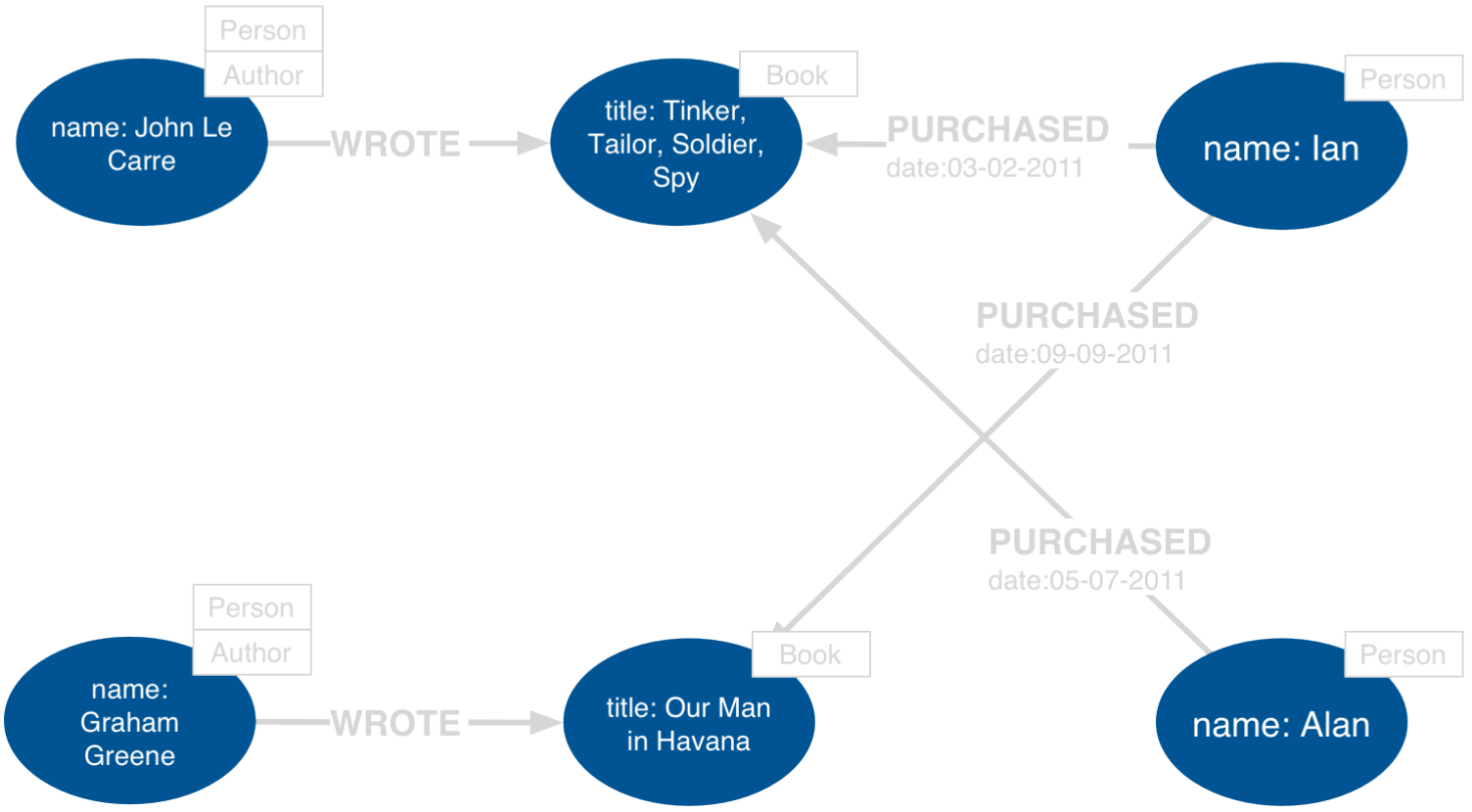


Four Building Blocks

- Nodes
- Relationships
- Properties
- Labels

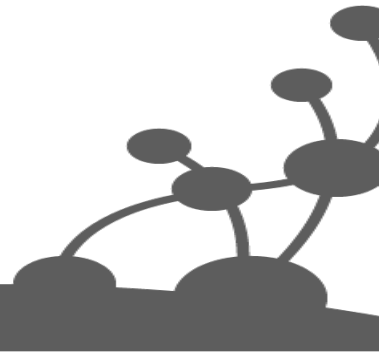


Nodes


Person
Author
name: John Le Carre
WROTE
Book
title: Tinker, Tailor, Soldier, Spy
PURCHASED
date:03-02-2011
Person
name: Ian
PURCHASED
date:09-09-2011
PURCHASED
date:05-07-2011
Person
Author
name: Graham Greene
WROTE
Book
title: Our Man in Havana
Person
name: Alan

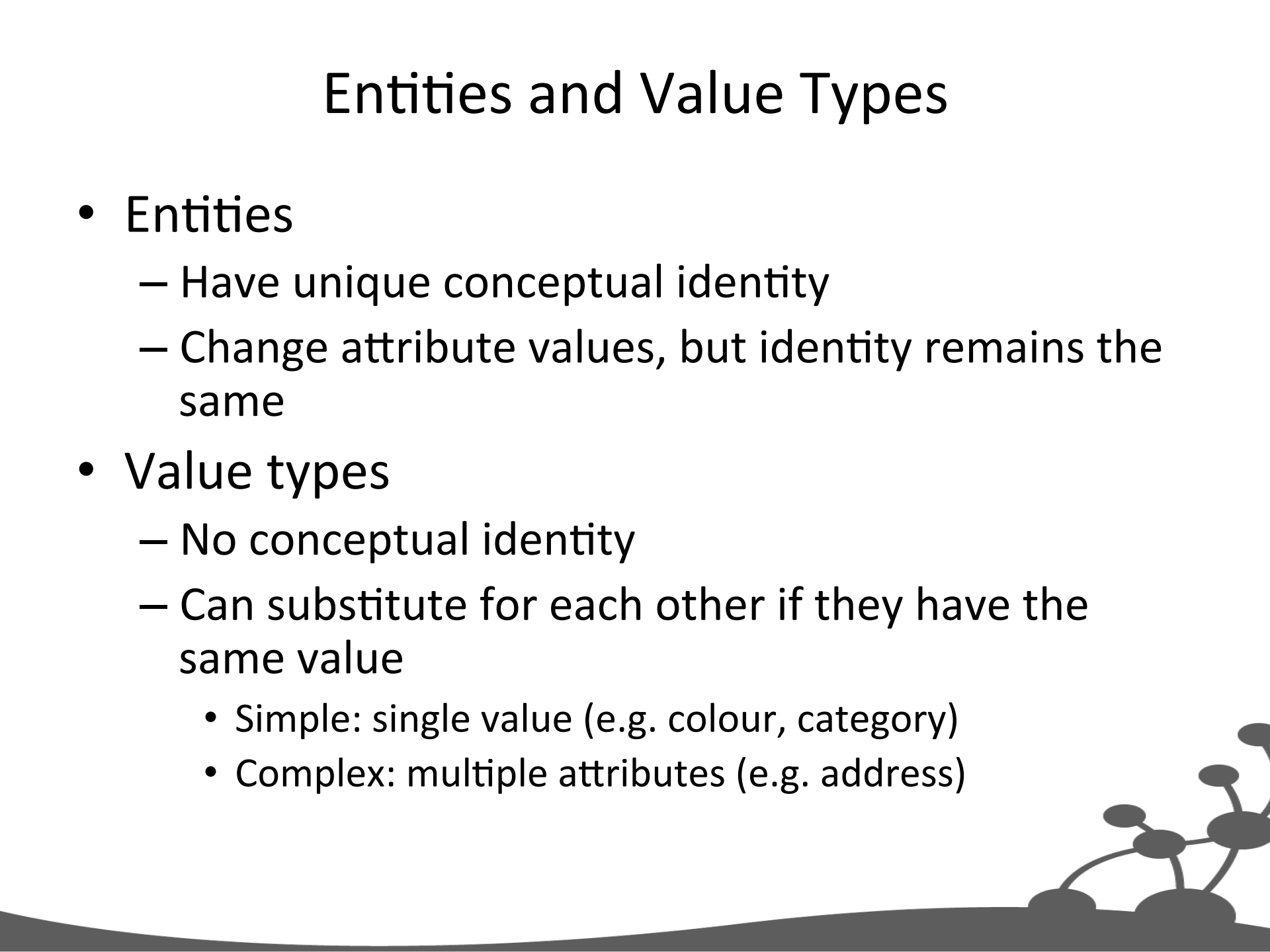
Nodes

- Used to represent *entities* and *complex value types* in your domain
- Can contain properties
  - Used to represent entity *attributes* and/or *metadata* (e.g. timestamps, version)
  - Key-value pairs
    - Java primitives
    - Arrays
    - `null` is not a valid value
  - Every node can have different properties

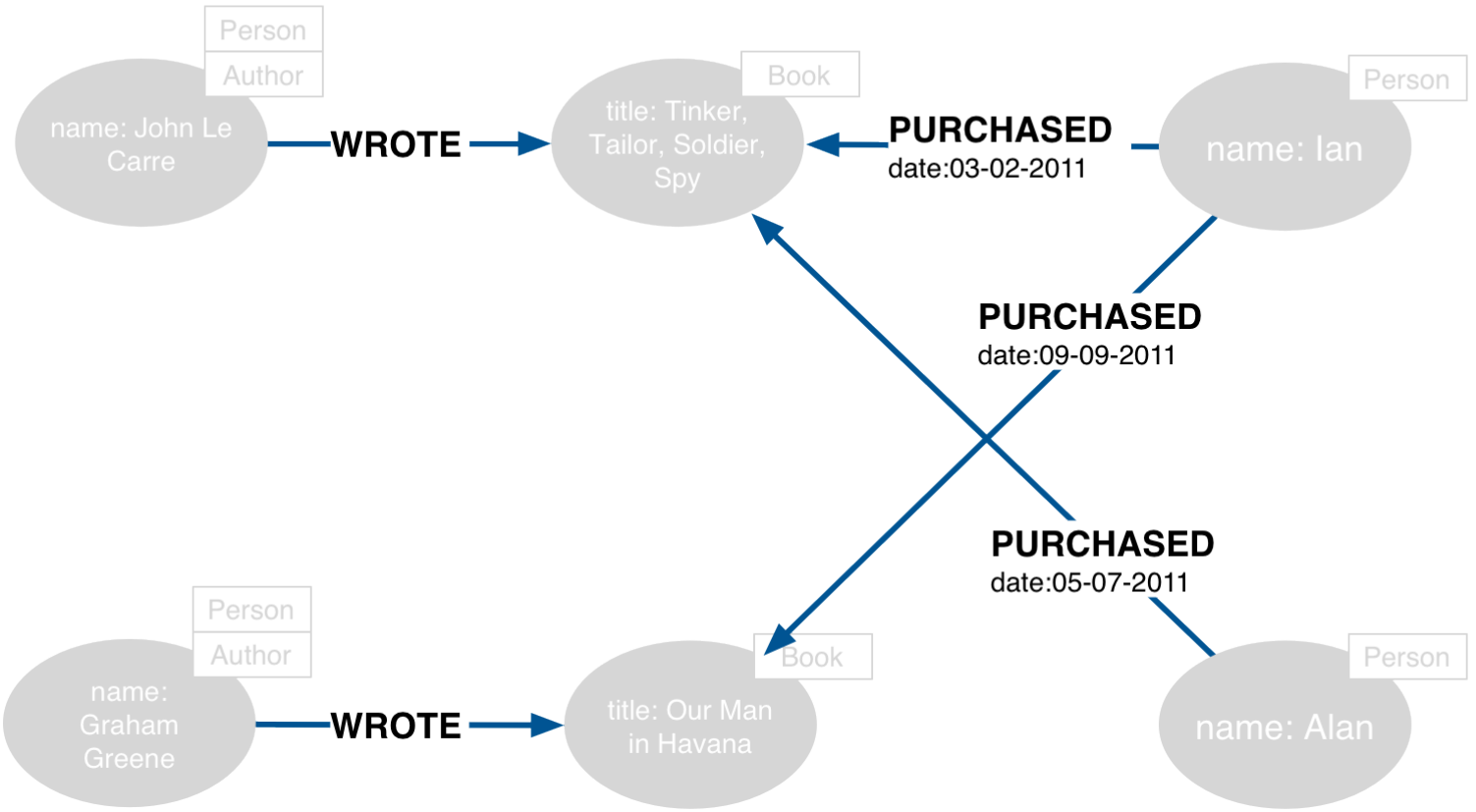


Entities and Value Types

- Entities
  - Have unique conceptual identity
  - Change attribute values, but identity remains the same
- Value types
  - No conceptual identity
  - Can substitute for each other if they have the same value
    - Simple: single value (e.g. colour, category)
    - Complex: multiple attributes (e.g. address)

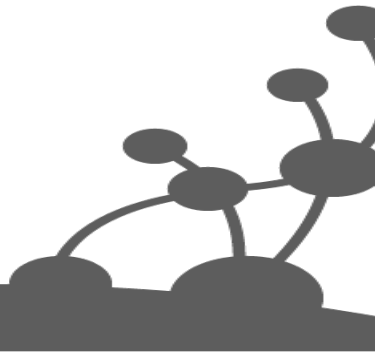


Relationships

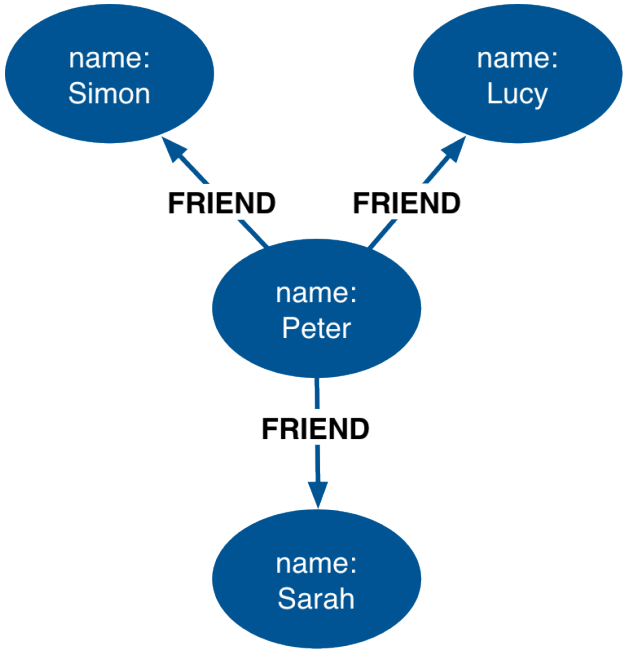


Relationships

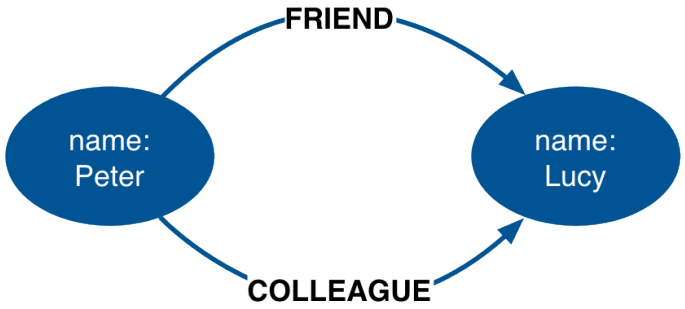
- Every relationship has a *name* and a *direction*
  - Add structure to the graph
  - Provide semantic context for nodes
- Can contain properties
  - Used to represent *quality* or *weight* of relationship, or *metadata*
- Every relationship must have a *start node* and *end node*
  - No dangling relationships



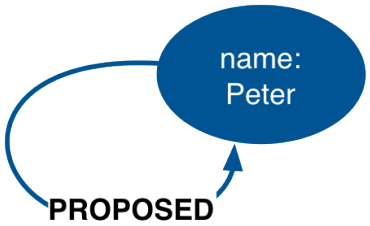
Relationships (continued)



Nodes can have more than one relationship



Nodes can be connected by more than one relationship

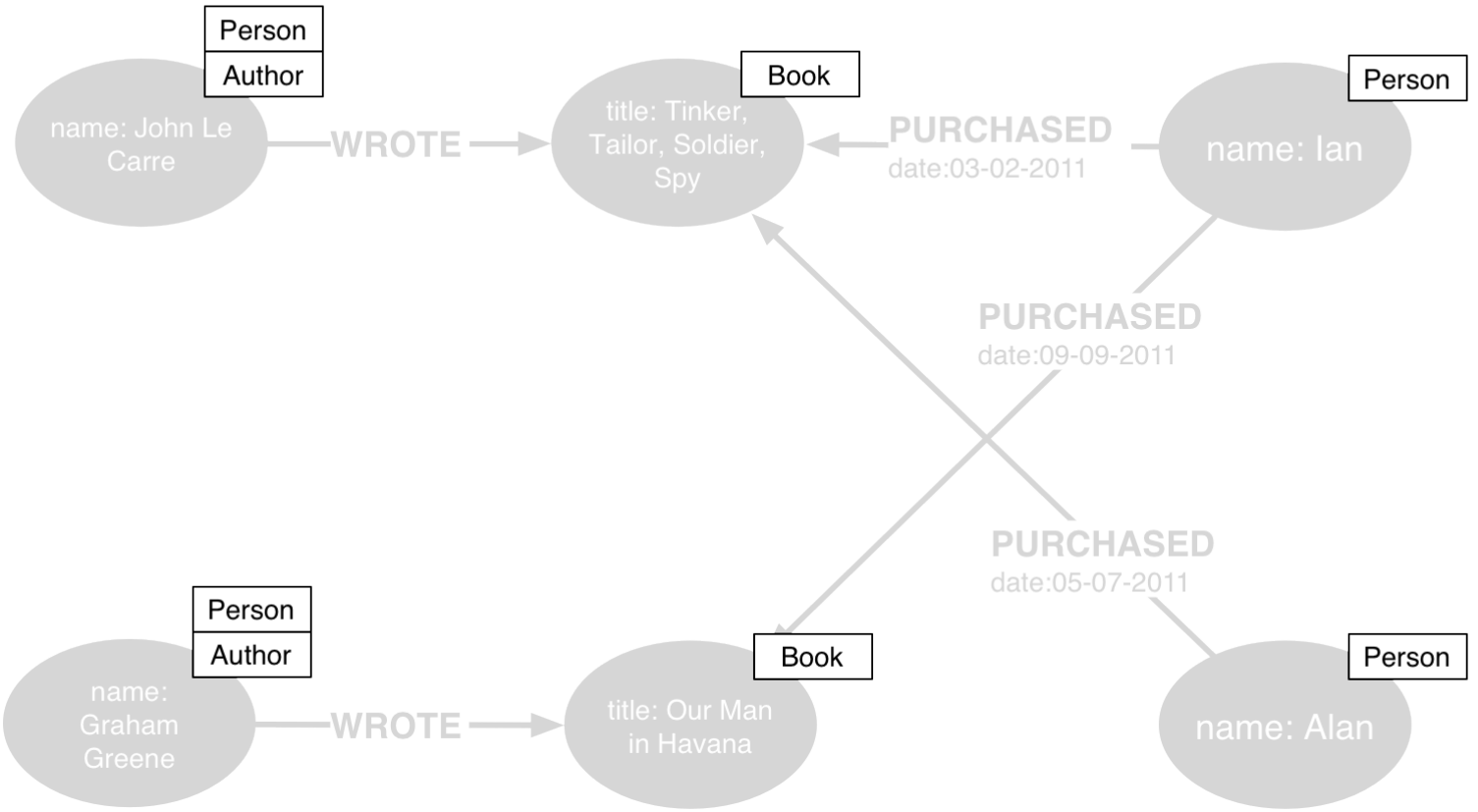


Self relationships are allowed

Variable Structure

- Relationships are defined with regard to node *instances*, not *classes* of nodes
  - Two nodes representing the same kind of "thing" can be connected in very different ways
    - Allows for structural variation in the domain
  - Contrast with relational schemas, where foreign key relationships apply to all rows in a table
    - No need to use *null* to represent the absence of a connection

Labels

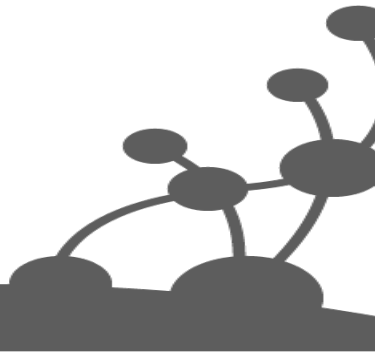

Person
Author
name: John Le Carre
WROTE
Book
title: Tinker, Tailor, Soldier, Spy
PURCHASED
date:03-02-2011
Person
name: Ian
PURCHASED
date:09-09-2011
PURCHASED
date:05-07-2011
Person
Author
name: Graham Greene
WROTE
Book
title: Our Man in Havana
Person
name: Alan

Labels

- Every node can have *zero or more* labels
- Used to represent *roles* (e.g. user, product, company)
  - Group nodes
  - Allow us to associate *indexes* and *constraints* with groups of nodes

Four Building Blocks

- Nodes
  - Entities
- Relationships
  - Connect entities and structure domain
- Properties
  - Entity attributes, relationship qualities, and metadata
- Labels
  - Group nodes by role



Cypher Query Language

Nodes and Relationships

()-->()

Labels and Relationship Types

```
(:Person)-[:FRIEND]->(:Person)
```

Properties

```
(:Person{name:'Peter'})-[:FRIEND]->(:Person{name:'Lucy'})
```

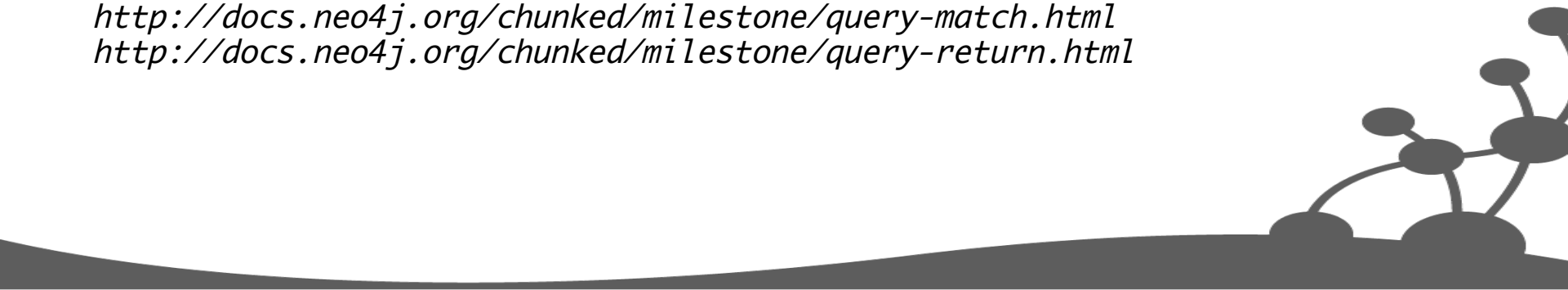
Identifiers

```
(p1:Person{name:'Peter'})-[r:FRIEND]->(p2:Person{name:'Lucy'})
```

