Introduction to Neo4j at Foo Café

About Me

- Information Developer
- IT Project Manager
- Oracle DBA

maria.scharin@neo4j.com

Neo4j

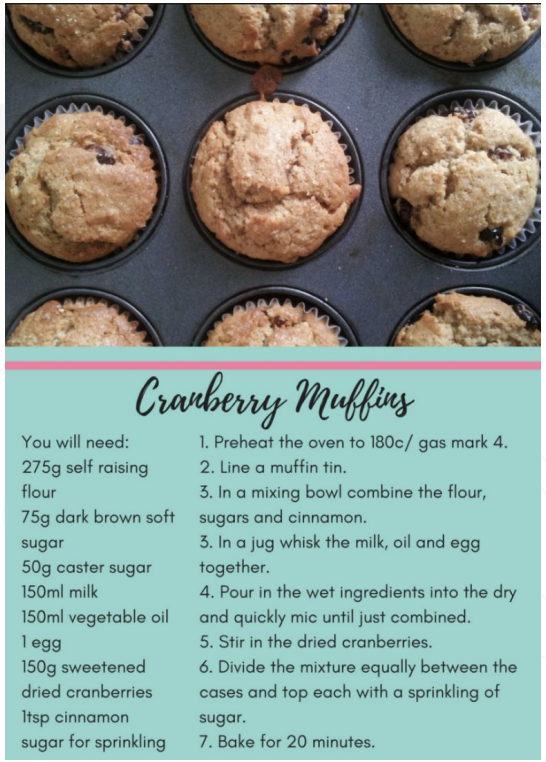
- Creators of Neo4j, the world's leading graph database
- Swedish founders
- 260 employees; main offices in
  - Malmö (Eng. HQ)
  - San Mateo (HQ)
  - London


PINK PROGRAMMING
Pink Programming is a non-profit association whose goal is to increase the number of women to program. We work for a programming world where women are as obvious as men. Programming joy should be available to everyone.

What will we do?

- Relational databases and Neo4j
- Introduction to the property graph model
- Introduction to Cypher
- Some Neo4j concepts for non-programmers
- *Hello World!*

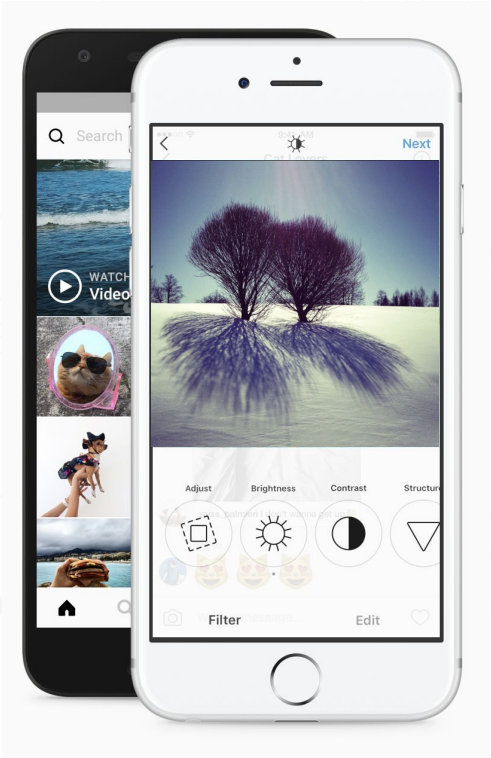
Programming is like cooking



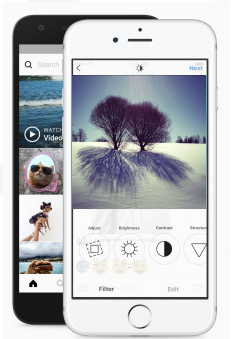
A computer program is like a recipe:

A set of instructions in a particular language that need to be followed in a particular order.

What's behind an app or a web site?



What's behind an app or a web site?



Application

contains
business logic

Database

stores
data

What's behind an app or a web site?

Frontend

Backend

Application

Could be written in HTML, CSS and JavaScript

Could be written in some language, for example: Java, Python

Driver

Database

stores data

Languages

- Natural languages - Languages that we speak
- Programming languages
  For example: Java, Python, JavaScript, C#
- Query languages - Used to “talk with” databases
  For example: SQL, Cypher

Languages

Similarities between a natural language and a programming language:

All languages...

- can describe things: "Ann has a yellow blouse"
- can give instructions: "Pass the butter"
- have grammar or *rules* in them, that need to be followed in order to make sense.

Languages

Specific to programming languages

- There is no room for vagueness in programming. Either you do something or you don't!
- A programming language uses specific words to describe things and give instructions. We call those words *keywords*.
- Programming languages make things more efficient by using *variables*.
- Programming languages are very good at:
  - sorting large amounts of data
  - doing the same thing over and over (like a million times)
  - deciding whether to do something based on conditions (**if** somebody likes an instagram post **then** add a )

Databases - Relational databases

Relational databases store data in tables.

Examples of relational databases:

- Oracle
- MSSQL
- MySql

FIRST_NAME	LAST_NAME	CITY
Maria	Scharin	Lund
Chelsi	Nolasko	Malmö
Myky	Tran	Lund
Emma	Herrlin	Malmö

You use SQL to write to and read from a relational database.

