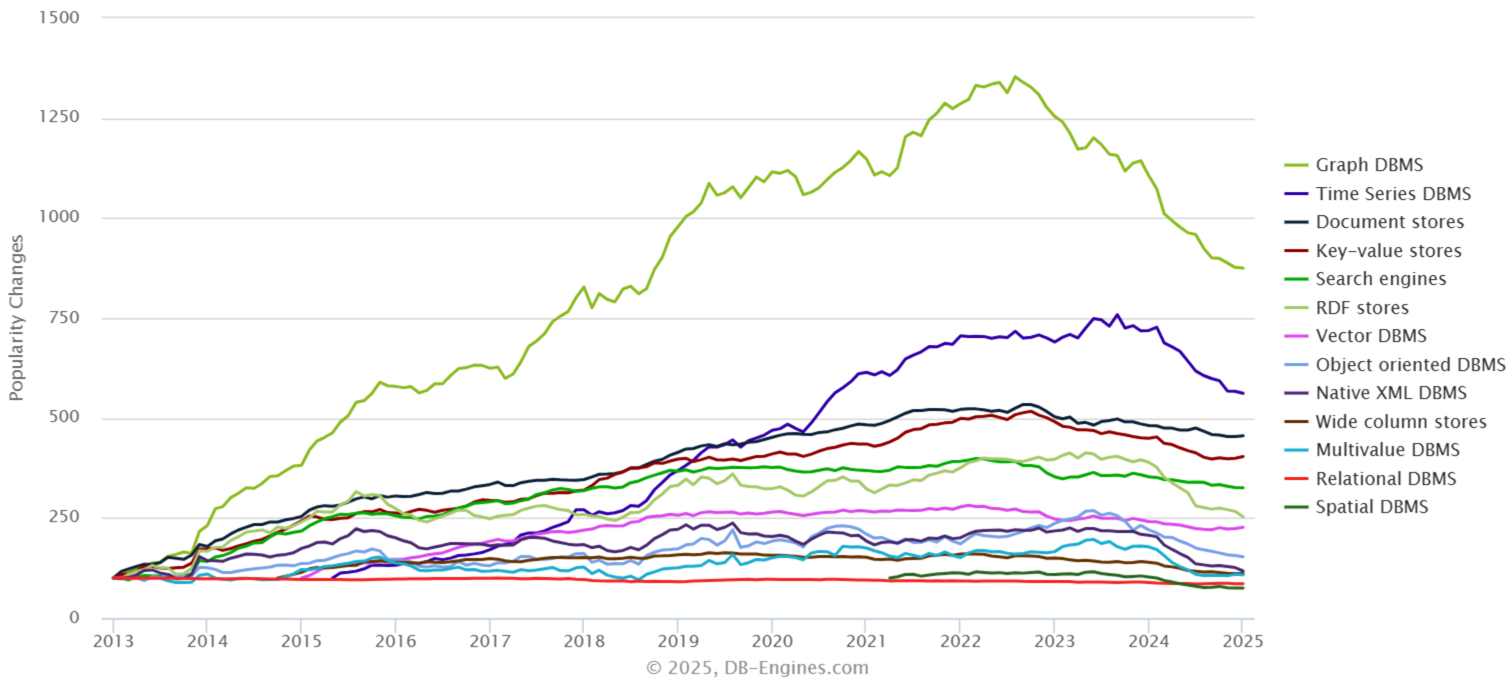
Graph Databases

The "Others"

