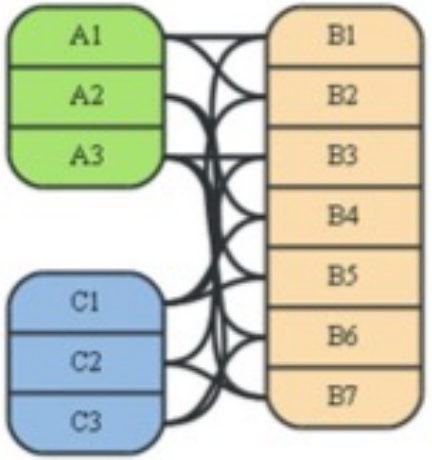
Graph Databases - neo4j

Outline

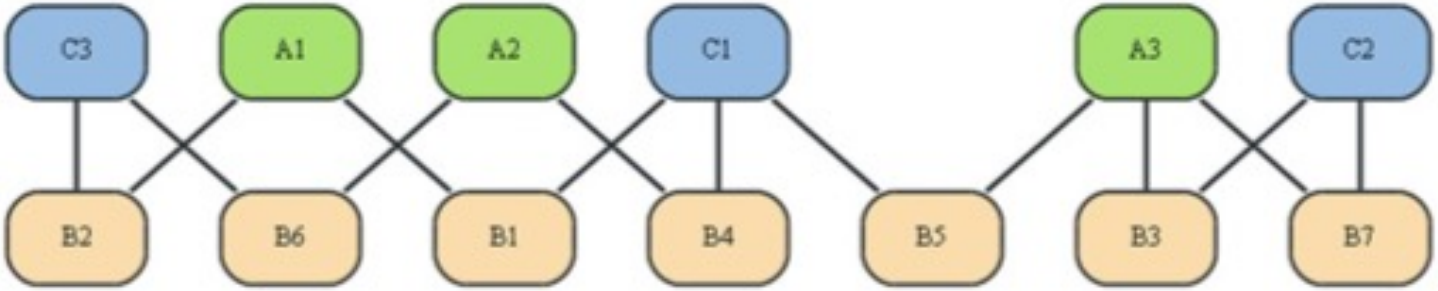
- Short intro to graph databases
- Neo4j
  - Introduction
  - Data model & interaction
  - Architecture
- Use cases

Graph databases

- Graph databases are motivated by the issues of relational databases in coping with complex relationships

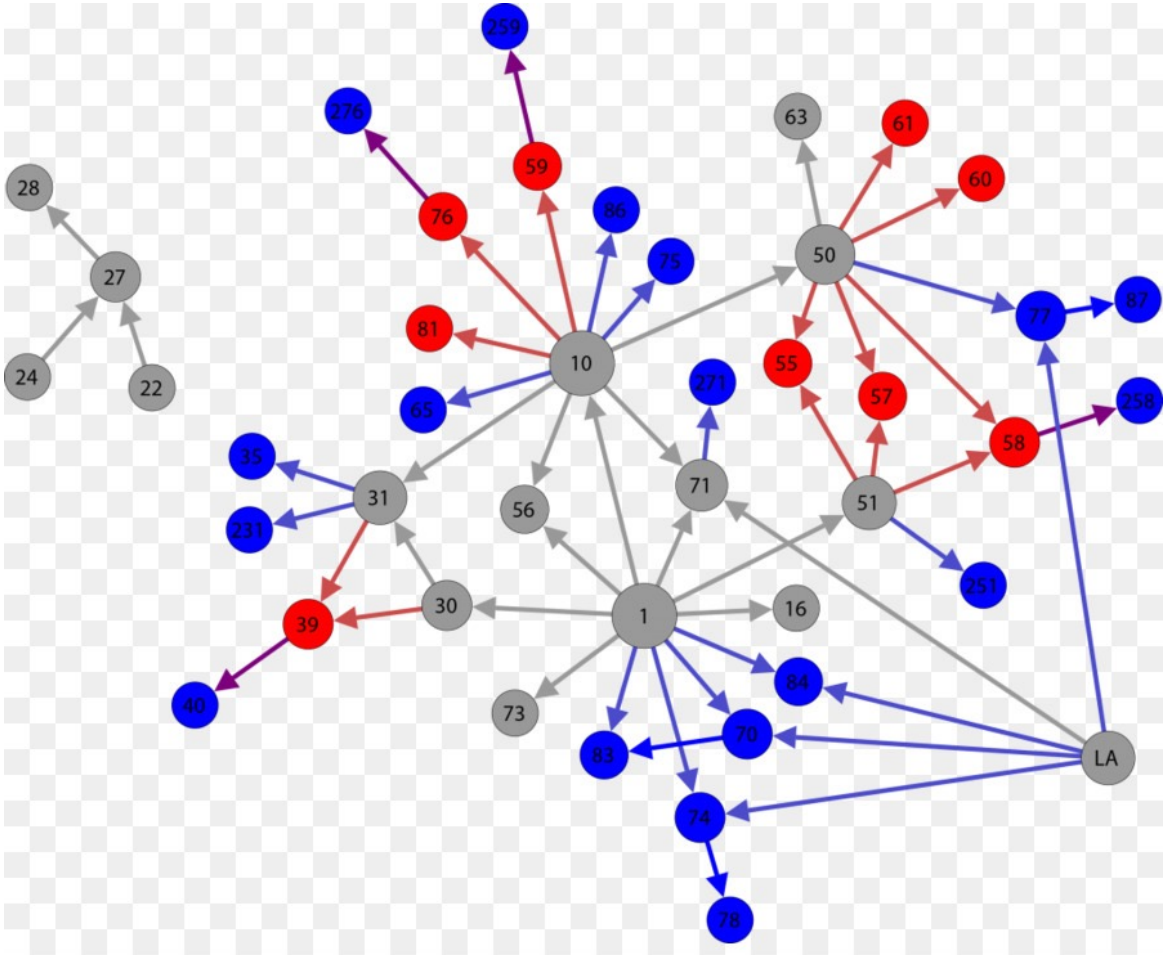


- Complex relationships require complex join



Graph databases

- Capture data consisting of complex relationships
- Focus on data inter-connection and topology

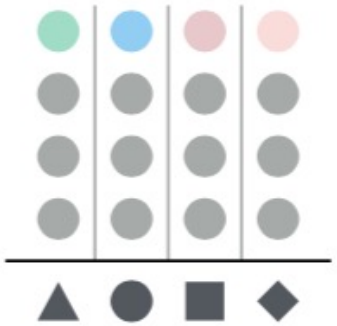


Why Graphs? (recall)

- Well-known generic data structure
- One way to address the impedence mismatch
  (objects in your Java don't match the structure of a DB)
- Maths and algorithms are well understood
- «blackboard friendly»

- Data naturally modelled as graphs

Which is Good at What?



Relational Database

Good for:

- Well-understood data structures that don't change too frequently
- Known problems involving discrete parts of the data, or minimal connectivity

Graph Database

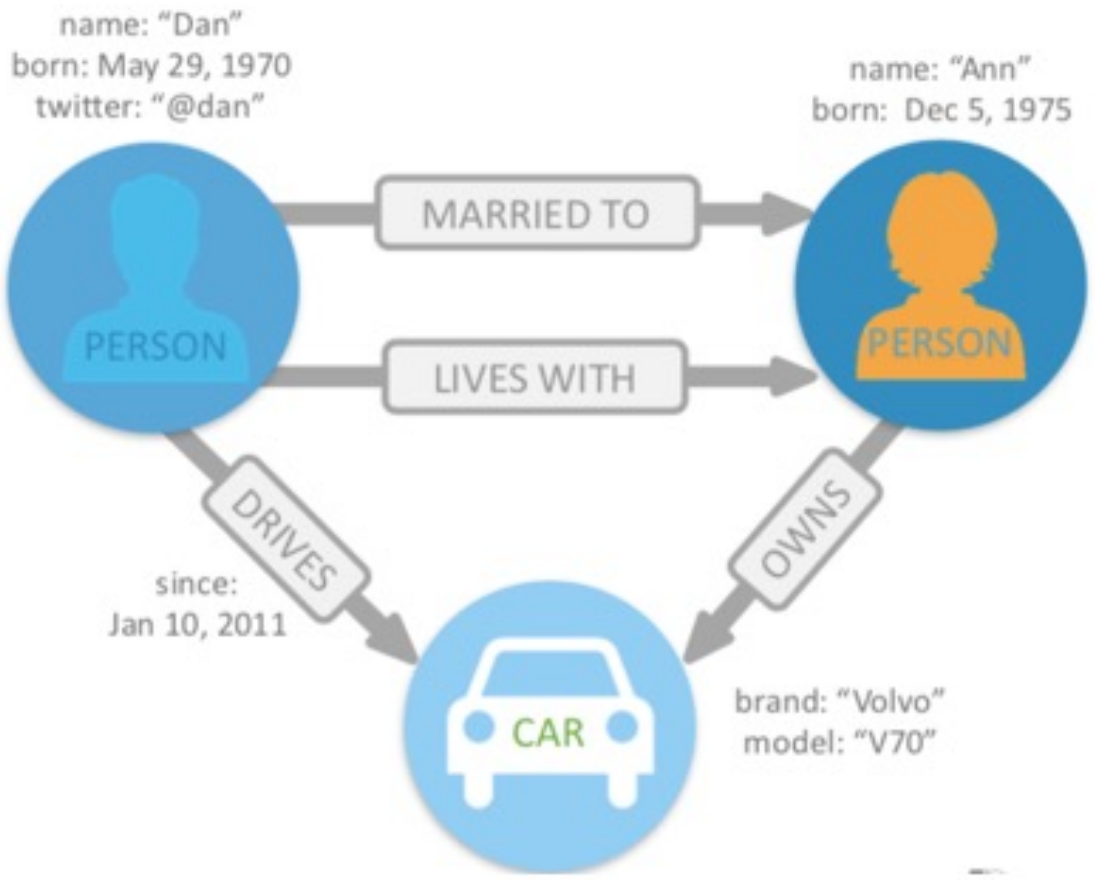
Good for:

- Dynamic systems: where the data topology is difficult to predict
- Dynamic requirements: that evolve with the business
- Problems where the relationships in data contribute meaning & value

Graph Model

- Complex, densely-connected domains
  - Lots of join tables? Relationships
  - Lots of sparse tables? Semi-structure
- Messy data
  - Ad hoc exceptions
- Relationships as first-class elements
  - Semantic clarity: named, directed
  - Not simply constraints
  - Can have properties

Graph Data Model (property graph)



NoSQL databases
graph vs aggregate-oriented databases

Graph databases

- graph-based query languages
- **partitioning is difficult**: more likely to run on a single server
- transactions maintain **consistency** over multiple nodes and edges

Aggregate-oriented

- simple query languages
- distributed across clusters
- no ACID guarantees

Popular graph dbs/systems

Property graph model: Neo4j, Titan, InfiniteGraph, …
Triple store model (RDF): AllegroGraph, Virtuoso, …

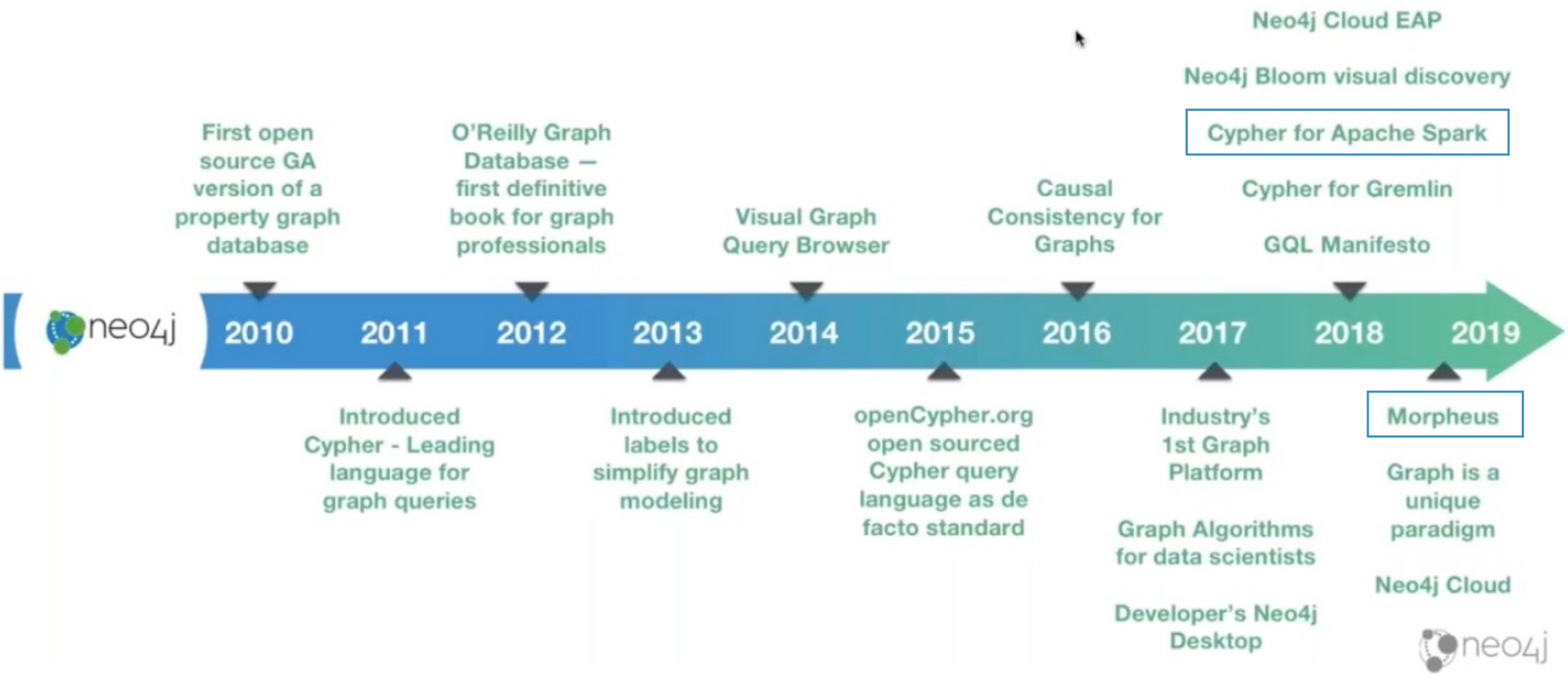
Neo4j - Introduction

neo4j in short

Feature	neo4j
Model	Graph-based
Query language	Supported, Cypher
Reference scenarios	transactional (read intensive) & analytical
Partitioning	Difficult (application-level)
Indexes	on properties (simple and composite), full-text
Replication	Master-slave, single-leader
Consistency	Strong
Availability	Load balancing among read replicas
Fault tolerance	By re-electing a master in case it goes down
Transactions	ACID (mantain consistency over multiple nodes and edges)
CAP theorem	CA
Distributed by	Neo4j Inc.

neo4j

- First released in 2010
- Developed in Java
- Community and Enterprise versions, dual-licensed: GPL v3 and a commercial license
  - the Community Edition is free but is limited to running on one node only due to the lack of clustering
- Query language: Cypher
- Java traversal API


Neo4j Cloud EAP
Neo4j Bloom visual discovery
Cypher for Apache Spark
Cypher for Gremlin
GQL Manifesto
First open source GA version of a property graph database
O'Reilly Graph Database — first definitive book for graph professionals
Visual Graph Query Browser
Causal Consistency for Graphs
neo4j
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
Introduced Cypher - Leading language for graph queries
Introduced labels to simplify graph modeling
openCypher.org open sourced Cypher query language as de facto standard
Industry's 1st Graph Platform
Graph Algorithms for data scientists
Developer's Neo4j Desktop
Morpheus
Graph is a unique paradigm
Neo4j Cloud
neo4j

Who uses neo4j?

