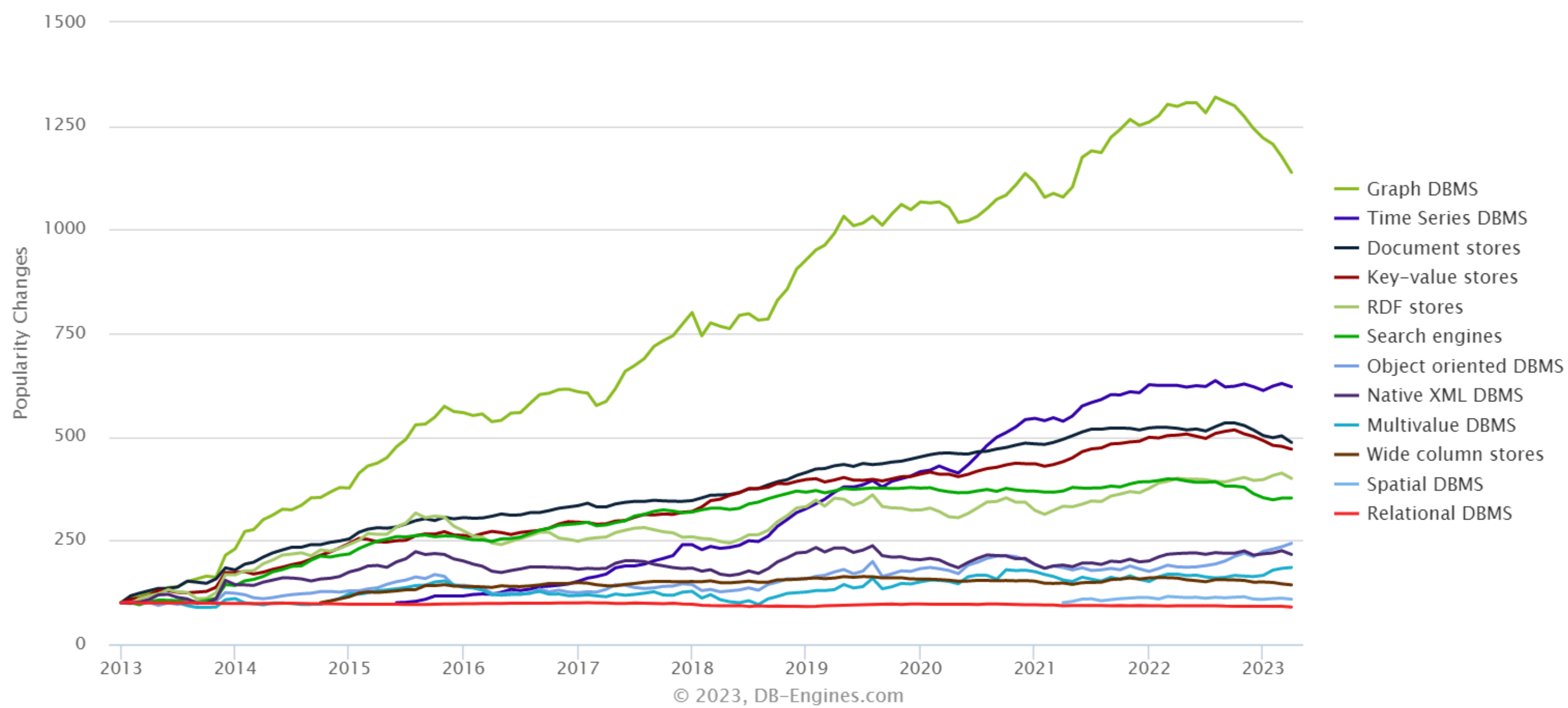
Graph Databases

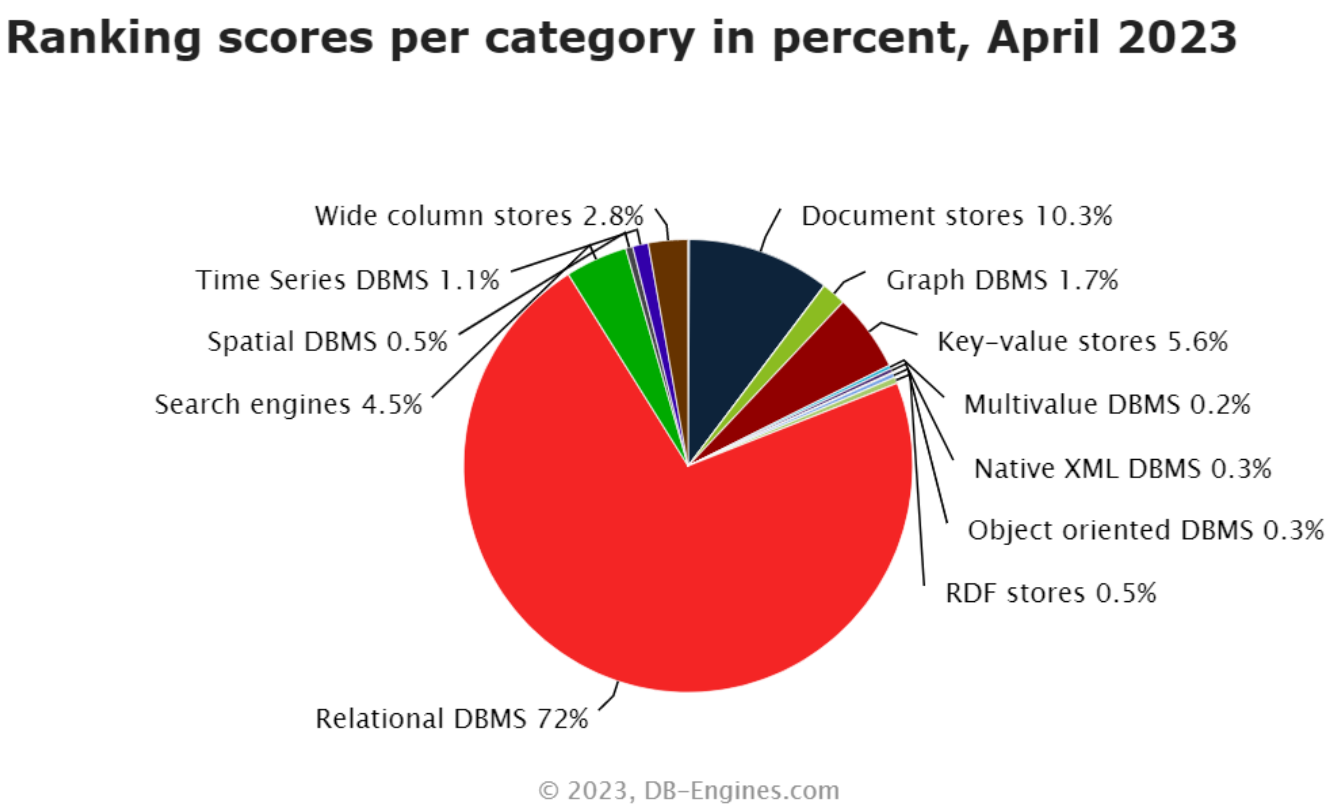
The "Others"