Popularity changes per category, January 2025

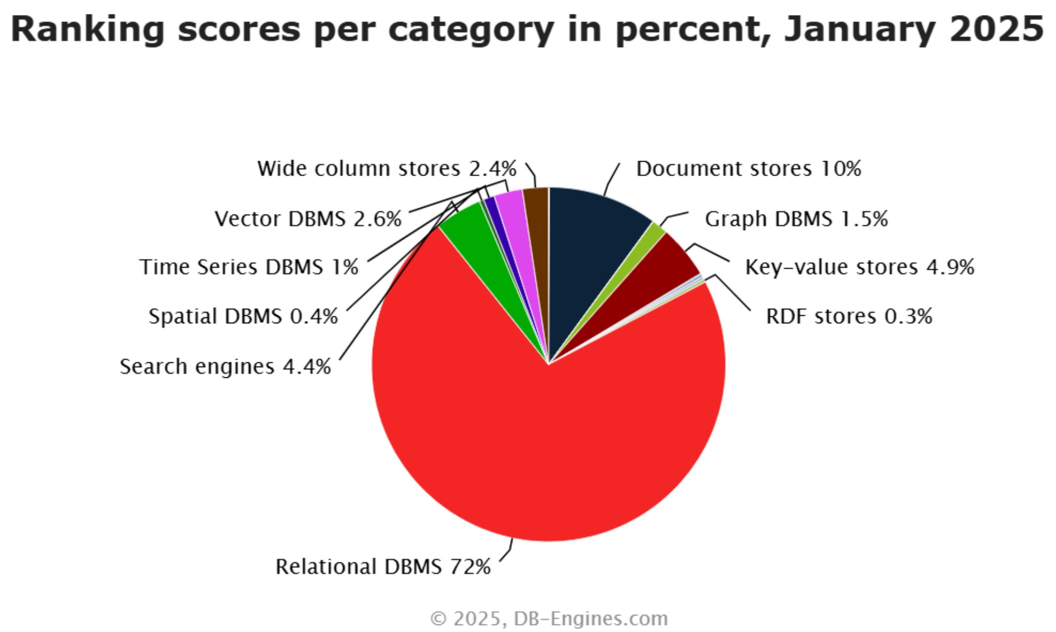
The following charts show the historical trend of the categories' popularity. In the ranking of each month the best six systems per category are chosen and the average of their ranking scores is calculated. In order to allow comparisons, the initial value is normalized to 100.

Complete trend, starting with January 2013



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

The “Others”

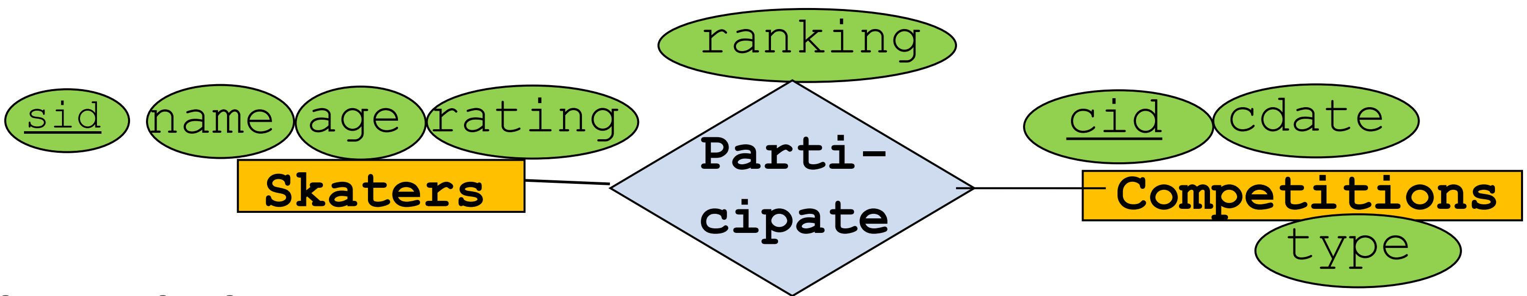


This chart shows the popularity of each category. It is calculated with the popularity (i.e. the [ranking scores](https://db-engines.com/en/ranking_definition)) of all individual systems per category. The sum of all ranking scores is 100%.

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

Recall: E/R and relational model

- ***An E/R schema*** :
  - A set of entity sets / relationship sets that represents an application domain using the E/R model



- ***A relational schema***
  - The description of a set of tables with their attributes that represent an application domain:
    - Comp(<u>cid</u>, cdate,type), Skaters(<u>sid</u>,name,rating,age), Partic(<u>sid,cid</u>,ranking)