Databases - Relational databases

Benefits:

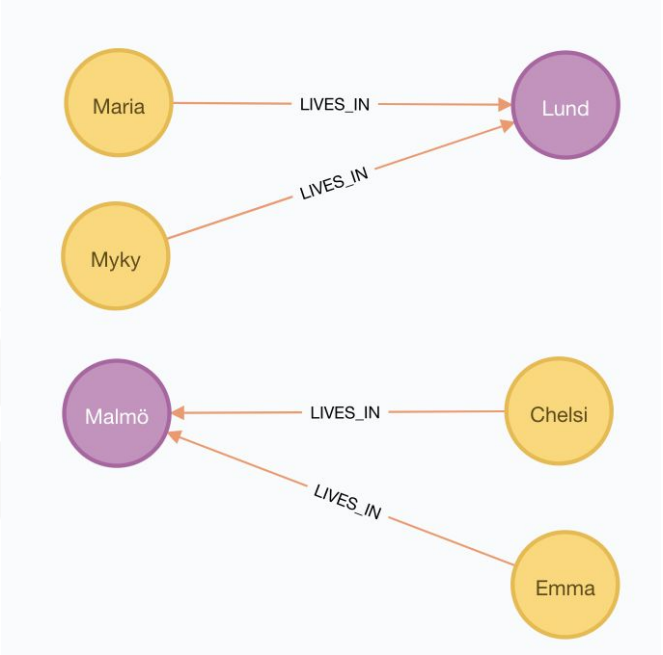
- They have been around for a long time; many people know them
- They store *structured data* very efficiently

Drawbacks:

- They require specialized people, data modellers, to translate between the users and the developers. The model is abstracted.
- It is complicated to efficiently describe certain use cases;
  after the modelling is done it is hard for a non-IT person to recognize.

Databases - Neo4j

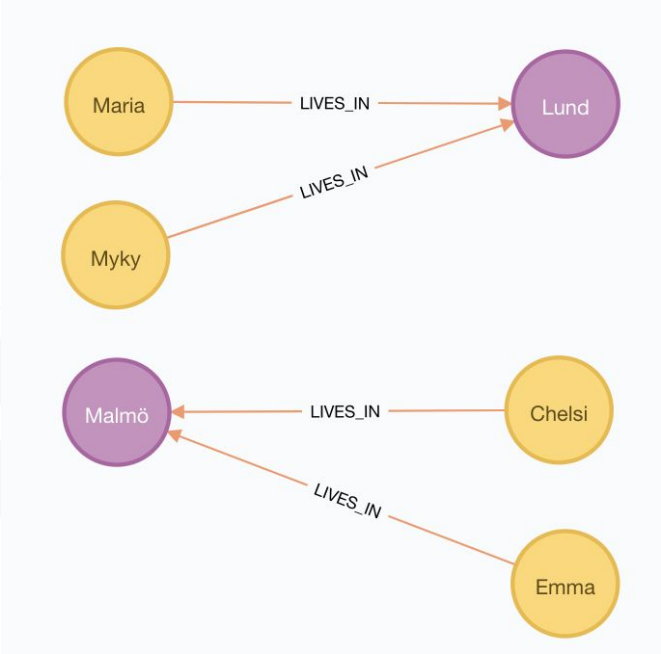
- Neo4j is a Graph Database.
- Neo4j stores data in nodes and relationships.
- You use Cypher to write to and read from Neo4j.



Databases - Neo4j

Benefits of graph databases:

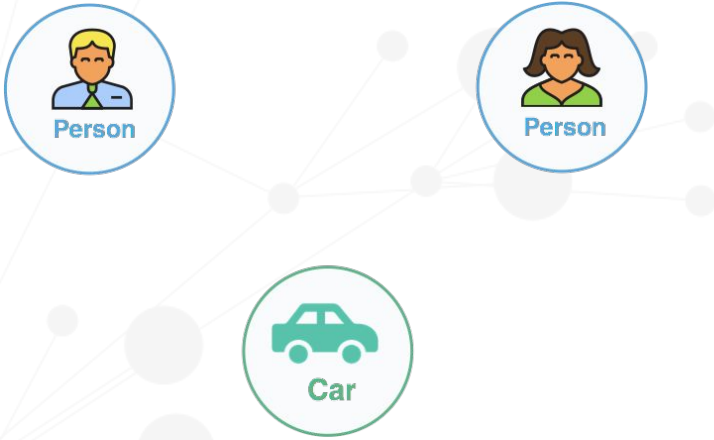
- They describe things the way we see them; no abstraction needed.
- They store and retrieve connected data very efficiently.
- They are well suited for combining different types of data:
  - *Structured data* (structured in tables)
  - *Unstructured data* (emails, books etc.)
  - Data with various *data quality*.



Introduction to the property graph model

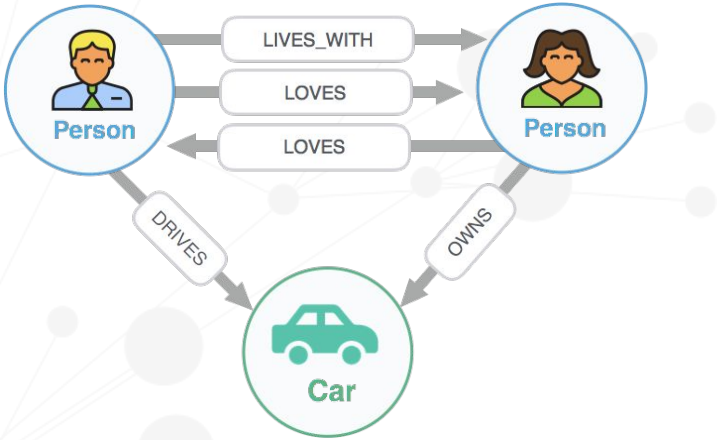
Nodes

- Nouns in your model
- Represent the objects or entities in the graph
- Can be *labeled*:
  - Person
  - Location
  - Residence
  - Business
- A node can have any number of labels



