Bases de Dados de Grafos

AULA PL07

Hugo Peixoto

2019 – 2020 Universidade do Minho

Agenda

Introduction to Graph Databases

neo4j

Use Cases

Cypher

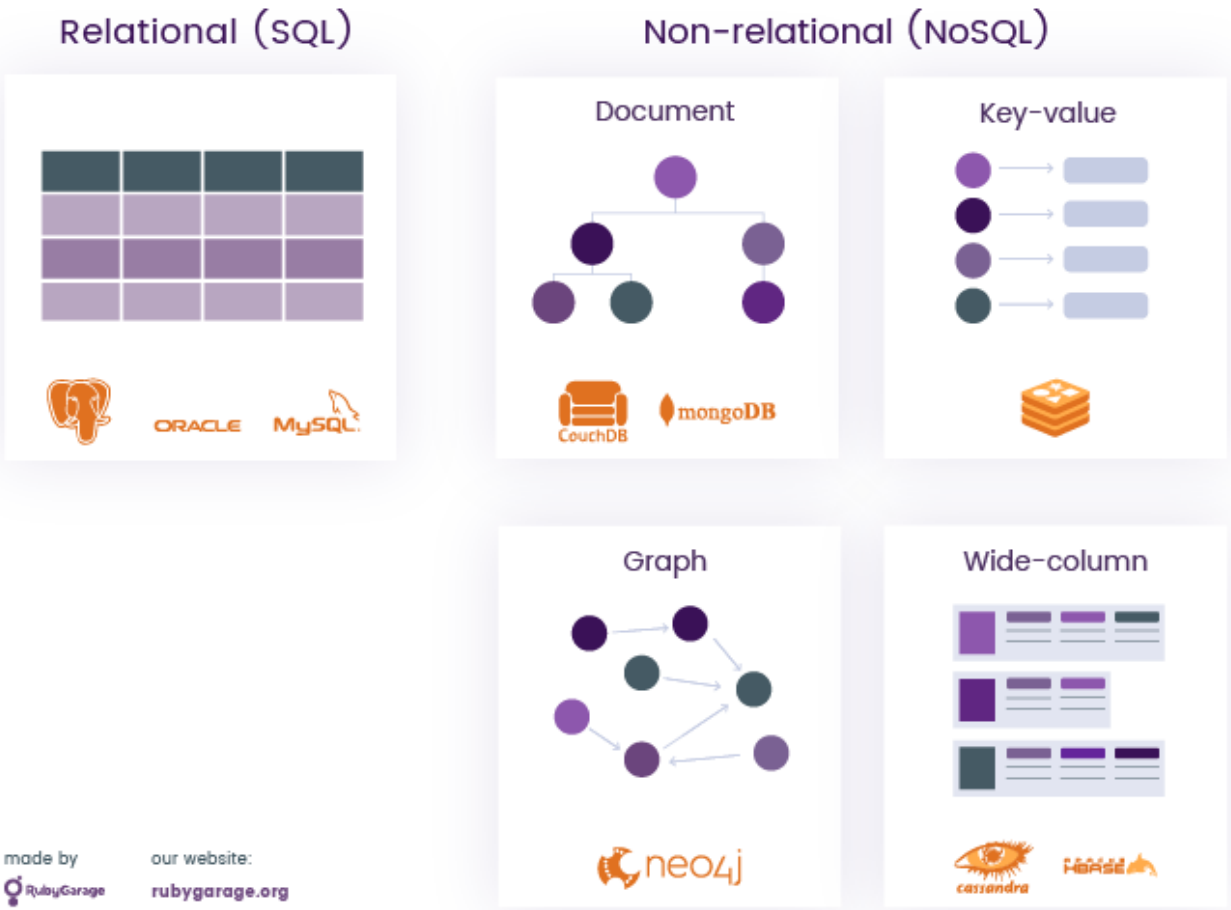
Lab

FE07

https://docs.mongodb.com/manual/


neo4j

Types of Databases



What is a Graph

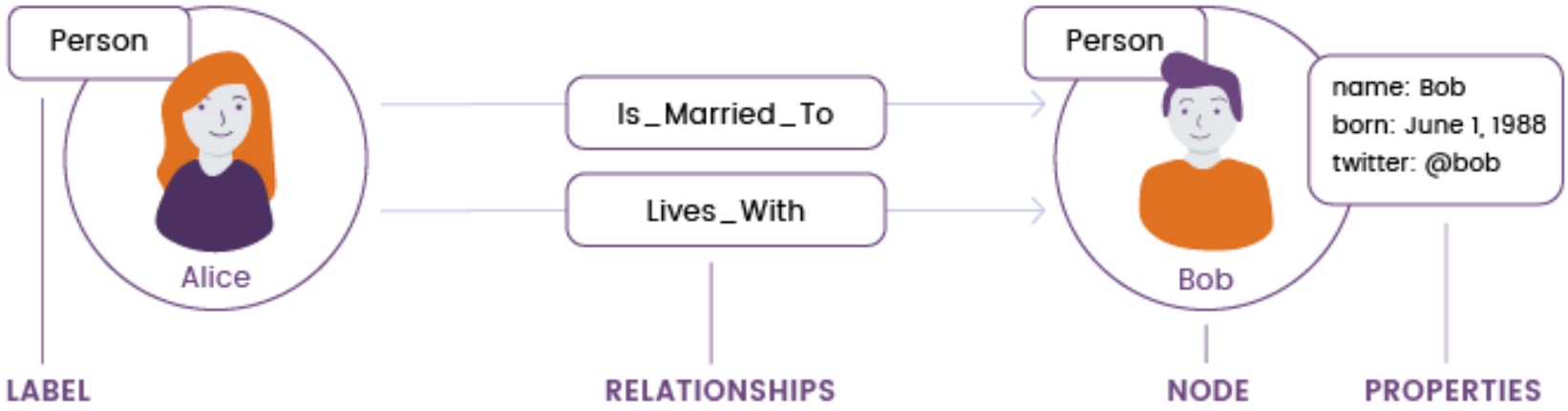
Formally, a graph is just a collection of *vertices* and *edges.*

A set of nodes and the relationships that connect them

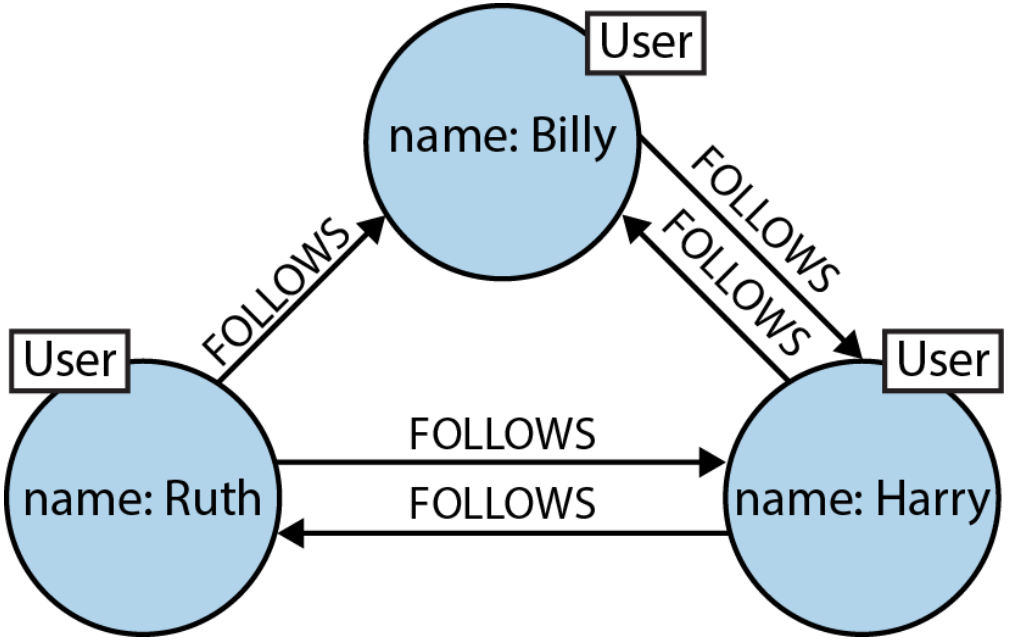
Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships.