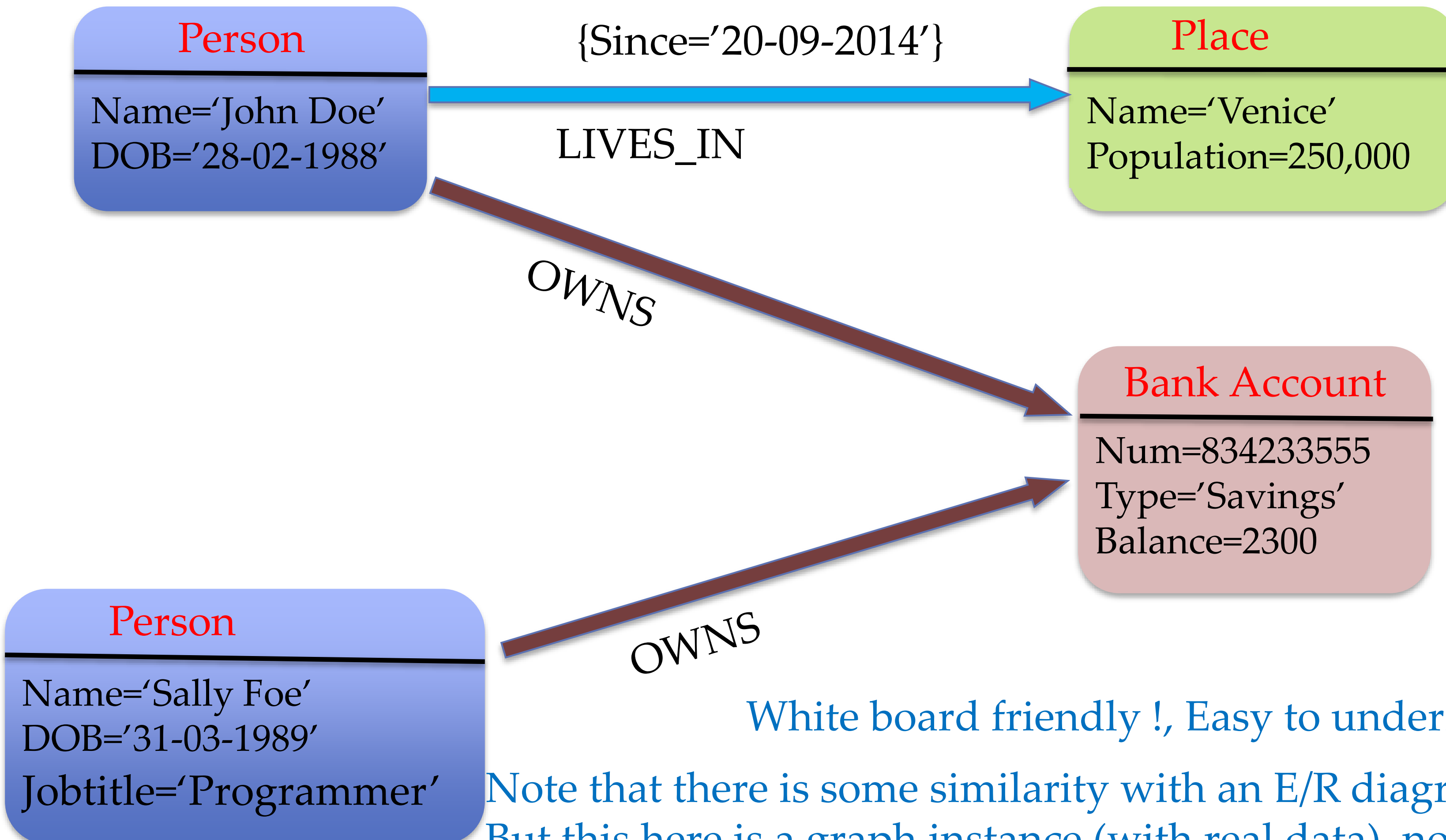
- ***An instance of a relational schema*** is the set of tables with concrete tuples

<u>sid</u>	sname	rating	age
28	yuppy	7	15
31	debby	7	10

Property Graph Model

- 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.