https://neo4j.com/customers

Data model and interaction

Property Graph, Traversals, and Cypher Query Language

Property Graph Model

A property graph *G = (V, E)* is a directed multigraph

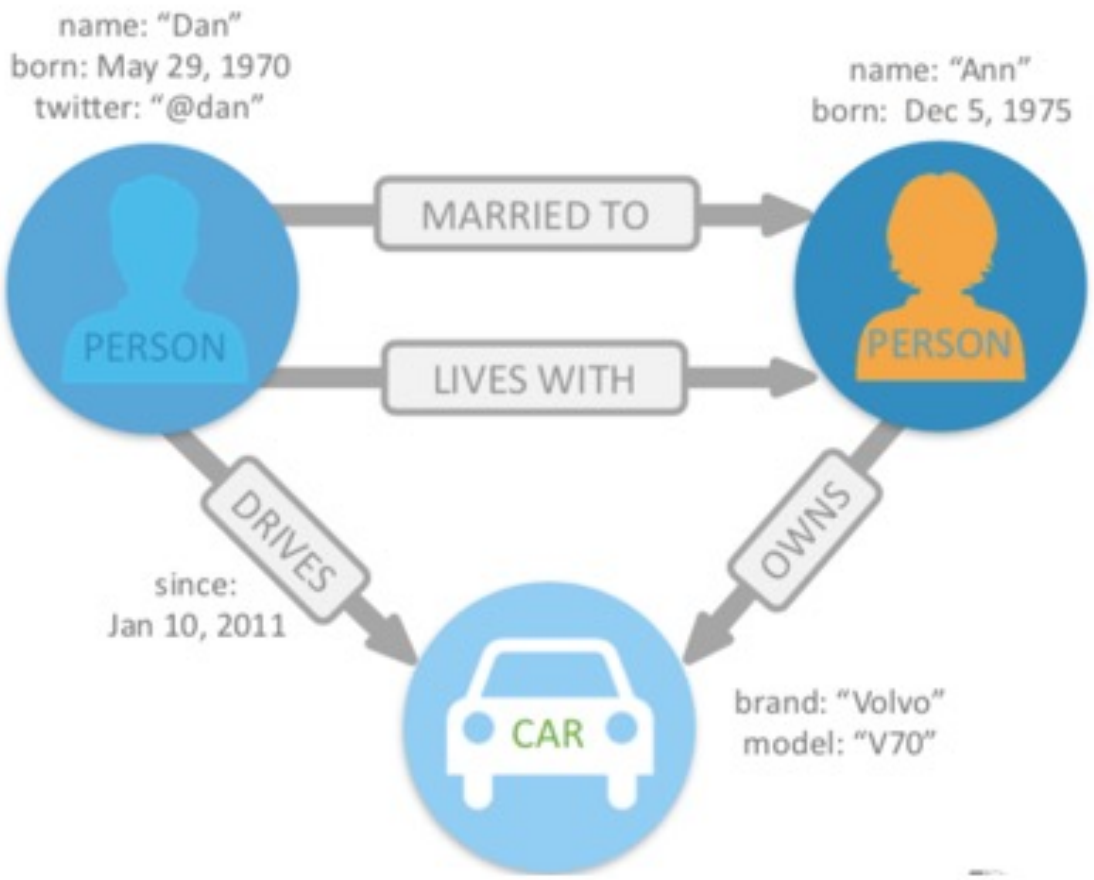
where:

- every node v ∈ V and every edge e ∈ E can be associated with a set of pairs <key, value>, called properties
  - value can be of primitive type or an array of primitive type
- every node v ∈ V can be tagged with one or more label(s)
- every edge e ∈ E is associated with a type

Property Graph Model

Edges have incoming/outgoing edges but traversal is equally efficient in both directions

Schema-less model



Graph vs Relational Data Model

- The relational model is designed for a single type of relationship

- Adding another relationship usually means a lot of schema changes
- In RDBMS we model the graph beforehand based on the traversal we want
  - If the traversal changes, the data will have to change

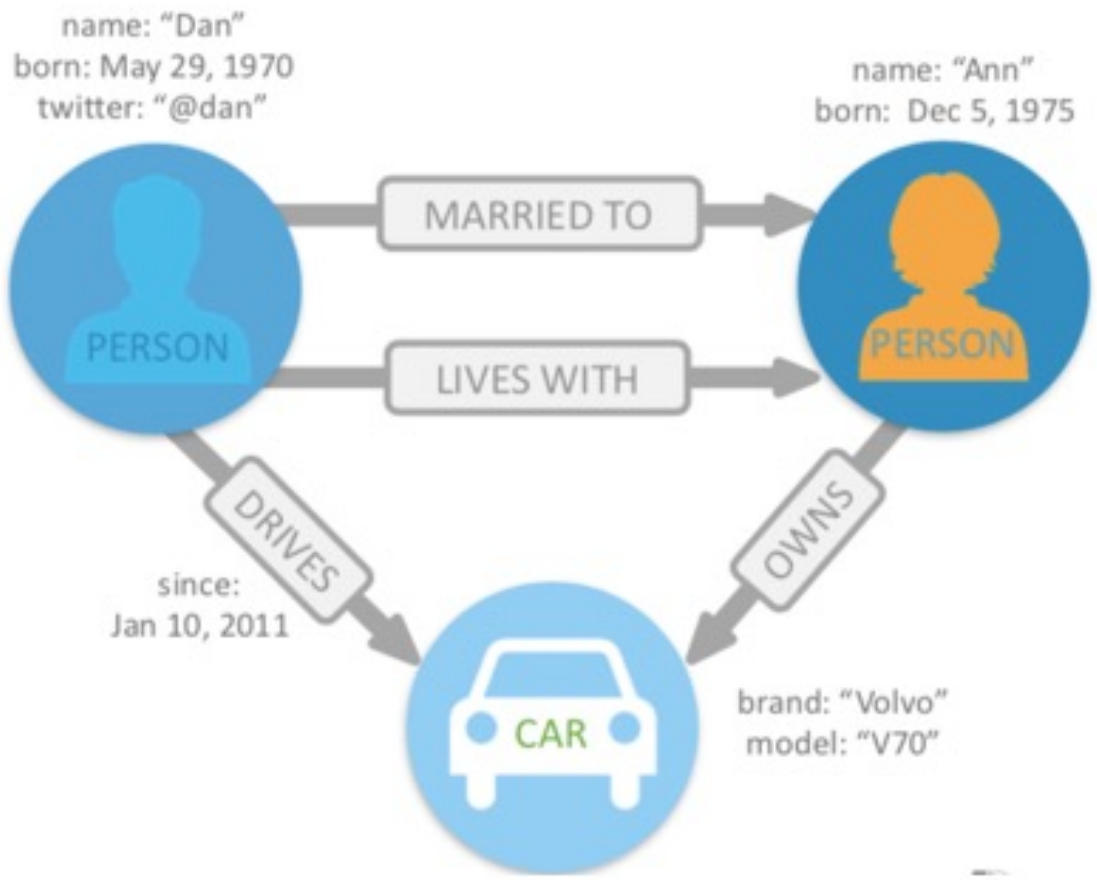
Schemaless Model: Flexibility

- Flexible property graph schema
- Neither node labels nor relationship types prescribe properties
  - freely add new edges
  - freely add properties to nodes and edges
- Schema can adapt over time
  - easy to add new nodes and relationships when the business needs change
  - changing existing nodes and their relationships is costly! (similar to data migration)

