# LearningDataAnalysis

## Content



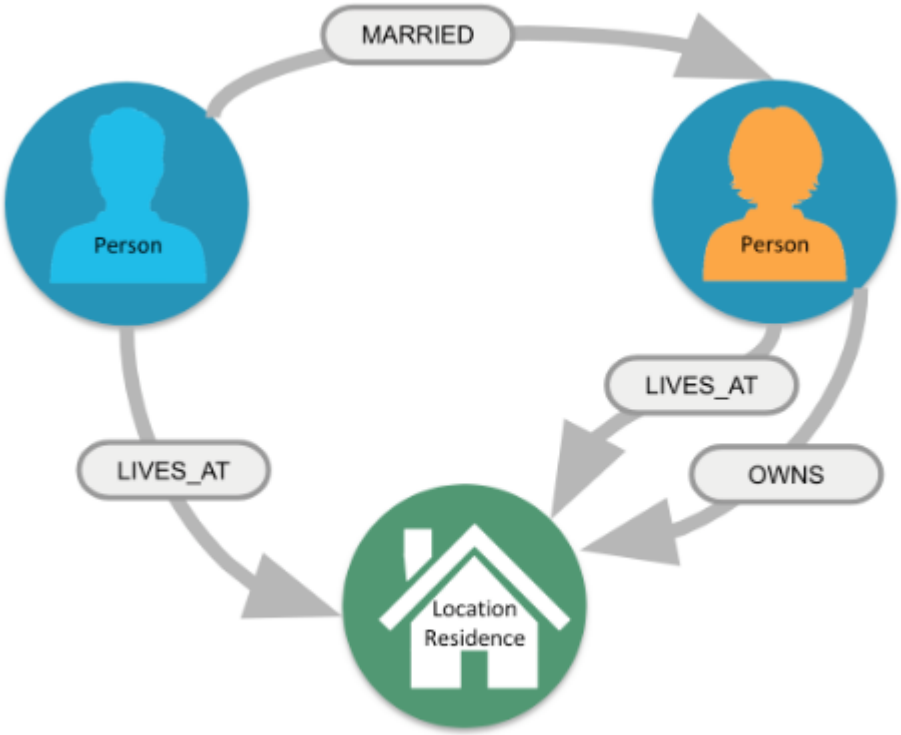
## Graph Database Fundamentals

A graph database can store any kind of data using a few simple concepts:

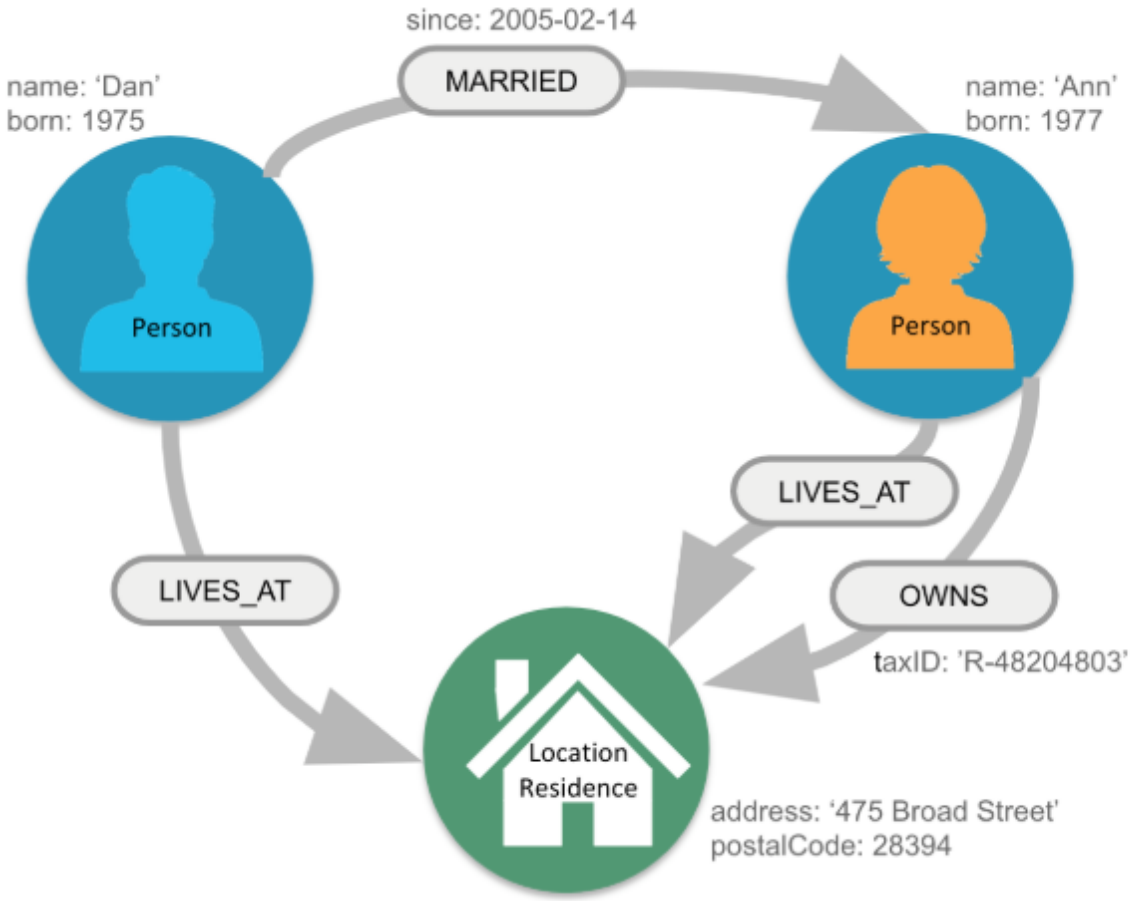
1. **Nodes** - graph data records

2. **Labels** - specifies the type of the node


3. **Relationships** - connect nodes




4. **Properties** - key-value pair properties

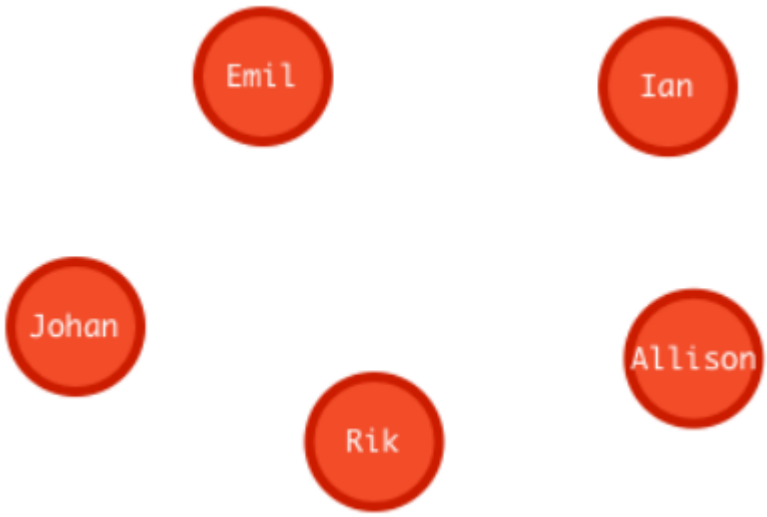



The simplest graph has just a single **node** with some named values called **Properties**:


Nodes are the name for data records in a graph and the data is stored as Properties that can be simple key-value pairs.

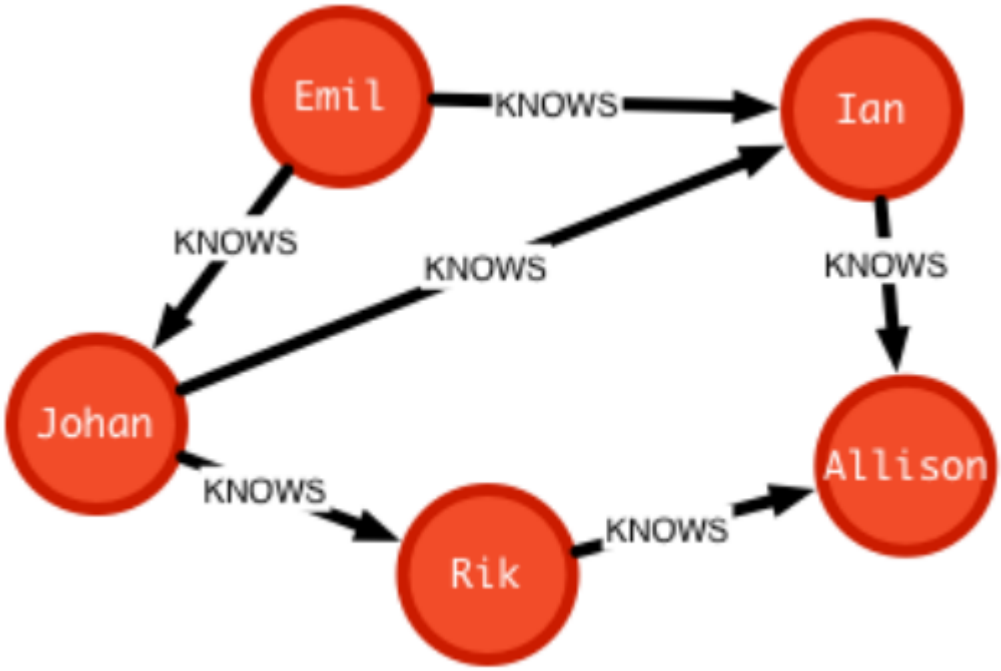
Nodes can be grouped together by applying a Label to each member. In the example above we can set to that node the label **Person**. Is important to know that a label is not a object and can't have any properties, is used only to categorize the nodes in a graph. A node can have zero or more labels based on the definition of that node.

To add more records we can simply add more nodes.