Property Graph *Instance* Example



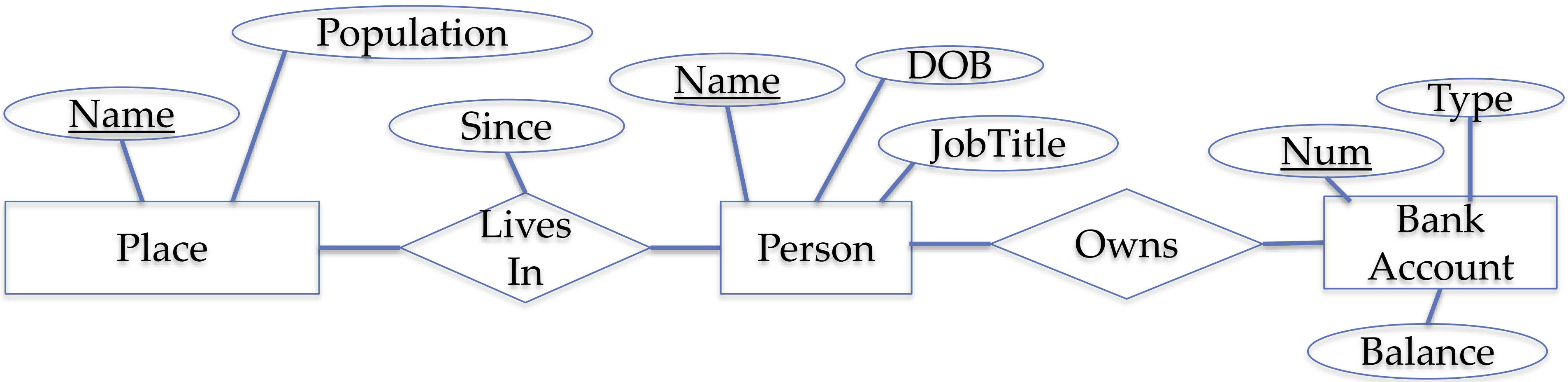
White board friendly !, Easy to understand.

Note that there is some similarity with an E/R diagram
But this here is a graph instance (with real data), not a schema

Property Graph Schema?

- Very few Graph DBMS support the concept of a schema
- How could it look like?
  - E.g., indicate that there is a certain set of node types and edge types
    - (e.g., only employee and department node types, and works-in edge types)
  - Indicate the property names that are possible for a certain node/edge type
    - (e.g., employees can only have name, dob, and jobtitle properties)
  - Indicate for a specific type of edge, what types of the nodes must be that this edge connects the end nodes must have
    - (e.g., that works-in edge are always directed edges from nodes of type employee to nodes of type department)
  - Note the similarity with a E/R schema – but not the same

ER / Relational



Place

<u>Name</u>	Population
Venice	250,000

Person

<u>Name</u>	DOB	jobTitle
John Doe	28-02-1988	NULL
Sally Foe	31-03-1989	Programmer