Queries as Traversals

Queries = Traversals

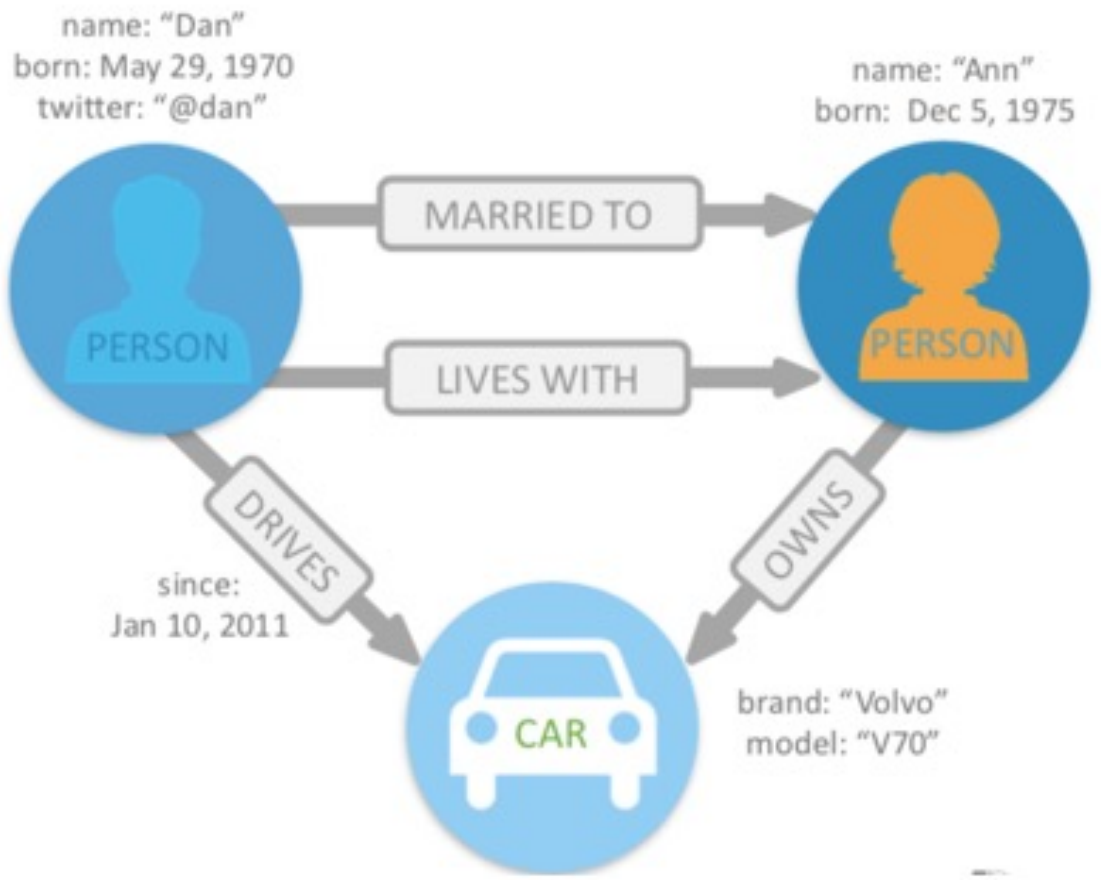
get the spouse of the owner of a Volvo



Queries = Traversals

get the spouse of the owner of a Volvo

get the owner of a car a person drives

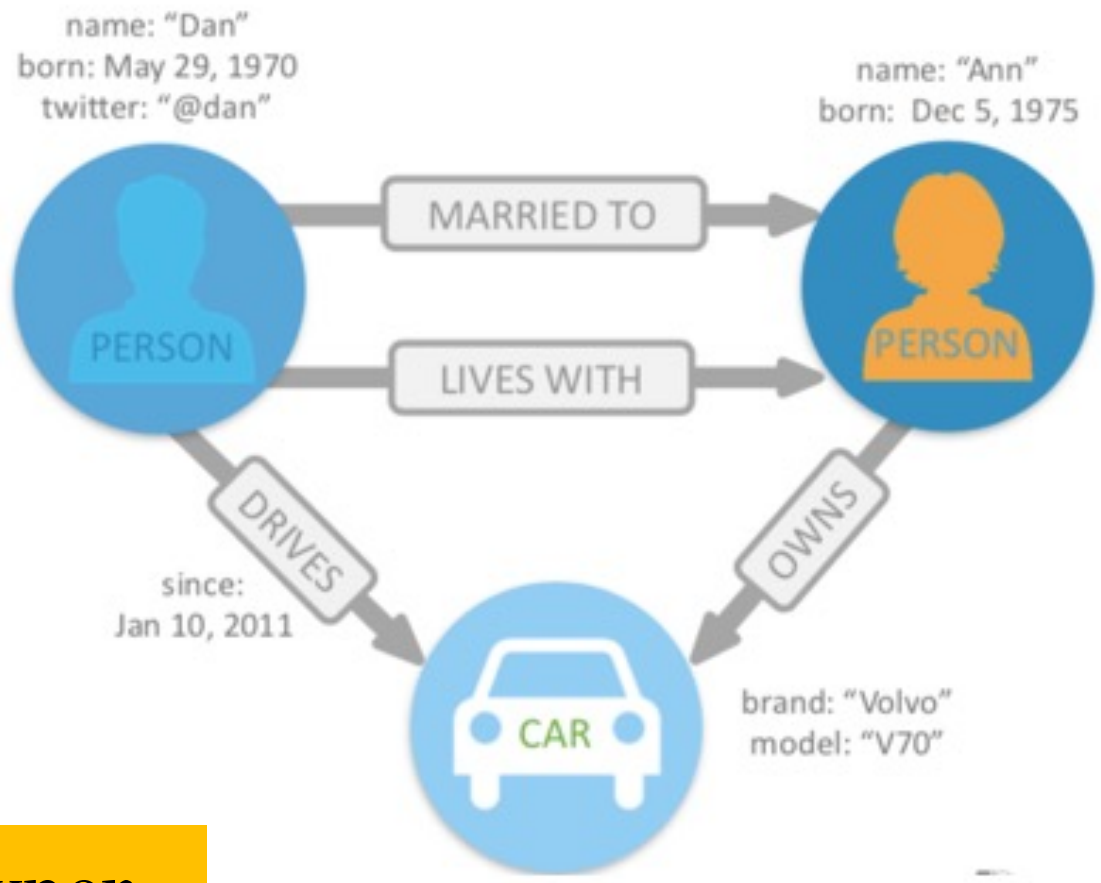


Queries = Traversals

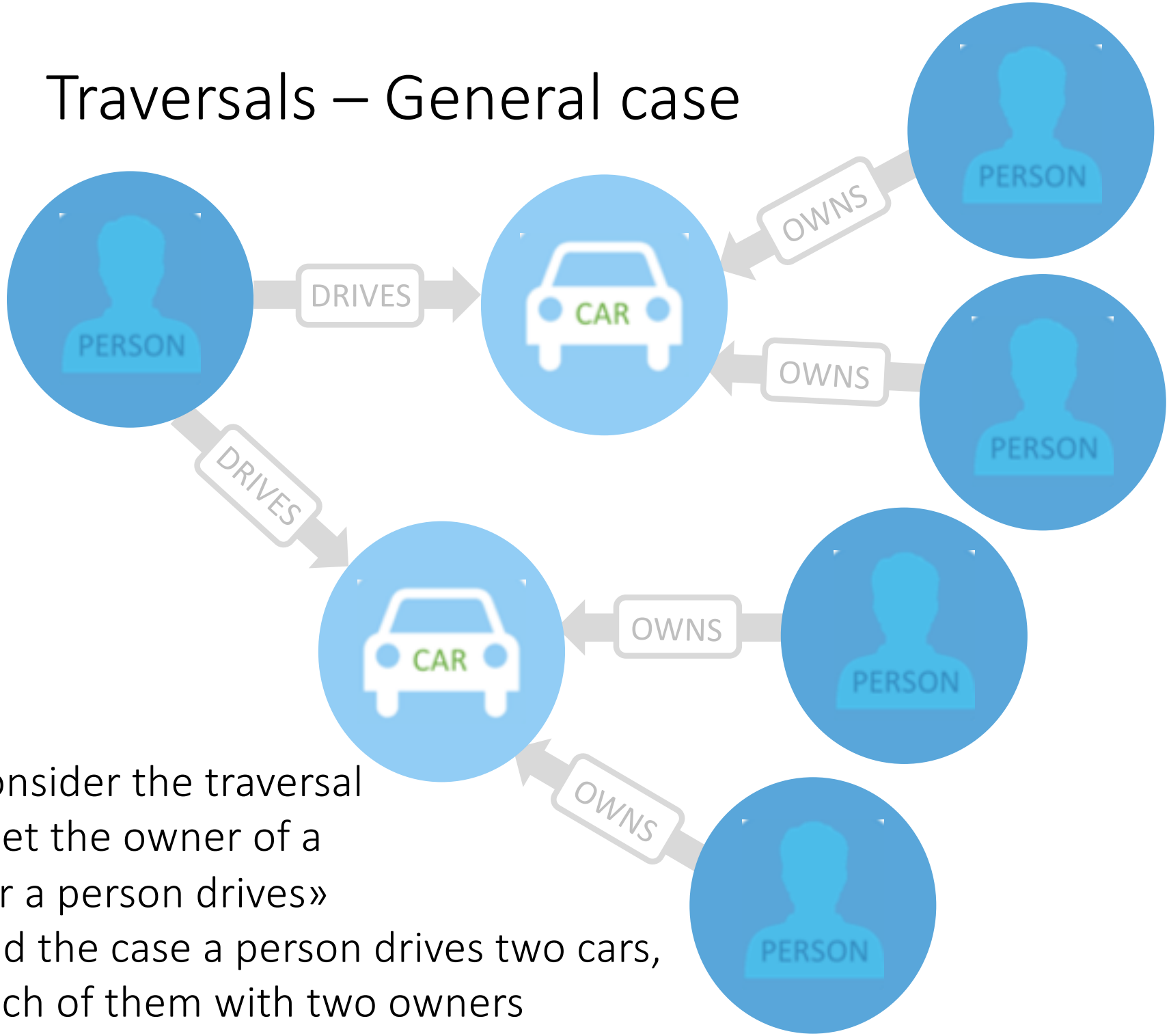
get the spouse of the owner of a Volvo

get the owner of a car a person drives

get the spouse of the owner of a car a person drives

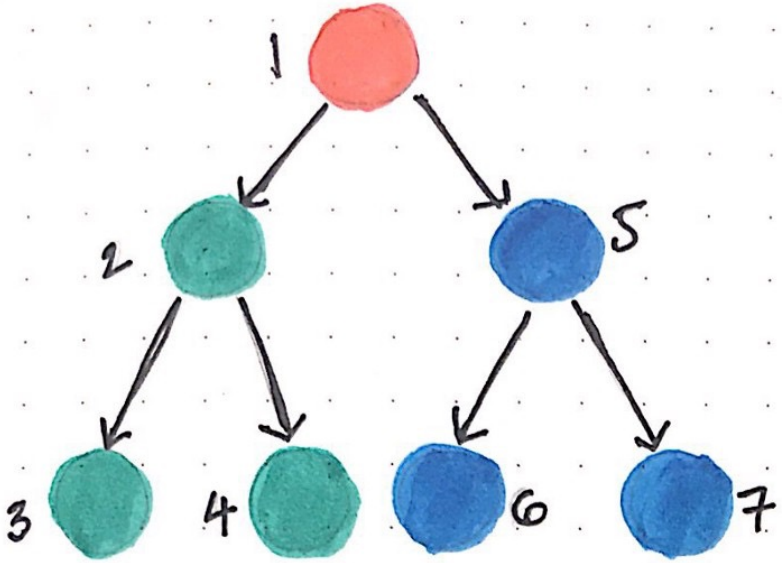


Traversals – General case



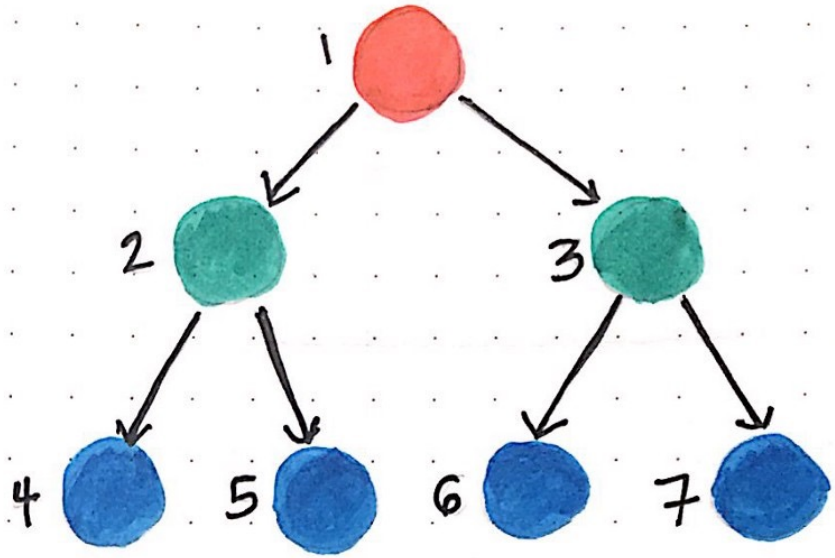
Consider the traversal
«get the owner of a
car a person drives»
and the case a person drives two cars,
each of them with two owners

Graph Traversals



Depth-first search

- Traverse through left subtree(s) first, then traverse through the right subtree(s).



Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Constant Time Traversal

- Constant time traversals in big graphs for both depth and breadth due to efficient representation of nodes and relationships
- Enables scale-up to billions of nodes on moderate hardware

Graph traversal

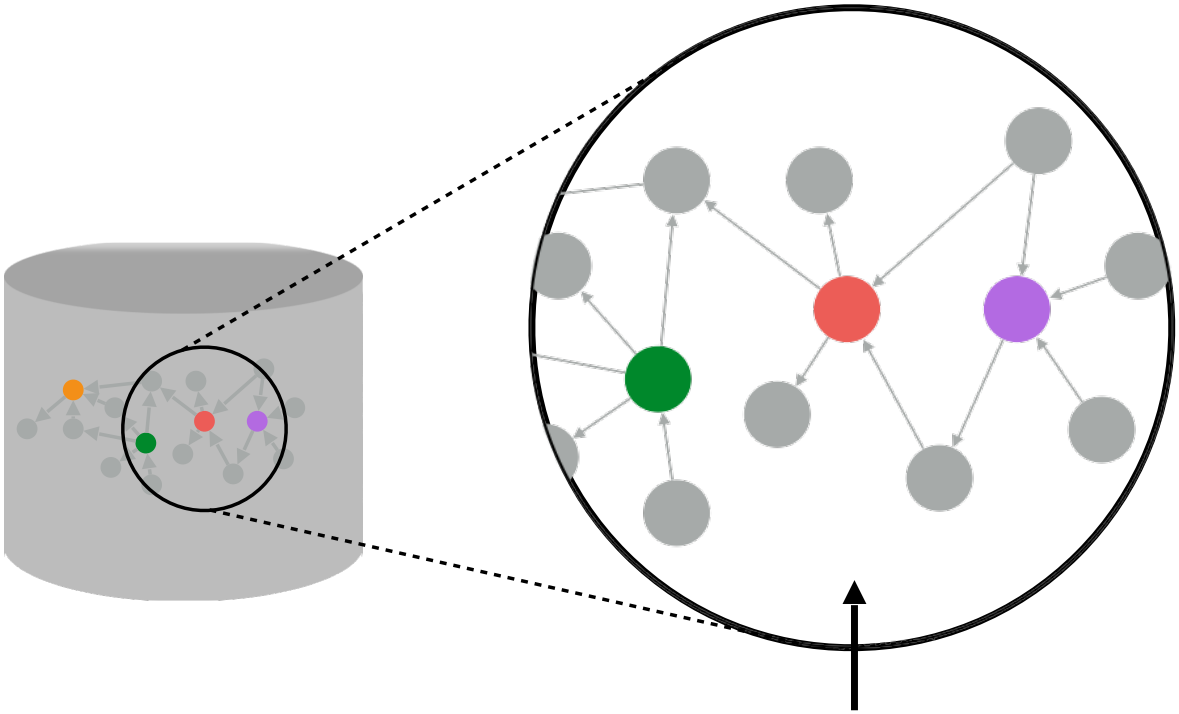
- Traversing relationships is very fast
- The relationship between nodes is not calculated at query time but it is actually persisted
- Traversing persisted relationships is faster than calculating them for every query

Index-free adjacency

We say that a (graph) database *g* satisfies the index-free adjacency if the existence of an edge between two nodes n_1 and n_2 in *g* can be tested on those nodes and does not require to access an external, global, index.

The cost of a basic traversal is independent of the size of the database

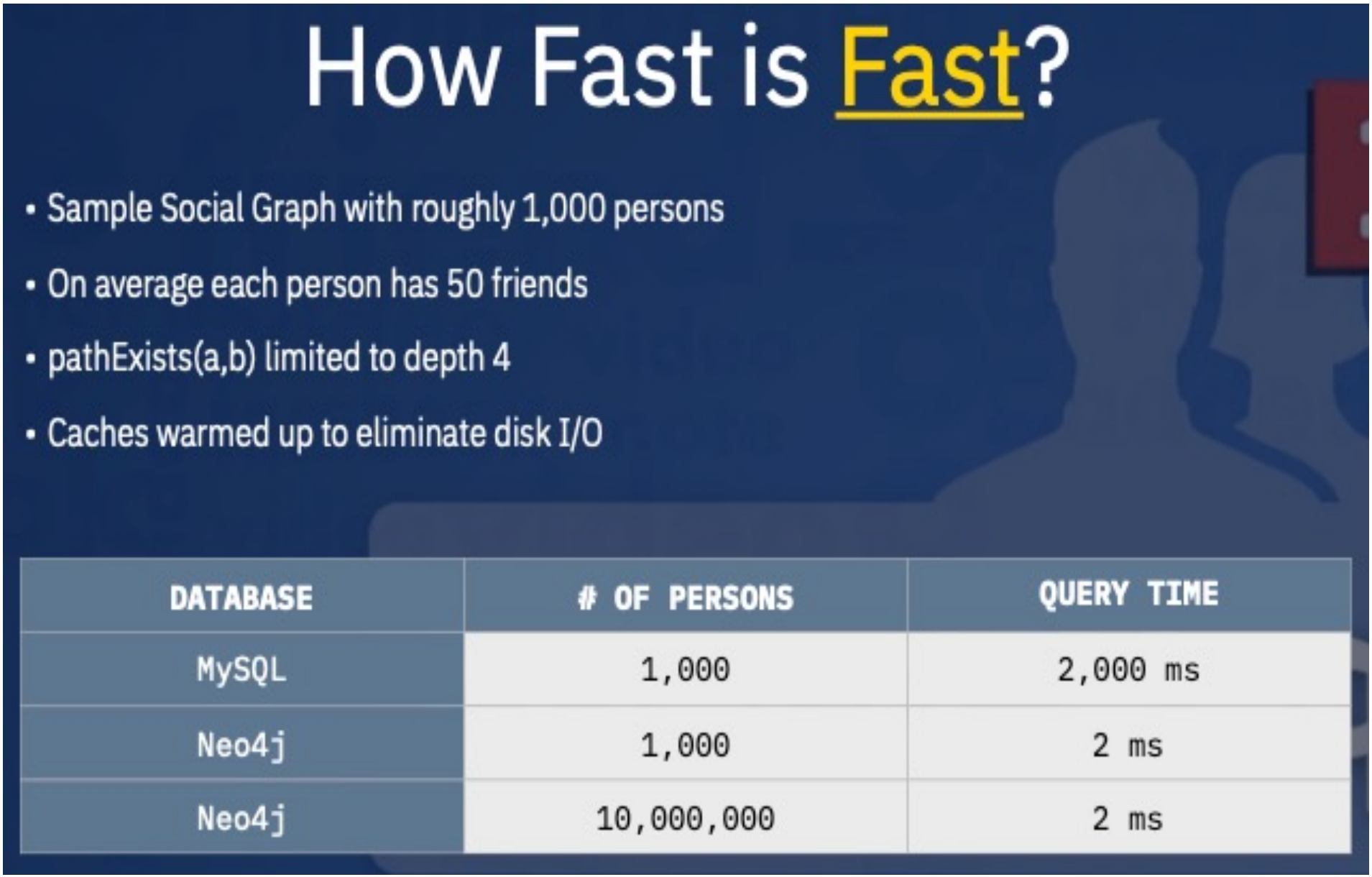
Index-free adjacency



At Write Time:
Data is ***connected***
as it is stored

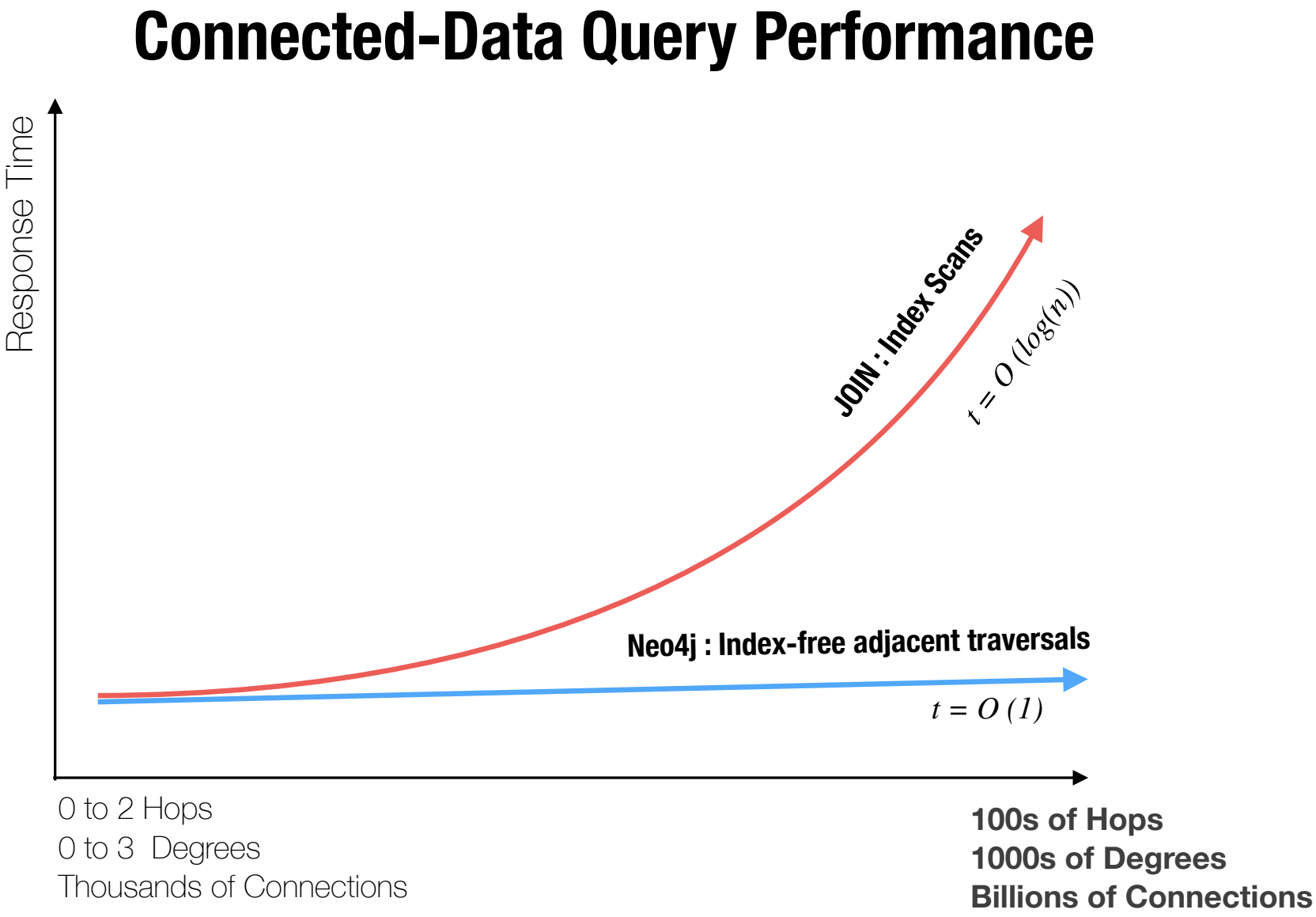
At Read Time:
Lightning-fast retrieval of data and relationships
via pointer chasing

Fast Graph Traversal



DATABASE	# OF PERSONS	QUERY TIME
MySQL	1,000	2,000 ms
Neo4j	1,000	2 ms
Neo4j	10,000,000	2 ms

Index-Free Traversal



Graph vs relational data model

- Relational databases
  - implement relationships using foreign keys
  - joins require to navigate around and can get quite expensive

- Graph databases
  - make traversal along the relationships very cheap
  - performance is better for highly connected data
  - shift most of the work from query time to insert time
  - good when querying performance is more important than insert speed

Cypher Query Language

Declarative

- Most of the time, Neo4j knows better than you

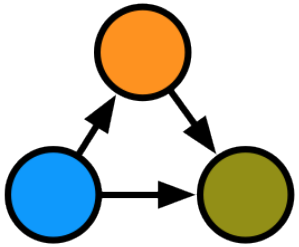
Imperative

- follow relationship
  breadth-first vs depth-first
- explicit algorithm

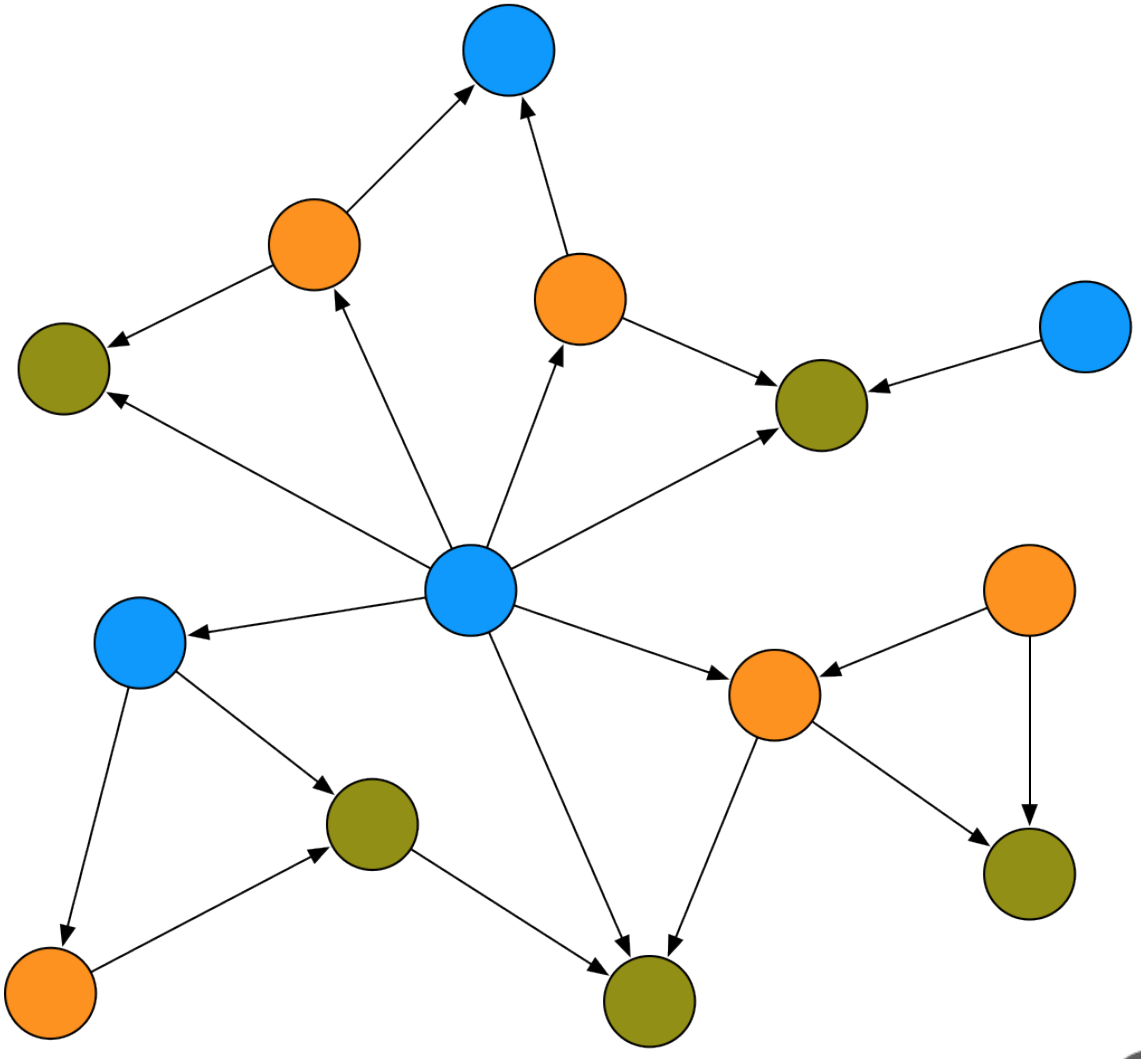
Declarative

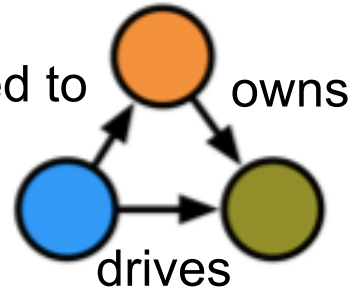
- specify starting point
  specify desired outcome
- algorithm adaptable based on query

