Introduction to Graph Database with Cypher & Neo4j

Zeyuan Hu

03/18/24

Austin, TX

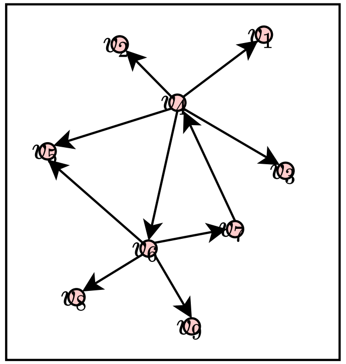
(Graph) Database (Graph mostly acyclic)

External Level

Application View (e.g., social networks)

(graph) database

Logic Level



Graph Model

$Edge$

tail	head
v_4	v_2
v_4	v_1
v_4	v_5
v_4	v_3
v_4	v_6
v_6	v_5
v_6	v_7
v_6	v_8
v_6	v_9
v_7	v_4

Relational Model

...

Physical Level

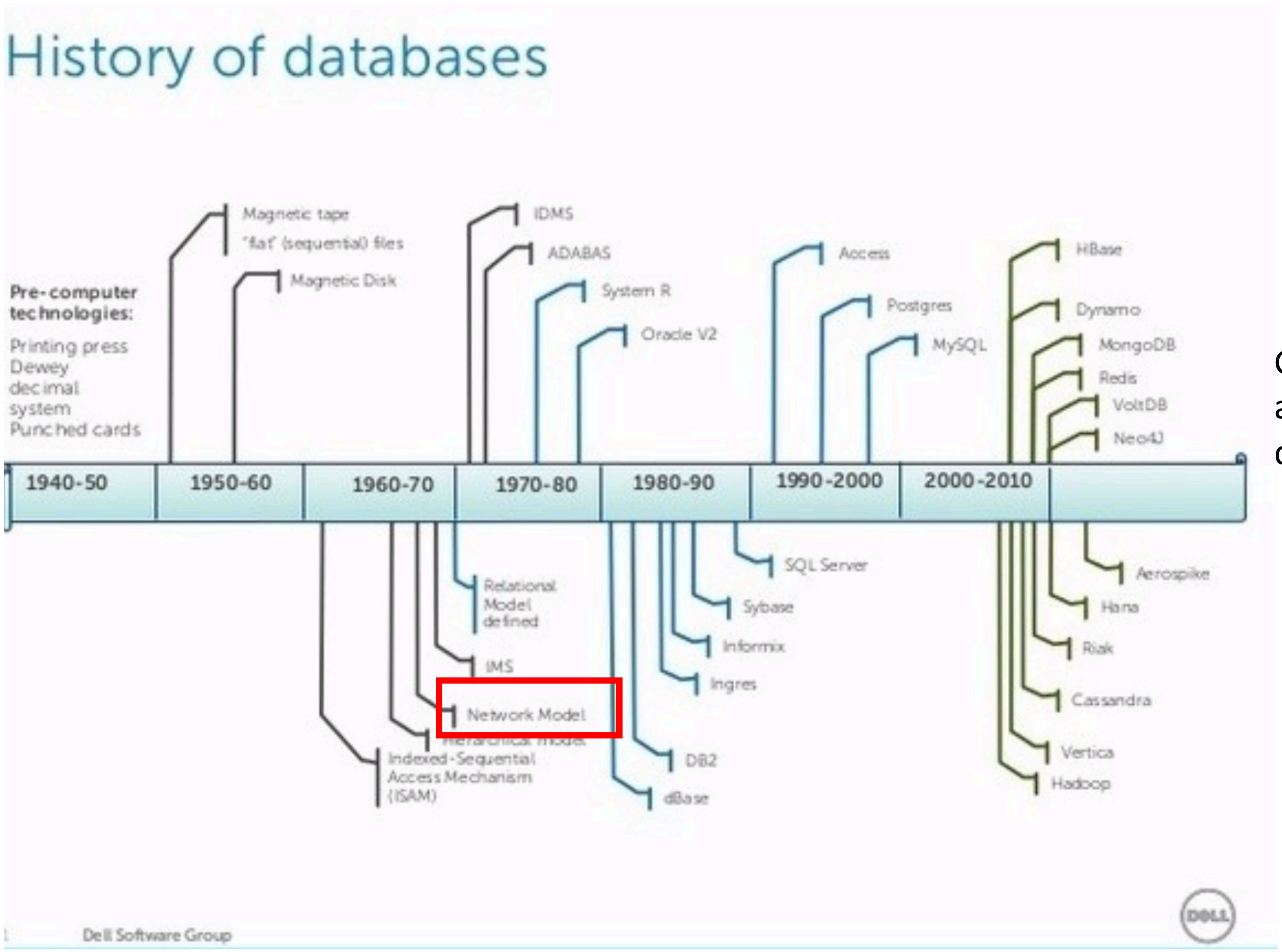
Store graph natively (e.g., adjacency list, adjacency matrix, edge list)

Store graph as a collection of tuples (e.g., vertex = tuple, edge = pointers between two tuples)

Store graph as a collection of tables (e.g., vertex table, edge table)

Store graph as a collection of key-value pairs (e.g., key = vertex, value = the vertex's neighbors)

...

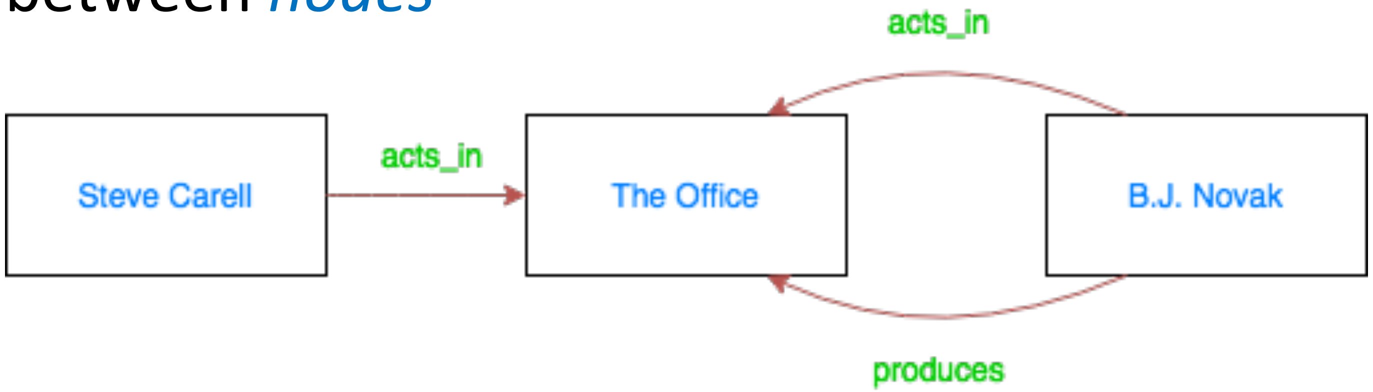


History of databases

Graph database uses data models that are "spiritual successors" of Network data model that is popular in 1970's.

Edge-labelled Graph

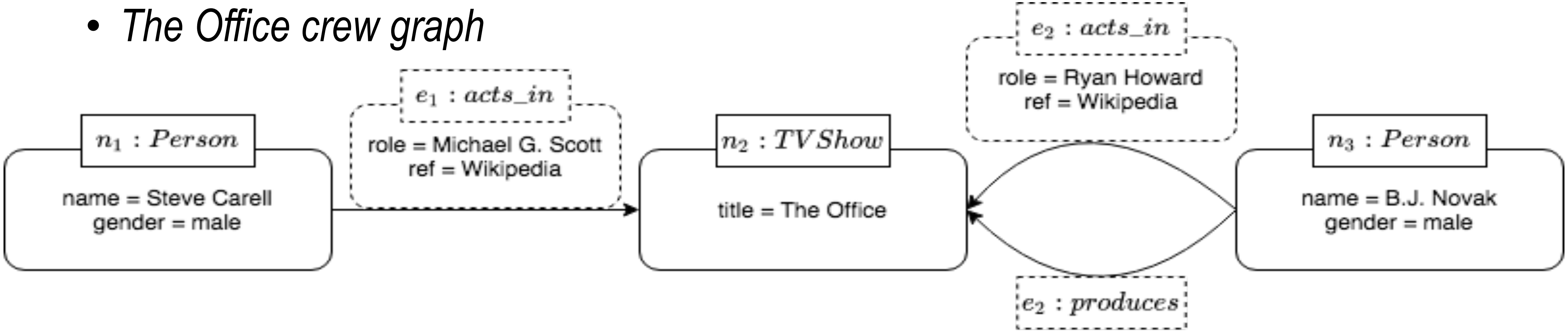
- We assign *labels* to *edges* that indicate the different types of relationships between *nodes*



- *Nodes* = {*Steve Carell*, *The Office*, *B.J. Novak*}
- *Edges* = {(*Steve Carell*, *acts_in*, *The Office*), (*B.J. Novak*, *produces*, *The Office*), (*B.J. Novak*, *acts_in*, *The Office*)}
- Basis of *Resource Description Framework (RDF) aka. "Triplestore"*

The Property Graph Model

- Extends Edge-labelled Graph with labels
  - Both edges and nodes can be labelled with a set of property-value pairs *attributes* directly to each edge or node.
  - *The Office crew graph*



- Node n_1 has node label *Person* with *attributes*: <name, Steve Carell>, <gender, male>
- Edge e_1 has edge label *acts_in* with *attributes*: <role, Michael G. Scott>, <ref, Wikipedia>

Property Graph v.s. Edge-labelled Graph

- Having node labels as part of the model can offer a more direct abstraction that is easier for users to query and understand
    - *Steve Carell and B.J. Novak can be labelled as Person*
- Suitable for scenarios where various new types of meta-information may regularly need to be added to edges or nodes

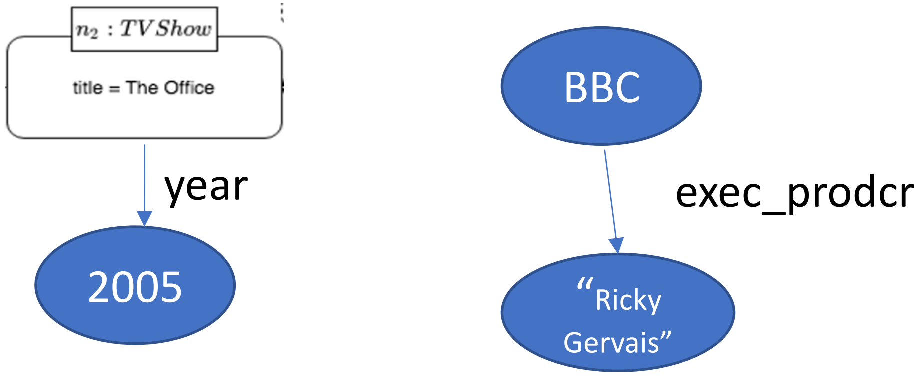
Graph v.s. Relational Model

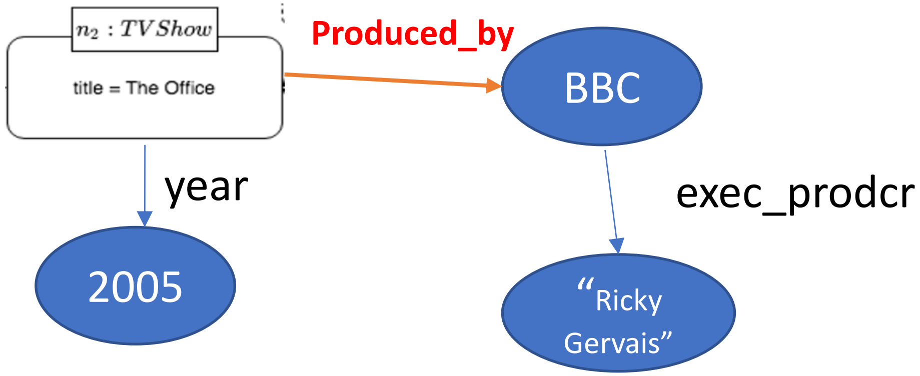
Schema Extensibility

Useful for data integration tasks (e.g., link discovery)

```
CREATE TABLE TVSHOW(title, year);
CREATE TABLE Production_Company(name,
    exec_prodcr)
```