Complete trend, starting with January 2013



Source: https://db-engines.com/en/ranking_categories

The “Others”

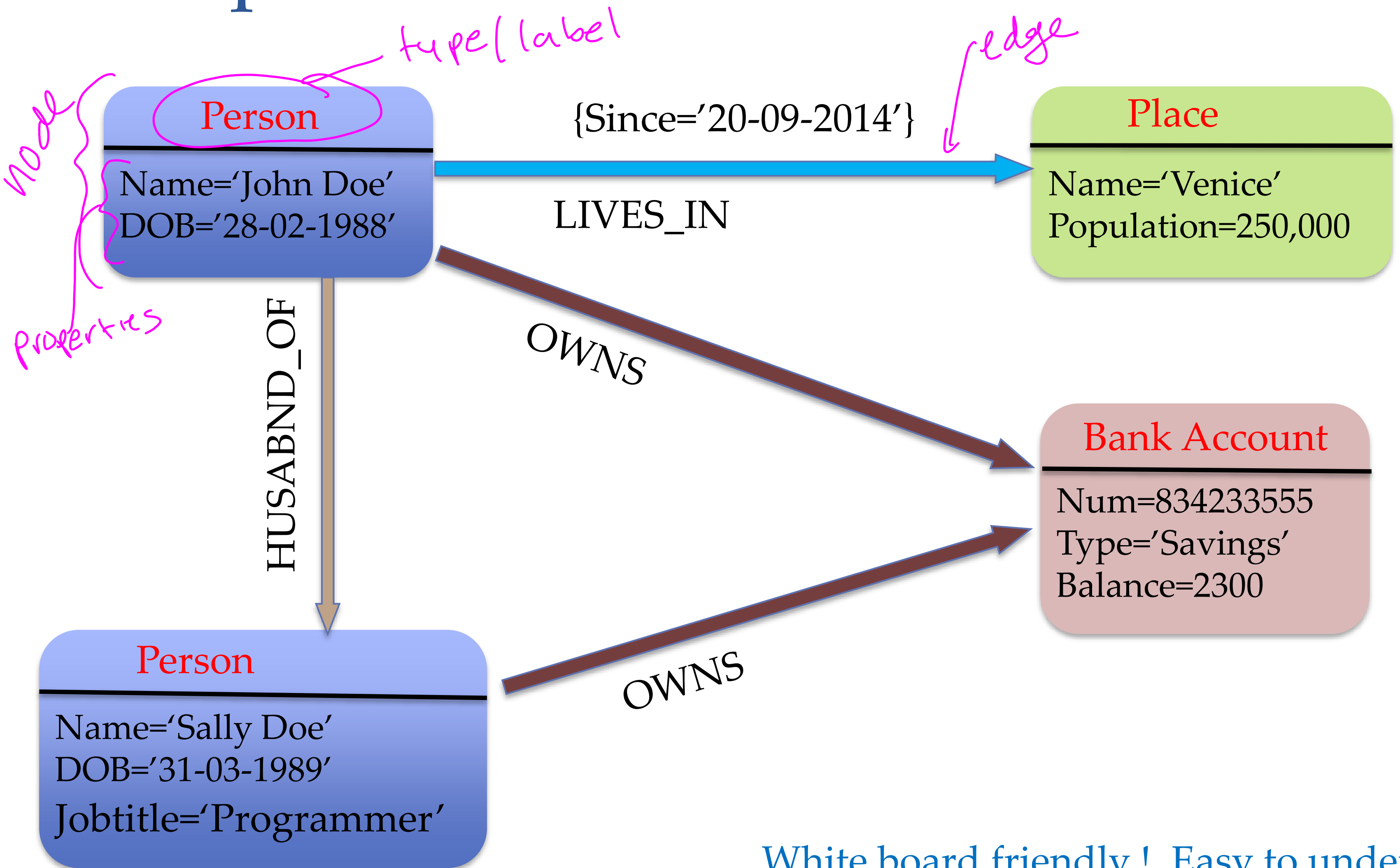


Source: https://db-engines.com/en/ranking_categories

Introduction

- Based on Euler's graph theory
- Data Model
  - Nodes/Vertices of the graph → Represents real-world entities (Eg. a Person (John Doe), a Place (Venice), a Bank account (834233555), etc.)
    - Nodes may be associated with a Label/Type (Eg. Person, Place, etc.)
  - Edges between nodes in the graph → Represents relationships between two entities (Eg. LivesIn, Owns, etc.)
    - Neo4J:- Relationships are directional in nature.
  - Properties are key-value pairs that are associated with either a particular node or a particular relationship. (Eg. A person can have the following properties { name:'John Doe' , dob:'29-02-1988' }
  - Each node/edge are free to have its own set of (possibly different) set of properties for example some nodes representing a person can have the property jobtitle whereas others may not. → concept of a sparse schema.