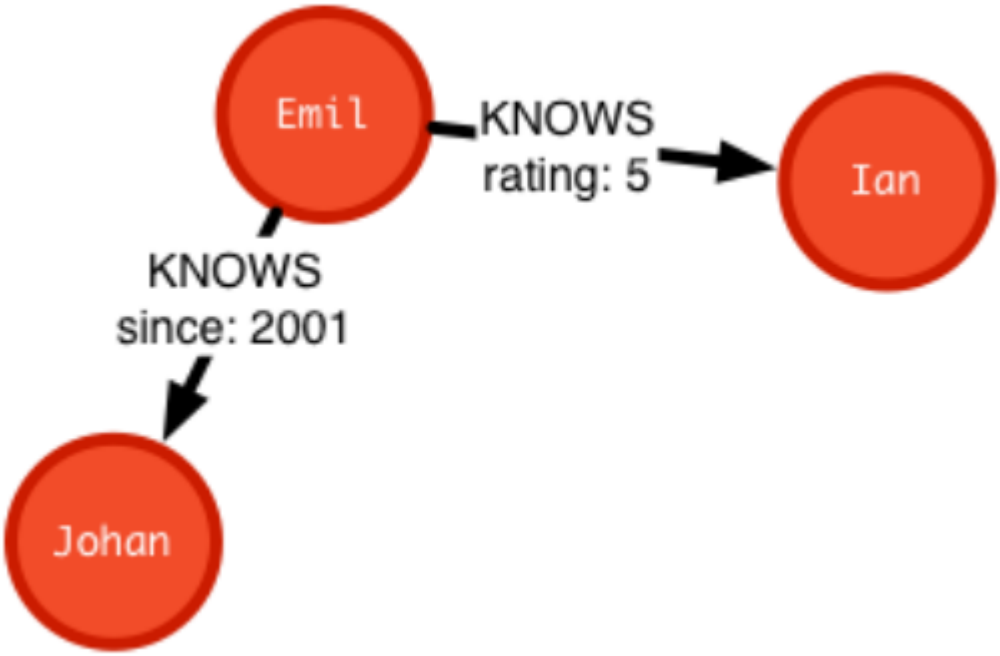



Similar nodes can have different properties with different type: string, number or even boolean. The dimension of a graph like this can be infinite because there is no limit to the number of nodes that can be added.

One of the properties of a database is to connect data, in a graph database the link is made by **Relationships**. To associate two nodes we can add **Relationship** between them wich describe how the records are related.




A relationship are data records that need to have two properties: **direction** and **type**, and can also contains properties like nodes.




A Graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model. Graph database are generally build for use with OLTP systems, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

Unlike the other databases, relationships take first priority in graph databases so the foreign keys or out-of-band processing is no more necessary to link a data to another.

By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build sophisticated models that map closely to out problem domain.

Many applications' data is modeled as relational data, indeed there are some similarities between a relational model nad a graph model:

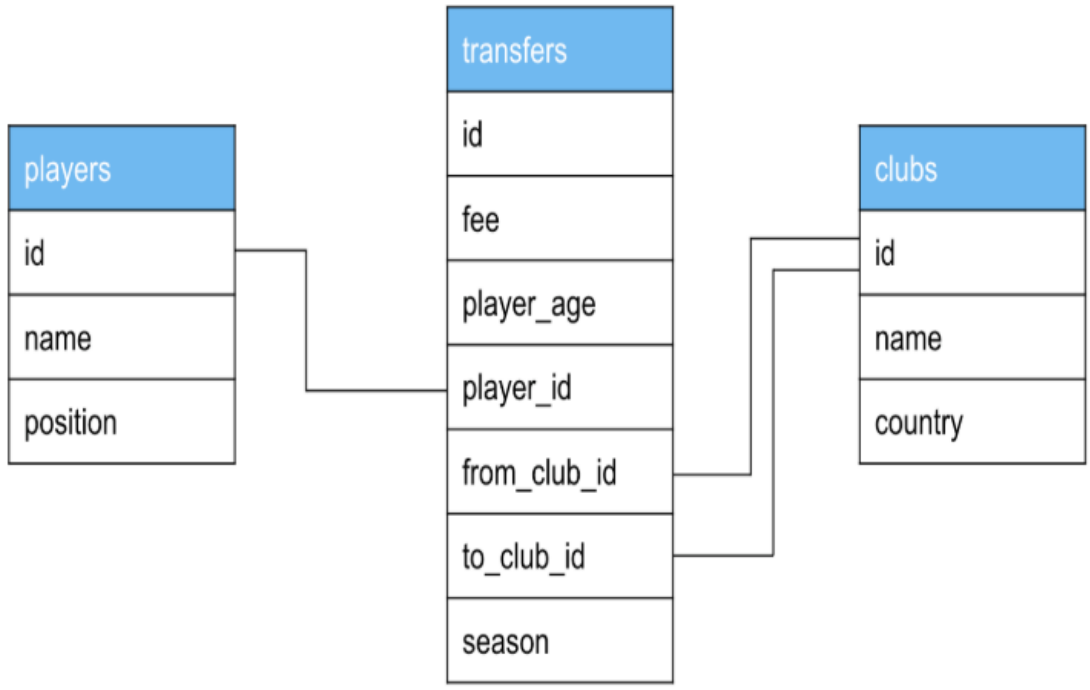
| **Relational** | **Graph** |
|---|---|
| Rows | Nodes |
| Joins | Relationships |
| Table names | Labels |
| columns | Properties |

There are even difference between this two databases:

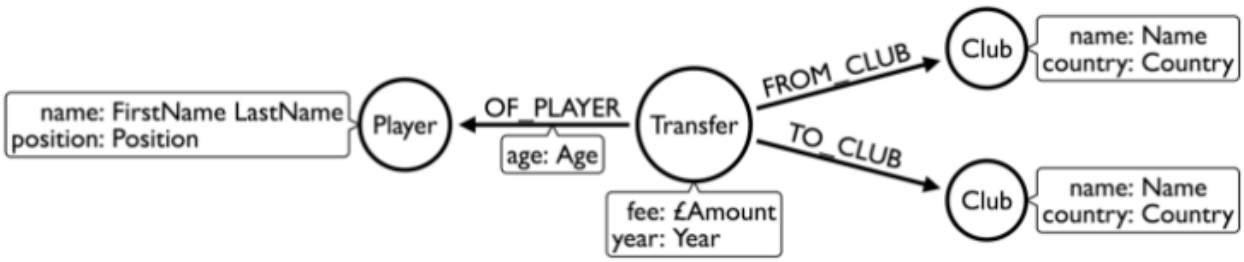
| **Relational** | **Graph** |
|---|---|
| Each column must have field value | Nodes with the same label arent' required to have the same set of properties |
| Joins are calculated at query time | Relationships are stored on disk when they are created |

| Relational | Graph |
| --- | --- |
| A row can belong to one table | A node can have many labels |
| Try to get the schema defined and then make minimal changes to it after that | It's common for the schema to evolve with the application |
| More abstract focus when modeling | Common to use actual data items when modeling |

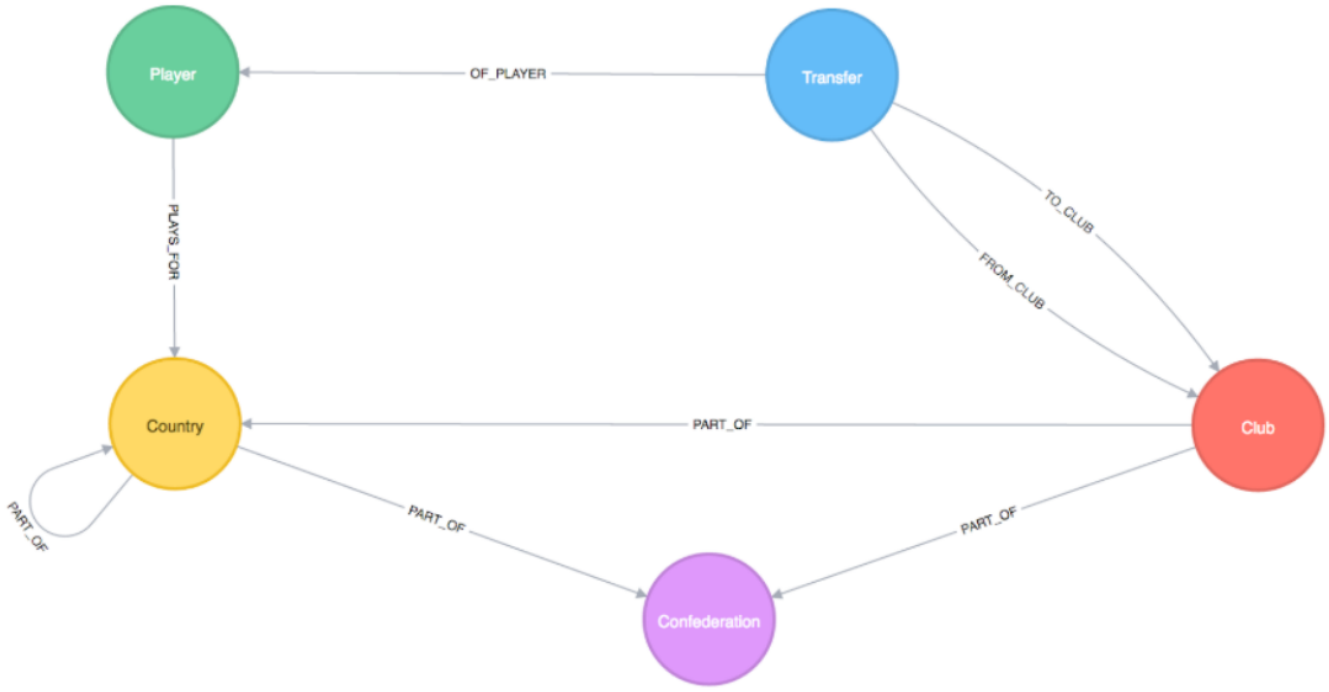
Here is the relational model:




And here is the correspondig graph model:




The graph model can be more versatile and can be upgrade without efforts, for example we want to add the confederation and country:




---

# Neo4J


video youtube

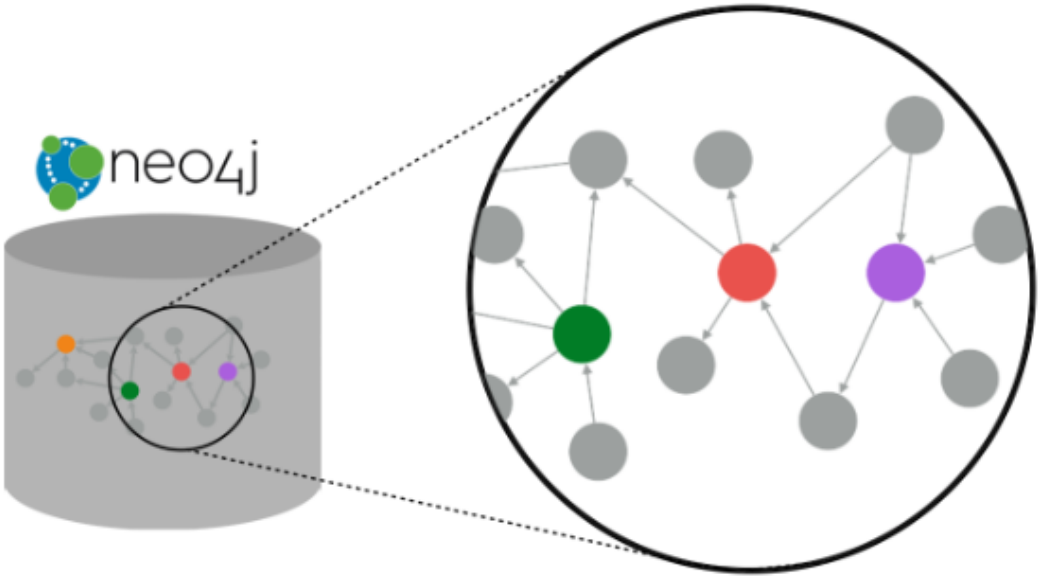
Connected information is everywhere in the world around us. Neo4j was build to efficiently store, handle, and query higly-connected data in your data model.