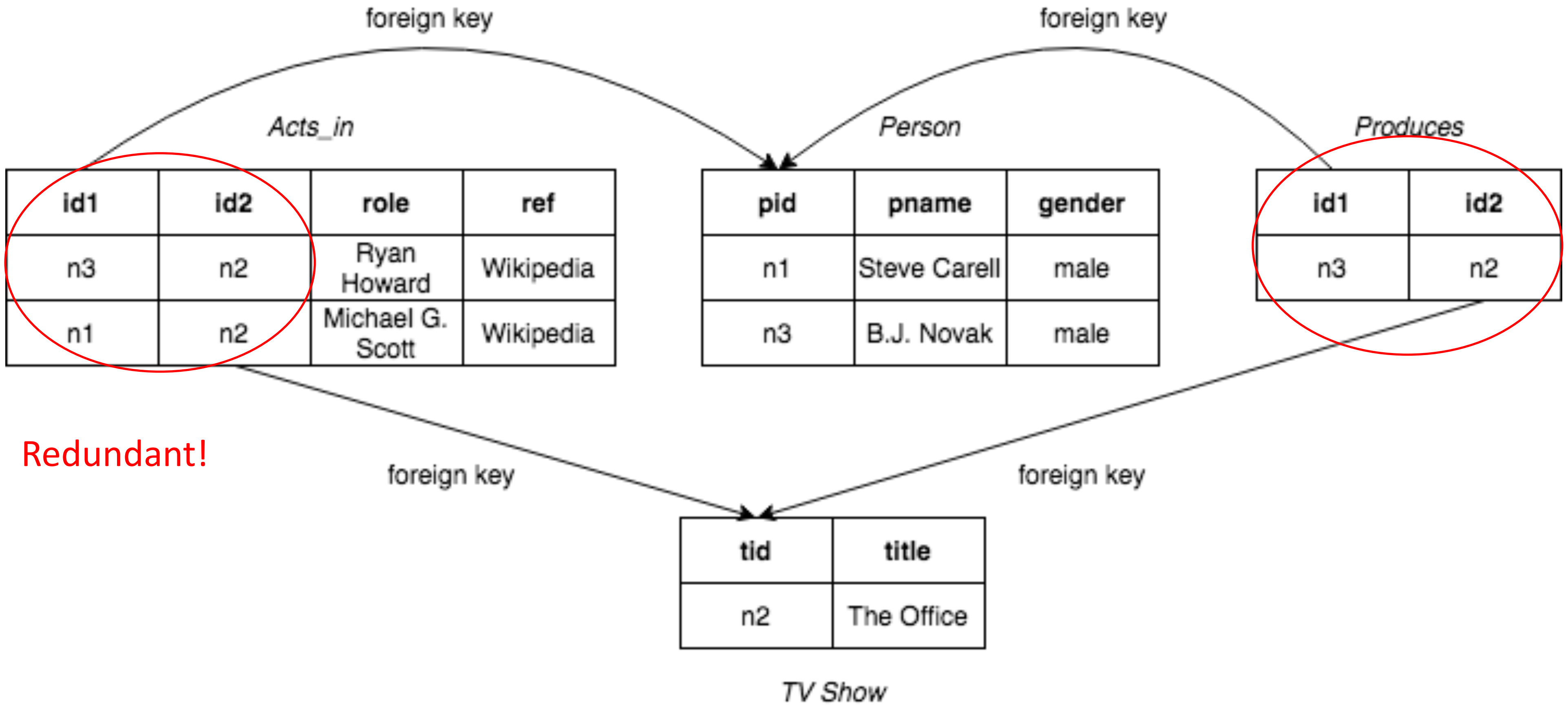
```
graph: add an edge
SQL: redefine relational schema
 ALTER TABLE TVSHOW(title, year,
    production_company references …);
-- foreign key constraint
```





Same Data, Different Model

- The same data represented in relational model



Acts_in

id1	id2	role	ref
n3	n2	Ryan Howard	Wikipedia
n1	n2	Michael G. Scott	Wikipedia

Person

pid	pname	gender
n1	Steve Carell	male
n3	B.J. Novak	male

Produces

id1	id2
n3	n2

tid	title
n2	The Office

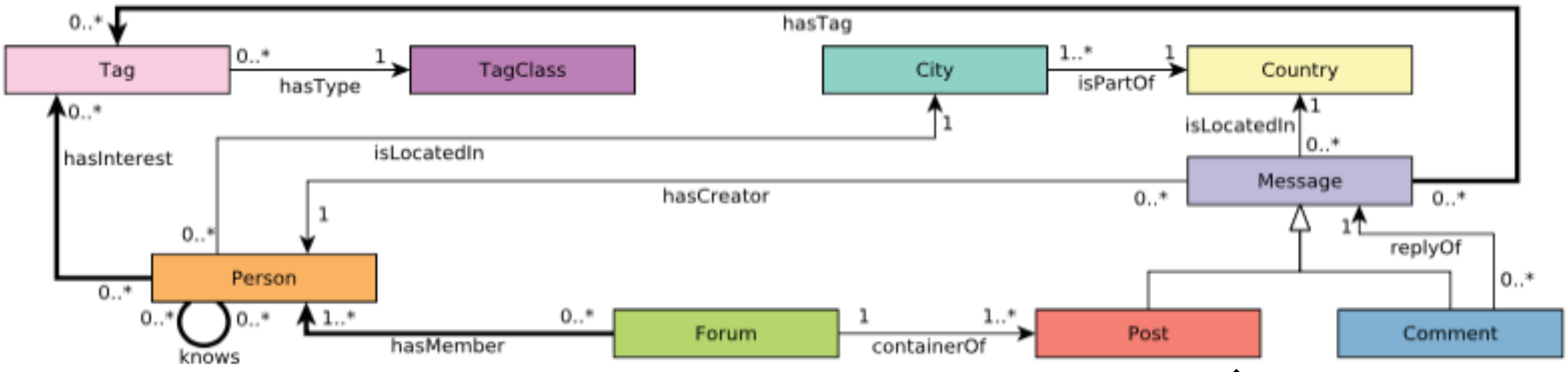
TV Show

Redundant!

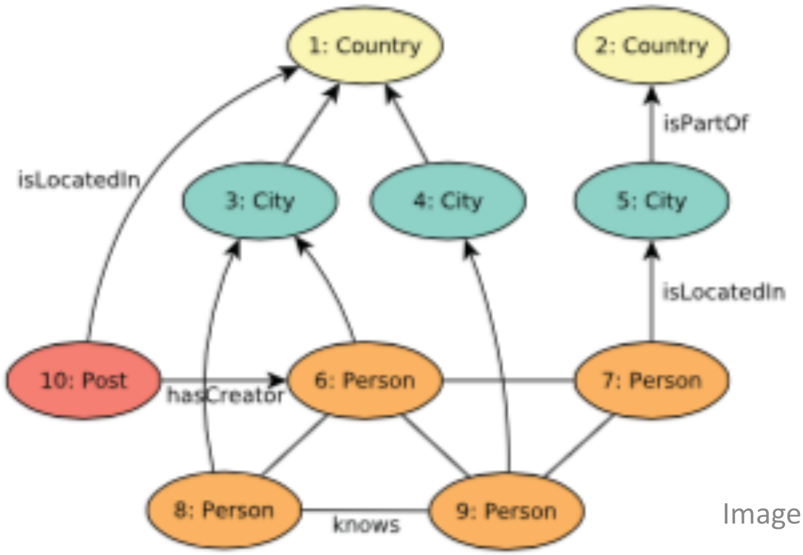
Property Graph v.s. Relational Model

- Graph Structure is more intuitive than a collection of tables (e.g., org chart)
- Ambiguity in graph representation using relational model (directed or undirected?)
- Avoid repetitive data storage *from user perspective* (e.g., primary key & foreign key)
- Enable same relation name with different attributes
  - `CREATE TABLE TVSHOW(title, year);`
  - `CREATE TABLE TVSHOW(title, year, production_company);` // Not possible!
- Nice query language for graph problems

Data Modeling is still relevant in Graphs



Graph schema visualized in UML



Example Graph

Same data model, different physical model (relational, property graph, etc)

Images are from "LSQB: A Large-Scale Subgraph Query Benchmark"

Neo4j

- Neo4j is a graph database that uses *property graph* data model with a query language called *Cypher*
- In graph database domain, there is no standard query language (yet). Many vendor-dependent flavors
  - SPARQL for RDF
  - Cypher, Gremlin, etc. for property graph
  - *Ex: Find co-stars of The Office*

```
PREFIX : <http://ex.org/#>
SELECT ?x1 ?x2
WHERE {
      ?x1 :acts_in ?x3 . ?x1 :type :Person .
      ?x2 :acts_in ?x3 . ?x2 :type :Person .
      ?x3 :title "The Office" . ?x3 :type :TVSHOW .
      FILTER(?x1 != ?x2)
}
```

```
MATCH (x1:Person) -[:acts_in]->
      (:TVSHOW {title: "The Office"})
      <-[:acts_in]- (x2:Person)
RETURN x1, x2
```

```
g.V().has("TVSHOW", "title", "The Office").
in('acts_in').hasLabel("Person").
values("name")
```