What is a Graph

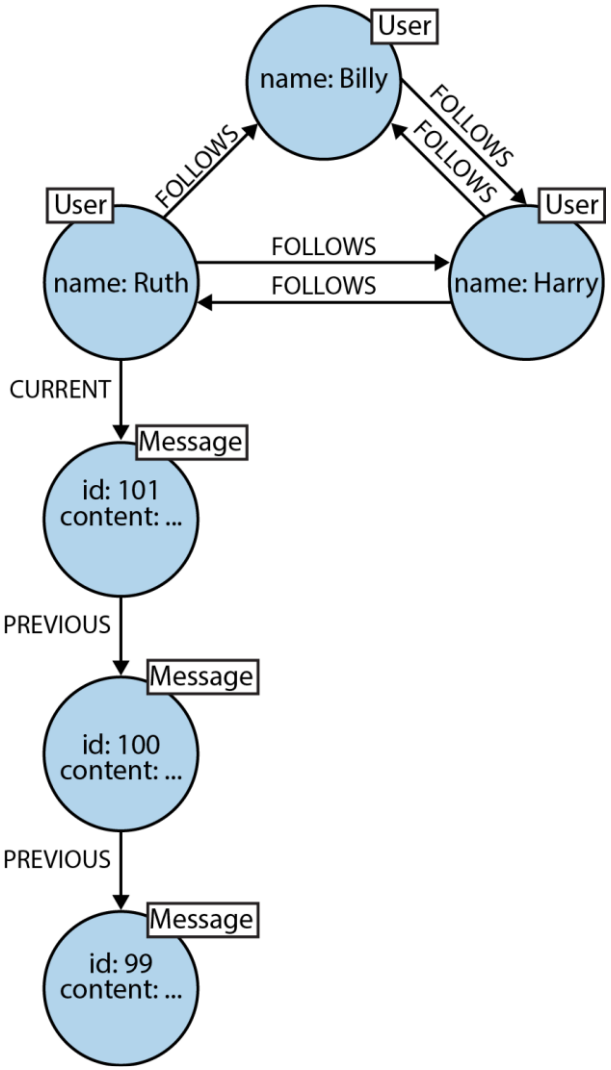
A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way



Network of Twitter users



Twitter timeline



Property Graph Model

- It contains nodes and relationships.
- Nodes contain properties (key-value pairs).
- Nodes can be labeled with one or more labels.
- Relationships are named and directed, and always have a start and end node.
- Relationships can also contain properties.

Property Graph Model - Components

- **Nodes** (equivalent to vertices in graph theory). These are the main data elements that are interconnected through relationships. A node can have one or more labels (that describe its role) and properties (i.e. attributes).

- **Relationships** (equivalent to edges in graph theory). A relationship connects two nodes that, in turn, can have multiple relationships. Relationships can have one or more properties.

- **Labels**. These are used to group nodes, and each node can be assigned multiple labels. Labels are indexed to speed up finding nodes in a graph.

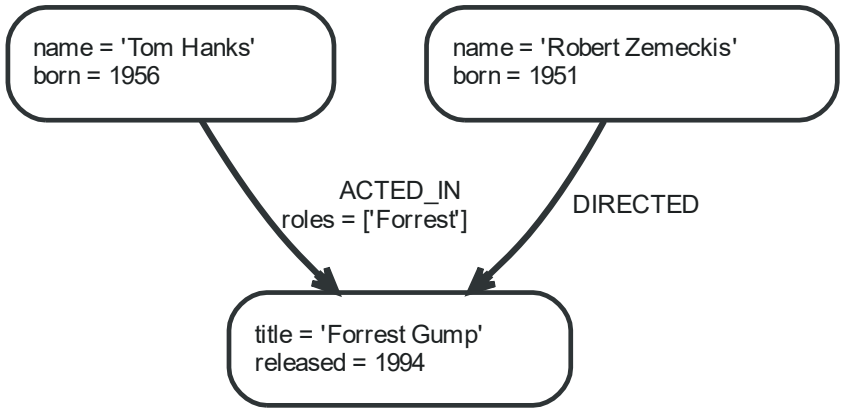
- **Properties**. These are attributes of both nodes and relationships. Neo4j allows for storing data as key-value pairs, which means properties can have any value (string, number, or boolean).

Nodes

A node in Neo4j is a node as described in the property graph model, with properties and labels.

Relationships

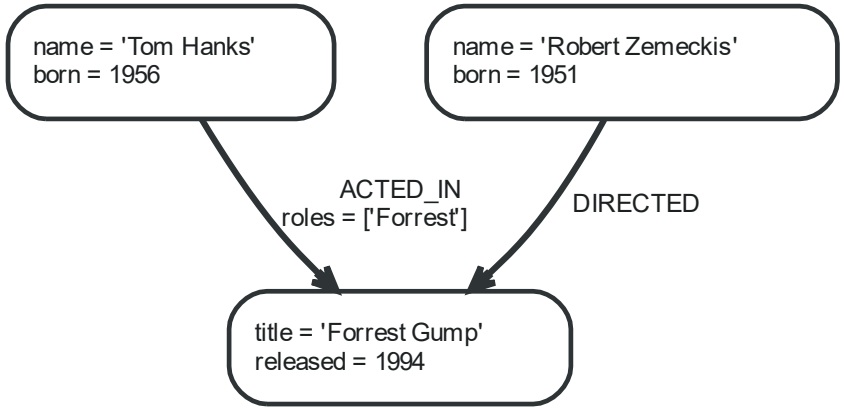
A relationship in Neo4j is a relationship as described in the property graph model, with a relationship type and properties.



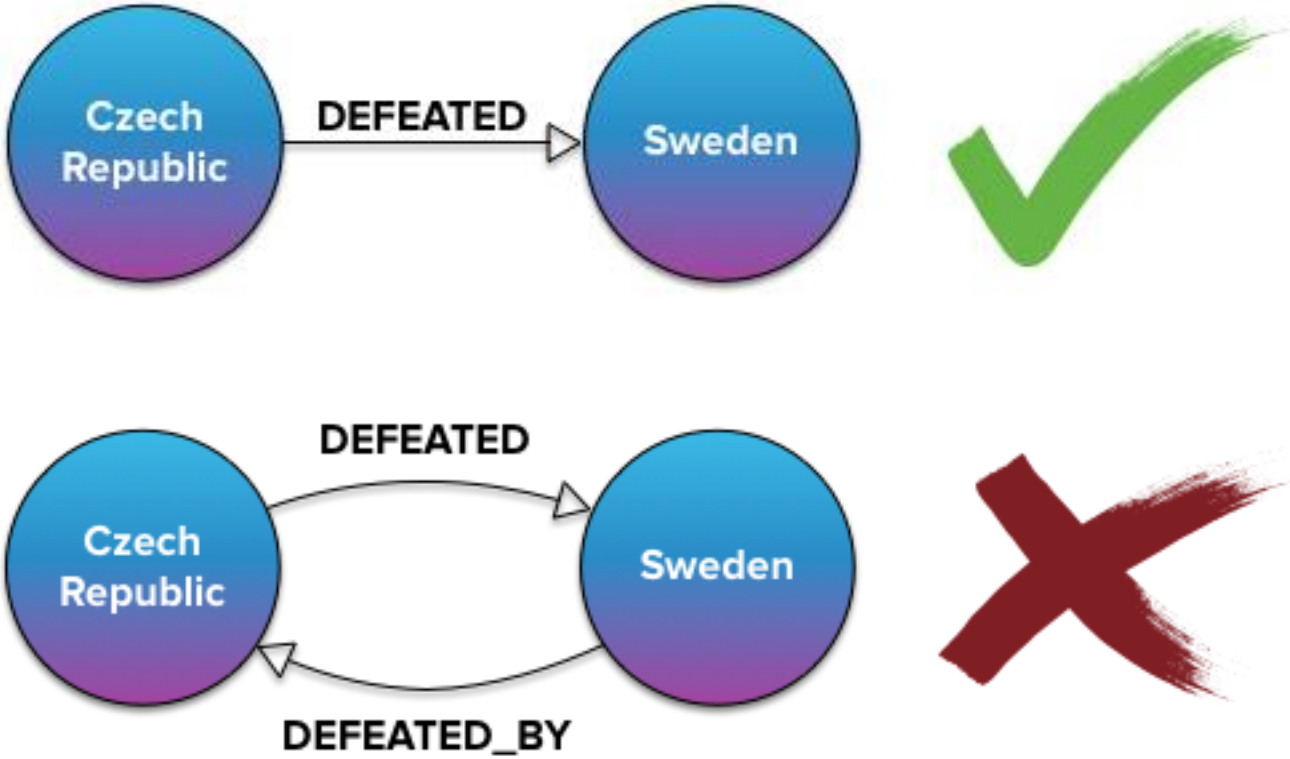
Relationships between nodes are the key feature of graph databases, as they allow for finding related data.

Relationships

ACTED_IN and DIRECTED are relationship **types**. The roles **property** on the ACTED_IN relationship has an array value with a single item in it.



Relationships



Properties

A property in Neo4j is a property as described in the property graph model. Both nodes and relationships may have properties.