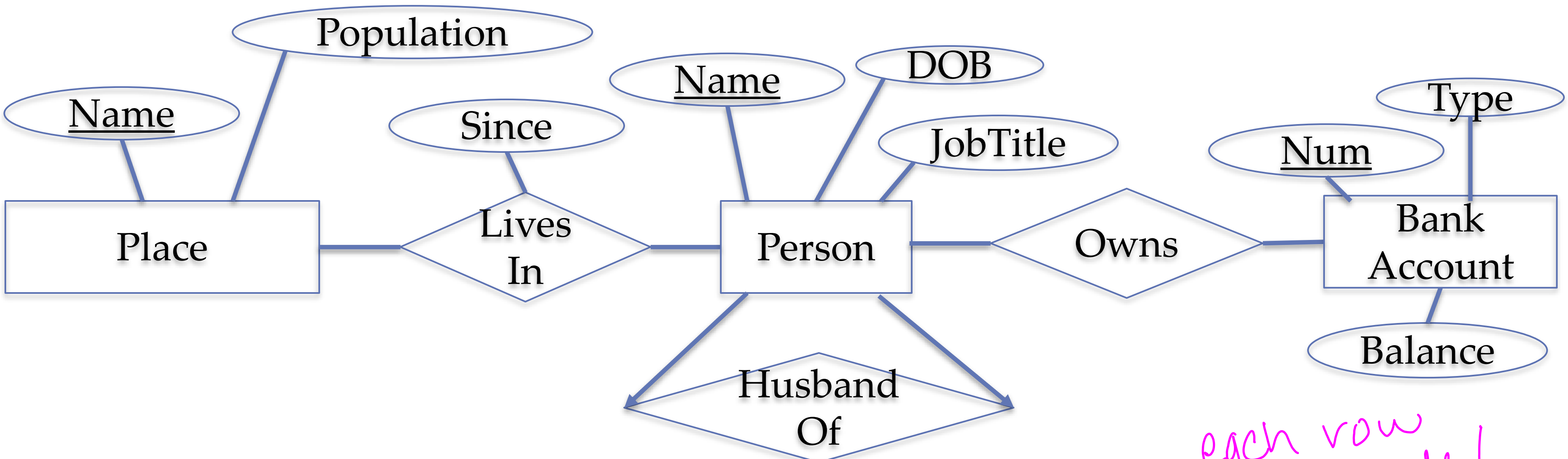
Graph Data Model Instance Example



White board friendly !, Easy to understand.