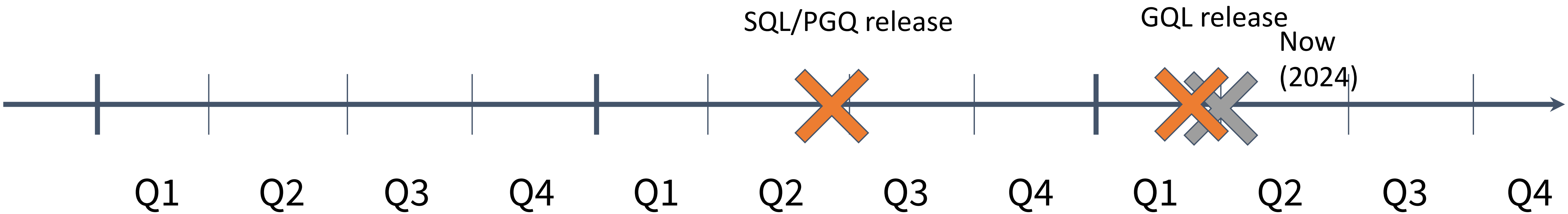
Query languages

neo4j	Oracle Labs PGX	TigerGraph	XTDB
Cypher	PGQL	GSQL	Datalog
relationalAI	Dgraph	Amazon Neptune	JanusGraph
Rel	DQL	SPARQL, Cypher, Gremlin	Gremlin

Slides adapted from "Efficient Analytical and Graph Queries ” by Peter Boncz (CWI)

GQL and SQL/PGQ

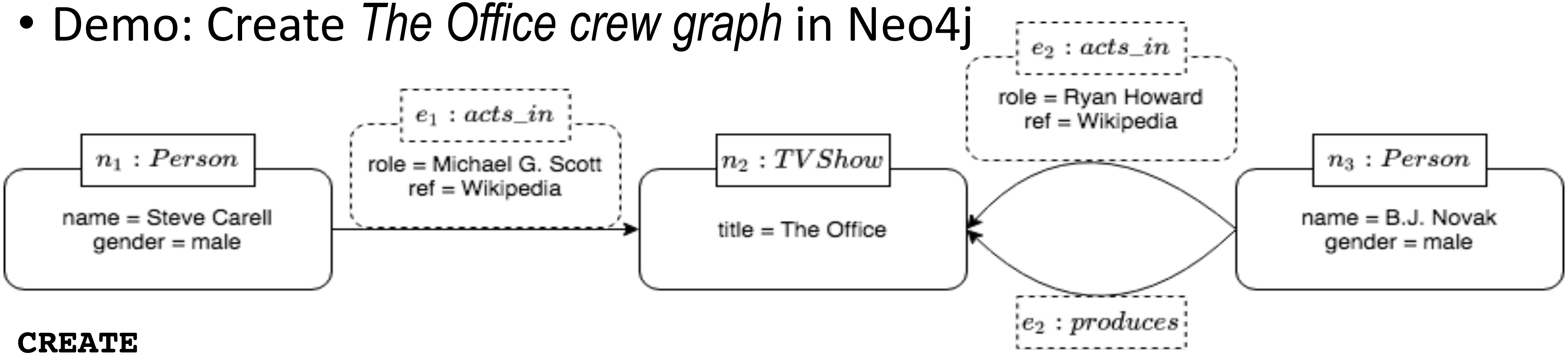
- Ongoing standardization effort just like SQL for relational model
- Two efforts
  - Graph Query Language (GQL) – completely new language for graph database
  - SQL/PGQ (Property Graph Queries) – extend SQL with graph query capabilities, e.g., path finding, pattern matching
- Graph component of two languages are compatible



Slides adapted from "Efficient Analytical and Graph Queries " by Peter Boncz (CWI)

First Property Graph with Neo4j

- Demo: Create *The Office crew graph* in Neo4j

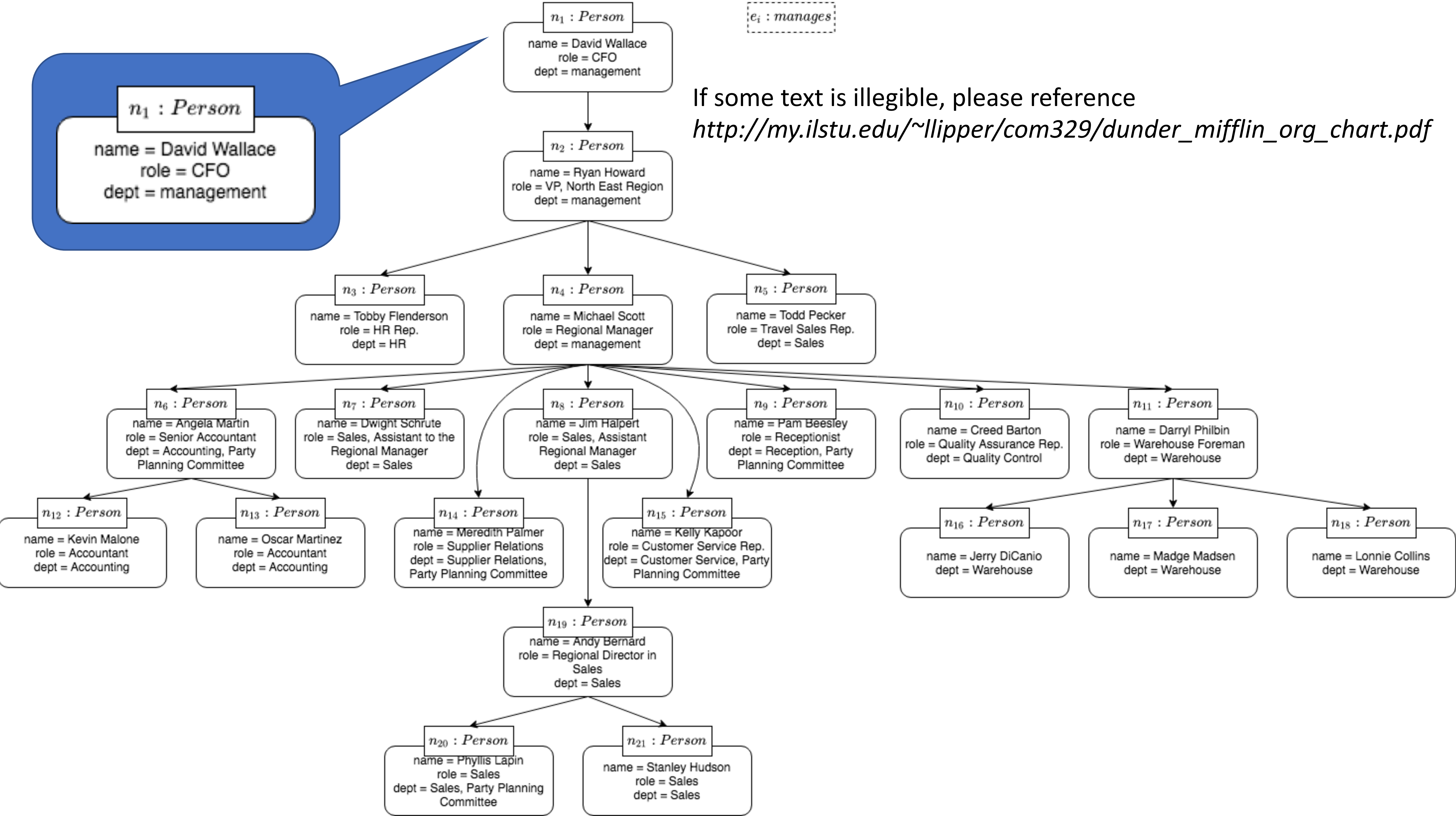


```
CREATE
(n1:Person {name: "Steve Carell", gender: "male"}),
(n2:Person {name: "B.J. Novak", gender: "male"}),
(n3:TVShow {title: "The Office"}),
(n1)-[:acts_in {role: "Michael G. Scott", ref: "Wikipedia"}]->(n2),
(n2)-[:acts_in {role: "Ryan Howard", ref: "Wikipedia"}]->(n3),
(n2)-[:produces]->(n3);
```

Let’s Practice

- Let’s create the org. chart of the paper company Dunder Mifflin, Scranton Branch [1] in The Office.
- All edges have labels $e_i\!:\!manages$ with i being numbers from 1 to n, the number of edges
- Some useful commands & notes
  - See the graph - `MATCH (n) RETURN n LIMIT 50`
  - Delete the graph - `MATCH (n) DETACH DELETE n`
  - To create list of values, use ”[]”
    - For example, `role: ["Sales", "Assistant Regional Manager"]`

[1] Season 4 Ep 16. Source: https://www.officetally.com/wp-content/uploads/2008/05/dunder_mifflin_org_chart.pdf



$e_i : manages$

$n_1 : Person$

name = David Wallace
role = CFO
dept = management

$n_1 : Person$

name = David Wallace
role = CFO
dept = management

$n_2 : Person$

name = Ryan Howard
role = VP, North East Region
dept = management

$n_3 : Person$

name = Tobby Flenderson
role = HR Rep.
dept = HR

$n_4 : Person$

name = Michael Scott
role = Regional Manager
dept = management

$n_5 : Person$

name = Todd Pecker
role = Travel Sales Rep.
dept = Sales

$n_6 : Person$

name = Angela Martin
role = Senior Accountant
dept = Accounting, Party Planning Committee

$n_7 : Person$

name = Dwight Schrute
role = Sales, Assistant to the Regional Manager
dept = Sales

$n_8 : Person$

name = Jim Halpert
role = Sales, Assistant Regional Manager
dept = Sales

$n_9 : Person$

name = Pam Beesley
role = Receptionist
dept = Reception, Party Planning Committee

$n_{10} : Person$

name = Creed Barton
role = Quality Assurance Rep.
dept = Quality Control

$n_{11} : Person$

name = Darryl Philbin
role = Warehouse Foreman
dept = Warehouse

$n_{12} : Person$

name = Kevin Malone
role = Accountant
dept = Accounting

$n_{13} : Person$

name = Oscar Martinez
role = Accountant
dept = Accounting

$n_{14} : Person$

name = Meredith Palmer
role = Supplier Relations
dept = Supplier Relations, Party Planning Committee

$n_{15} : Person$

name = Kelly Kapoor
role = Customer Service Rep.
dept = Customer Service, Party Planning Committee

$n_{16} : Person$

name = Jerry DiCanio
dept = Warehouse

$n_{17} : Person$

name = Madge Madsen
dept = Warehouse

$n_{18} : Person$

name = Lonnie Collins
dept = Warehouse

$n_{19} : Person$

name = Andy Bernard
role = Regional Director in Sales
dept = Sales

$n_{20} : Person$

name = Phyllis Lapin
role = Sales
dept = Sales, Party Planning Committee

$n_{21} : Person$

name = Stanley Hudson
role = Sales
dept = Sales

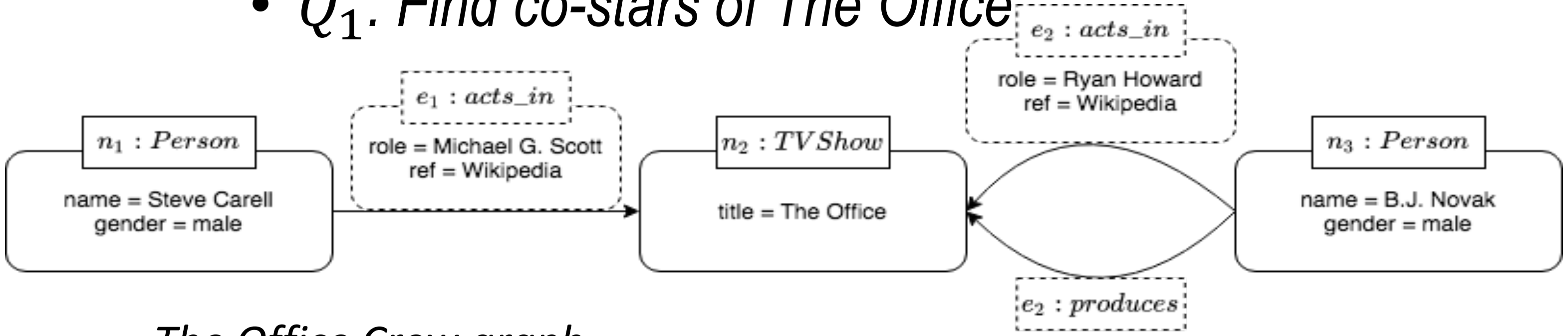
If some text is illegible, please reference *http://my.ilstu.edu/~llipper/com329/dunder_mifflin_org_chart.pdf*

Graph Query Languages

- Two important usage patterns for graph query languages:
    - Graph Pattern Matching
    - Graph Navigation
- We’ll focus on Cypher in this tutorial. However, any significant graph query languages will have these two important patterns in their languages.