Pattern Matching

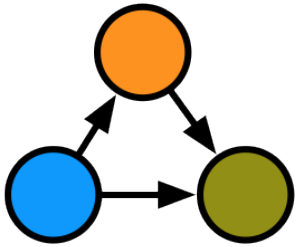


Pattern

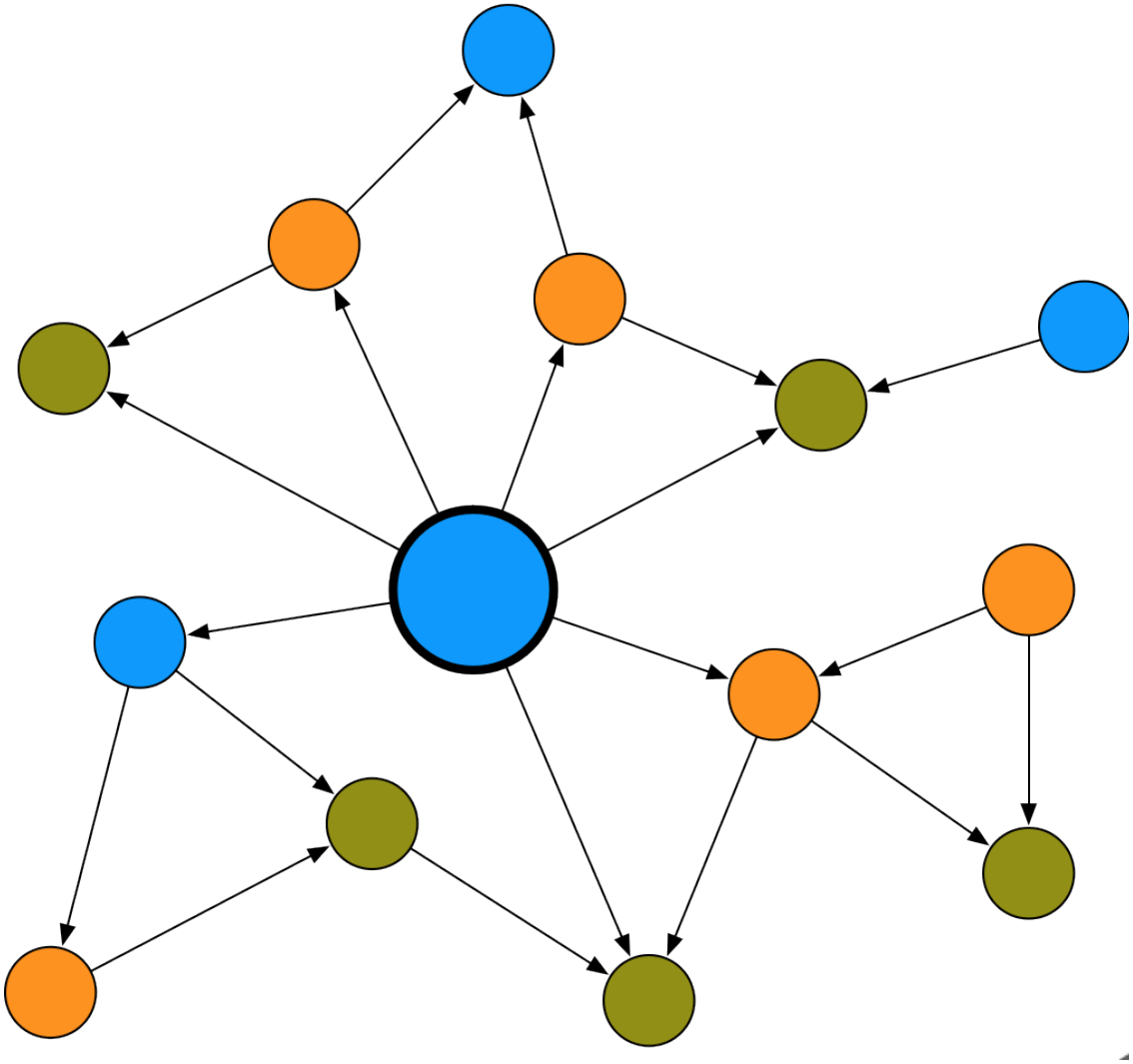




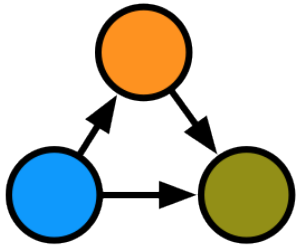
Pattern Matching – Start Node



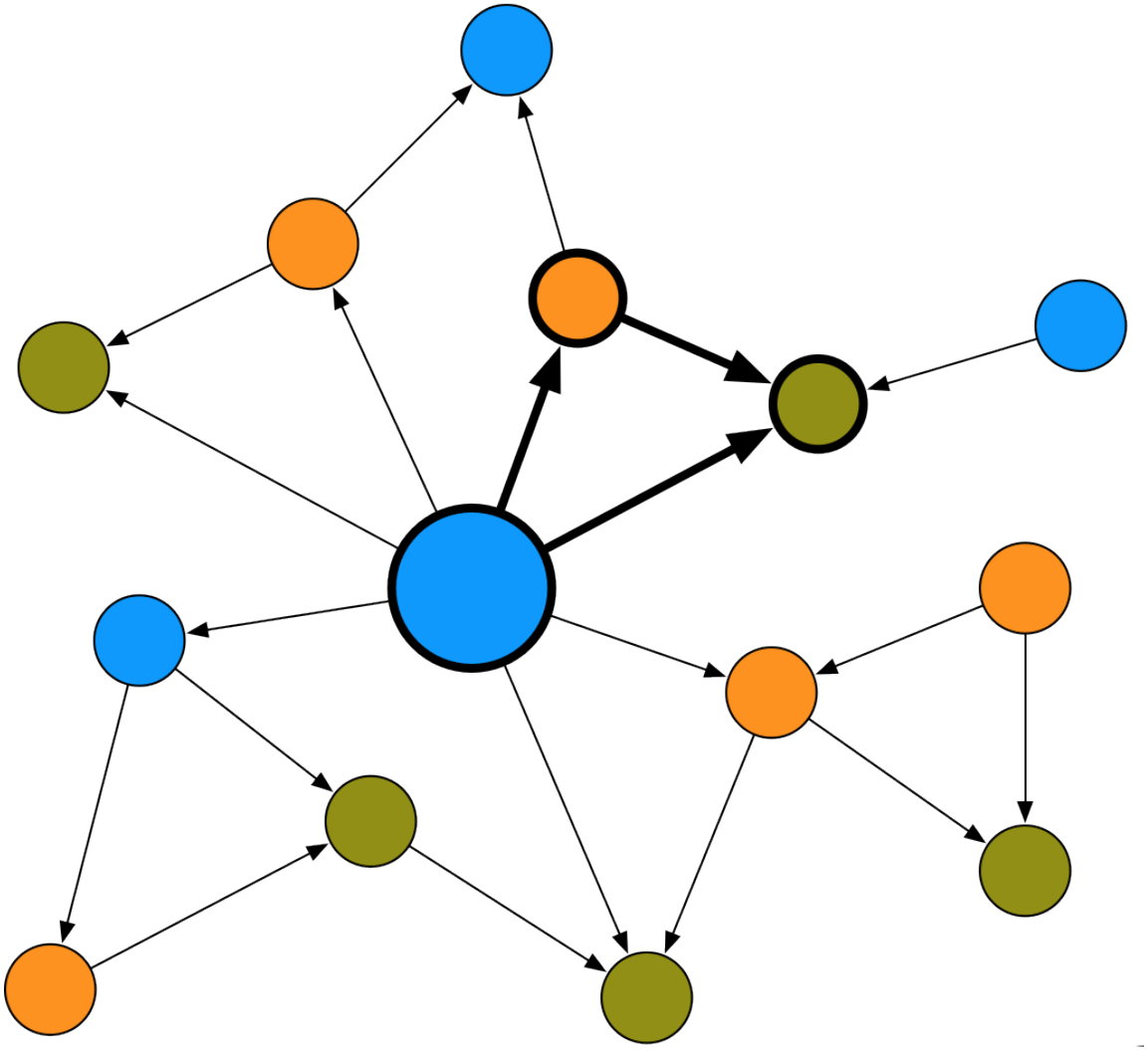
Pattern



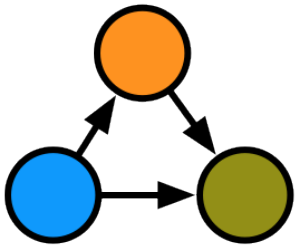
Pattern Matching - Match



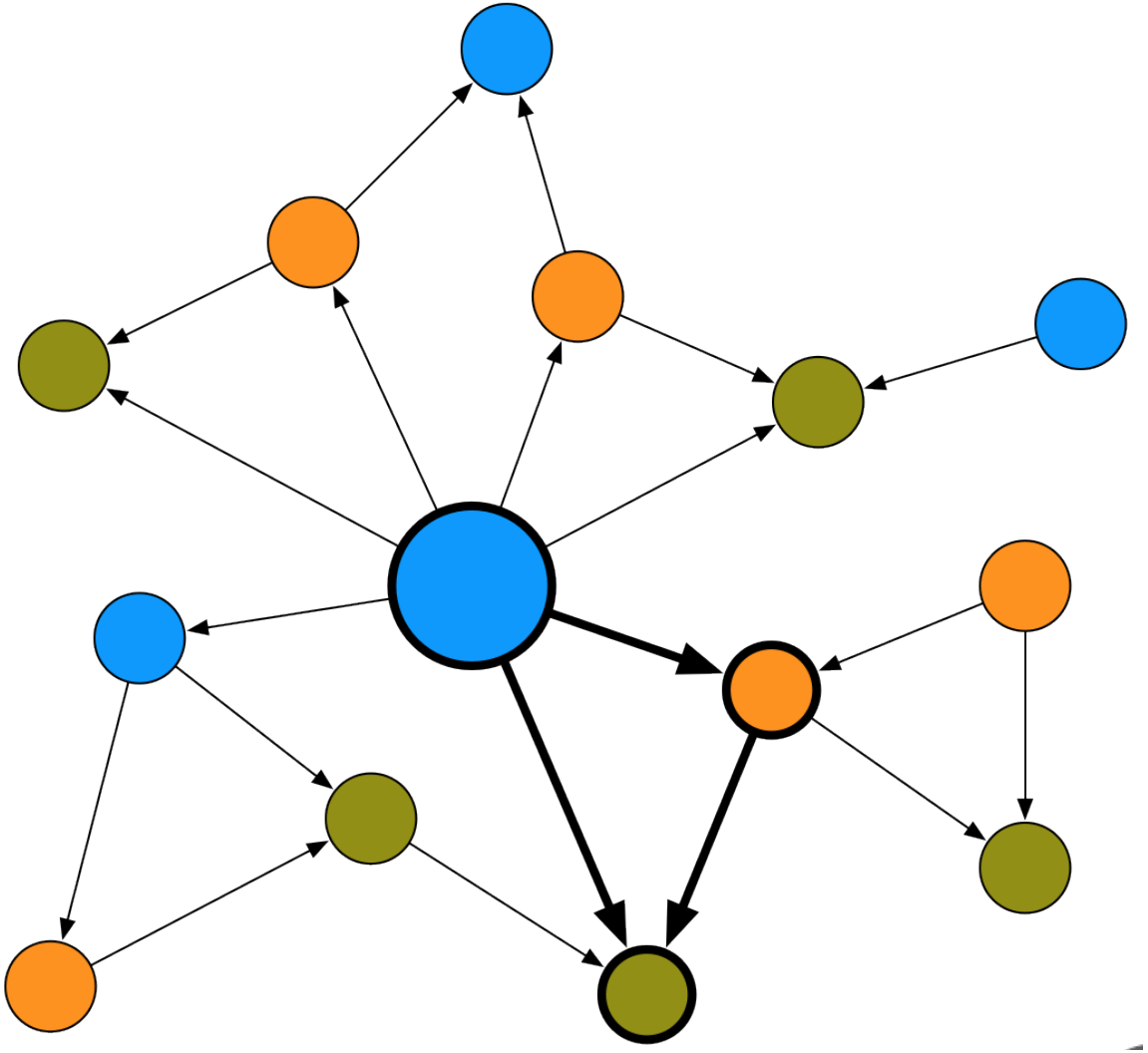
Pattern



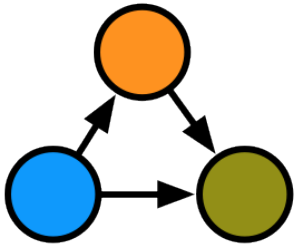
Pattern Matching - Match



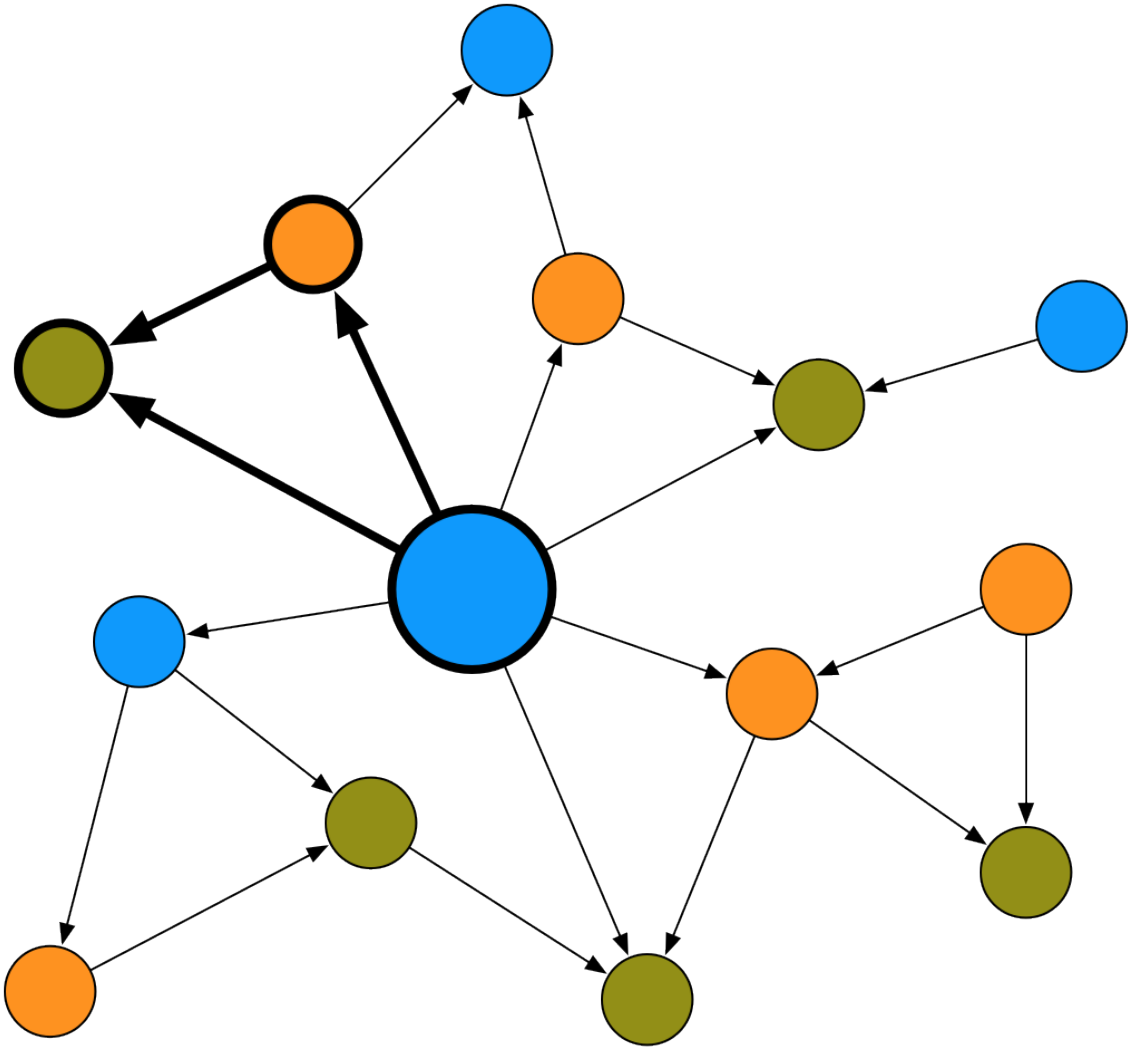
Pattern



Pattern Matching - Match



Pattern

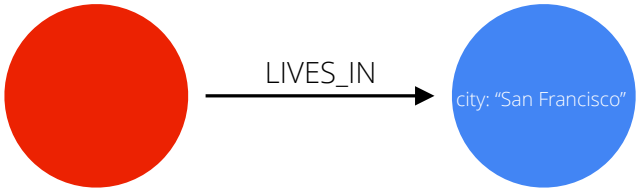


ASCII Art Patterns

Person

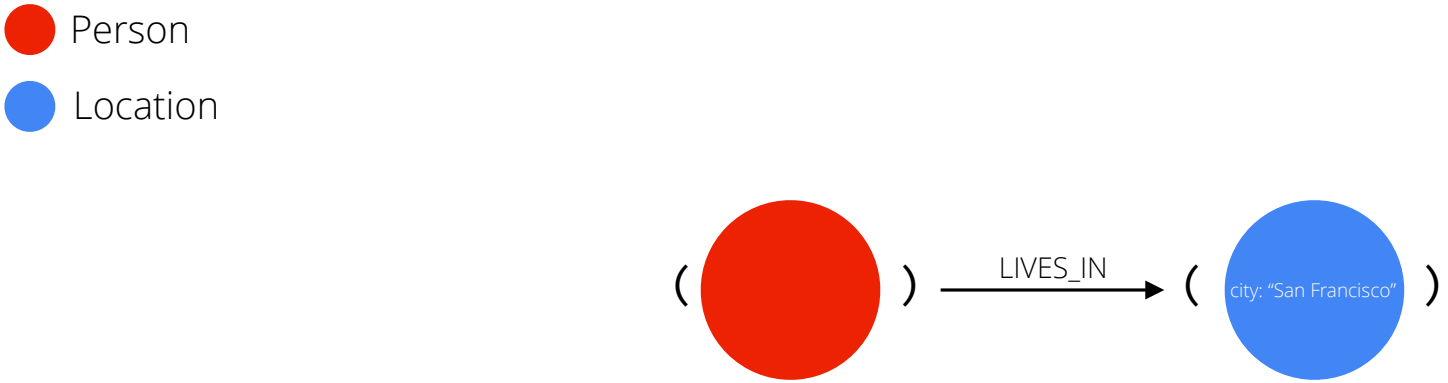
Location

Pattern: Persons who live in San Francisco



ASCII Art Patterns

Pattern: Persons who live in San Francisco



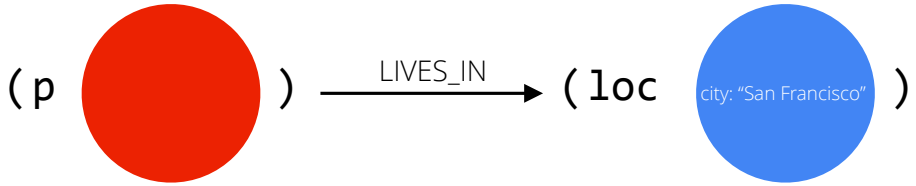

```
node        (   )
```

ASCII Art Patterns

Person

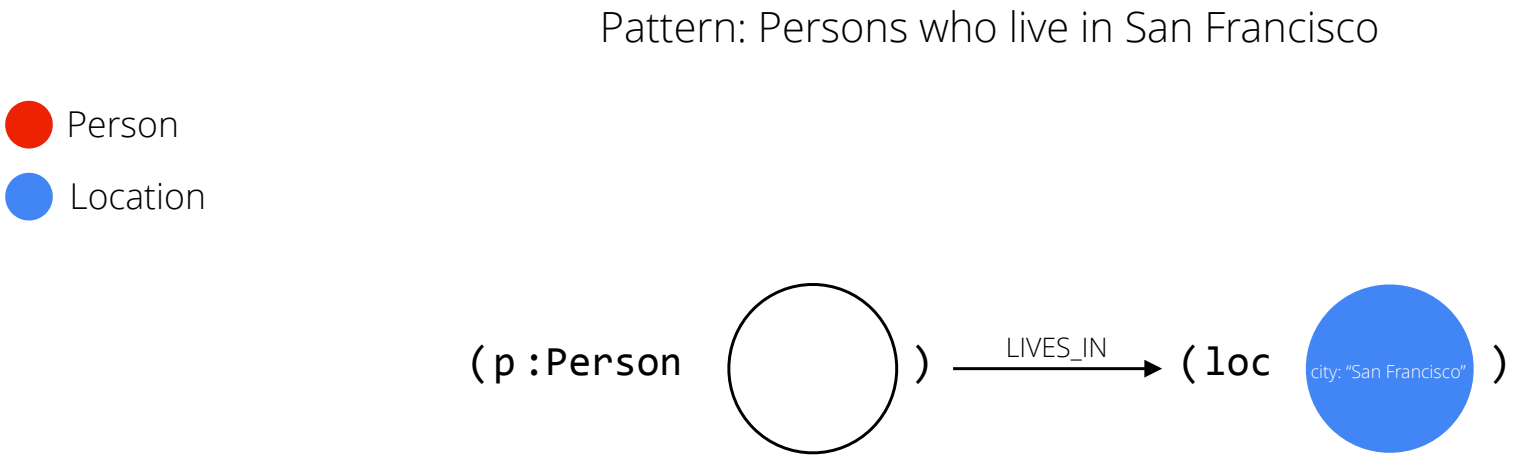
Location

Pattern: Persons who live in San Francisco



node variable `( p )`

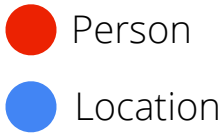
ASCII Art Patterns

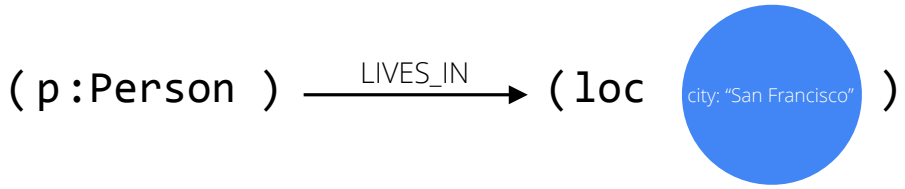


node label `( :Person )`

ASCII Art Patterns

Pattern: Persons who live in San Francisco

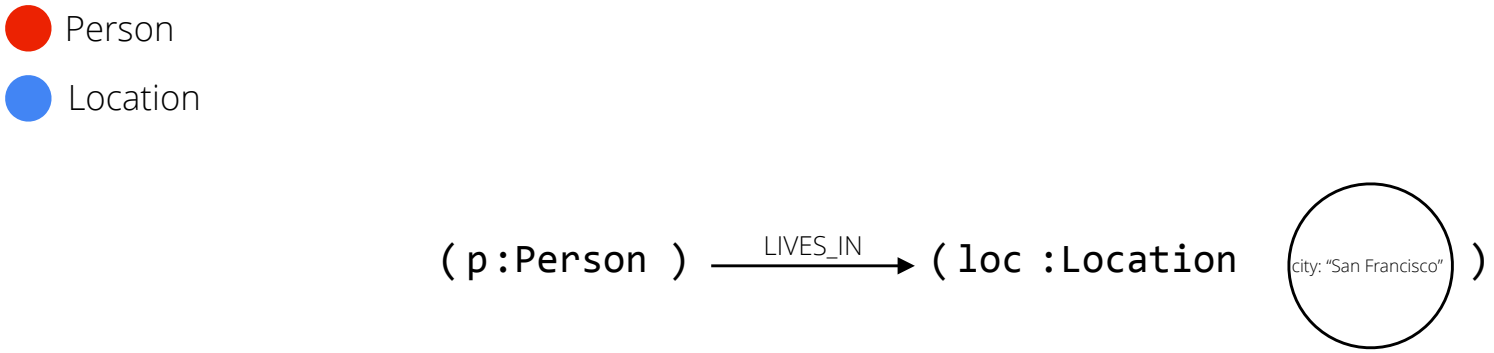




node label `(:Person)`

ASCII Art Patterns

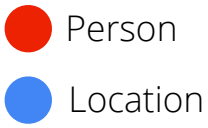
Pattern: Persons who live in San Francisco



node label `(:Location)`

ASCII Art Patterns

Pattern: Persons who live in San Francisco

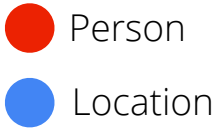



```
(p:Person )  -[LIVES_IN]->  (loc :Location  {city: “San Francisco”}  )
```

```
property   (city:’’San Francisco’’)
```

ASCII Art Patterns

Pattern: Persons who live in San Francisco

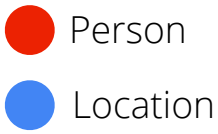



```
(p:Person )-[:LIVES_IN]->(loc :Location  {city: “San Francisco”}   )
```

edge `-->`

ASCII Art Patterns

Pattern: Persons who live in San Francisco

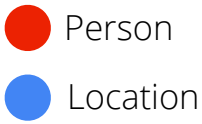



```
(p:Person) -[:LIVES_IN]-> (loc:Location {city: “San Francisco”})
```

edge type `-[:LIVES_IN]->`

ASCII Art Patterns

Pattern: Persons who live in San Francisco




```
MATCH  (p:Person ) -[:LIVES_IN]  ->  (loc :Location  {city: “San Francisco”}   )
RETURN  p
```

```
MATCH (node1:Label1 {property:value} )-->(node2:Label2)
RETURN node1.property
```

```
MATCH (n1:Label1)-[rel:TYPE {property:value}]->(n2:Label2)
RETURN rel.property, type(rel)
```

Structural Patterns

"ASCII art"

variable
:node label
property values {k:v}