Cypher

```
MATCH  graph_pattern
RETURN results
```

http://docs.neo4j.org/chunked/milestone/query-match.html
http://docs.neo4j.org/chunked/milestone/query-return.html



Example Query

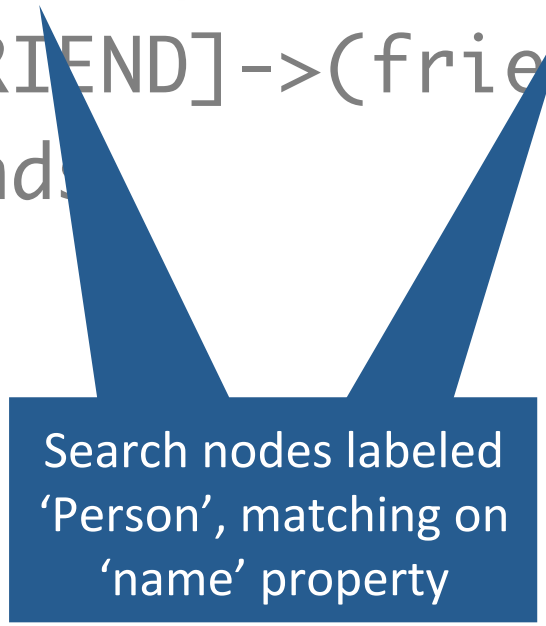
```
MATCH  (:Person{name:'Peter'})
       -[:FRIEND]->(friends)
RETURN friends
```

Find This Pattern

```
MATCH  (:Person{name:'Peter'})
       -[:FRIEND]->(friends)
RETURN friends
```

Lookup Using Identifier + Label

```
MATCH   (:Person{name:'Peter'})
        -[:FRIEND]->(friends)
RETURN friends
```



Return Nodes

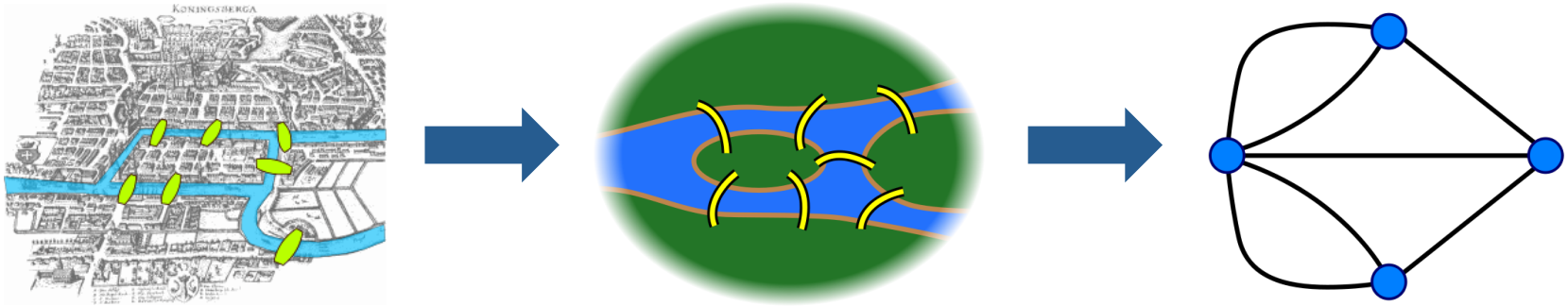
```
MATCH  (:Person{name:'Peter'})
       -[:FRIEND]->(friends)
RETURN friends
```

Exercise 1

Modeling Example

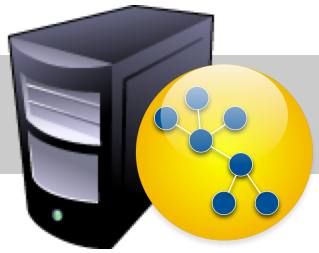
Models

Purposeful abstraction of a domain designed to satisfy particular application/end-user goals



Images: en.wikipedia.org

Example Application



- Knowledge management
  - People, companies, skills
  - Cross organizational
- Find my professional social network
  - Exchange knowledge
  - Interest groups
  - Help
  - Staff projects

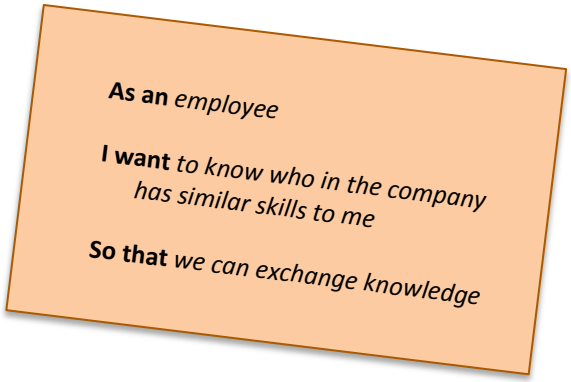
Application/End-User Goals

As an *employee*

I want *to know who in the company has similar skills to me*

So that *we can exchange knowledge*

Questions To Ask of the Domain



Which people, who work for the same company as me, have similar skills to me?

Identify Entities

Which people, who work for the same company as me, have similar skills to me?

Person

Company

Skill

Identify Relationships Between Entities

Which people, who work for the same company as me, have similar skills to me?

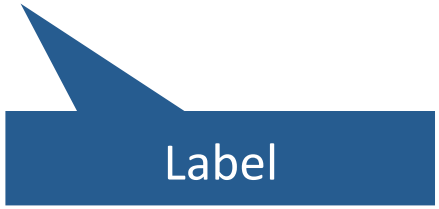
Person WORKS_FOR Company

Person HAS_SKILL Skill

Convert to Cypher Paths

Person WORKS_FOR Company

Person HAS_SKILL Skill

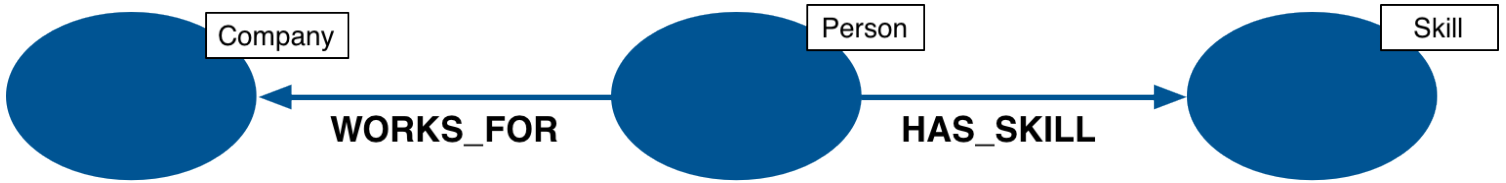


```
(:Person)-[:WORKS_FOR]->(:Company),
(:Person)-[:HAS_SKILL]->(:Skill)
```

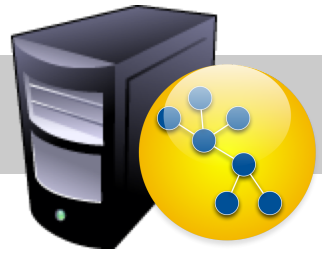
Consolidate Paths

```
(:Person)-[:WORKS_FOR]->(:Company),
(:Person)-[:HAS_SKILL]->(:Skill)
```

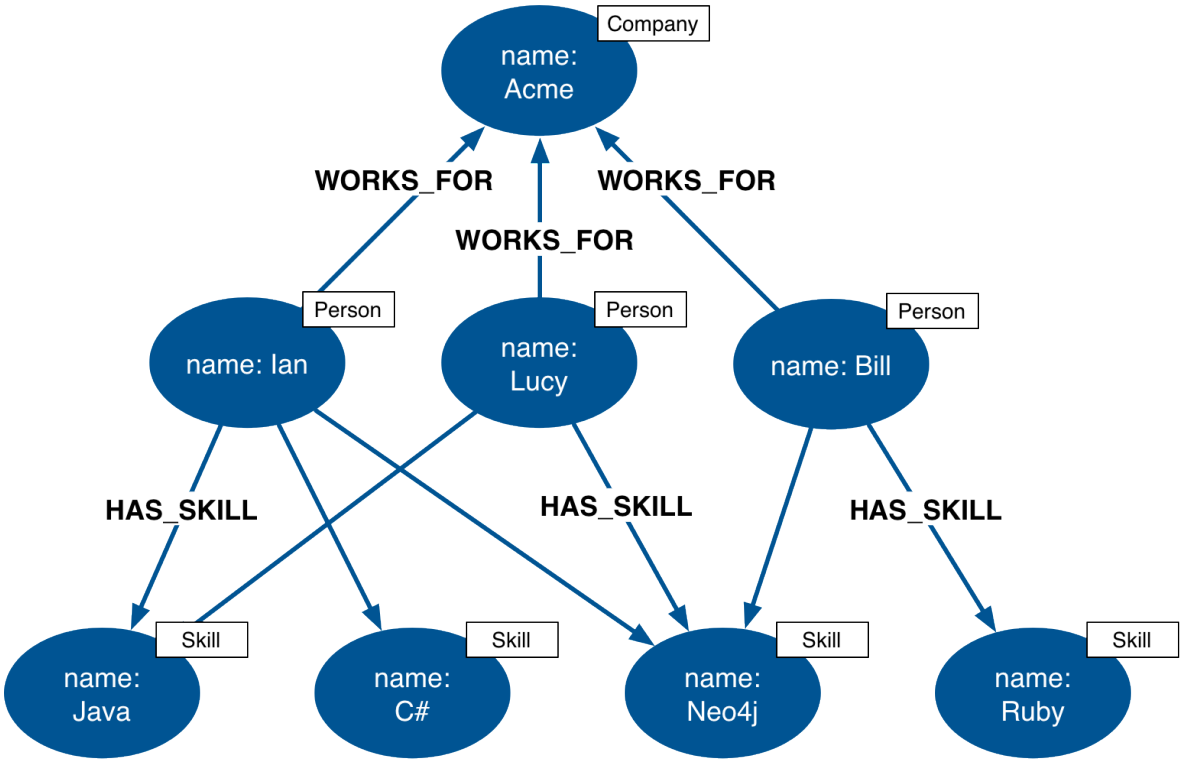
```
(:Company)<-[:WORKS_FOR]-(:Person)-[:HAS_SKILL]->(:Skill)
```



Candidate Data Model

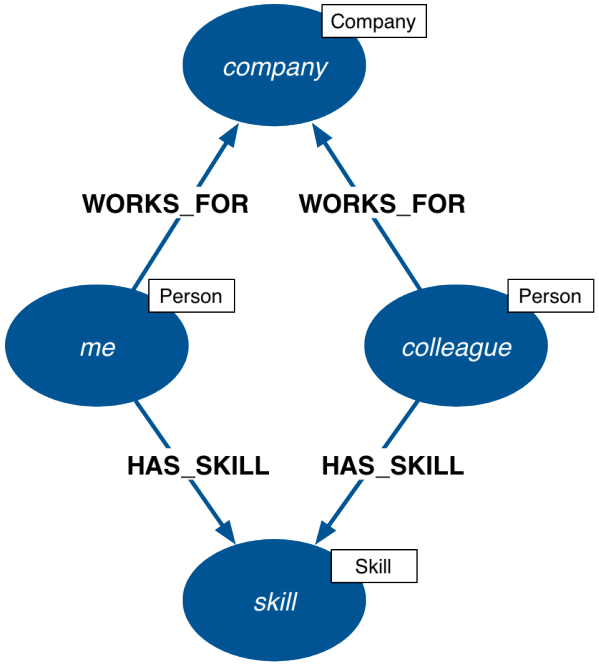


```
(:Company)<-[:WORKS_FOR]-(:Person)-[:HAS_SKILL]->(:Skill)
```

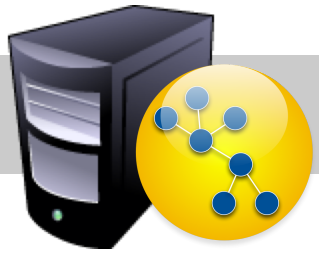


Express Question as Graph Pattern

Which people, who work for the same company as me, have similar skills to me?

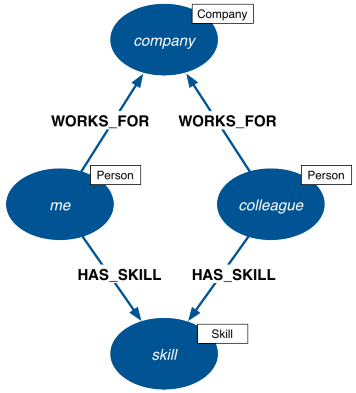


Cypher Query



Which people, who work for the same company as me, have similar skills to me?

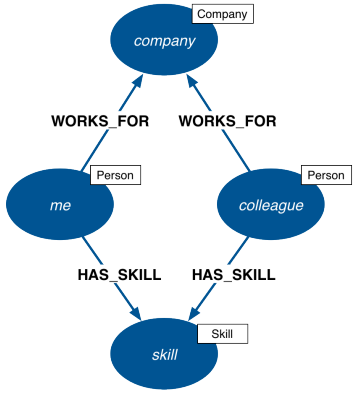
```
MATCH (company)<-[:WORKS_FOR]-(:Person{name:'Ian'})
                 -[:HAS_SKILL]->(skill),
      (company)<-[:WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
RETURN colleague.name AS name,
       count(skill) AS score,
       collect(skill.name) AS skills
ORDER BY score DESC
```



Graph Pattern

Which people, who work for the same company as me, have similar skills to me?

```
MATCH (company)<-[:WORKS_FOR]-(:Person{name:'Ian'})
                 -[:HAS_SKILL]->(skill),
      (company)<-[:WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
RETURN colleague.name AS name,
       count(skill) AS score,
       collect(skill.name) AS skills
ORDER BY score DESC
```

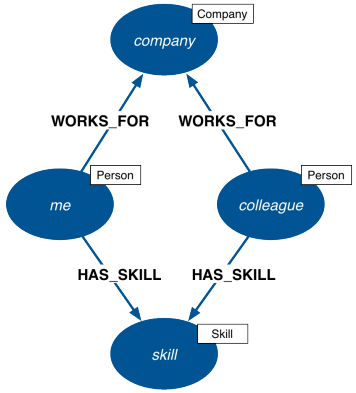


Anchor Pattern in Graph

Which people, who work for the same company as me, have similar skills to me?

```
MATCH (company)<-[:WORKS_FOR]-(:Person{name:'Ian'})
                -[:HAS_SKILL]->(skill),
      (company)<-[:WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
RETURN colleague.name AS name,
       count(skill) AS score,
       collect(skill.name) AS skills
ORDER BY score DESC
```

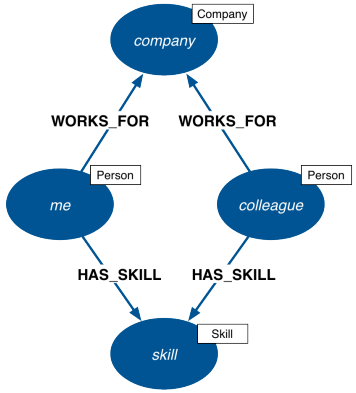
Search nodes labeled ‘Person’, matching on ‘name’ property



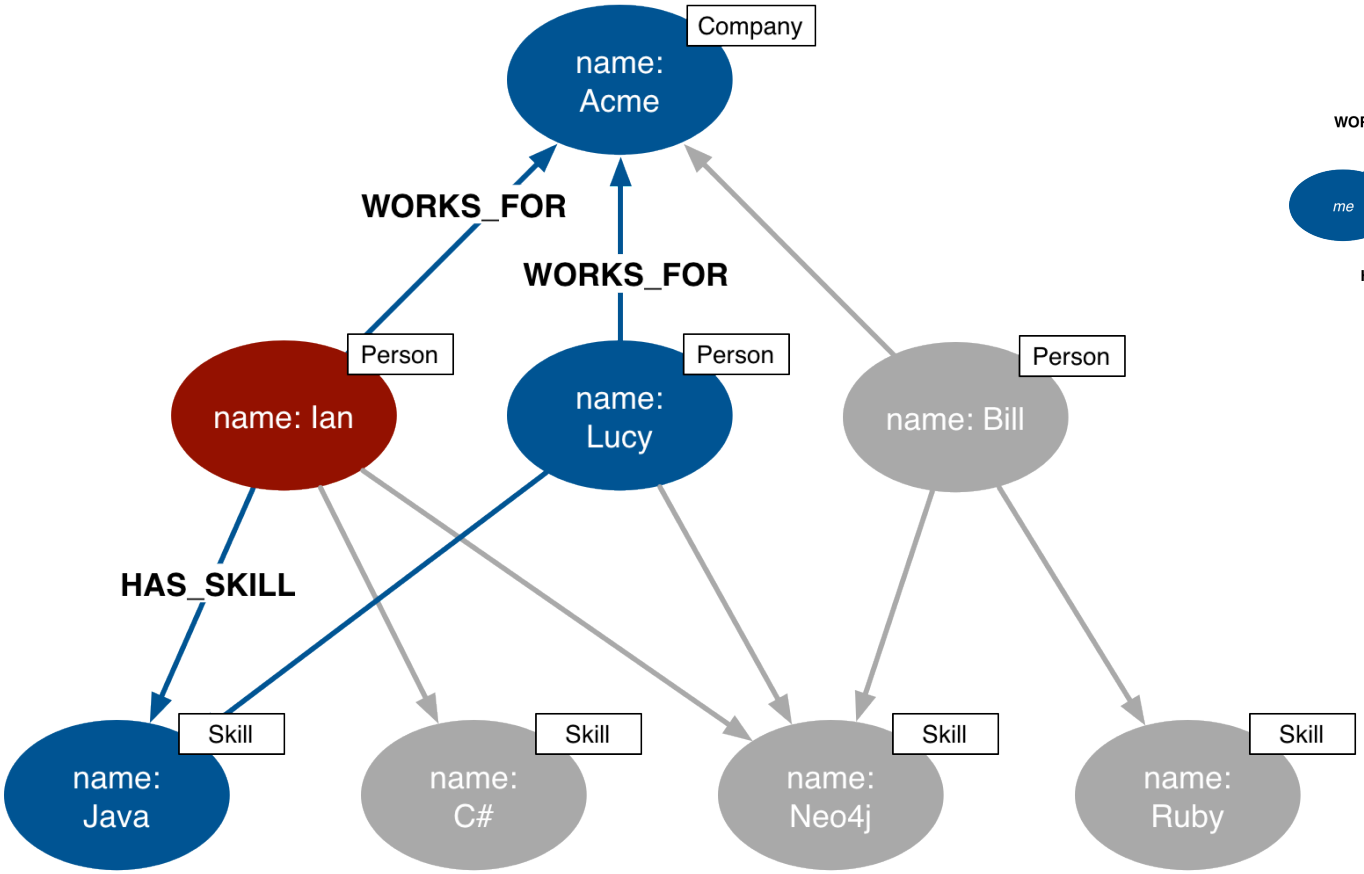
Create Projection of Results

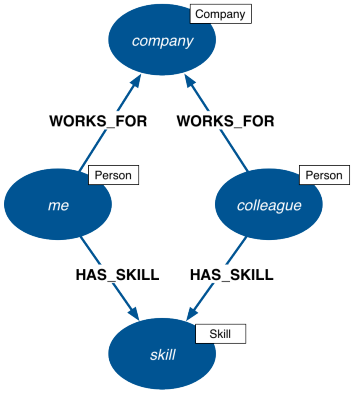
Which people, who work for the same company as me, have similar skills to me?

```
MATCH (company)<-[:WORKS_FOR]-(:Person{name:'Ian'})
                -[:HAS_SKILL]->(skill),
      (company)<-[:WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
RETURN colleague.name AS name,
       count(skill) AS score,
       collect(skill.name) AS skills
ORDER BY score DESC
```

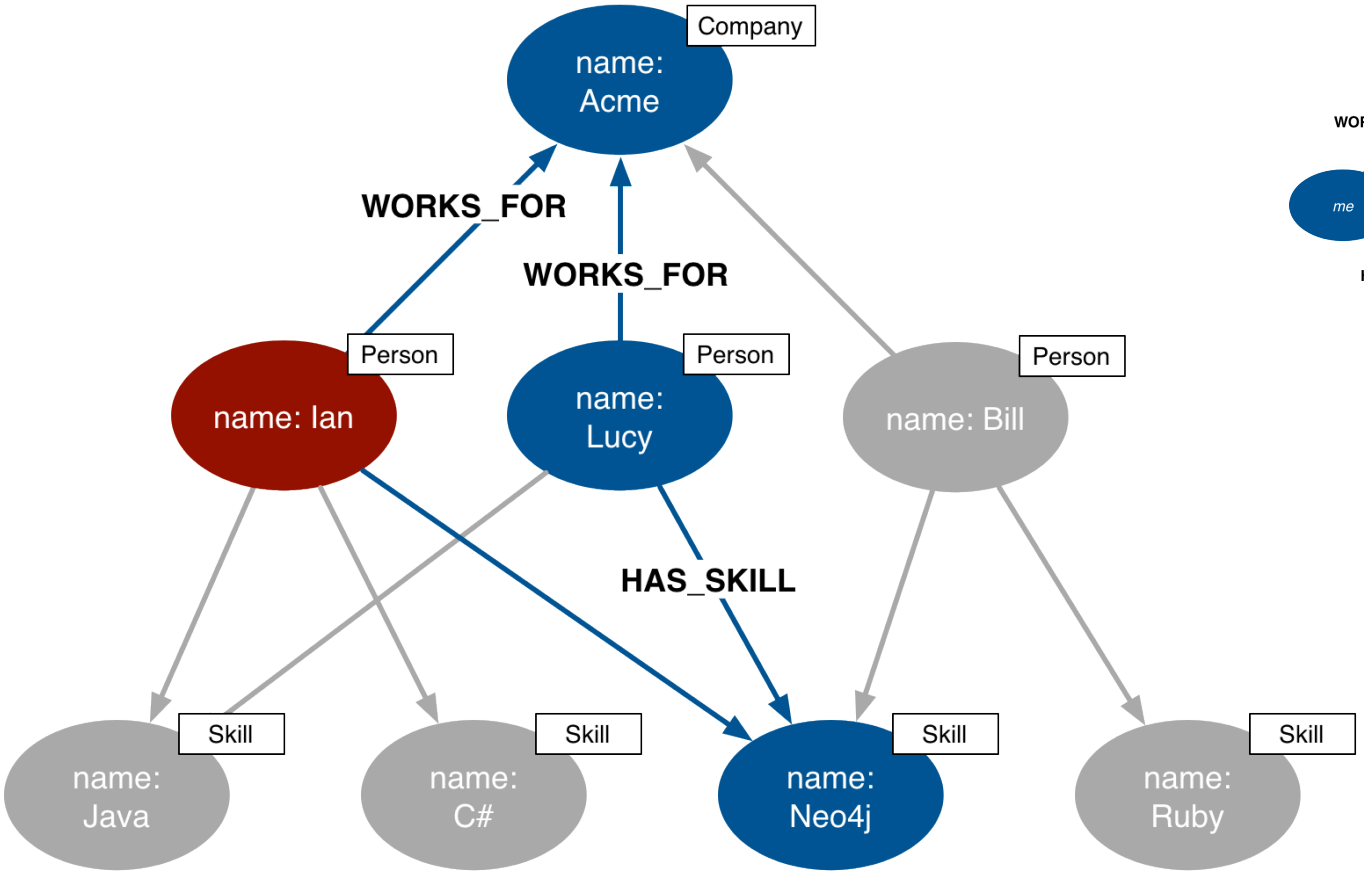


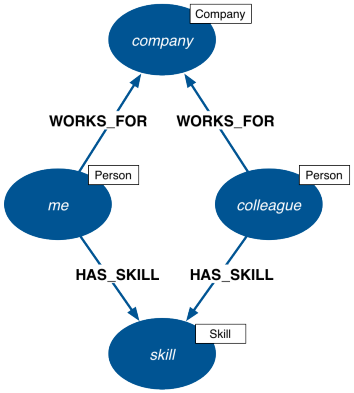
First Match


Company
name: Acme
WORKS_FOR
WORKS_FOR
Person
name: Ian
Person
name: Lucy
Person
name: Bill
HAS_SKILL
Skill
name: Java
Skill
name: C#
Skill
name: Neo4j
Skill
name: Ruby