Graph Pattern Matching

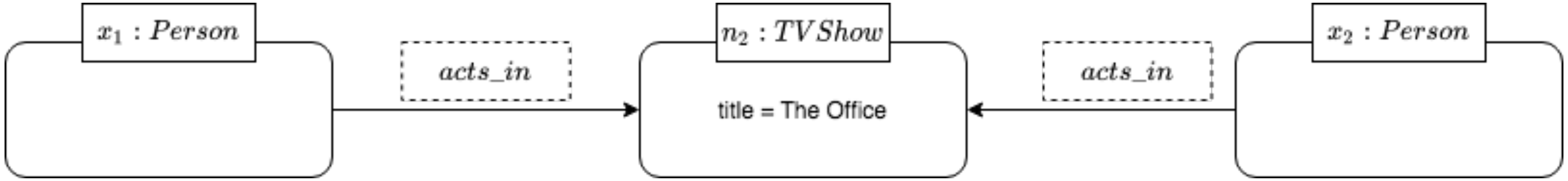
- Graph Pattern Matching
  - A *match* is a mapping from variables to constants such that when the mapping is applied to the given pattern, the result is, roughly speaking, contained within the original graph (i.e., subgraph).
  - *Q_1: Find co-stars of The Office*



The Office Crew graph

x_1	x_2
Steve Carell	B.J. Novak
B.J. Novak	Steve Carell

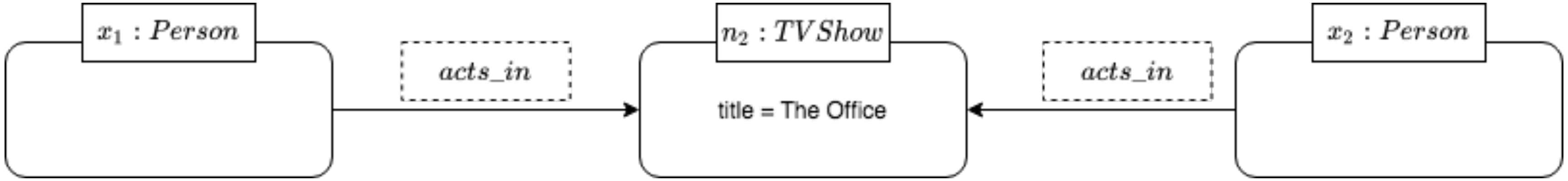
Result set (i.e., *matching*) for Q_1



graph pattern for Q_1

Graph Pattern Matching in Cypher

- Cypher has no-repeated-edges, bags semantics
- Q_1: *Find co-stars of The Office*



```
MATCH (x1:Person) -[:acts_in]-> (:TVSHOW {title: "The Office"}) <-[:acts_in]- (x2:Person)
RETURN x1, x2
```

Match pattern

x_1 has to connect to `TVShow` node through an incoming edge with edge label `acts_in`

We want to match variable x_1 to node with node label Person

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

Let’s Practice

- Who’s inside Party Planning Committee (PPC)? (*hint: PPC is a dept)*

```
MATCH (p:Person)
WHERE "Party Planning Committee" in p.dept
return p.name
```

- How many people does Michael directly manage? (*hint: use* ***`count()`***)

```
MATCH (p:Person)<-[:manages]-(n:Person)
WHERE n.name = "Michael Scott"
RETURN count(p)
```

Let’s Practice

- Find all the employees that are directly managed by someone that reports to Michael

```
MATCH (p {name: 'Michael Scott'})-[:manages]->()-[:manages]->(q)
RETURN q.name
```

- Does Michael directly manage more employees than Jim Halpert?

```
MATCH (p:Person)<-[:manages]-(n:Person)
WHERE n.name = "Michael Scott"
WITH count(p) AS c1
MATCH (p:Person)<-[:manages]-(m:Person)
WHERE m.name = "Jim Halpert"
RETURN c1 > count(p)
```

Each MATCH ... WHERE can be thought as a SELECT ... FROM ... WHERE

```
MATCH (p:Person)<-[:manages]-(n:Person)
WHERE n.name = "Michael Scott"
MATCH (q:Person)<-[:manages]-(m:Person)
WHERE m.name = "Darryl Philbin"
RETURN p.name, q.name
```

Graph Navigation

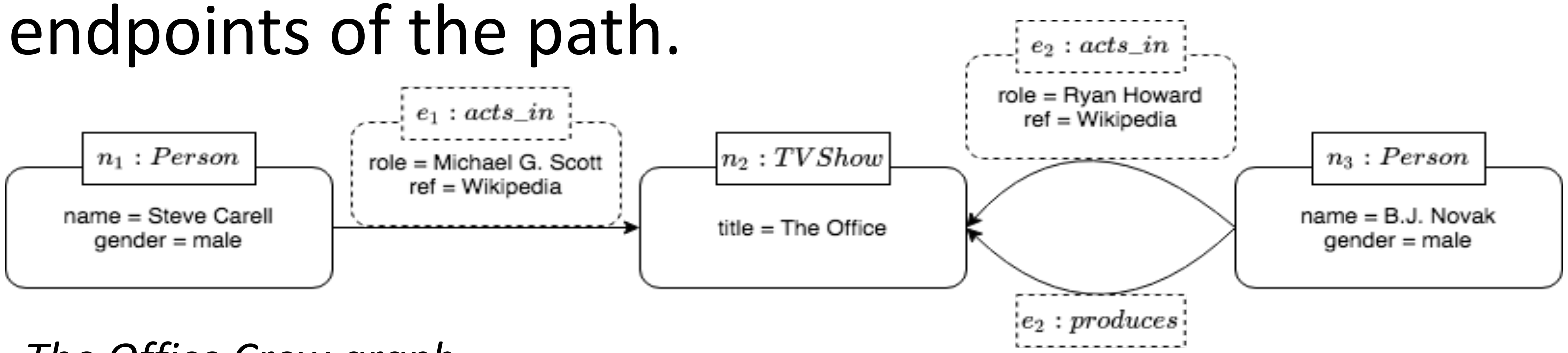
- A mechanism provided by graph query languages to *navigate* the topology of the data.
- Two important query classes:
  - Path Query
  - Path Query + Graph Pattern Matching (i.e., navigational graph pattern)

Path Query

- Previously, we match a graph pattern; now, we attern.

Often represented using Regular Expressions

- Path query has the general form $P = x \xrightarrow{\alpha} y$ where α specifies conditions on the <span style="color:red">paths</span> we wish to retrieve and x and y are the endpoints of the path.



The Office Crew graph

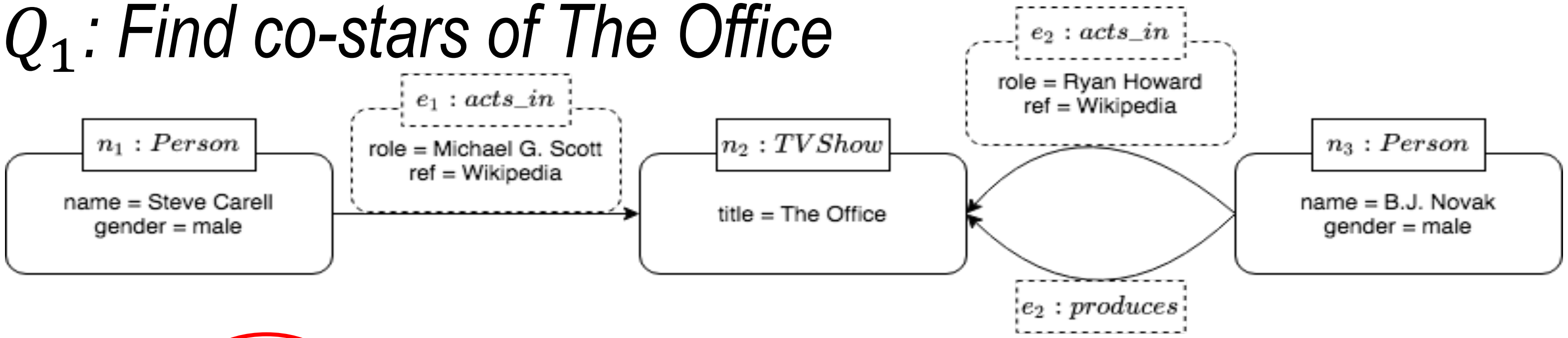
- Q_1: *Find co-stars of The Office*

$$P := x \xrightarrow{acts\_in \cdot acts\_in^{-}} y$$

Edge has direction!