BankAccount

<u>Num</u>	Type	Balance
834233555	Savings	2300

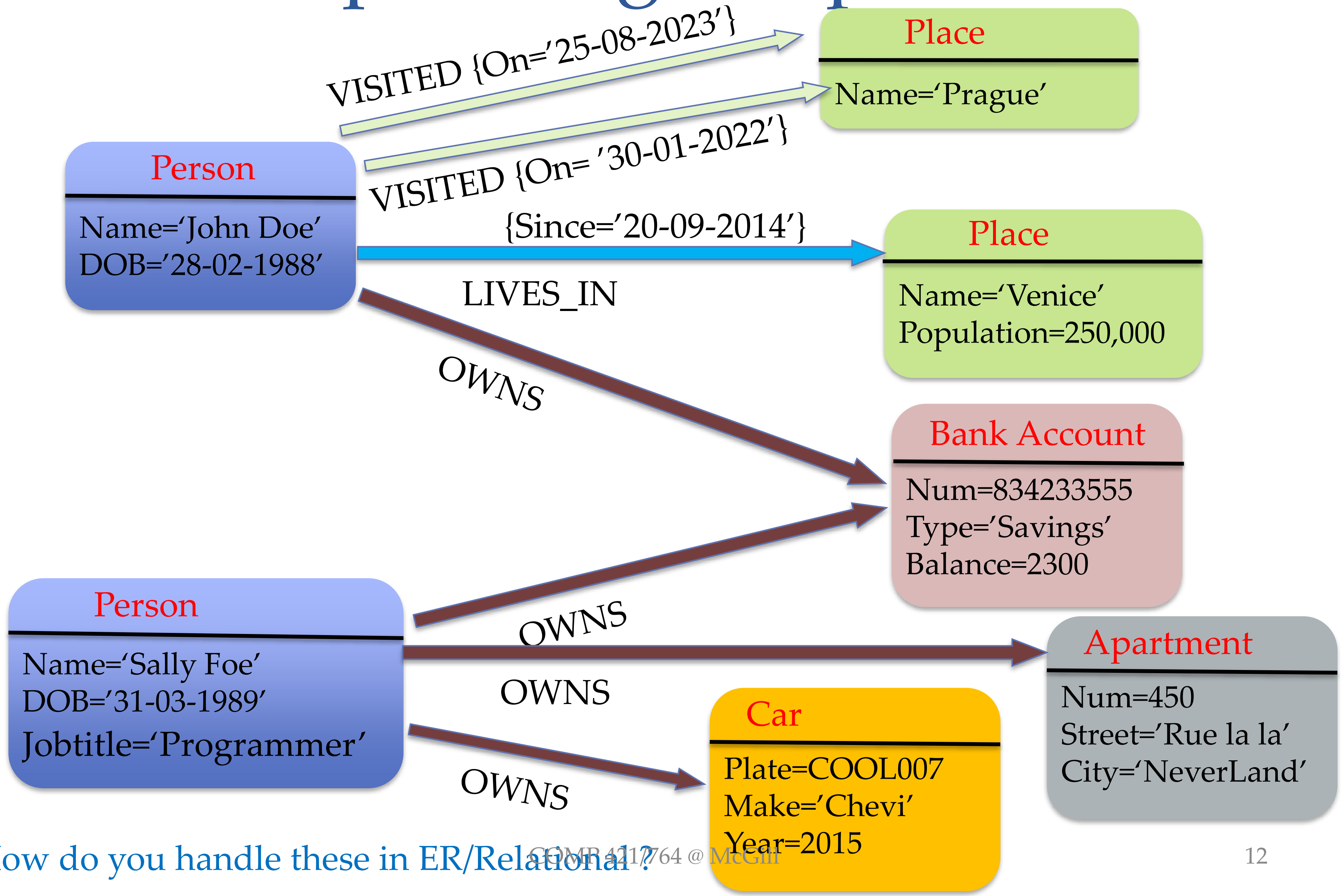
LivesIn

<u>PName</u>	<u>CityName</u>
John Doe	Venice

Owns

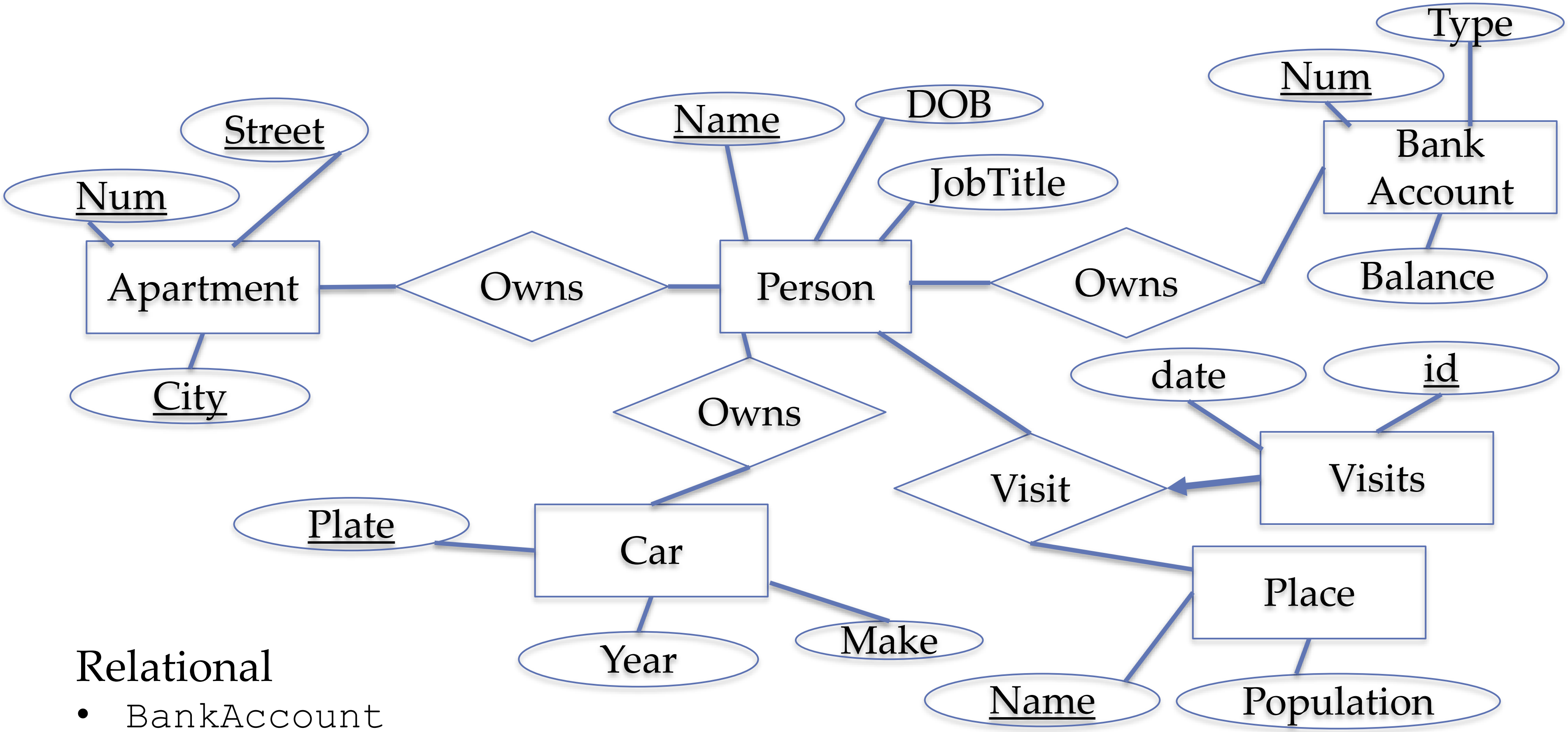
<u>PName</u>	<u>Num</u>
John Doe	834233555
Sally Foe	834233555