```
( ) - - ( )

( ) <- - ( )

( ) - -> ( )
```

Conditions on the relationship may be specified inside []

variable
:edge type
property values

Structural Patterns - Nodes

- `()`
- `(matrix)`
- `(:Movie)`
- `(matrix:Movie)`
- `(matrix {title: "The Matrix"})`
- `(:Movie {title: "The Matrix", released: 1999})`
- `(matrix:Movie {released: 1999})`
- …

Structural Patterns - Relationships

More “ASCII art” (combines with <, >, :)

variable length path

```
() — [*] - ()
```

constrained length path

```
() — [*min..max] - ()
```

Structural Patterns - Paths

• `(a)-[*]->(b)`	traverse any depth
• `(a)-[*depth]->(b)`	exactly *depth* steps long
• `(a)-[*1..4]->(b)`	from one to four levels deep
• `(a)-[:KNOWS*3]->(b)`	relationships of type KNOWS at 3 levels distance
• `(a)-[:KNOWS*..5]->(b)`	relationships of type KNOWS at most 5 levels distance
• `(a)-[:KNOWS|:LIKES*2..]->(b)`	relationships of type KNOWS or LIKES from 2 levels distance

Stuctural Patterns

- friend-of-a-friend

```
(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
```

- shortest path:

```
path = shortestPath( (user)-[:KNOWS*..5]-(other) )
```

- collaborative filtering

```
(user)-[:PURCHASED]->(product)<-[:PURCHASED](otherUser)
```

NOTE we need to impose that otherUser <> user !

- tree navigation

```
(root)<-[:PARENT*]-(leaf:Category)-[:ITEM]->(data:Product)
```

Stuctural Patterns

- collaborative filtering

:Product :Customer

```
(user)-[:PURCHASED]->(product)<-[:PURCHASED](otherUser)
otheruser <> user
```

What if we are looking for a product bought by (at least) 3 users?

We may have multiple paths in the same MATCH, comma separated, sharing variables

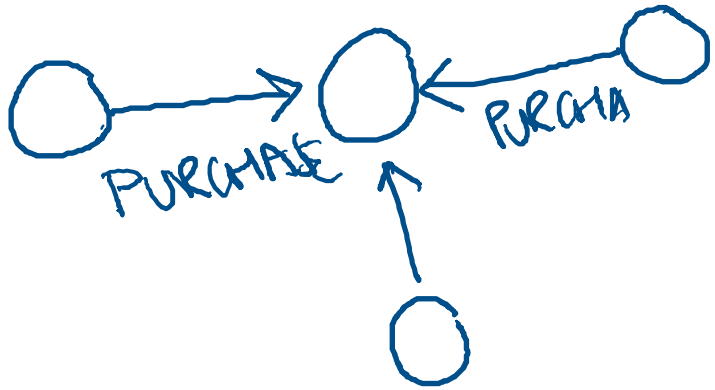
Stuctural Patterns

- collaborative filtering

```
(user)-[:PURCHASED]->(product)<-[:PURCHASED](otherUser)
```

```
otheruser <> user
```

What if we are looking for a product bought by (at least) 3 users?



We may have multiple paths in the same MATCH, comma separated, sharing variables

```
MATCH (user)-[:PURCHASED]->(product)<-
[:PURCHASED](otherUser),
(product)<-[:PURCHASED]-(thirdUser)
```

```
otheruser <> user, thirduser <> user, thirduser <> otheruser
```

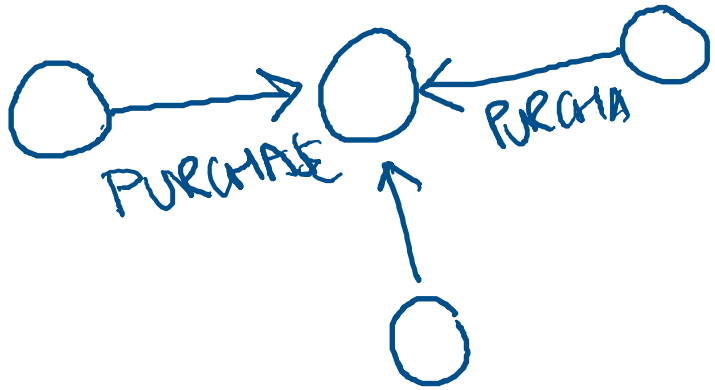
Stuctural Patterns

- collaborative filtering

```
(user)-[:PURCHASED]->(product)<-[:PURCHASED](otherUser)
```

```
otheruser <> user
```

What if we are looking for a product bought by (at least) 3 users?