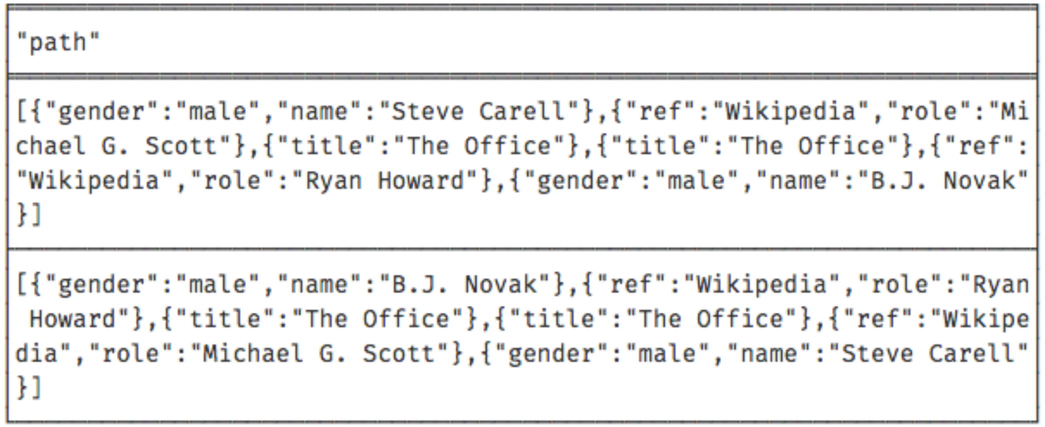
Path Query in Cypher

- Cypher has no-repeated-edge, bags semantics
- *Q_1: Find co-stars of The Office*



```
MATCH path=(p:Person)-[:acts_in]->(:TVShow)<-[:acts_in]-(q:Person)
return path
```

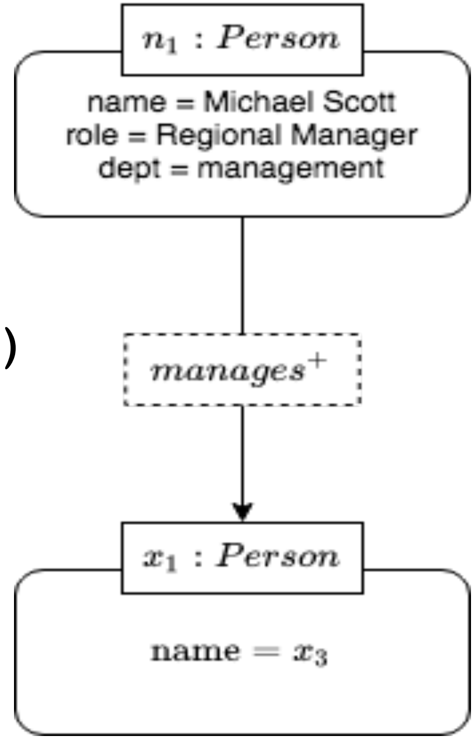
- Nothing new but we return a path now!



```
"path"

[{"gender":"male","name":"Steve Carell"},{"ref":"Wikipedia","role":"Mi
chael G. Scott"},{"title":"The Office"},{"title":"The Office"},{"ref":
"Wikipedia","role":"Ryan Howard"},{"gender":"male","name":"B.J. Novak"
}]

[{"gender":"male","name":"B.J. Novak"},{"ref":"Wikipedia","role":"Ryan
 Howard"},{"title":"The Office"},{"title":"The Office"},{"ref":"Wikipe
dia","role":"Michael G. Scott"},{"gender":"male","name":"Steve Carell"
}]
```

Navigational Graph Pattern in Cypher

- We can combine path query with graph pattern matching by allowing edge labels in the graph pattern to be paths
- *Q2: Find all the people that Michael Scott manages*



```
MATCH path=(p:Person)-[:manages*1..]->(q:Person)
WHERE p.name = "Michael Scott"
return q.name
```

- *Resources: https://neo4j.com/docs/cypher-manual/current/syntax/patterns/#cypher-pattern-relationship*

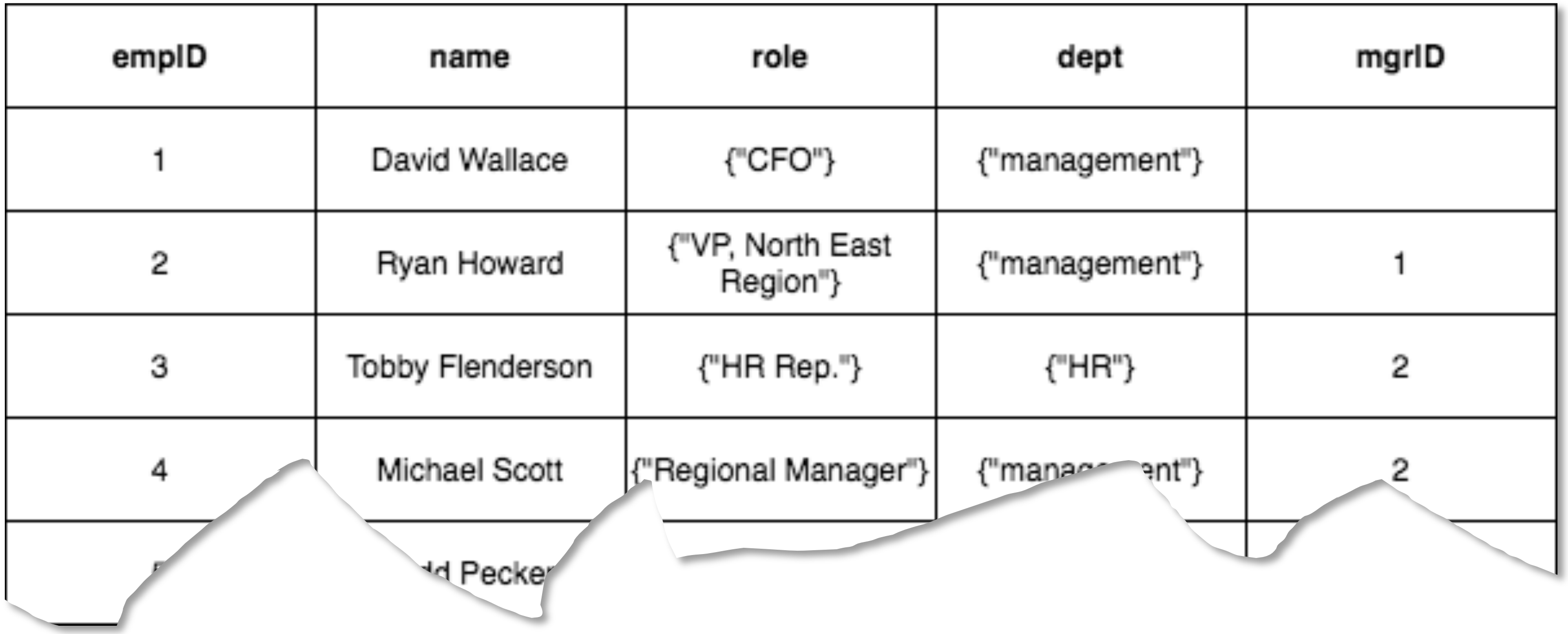
Let’s Practice

- Get the Dunder Mifflin employees that are on the same level as “Michael Scott” (*hint: use `length()`on path*)

```
MATCH p1 = (n:Person)<-[:manages*]-(p:Person)
MATCH p2 = (m:Person)<-[:manages*]-(p:Person)
WHERE length(p1) = length(p2) AND m.name <> n.name AND n.name = "Michael Scott"
RETURN m
```

Same Data, Different Model

- Let’s query the same data in Relational Model



empID	name	role	dept	mgrID
1	David Wallace	{"CFO"}	{"management"}	
2	Ryan Howard	{"VP, North East Region"}	{"management"}	1
3	Tobby Flenderson	{"HR Rep."}	{"HR"}	2
4	Michael Scott	{"Regional Manager"}	{"manag ent"}	2
[illegible]	ld Pecker			

- Actual schema and data see “sql-ex-2.sql”

Same Data, Different Model

```
MATCH p1 = (n:Person)<-[:manages]-(p:Person)
MATCH p2 = (m:Person)<-[:manages]-(p:Person)
WHERE length(p1) = length(p2) AND m.name <>
n.name AND n.name = "Michael Scott"
RETURN m
```

- Get the Dunder Mifflin employees that are on the same level as "Michael Scott"

```
with recursive samelevel(s1, s2, s3, s4) as (
     (select a1.name, a1.mgrID, a2.name, a2.mgrID
      from dunderMifflin a1, dunderMifflin a2
      where a1.mgrID = a2.mgrID)
     union
     (select a1.name, a1.mgrID, a2.name, a2.mgrID
      from dunderMifflin a1, dunderMifflin a2, samelevel l1
      where a1.mgrID = l1.s2 and a2.mgrID = l1.s4)
) select l2.s3 from samelevel L2 where l2.s1 = 'Michael Scott' and l2.s1 <> l2.s3;
```

Base case: if two people are at the same level, their manager has to be the same.

Recursion: Same idea as base case but use the base relation and the result table we just computed in base case.

Let’s Practice

- Does Jim Halpert manage Phyllis Lapin?

```
MATCH path=(p:Person)-[:manages*1..]->(q:Person)
WHERE p.name = "Jim Halpert" and q.name = "Phyllis Lapin"
return count(path) > 0
```

- Find all people that are indirectly managed by Michael Scott (*hint: use* ***`distinct`***)

```
MATCH path=(p1:Person {name: "Michael Scott"})-[:manages*1..]->()
            -[:manages*1..]->(p2:Person)
return distinct p2
```

Why do we use `distinct p2` rather than `distinct path`?

Graph Algorithms in Cypher

- Cypher and many graph query languages allow user to directly embed graph algorithms inside the query
- *Q3: Find the shortest path between David Wallace and Andy Bernard*

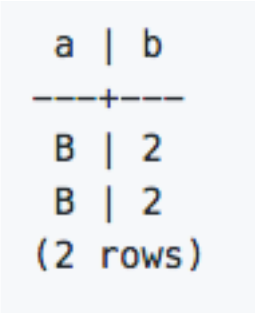
```
MATCH path = shortestPath(
(p:Person {name: "David Wallace"})-[:manages*1..]-(q:Person {name: "Andy Bernard"}))
RETURN path
```

- More graph algorithms: PageRank, Centrality, Connected Component algorithms, etc.

Understand a Cypher Query

- Neo4j has `EXPLAIN` command; just like `EXPLAIN` in any RDBMS vendor
- Running example for this section
  - Data modelled in Relational Model

```
CREATE TABLE R3(a char(1));
CREATE TABLE R2(a char(1), b integer);
CREATE TABLE R1(b integer);

INSERT INTO R3(a) VALUES (‘A’),(‘B’),(‘B’);
INSERT INTO R2(a,b) VALUES
(‘A’,1),(‘A’,1),(‘B’,1),(‘B’,2);
INSERT INTO R1(b) VALUES (2),(3);
```