Types:

Number, an abstract type, which has the following subtypes:
- Integer
- Float

String

Boolean

Spatial types:
- Point

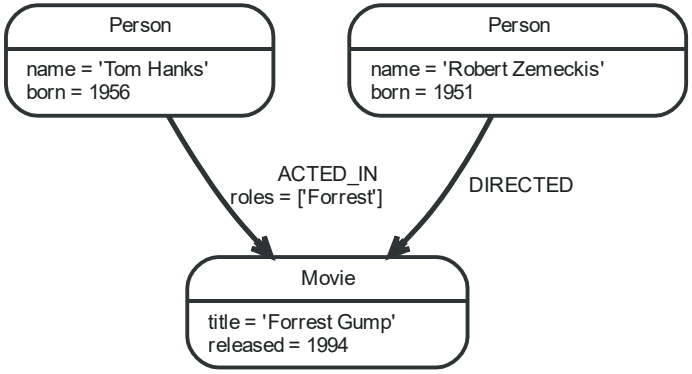
Temporal types:
- Date
- Time
- LocalTime
- DateTime
- LocalDateTime
- Duration

null is not a valid property value. Instead of storing it in the database, null can be modeled by the absence of a property key.

Labels

A label in Neo4j is a label as described in the property graph model. Labels assign roles or types to nodes.

A label is a named graph construct that is used to group nodes into sets; all nodes labeled with the same label belongs to the same set.



Advantages

- **Performance -** In relational databases, performance suffers as the number and depth of relationships increases. In graph databases, performance remains high even if the amount of data grows significantly.

- **Flexibility -** As the structure and schema of a graph model can be easily adjusted to the changes in an application. Also, you can easily upgrade the data structure without damaging existing functionality.

- **Agility -** The structure of a gaph database is easy-to-upgrade, so the data store can evolve along with your application.

Lack of relationships - Relational DB

Relationships do exist in the vernacular of relational databases, but **only at modeling time**, as a means of **joining tables**;

As data multiplies, and the overall structure of the dataset becomes more complex and less uniform, the relational model becomes burdened with large join tables, sparsely populated rows, and lots of null-checking logic.

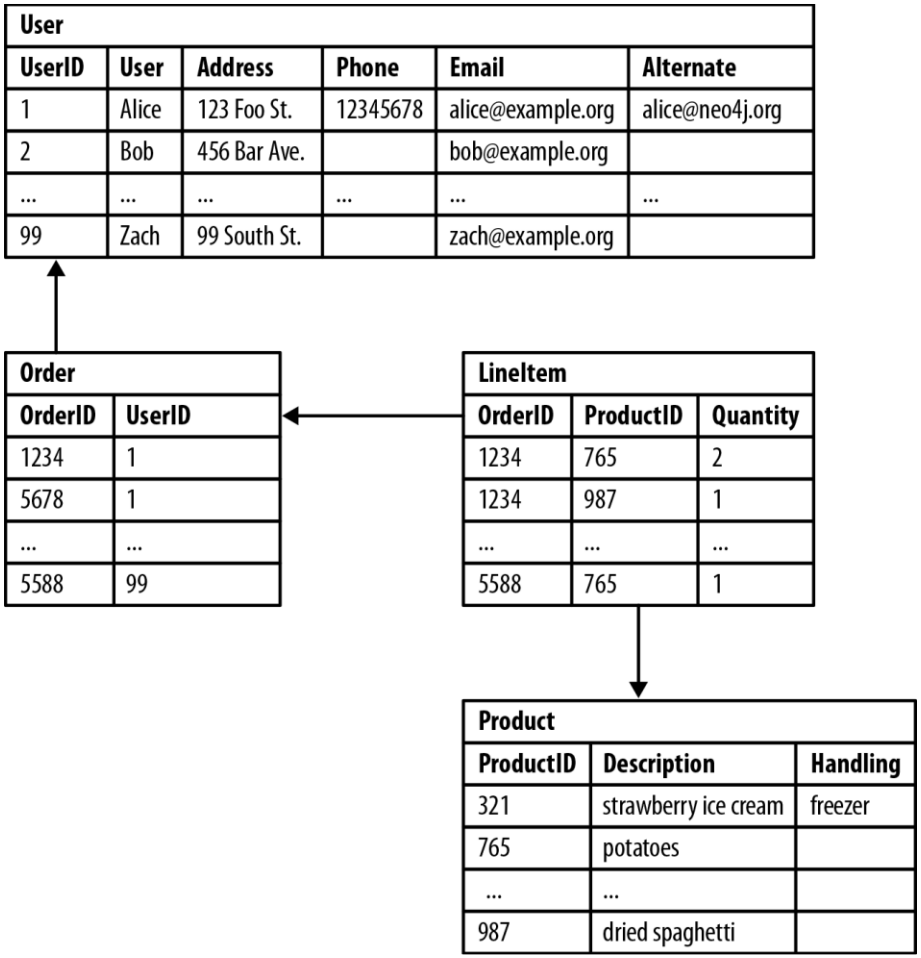
Lack of relationships – Relational DB

Join tables add accidental complexity; they mix business data with foreign key metadata.

Foreign key constraints **add additional development** and maintenance overhead just to make the database work.

Sparse tables with **nullable columns require special checking in code**, despite the presence of a schema.

Lack of relationships – Relational DB



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

Lack of relationships – Relational DB

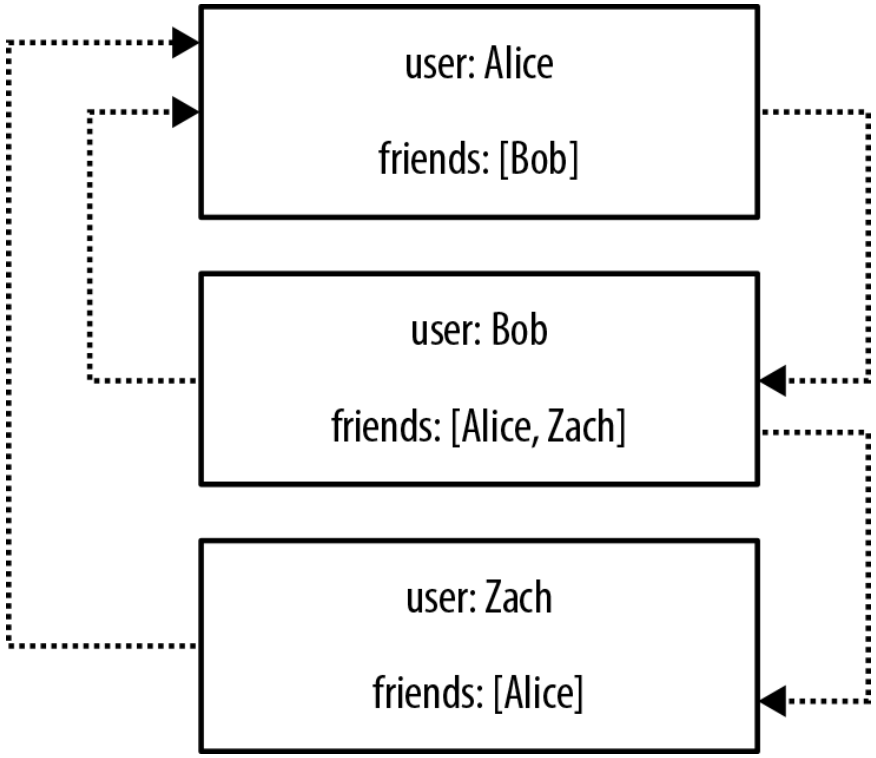
Several **expensive** joins are needed just to discover what a customer bought.

Reciprocal queries are even more costly.

“What products did a customer buy?” is relatively cheap compared to “which customers bought this product?”, which is the basis of recommendation systems.

Lack of relationships – NoSQL DB

“Who are Bob friends?”