nodes → entity sets
edges → relationship sets

ER / Relational



each row is a node/edge

Place

Name	Population
Venice	250,000

BankAccount

Num	Type	Balance
834233555	Savings	2300

Person

Name	DOB	jobTitle
John Doe	28-02-1988	NULL
Sally Doe	31-03-1989	Programmer

LivesIn

PName	CityName
John Doe	Venice

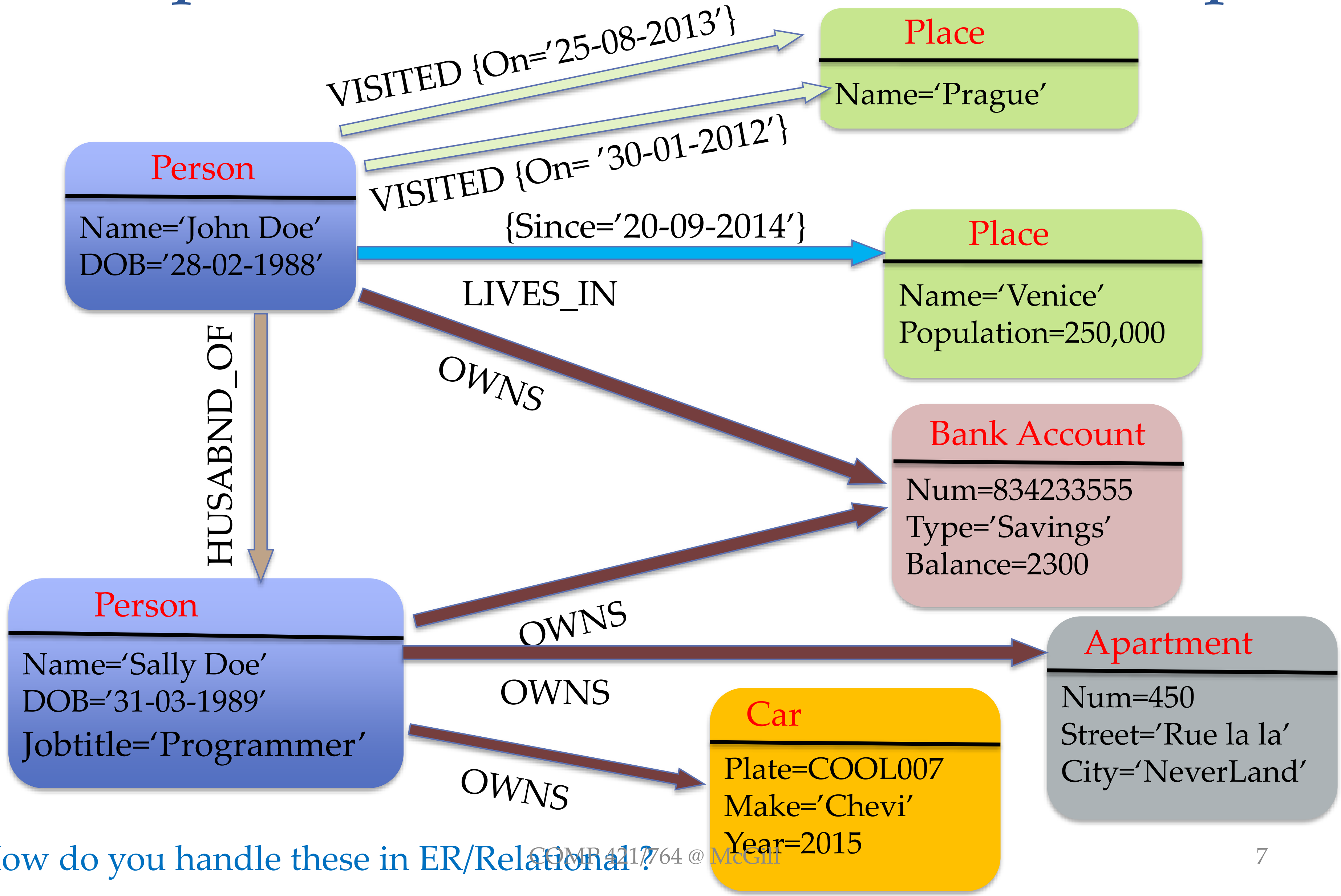
Owns

PName	Num
John Doe	834233555
Sally Doe	834233555

HusbandOf

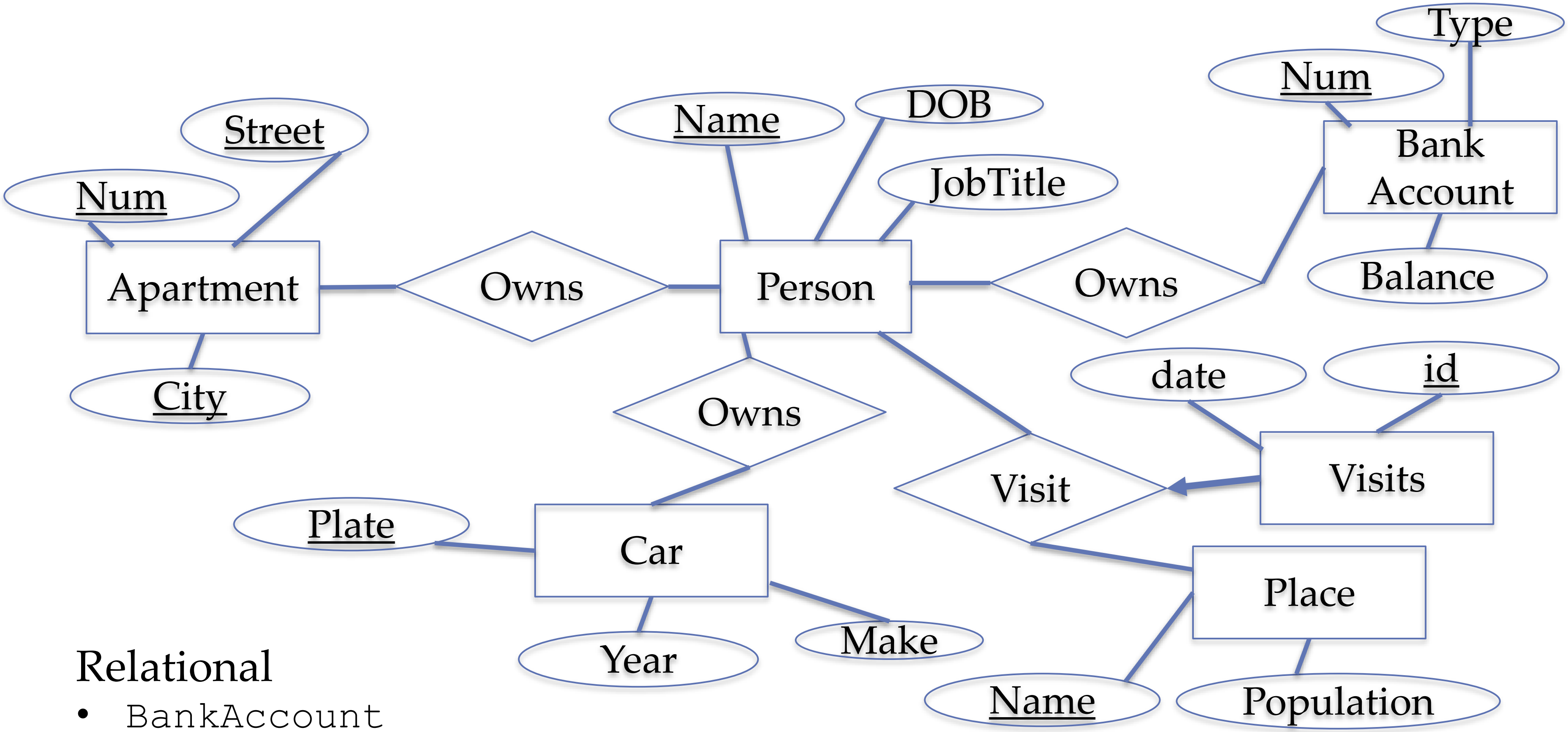
HName	WName
John Doe	Sally Doe

Graph Data Model Instance Example



How do you handle these in ER/Relational?