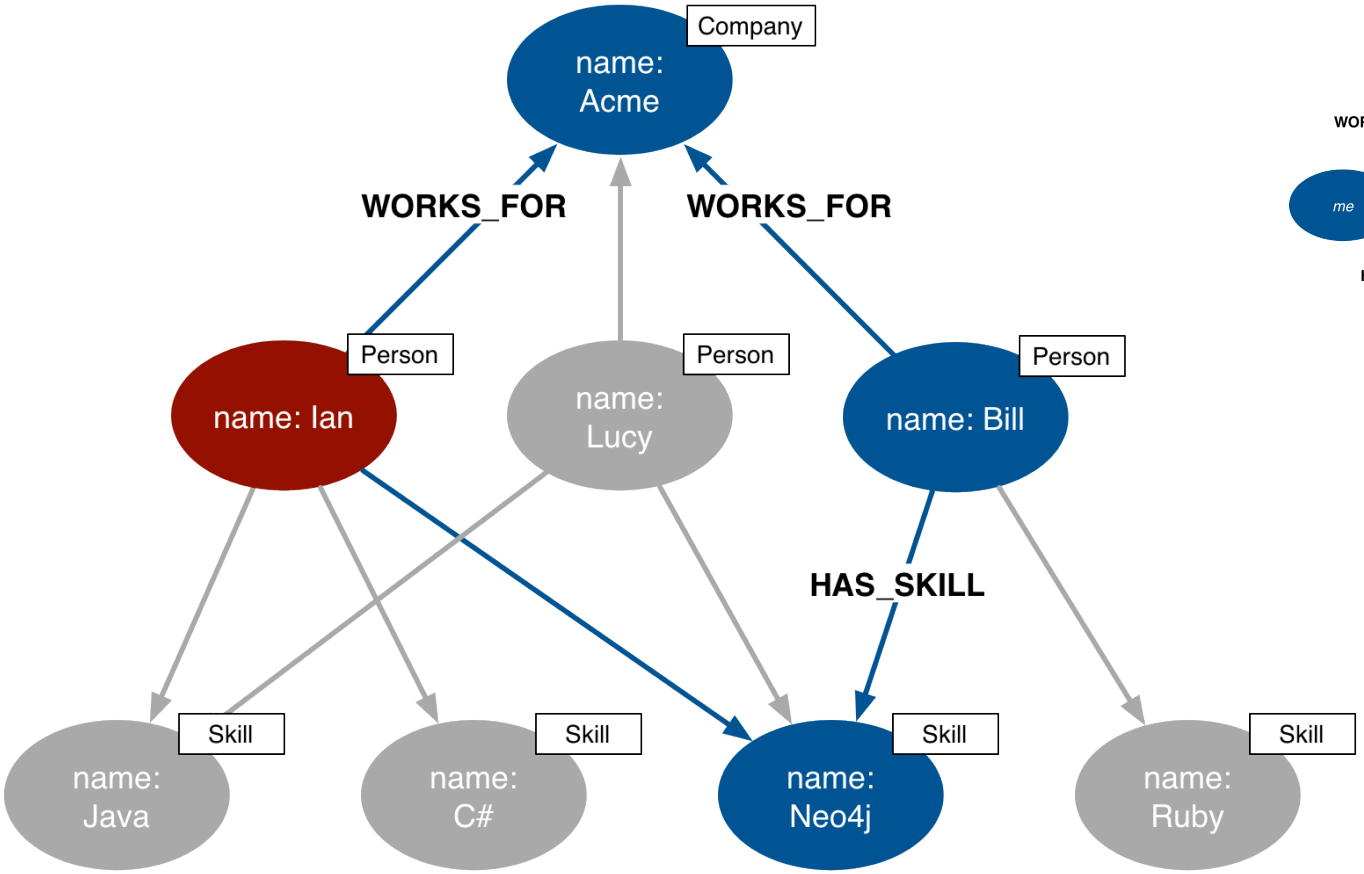

Company
company
WORKS_FOR
WORKS_FOR
Person
me
Person
colleague
HAS_SKILL
HAS_SKILL
Skill
skill

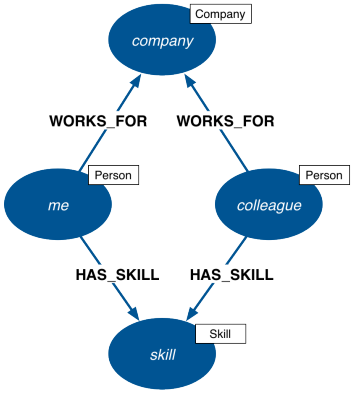
Second Match





Third Match


Company
name: Acme
WORKS_FOR
WORKS_FOR
Person
name: Ian
Person
name: Lucy
Person
name: Bill
HAS_SKILL
Skill
name: Java
Skill
name: C#
Skill
name: Neo4j
Skill
name: Ruby


Company
company
WORKS_FOR
WORKS_FOR
Person
me
Person
colleague
HAS_SKILL
HAS_SKILL
Skill
skill

Running the Query

```
+-----------------------------------+
| name    | score | skills          |
+-----------------------------------+
| "Lucy" | 2     | ["Java","Neo4j"] |
| "Bill" | 1     | ["Neo4j"]        |
+-----------------------------------+
2 rows
```

From User Story to Model and Query

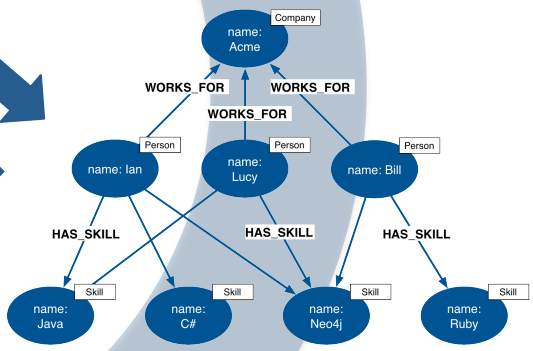
As an *employee*

I want *to know who in the company has similar skills to me*

So that *we can exchange knowledge*

```
MATCH (company)<-[:WORKS_FOR]-(me:Person)-[:HAS_SKILL]->(skill),
      (company)<-[:WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
WHERE  me.name = {name}
RETURN colleague.name AS name,
       count(skill) AS score,
       collect(skill.name) AS skills
ORDER BY score DESC
```

Which people, who work for the same company as me, have similar skills to me?



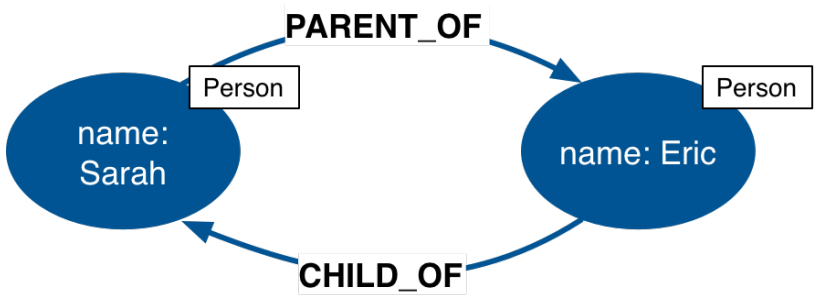
Person WORKS_FOR Company

Person HAS_SKILL Skill

```
(:Company)<-[:WORKS_FOR]-(:Person)-[:HAS_SKILL]->(:Skill)
```

Modeling Guidelines

Symmetric Relationships



Infer Symmetric Relationship

Find child:

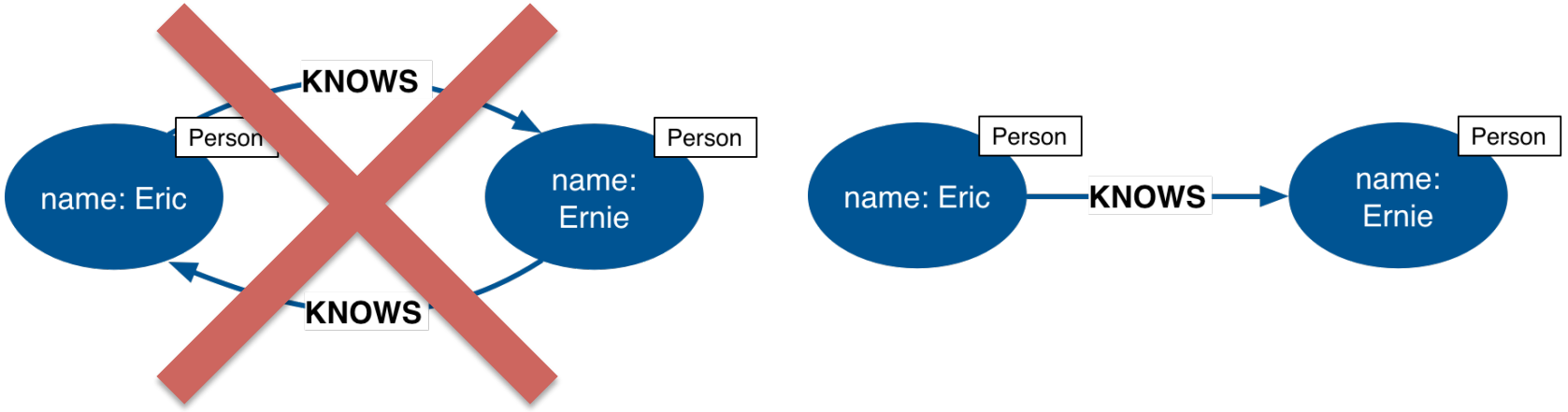
```
MATCH (parent{name:'Sarah'})
      -[:PARENT_OF]->(child)
RETURN child
```

Find parent:

```
MATCH (parent)-[:PARENT_OF]->
      (child{name:'Eric'})
RETURN parent
```

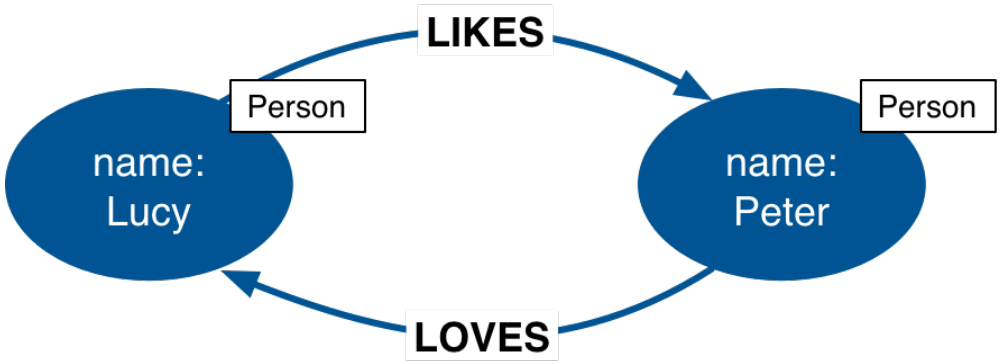
Bi-Directional Relationships

Use Single Relationship and Ignore Relationship Direction in Queries

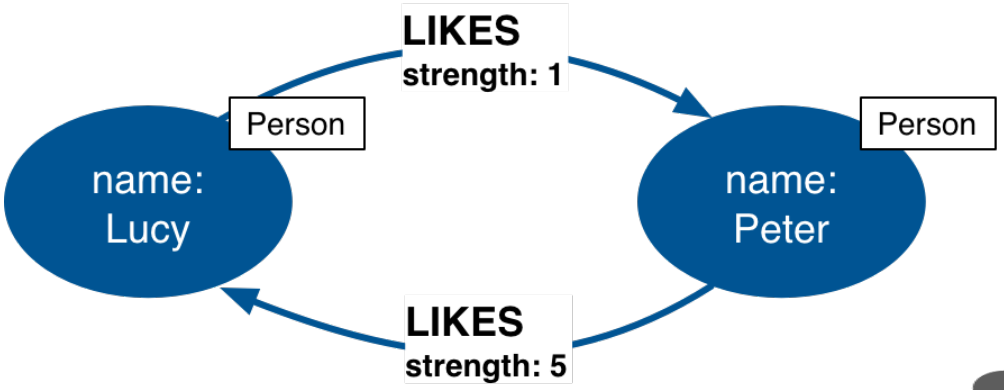



```
MATCH (p1{name:'Eric'})
      -[:KNOWS]-(p2)
RETURN p2
```

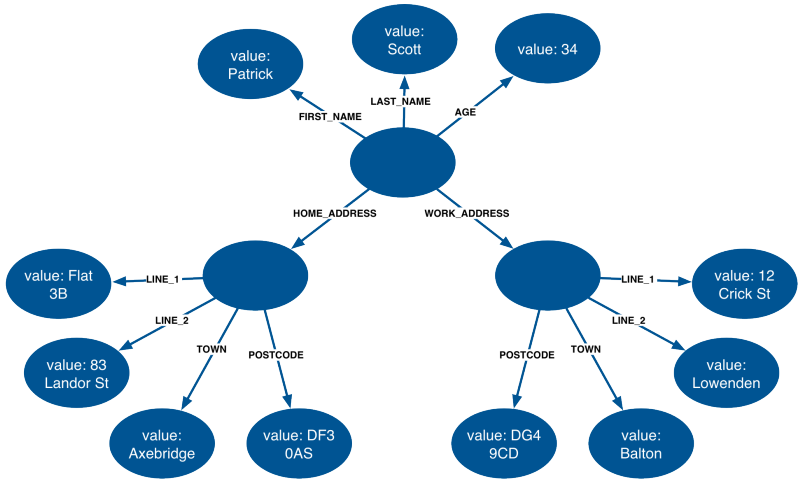
Qualified Bi-Directional Relationships



OR



Properties Versus Relationships

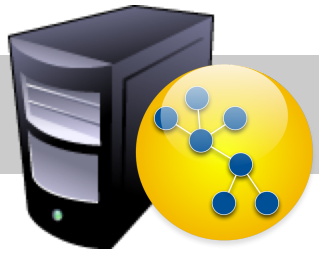


first-name: Patrick
last-name: Scott
age: 34

home-address: Flat 3B,
83 Landor St,
Axebridge,
DF3 0AS

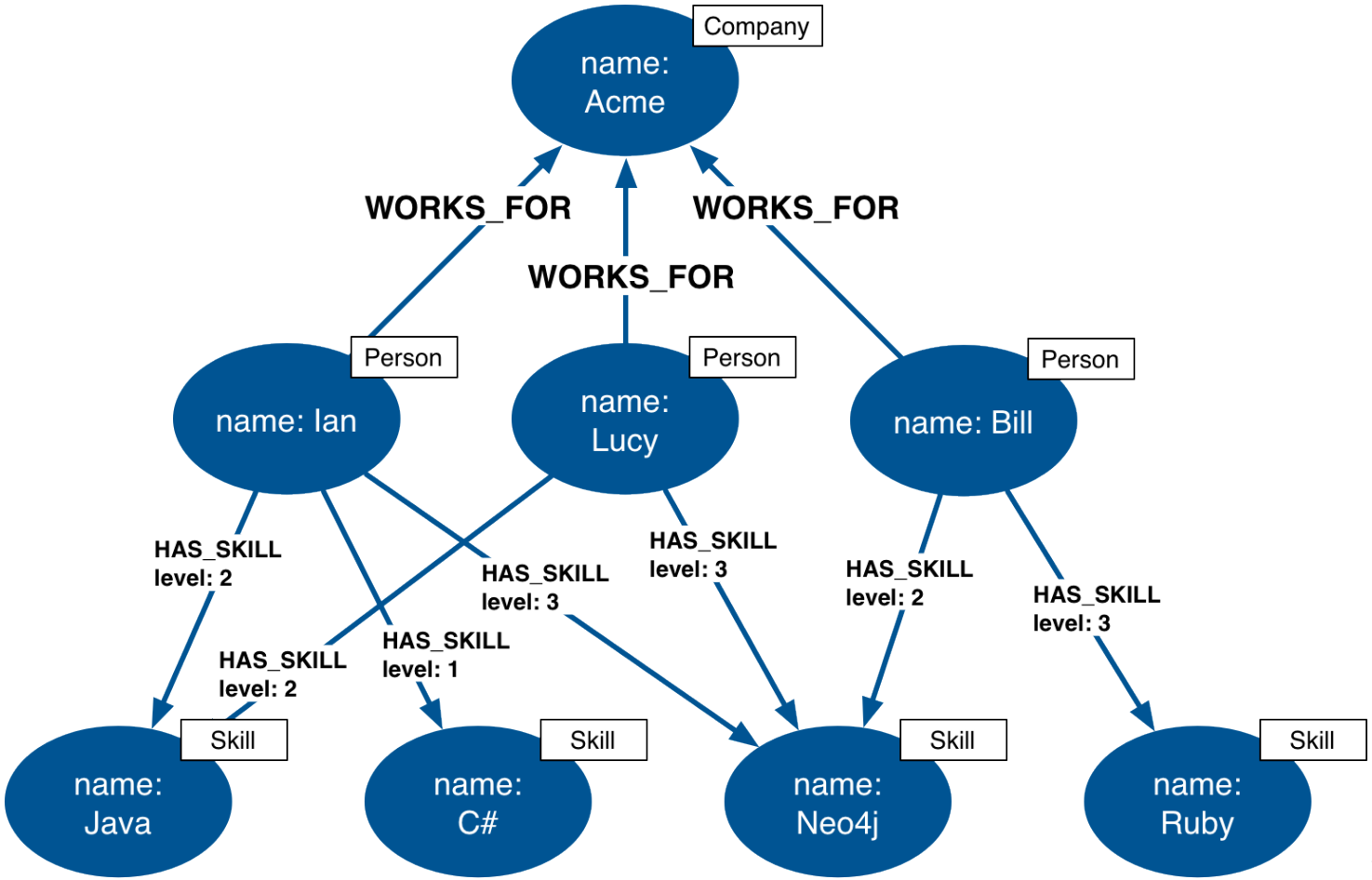
work-address: Acme Ltd,
12 Crick St,
Balton,
DG4 9CD

Use Relationships When…

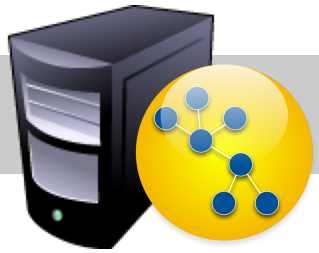


- You need to specify the weight, strength, or some other quality of the *relationship*
- AND/OR the attribute value comprises a complex value type (e.g. address)
- Examples:
  - Find all my colleagues who are *level 2 or above* (relationship quality) in a *skill* (attribute value) we have in common
  - Find all recent orders delivered to the same *delivery address* (complex value type)

Example: Find Expert Colleagues

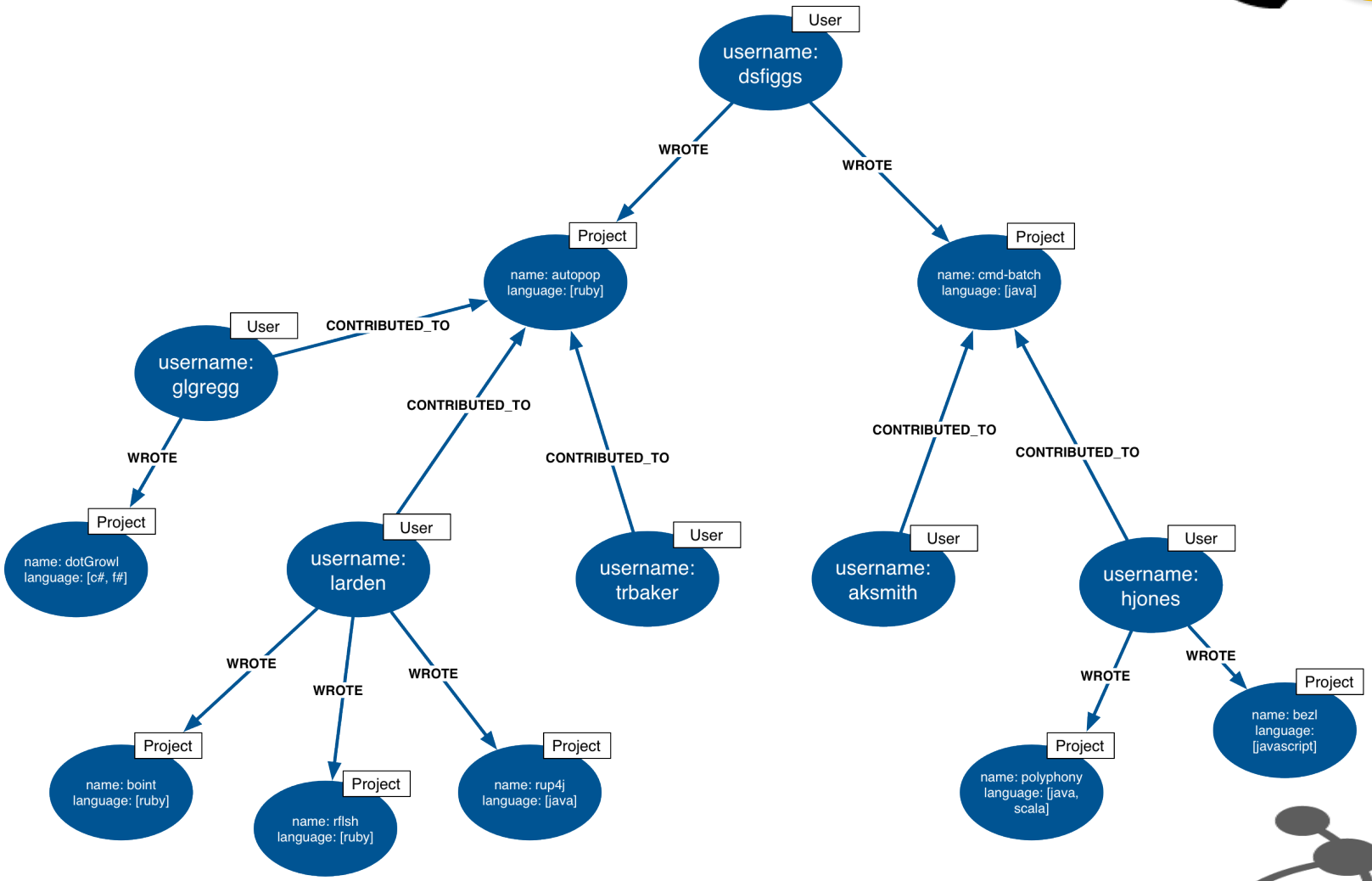


Use Properties When…

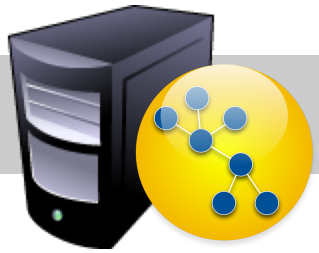


- There's no need to qualify the relationship
- AND the attribute value comprises a simple value type (e.g. colour)
- Examples:
  - Find those projects written by contributors to my projects that use the same *language* (attribute value) as my projects