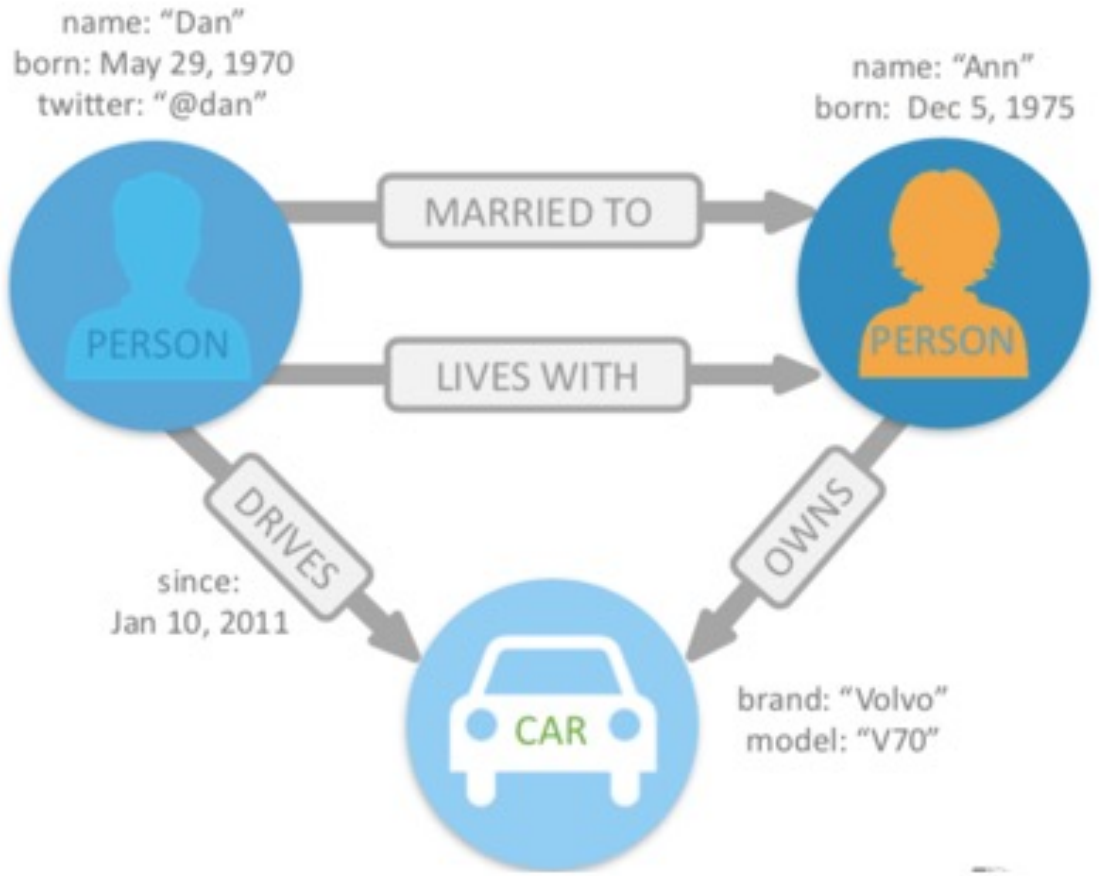
We may have multiple paths in the same MATCH, comma separated, sharing variables

```
MATCH (user)-[:PURCHASED]->(product),
(product) <-[:PURCHASED]-(otherUser),
(product)<-[:PURCHASED]-(thirdUser)
```

```
otheruser <> user, thirduser <> user, thirduser <> otheruser
```

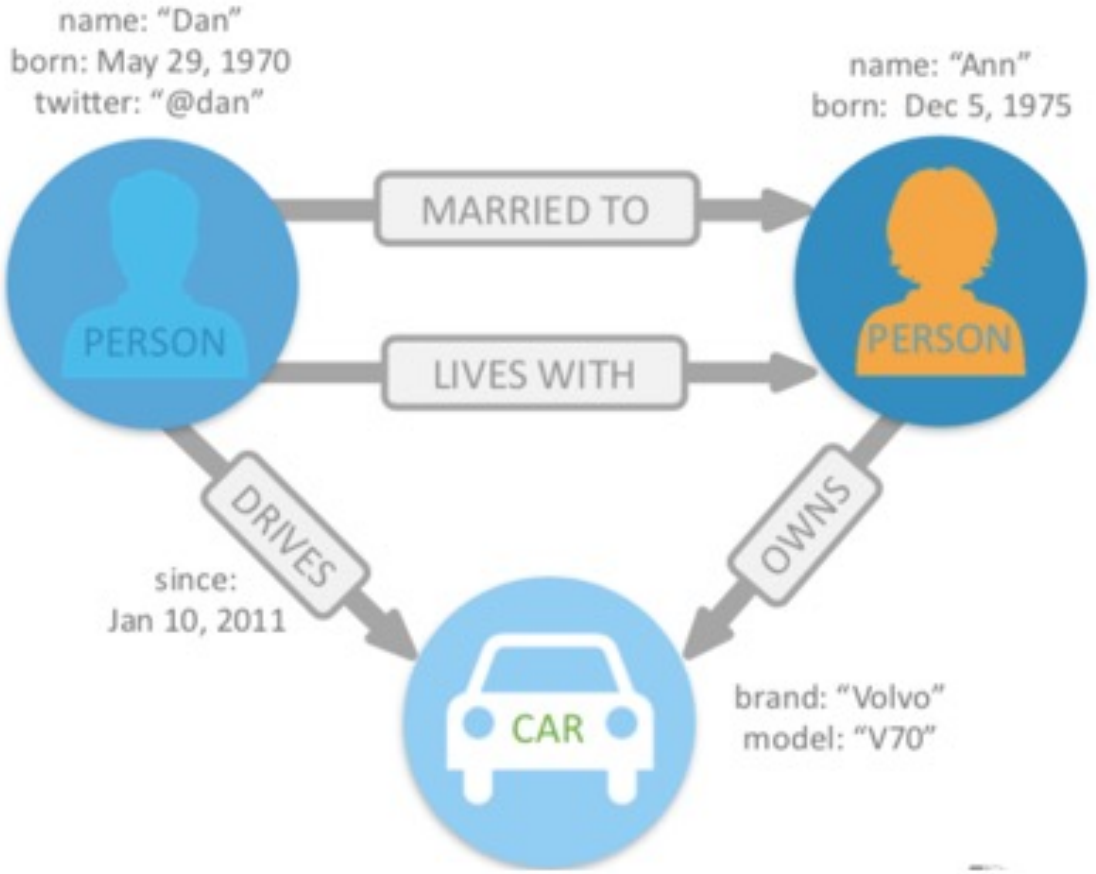
Queries = Traversals

get the spouse of the owner of a Volvo



Queries = Traversals

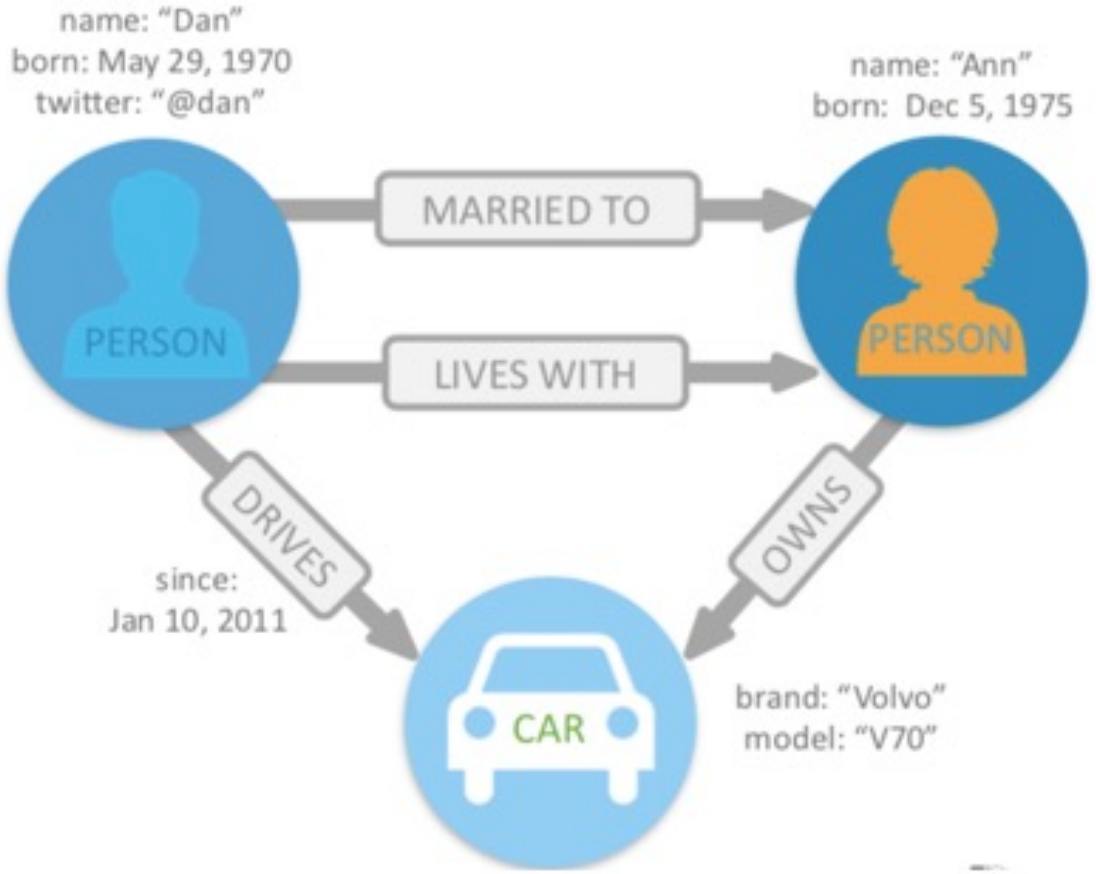
get the spouse of the owner of a Volvo




```
MATCH (:Car {brand:'Volvo'})<-[:OWNS]-
(:Person)-[:MARRIED_TO]-(p:Person)
RETURN p
```

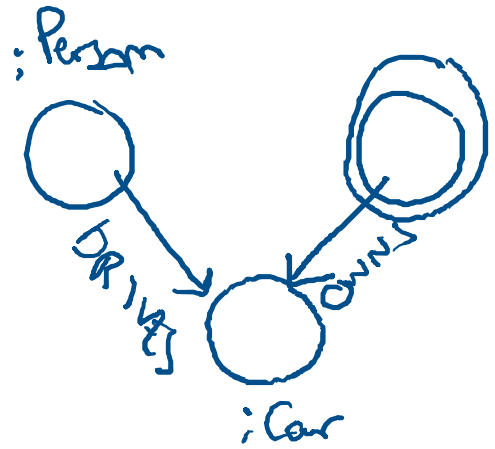
Queries = Traversals

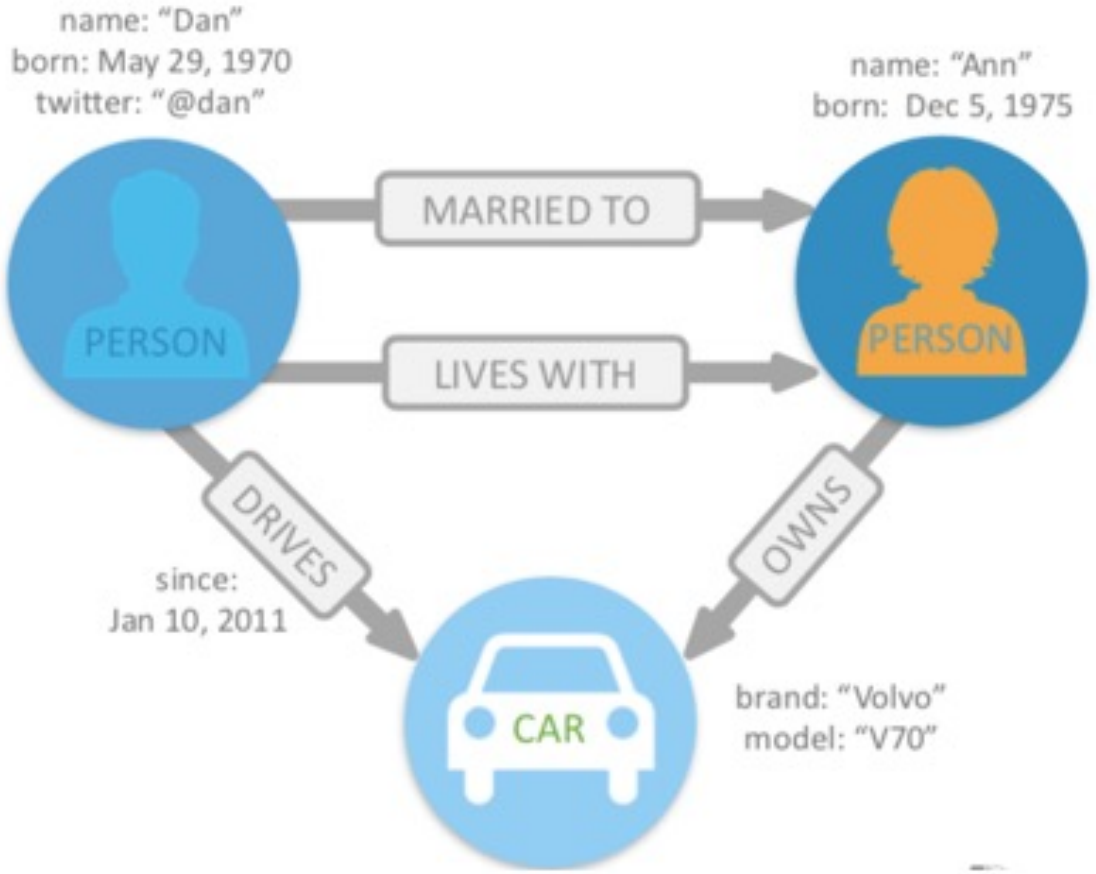
get the owner of a car a person drives



Queries = Traversals

get the owner of a car a person drives



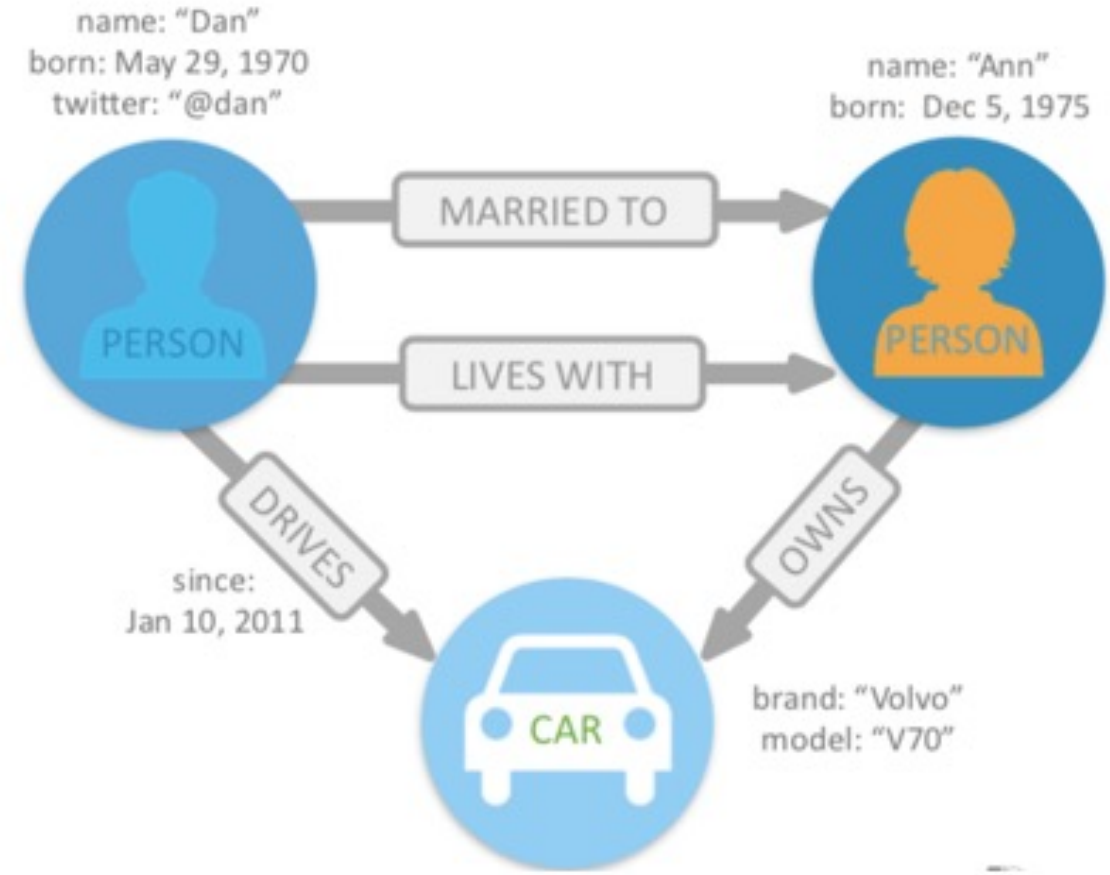



```
MATCH (:Person)-[:DRIVES]->(:Car)<-[:OWNS]-(p:Person)
RETURN p
```

```
MATCH (c:Car)<-[:DRIVES]-(:Person),(p:Person)-[:OWNS]->(c)
MATCH (p:Person)-[:OWNS]->(:Car)<-[:DRIVES]-(:Person)
```