ER / Relational



Relational

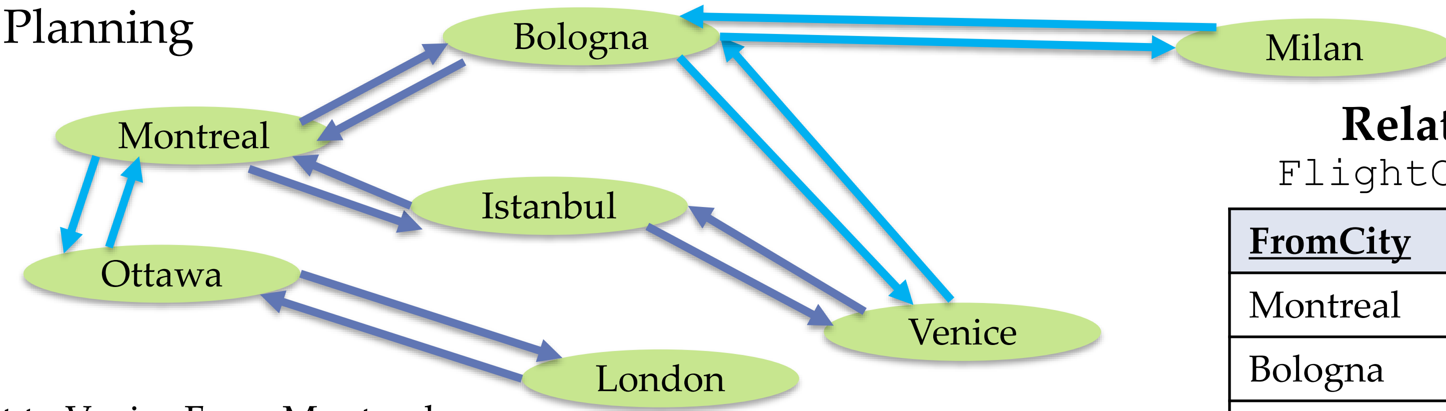
- BankAccount
- Apartment
- Car
- BankAccountOwnerShip
- ApartmentOwnerShip
- CarOwnerShip
- ...

What does a SQL to retrieve all the things that Sally owns look like ?

Might have to write separate SQLs as the relations does not have compatible structures

Working With Varying Depth in Relationships

Trip Planning



Relational

`FlightConnected`

FromCity	ToCity
Montreal	Bologna
Bologna	Montreal
Montreal	Istanbul
Istanbul	Montreal
Istanbul	Venice
….	….

To Get to Venice From Montreal

Direct Flight Connection ?

```
SELECT * FROM FlightConnected
WHERE FromCity = 'Montreal' AND ToCity='Venice'
```

One Stop Flight Connection ?

```
SELECT F1.FromCity, F1.ToCity, F2.ToCity
FROM FlightConnected F1, FlightConnected F2
WHERE F1.ToCity = F2.FromCity
  AND F1.FromCity = 'Montreal' AND F2.ToCity='Venice'
```

`RailConnected`

FromCity	ToCity
Milan	Bologna
Bologna	Milan
Venice	Bologna
Bologna	Venice
….	….

- Two Stop Flights ?
  - One more self join
- Flight & Rail ?
  - Unions + Joins

lots of Joins? ⇒ graph database

Why Graph Databases

- Schema flexibility → new entity types, properties, relationships, etc. can emerge without significant impact on existing data model. I.e, the data model need not be completely developed ahead.
- Graph data model can accommodate the concept of relationship between entities a lot more efficiently than a relational model.

Use Cases

- Social/Professional Networks
- Computer Networks
- Complex Hierarchies
- Geo-Spatial Data (Maps, Flight Reservation, etc.)
- Relationship Between Webpages ? (Web Graph)
- Maintain Knowledge Base
- Maintain IT Infrastructure
- Real-time Recommendation
- Fraud Detection

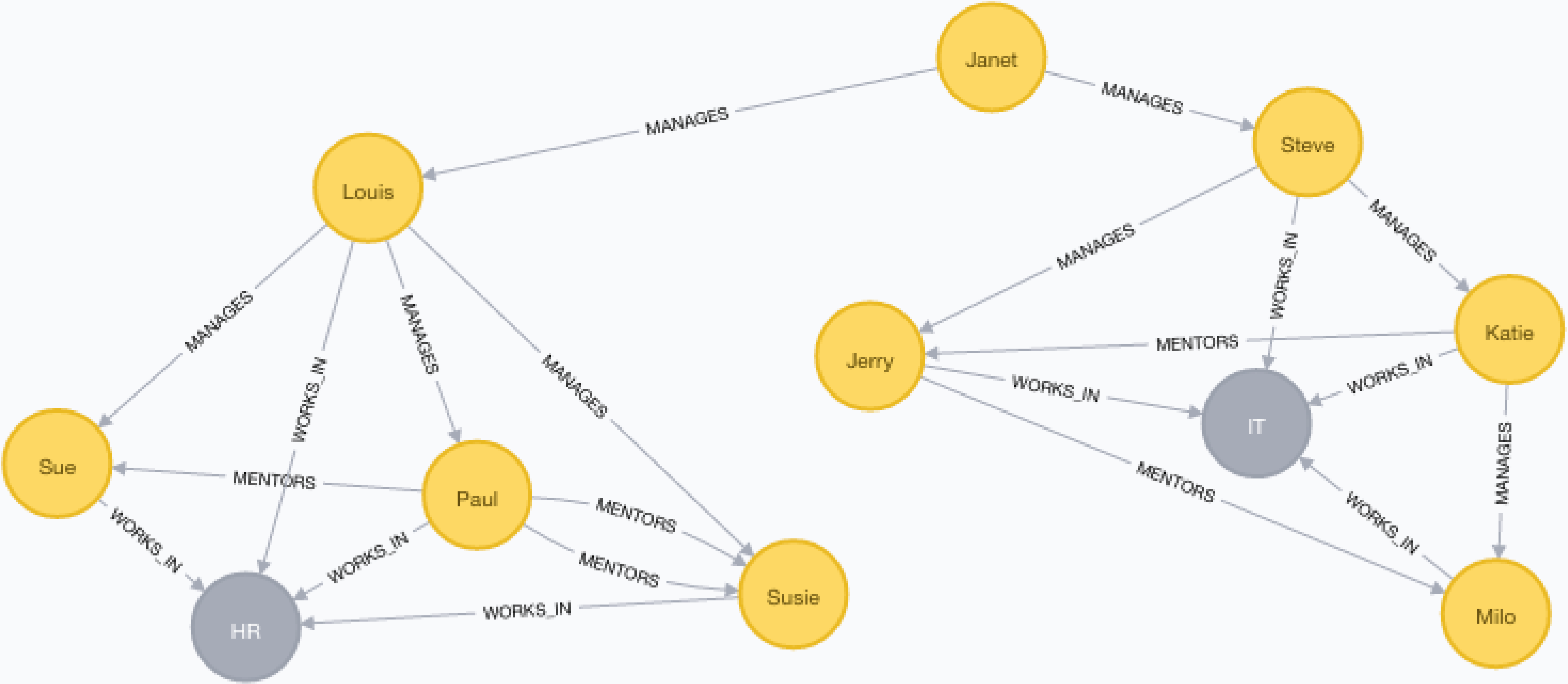
How to interact with the Graph ?