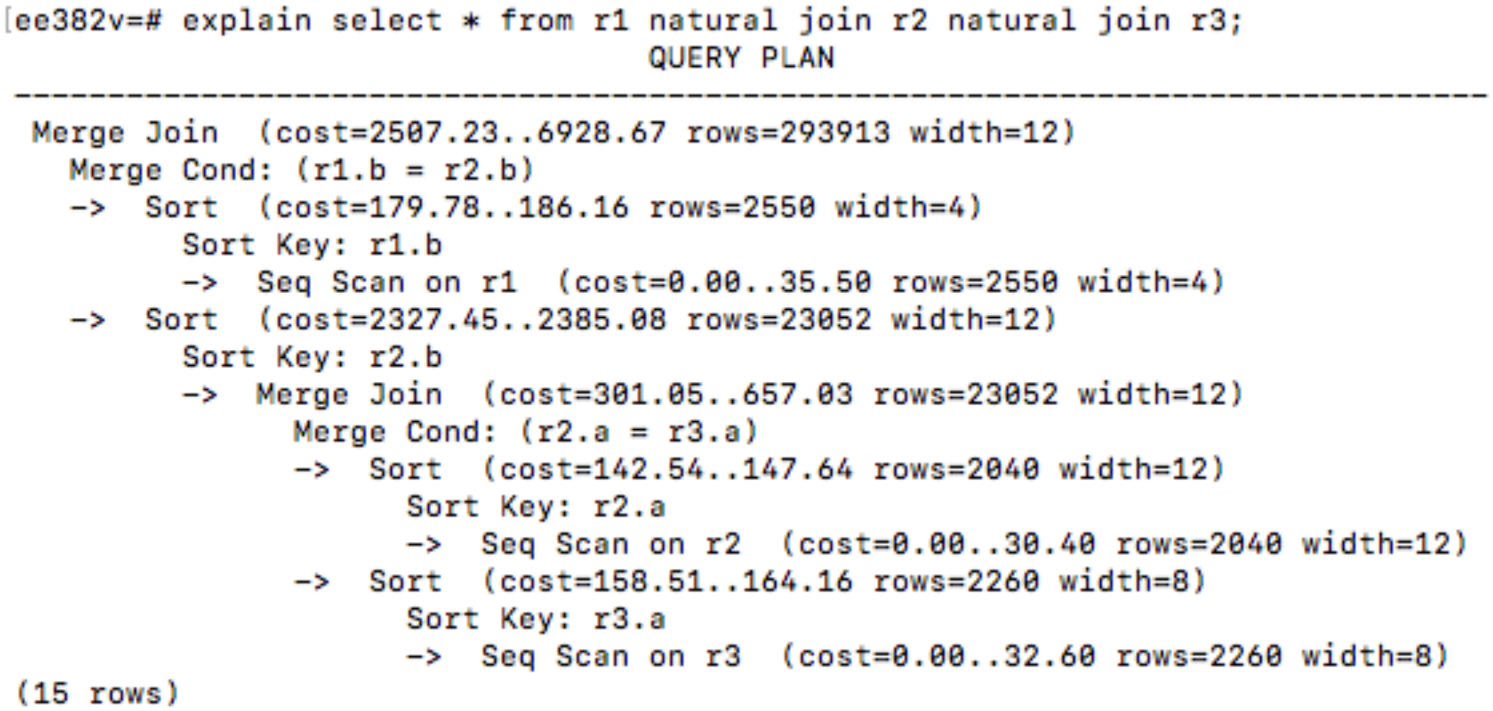


```
 a | b
---+---
 B | 2
 B | 2
(2 rows)
```

  - Query: `SELECT * FROM r1 NATURAL JOIN r2 NATURAL JOIN r3;`

Understand a Cypher Query

- The plan for SQL query looks like



```
[ee382v=# explain select * from r1 natural join r2 natural join r3;
                                   QUERY PLAN
 -------------------------------------------------------------------------------
  Merge Join  (cost=2507.23..6928.67 rows=293913 width=12)
    Merge Cond: (r1.b = r2.b)
    ->  Sort  (cost=179.78..186.16 rows=2550 width=4)
          Sort Key: r1.b
          ->  Seq Scan on r1  (cost=0.00..35.50 rows=2550 width=4)
    ->  Sort  (cost=2327.45..2385.08 rows=23052 width=12)
          Sort Key: r2.b
          ->  Merge Join  (cost=301.05..657.03 rows=23052 width=12)
                Merge Cond: (r2.a = r3.a)
                ->  Sort  (cost=142.54..147.64 rows=2040 width=12)
                      Sort Key: r2.a
                      ->  Seq Scan on r2  (cost=0.00..30.40 rows=2040 width=12)
                ->  Sort  (cost=158.51..164.16 rows=2260 width=8)
                      Sort Key: r3.a
                      ->  Seq Scan on r3  (cost=0.00..32.60 rows=2260 width=8)
 (15 rows)
```

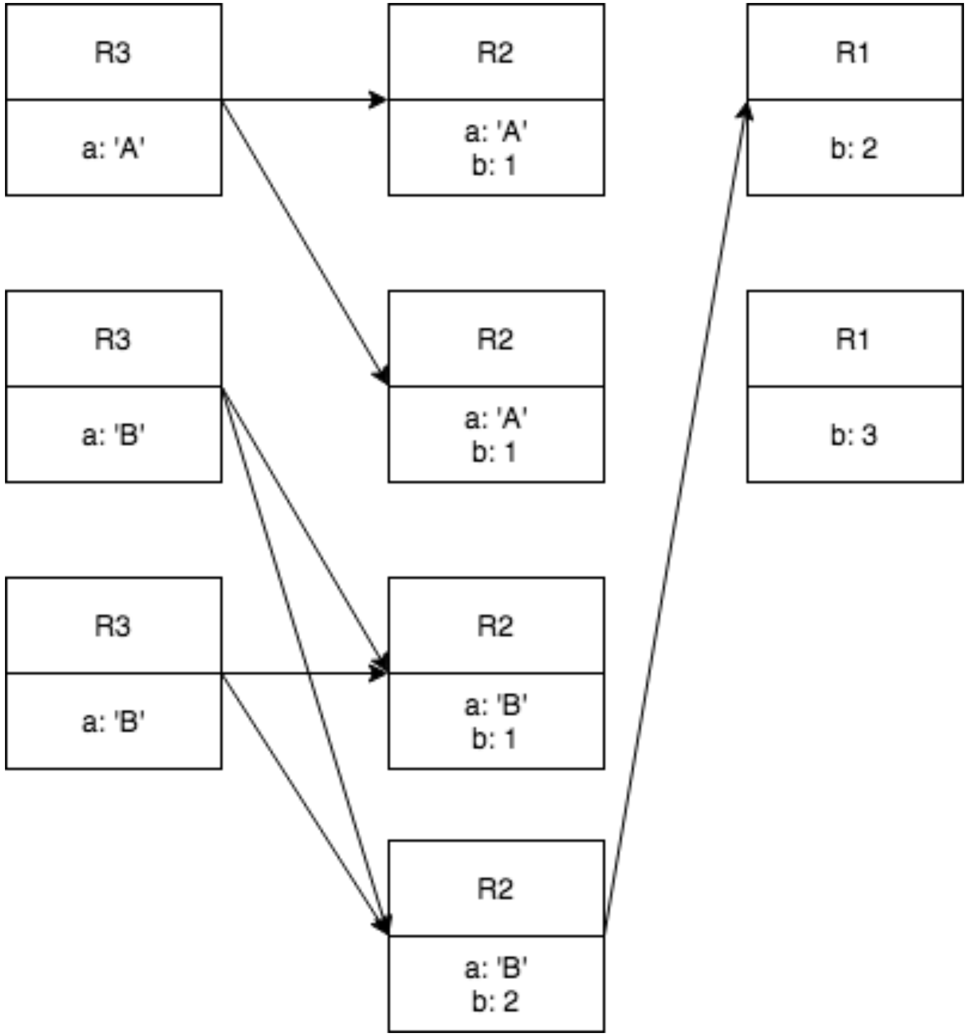
Understand a Cypher Query

- Let’s model the same data in property graph model
- Guidelines to map relational model to graph model
  - A row is a node
  - A table name is a label name
  - Attributes in relational schema become properties associated with nodes
  - A join or foreign key is a relationship (i.e., edge)

Understand a Cypher Query

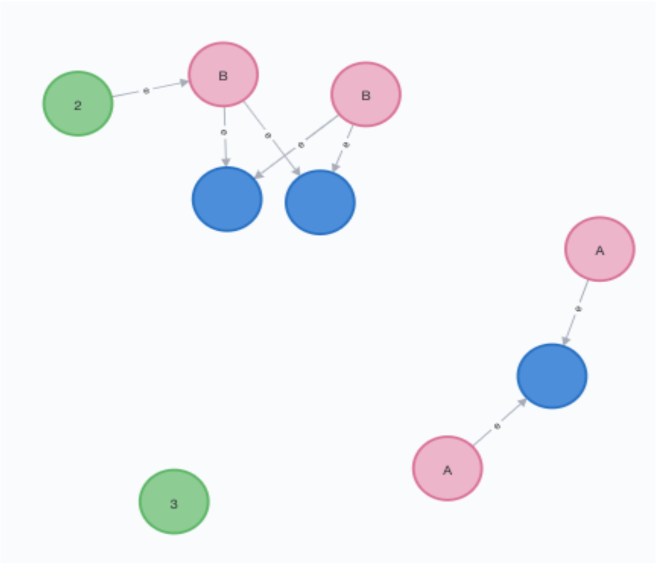
```
CREATE
(n1:R3 {a: "A"}),
(n2:R3 {a: "B"}),
(n3:R3 {a: "B"}),
(n4:R2 {a: "A", b: 1}),
(n5:R2 {a: "A", b: 1}),
(n6:R2 {a: "B", b: 1}),
(n7:R2 {a: "B", b: 2}),
(n8:R1 {b: 2}),
(n9:R1 {b: 3});

MATCH (r1:R1),(r2:R2)
WHERE r1.b = r2.b
CREATE (r1)-[:e]->(r2)

MATCH (r2:R2),(r3:R3)
WHERE r2.a = r3.a
CREATE (r2)-[:e]->(r3)
```

Without them, we have cartesian product (a complete graph)



Understand a Cypher Query

- Resulting graph in Neo4j



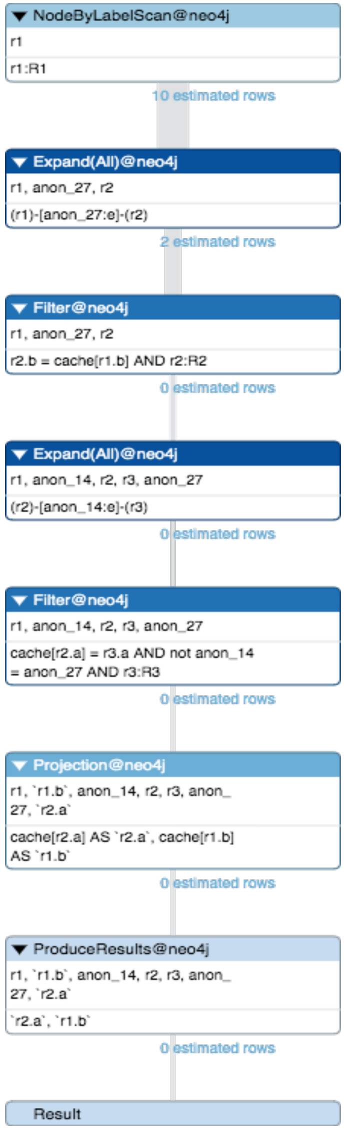
- Observation
  - Neo4j doesn't have notion of undirected edges during graph creation; user can ignore directions when they query the graph
  - Unlike relational model, part of join computation is done during the graph model creation (e.g., when create relationships)

Understand a Cypher Query

- Recall the semantics of semi-join in graph
  - $A \ltimes B$ means given a set of starting vertices in B, return the set of vertices in A that connect to those starting vertices via edges
- A multi-way join query = A fixed-length path query
- Cypher query

```
MATCH (r3:R3)-[:e]-(r2:R2)-[:e]-(r1:R1)
WHERE r2.a = r3.a and r2.b = r1.b
RETURN r2.a, r1.b
```