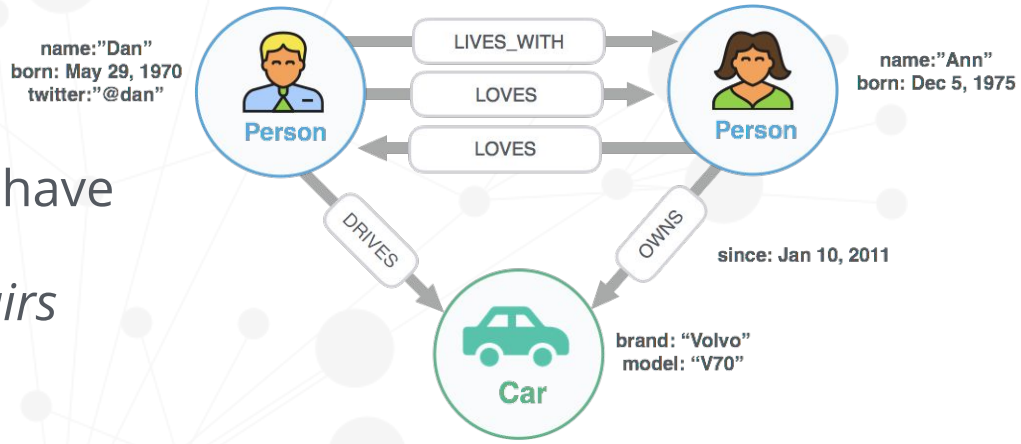
Relationships

- Verbs in your model
- Represent the connection between nodes in the graph
- Has a type (exactly one):
  - LIVES_WITH
  - LOVES
  - OWNS
- Has a direction



Properties

- Used to *describe* nodes and relationships
- Nodes and relationships can have any number of properties
- Are described as *key/value pairs*
  - `name: "Dan"`
  - `born: "May 29, 1970"`
  - `since: "Jan 10, 2011"`

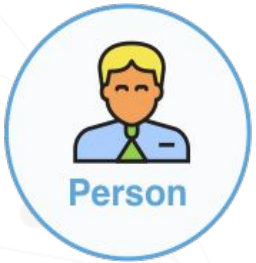


Introduction to Cypher

Cypher: The Graph Query Language

- Declarative - Focus on *what* to retrieve, not *how* to retrieve it
- Use pattern matching
- Intuitive and easy to learn (ASCII-Art) ¯_(ツ)_/¯

ASCII-Art



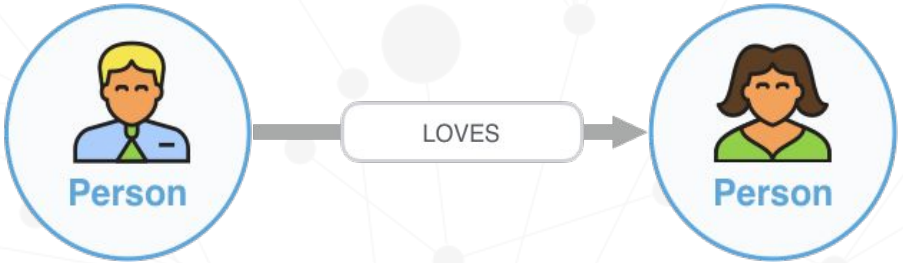
Node

(:Person)

Node

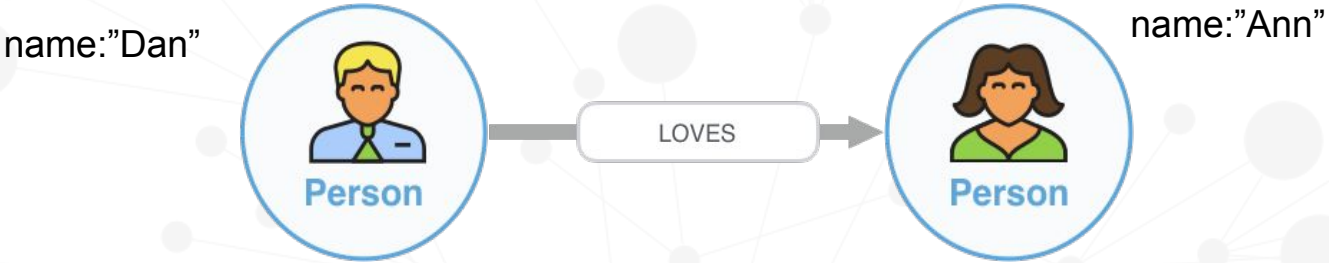
(:Person)

ASCII-Art



Node	Relationship	Node
(:Person)	-[:LOVES]->	(:Person)

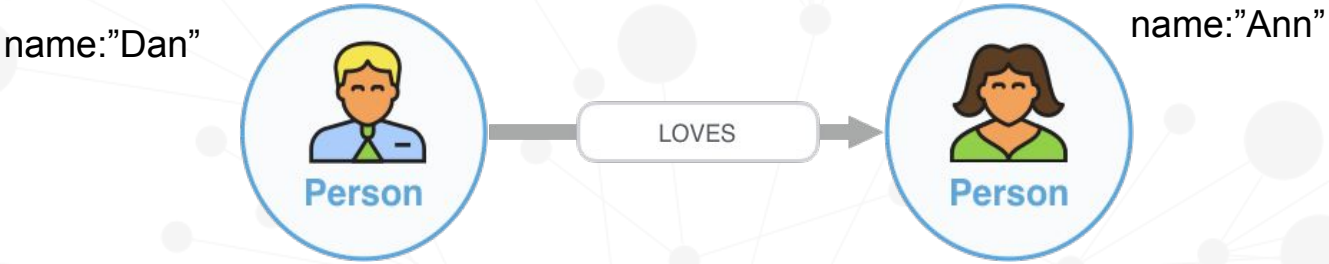
ASCII-Art



Node Relationship Node

(:Person{name: "Dan"}) -[:LOVES]-> (:Person{name: "Ann"})