Corresponding Graph Instance



How do you handle these in ER/Relational?

ER / Relational



Relational

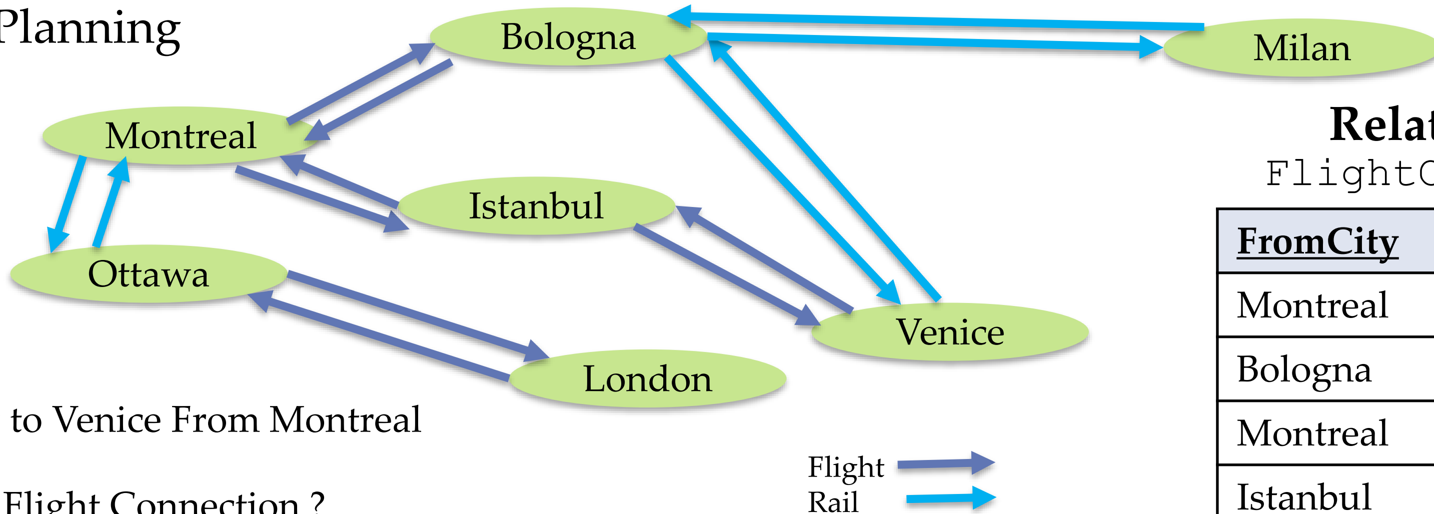
- `BankAccount`
- `Apartment`
- `Car`
- `BankAccountOwnerShip`
- `ApartmentOwnerShip`
- `CarOwnerShip`
- `...`