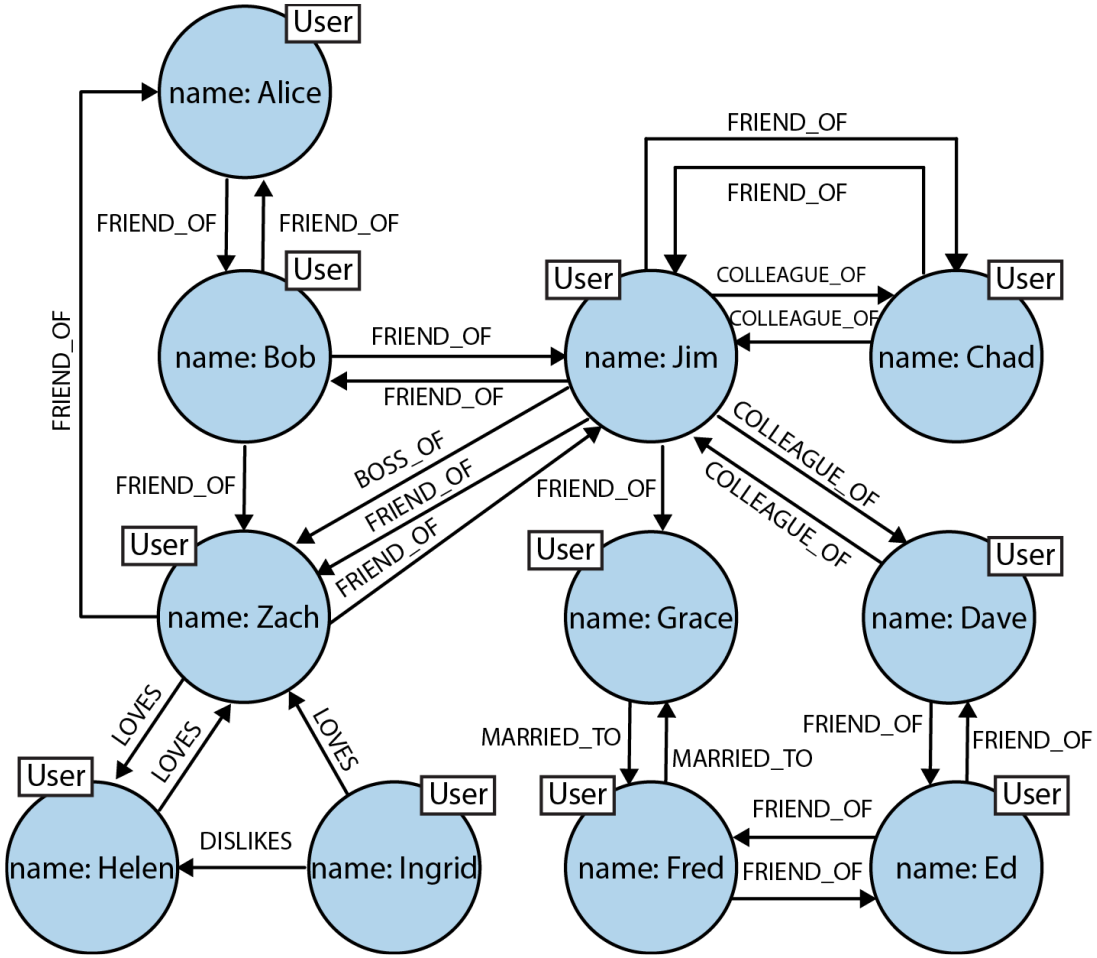


Not always symmetric.

“Who is friends with Bob?”

brute-force scan across the whole dataset looking for friends' entries that contain Bob.

Relationships – Graph DB



Comparison

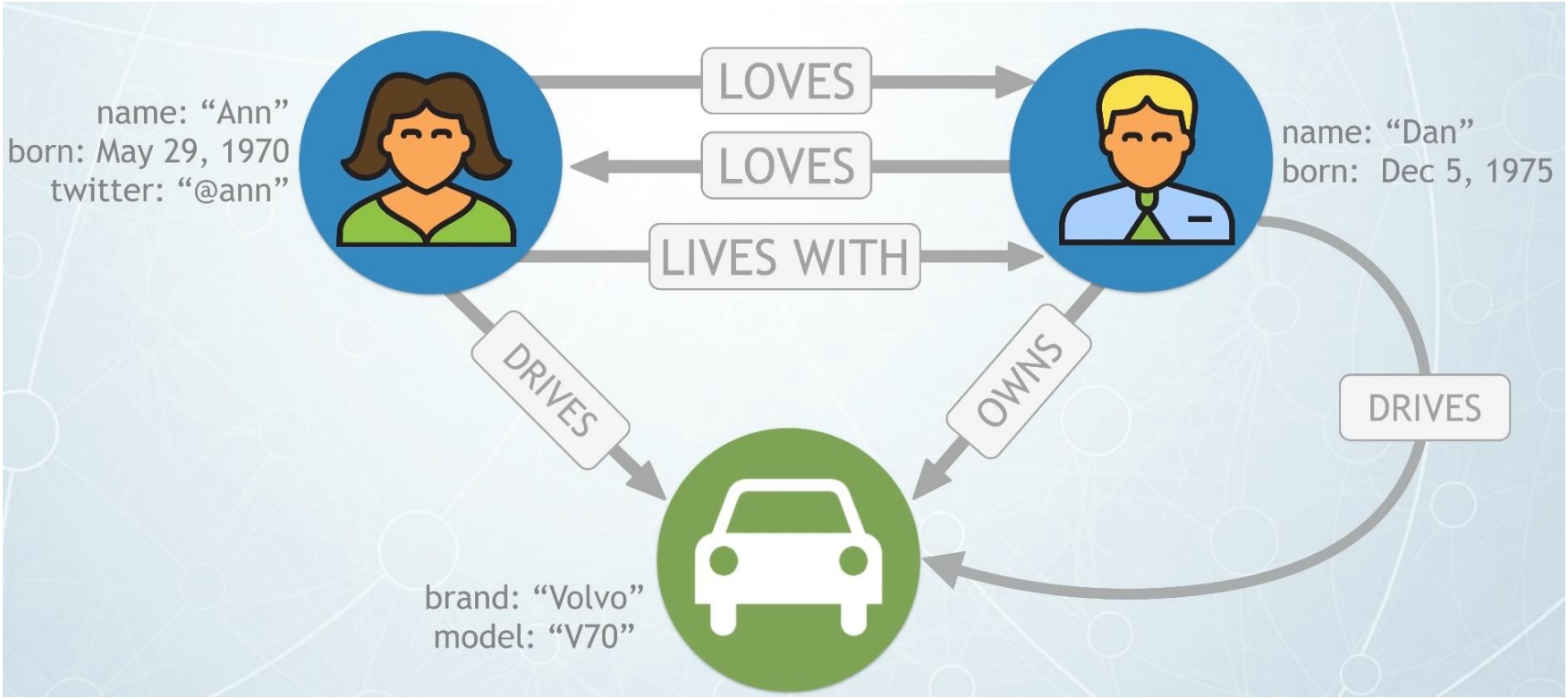
	Neo4j	Relational databases	NoSQL databases
Data storage	Graph storage structure	Fixed, predefined tables with rows and columns	Connected data not supported at the database level
Data modeling	Flexible data model	Database model must be developed from a logical model	Not suitable for enterprise architectures
Query performance	Great performance regardless of number and depth of connections	Data processing speed slows with growing number of joins	Relationships must be created at the application level
Query language	Cypher: native graph query language	SQL: complexity grows as the number of joins increases	Different languages are used but none is tailored to express relationships
Transaction support	Retains ACID transactions	ACID transaction support	BASE transactions prove unreliable for data relationships
Processing at scale	Inherently scalable for pattern-based queries	Scales through replication, but it's costly	Scalable, but data integrity isn't trustworthy

Use Cases

- Fraud detection and analytics
- Network and database infrastructure monitoring
- Recommendation engines
- Social networks
- Identity and access management

CYPHER, THE GRAPH QUERY LANGUAGE

Detailed Property Graph



Who drives a car owned by a lover?

```
MATCH
    (p1:Person)-[:DRIVES]->(c:Car)-[:OWNED_BY]->(p2:Person)<-
[:Loves]-(p1)
RETURN
   p1
```

Components of a Cypher Query

```
MATCH (p:Person)-[:ACTED_IN]->(:Movie)
RETURN p
```

MATCH and RETURN are Cypher keywords

p is a variable

:Movie is a node label

:ACTED_IN is a relationship

AsciiArt for Nodes

Nodes are surrounded by parenthesis

```
()  or (p)
```

Labels, or tags, start with : and group nodes by roles or types

```
(p:Person:Mammal)
```

Nodes can have properties

```
(p:Person {name : 'Veronica'})
```

AsciiArt for Nodes

```
()
(matrix)
 (:Movie)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

() represents an anonymous, uncharacterized node. To refer to the node elsewhere, a variable can be added, for example: (matrix). A variable is restricted to a single statement.

The Movie label (prefixed in use with a colon) declares the node's type. This restricts the pattern, keeping it from matching (say) a structure with an Actor node in this position.

AsciiArt for Relationships

Relationships are wrapped with hyphens or square braquets

```
--> or  - [h:HIRED] ->
```

Direction of the relationship is specified with <>

```
(p1) - [:HIRED] -> (p2)  or (p1) <- [:HIRED] - (p2)
```

Relationships have properties too

```
- [:HIRED {type: 'full-time'}] ->
```

What are those?

Relationships are wrapped with hyphens or square braquets

```
--> or  - [h:HIRED] ->
```

Direction of the relationship is specified with <>