Understand a Cypher Query

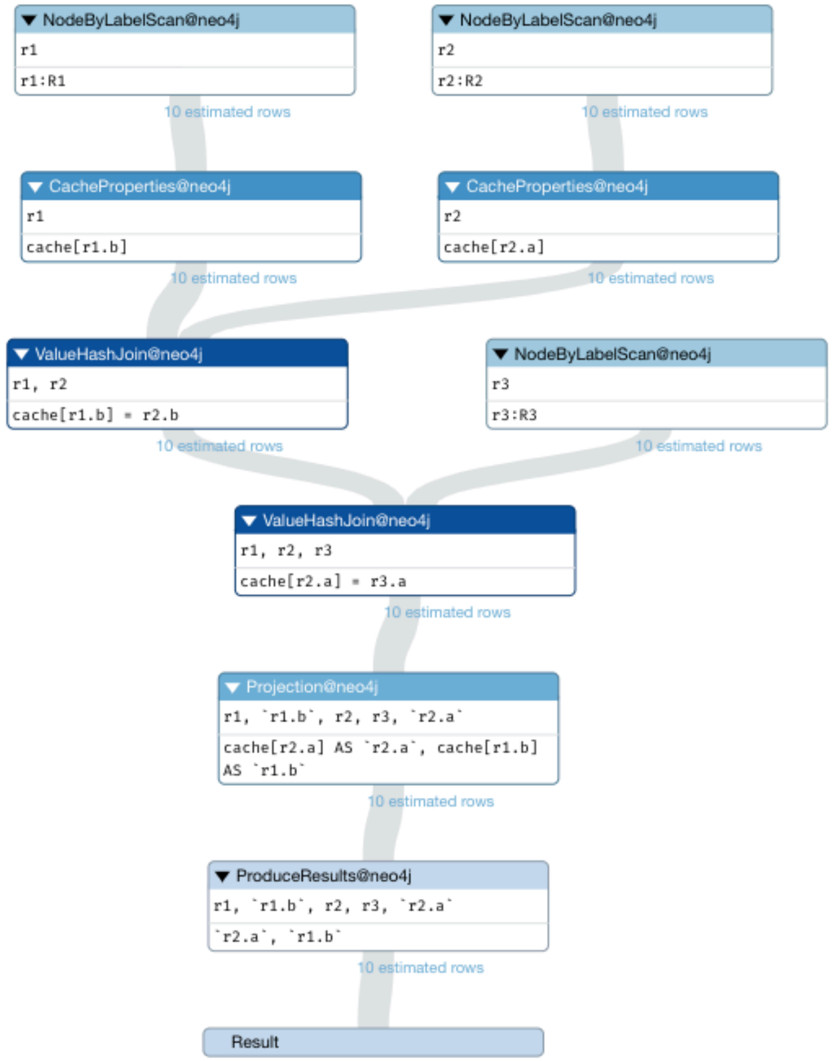


```
▼ NodeByLabelScan@neo4j
r1
r1:R1
10 estimated rows

▼ Expand(All)@neo4j
r1, anon_27, r2
(r1)-[anon_27:e]-(r2)
2 estimated rows

▼ Filter@neo4j
r1, anon_27, r2
r2.b = cache[r1.b] AND r2:R2
0 estimated rows

▼ Expand(All)@neo4j
r1, anon_14, r2, r3, anon_27
(r2)-[anon_14:e]-(r3)
0 estimated rows

▼ Filter@neo4j
r1, anon_14, r2, r3, anon_27
cache[r2.a] = r3.a AND not anon_14
= anon_27 AND r3:R3
0 estimated rows

▼ Projection@neo4j
r1, `r1.b`, anon_14, r2, r3, anon_
27, `r2.a`
cache[r2.a] AS `r2.a`, cache[r1.b]
AS `r1.b`
0 estimated rows

▼ ProduceResults@neo4j
r1, `r1.b`, anon_14, r2, r3, anon_
27, `r2.a`
`r2.a`, `r1.b`
0 estimated rows

Result
```

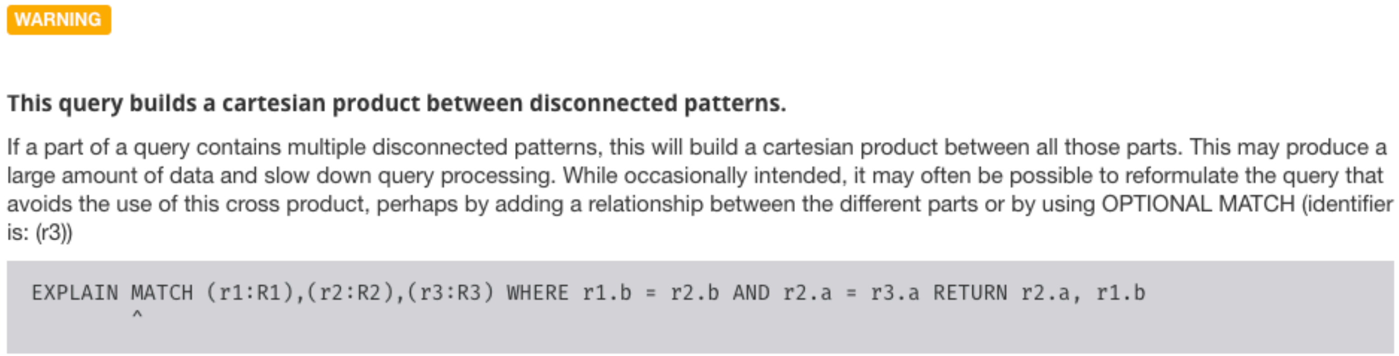
- `Expand(All)`
  - Given a start node, and depending on the pattern relationship, the `Expand(All)` operator will traverse incoming or outgoing relationships.
  - = breadth-wise expansion of the search tree
- Join = Expand(All) + Filter
  - The execution starts with $r_1$and follows the edges from r_1 by one more level to find all r_2 that satisfies `(r1)-[anon_27:e]-(r2)`
  - Then `Filter` is applied with predicate `r2.b = r1.b`.
- Recall: Each `MATCH ... WHERE` can be thought as a `SELECT ... FROM ... WHERE`

```
MATCH (r2:R2)-[:e]-(r3:R3)
WHERE r2.a = r3.a
with r2
MATCH (r2)-[:e]-(r1:R1)
WHERE r2.b = r1.b
RETURN r2.a, r1.b
```

Understand a Cypher Query



```
MATCH (r1:R1),(r2:R2),(r3:R3)
WHERE r1.b = r2.b AND r2.a =
r3.a
RETURN r2.a, r1.b
```



WARNING

This query builds a cartesian product between disconnected patterns.

If a part of a query contains multiple disconnected patterns, this will build a cartesian product between all those parts. This may produce a large amount of data and slow down query processing. While occasionally intended, it may often be possible to reformulate the query that avoids the use of this cross product, perhaps by adding a relationship between the different parts or by using OPTIONAL MATCH (identifier is: (r3))

```
EXPLAIN MATCH (r1:R1),(r2:R2),(r3:R3) WHERE r1.b = r2.b AND r2.a = r3.a RETURN r2.a, r1.b
          ^
```

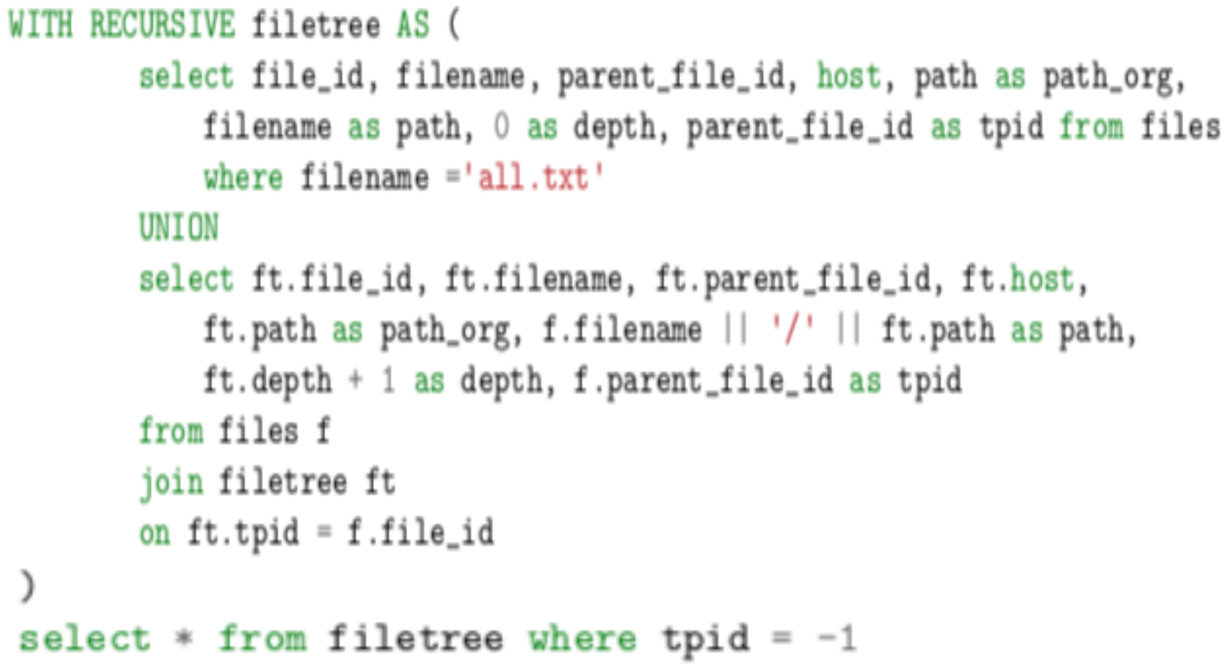
Suggests the above Cypher query is ”anti-pattern”

Relational vs. Graph: A Case Study

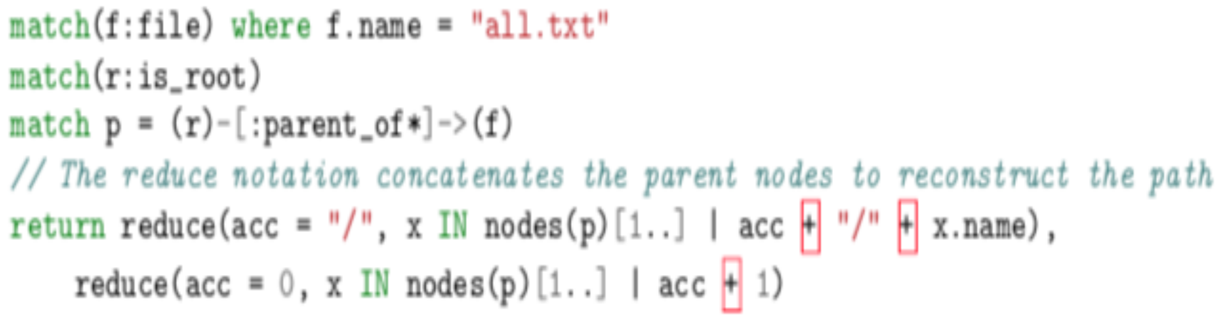
- A term project done by a student in the same class in Spring 2019
- The goal is to enable run queries against a large collection of data (~2.2M rows) gathered from student’s workplace
  - Example query: Where are all my photos?
  - Similar scenarios:
    - Query against files stored in S3 bucket
    - Query against log files collected over time from different services on AWS
- Method: model data in relational model (Postgres 11) and graph model (Neo4j 3.5) and compare performance
- Independent of the DBMS, *the raw data comprised a labeled directed graph*