Example: Similar By Language

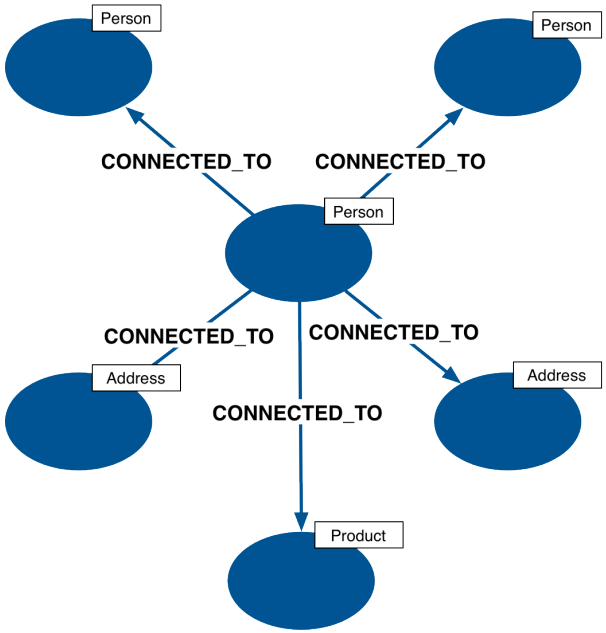


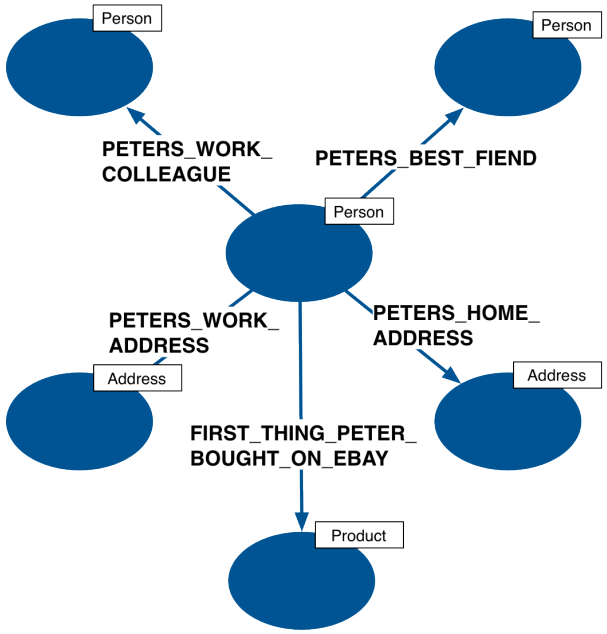
If Performance is Critical…



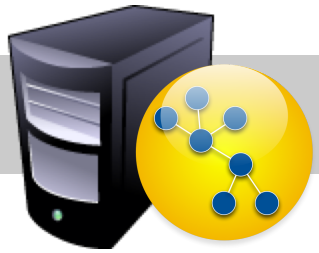
- Small property lookup on a node will be quicker than traversing a relationship
  - But traversing a relationship is still faster than a SQL join…
- However, *many small properties* on a node, or a lookup on a *large string* or *large array* property will impact performance
  - Always performance test against a representative dataset

Relationship Granularity



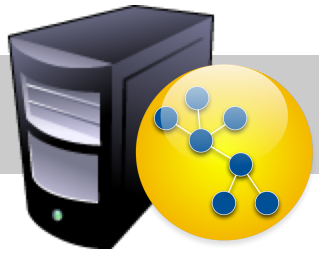


Align With Use Cases

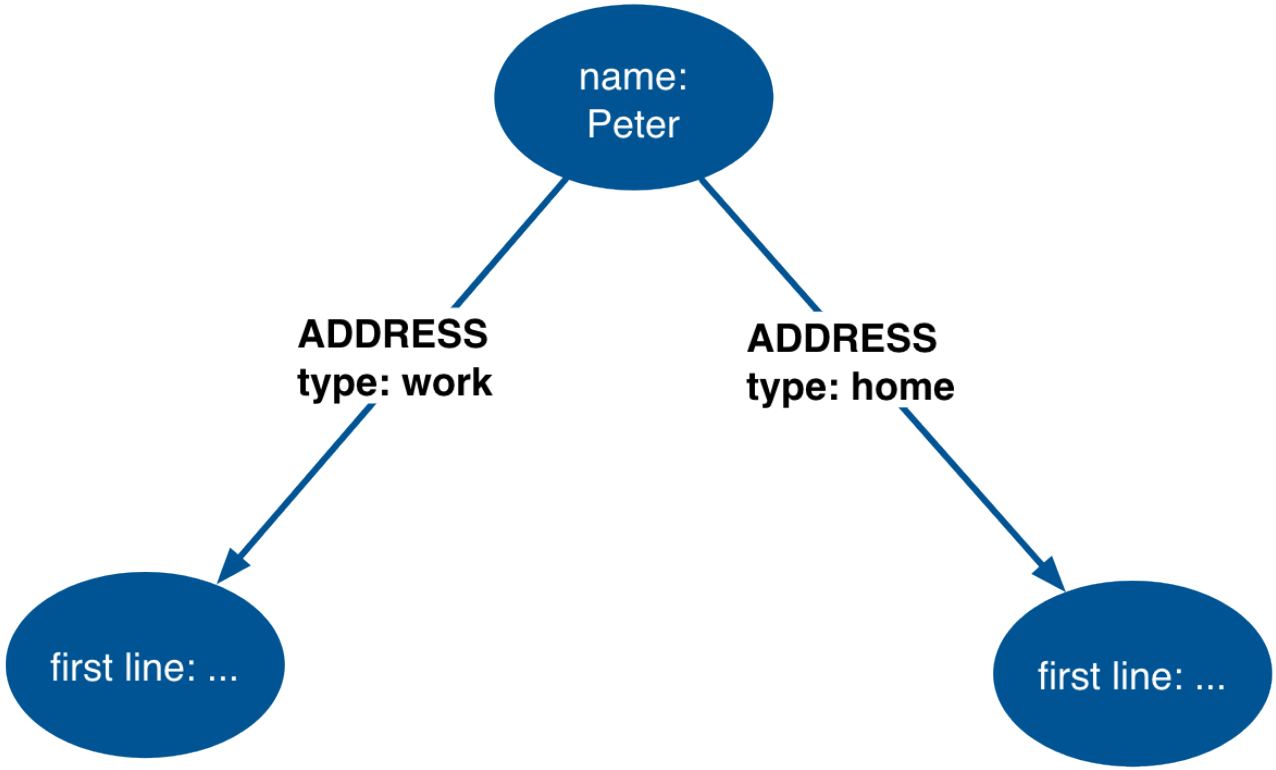


- Relationships are the “royal road” into the graph
- When querying, well-named relationships help discover only what is *absolutely necessary*
  - And eliminate unnecessary portions of the graph from consideration

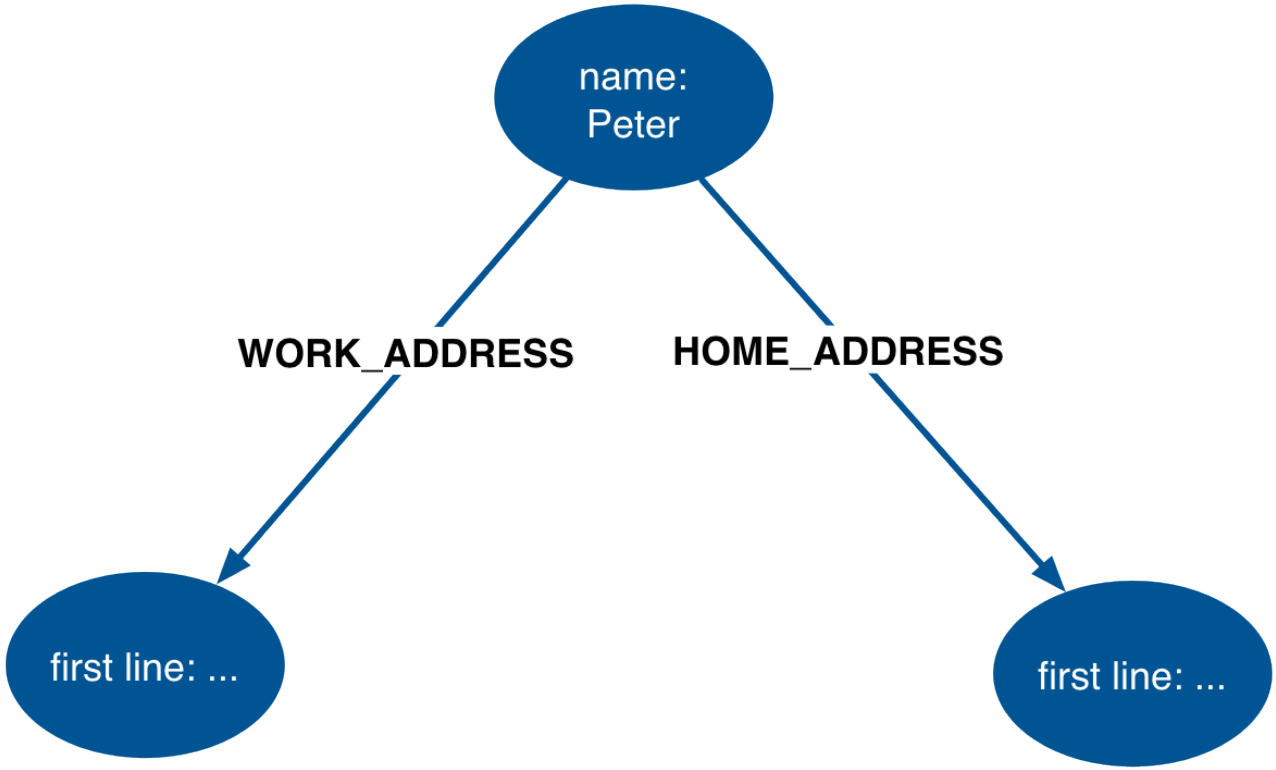
General Relationships



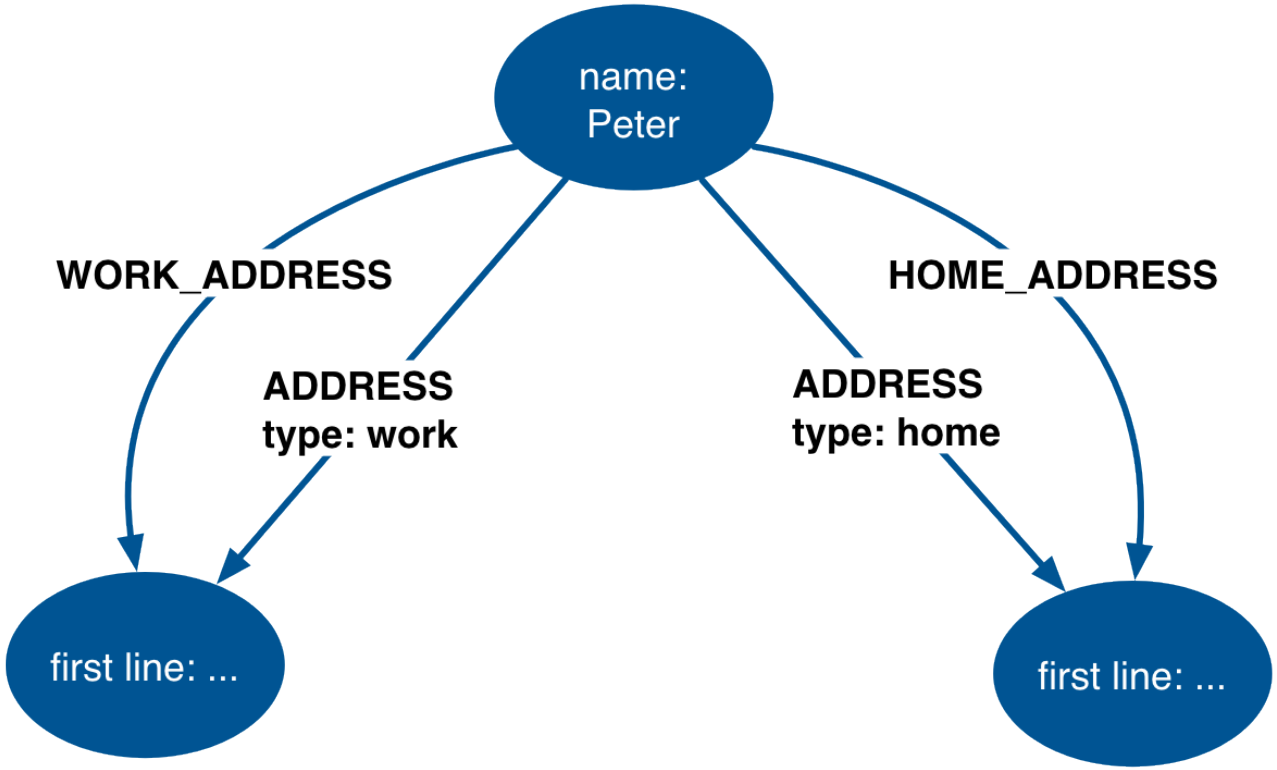
- Qualified by property



Specific Relationships



Best of Both Worlds



Exercise 2

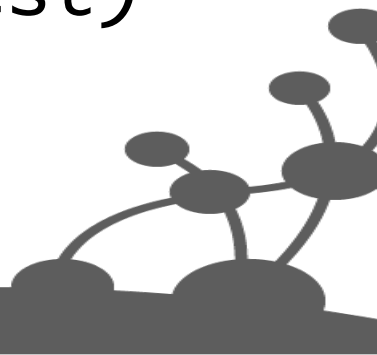
Creating Data

Exercise 2 - Create Some data

- Clean the database
- Execute *create-1.txt*
- View the results
  - `MATCH (n) RETURN n`

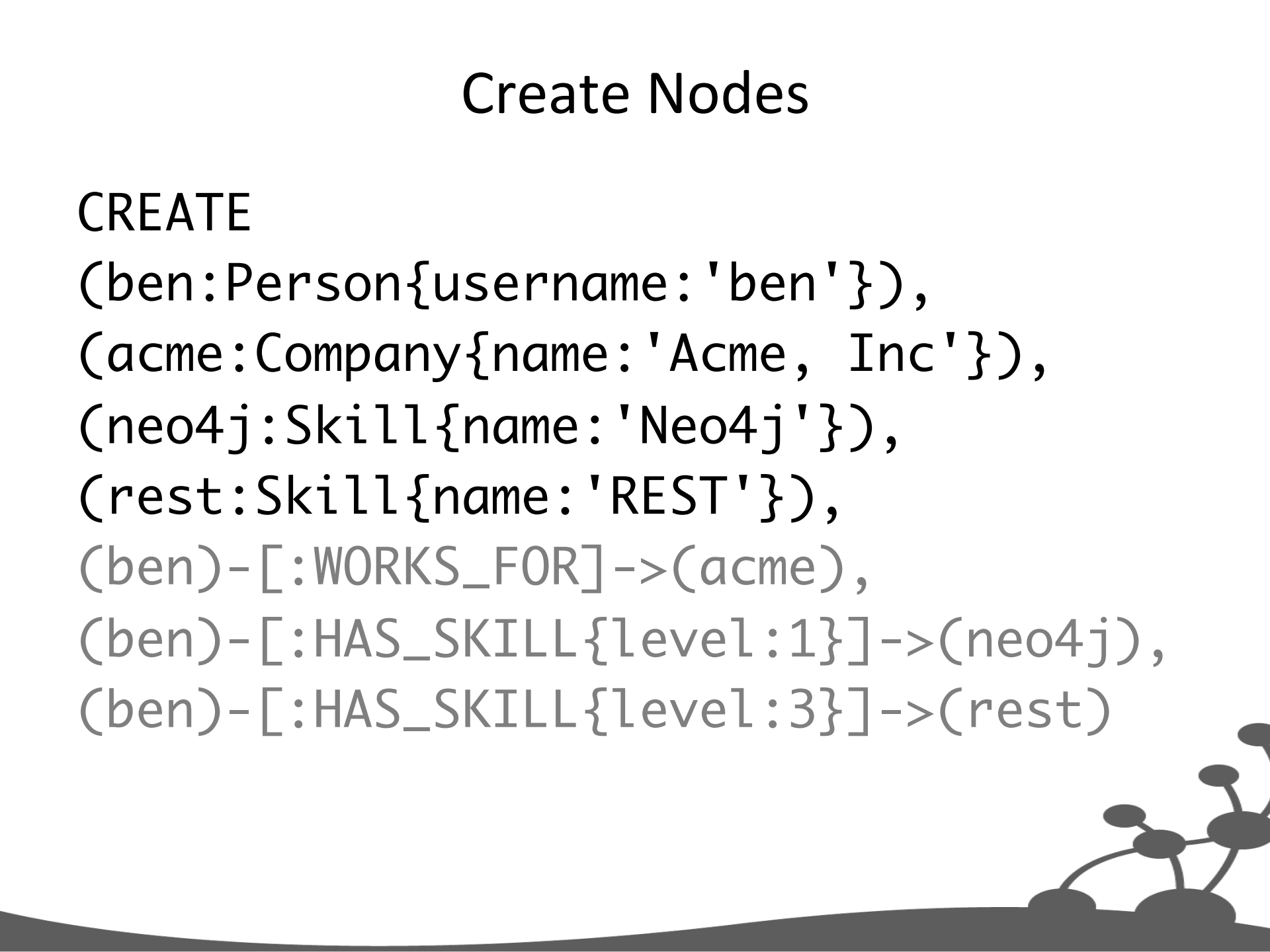
create-1.txt

```
CREATE
(ben:Person{username:'ben'}),
(acme:Company{name:'Acme, Inc'}),
(neo4j:Skill{name:'Neo4j'}),
(rest:Skill{name:'REST'}),
(ben)-[:WORKS_FOR]->(acme),
(ben)-[:HAS_SKILL{level:1}]->(neo4j),
(ben)-[:HAS_SKILL{level:3}]->(rest)
```



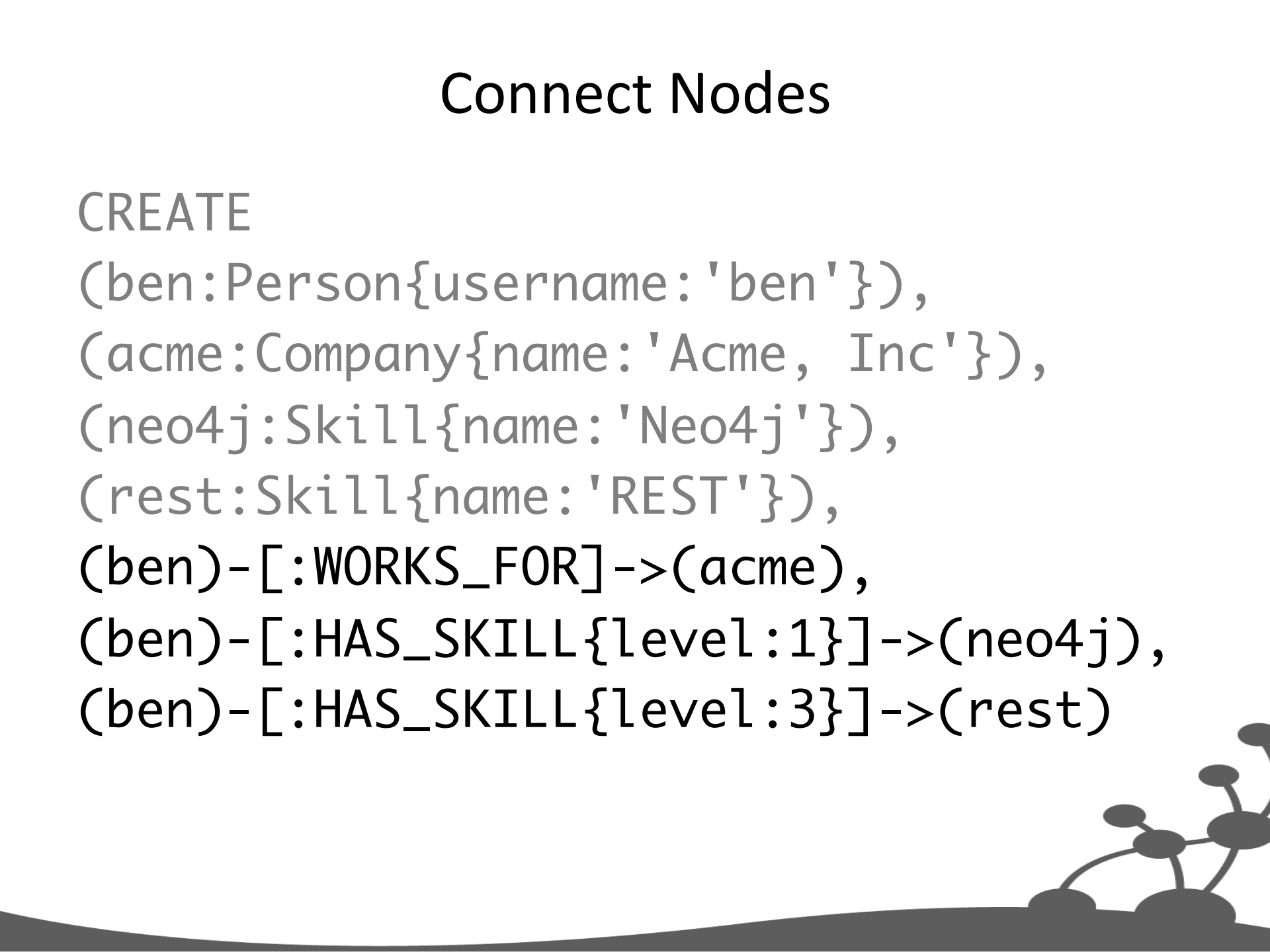
Create Nodes

```
CREATE
(ben:Person{username:'ben'}),
(acme:Company{name:'Acme, Inc'}),
(neo4j:Skill{name:'Neo4j'}),
(rest:Skill{name:'REST'}),
(ben)-[:WORKS_FOR]->(acme),
(ben)-[:HAS_SKILL{level:1}]->(neo4j),
(ben)-[:HAS_SKILL{level:3}]->(rest)
```



Connect Nodes

```
CREATE
(ben:Person{username:'ben'}),
(acme:Company{name:'Acme, Inc'}),
(neo4j:Skill{name:'Neo4j'}),
(rest:Skill{name:'REST'}),
(ben)-[:WORKS_FOR]->(acme),
(ben)-[:HAS_SKILL{level:1}]->(neo4j),
(ben)-[:HAS_SKILL{level:3}]->(rest)
```



Create Some More Data

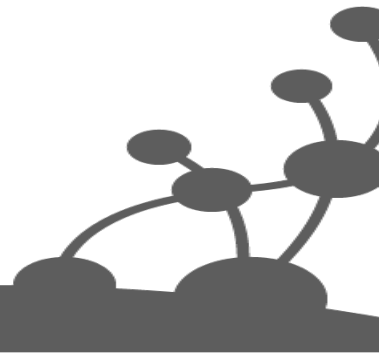
- Create more people
  - Same skills (Neo4j and REST)
  - Same company (Acme)
- View the results
  - `MATCH (n) RETURN n`

Your Turn

- Clean the database
- Execute *create-2.txt*, *create-3.txt* and *create-4.txt*
  - After each operation, view the results
- What happens if you add or remove properties when specifying unique nodes and relationships?