- Custom Programming language APIs
- Query Languages
  - Cypher (Most Prominent and Open)
    - Simple, ASCII art type of queries.
  - Gremlin
  - SPARQL
  - XPath/XQuery (For XML based databases)
- Graph Query Language (GQL) – proposed standard
  - Characteristics from Cypher, SQL, etc.
- Neo4j https://neo4j.com/download [Desktop Version]
  - Programming language APIs in various host languages (Java, Ruby, Python …)
  - Cypher (In this class...)

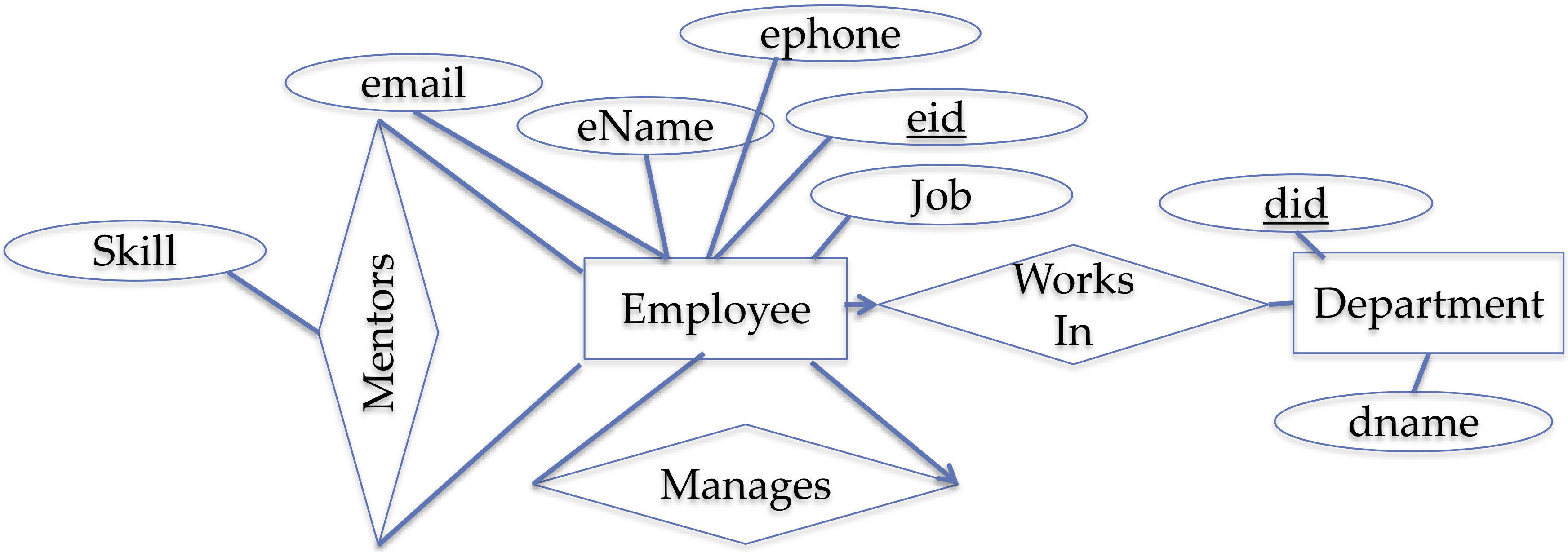
Cypher

- Declarative language for Graph Database Systems, similar to SQL for Relational Database Systems.
  - Specifies what data to retrieve, not how to retrieve them (Similar concept to SQL).
  - Heavily influenced by SQL and SPARQL.
- Data Types
  - Standard data types
    - Integers, Floating point, Strings, Boolean
  - Extended Graph data types
    - Nodes, Relationships, Paths, Maps, Lists

Sample Graph Database



ER / Relational



Relational

```
Department(did, dname)
Employee(eid, ename, ephone, email, job, deptid) deptid is FK to Department(did)
Manages(mgreid, empeid) both mgreid and empeid is FK to Employee(eid)
Mentorship(mentor_eid, mentee_eid) both mentor_eid, mentee_eid FK to employee
```

"SELECT"

SQL

Return All Employee Records
(all columns)

```
SELECT *
FROM Employees
```

Cypher

Return All Employee Nodes
(all properties)

Variable names are required only if the node is being created or its contents needs to be referred elsewhere again. They are case sensitive

type

```
MATCH(e:Employee)
RETURN e
```

match any node of type employee

Returns the entire node

“PROJECTION”

SQL

```
SELECT email, ephone
FROM Employees
```

Cypher

```
MATCH(e:Employee)
RETURN e.email, e.ephone
```

Returns only specific fields

ORDERING Output

```
SELECT email, ephone
FROM Employees
ORDER BY email
```

```
MATCH(e:Employee)
RETURN e.email, e.ephone
ORDER BY e.email
```