The CREATE keyword



Node Relationship Node

```
CREATE (:Person{name: "Dan"}) -[:LOVES]-> (:Person{name: "Ann"})
```

CREATE nodes

`CREATE ()`	Create an "anonymous" node
`CREATE (:Person)`	Create a node with the label *Person*
`CREATE (:Person{name: "Maria"})`	Create a node with the label *Person* and a property called *name* with the value "Maria"
`CREATE (:City{name: "Lund"})`	Create a node with the label *City* and a property called *name* with the value "Lund"

CREATE relationships

Remember:

1. Every relationship must have a startnode and an end-node
2. Every relationship has a direction

```
CREATE ()-[]->()
```

Create an "anonymous" relationship between two “anonymous” nodes

```
CREATE ()-[:LIVES_IN]->()
```

Create an LIVES_IN relationship between two “anonymous” nodes

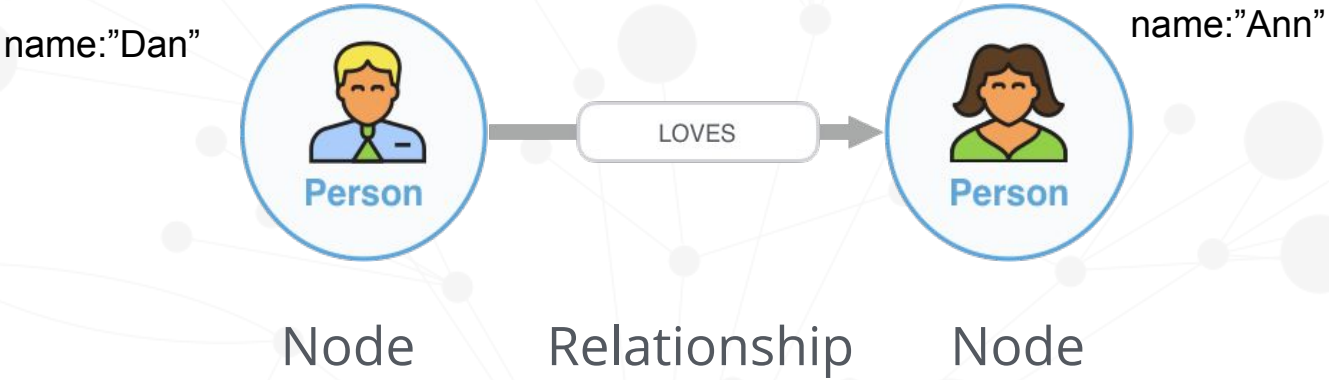
CREATE nodes and relationships

```
CREATE (:Person{name: "Maria"})-[:LIVES_IN]->(:City{name: "Lund"})
```

This does three things:

1. Create a node with the label Person
   and a property called name with the value "Nina"
2. Create a node with the label City
   and a property called name with the value "Lund"
3. Create a relationship of type LIVES_IN between Nina and Lund
   (with direction from Nina to Lund)

The MATCH and RETURN keywords




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

We need to use a variable in order to be able to use what we have found later on.
It can be a letter or a short word. In this example we chose “p” for “Person”.

Using variables - 1

- A variable is a temporary container that we can put a value in, in order to use it later.
- In Cypher, we put values into variables in order to be able to use them later on in the query.
- We can give the variable any name we like,
  for example: `i`, `p`, `x`, `person`, `thisPerson`, `city`, `dan`....
- In these first examples, we use the variable to **RETURN** the value.

```
MATCH (p:Person)
RETURN p
```

is the same as:

```
MATCH (person:Person)
RETURN person
```

Using variables - 2

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

Finds:

- all nodes with label *Person (store these in the variable p)*
- that has a *LOVES* relationship with
- another *Person* node that has a parameter *name with a value "Ann" (store this in the variable ann)*

Returns all the persons that we found (p and ann)

-- Everybody who loves a person called Ann!

Using variables - 3

```
MATCH (p:Person)-[:LIVES_IN]->(city:City)
RETURN p, city
```

Finds:

- all nodes with label *Person (store these in the variable p)*
- that has a *LIVES_IN* relationship with
- another node that has a label *City (store these in the variable c)*

Returns all the persons and cities that we found (p and c)

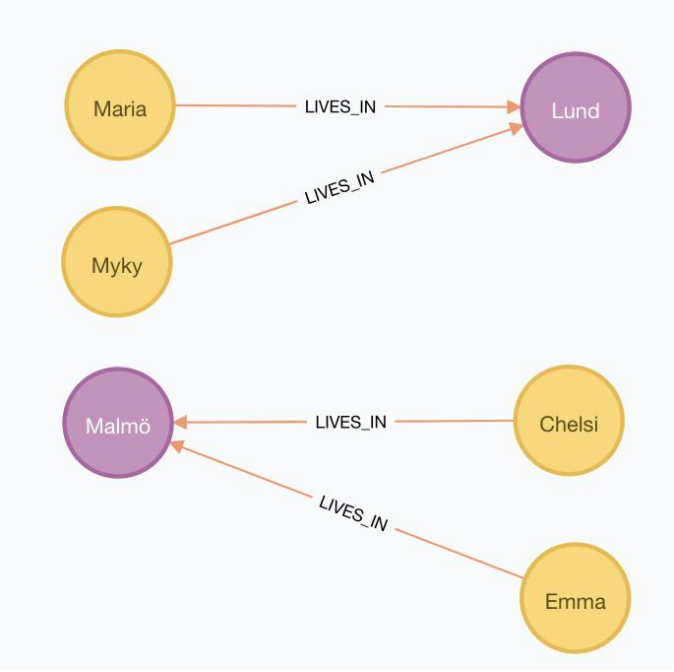
Using variables - 4

```
MATCH (:Person)-[rel:LIVES_IN]->(:City)
RETURN rel
```