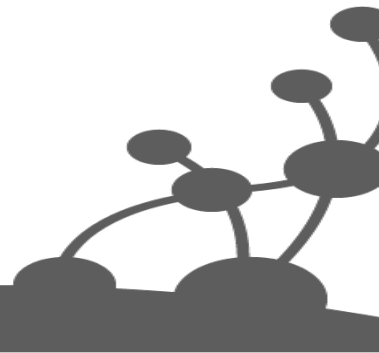
Creating Unique Nodes and Relationships

```
MERGE (c:Company{name:'Acme'})
MERGE (p:Person{username:'ian'})
MERGE (s1:Skill{name:'Java'})
MERGE (s2:Skill{name:'C#'})
MERGE (s3:Skill{name:'Neo4j'})
MERGE (c)<-[:WORKS_FOR]-(p)
MERGE (p)-[r1:HAS_SKILL]->(s1)
MERGE (p)-[r2:HAS_SKILL]->(s2)
MERGE (p)-[r3:HAS_SKILL]->(s3)
SET r1.level = 2
SET r2.level = 2
SET r3.level = 3
RETURN c, p, s1, s2, s3
```



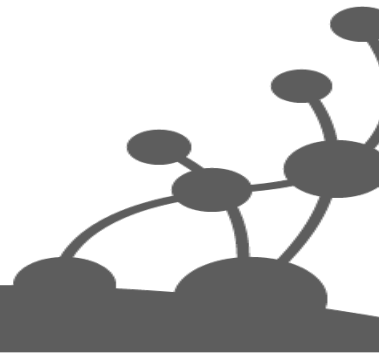
Create Unique Nodes

```
MERGE (c:Company{name:'Acme'})
MERGE (p:Person{username:'ian'})
MERGE (s1:Skill{name:'Java'})
MERGE (s2:Skill{name:'C#'})
MERGE (s3:Skill{name:'Neo4j'})
MERGE (c)<-[:WORKS_FOR]-(p)
MERGE (p)-[r1:HAS_SKILL]->(s1)
MERGE (p)-[r2:HAS_SKILL]->(s2)
MERGE (p)-[r3:HAS_SKILL]->(s3)
SET r1.level = 2
SET r2.level = 2
SET r3.level = 3
RETURN c, p, s1, s2, s3
```



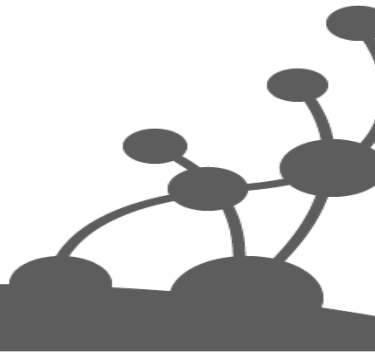
Create Unique Relationships

```
MERGE (c:Company{name:'Acme'})
MERGE (p:Person{username:'ian'})
MERGE (s1:Skill{name:'Java'})
MERGE (s2:Skill{name:'C#'})
MERGE (s3:Skill{name:'Neo4j'})
MERGE (c)<-[:WORKS_FOR]-(p)
MERGE (p)-[r1:HAS_SKILL]->(s1)
MERGE (p)-[r2:HAS_SKILL]->(s2)
MERGE (p)-[r3:HAS_SKILL]->(s3)
SET r1.level = 2
SET r2.level = 2
SET r3.level = 3
RETURN c, p, s1, s2, s3
```



Set Relationship Properties

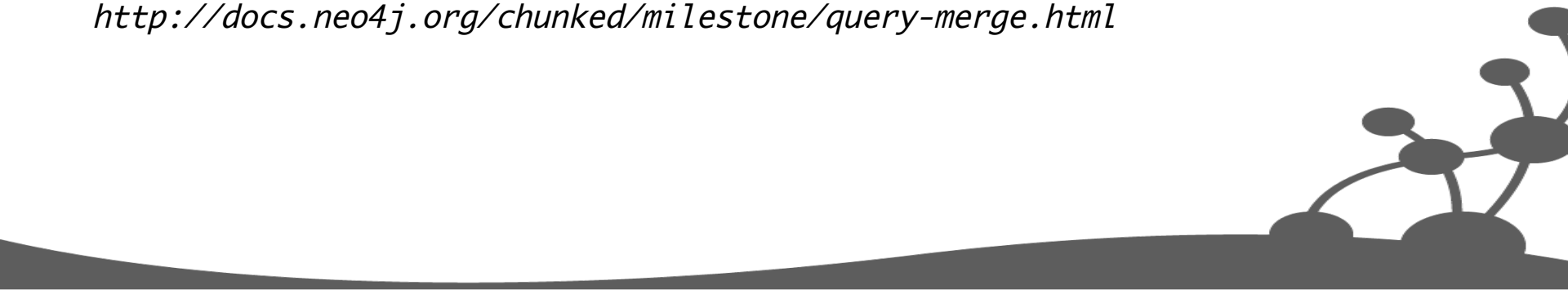
```
MERGE (c:Company{name:'Acme'})
MERGE (p:Person{username:'ian'})
MERGE (s1:Skill{name:'Java'})
MERGE (s2:Skill{name:'C#'})
MERGE (s3:Skill{name:'Neo4j'})
MERGE (c)<-[:WORKS_FOR]-(p)
MERGE (p)-[r1:HAS_SKILL]->(s1)
MERGE (p)-[r2:HAS_SKILL]->(s2)
MERGE (p)-[r3:HAS_SKILL]->(s3)
SET r1.level = 2
SET r2.level = 2
SET r3.level = 3
RETURN c, p, s1, s2, s3
```



MERGE

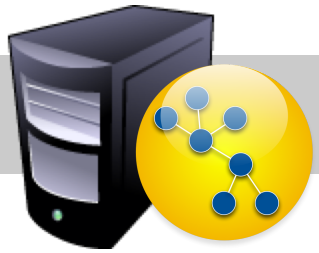
MERGE ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

http://docs.neo4j.org/chunked/milestone/query-merge.html

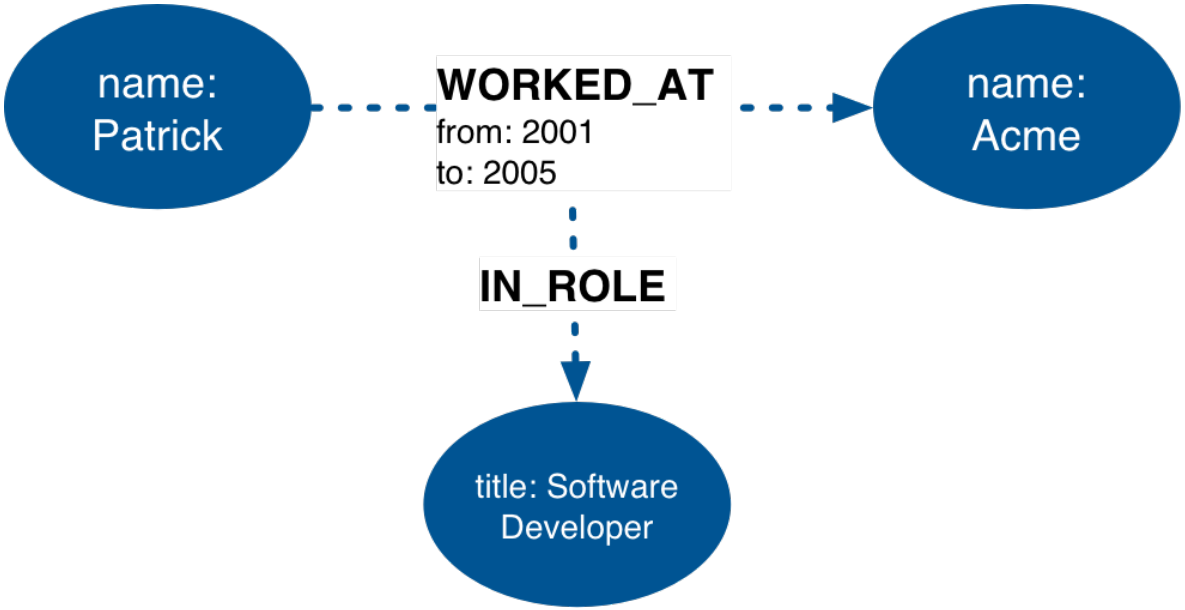


Common Graph Structures

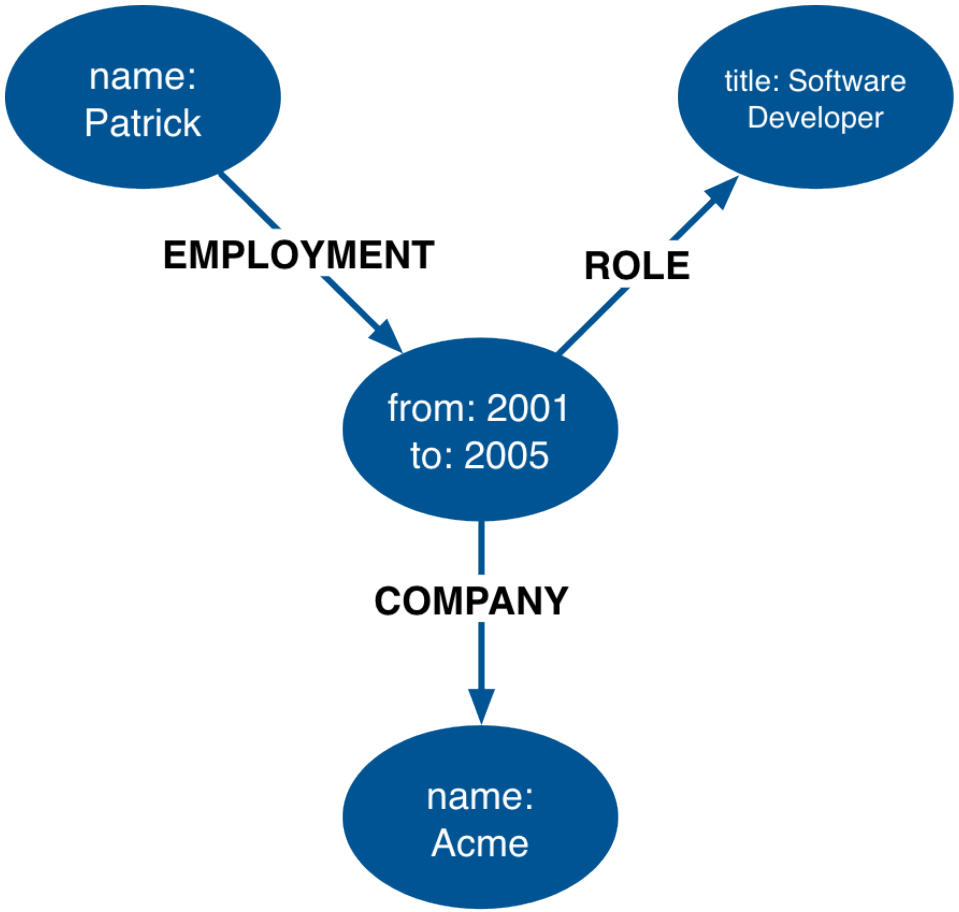
Intermediate Nodes



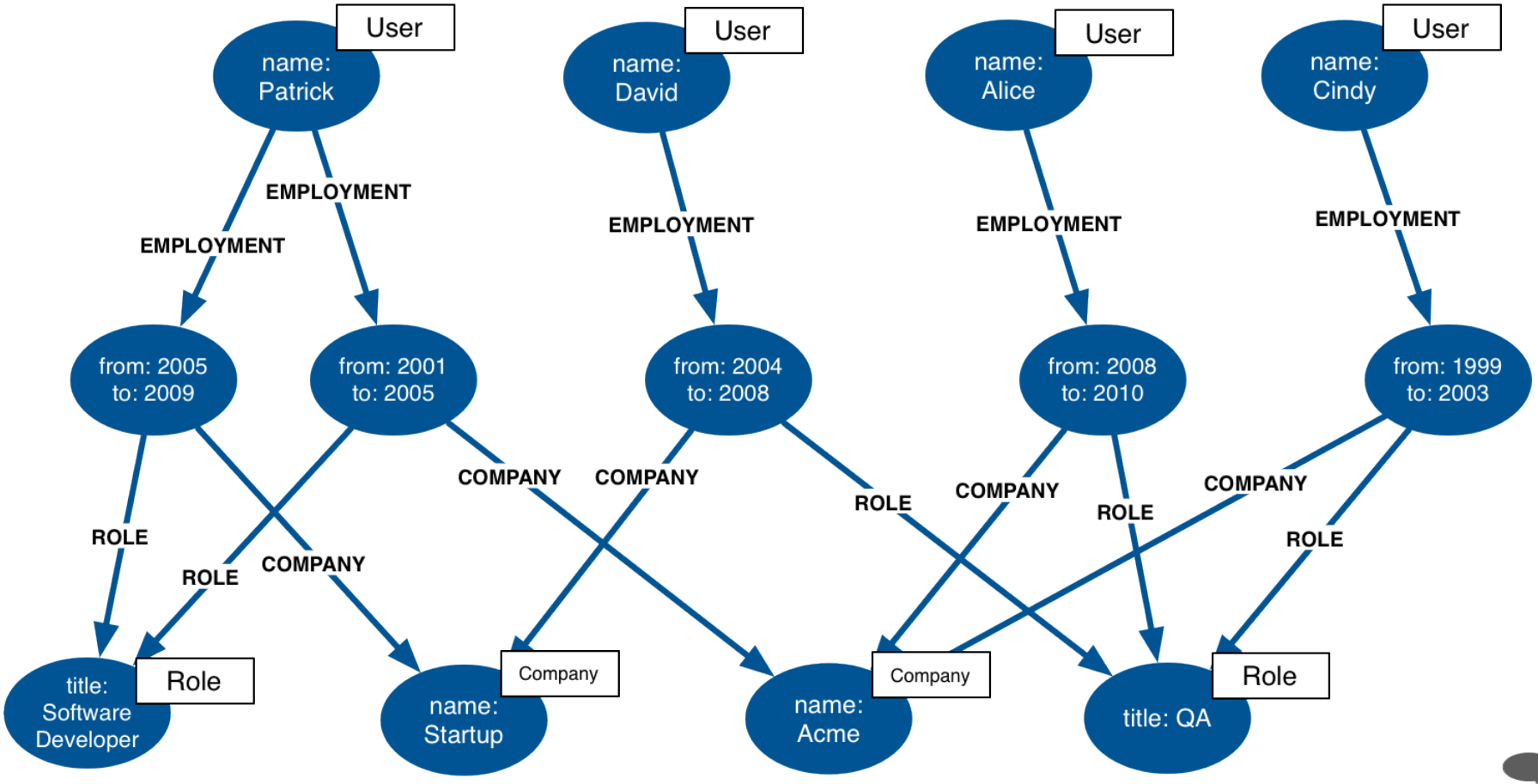
- Connect more than 2 nodes in a single context
  - Hyperedges (n-ary relationships)
- Relate something to a relationship



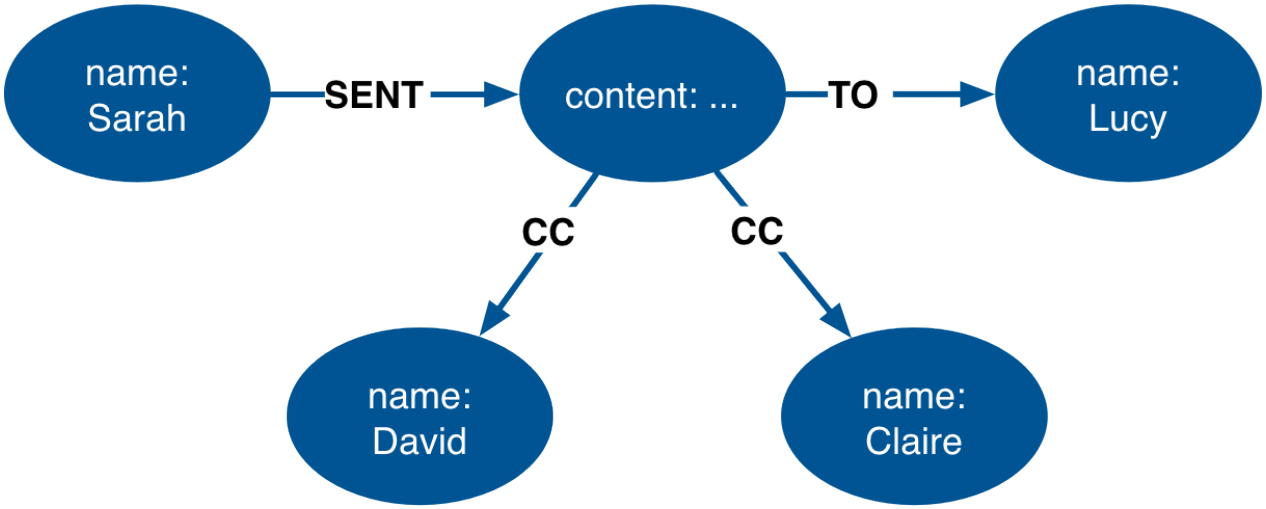
Rich Context, Multiple Dimensions



Dimensions Shared Between Contexts



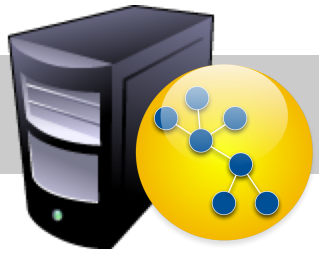
Multiple Parties



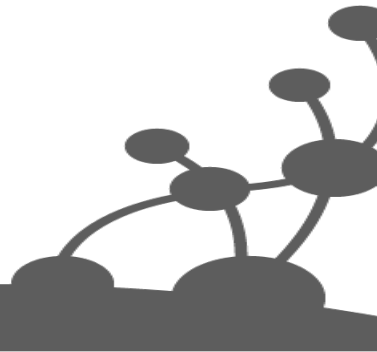
Considerations

- An intermediate node provides flexibility
  - It allows more than two nodes to be connected in a single context
- But it can be overkill, and will have an impact on performance

Linked Lists



- Entities are linked in a sequence
- You need to traverse the sequence
- You may need to identify the beginning or end (first/last, earliest/latest, etc.)
- Examples
  - Event stream
  - Episodes of a TV series
  - Job history

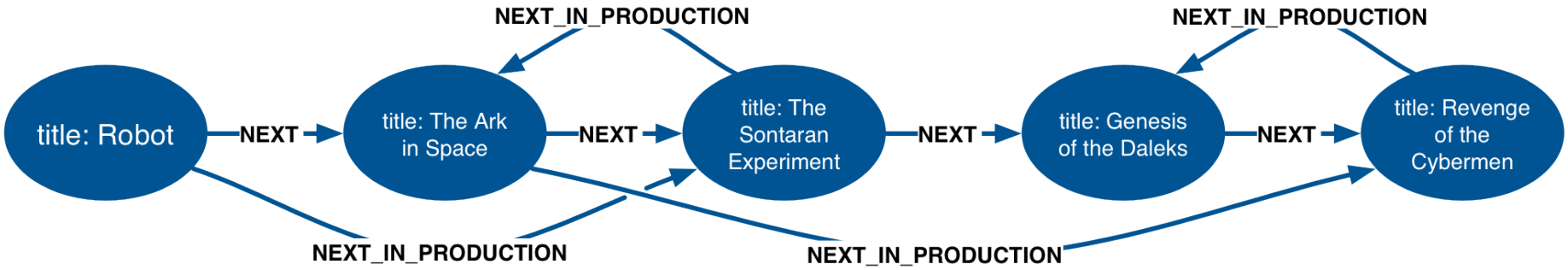


Linked List

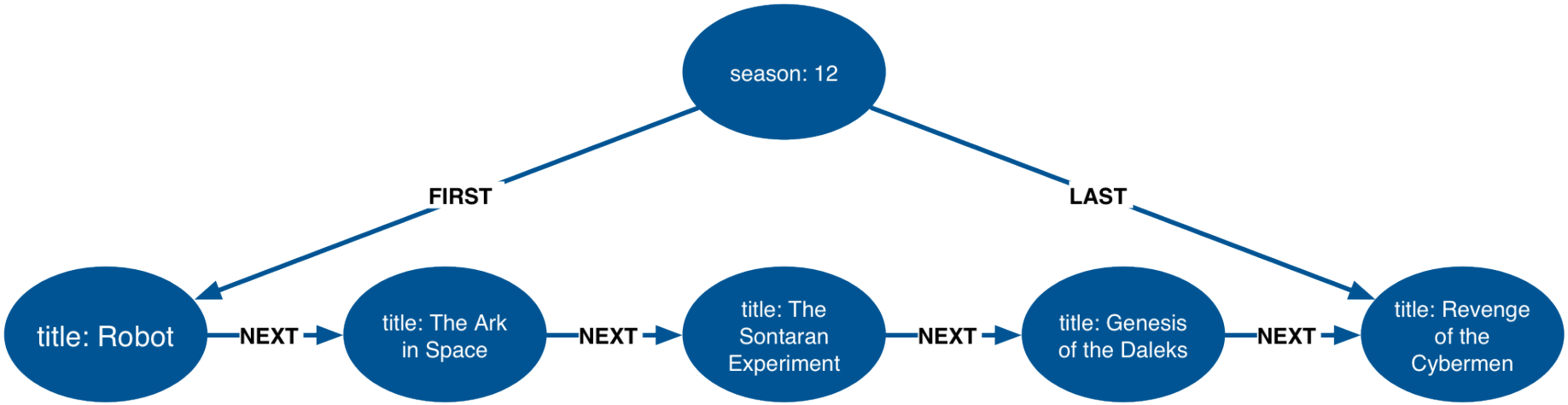

title: Robot
NEXT
title: The Ark in Space
NEXT
title: The Sontaran Experiment
NEXT
title: Genesis of the Daleks
NEXT
title: Revenge of the Cybermen