"SELECT" with a Condition

SQL	Cypher

Find the Employee info of Janet

```
SELECT *
FROM Employees
WHERE ename = 'Janet'
```

```
MATCH(e:Employee)
WHERE e.ename = 'Janet'
RETURN e
```

or

```
MATCH(e:Employee {ename:'Janet'})
RETURN e
```

Search for multiple employees at the same time

```
SELECT *
FROM Employees
WHERE ename IN ( 'Janet',
'Steve' )
```

```
MATCH(e:Employee)
WHERE e.ename IN [ 'Janet', 'Steve' ]
RETURN e;
```

Pattern matching, names starting with S

```
WHERE ename LIKE 'S%'
```

```
WHERE e.ename =~ 'S.*'
```

"SELECT" with a Condition

SQL | Cypher

Find Employees without a department assigned

```
SELECT *
FROM Employees
WHERE deptid IS NULL
```

```
MATCH(e:Employee)
WHERE NOT (e)-[:WORKS_IN]->()
RETURN e
```

employees with a works_in edge to any other node

negated

e $\xrightarrow{\text{works in}}$ ()

Find Employees who are not mentoring anyone

```
SELECT *
FROM Employees
WHERE eid NOT IN
 (SELECT mentor_eid  FROM Mentorships)
```

```
MATCH(e:Employee)
WHERE NOT (e)-[:MENTORS]->()
RETURN e
```

Find Employees who are not mentored by anyone

```
SELECT *
FROM Employees
WHERE eid NOT IN
 (SELECT mentee_eid
  FROM Mentorships)
```

```
MATCH(e:Employee)
WHERE NOT (e)<-[:MENTORS]-()
RETURN e
```

anyone mentors e

remove direction from edge

How to Find Employees who are not mentoring/mentored by anyone ?

NULL

Cypher

A non existent property in the node is treated as NULL

```
MATCH(e:Employee)
WHERE e.job IS NULL
RETURN e
```

Operators (Not a Complete List)

General	`DISTINCT`
Math	`+, -, *, /, %, ^`
Comparison	`=, <>, <, >, <=, >=, IS NULL, IS NOT NULL`
String comparison	`STARTS WITH, ENDS WITH, CONTAINS`
Boolean	`AND, OR, XOR, NOT`
String operators	`+ (Concatenation), =~ (regex matching)`

Modifying a Graph

- CREATE / DELETE
  – Create / Delete nodes/relationships
- SET/REMOVE
  – Set values to properties Add/Remove labels to nodes and relationships
- MERGE
  – Finding an existing node / create a new node

"INSERT"

SQL

INSERT INTO Department ('PR', 12)

INSERT INTO Employee

(201, 'Jane', '111-333-9999', 12)

Cypher

no restrictions

CREATE (d:Department {dname:"PR", did:12}) <-[WORKS_IN]-
(e:Employee {ename:"Jane", eid:201, ephone:"111-333-9999"})

OR

CREATE (e:Employee {ename:"Jane", eid:201, ephone:"111-333-9999"})
-[WORKS_IN]-> (d:Department {dname:"PR", did:12})

INSERT

Followed by

SELECT

CREATE (e:Employee {ename:"Jane", eid:201, ephone:"111-333-9999"})
-[r:WORKS_IN]-> (d:Department {dname:"PR", did:12})
RETURN e,d,r

In Cypher any Query can return data

"INSERT"

Cypher

Add a new relationship between two existing nodes.

```
MATCH (n1:Employee {eid:101}), (n2:Employee {eid:201})
CREATE (n1) -[:MANAGES]-> (n2);
```

Inserts can have multiple nodes of same or different types (no restrictions)

```
CREATE
 (n1:Employee {ename:'Janet', eid:101, email:'ja@comp.com', ephone:'123-456-1111', job:'CEO'})
,(n2:Employee {ename:'Steve', eid:102, email:'st@comp.com', ephone:'123-456-1112', job:'VP IT'})
,(n3:Employee {ename:'Louis', eid:103, email:'lo@comp.com', ephone:'123-456-1113', job:'VP HR'})
, (d1:Department {dname:'IT', did:10})
,(d2:Department {dname:'HR', did:11});
```

DELETE

SQL | **Cypher**

Delete the records from the referencing table first, followed by the records from the referenced table.

```
DELETE FROM Employee WHERE empid = 201;
DELETE FROM Department WHERE deptid = 12;
```

Delete the relationships (edges) interacting with that node before/while deleting the nodes themselves.

```
MATCH(e:Employee{empid:201})-[r:WORKS_IN]->(d:Department{deptid:12})
DELETE e,d,r
```

Delete Only the Employee

```
DELETE FROM Employee WHERE empid = 201;
```

Deletes any edges as well as the Node

```
MATCH(e:Employee{empid:201})
DETACH DELETE e
```

Could be a better way of deleting a node, as we do not have to know about all the edges that involves this node

SQL “JOINS” / Traversals Cypher

Find Employees working in HR department.

```
SELECT e.*
FROM Employees e, Dept d
WHERE e.deptid = d.deptid
    AND dname = 'HR'
```

```
MATCH(e:Employee) -[w:WORKS_IN]->
(d:Department{dname:"HR"})
RETURN e
```

```
SELECT e.*
FROM Employees e
    INNER JOIN Dept d
    ON e.deptid = d.deptid
WHERE  dname = 'HR'
```

```
MATCH(e:Employee) -[w:WORKS_IN]->
(d:Department)
WHERE d.dname = "HR"
RETURN e
```

Traversals in Various Forms

-[]- } any edge
any type, any direction

Cypher

Returns information on Paul and all the nodes he has relationships with.

```
MATCH(e:Employee) -- (n) WHERE e.ename = 'Paul'  RETURN e,n

MATCH(e:Employee) -[]- (n) WHERE e.ename = 'Paul'  RETURN e,n
```