```
(p1) - [:HIRED] -> (p2)  or (p1) <- [:HIRED] - (p2)
```

Relationships have properties too

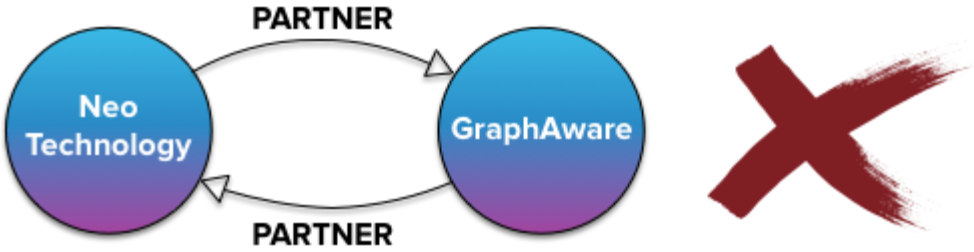
```
- [:HIRED {type: 'full-time'}] ->
```

AsciiArt for Relationships

```
-->
-[role]->
-[:ACTED_IN]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Cypher uses a pair of dashes (--) to represent an undirected relationship. Directed relationships have an arrowhead at one end (<--, -->). Bracketed expressions ([…]) can be used to add details

AsciiArt for Relationships

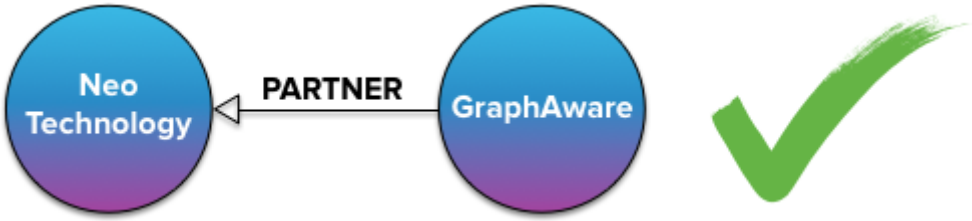


MATCH (neo)-[:PARTNER]-(partner)

The result would be the same as executing and merging the results of the following two different queries:

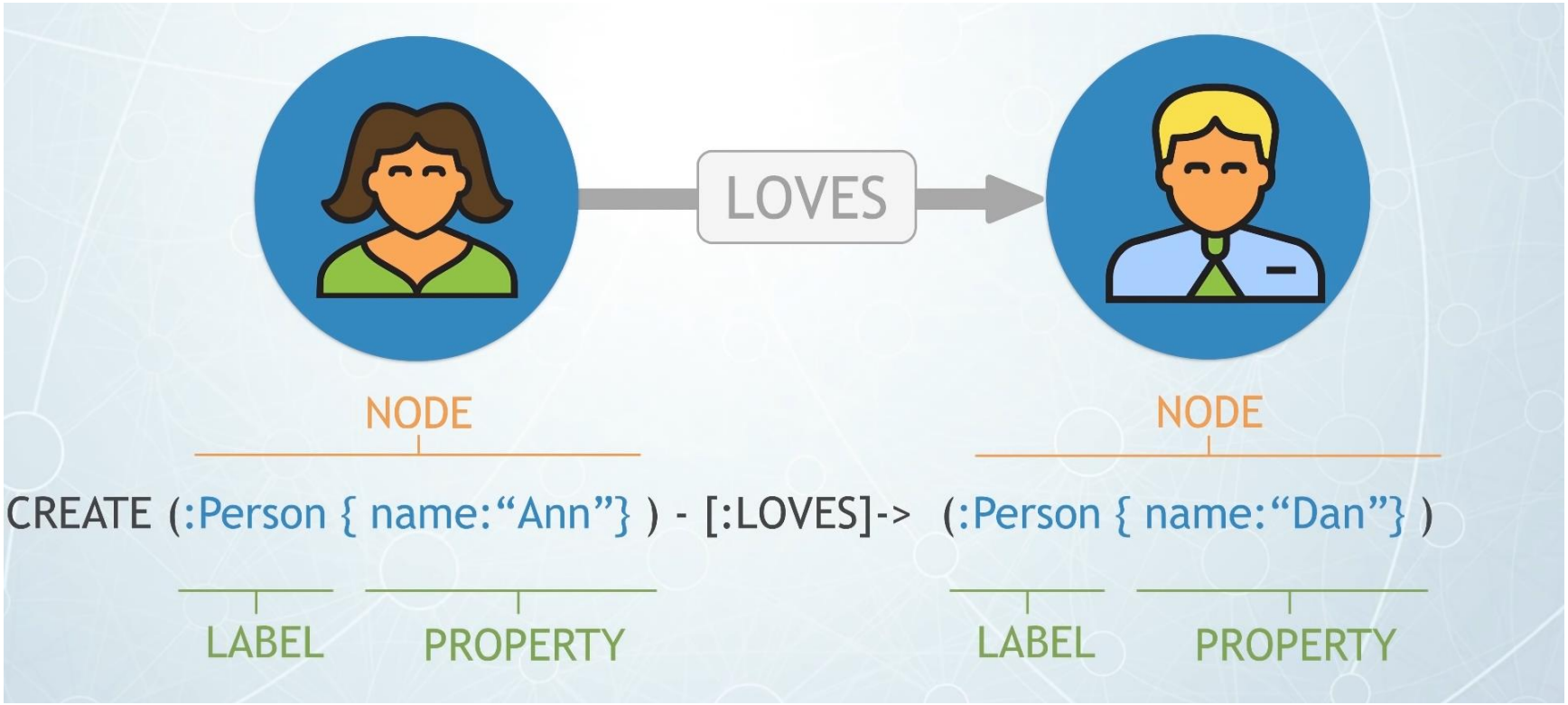
MATCH (neo)-[:PARTNER]->(partner) and MATCH (neo)<-[:PARTNER]-(partner)

AsciiArt for Relationships

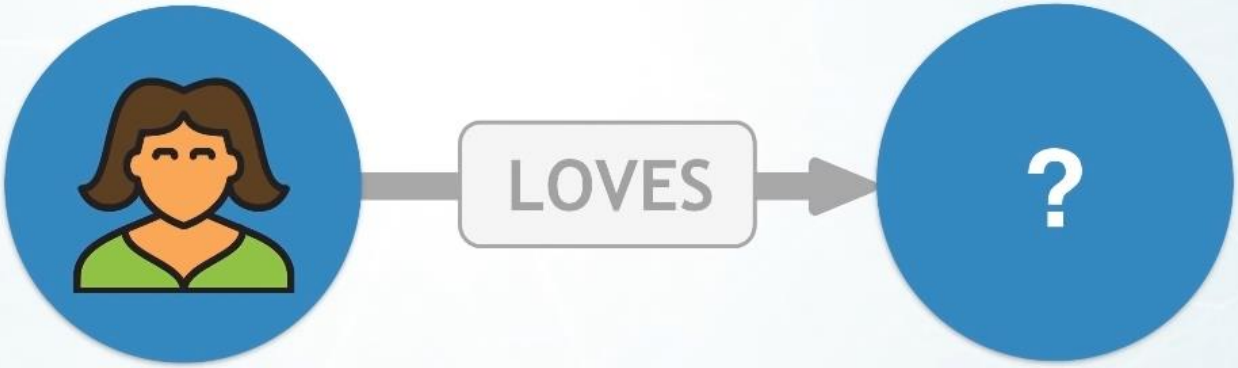


Therefore, the **correct** (or at least most efficient) way of modelling the partner relationships is using a **single** *PARTNER* **relationship with an arbitrary direction**.

Creating the Data



Whom does Ann love?




```
MATCH
   (:Person {name:"Ann"})-[:LOVES]->(op:Person)
RETURN
   op
```

How do I find Ann's car?

```
MATCH
  (:Person {name: 'Ann'})-[:DRIVES]->(c:Car)
RETURN
  c
```

How do I find Ann's car?