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



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'
```

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

Relational

FlightConnected

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

RailConnected

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

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

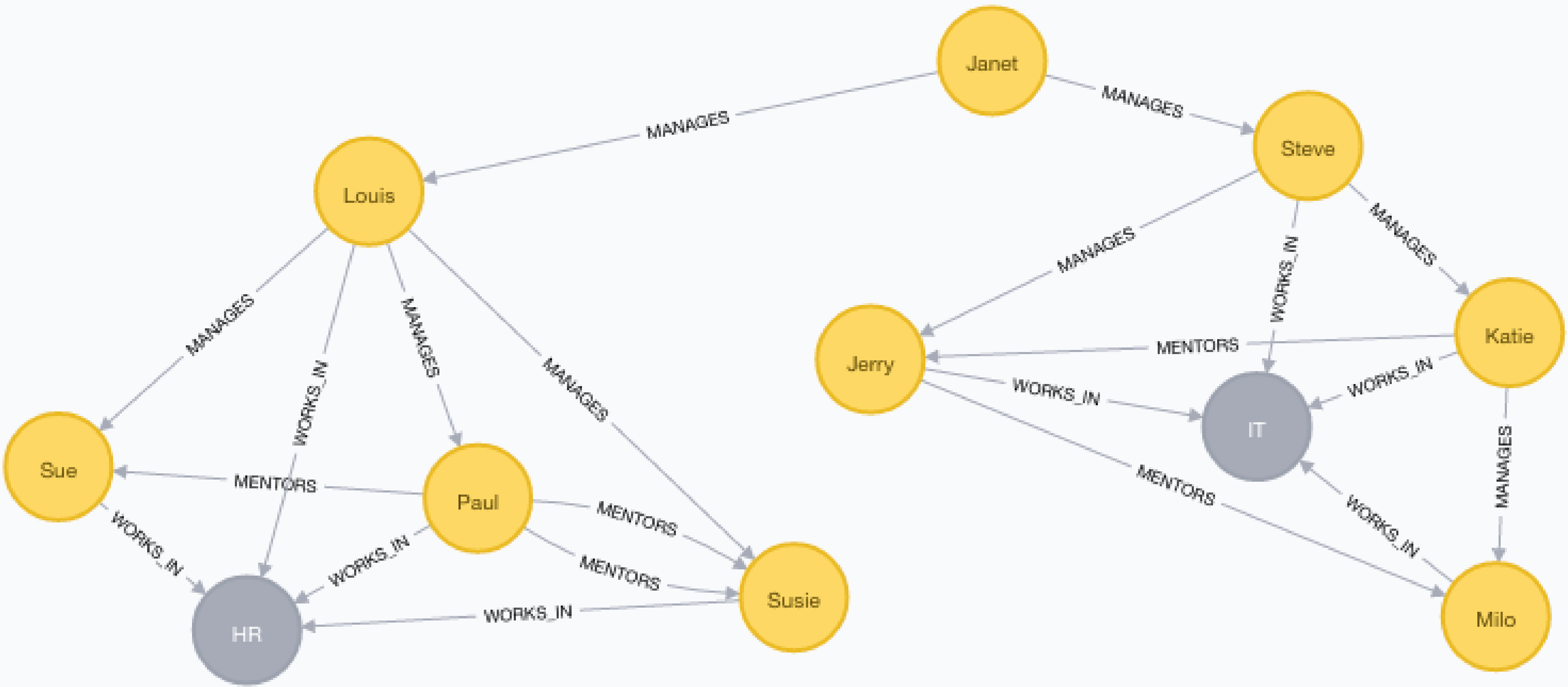
How to interact with the Graph ?

- 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...)

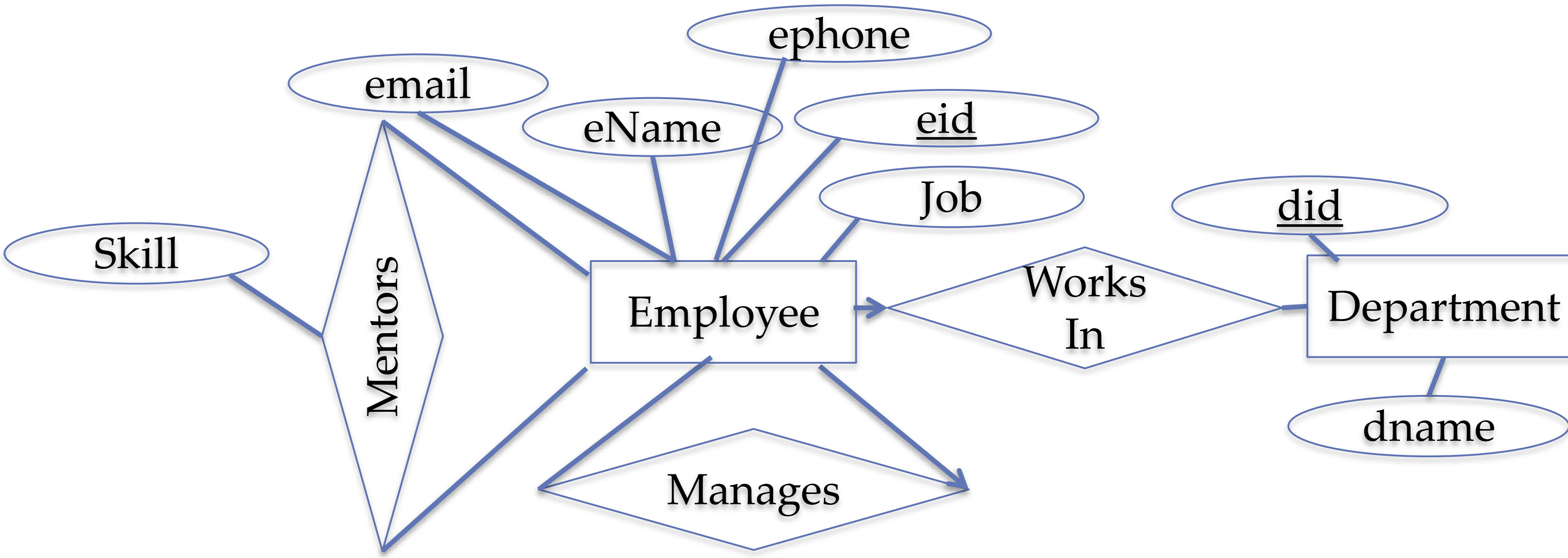
Cypher

- 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

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

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

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
```