Neo4J is a high performance graph store with all the feature expected of a mature and robust database. The network structure is made by nodes and relationships rather than static tables.

Some definitions:

- Index free adjacency

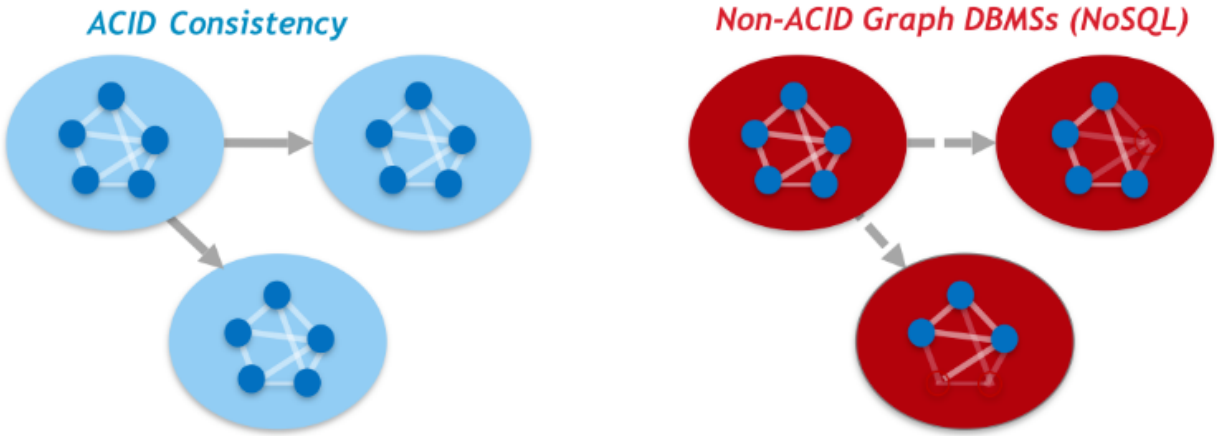
  With index free adjaceny, when a node or relationship is written to the database, it is stored in the database as connected and any subsequent access to the data is done using pointer navigation wich is very fast. Since Neo4j is a native graph database, it supports very large graphs where connected data can be traversed in constant time without the need for an index.




To know more read -> Index-free adjacency

- ACID

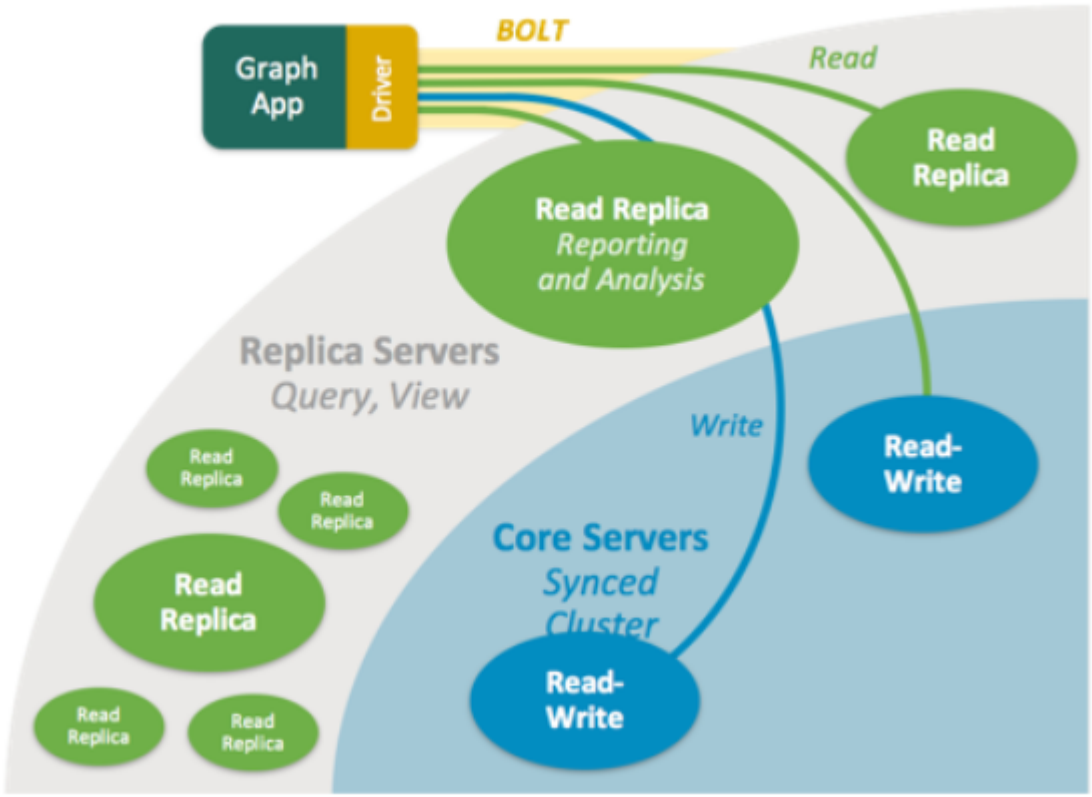
  Transactionality is very important for robust applications that require an atomicity, consistency, isolation, and durability guarantees for their data. If a relationship between nodes is created, not only is the relationship created, but the nodes are updated as connected. All of these updates to the database must all succeed or fail.




To know more read -> ACID

- Clusters

  Neo4j supports clusters that provide high availablity, scalability for read access to the data and failover which is important to many enterprises.




To know more read -> Cluster

- Graph engine

  The Neo4j graph engine is used to interpret Cypher statements and also executes kernel-level code to store and retrive data, whether it is on disk, or cached in memory.

- Bolt

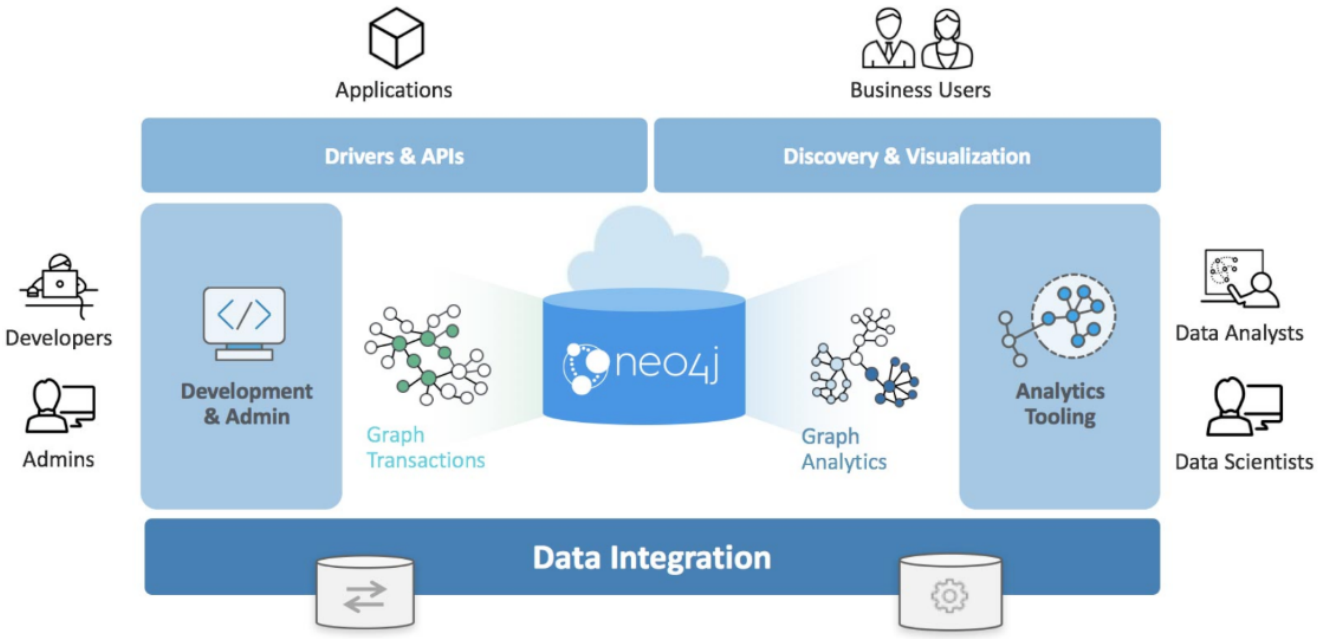
  Neo4j supports Java, JavaScript, Python, C#, and Go drivers that use Neo4j's bolt protocol for binary access to the database layer. Bolt is an efficiant binary protocol that compresses data sent over the wire as well encrypting the data. It's possible to create a java application that uses the bolt driver to access the Neo4j database and the application may use other packages that allow data integration between Neo4j and other data stores or uses as common framework such as spring.

- Tools

  Neo4j browser is an application that uses the JavaScript Bolt driver to access the graph engine of the Neo4j database server.

  Bloom enables you to visualize a graph without knowing much about Cypher (youtube video).

  ETL used to importing and exporting data between flat files and a neo4j Database.




To use Neo4j there are two options:

- desktop application

  " The Neo4j Desktop includes the Neo4j Database server which includes the graph engine and kernel so that Cypher statements can be executed to access a database on your system. It includes an application called Neo4j Browser. Neo4j Browser enables you to access a Neo4j database using Cypher. You can also call built-in procedures that communicate with the database server. There are a number of additional libraries and drivers for accessing the Neo4j database from Cypher or from another programming language that you can install in your development environment. If you are looking to use your system for application development and you want to be able to create multiple Neo4j databases on your machine, you should consider downloading the Neo4j Desktop (free download). The Neo4j Desktop runs on OS X, Linux, and Windows. "

  How to use on:

  - OSX: youtube video
  - Windows: youtube video
  - Linux: youtube video

- browser sandbox

  " The Neo4j sandbox is another way that you can begin development with Neo4j. It is a temporary, cloud-based instance of a Neo4j Server with its associated graph that you can access from any Web browser. The database in a Sandbox may be blank or it may be pre-populated. It is started automatically for you when you create the Sandbox.