Doubly Linked List

Interleaved Linked Lists



Pointers to Head and Tail

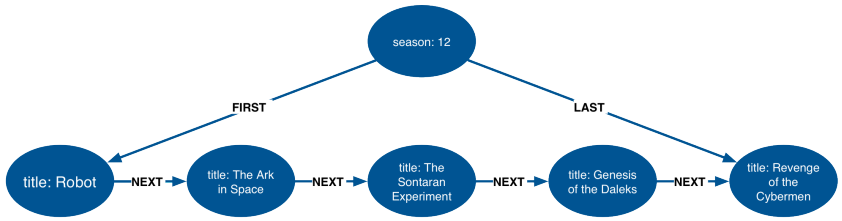


Exercise 3

Linked List Example

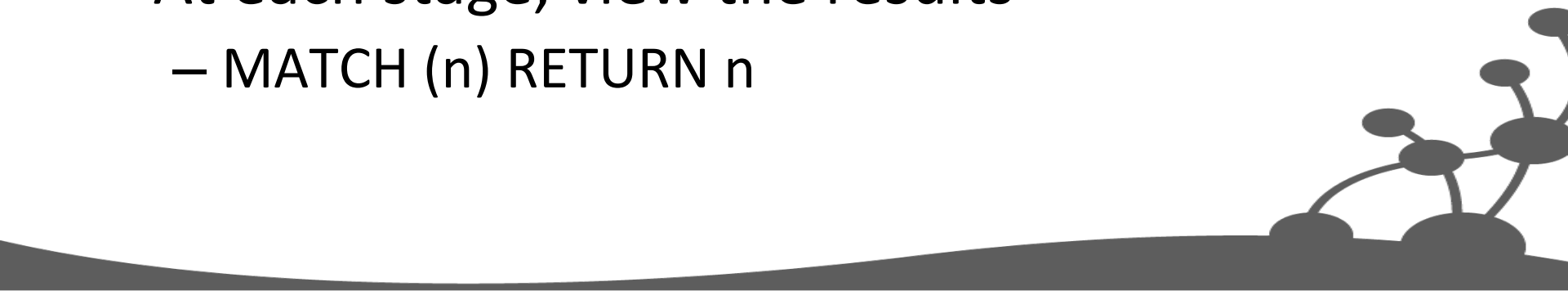
Season 12 of Doctor Who

- Add stories as they are broadcast
  - Maintain pointer to FIRST and LAST stories broadcast
- Find all stories broadcast so far
- Find latest story broadcast so far



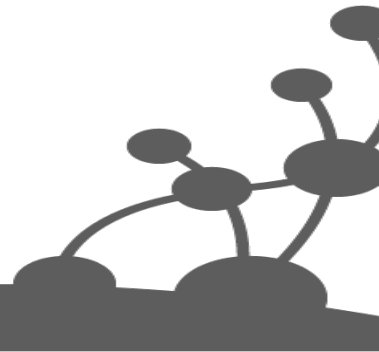
Your Turn

- Clean the database
- Execute *setup.tx*t
  - Creates root season node
- Execute *add-node.txt*
  - Adds *Robot*
- Modify the query to add more stories in broadcast order
- At each stage, view the results
  - MATCH (n) RETURN n



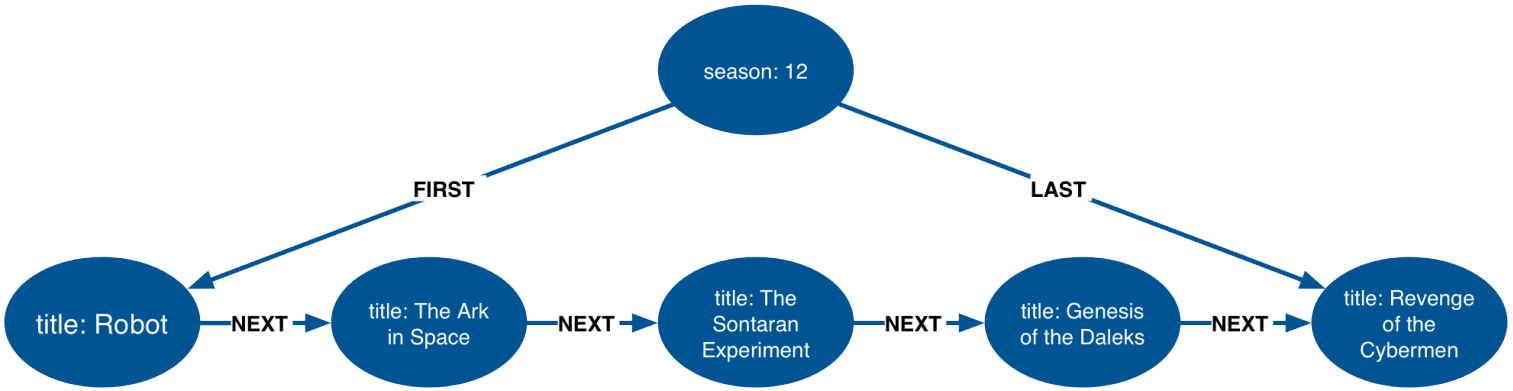
Add Story to Season

```
MERGE (season:Season{season:12})
MERGE (season)-[:LAST]->(newStory:Story{title:'Robot'})
WITH season, newStory

// Determine whether first story already exists
WITH season, newStory,
     CASE WHEN NOT ((season)-[:FIRST]->()) THEN [1] ELSE []
     END
  AS firstExists

// Create FIRST rel newStory is first story
FOREACH (i IN firstExists | MERGE (season)-[:FIRST]->(newStory))
WITH season, newStory

// Delete old LAST relationship
MATCH (newStory)<-[:LAST]-(season)-[oldRel:LAST]->(oldLast)
DELETE oldRel
MERGE (oldLast)-[:NEXT]->(newStory)
```



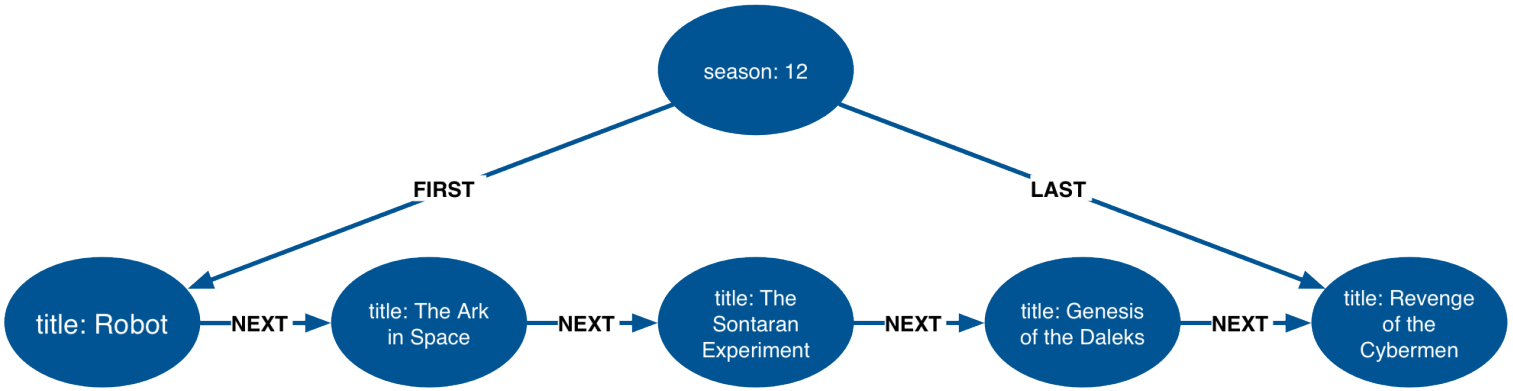
Query-1 - Find All Stories Broadcast So Far

```
MATCH (season:Season)-[:FIRST]->(firstStory)
      -[:NEXT*0..]->(nextStory)
RETURN nextStory.title AS nextStory
```

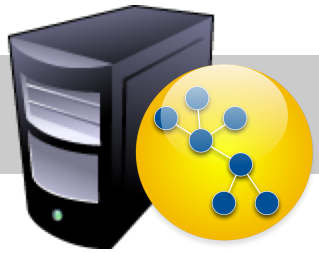


Query-2 - Find Last Story to be Broadcast

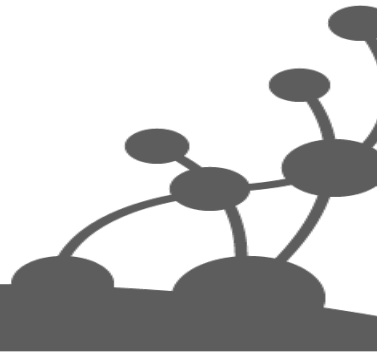
```
MATCH (season:Season)-[:LAST]->(lastStory)
RETURN lastStory.title AS lastStory
```



In-Graph Indexes



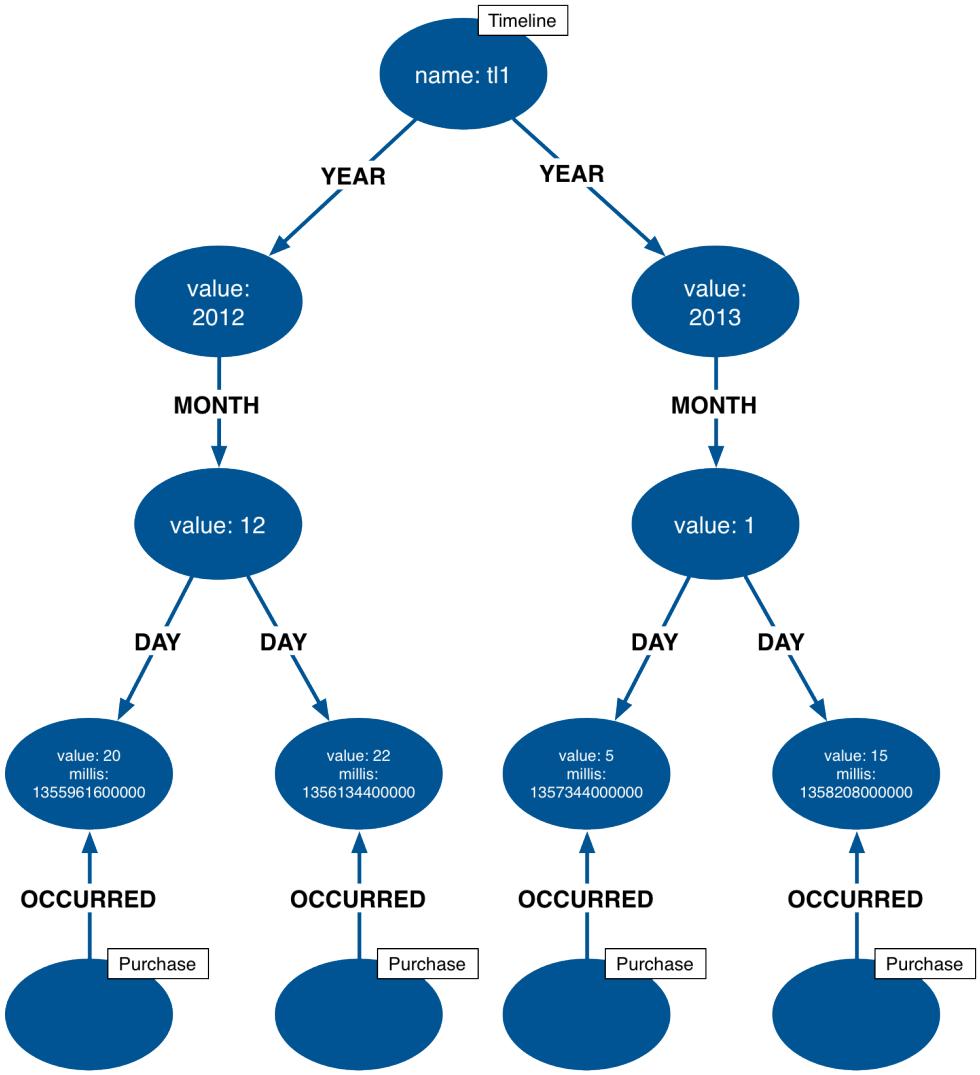
- Indexes are graphs:
  - B-tree (binary search)
  - R-tree (spatial access, multi-dimensional information)



Timeline Tree

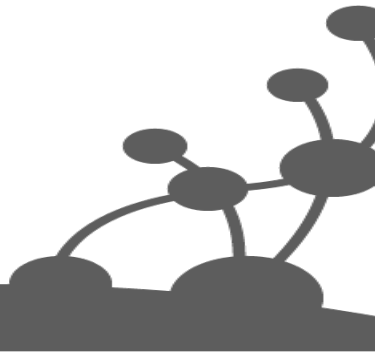
- Discrete events
  - No natural relationships to other events
- You need to find events at differing levels of granularity
  - Between two days
  - Between two months
  - Between two minutes

Example Timeline Tree



Exercise 4

Timeline Tree Example



Your Turn

- Clean the database
- Execute *create-1.txt* to *create-6.txt* in order
- At each stage, view the results
  - MATCH (n) RETURN n

Lazily Insert Date Elements

```
MERGE (timeline:Timeline{name:'timeline-1'})
MERGE (timeline)-[:YEAR]->(year{value:2007})
MERGE (year)-[:MONTH]->(month{value:1})
MERGE (month)-[:DAY]->(day{value:14})
MERGE (day)<-[:OCCURRED]-
      (n:Purchase{name:'purchase-1'})
```

Create or Insert Root

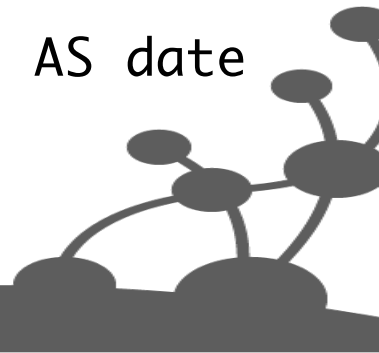
```
MERGE (timeline:Timeline{name:'timeline-1'})
MERGE (timeline)-[:YEAR]->(year{value:2007})
MERGE (year)-[:MONTH]->(month{value:1})
MERGE (month)-[:DAY]->(day{value:14})
MERGE (day)<-[:OCCURRED]-
      (n:Purchase{name:'purchase-1'})
```

The Add ‘Locally Unique’ Nodes

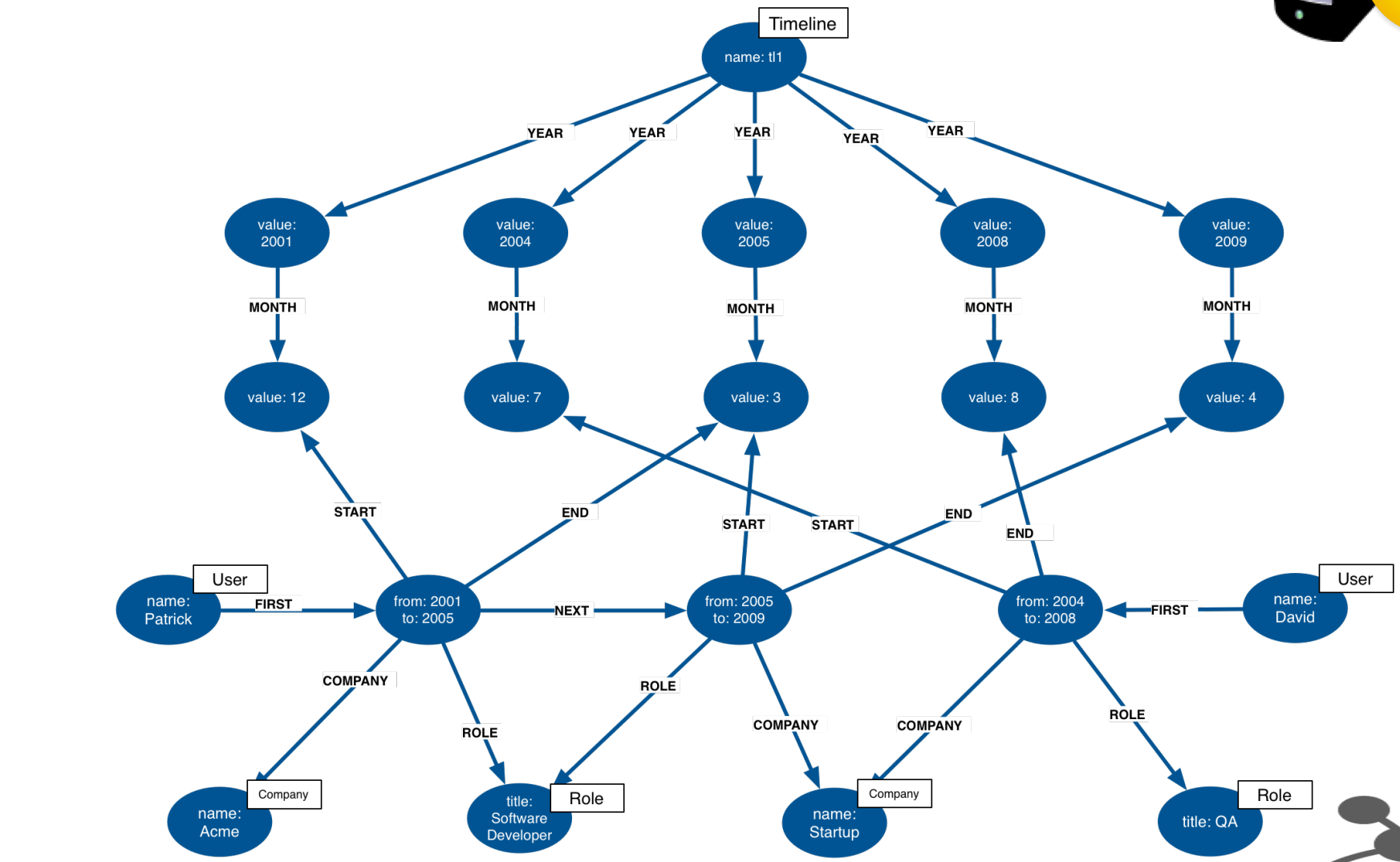
```
MERGE (timeline:Timeline{name:'timeline-1'})
MERGE (timeline)-[:YEAR]->(year{value:2007})
MERGE (year)-[:MONTH]->(month{value:1})
MERGE (month)-[:DAY]->(day{value:14})
MERGE (day)<-[:OCCURRED]-
      (n:Purchase{name:'purchase-1'})
```

Query-1 - Get All Events Between Two Dates

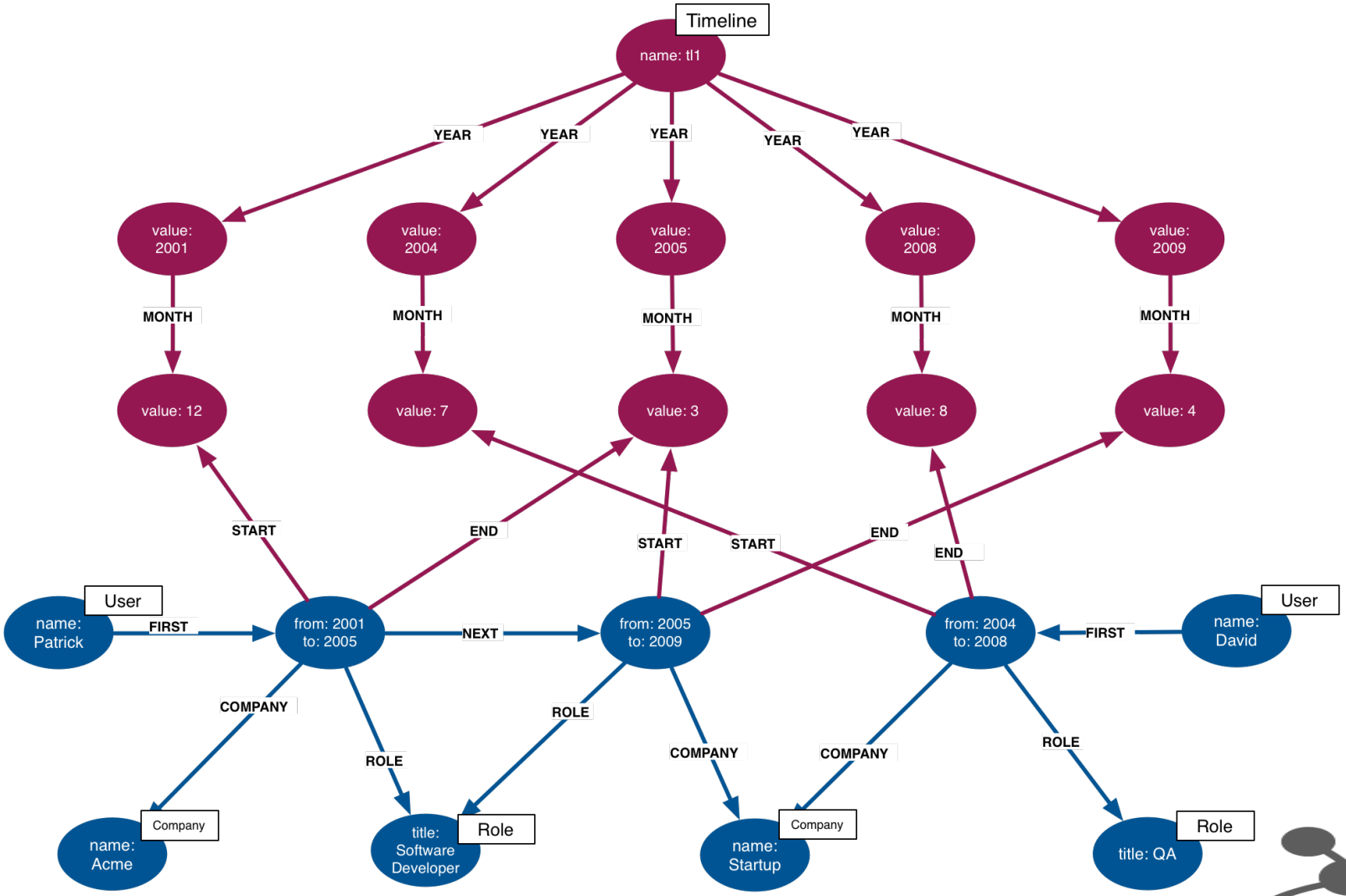
```
MATCH (timeline:Timeline{name:'timeline-1'})
       -[:YEAR]->(y)
      -[:MONTH]->(m)
      -[:DAY]->(d)<-[:OCCURRED]-(n)
WHERE (y.value > {startYear} AND y.value < {endYear})
OR ({startYear} = {endYear})
OR (y.value = {startYear}
    AND ((m.value > {startMonth})
        OR (m.value = {startMonth}
           AND d.value >= {startDay})))
OR (y.value = {endYear}
    AND ((m.value < {endMonth})
        OR (m.value = {endMonth}
           AND d.value <= {endDay})))
RETURN n.name,
      (d.value + "-" + m.value + "-" + y.value) AS date
```