Finds:

- all nodes with label *Person*
- that has a *LIVES_IN* relationship with
- another node that has a label *City*

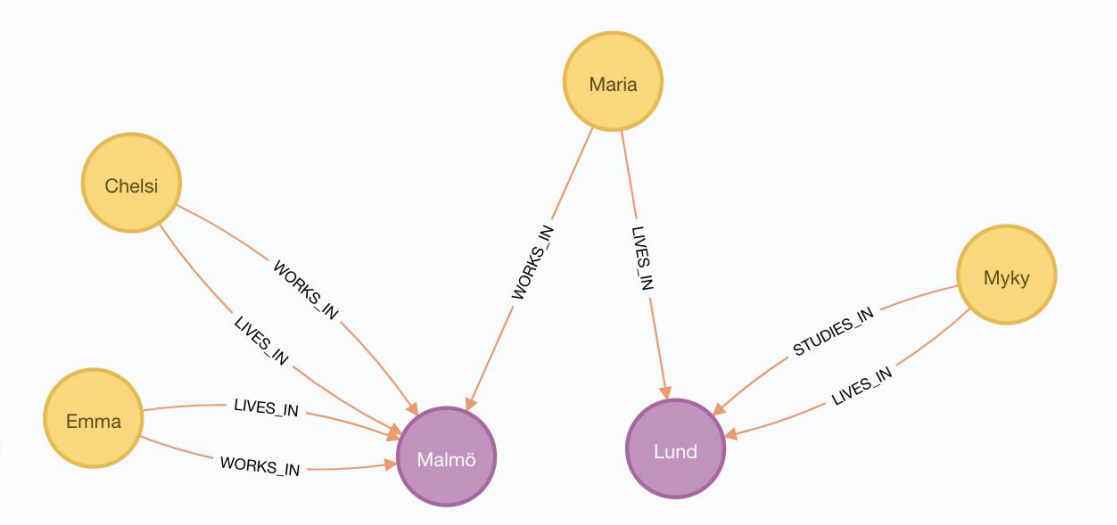
Returns all the relationships (rel)



Using variables - 5

```
MATCH (:Person)-[rel:LIVES_IN]->(:City)
RETURN rel
```

The query still only returns four relationships, since it specifically targets those with the type LIVES_IN



MATCH and CREATE in the same query

```
MATCH (p:Person{name:"Maria"})
MATCH (c:City{name:"Lund"})
CREATE (p)-[rel:LIVES_IN]->(c)
RETURN p, rel, c
```

What happens if there are many persons named Maria or many cities named Lund?

This does four things:

1. Find a node with label *Person* and name *Maria*; call it p
2. Find a node with label *City* and name *Lund*; call it c
3. Crete a *LIVES_IN* relationship between p and c; call it rel
4. Return p, rel and c

RETURN values of properties on nodes

So far, we have returned the nodes, for example:

```
MATCH (p:Person)
RETURN p
```

If we instead want to get properties **in a list**, we use this syntax:

```
MATCH (p:Person)
RETURN p.name
```

p.name
"Maria"
"Myky"

RETURN values of properties on relationships

Find all *Person* nodes that has a *LIVES_IN* relationship with a *City* node and return the value of the properties *name* and *since:*

```
MATCH (p:Person)-[rel:LIVES_IN]-(c:City)
RETURN p.name, rel.since
```

p.name	rel.since
"Maria"	2004
"Myky"	2015

Watch out for case sensitivity!

Case sensitive:	
Node labels	**:Person** is not the same as **:person**
Relationship types	**:LIVES_IN** is not the same as **:lives_in**
Property keys	**Name** is not the same as **name**

Watch out for case sensitivity!

Case insensitive: Cypher keywords	
`MATCH` is the same as **`MaTcH`**	but **`MATCH`** looks better
`RETURN` is the same as **`return`**	but **`RETURN`** looks better

Hello World!