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
```

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)

Path Queries and Graph patterns

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

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

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

- Returns information of people who are managed by employees who mentor Katie

MATCH(e:Employee) <-[:MENTORS]- () -[:MANAGES]→(n)

WHERE e.ename = ‘Katie’

RETURN n

Arrows can go in both directions

Traversals in Various Forms
OR / AND

Cypher

- Returns information of people who are managed OR mentored by Katie

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

- Returns information of people who are neither managed nor mentored by Katie

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

- Returns information of people who managed AND mentored by Katie

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

Two path patterns with same variable names – so have to bind to the same nodes

Traversals in Various Forms

Cypher

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

```
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

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)

Query Output: table of variable bindings --- not a graph as output

- One or multiple sets of nodes and edges

MATCH(e:Employee) -[ed:MANAGES]→(n)
RETURN e, ed, n

Table with 3 columns: each record: the manager node, the manager edge, and the employee node that is managed

- Attributes

MATCH(e:Employee) -[:MANAGES]→(n)
RETURN n.ename, n.ephone

Table with 2 columns: each record has the name and phone of the managed employee

- Aggregates and other functions

MATCH(e:Employee) -[:MANAGES]→(n)
RETURN MAX (n.esalary)

Table with 1 column and 1 record with the max salary among all employees that have a manager

- Paths

MATCH pa = (e:Employee) -[:MANAGES]→ (n)
RETURN pa

Table with 1 column and one record for each qualifying path. Path contains information about all nodes and edges on the path and their properties/labels.
Table with "mini-graphs"

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

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

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

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

```
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

Other Aspects

- Constraints
  - CREATE CONSTRAINT ON (e: Employee) ASSERT e.eid IS UNIQUE
  - CREATE CONSTRAINT ON (e: Employee) ASSERT exists(e.ename)
  - 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