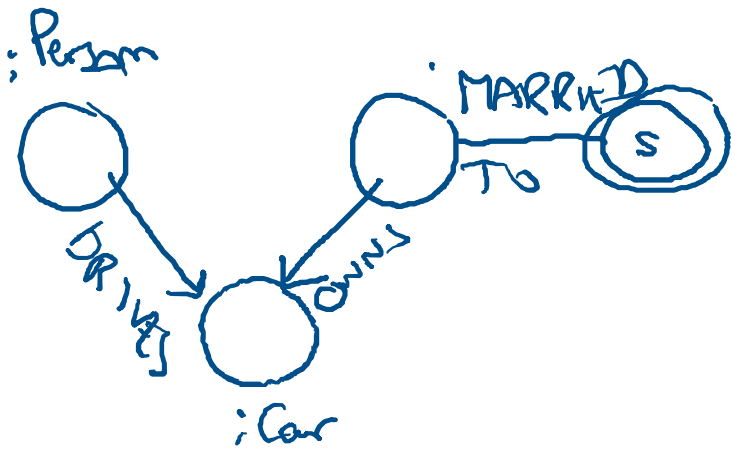
Queries = Traversals

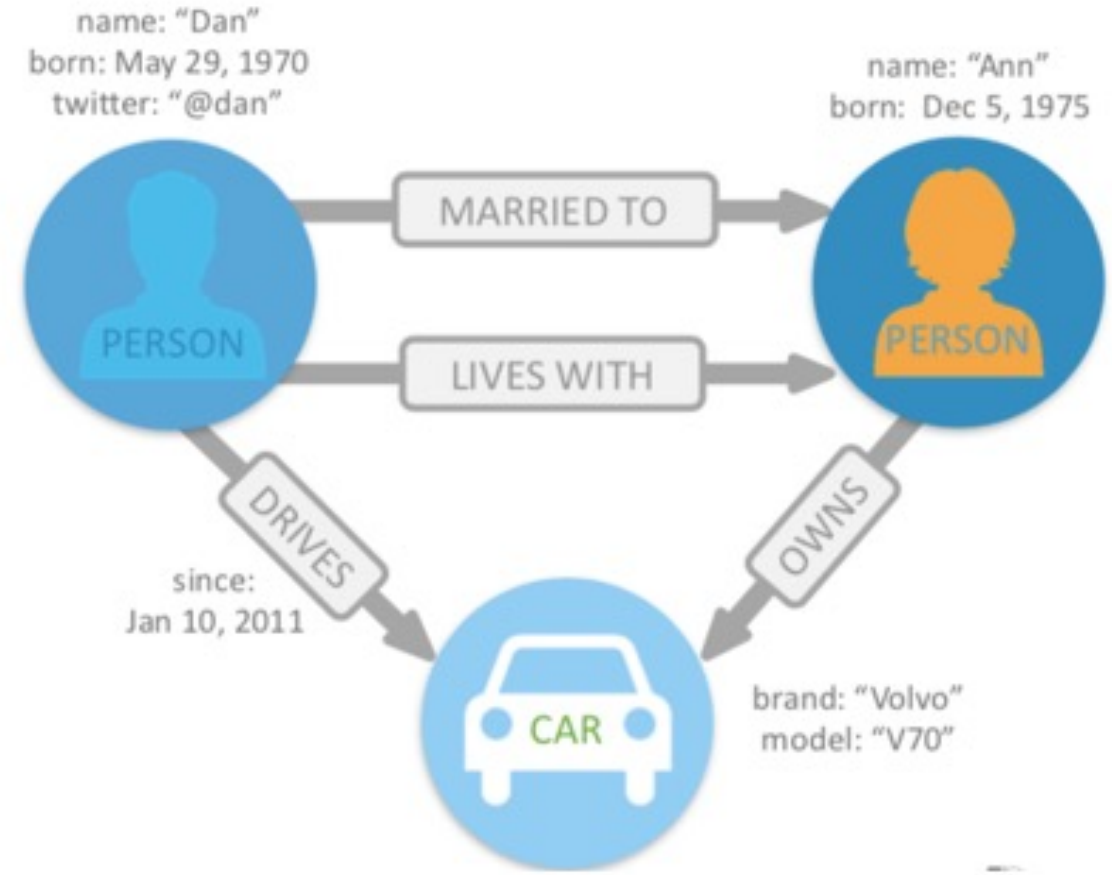
get the spouse of the owner of a car a person drives



Queries = Traversals

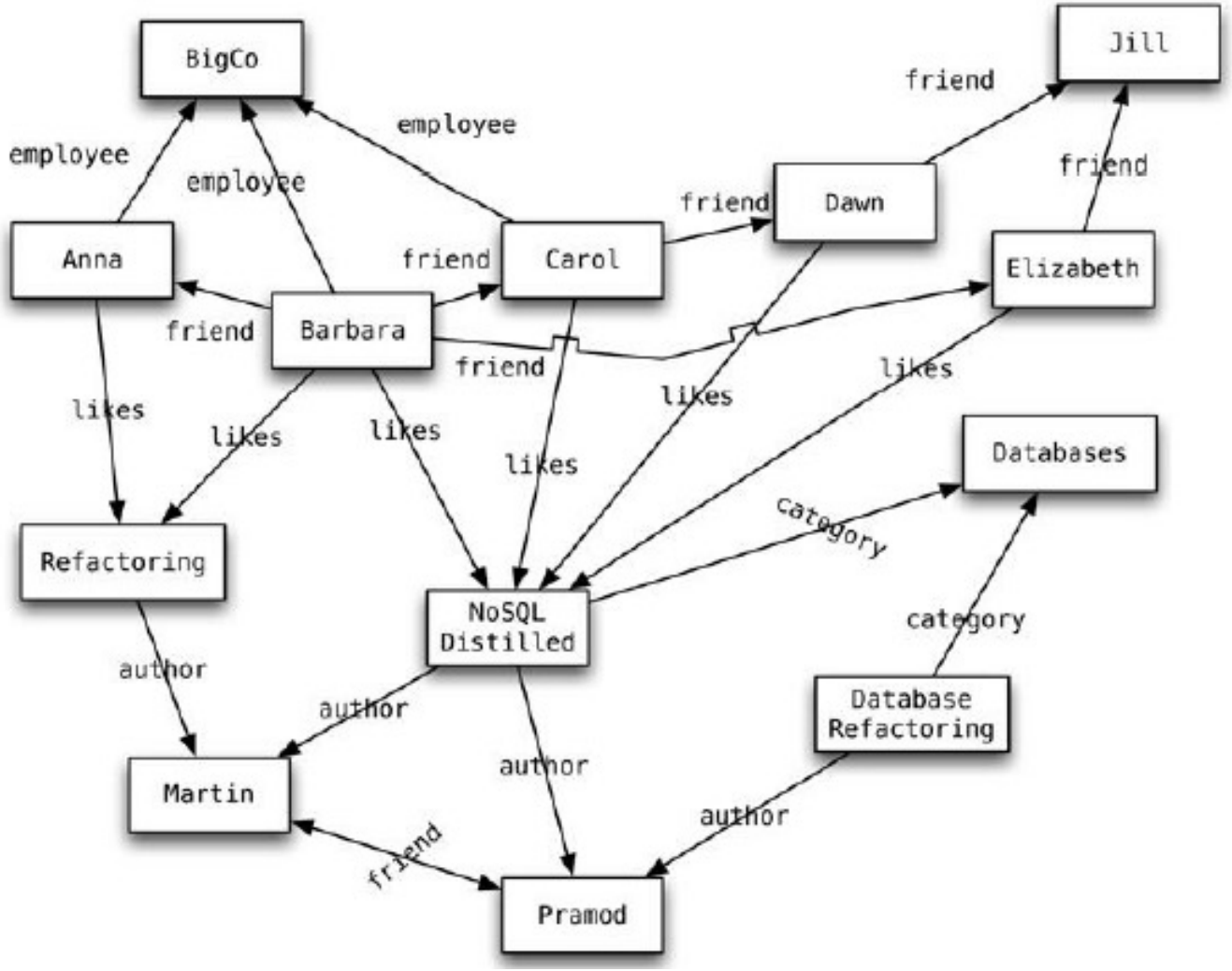
get the spouse of the owner of a car a person drives





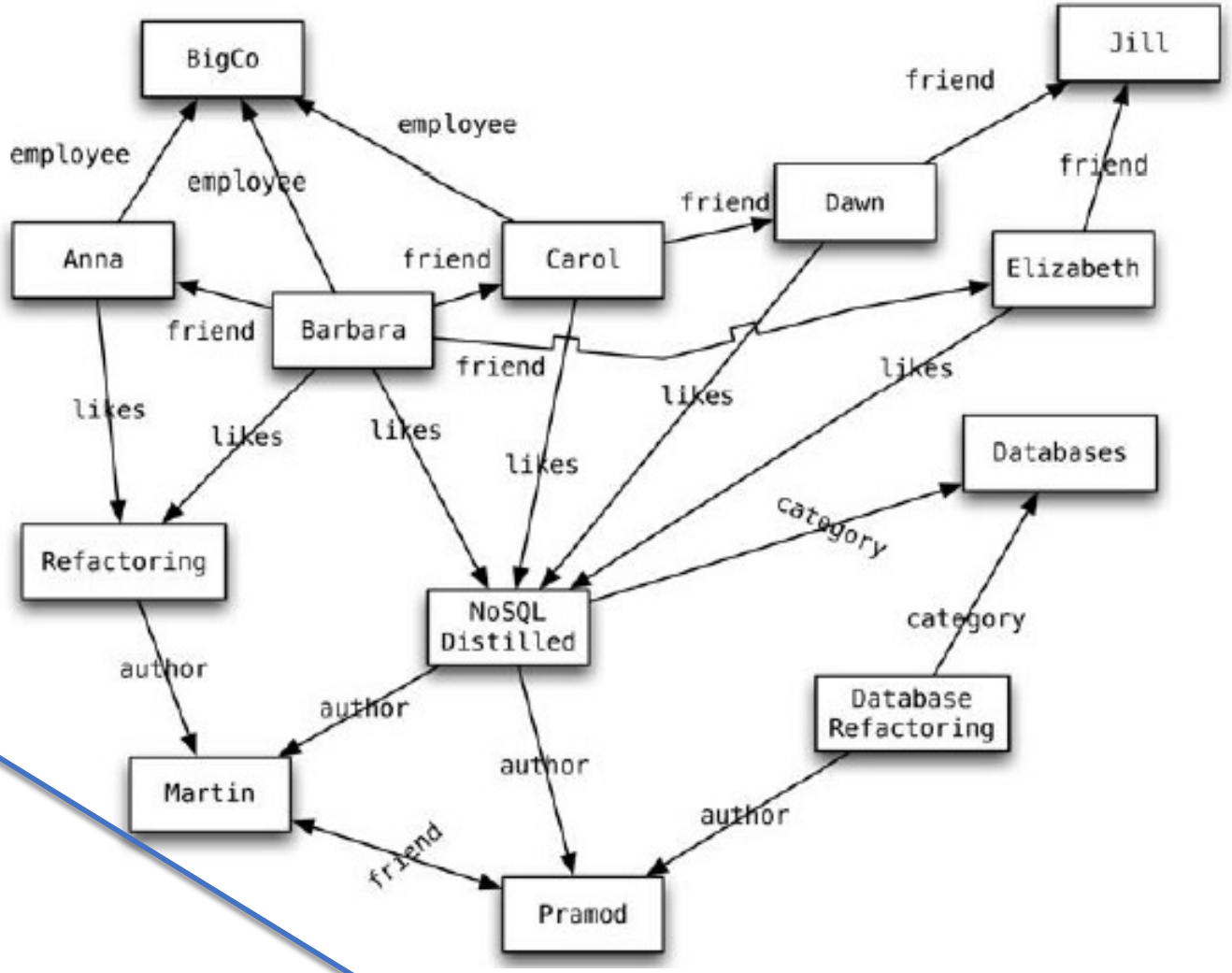

```
MATCH (:Person)-[:DRIVES]->(:Car)<-[:OWNS]-
(:Person)-[:MARRIED_TO]-(s:Person)
RETURN s
```

"find the friends of Barbara that like a node of database category."



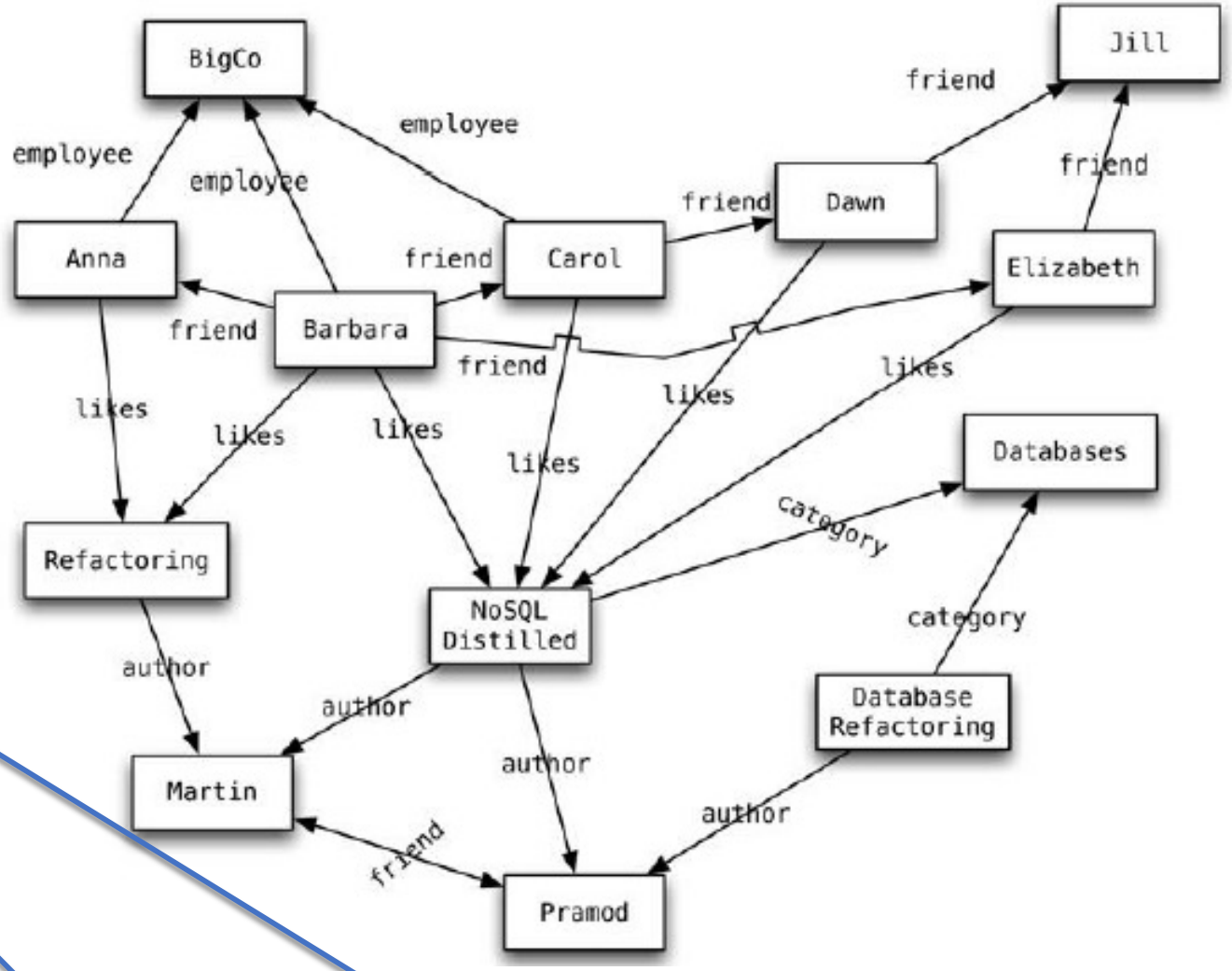
www.wooclap.com/GRAPHDB

"find the friends of Barbara that like a node of database category."