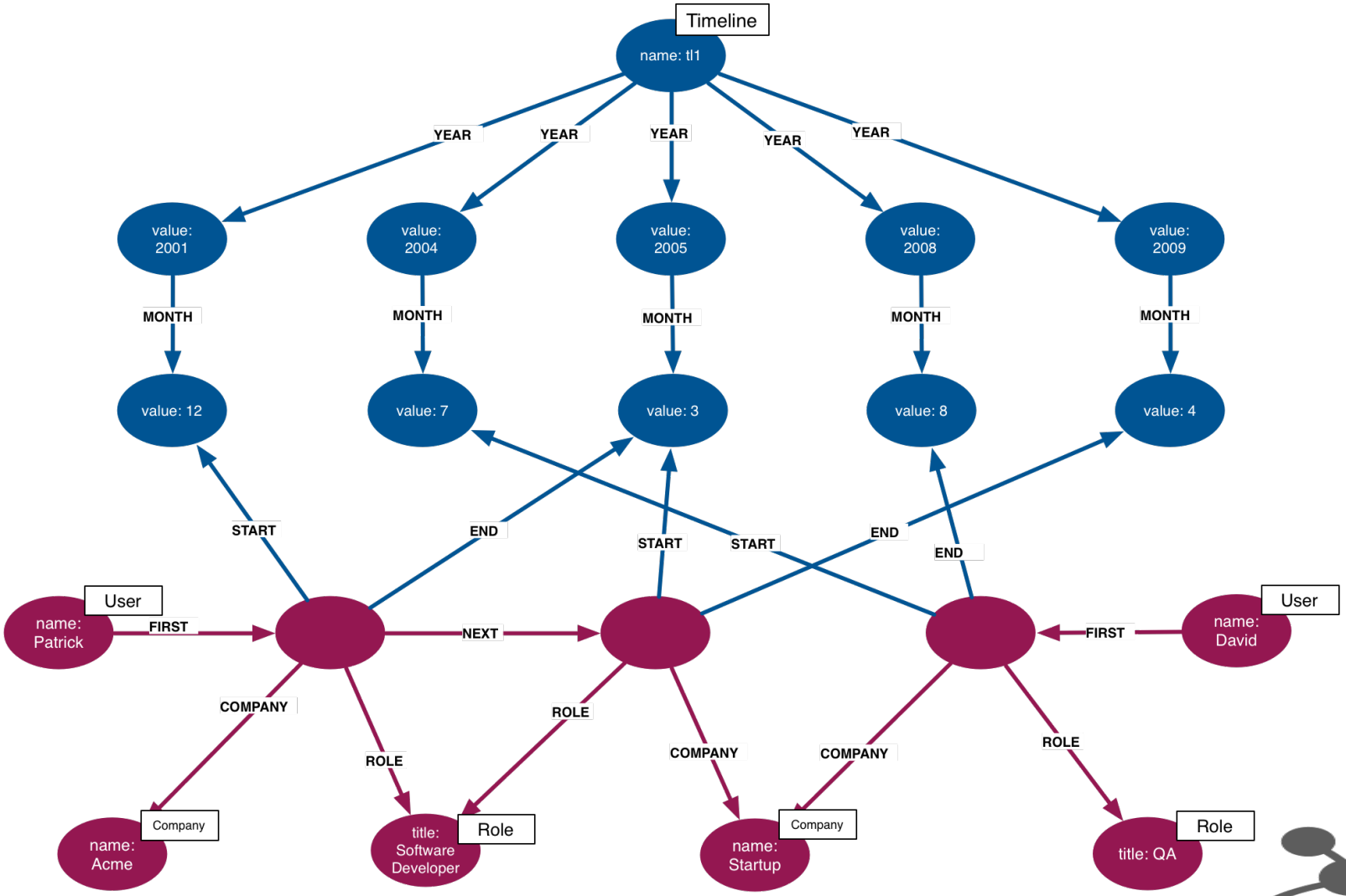
Composing Structures



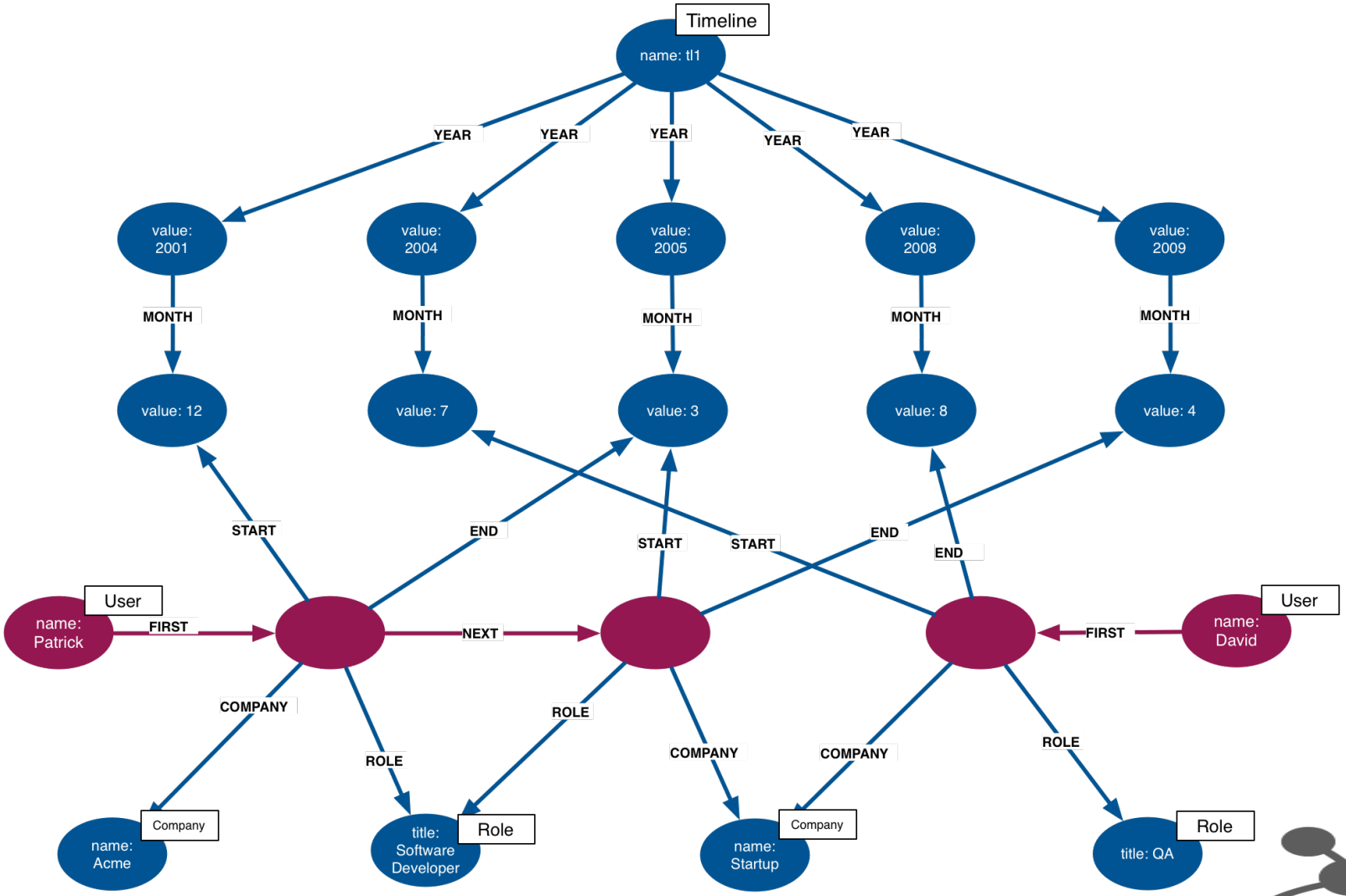
Timeline Tree



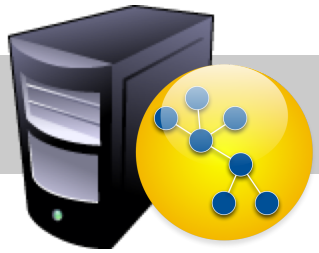
Intermediate Nodes



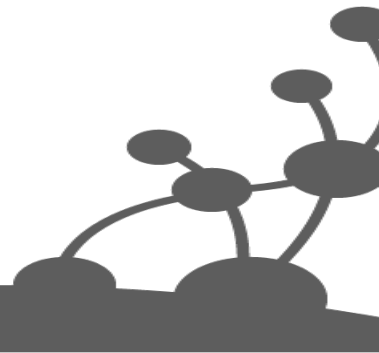
Linked Lists



Versioning Graphs

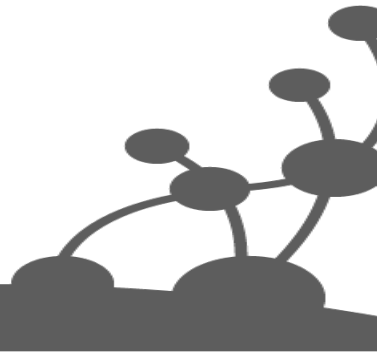


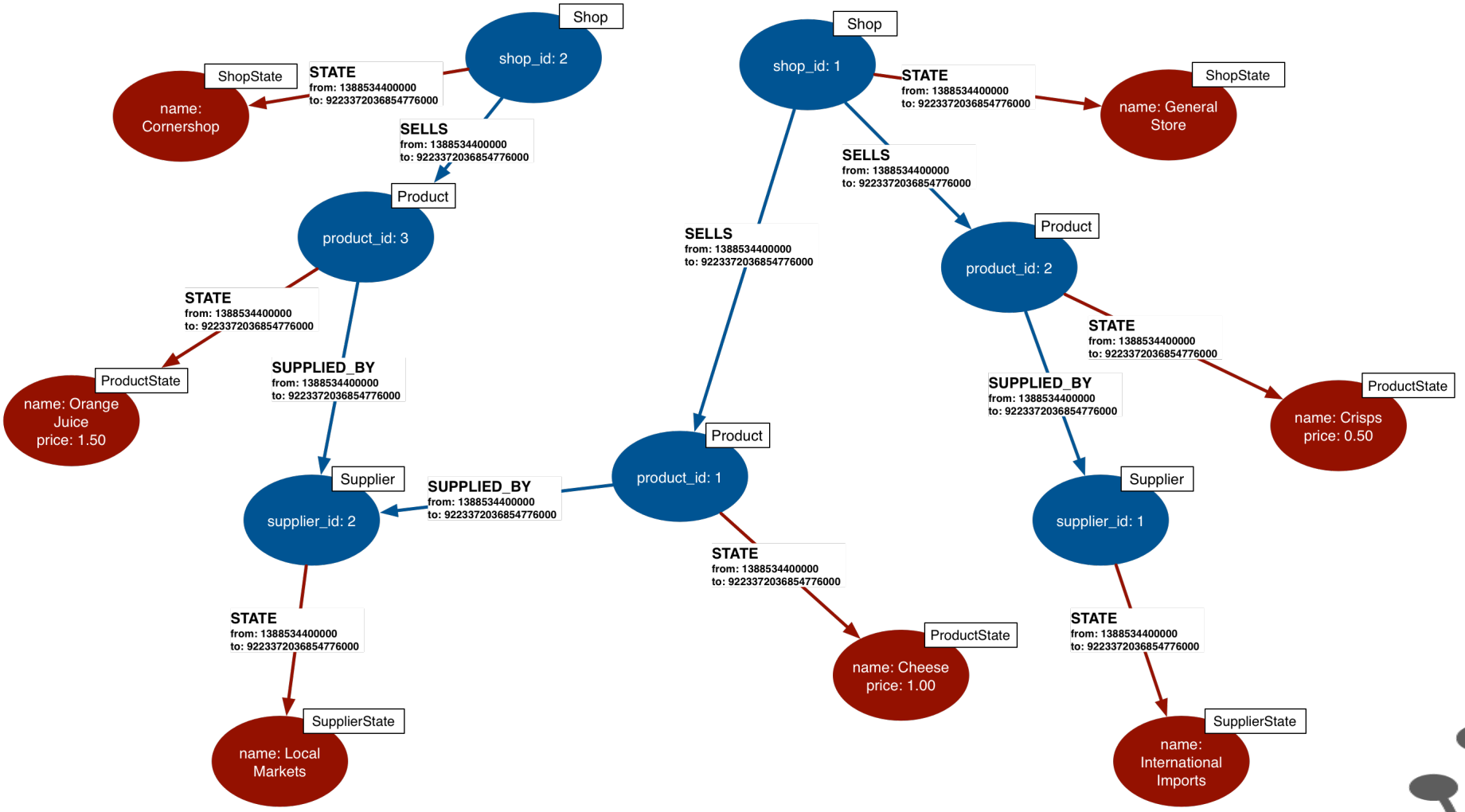
- Time-based
  - Universal versioning schema
  - Discrete, continuous sequence
    - Millis since the epoch



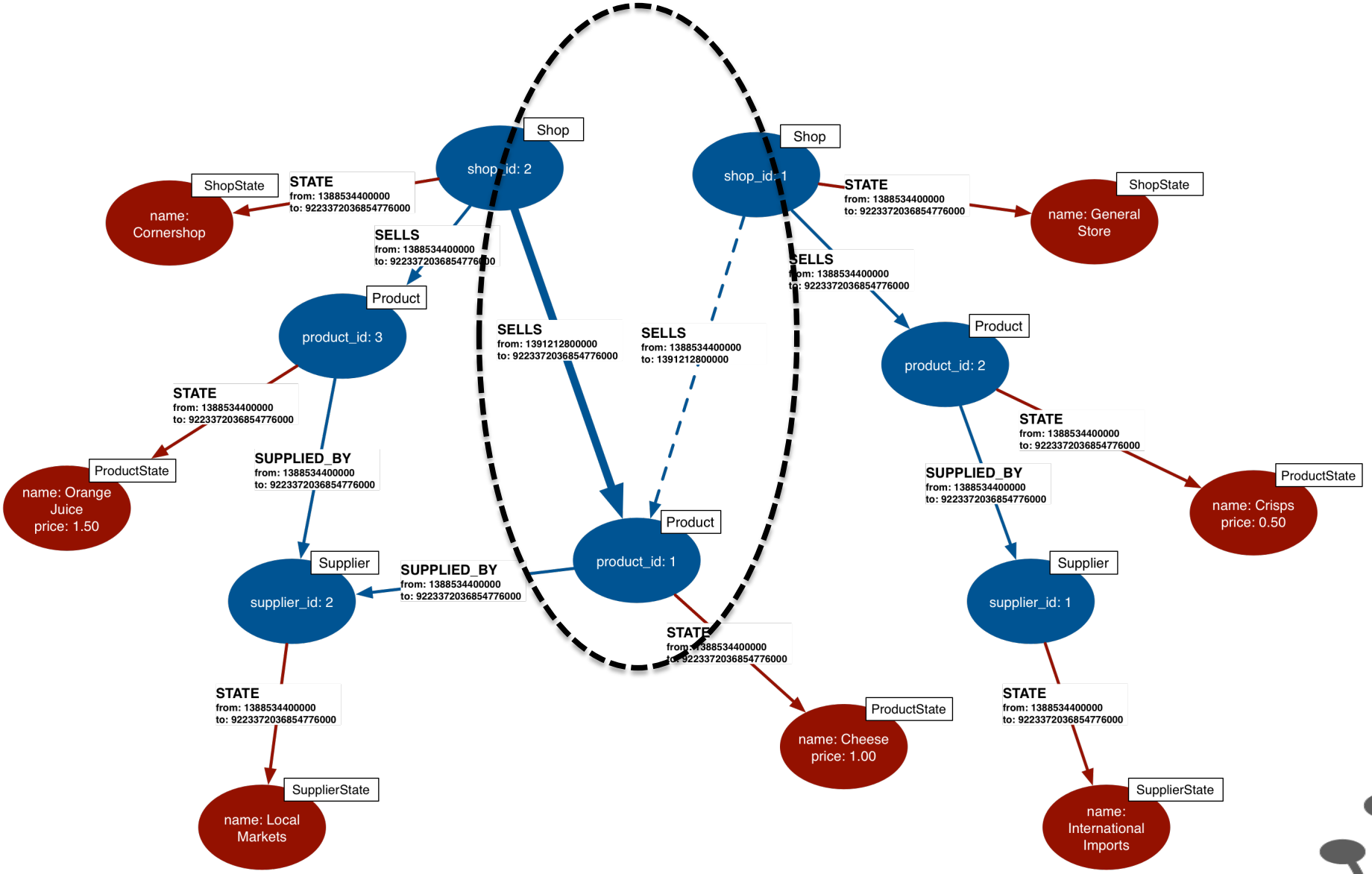
Separate Structure from State

- Structure
  - Identity nodes
    - Placeholders
  - Timestamped identity relationships
    - i.e. normal domain relationships
- State
  - State nodes
    - Snapshot of entity state
  - Timestamped state relationships





1 Jan 2014 = 1388534400000



1 Feb 2014 = 1391212800000

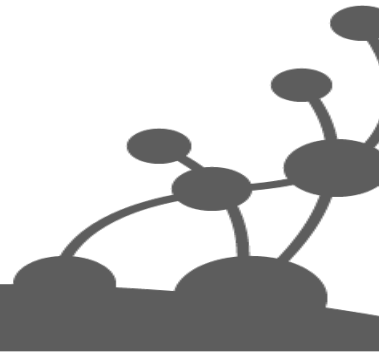
Find Current Structural Relationship

```
MATCH (p:Product{product_id:1})<-[r:SELLS]-(:Shop)
WHERE r.to = 9223372036854775807
MATCH (s:Shop{shop_id:2})
SET r.to = 1391212800000
CREATE (s)
       -[:SELLS{from:1391212800000,to:9223372036854775807}]->(p)
```

9223372036854775807 = End of Time = EOT

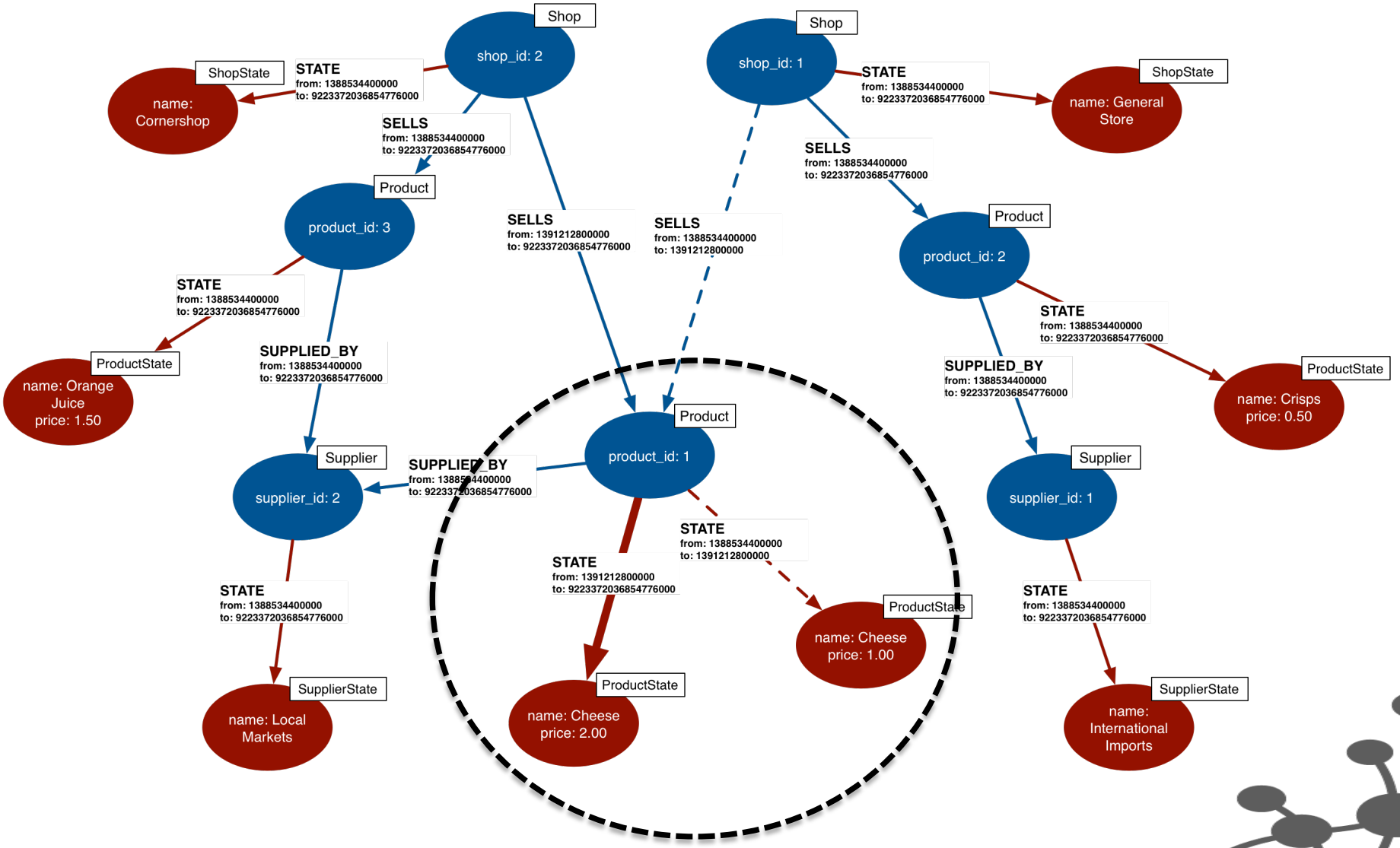
Archive Structural Relationship

```
MATCH (p:Product{product_id:1})<-[r:SELLS]-(:Shop)
WHERE r.to = 9223372036854775807
MATCH (s:Shop{shop_id:2})
SET r.to = 1391212800000
CREATE (s)
       -[:SELLS{from:1391212800000,to:9223372036854775807}]->(p)
```



Create New Structural Relationship

```
MATCH (p:Product{product_id:1})<-[r:SELLS]-(:Shop)
WHERE r.to = 9223372036854775807
MATCH (s:Shop{shop_id:2})
SET r.to = 1391212800000
CREATE (s)
       -[:SELLS{from:1391212800000,to:9223372036854775807}]->(p)
```



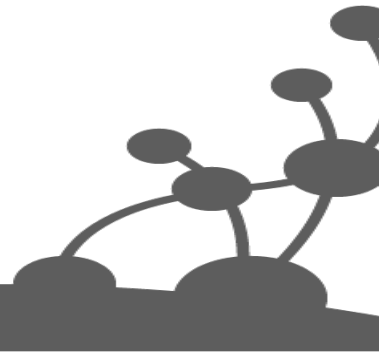
1 Feb 2014 = 1391212800000

All Products Sold by Shop 1 on 5 January 2014

```
MATCH (s:Shop{shop_id:1})-[r1:SELLS]->(p:Product)
WHERE (r1.from <= 1388880000000 AND r1.to > 1388880000000)
MATCH (p)-[r2:STATE]->(ps:ProductState)
WHERE (r2.from <= 1388880000000 AND r2.to > 1388880000000)
RETURN p.product_id AS productId,
       ps.name AS product,
       ps.price AS price
ORDER BY price DESC
```

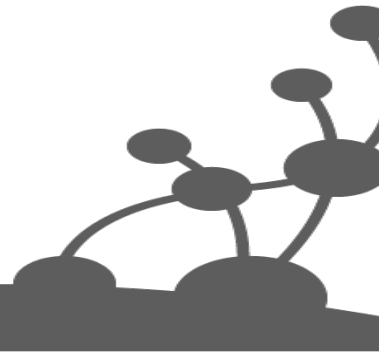
Find Structure

```
MATCH (s:Shop{shop_id:1})-[r1:SELLS]->(p:Product)
WHERE (r1.from <= 1388880000000 AND r1.to > 1388880000000)
MATCH (p)-[r2:STATE]->(ps:ProductState)
WHERE (r2.from <= 1388880000000 AND r2.to > 1388880000000)
RETURN p.product_id AS productId,
       ps.name AS product,
       ps.price AS price
ORDER BY price DESC
```