Another way:

```
MATCH
  (a:Person)-[:DRIVES]->(c:Car)
WHERE
  a.name='Ann'
RETURN
  c
```

Uniqueness

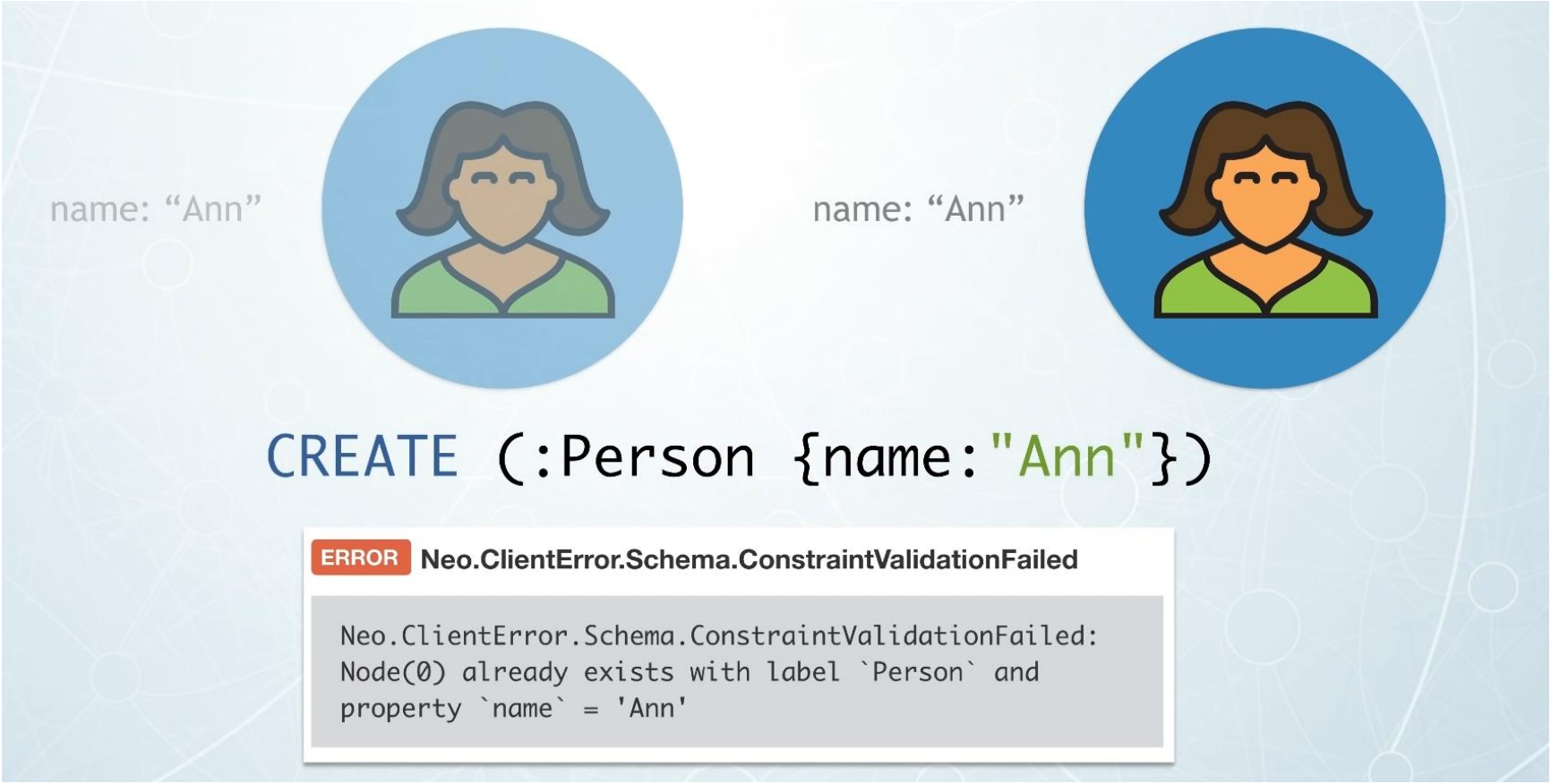

name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”

There can only be only One!

name: “Ann”

```
CREATE CONSTRAINT ON (p:Person)
ASSERT p.name IS UNIQUE
```

I want another Ann!



Creating and Querying Nodes

```
CREATE (me:Person {name: "My Name" })
RETURN me

MATCH (me:Person )
WHERE me.name="My Name"
RETURN me.name
```

or in a short-hand syntax:

```
MATCH (me:Person {name:"My Name"})
RETURN me.name
```

The All Nodes Query

Return **all nodes** in the graph:

```
MATCH (n)
RETURN n
```

(In a larger graph this will return A LOT of data, usually you add a LIMIT 100 or such.)

Full graph search.

Visits every single node to see whether it matches the pattern of (n). In this concrete case

The pattern is simply a node that may or may not have a label or relationships, so it will match every single node in the graph.

The RETURN clause then returns all of the information about each of those nodes, including all of their properties.

Query Data

:play movie-graph

```
MATCH (p:Person {name: "Tom Hanks"})-
[r:ACTED_IN|DIRECTED]-(m:Movie)
RETURN p,r,m
```

```
MATCH (p:Person {name: "Tom Hanks"})-
[r:ACTED_IN|DIRECTED]-(m:Movie)
RETURN p.name, type(r), m.title
```

Filtering Results

```
MATCH (m:Movie {title: "The Matrix"})
RETURN m
```

```
MATCH (m:Movie)
WHERE m.title = "The Matrix"
RETURN m
```

Where clause

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE m.released >= 2000
RETURN m.released, p.name
```

Filtering Results

Filter by comparing properties of different nodes. For example, we could RETURN all of the actors who acted with Tom Hanks and are older than him:

```
MATCH (tom:Person)-[:ACTED_IN]->()<-[:ACTED_IN]-(actor:Person)
WHERE tom.name = "Tom Hanks"
AND actor.born < tom.born
RETURN actor.name AS Name
```

Operators

```
=, <>, <, >, <=, >=, IS NULL, IS NOT NULL
```

Regular expressions

You can match on regular expressions by using =~ 'regexp', like this:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released >= 2000
RETURN p, r, m
```

Adding Properties

Add a tagline to the "Mystic River" :Movie node we've just added. First, locate the single movie again by its title, then SET the tagline property. The query:

```
MATCH (movie:Movie)
WHERE movie.title = "Mystic River"
SET movie.tagline = "We bury our sins here,
Dave. We wash them clean."
RETURN movie.title AS title, movie.tagline
AS tagline
```

Update Property

```
MATCH (movie:Movie)
WHERE movie.title = "Mystic River"
SET movie.released = 2003
RETURN movie.title AS title, movie.released AS released
```

The syntax is the same for updating or adding a property. You SET a property. If the property exists, SET will update it. If the property doesn't exist, SET will add it.

Creating Relationships

```
CREATE (movie:Movie {title: "Mistic River", released:1993 })
```

find the actor "Kevin Bacon" and the movie "Mystic River" and add the relationship between the movie and the actor to the dataset:

```
MATCH (kevin:Person) WHERE kevin.name = "Kevin Bacon"
MATCH (mystic:Movie) WHERE mystic.title = "Mystic River"
CREATE (kevin)-[r:ACTED_IN {roles:["Sean"]}]->(mystic)
RETURN mystic, r, kevin
```

Creating Relationships

Create ourselves first in the database:

```
CREATE (me:Person {name:"My Name"}) RETURN me.name
```

Rate the movie "Mystic River":

```
MATCH (kevin:Person), (movie:Movie)
WHERE me.name="My Name" AND movie.title="Mystic River"
CREATE (me)-[r:REVIEWED {rating:80, summary:"tragic character movie"}]->(movie)
RETURN me, r, movie
```

Deleting Nodes

```
CREATE (me:Person {name:"My Name"}) RETURN me.name
```

Let's then run the following query to make sure you have been added successfully to the graph.

```
MATCH (p:Person {name:"My Name"})
RETURN p.name
```

Deleting Nodes

To remove both yourself and any relationships you may or may not have, you need to run:

```
MATCH (p:Person {name:"My Name"})
OPTIONAL MATCH (me)-[r]-()
DELETE me,r
```

The OPTIONAL MATCH clause is used to search for the pattern described in it, while using nulls for missing parts of the pattern.

Deleting Nodes

As this is a frequent task, `DETACH DELETE` was added to Cypher, which deletes a node with all its relationships.

```
MATCH (emil:Person {name:"Emil Eifrem"})
DETACH DELETE emil
```

Delete all nodes and relationships

```
MATCH (n)
DETACH DELETE n
```

Order by, Skip and Limit

In Cypher it's easy to order results using an ORDER BY command.

Display the oldest people in the database. We could use the following query:

```
MATCH (person:Person)
RETURN person.name, person.born
ORDER BY person.born
```

```
MATCH (actor:Person) -[:ACTED_IN]->(movie:Movie)
RETURN actor.name AS Actor, movie.title AS Movie
SKIP 10 LIMIT 10
```

Case Sensitivity

Case sensitive:

- Node Labels
- Relationship types
- Porperty keys

Case insensitive:

- Cypher keywords

Note on null

`null` is not `null`

null represents missing or undefined values.

You do not store a null value in a property. It just doesn't exist on that particular node.

Indexing and Labels

A database index is a redundant copy of some of the data in the database for the purpose of making searches of related data more efficient.

This comes at the cost of additional storage space and slower writes, so deciding what to index and what not to index is an important and often non-trivial task

Indexing and Labels

To create index:

```
CREATE INDEX ON :Movie(title)
```

Drop the index created:

```
DROP INDEX ON :Movie(title)
```

LAB

Lab – Publicidade baseada em plataforma de ecommerce

#1: Instalar neo4j

Download e instalação da community edition do neo4j

#2: Abrir o browser do neo4j

“Add graph” > “Play” > Abrir “neo4J browser”

#3: Objetivo

Oferecer os produtos ou serviços mais relevantes aos clientes

Lab – Publicidade baseada em plataforma de ecommerce

#4: Importar os dados

Copiar o conteúdo do ficheiro email_neo4j.txt para o browser e correr.