  By default, the Neo4j sandbox is available for three days, but you can extend it for up to 10 days. If you do not want to install Neo4j Desktop on your system, consider creating a Neo4j sandbox. You must make sure that you extend your lease of the sandbox, otherwise you will lose your graph and any saved Cypher scripts you have created in the sandbox. However, you can use Neo4j Browser Sync to save

Cypher scripts from your sandbox. We recommend you use the Desktop for a real development project. The Sandbox is intended as a temporary environment or for learning about the features of Neo4j as well as specific graph use-cases. "

youtube video - Creating a Neo4j Sandbox

Both of them use Neo4j Browser application to perform querying in the database -> GettingStartedBrowser

---

## Cypher

***This notes below can be read on neo4J browser sandbox by type the command: :play cypher***

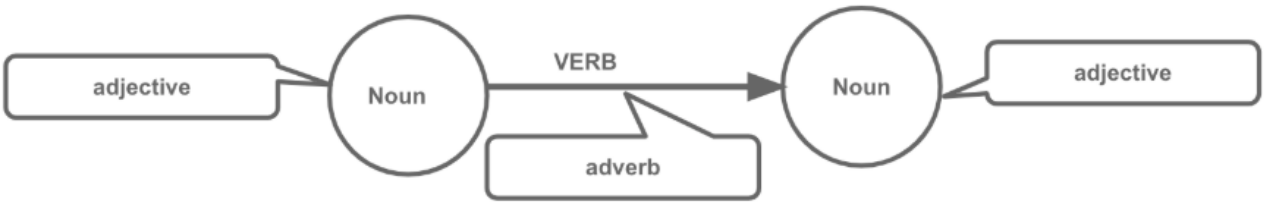
***All of the query are run in the Neo4J browser sandbox***

Neo4J's Cypher language is purpose built for working with graph data, is a declarative query language that allows for expressive and efficient querying and updating of graph data. It uses patterns to describe graph data and is familiar to sql-like clauses. This query language allows users to store and retrive data from the Neo4J graph database with a visual and logical syntax to match patterns of nodes and relationships in the graphs. It allow to state what we want to select, insert, update, or delete from our graph data without a description of exactly how to do it:

**" Describing what to find and not how to find it "**

This means that complex database queries can easily be expressed through Cypher, allowing you to focus on your domain instead of getting lost in the syntax of database access. Also give an expressive and efficient queries to handle needed create, read, update, and delete functionality (also know as CRUD operations).

The unwritten rule wants to rappresents the nouns as the nodes of the graph, the verbs as the relationships, the adjectives and adverbs are the properties:




Graph patterns are expressed in Cypher using ASCII-art like syntax to make queries more self-explanatory:

- **NODES** uses a pair of parentheses like `()` or `(node)` to rapresent a node, similar to a circle on witheboard. An anonymous node `()` represents one or more nodes during a query processing where there are no restrictions of the type of the node, a name inside the parentheses `(node)` tells the query processor that for this query is used the variable called `node` to rapresents all the nodes of the graph.
- **LABELS** are used to group nodes and filter queries against the graph and is defined with a colon `(:Label)`. A node can have zero or more labels for example `(node)`, `(node:Label)`, `(node:Label1:Label2)`, `(:Label)`, `(:Label1:Label2)`.
- **RELATIONSHIPS** are defined within square brackets `[]` and optionally we can specify type and direction like `()<-[:RELATIONSHIP]-()`.

- **ALIASES** are used to referred elements to later in the query defined by a name before a name like `(node1:Label1)<-[relationship:RELATIONSHIP]-(node2:Label2)` where node1, node2 and relationship are aliases.
- **Predicates** are filters that can be applied to limit the matching paths: boolean logi operators, regular expressions and string comparison operators.

The properties of a node are accessed using `{variable}.{property_key}`, for example `emil.name` or `movie.title`.

The Cypher language are case insensitive and sensitive:

| Sensitive | Insensitive |
|---|---|
| Node labels | Cypher keywords |
| Relationship type | ----- |
| Property keys | ---- |

Later on the cypher keywords are upper-case, this is a coding convention and is described in the Cypher Style Guyide.

---

#### Part one

---

### Comments

You can place comments anywhere in the query and to specify that the rest of the line is interpreted as a comment you need to put a double slash `// comment`.

### Match

#### *on neof4j browser run the command `:help MATCH`*

youtube video - how to execute a MATCH statement

The most widely used Cypher clause is **MATCH**, this performs a pattern match against the data in the graph. During the query processing, the graph engine traverses the graph to find all nodes that match the graph pattern.

A query with match need to be present with the **RETURN** clause. This clause must be the last of a query to the graph. Here some examples:

```
// returns all nodes in the graph
MATCH (variable)
RETURN variable
```

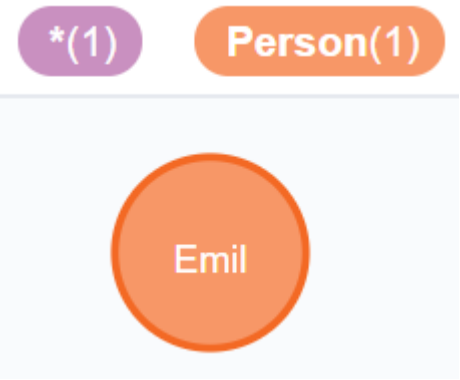
```
// returns all Label nodes in the graph
MATCH (variable:Label)
RETURN variable // returns
```

When you specify a pattern for a **MATCH** clause, you should always specify a node label if possible. In doing so, the graph engine uses an index to retrive the nodes which will perform better than not using a label for the **MATCH**.

**Type of query output**

The output of a query can be different:

- by **graph**:




- by **table**:

```
{
    "name": "Emil",
    "from": "Sweden",
    "klout": 99
}
```

- by **text**:

```
╒══════════════════════════════════════════╕
│"ee"                                      │
╞══════════════════════════════════════════╡
│{"name":"Emil","from":"Sweden","klout":99}│
└──────────────────────────────────────────┘
```

## Exercises part one

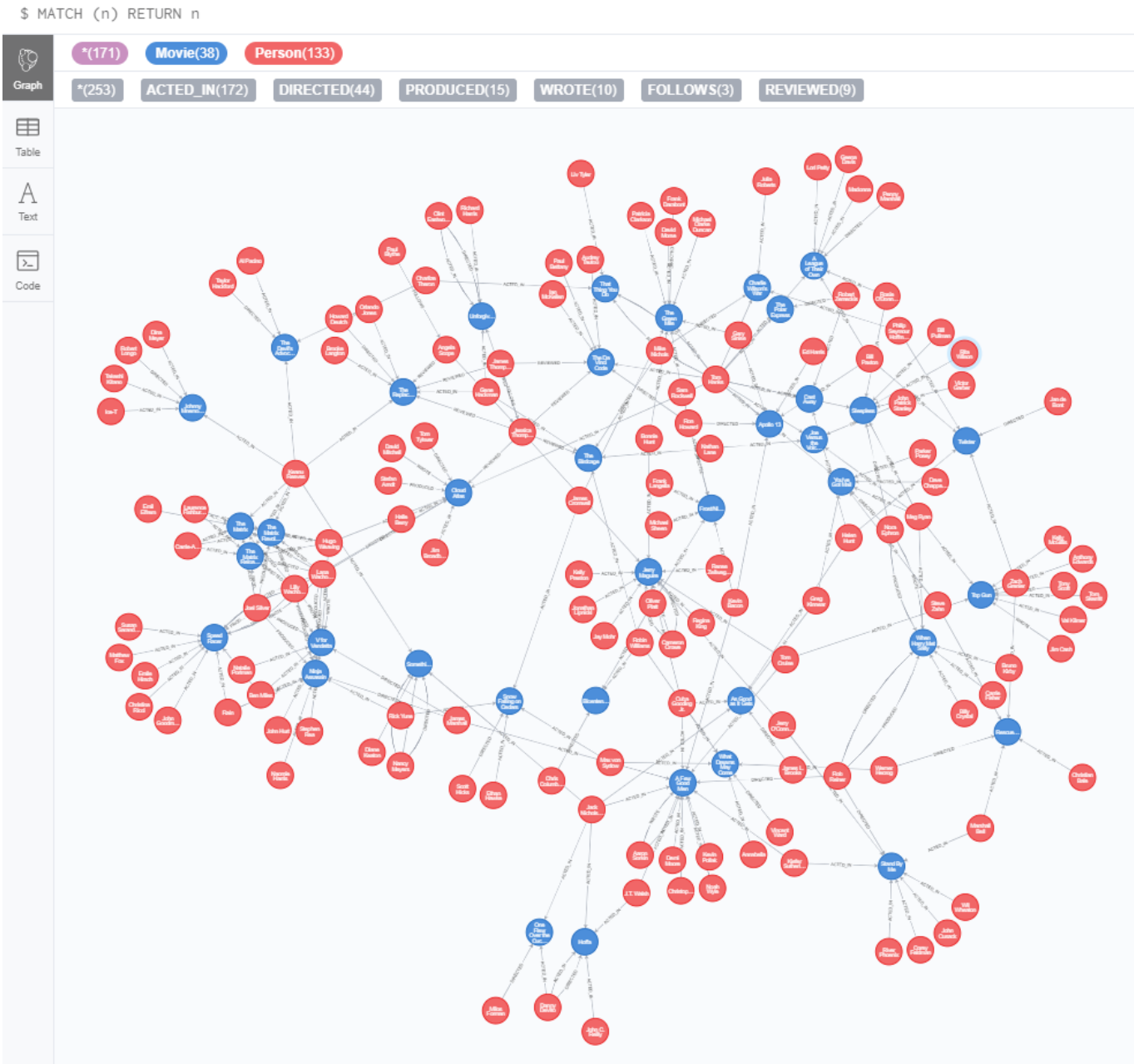
***on neof4j browser run the command*** *:play intro-neo4j-exercises* ***and follow exercise 1 instructions***

First of all use the script found at Cypher/exercises/part_one/createGraph.cql to create the basic graph:

```
Added 171 labels, created 171 nodes, set 564 properties, created 253
relationships, completed after 24 ms.
```