Relational vs. Graph: A Case Study

- Query 1: Exact Filename Match
  - Return full paths of files that match a given file name



```
WITH RECURSIVE filetree AS (
        select file_id, filename, parent_file_id, host, path as path_org,
            filename as path, 0 as depth, parent_file_id as tpid from files
            where filename ='all.txt'
        UNION
        select ft.file_id, ft.filename, ft.parent_file_id, ft.host,
            ft.path as path_org, f.filename || '/' || ft.path as path,
            ft.depth + 1 as depth, f.parent_file_id as tpid
        from files f
        join filetree ft
        on ft.tpid = f.file_id
 )
 select * from filetree where tpid = -1
```

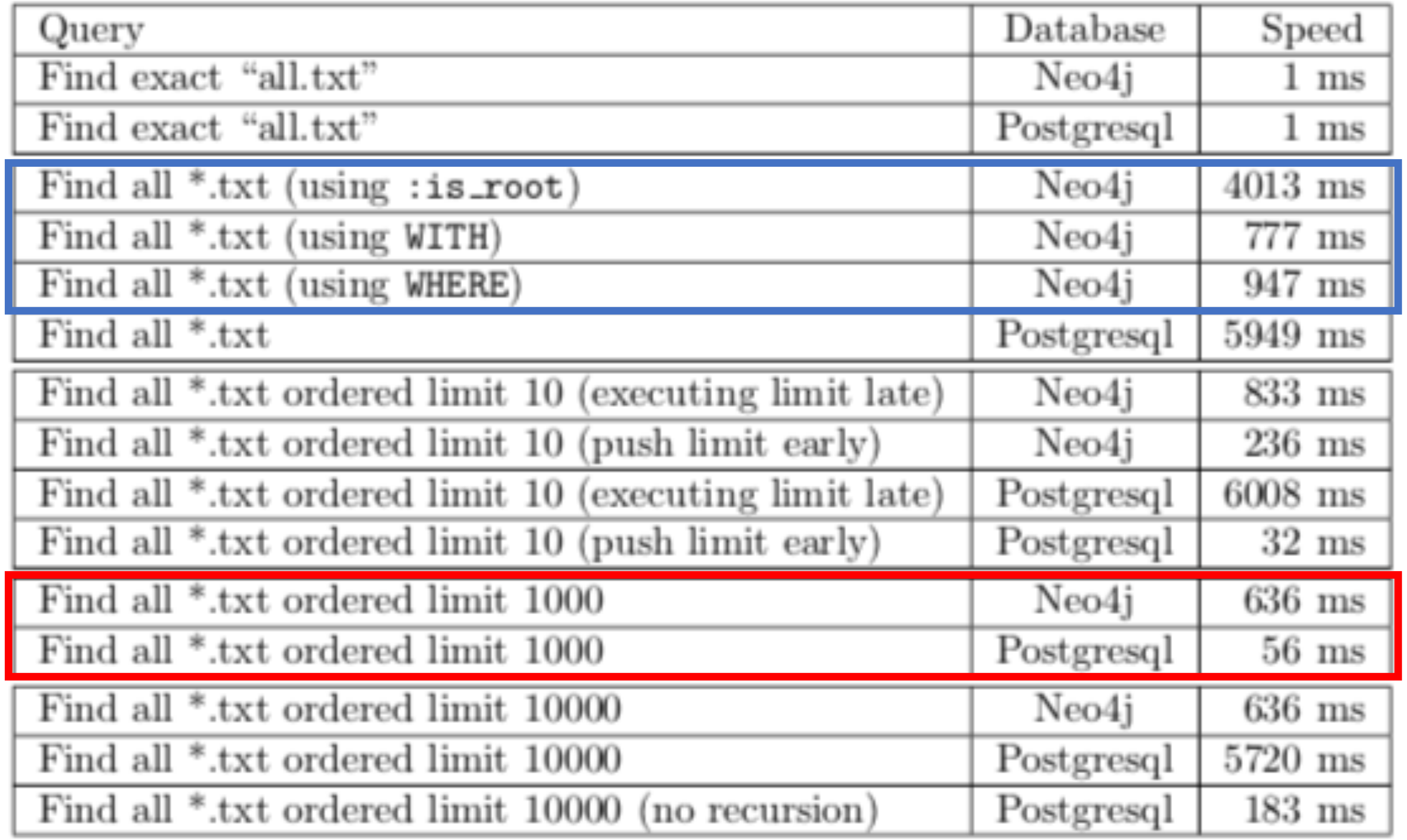


```
match(f:file) where f.name = "all.txt"
match(r:is_root)
match p = (r)-[:parent_of*]->(f)
// The reduce notation concatenates the parent nodes to reconstruct the path
return reduce(acc = "/", x IN nodes(p)[1..] | acc + "/" + x.name),
    reduce(acc = 0, x IN nodes(p)[1..] | acc + 1)
```

Cypher query is much shorter* than SQL counterpart → easier development and code maintenance

*average < 1/3 LoC. Newer, similar studies show even greater reduction (e.g., occasionally < 1/10 LoC)

Relational vs. Graph: A Case Study

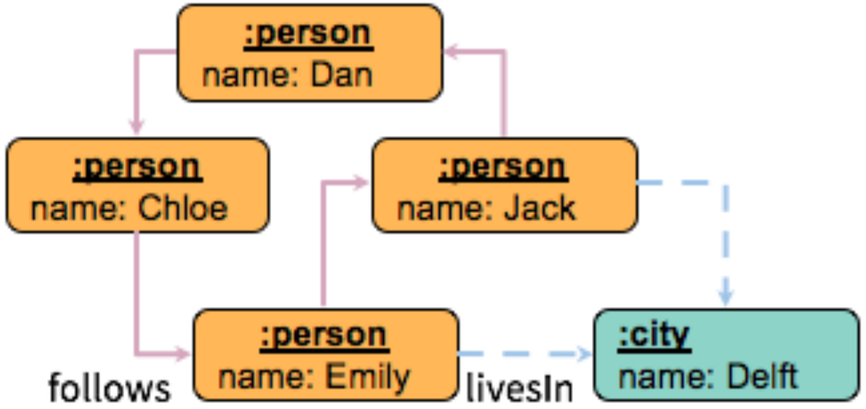


Query	Database	Speed
Find exact "all.txt"	Neo4j	1 ms
Find exact "all.txt"	Postgresql	1 ms
Find all *.txt (using `:is_root`)	Neo4j	4013 ms
Find all *.txt (using `WITH`)	Neo4j	777 ms
Find all *.txt (using `WHERE`)	Neo4j	947 ms
Find all *.txt	Postgresql	5949 ms
Find all *.txt ordered limit 10 (executing limit late)	Neo4j	833 ms
Find all *.txt ordered limit 10 (push limit early)	Neo4j	236 ms
Find all *.txt ordered limit 10 (executing limit late)	Postgresql	6008 ms
Find all *.txt ordered limit 10 (push limit early)	Postgresql	32 ms
Find all *.txt ordered limit 1000	Neo4j	636 ms
Find all *.txt ordered limit 1000	Postgresql	56 ms
Find all *.txt ordered limit 10000	Neo4j	636 ms
Find all *.txt ordered limit 10000	Postgresql	5720 ms
Find all *.txt ordered limit 10000 (no recursion)	Postgresql	183 ms

- Postgres outperforms Neo4j for certain queries
- Cypher query still needs many hand tuning* to reach acceptable performance → indicates lack of maturity for Neo4j optimizer

*writing query in a different way, and/or overriding Neo4j's Cypher Query Optimizer's choice of query plan.

Preview of SQL/PGQ

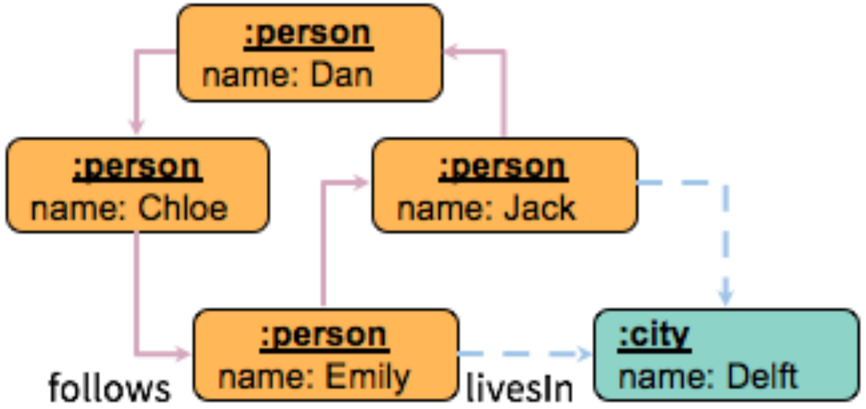


```
CREATE TABLE city (id bigint PRIMARY KEY, name varchar);
CREATE TABLE person (id bigint PRIMARY KEY, name varchar, livesIn bigint,
  CONSTRAINT livesIn_city FOREIGN KEY (livesIn) REFERENCES city (id)
);
CREATE TABLE follows (p1id bigint, p2id bigint,
  CONSTRAINT p1 FOREIGN KEY (p1id) REFERENCES person (id),
  CONSTRAINT p2 FOREIGN KEY (p2id) REFERENCES person (id)
);
```

```
CREATE PROPERTY GRAPH socialNetwork
  VERTEX TABLES (city, person)
  EDGE TABLES (
    follows SOURCE person DESTINATION person,
    person SOURCE person DESTINATION city  LABEL livesIn
);
```

Slides adapted from "Efficient Analytical and Graph Queries " by Peter Boncz (CWI)

Preview of SQL/PGQ



```
SELECT count(gt.id) AS cp2
FROM GRAPH_TABLE (socialNetwork,
  MATCH
   (p1:person WHERE name = 'Dan')-[:follows]->*
   (p2:person)-[:livesIn]->(c:city WHERE name = 'Delft')
  COLUMNS (p2.id)
) gt
```

Slides adapted from "Efficient Analytical and Graph Queries ” by Peter Boncz (CWI)

Conclusion

- Introduced Edge-label Graph, Property Graph
  - Discussed their difference with each other and with Relational Model
- Introduced graph query languages
  - SPARQL for RDF (i.e., Edge-label Graph), Gremlin and Cypher for Property Graph
  - Introduced three important usage patterns in graph query languages
    - Graph Pattern Matching
    - Path Query
    - Navigational Graph Pattern Matching
  - Demonstrated and practiced those usage patterns in Cypher with Neo4j
  - Introduced Cypher query profiling in Neo4j
- A real world case study on relational model vs. graph model
  - Graph query is easier to write and maintain
  - Neo4j has a long way to go to build a sophisticated optimizer like Postgres