Verificar a correta importação:

```
MATCH (n)
RETURN n
```

Lab – Publicidade baseada em plataforma de ecommerce

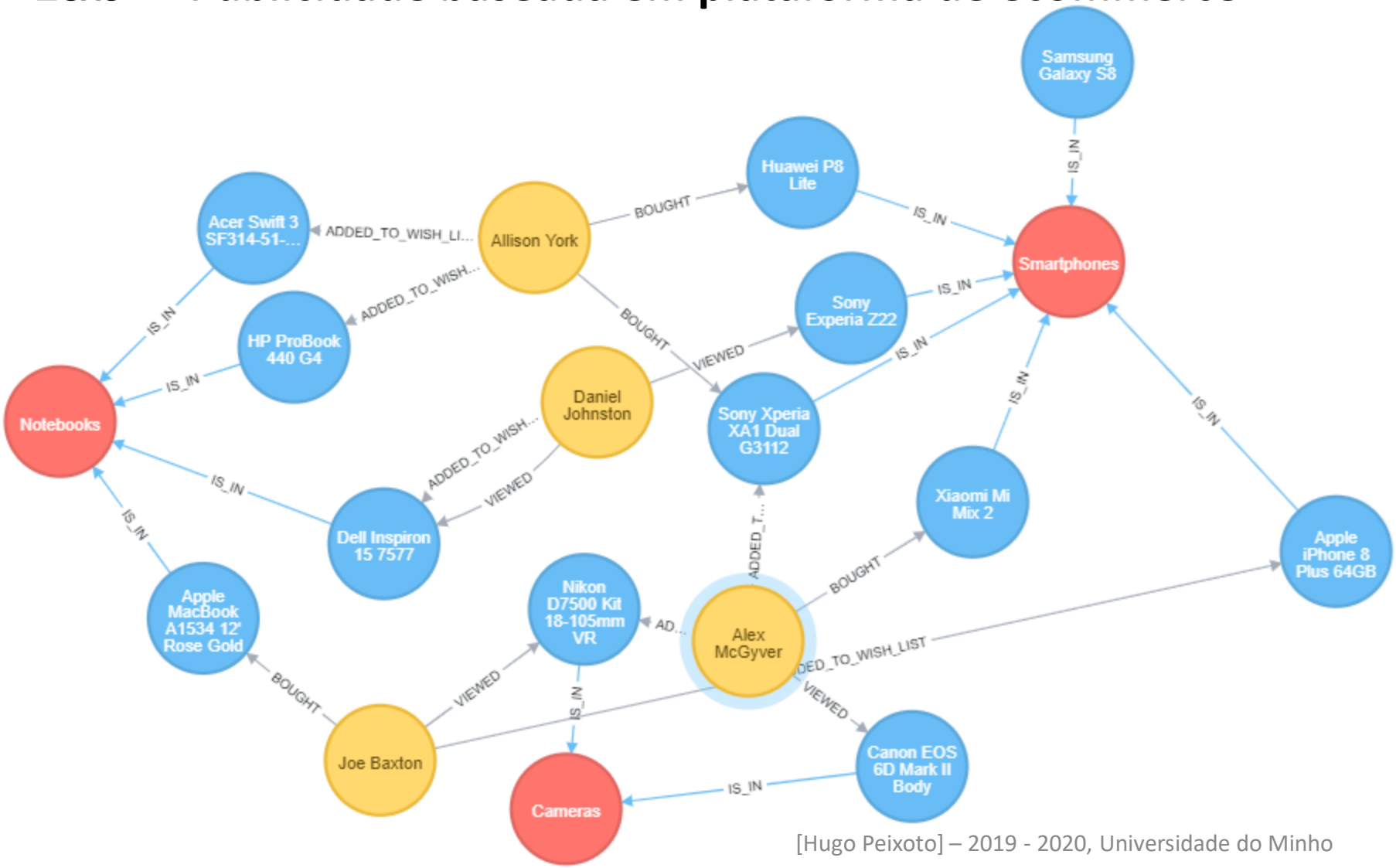
O Sistema contém as seguintes entidades(atributos):

Category (title)
Product (title, description, price, availability, shippability)
Customer (name, email, registration date)
Promotional Offer (type, content)

Relações:

Product is_in **Category**
Customer added_to_wish_list **Product**
Customer bought **Product**
Customer viewed (clicks_count) **Product**

Lab – Publicidade baseada em plataforma de ecommerce



Lab – Publicidade baseada em plataforma de ecommerce

Exemplo #1: Utilizar o neo4j para determinar as preferências de um determinado cliente.

É necessário aprender quais as preferências dos clientes de forma a criar uma oferta profissional para uma determinada categoria, neste caso **notebooks**.

Lab – Publicidade baseada em plataforma de ecommerce

Query for all the notebooks that users have viewed or added to their list:

```
MATCH (:Customer)-[:ADDED_TO_WISH_LIST|:VIEWED]-
>(notebook:Product)-[:IS_IN]->(:Category {title:
'Notebooks'})
RETURN notebook;
```

OU

```
MATCH (:Customer)-[:ADDED_TO_WISH_LIST|:VIEWED]-
>(notebook:Product)-[:IS_IN]->(cat:Category)
WHERE cat.title = 'Notebooks'
RETURN notebook;
```

Lab – Publicidade baseada em plataforma de ecommerce

Incluir os notebooks numa proposta promocional:

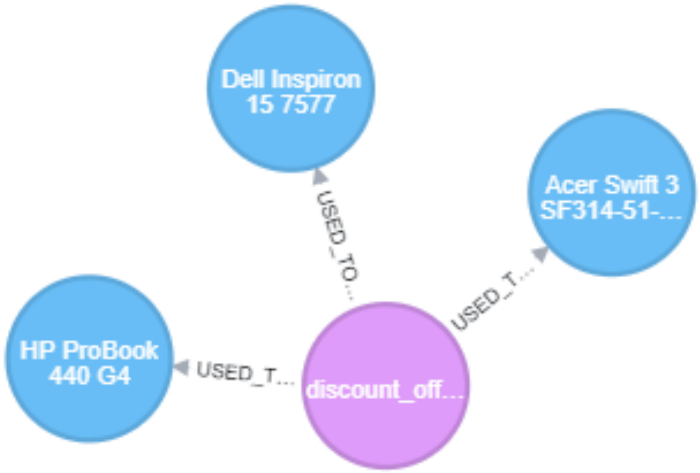
```
CREATE(offer:PromotionalOffer {type: 'discount_offer',
content: 'Notebooks discount offer...'})
WITH offer
MATCH (:Customer)-[:ADDED_TO_WISH_LIST|:VIEWED]-
>(notebook:Product)-[:IS_IN]->(:Category {title:
'Notebooks'})
MERGE(offer)-[:USED_TO_PROMOTE]->(notebook);
```

MERGE = MATCH + CREATE

Lab – Publicidade baseada em plataforma de ecommerce

Validar a criação da oferta promocional.

```
MATCH (offer:PromotionalOffer)-[:USED_TO_PROMOTE]->(product:Product)
RETURN offer, product;
```

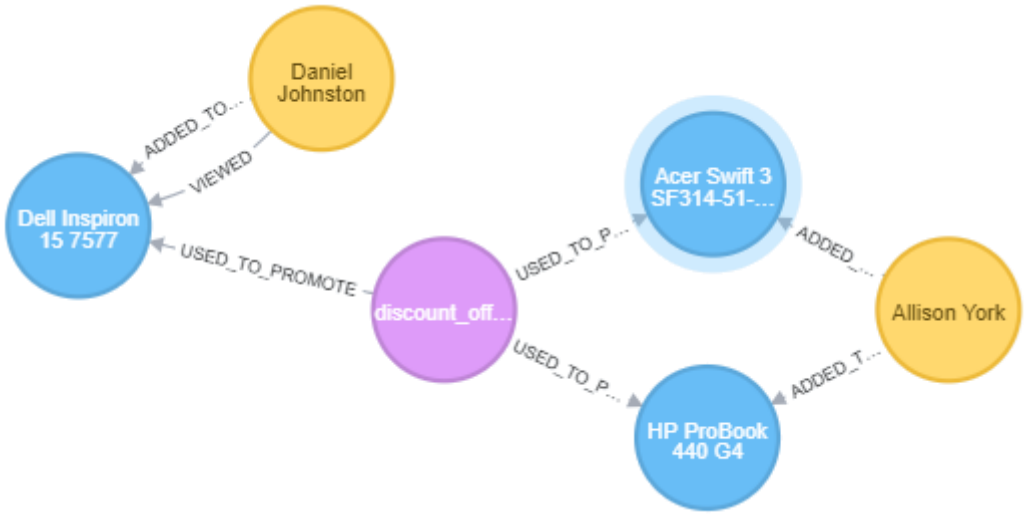


Lab – Publicidade baseada em plataforma de ecommerce

Ao construir a oferta profissional é importante perceber quais os clientes que viram ou adicionaram os notebooks selecionados à sua lista:

```
MATCH (offer:PromotionalOffer {type: 'discount_offer'})-
[:USED_TO_PROMOTE]->(product:Product)<-
[:ADDED_TO_WISH_LIST|:VIEWED]-(customer:Customer)
RETURN offer, product, customer;
```

Lab – Publicidade baseada em plataforma de ecommerce



Lab – Publicidade baseada em plataforma de ecommerce

Exemplo #2

É necessário desenvolver uma campanha professional mais eficiente cuja taxa de conversão seja mais elevada. Deve para isso ser feita uma oferta de produtos alternativos aos clientes. Por exemplo, se um cliente mostrou interesse num produto e não o comprou, pode ser criada uma oferta promocional que ofereça produtos relacionados.