Exercise 1.1 Retrive all nodes from the database

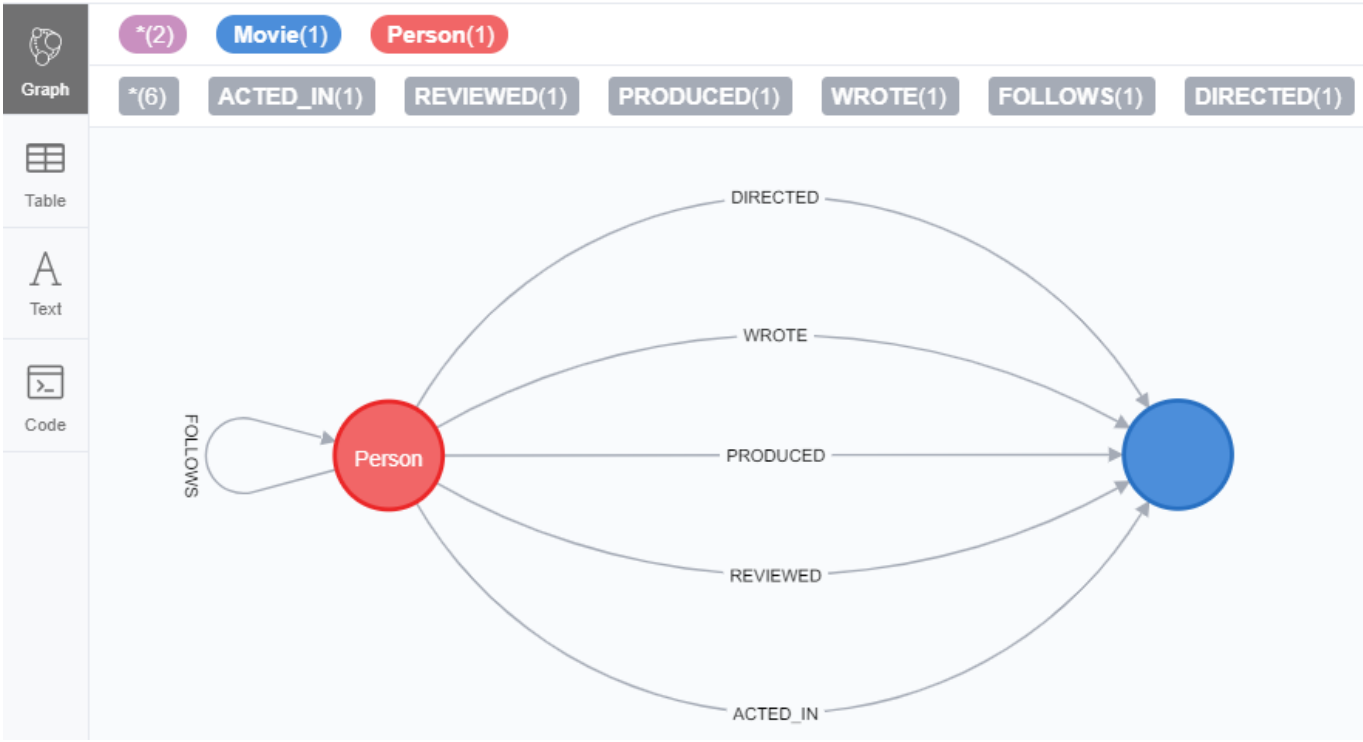
```
MATCH (n)
RETURN n
```




Exercise 1.2 Examine the schema of your database

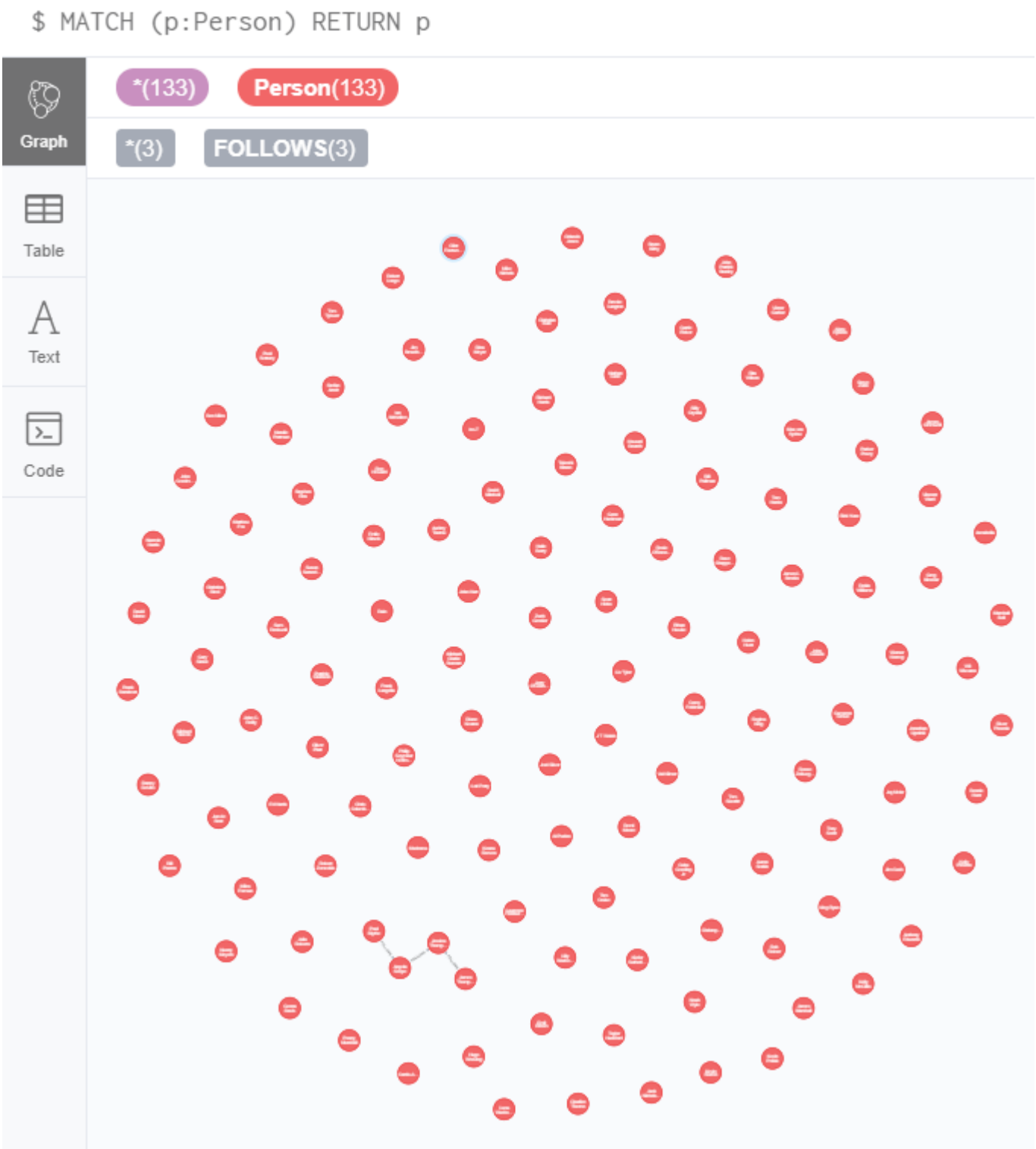
```
CALL db.schema()
```

```
$ CALL db.schema()
```