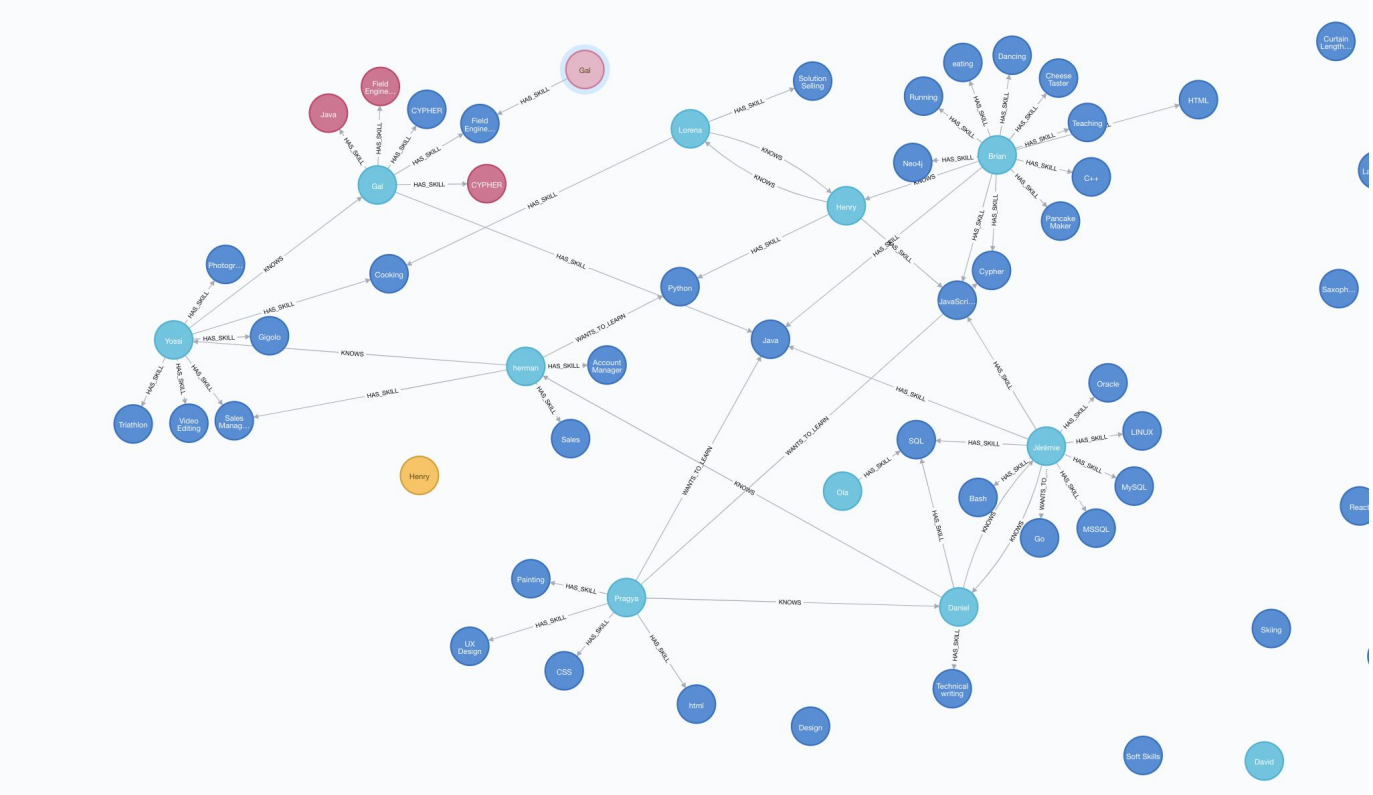
Let's graph our class!

Open document: https://bit.ly/2GW4fbn

Don't hesitate! This is just a play environment

neo4j

Example from last class

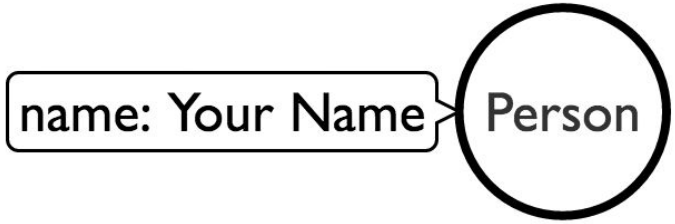


Skill(81) Preson(4) Person(18) City(2) SKILL(3) person(3)

neo4j

Let's try it - `CREATE` myself!

`CREATE` a person node within the graph to represent yourself!



Label of `:Person`

Property `name` should be `'your name'`

Welcome to the graph

CREATE statement:

```
CREATE (p:Person {name: 'Maria'})
RETURN p
```

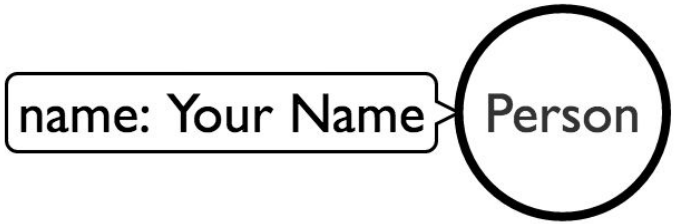
MATCH statement:

```
MATCH (p:Person {name: 'Maria'})
RETURN p
```

Your turn: `CREATE` a node for yourself

CREATE yourself:

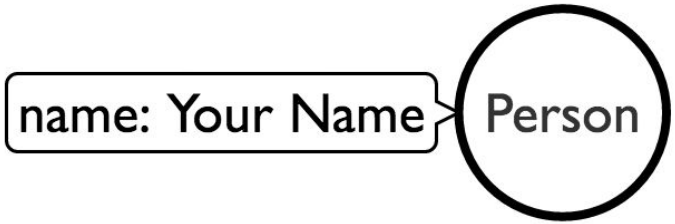
```
CREATE (p:Person {name: 'Your Name'})
RETURN p
```



Your turn: MATCH and RETURN yourself

Find yourself:

```
MATCH (p:Person {name: 'Your Name'})
RETURN p;
```



SET and update properties

Find yourself and **SET** your city as a property.

```
MATCH (p:Person {name: 'Your Name'})
SET p.city = 'Your City'
RETURN p
```

Find your neighbour

Ask the person to your **left** their name.
Find the pair of you:


```
MATCH (p1:Person {name: 'Your Name'})
MATCH (p2:Person {name: 'Your Neighbour'})
RETURN p1, p2
```