Lab – Publicidade baseada em plataforma de ecommerce

All products that don't have either:

`ADDED_TO_WISH_LIST`,
`VIEWED`,
`BOUGHT` relationships with a client named Alex McGyver.

Opposite query that finds all products that Alex McGyver has

`ADDED_TO_WISH_LIST`,
`VIEWED`,
`BOUGHT`.

Two queries select products in the same categories.

Only products that cost 20 percent more or less than a specific item should be recommended to the customer.

Lab – Publicidade baseada em plataforma de ecommerce

```
MATCH (alex:Customer {name: 'Alex McGyver'})
MATCH (free_product:Product)
WHERE NOT ((alex)-->(free_product))
MATCH (product:Product)
WHERE ((alex)-->(product))
MATCH (free_product)-[:IS_IN]->(category)<-[:IS_IN]-(product)
WHERE  ((product.price-product.price*0.20)  >=  free_product.price
<= (product.price+product.price*0.20))
RETURN free_product;
```

Lab – Publicidade baseada em plataforma de ecommerce

The **product** variable is supposed to contain the following items:

- **Xiaomi Mi Mix 2** (price: $420.87): Price range for recommendations: from $336.70 to $505.04.
- **Sony Xperia XA1 Dual G3112** (price: $229.50): Price range for recommendations: from $183.60 to $275.40.

The **free_product** variable is expected to have these items:

- Apple iPhone 8 Plus 64GB (price: $874.20)
- Huawei P8 Lite (price: $191.00)
- Samsung Galaxy S8 (price: $784.00)
- Sony Xperia Z22 (price: $765.00)

Lab – Publicidade baseada em plataforma de ecommerce

The **product** variable is supposed to contain the following items:

- Xiaomi Mi Mix 2 (price: $420.87): Price range for recommendations: from $336.70 to $505.04.
- Sony Xperia XA1 Dual G3112 (price: $229.50): Price range for recommendations: from $183.60 to $275.40.

The **free_product** variable is expected to have these items:

- Apple iPhone 8 Plus 64GB (price: $874.20)
- **Huawei P8 Lite (price: $191.00)**
- Samsung Galaxy S8 (price: $784.00)
- Sony Xperia Z22 (price: $765.00)

Lab – Publicidade baseada em plataforma de ecommerce

É possível agora criar uma oferta promocional, com:

- tipo: 'personal_replacement_offer’
- conteúdo: 'Personal replacement offer for ' + alex.name.

O que vai ser armazenado vai ser o email do cliente como propriedade da relação “USED_TO_PROMOTE” entre o cliente e o produto.

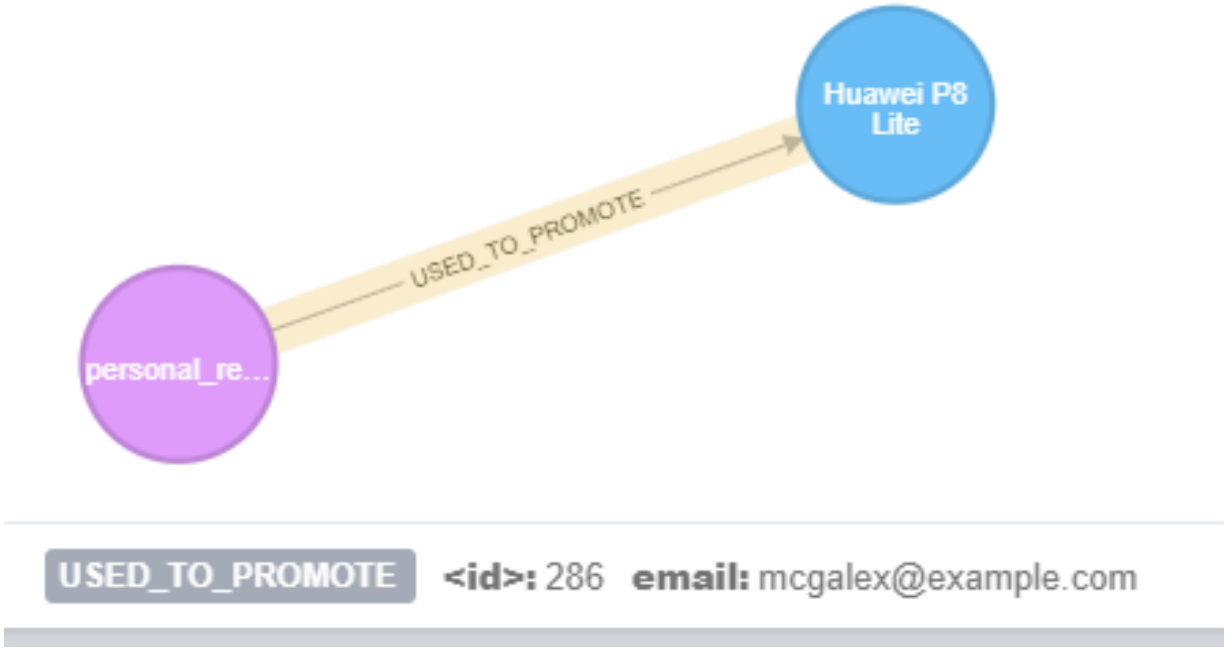
Lab – Publicidade baseada em plataforma de ecommerce

```
MATCH (alex:Customer {name: 'Alex McGyver'})
MATCH (free_product:Product)
WHERE NOT ((alex)-->(free_product))
MATCH (product:Product)
WHERE ((alex)-->(product))

MATCH (free_product)-[:IS_IN]->()<-[:IS_IN]-(product)
WHERE  ((product.price-product.price*0.20)  >=  free_product.price
<= (product.price+product.price*0.20))

CREATE(offer:PromotionalOffer{type:'personal_replacement_offer',
content: 'Personal replacement offer for ' + alex.name})
WITH offer, free_product, alex
MERGE(offer)-[rel:USED_TO_PROMOTE{email:alex.email}]-
>(free_product)
RETURN offer, free_product, rel;
```

Lab – Publicidade baseada em plataforma de ecommerce



FE07

FE07

Usar a mesma BD dos exemplos anteriores:

#1: Criar uma query que que descubra clientes com interesses semelhantes.

#2: Com base nas evidências encontradas recomendar ao cliente os produtos dos clientes com interesses semelhantes que não fazem parte dos seus interesses.

FE07

Add graph > “Play” > open Neo4J browser.

Import “movies” database:

- :play movies > 2nd page > click the code > press play

FE07

Try out query:

```
MATCH (n)
RETURN n
```

Try out query:

```
MATCH (n) WITH COUNT(n) AS numVertices
MATCH (a)-[e]->(b) WITH COUNT(e) AS numEdges
RETURN numVertices, numEdges
```

FE07

1) RETURN a list of all the characters in the movie The Matrix.

Movies have the label Movie and a title property you want to compare to.

We're looking for the characters—the roles which are a property of the ACTED_IN relationships—not the names of the actors.

2) Find all of the movies that Tom Hanks acted in?

3) Limit that to movies which were released after 2000? Note that there is a released property on Movie.

FE07

4) Find directors acting in their movies?

5) Find all movies in which Keanu Reeves played the role Neo.

You need an variable for the relationship.

The **ACTED_IN** relationship has a **roles** property (which is an array).

The syntax for seeing whether an element is in an array is **{element} IN r.roles**.

Generally check for the existence of the value of **{expression} IN {collection}**

6) Return the names of all the directors each actor has worked with.

FE07

7) Return the count of movies in which each actor has acted.

8) Return the count of movies in which an actor and director have jointly worked.

9) Write a query that will display the five (5) busiest actors, i.e. the ones who have been in the most movies.

Bases de Dados de Grafos

AULA PL07

Hugo Peixoto

2019 – 2020 Universidade do Minho