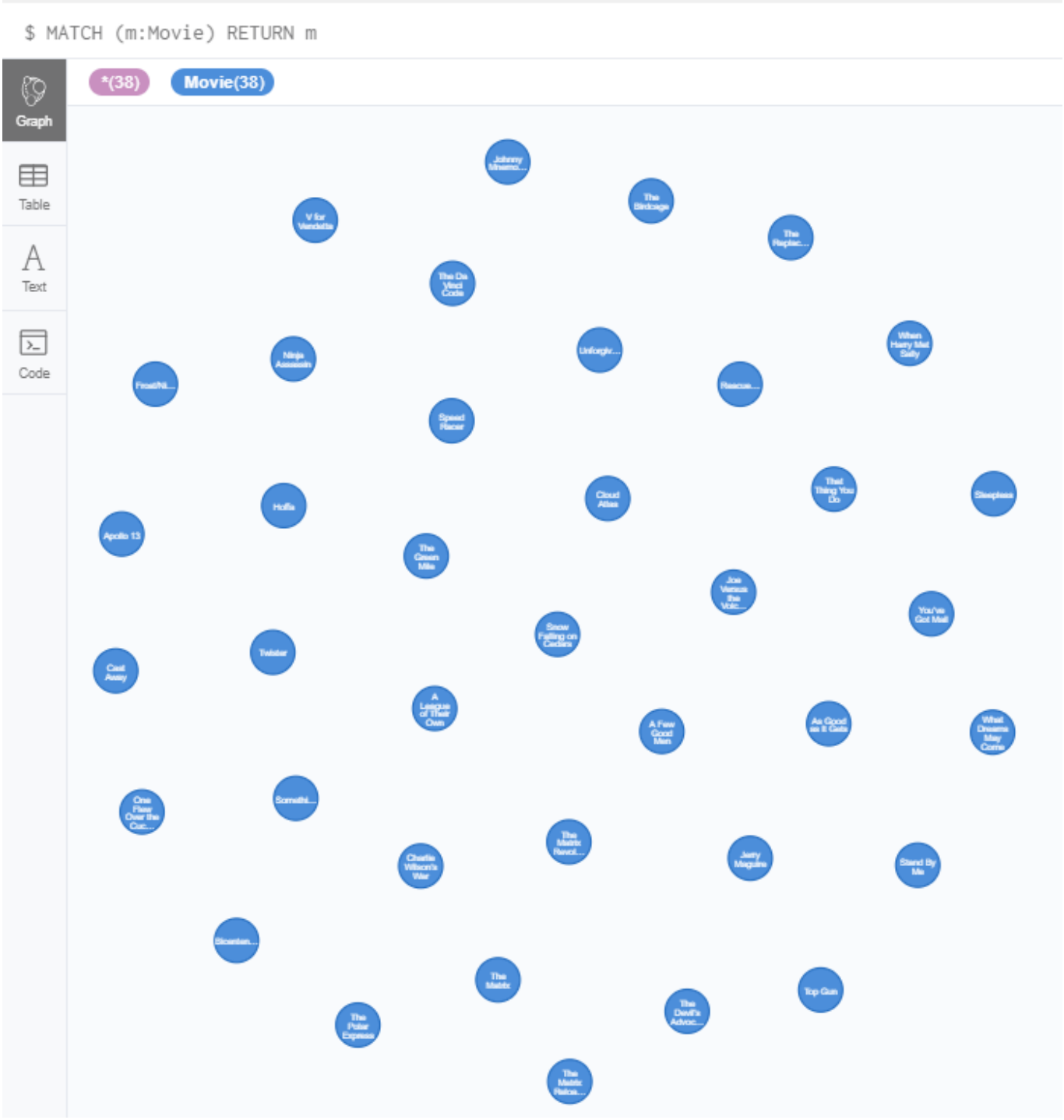

Exercise 1.3 Retrive all Person nodes

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




Exercise 1.4 Retrive all Movie nodes

```
MATCH (m:Movie)
RETURN m
```




---

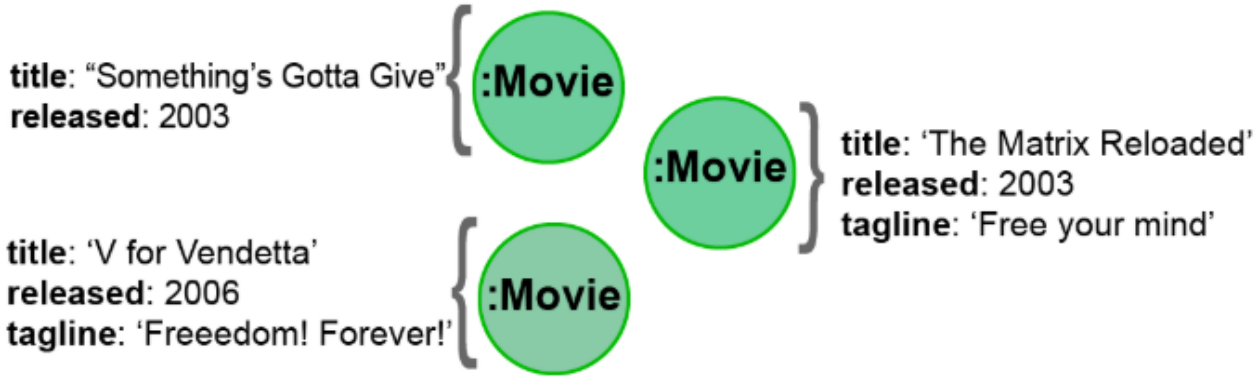
**Part two**

---

## Properties

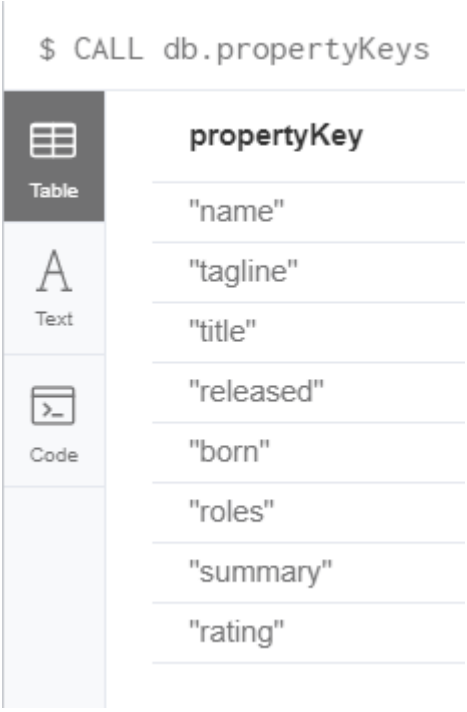
In Neo4j a node can have properties that are used for further define a node. A property is identified with a key and defined for a node and not for a type of node. All nodes of the same type need not have the same properties.

For example in the Movie graph all Movie nodes have both title and released properties, however it is not requirement that every Movie node has a property tagline:




Properties can be used to filter queries so that a subset of the graph is retrived. In addition, with the `RETURN` clause, you can return property values from the retrived nodes, rather than the nodes.

The propertiy keys of a graph can be view by execute `CALL db.propertyKeys` which call the Neo4j library method that returns the property keys for the graph. For example run this command in the movie graph returns the result stream contains all property keys in the graph:




**Nodes properties filtering**

youtube video - using match to return propery values

It's possible to filter the nodes of the graph to a specify a value for a property, any node that matches the value will be retrived. Here some examples:

```
MATCH (variable {propertyKey: propertyValue})
RETURN variable
```

```
MATCH (variable:Label {propertyKey: propertyValue})
RETURN variable
```

```
MATCH (variable:Label {propertyKey1: propertyValue1, propertyKey2:
propertyValue2})
RETURN variable
```

It's possible to retrive a property values of nodes in a query and return on output:

```
MATCH (variable {property1: value})
RETURN variable.property2
```

```
MATCH (variable:Label {property1: value})
RETURN variable.property2
```

```
MATCH (variable:Label {property1: value, property2: value})
RETURN variable.property2, variable.property3
```

In the graph database we can filter the person born on 1970:

```
MATCH (p:Person {born: 1970})
RETURN p.name, p.born
```

**Aliases**

To customize the headings for a table containing property value it can be use aliases:

```
MATCH (variable:Label {property1: value, property2: value})
RETURN variable.property2 AS alias1, variable.property3 AS alias2
```

In the graph database we can specify aliases for the returned property values:

```
MATCH (p:Person {born: 1970})
RETURN p.name AS name, p.born AS `birth year`
```

## Exercises part two

***on neof4j browser run the command** `:play intro-neo4j-exercises` **and follow exercise 2 instructions***

First of all use the script found at Cypher/exercises/part_one/createGraph.cql to create the basic graph:

```
Added 171 labels, created 171 nodes, set 564 properties, created 253
relationships, completed after 24 ms.
```

Exercise 2.1: Retrieve all Movie nodes that have a released property value of 2003.

```
MATCH (m:Movie {released: 2003})
RETURN m
```

Exercise 2.2: View the retrieved results as a table.

```
MATCH (m:Movie {released: 2003})
RETURN m
```

Exercise 2.3: Query the database for all property keys.

```
CALL db.propertyKeys
```

Exercise 2.4: Retrieve all Movies released in a specific year, returning their titles.

```
MATCH (m:Movie {released: 2006})
RETURN m.title
```

Exercise 2.5: Display title, released, and tagline values for every Movie node in the graph.

```
MATCH (m:Movie)
RETURN m.title, m.released, m.tagline
```

Exercise 2.6: Display more user-friendly headers in the table.

```
MATCH (m:Movie)
RETURN m.title AS `Movie title`, m.released AS `Released date`, m.tagline AS `Tag
line`
```

---

**Part three**

---

### Where

***on neof4j browser run the command*** `:help WHERE`

There are more method to filter the nodes of a match clause: by specify the value of the argumento on the match clause or by using the WHERE caluse. This clause is the answer for "how we filter the result for a particular match", so this filter all of the nodes and relationships. Some examples:

1.
```
MATCH (m:Movie {title: "The Matrix"})
RETURN m
```

2.
```
MATCH (m:Movie)
WHERE m.title = "The Matrix"
RETURN m
```

3.
```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE m.released >= 2000
RETURN m.released, a.name
```

It can be use several comparison operators: **=**, **<>**, **<**, **>**, **<=**, **>=**, **IS NULL**, **IS NOT NULL**, **=~**. There are 4 boolean operators that it can use: **AND**, **OR**, **XOR**, **NOT**. An example:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE
    m.released > 2000 OR
    (m.released = 1997 AND m.title='As Good as It Gets')
RETURN p.name, m.title, m.released
```

### Create

***on neof4j browser run the command*** `:help CREATE`

Let's create a small social graph using this query language.

To create a new data we use the **CREATE** clause:

```
CREATE (ee:Person {name: "Emil", from: "Sweden", klout:99})
```

***klout = influence based on the ability to drive action across the social web***

With this query we create a node **ee** of type Person that have 3 properties: name, from and klout.

The output of this query will be:

```
Added 1 label, created 1 node, set 3 properties, completed after 134 ms.
```

**Null**

Null represents missing or undefined values. You do not store a null value in a property. It just doesen't exist on that particular node. **Warning: null=null is not true but the result will be null because we don't know the value of a null propertie**

**More create at once**

**Create** clauses can create many nodes and relationships at once:

```
MATCH (ee:Person) WHERE ee.name = "Emil"
CREATE
(js:Person {name:"Johan", from:"Sweden", learn:"surfing"}),
(ir:Person {name:"Ian", from:"England", title:"author"}),
(rvb:Person {name:"Rik", from:"Belgium", pet:"Orval"}),
(ally:Person {name:"Allison", from: "California", hobby: "surfing" }),
(ee)-[:KNOWS {since: 2001}]->(js),
(ee)-[:KNOWS {rating:5}]->(ir),
(js)-[:KNOWS]->(ir), (js)-[:KNOWS]->(rvb),
(ir)-[:KNOWS]->(js), (ir)-[:KNOWS]->(ally),
(rvb)-[:KNOWS]->(ally)
```

If exits in the database a node with **name** property with value **Emil** then create 4 new nodes and 7 relationship between them. The relationships are created by defined the left node and the right node:

```
(left_node)-[:NAME_OF_RELATIONSHIP {name_of_property: value_of_property}]->
(right_node)
```

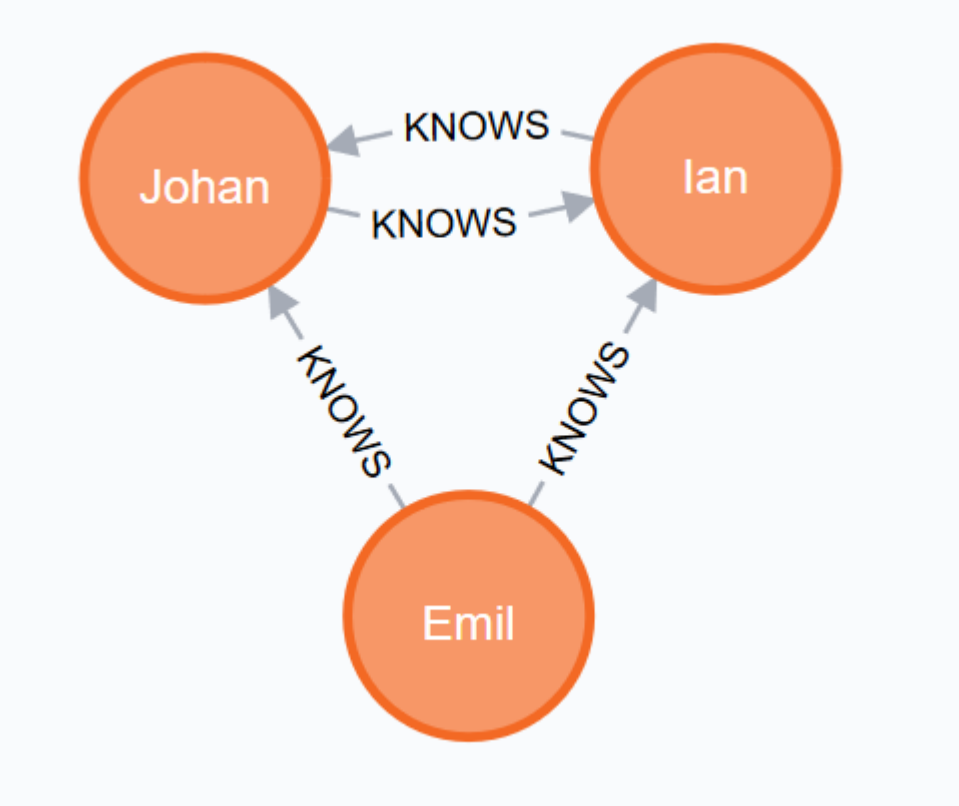
The output of the query above is:

```
Added 4 labels, created 4 nodes, set 14 properties, created 7 relationships,
completed after 33 ms.
```

To see what we created we need to run a new query that take all of the nodes in relationship with **Emil**:

```
MATCH (ee:Person)-[:KNOWS]-(friends)
WHERE ee.name="Emil"
RETURN ee.name, friends
```

Analyze this clause `MATCH (ee:Person)-[:KNOWS]-(friends)`, the meaning of `()-[:KNOWS]-()` is to maches **KNOWS** relationship in either direction and takes all nodes with label Person that have a relationship with other nodes. The output will be all the relationships between node **ee** with property name set as Emil and nodes **friends**, the query graph result will be:




**Distinct**

Pattern matching can be used to make recommendations, to suggest a new friend or a new film to watch. For example we can make recomandation to johan that is learning to surf, he may want to find a new frend who already does:

```
MATCH (js:Person)-[:KNOWS]-()-[:KNOWS]-(surfer)
WHERE js.name = "Johan" AND surfer.hobby = "surfing"
RETURN DISTINCT surfer
```

The clause **DISTINCT** is use to avoid an output of the same nodes because more than one Person can be friend to the same Person at the same time more than one nodes can be related to the same node. The pattern put on the **MATCH** clause contains 2 relations and the nodes in the center `()` is not important in our recommendation so is ignored and not referred with a name. This query return all of the Person who have the hobby "surfing" that are connected to a friend of a friend of Johan. In our database only one node correspond to this filter: "Allison".


### Explain Profile

***on neof4j browser run the command*** `:help EXPLAIN`

To understand how the query works you can use the **EXPLAIN** or **PROFILE** clause put at the beginning of the query, like this:

```
PROFILE MATCH (js:Person)-[:KNOWS]-()-[:KNOWS]-(surfer)
WHERE js.name = "Johan" AND surfer.hobby = "surfing"
RETURN DISTINCT surfer
```

The outcome will be a cause effect of how the engine find the result.

### Set

***on neof4j browser run the command*** `:help SET`

To change or add properties of a node it's possible to use the SET clause, it can be use 2 format: JSON or OBJECT.

- Json

```
MATCH
    (a:Person)-[:DRIVES]->(c:Car)
WHERE
    a.name = 'Ann'
SET
    c.brand = 'Volvo'
    c.model = 'V70'
RETURN
    c
```

- OBJECT

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

```
    c += {brand: 'Volvo', model:'V70'}
RETURN
    c
```

### Aggregates

We implicitly group by any non-aggregate fields in the RETURN statement

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, count(*) AS numberOfMovies
```

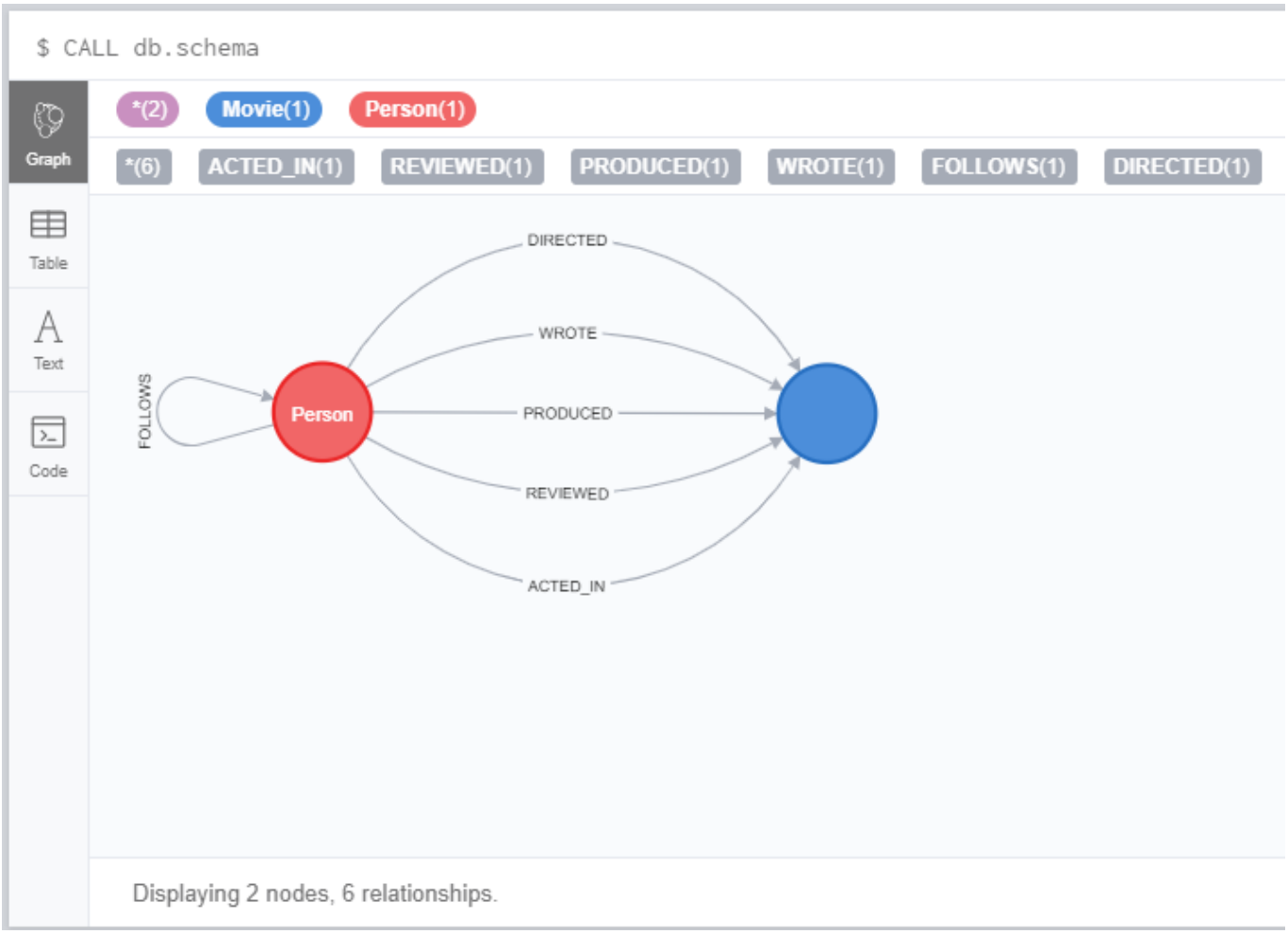
This query is grouped by name and aggregate the numberOfMovie associated by the same name of node.

There are a plenty of procedure for aggregations, check for more with apoc reference:

- github -> github.com/neo4j-contrib/neo4j-apoc-procedures
- neo4j docs -> neo4j-contrib.github.io/neo4j-apoc-procedures/

### Examining the data model

It is helpful to examine the data model of the graph, it can be done by executing `CALL db.schema` which calls the Neo4j procedure that returns information about the nodes, labels, and relationship in the graph. If we run this command in the movie graph the result will be:



**Libraries**

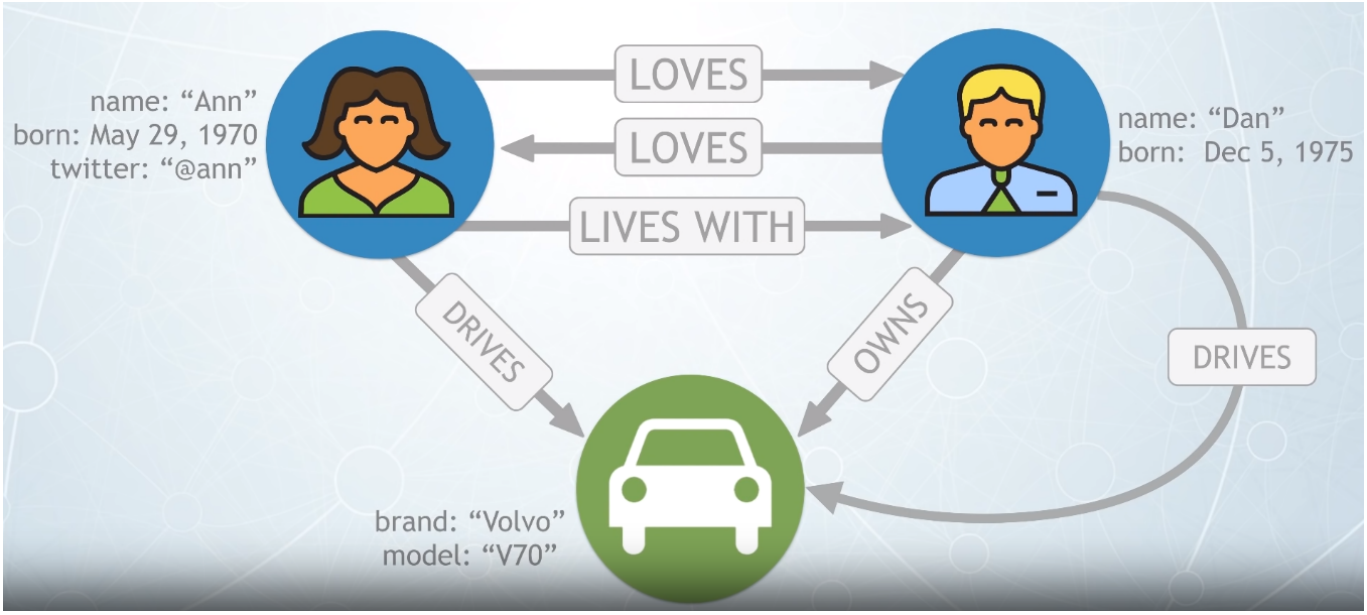
Neo4j has a published, opnen source Cypher library, Awesome Procedures on Cypher (APOC) that contain many useful procedures you can call from Cypher. Another Cypher library is the Graph Algorithms library to help users to analyze data in graphs.