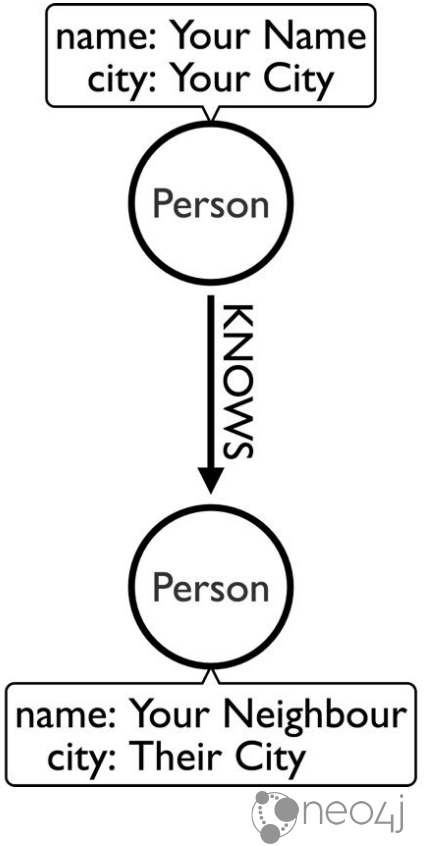
Or in one statement:


```
MATCH (p1:Person {name: 'Your Name'}),
      (p2:Person {name: 'Your Neighbour'})
RETURN p1, p2
```

Creating relationships

Create a relationship between you and the person sitting next to you.

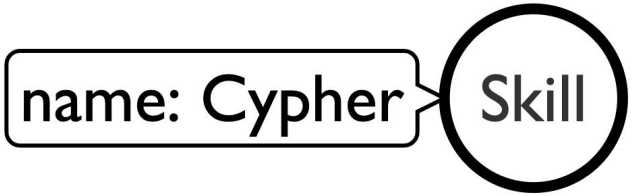
```
MATCH (p1:Person {name: 'Your Name'})
MATCH (p2:Person {name: 'Your Neighbour'})
CREATE (p1)-[:KNOWS]->(p2)
```



The Skills Graph

Finding skills

We've pre-populated the graph with a set of **Skills.**
See if you can write a query to find them.



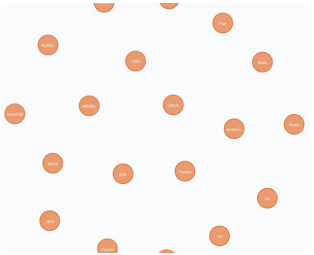
Finding skills

Your query should look something like this:

```
MATCH (s:Skill)
RETURN s
```

or

```
MATCH (s:Skill)
RETURN s.name
```



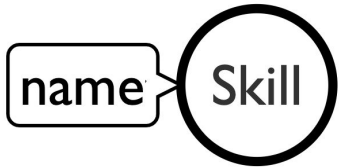
s.name
"Java"
"Python"
"C#"
"Go"
"JavaScript"
"Scala"
"Bash"
"LINUX"
"Perl"
"HTML"
"CSS"
"Oracle"

Missing skills

The database contains a bunch of languages and databases.
Chances are you are missing your skills in there!
Can you create some?

NOTE: Each skill should have:

- a label called `Skill`
- a property called `name`



Missing skills

Your create statement could look something like this:

```
CREATE (:Skill { name: 'Your skill' })
```

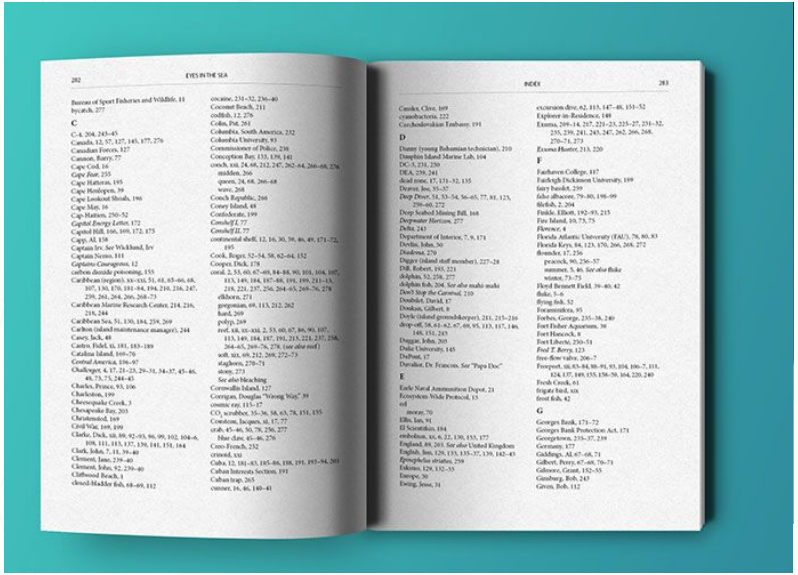
Indexes and Constraints

Indexes

You have a large amount of information of different kinds, for example in a book, and you need to find something very specific very quickly.

For example: you need to find all the pages where a specific character is mentioned.

What will you do?



Indexes

If you know how your book will be used for specific situations, you may create different indexes for different types of information.

Here, for example, there is a special index to find pages where a character is mentioned in the book.

We do the same in the database!

Constraints

Constraints are used to enforce business rules.