Returns information on Paul and all the outgoing relationships he has along with the related nodes

any type

```
MATCH(e:Employee) -[r]-> (n) WHERE e.ename = 'Paul'  RETURN e,r,n
```

need variable name to return

Returns information names of all the employees that Steve is managing.

```
MATCH(e:Employee) -[:MANAGES]-> (n:Employee)
WHERE e.ename = 'Steve'
RETURN n.ename
```

Traversals in Various Forms

Cypher

Returns information of people who works for or is mentored by Katie

```
MATCH(e:Employee) -[:MANAGES|MENTORS]-> (n)
WHERE e.ename = 'Katie'
RETURN n
```

or

Returns information of people who neither works for nor is mentored by Katie

```
MATCH(e:Employee), (n:Employee)
WHERE e.ename = 'Katie' AND NOT (e) -[:MANAGES|MENTORS]-> (n)
RETURN n;
```

not + or = nor

→ NOT goes in where clause only

Traversals in Various Forms

Cypher

Returns information of people who is reporting to managers who themselves report to Steve

anyone

```
MATCH(e:Employee) -[:MANAGES]->() -[:MANAGES]->(n)
WHERE e.ename = 'Steve'
RETURN n
```

=

```
MATCH(e:Employee) -[:MANAGES*2]->(n)
WHERE e.ename = 'Steve'
RETURN n
```

Traversals in Various Forms

Cypher

```
(e)-[*]->(n)          // All the way (outgoing edges)

(e)-[*..5]->(n)       // Up to a depth of 5 edges (outgoing)

(e)-[*3..]->(n)       // 3 or more edges (outgoing)

(e)-[*3..5]->(n)      // 3 to 5 edges (outgoing)

(e)<-[*3..5]-(n)      // 3 to 5 edges (incoming)

(e)-[*3..5]-(n)       // 3 to 5 edges (incoming or outgoing)
```

Traversals in Various Forms

Cypher

Employees under Steve who are not managers.

```
MATCH(e:Employee) -[:MANAGES]-> (e2:Employee)
WHERE e.ename = 'Steve' AND NOT (e2)-[:MANAGES]->()
RETURN e2;
```

Steve manages someone and that someone does not manage anyone

Employees who are mentored by a direct report of Steve.

```
MATCH(e:Employee) -[:MANAGES]->() -[:MENTORS]->(n)
WHERE e.ename = 'Steve'
RETURN n
```

person managed by steve

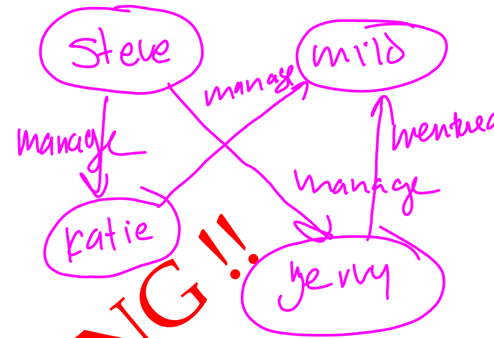
Traversals in Various Forms

Cypher

Employees who are NOT mentored by a direct report of Steve.

```
MATCH(e:Employee) -[:MANAGES]->(dr), (n:Employee)
WHERE e.ename = 'Steve' AND NOT (dr)-[:MENTORS]->(n)
RETURN n;
```

WRONG !!



would return milo even though jerry mentors him

Correct approaches

```
MATCH(n:Employee)
WHERE NOT (:Employee{ename:'Steve'})-[:MANAGES]->()-[:MENTORS]->(n)
RETURN n;
```

```
MATCH(e:Employee), (n:Employee)
WHERE e.ename = 'Steve' AND NOT (e)-[:MANAGES]->()-[:MENTORS]->(n)
RETURN n;
```

Traversals in Various Forms

Cypher

Important !!

For a given path output of a pattern, each edge is traversed only once !

For example if (Janet)-[:FRIEND_OF]->(Sue) and (Sue)-[:FRIEND_OF]->(Janet)

MATCH (e1:Employee(ename:'Janet')-[:FRIEND_OF*]->(e2:Employee)
RETURN e1,e2

Will return only two paths.

(Janet)->(Sue)
(Janet)->(Sue)->(Janet) — back to the beginning → stops

other friends of sue wont be returned

Traversals in Various Forms

Cypher

How to find if Janet is Managing an employee, who manages an employee who is friends with Janet ?

```
MATCH
(n:Employee{ename:'Janet'})-[:MANAGES]->(e1:Employee)-[:MANAGES]->(e2:Employee)
, (n)-[:FRIEND_OF]->(e2)
RETURN n,e1, e2
```

How to find a list of people who manages someone who mentors more than one employee ?

```
MATCH (b:Employee)-[:MANAGES]->(m:Employee)
        ,(m)-[:MENTORS]->(e1:Employee)
        , (m)-[:MENTORS]->(e2:Employee)
WHERE e1 <> e2
RETURN DISTINCT b
```

<> not equal

Other Aspects

- Constraints
  - CREATE CONSTRAINT ON (e: Employee) ASSERT e.eid IS UNIQUE
  - CREATE CONSTRAINT ON (e: Employee) ASSERT exists(e.ename) like not null
  - CREATE CONSTRAINT ON ()-[m:MENTORS]-() ASSERT exists(m.skill)

Other Aspects

- Transactions
- Dynamic property matching
- Multiple labels on the same node (not for relationships).
- Aggregation
- WITH clause
- Multiple MATCH clauses in a single statement

Other Resources

- Download Neo4j (Desktop)
  - https://neo4j.com/download
- Cypher Query Language
  - http://neo4j.com/docs/developer-manual/current/cypher/
- Neo4j webinar videos (If you get addicted to graph databases)
  - https://www.youtube.com/channel/UCvze3hU6OZBkB1vkhH2lH9Q