Find State

```
MATCH (s:Shop{shop_id:1})-[r1:SELLS]->(p:Product)
WHERE (r1.from <= 1388880000000 AND r1.to > 1388880000000)
MATCH (p)-[r2:STATE]->(ps:ProductState)
WHERE (r2.from <= 1388880000000 AND r2.to > 1388880000000)
RETURN p.product_id AS productId,
       ps.name AS product,
       ps.price AS price
ORDER BY price DESC
```

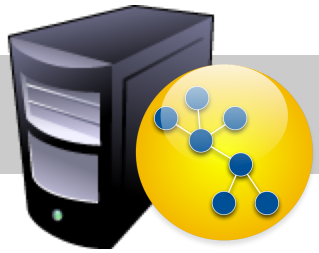


Return Results

```
MATCH (s:Shop{shop_id:1})-[r1:SELLS]->(p:Product)
WHERE (r1.from <= 1388880000000 AND r1.to > 1388880000000)
MATCH (p)-[r2:STATE]->(ps:ProductState)
WHERE (r2.from <= 1388880000000 AND r2.to > 1388880000000)
RETURN p.product_id AS productId,
       ps.name AS product,
       ps.price AS price
ORDER BY price DESC
```

Evolving a Graph Model

Refactoring



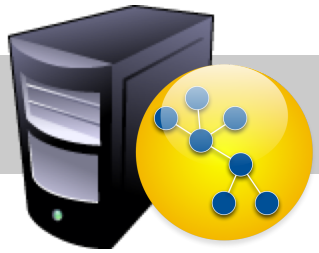
Definition

- Restructure graph without changing informational semantics

Reasons

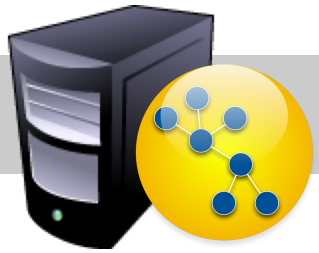
- Improve design
- Enhance performance
- Accommodate new functionality
- Enable iterative and incremental development of data model

Data Migrations



- Execute in repeatable order
- Backup database
- Execute in batches
  - Unbounded results will generate large transactions and may trigger Out of Memory exceptions
- Apply migrations to test data to ensure existing functionality doesn't break
- Ensure application can accommodate old and new structures if performing against live data

Extract Node from Property



Problem

- You've modeled an attribute as a property with a simple value, but now need to:
  - Qualify the attribute semantics AND/OR
  - Introduce a complex value AND/OR
  - Reify the relationship represented by the value

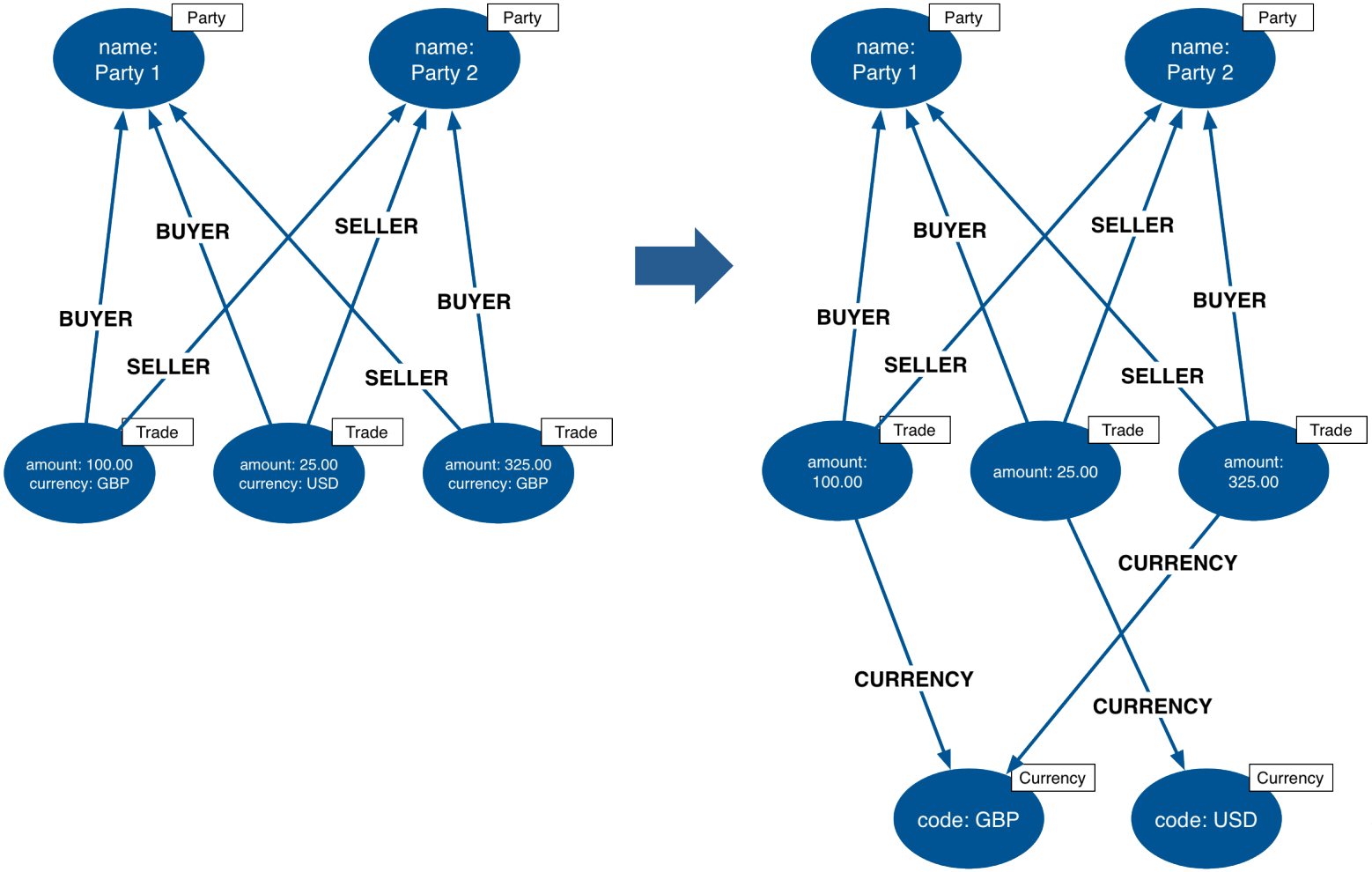
Solution

- Create a new node per unique property value
- Connect existing nodes to the new property nodes
- Remove the old property

Exercise 5

Extract Node From Property

Example: (n.currency) to (:Currency)



Your Turn

- Clean the database
- Execute *setup.txt*
- View the results
  - MATCH (n) RETURN n
- Execute *update-1.txt* repeatedly, until *numberRemoved* is zero
  - At each stage, view the results

Extract Node From Property

```
MATCH (t:Trade) WHERE has(t.currency)
WITH t LIMIT {batchSize}
MERGE (c:Currency{code:t.currency})
MERGE (t)-[:CURRENCY]->(c)
REMOVE t.currency
RETURN count(t) AS numberRemoved
```

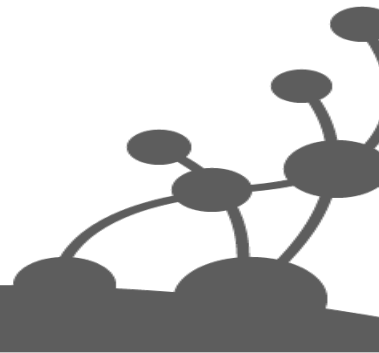
Select Batch of Nodes With Property

```
MATCH (t:Trade) WHERE has(t.currency)
WITH t LIMIT {batchSize}
MERGE (c:Currency{code:t.currency})
MERGE (t)-[:CURRENCY]->(c)
REMOVE t.currency
RETURN count(t) AS numberRemoved
```

Create Property Node

```
MATCH (t:Trade) WHERE has(t.currency)
WITH t LIMIT {batchSize}
MERGE (c:Currency{code:t.currency})
MERGE (t)-[:CURRENCY]->(c)
REMOVE t.currency
RETURN count(t) AS numberRemoved
```

Copy property value from existing node



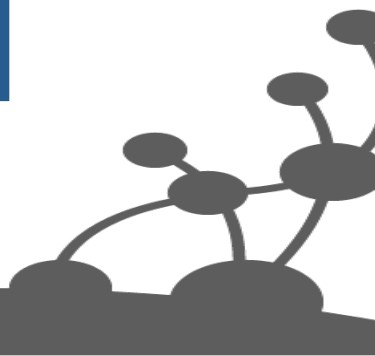
Relate Existing Node to Property Node

```
MATCH (t:Trade) WHERE has(t.currency)
WITH t LIMIT {batchSize}
MERGE (c:Currency{code:t.currency})
MERGE (t)-[:CURRENCY]->(c)
REMOVE t.currency
RETURN count(t) AS numberRemoved
```

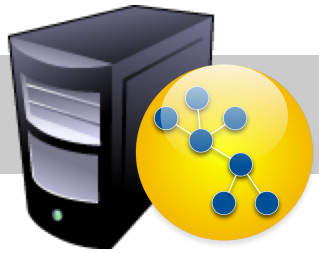
Remove Old Property

```
MATCH (t:Trade) WHERE has(t.currency)
WITH t LIMIT {batchSize}
MERGE (c:Currency{code:t.currency})
MERGE (t)-[:CURRENCY]->(c)
REMOVE t.currency
RETURN count(t) AS numberRemoved
```

Repeat until numberRemoved is zero



Extract Node from Array Property



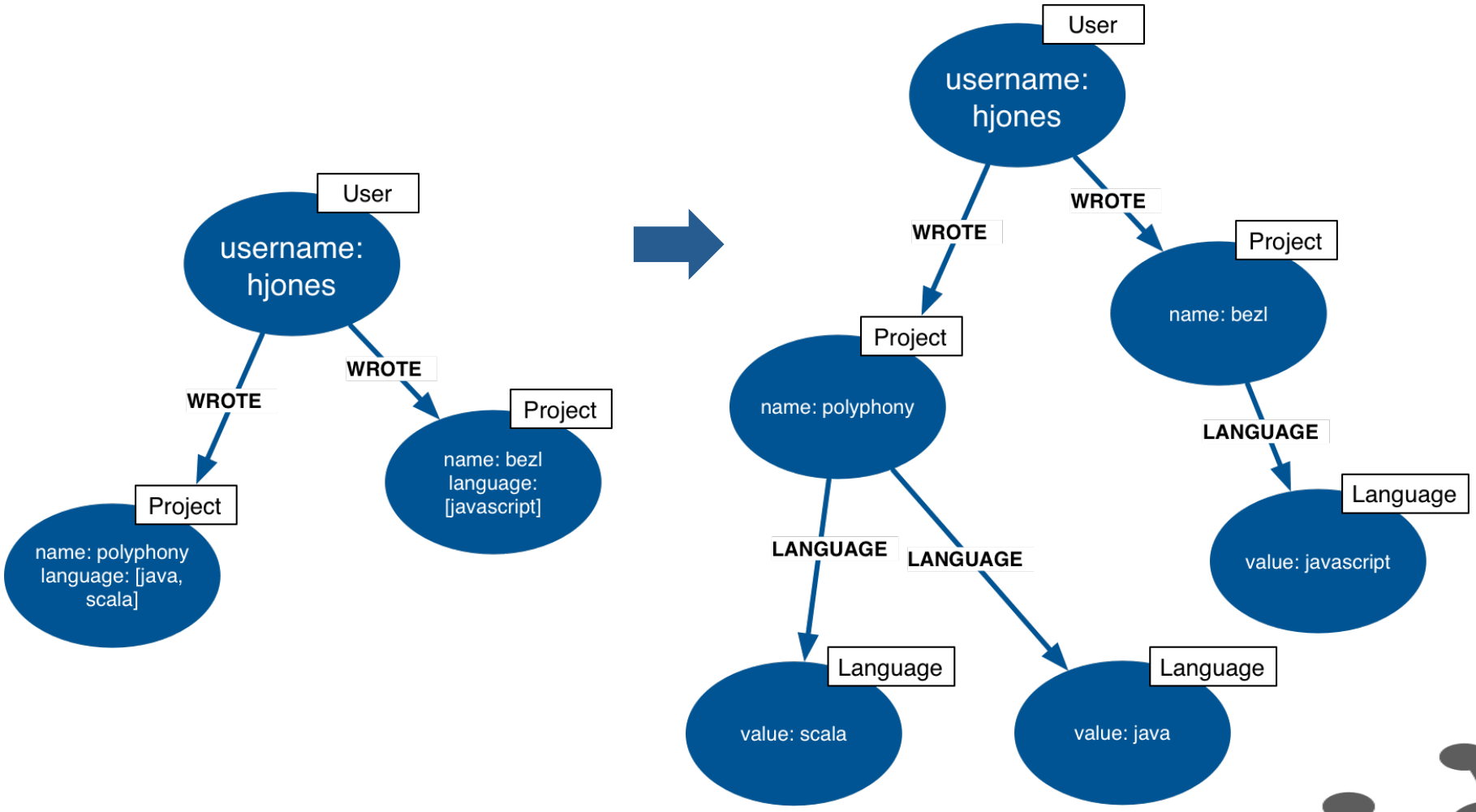
Problem

- You've modeled an attribute as a property with an array value, but now need to:
  - Qualify the attribute semantics AND/OR
  - Introduce a complex value AND/OR
  - Reify the relationship represented by the value

Solution

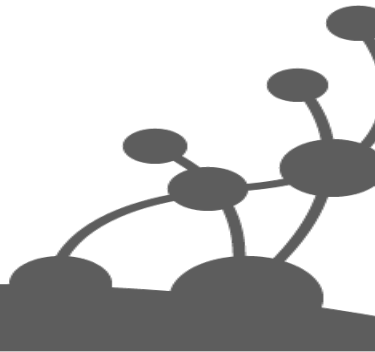
- Create a new node per unique property value
- Connect existing nodes to the new property nodes
- Remove the old property

Example: Extract Language Nodes



Exercise 6

Extract Node From Array Property



Your Turn

- Clean the database
- Execute *setup.txt*
- View the results
  - MATCH (n) RETURN n
- Execute *update-1.txt* repeatedly, until *numberRemoved* is zero
  - At each stage, view the results

Extract Node From Array Property

```
MATCH (project:Project)
WHERE has(project.language)
WITH project LIMIT 2
FOREACH (l IN project.language |
  MERGE (language:Language{value:l})
  MERGE (project)-[:LANGUAGE]->(language))
REMOVE project.language
RETURN count(project) AS numberRemoved
```

Select Batch of Nodes With Property

```
MATCH (project:Project)
WHERE has(project.language)
WITH project LIMIT 2
FOREACH (l IN project.language |
  MERGE (language:Language{value:l})
  MERGE (project)-[:LANGUAGE]->(language))
REMOVE project.language
RETURN count(project) AS numberRemoved
```

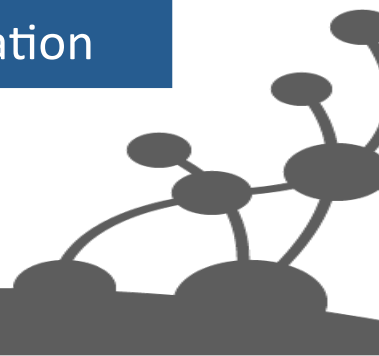
Loop Through Values in Array…

```
MATCH (project:Project)
WHERE has(project.language)
WITH project LIMIT 2
FOREACH (l IN project.language |
  MERGE (language:Language{value:l})
  MERGE (project)-[:LANGUAGE]->(language))
REMOVE project.language
RETURN count(project) AS numberRemoved
```

Create New Unique Node Per Value

```
MATCH (project:Project)
WHERE has(project.language)
WITH project LIMIT 2
FOREACH (l IN project.language |
  MERGE (language:Language{value:l})
  MERGE (project)-[:LANGUAGE]->(language))
REMOVE project.language
RETURN count(project) AS numberRemoved
```

Copy value from current iteration



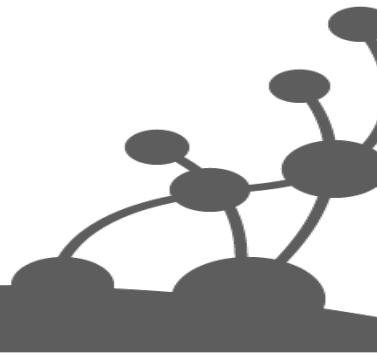
Relate Existing Node to Value Node

```
MATCH (project:Project)
WHERE has(project.language)
WITH project LIMIT 2
FOREACH (l IN project.language |
  MERGE (language:Language{value:l})
  MERGE (project)-[:LANGUAGE]->(language))
REMOVE project.language
RETURN count(project) AS numberRemoved
```

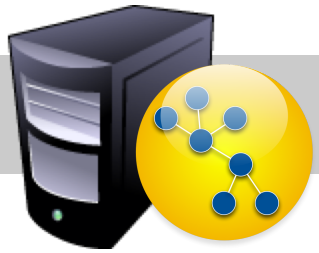
Remove Array Property

```
MATCH (project:Project)
WHERE has(project.language)
WITH project LIMIT 2
FOREACH (l IN project.language |
  MERGE (language:Language{value:l})
  MERGE (project)-[:LANGUAGE]->(language))
REMOVE project.language
RETURN count(project) AS numberRemoved
```

Repeat until numberRemoved is zero



Extract Node From Relationship



Problem

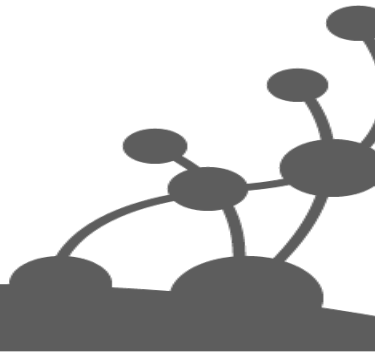
- You've modeled something as a relationship (with properties), but now need to connect it to more than two things

Solution

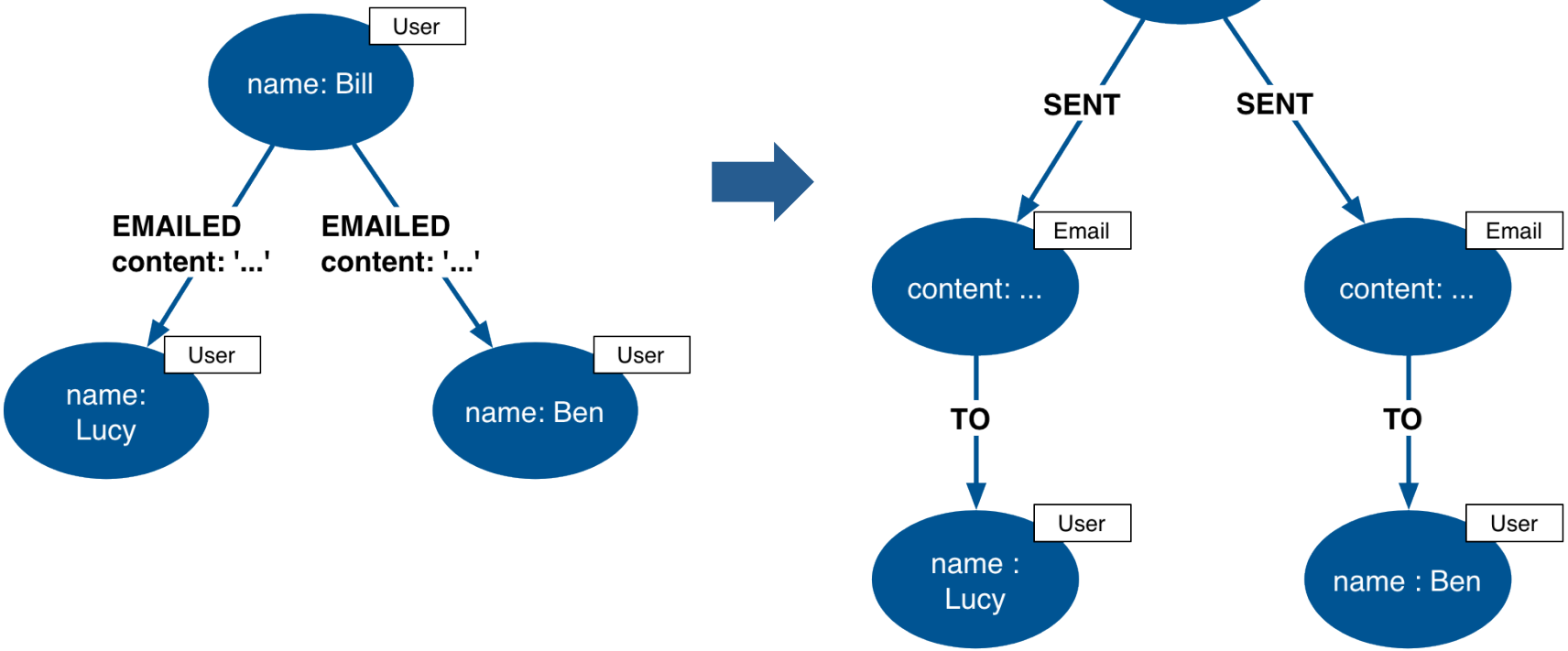
- Extract relationship into a new node (and two new relationships)
- Copy old relationship properties onto new node
- Delete old relationship

Exercise 7

Extract Node From Relationship



Example: [:EMAILED] to (:Email)



Your Turn

- Clean the database
- Execute *setup.txt*
- View the results
  - MATCH (n) RETURN n
- Execute *update-1.txt* repeatedly, until *numberDeleted* is zero
  - At each stage, view the results

Extract Node From Relationship

```
MATCH (a:User)-[r:EMAILED]->(b:User)
WITH a, r, b LIMIT 2
CREATE (email:Email{content:r.content})
MERGE (a)-[:SENT]->(email)-[:TO]->(b)
DELETE r
RETURN count(r) AS numberDeleted
```

Select Batch of Relationships

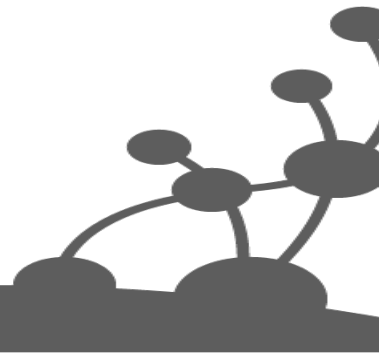
```
MATCH (a:User)-[r:EMAILED]->(b:User)
WITH a, r, b LIMIT 2
CREATE (email:Email{content:r.content})
MERGE (a)-[:SENT]->(email)-[:TO]->(b)
DELETE r
RETURN count(r) AS numberDeleted
```

Create New Node and Relationships

```
MATCH (a:User)-[r:EMAILED]->(b:User)
WITH a, r, b LIMIT 2
CREATE (email:Email{content:r.content})
MERGE (a)-[:SENT]->(email)-[:TO]->(b)
DELETE r
RETURN count(r) AS numberDeleted
```

Copy properties from old relationship

"Refactoring ID" ensures uniqueness



Delete Old Relationship

```
MATCH (a:User)-[r:EMAILED]->(b:User)
WITH a, r, b LIMIT 2
CREATE (email:Email{content:r.content})
MERGE (a)-[:SENT]->(email)-[:TO]->(b)
DELETE r
RETURN count(r) AS numberDeleted
```

Repeat until numberDeleted is zero