Examples:

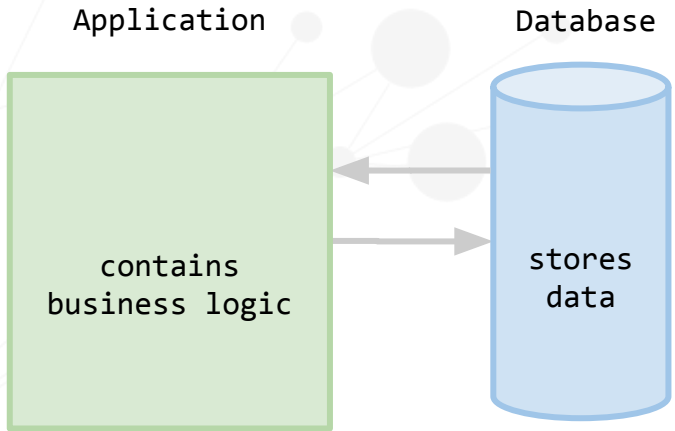
- Each person must have a name
- Two persons cannot have the same ID number

Constraints

Remember: business logic is also enforced by the application code.

Database constraints are a way to make sure that the data follows the rules we have set up for the application.

We can also say that they *ensure the integrity of the data*.



Constraints

Constraints are used to enforce business rules.

Examples:

- Each person must have a name
- Two persons cannot have the same ID number

Creating a constraint

Let’s create a constraint on `:Skill(name)`:

```
CREATE CONSTRAINT ON (s:Skill)
ASSERT s.name IS UNIQUE
```

The MERGE clause

MERGE - Find or create

Let's try to create ourselves twice. What happens?

```
CREATE (s:Person {name: 'Your Name'})
```

MERGE - Find or create

```
MERGE (p:Person {name: 'Maria'})
RETURN p
```

This query does the following:

Search for a node with the label *Person* and a property called *name* with the value "Maria"

- If it finds it: **RETURN** it
- If it doesn't find it: **CREATE** it

MERGE - Find or create

```
MERGE (p:Person {name: 'Maria', nationality: "Swedish"})
RETURN p
```

There is not a `:Person` node with `name:'Maria'` and `nationality:"Swedish"` in the graph, but there is a `:Person` node with `name:'Maria'`.

What do you think will happen here?

The MERGE Clause

```
MERGE (p:Person {name: 'Maria'})
SET p.nationality = "Swedish"
RETURN p
```

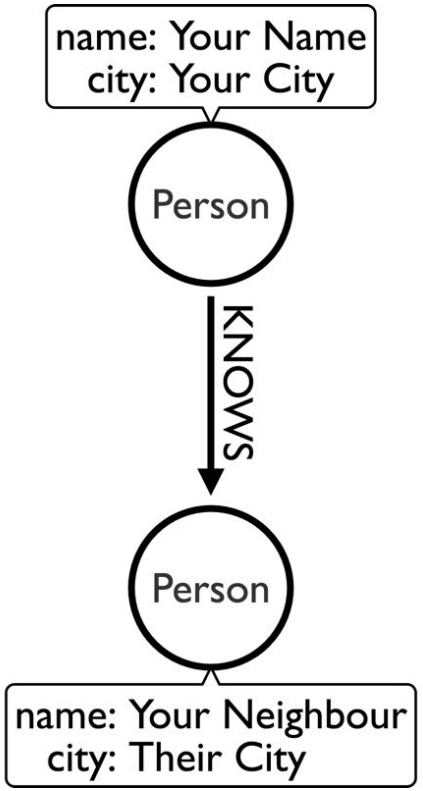
Merging relationships

To avoid **duplicate** relationships, use **MERGE**

```
MERGE (p1:Person {name: 'Your Name'})
MERGE (p2:Person {name: 'Your Neighbour'})
MERGE (p1)-[:KNOWS]->(p2)
```

Relationship is created uniquely by

- type
- direction (can be left off)
- properties



Your skills

Now let's associate you to some of those skills.

Can you write a query to add a **`HAS_SKILL`** relationship from you to whichever skills you have?

Hint: Don't forget to use the **MERGE** clause!

Your skills

Your query should look something like this:

```
MATCH (skill:Skill { name: 'Name of Skill' })
MATCH (me:Person { name: 'Your Name' })
MERGE (me)-[:HAS_SKILL]->(skill)
```

Multiple skills

What about if we want to do multiple skills in one query?

```
UNWIND ["skill 1", "skill 2", "skill 3"] as nameOfSkill
MATCH (skill:Skill { name: nameOfSkill })
MATCH (me:Person { name: 'Your Name' })
MERGE (me)-[:HAS_SKILL]->(skill)
```

Do your neighbours share any skills?

Now let's write a query to check if our neighbours have any of the same skills as us.

Do your neighbours share any skills?

Your query should look something like this:

```
MATCH (me:Person { name: 'Your Name' })-[:KNOWS]-(neighbour),
      (neighbour)-[:HAS_SKILL]->(skill)<-[:HAS_SKILL]-(me)
RETURN neighbour, skill
```

What skills do you want to learn?

Now let's associate you to some skills that you want to learn

Can you write a query to add a `WANTS_TO_LEARN` relationship from you to whichever skills you want to learn?

What skills do you want to learn?

Your query should look something like this:

```
MATCH (me:Person { name: 'Your Name' })
MATCH (skill:Skill { name: 'Name of Skill' })
MERGE (me)-[:WANTS_TO_LEARN]->(skill)
RETURN me, skill
```

Can your friends teach you?

Can you write a query to find out if any of your friends already have the skills that you want to learn?

Can your friends teach you?

Your query should look something like this:

```
MATCH (me:Person { name: 'Your Name' })
MATCH
(friend:Person)-[:HAS_SKILL]-(skill)-[:WANTS_TO_LEARN]-(me)
RETURN friend, skill
```