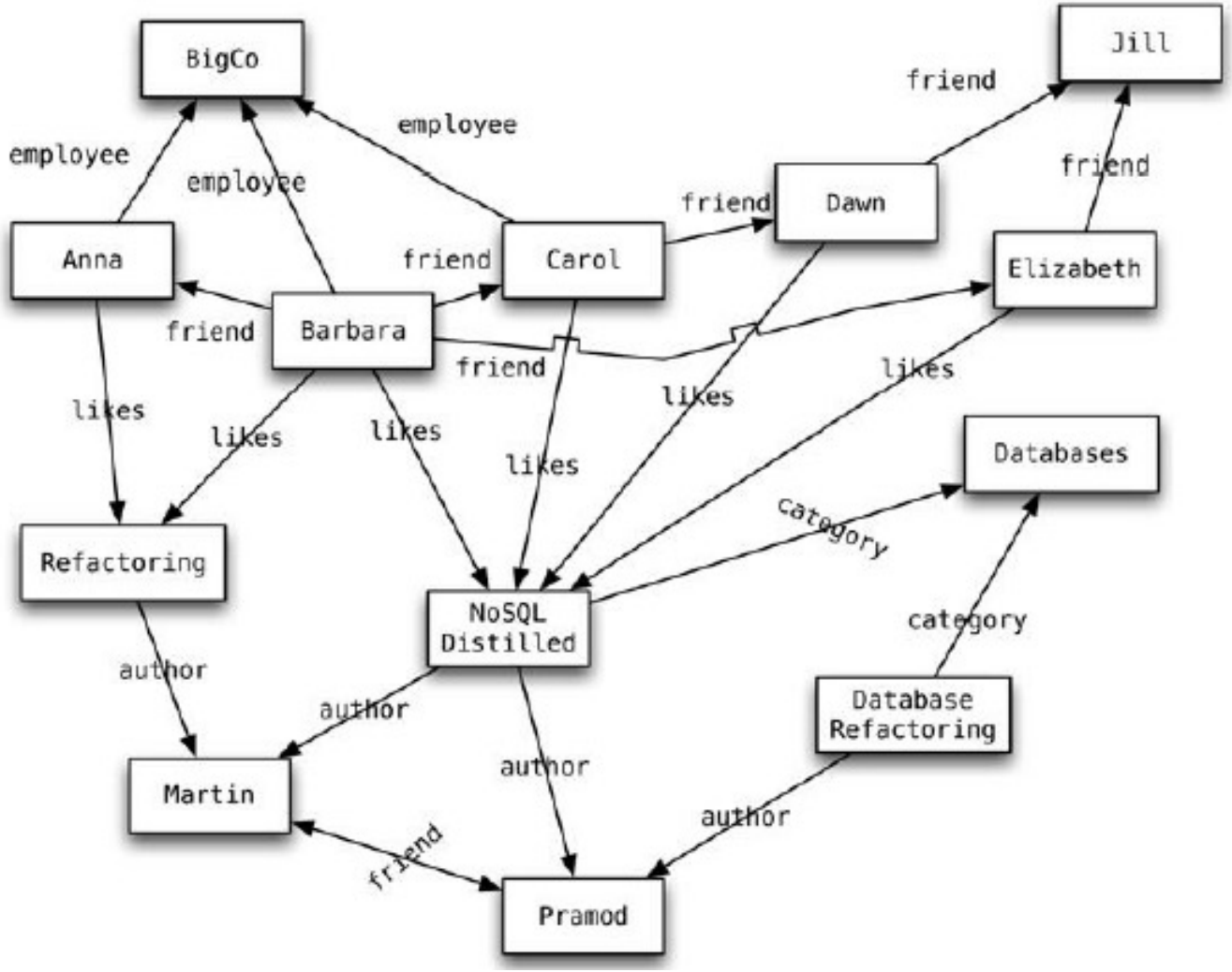
MATCH (:Barbara)-[:friend]-(node)-[:likes]->()- [:category]- > (:Databases)
RETURN node ;

"find the friends of Barbara that like a node of database category."



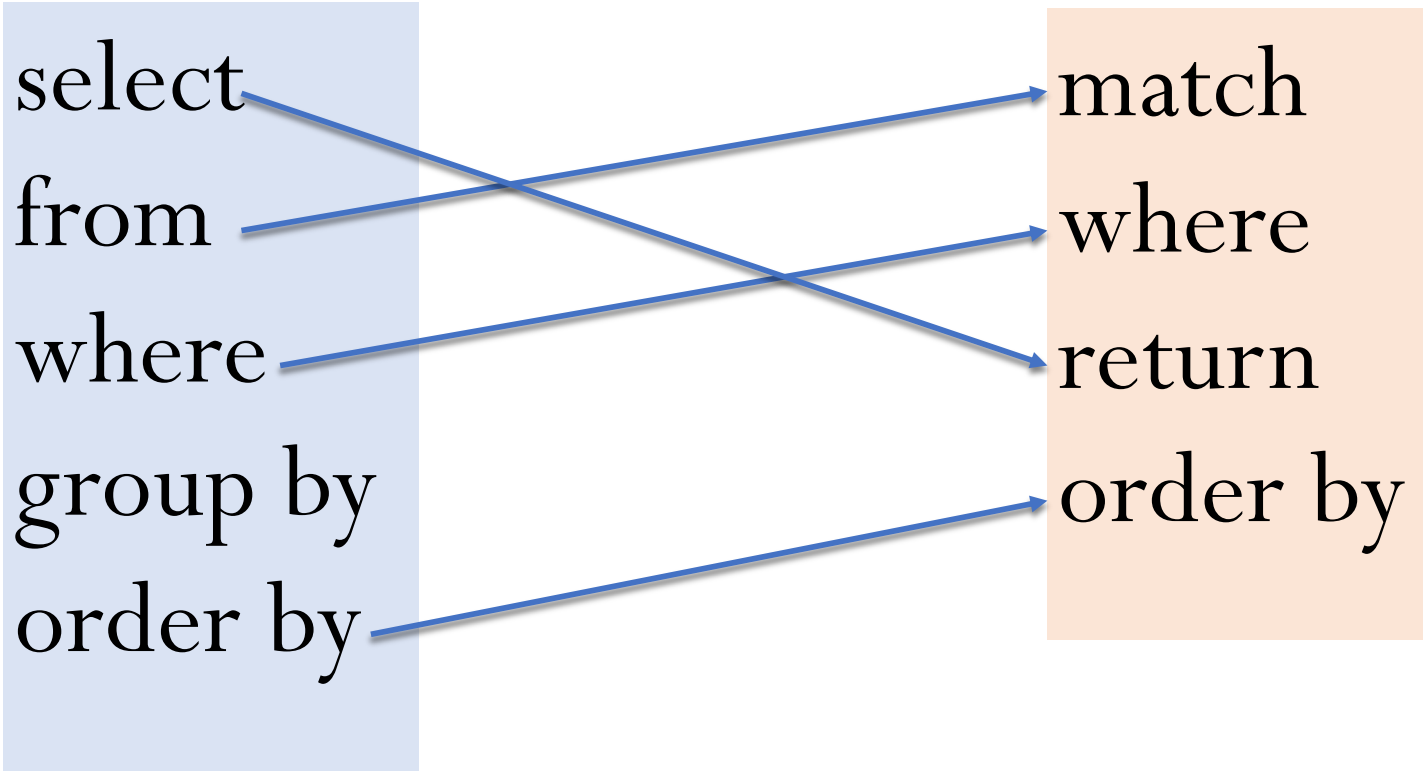
MATCH (:Barbara)-[:friend]-(node)-[:likes]->()- [:category]- > (:Databases)
RETURN node ;

MATCH (:Databases)<-[:category]-()<-[:likes]-(node)- [:friend]- (:Barbara)
RETURN node ;



www.wooclap.com/GRAPHDB. **node labels**

Cypher queries – Structure Familiar for SQL Users



Where

```
MATCH (node:Label {property:value})
MATCH (node:Label)
WHERE property=value
```

- usual big 6 (=, <, >, <=,>=, <>)
- Usual Boolean **(AND, OR, NOT)**
- Property existence checking: exists

```
MATCH (n)
WHERE exists(n.marriedTo)
RETURN n.name, n.marriedTo
```

- Collection membership: **IN**
- …

Returning nodes and values

```
MATCH (:Barbara)-[:friend]-(node)-
[:likes]->()- [:category]- > (:Databases)
RETURN node
```

```
MATCH (n)
WHERE exists(n.marriedTo)
RETURN n.name, n.marriedTo
```

Returning Paths

- `MATCH p=(:Barbara)-[:friend]-(node)-[:likes]->()- [:category]- > (:Databases)`

  `RETURN p`

  all of the nodes and relationships for each path, including all of their properties

- Just nodes in the path:
- `RETURN nodes(p)`

- Just relationships in the path:
- `RETURN rels(p)`

Cypher: Outputs (from yesterday)

A node	`MATCH (p:Person {name:"Tom"}) RETURN p`
A value	`MATCH (p:Person {name:"Tom"}) RETURN p.age`
A list of values	`MATCH (p:Person) RETURN p.name LIMIT 5`
An array	`MATCH p=shortestPath((a)-[*]->(b)) WHERE a.name="Axel" AND b.name="Tom" RETURN p`
A list of arrays	`MATCH p=((a)-[*]->(b)) WHERE a.name="Axel" AND b.name="Frank" RETURN p`

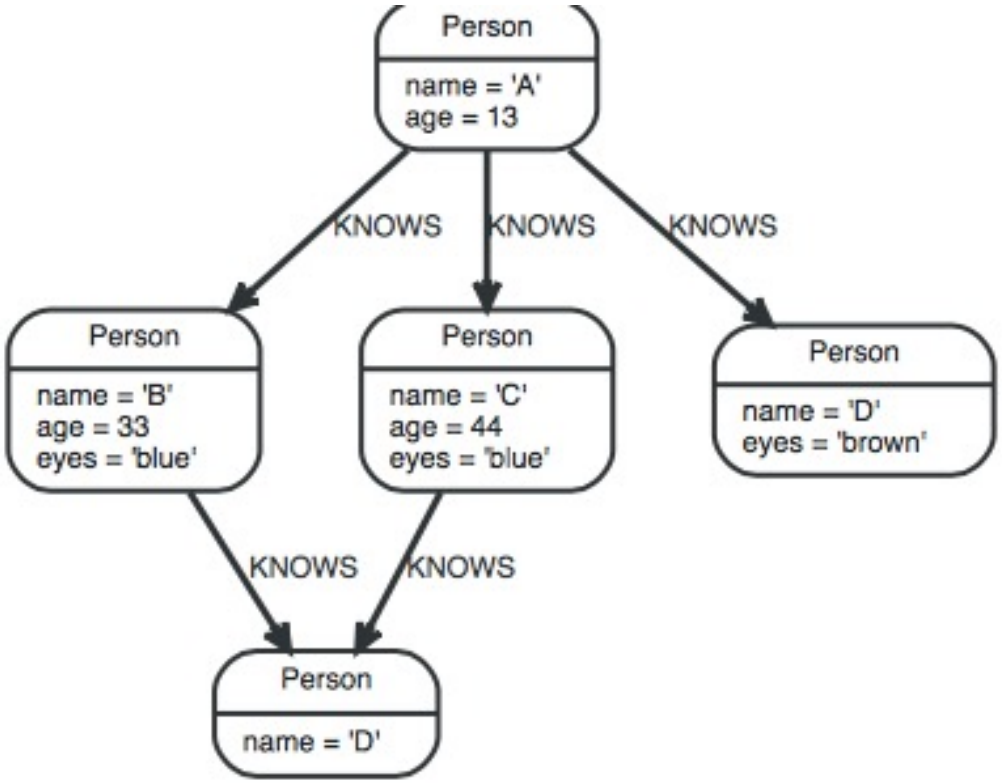
Return – aggregate functions

- MATCH (n:Person)
  RETURN avg(n.age)

  30

- MATCH (n:Person)
  RETURN collect(n.age)

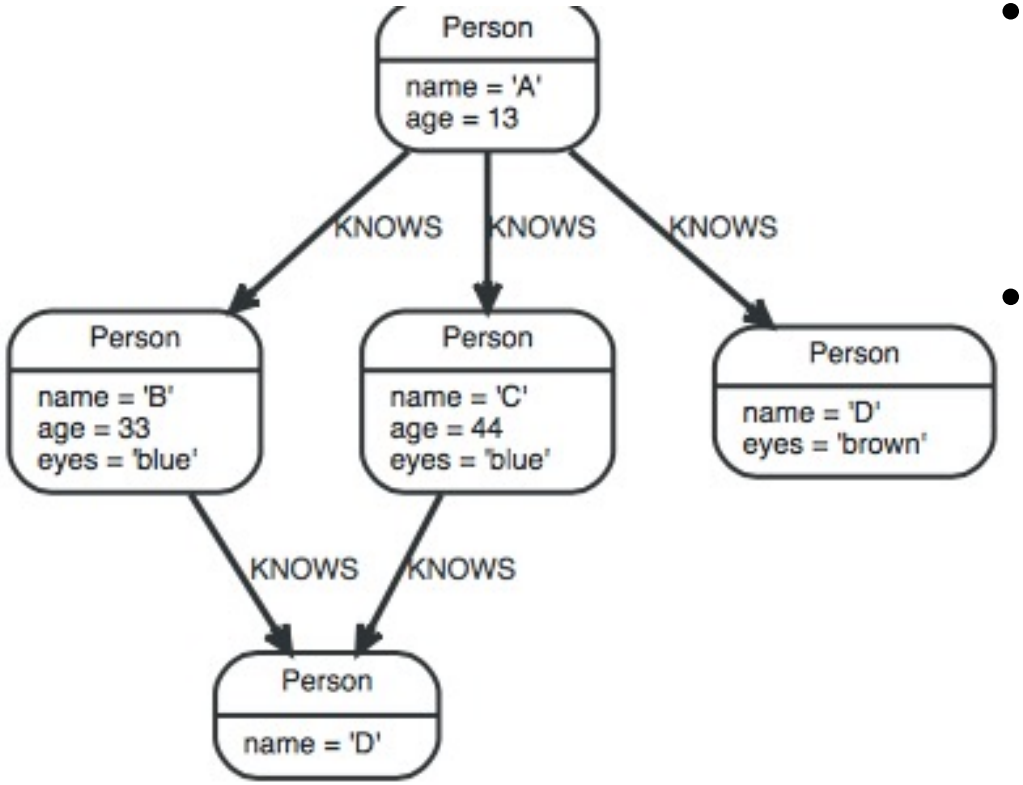
  (13, 33, 44)



avg(), collect(), count(), max(), min(), sum(), …

GROUP BY

- In Cypher, GROUP BY is done implicitly by all of the aggregate functions
- In a RETURN clause (as in a WITH, see next slides), any columns not part of an aggregate will be the GROUP BY key



- MATCH (n:Person)

  RETURN COUNT(n), n.eyes

  2,'blue' 1,'brown'

- MATCH (n:Person)

  RETURN COUNT(n), n.eyes, AVG(age)

  2,'blue',38.5 1,'brown'

GROUP BY – how many friends?

```
MATCH (p:Person)-[:Knows]->(friend)

WHERE p.age = 20

RETURN p.name, count(friend)
```

What if we only want people with at least 10 friends?
(a sort of SQL HAVING)
We need a WITH clause …

WITH – to divide a query into multiple parts

```
MATCH (p:Person)-[:Knows]->(friend)

WHERE p.age = 20

WITH p, count(friend) as friends

WHERE friends > 10

RETURN p.name, friends
```

Ex: Reformulate the 3 users queries

Cumulative

```
reduce(accumulator = initial,
variable IN list | expression)
```

1. iterate through each element *e* in the given list
2. run the expression on *e* and
3. store the new partial result in the accumulator

```
MATCH p =(a:{name:''Alice'')-[:KNOWS*..3]->
         (c: {name:''Bob''})
RETURN reduce(totalAge = 0, n IN nodes(p)|
totalAge + n.age) AS reduction
```

A few more Cypher queries

Return the pairs of nodes that are related by two different relationships

```
MATCH(n1)-[r1]-(n2)-[r2]-(n1) WHERE type(r1)<>type(r2)
RETURN n1, n2
```

Return the three nodes with the highest number of relationships.

```
MATCH (a)-[r]->()
RETURN a, COUNT(r) ORDER BY COUNT(r) DESC LIMIT 3
```

Return the length of the shortest path between two nodes of your choice.

```
MATCH p = shortestPath((p1:Person{name:'Alice'})-[*]-
                       (p2:Person{name:'Bob'}))
RETURN length(p)
```

Cypher: Clauses

- **MATCH**: The graph pattern to match
- **WHERE**: Filtering criteria
- **RETURN**: What to return
- **CREATE**: Creates nodes and relationships
- **DELETE**: Remove nodes, relationships, properties
- **SET**: Set values to properties
- **WITH**: Divides a query into multiple parts

Cypher vs SQL

```
 (SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.pid AS directReportees, 0 AS count
   FROM person_reportee manager
   WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
   SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
```

```
SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
   FROM(
   SELECT reportee.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages) AS
count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

Cypher vs SQL

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = “John Doe”
RETURN sub.name AS Subordinate,
count(report) AS Total;
```

Cypher vs SQL

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.pid AS directReportees, 0 AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
  SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages)
AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = "John Doe"
RETURN sub.name AS Subordinate,
  count(report) AS Total
```

Less time writing queries

- More time understanding the answers
- Leaving time to ask the next question

Less time debugging queries:

- More time writing the next piece of code
- Improved quality of overall code base

Code that's easier to read:

- Faster ramp-up for new project members
- Improved maintainability & troubleshooting

openCypher

Indexes and Query Tuning

- Indexes can be added to improve search performance

```
CREATE INDEX [index_name]
FOR (n:LabelName)
ON (n.propertyName)
```

  - Different indexes, recommended BTREE

- Composite (multiproperty) indexes can be defined as well
- Full-text indexes can be defined as well (Apache Lucene)

- The use of indexes by the query optimizer can be checked by looking at the query plan (EXPLAIN) or suggested by USING

https://neo4j.com/docs/cypher-manual/current/administration/indexes-for-search-performance/

Other querying mechanisms

- Java API
- Graph algorithm library

Finds the optimal path or evaluates route availability and quality

Determines the importance of distinct nodes in the network

Evaluates how a group is clustered or partitioned