---

## Example - Simple Graph

***This example below is found on lesson 5 neo4j***

Let's create this graph:



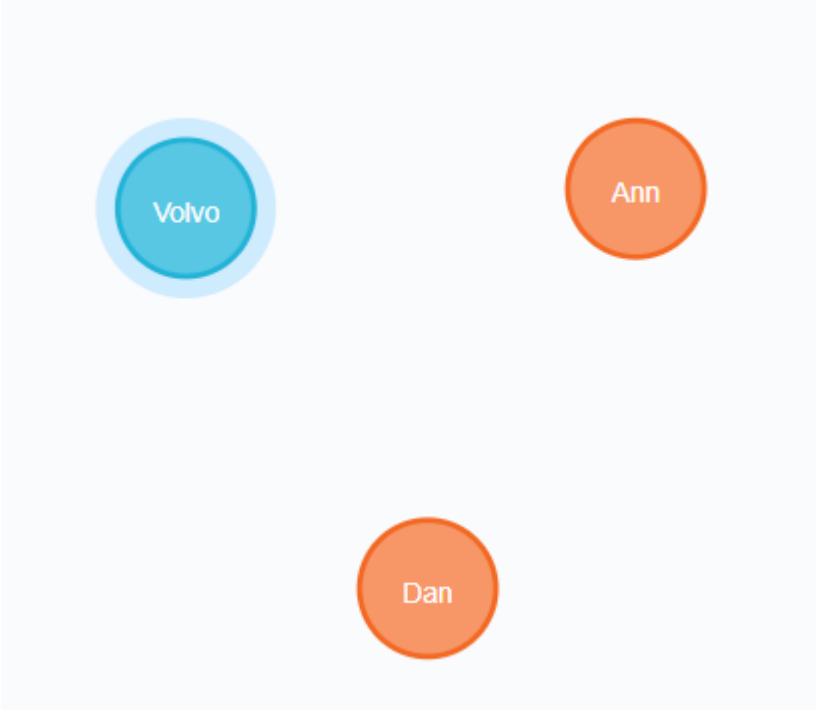

To see the graph result you can use this query:

```
MATCH (p)
RETURN p
```

- Creation of the nodes:

  ```
  CREATE (dan:Person {name:"Dan", born:"Dec 5, 1975"})
  CREATE (ann:Person {name:"Ann", born: "May 29, 1970", twitter: "@ann"})
  CREATE (car:Car {brand:"Volvo", model:"V70"})
  ```

  result:

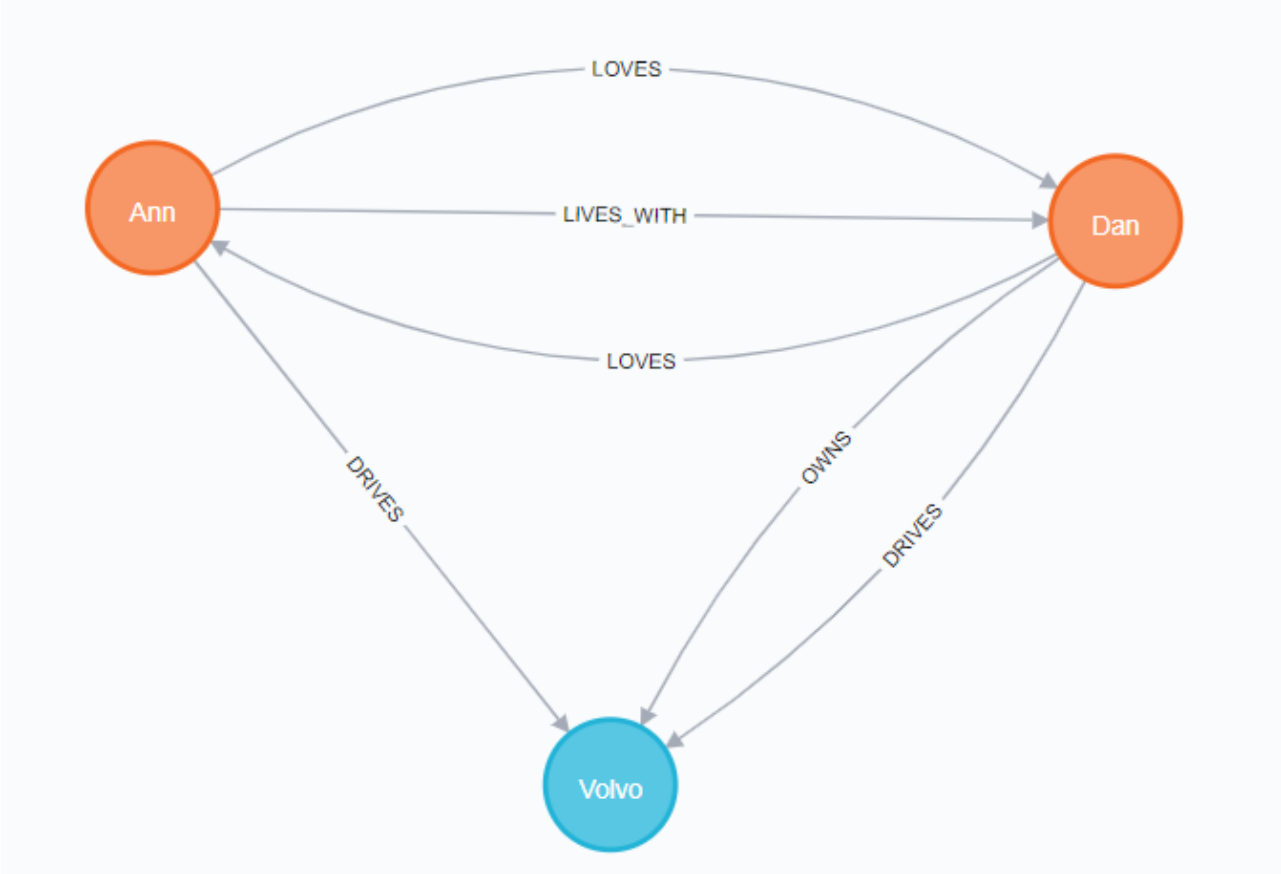



***Query find on:*** ***creationNodes***

- Creation of realtionships:

```
MATCH
        (dan:Person {name: "Dan"}),
    (ann:Person {name: "Ann"}),
    (car:Car {model:"V70"})
CREATE
    (dan)-[:LOVES]->(ann),
    (ann)-[:LOVES]->(dan),
    (ann)-[:LIVES_WITH]->(dan),
    (dan)-[:OWNS]->(car),
    (dan)-[:DRIVES]->(car),
    (ann)-[:DRIVES]->(car)
```

result:




***Query find on:*** ***creationRelationships***

- Basic query

    1. "Find who loves Ann":

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

    The result is simple "Dan".

    ***Query find on:*** ***lovesAnn***

    2. "Find the Ann's Car":

    ```
    MATCH
        (ann:Person {name: "Ann"})-[:DRIVES]-(car:Car)
    RETURN
        car
    ```

    The result is simple "Volvo".

    ***Query find on:*** ***annCar***

3. "Find the Dan's Volvo car and update value of that car with the number of the wheels":

```
MATCH
    (ann:Person {name: "Dan"})-[:OWNS]-(car:Car)
WHERE
    car.brand="Volvo"
SET
    car.wheels=4
RETURN
    car.wheels
```

The result is simple 4.

***Query find on:*** ***updateDanCar***

- Ensuring uniqueness: We don't want a bunch of nodes rapresenting the same object so to prevent this we can use the constraint on and unique properties

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

If we create a node Person with the same value of name of others than neo4j throws error. An example:

```
CREATE (a:Person {name:"Ann"})
CREATE (a)-[:HAS_PET]->(:Dog {name:"Sam"})
```

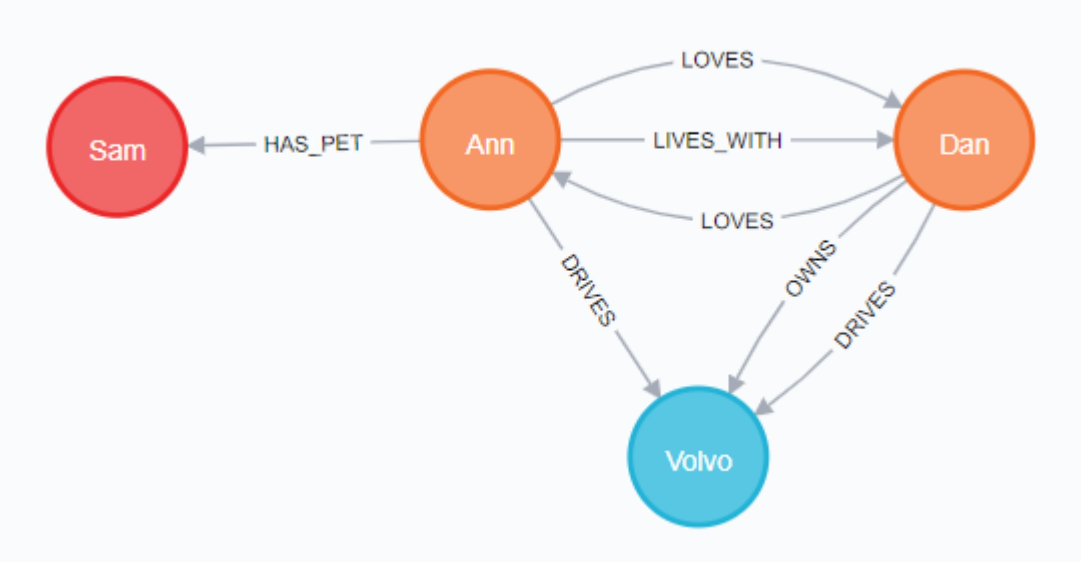
Output:

```
Neo.ClientError.Schema.ConstraintValidationFailed: Node(1) already exists with
label `Person` and property `name` = 'Ann'
```

To prevent this error we can use the MERGE clause:

```
MERGE (a:Person {name:"Ann"})
CREATE (a)-[:HAS_PET]->(:Dog {name:"Sam"})
```

The node with name Ann will be created if is not exists and if exists then it not create that. The new graph will be:




In the same way if the dog already exists the above query will give you an error, to prevent is necessary use MERGE instead of CREATE for the Dog creation. An example:

```
MERGE (a:Person {name:"Ann"})
ON CREATE SET
    a.facebook = "@annie"
MERGE (a)-[:HAS_PET]->(:Dog {name:"Sam"})
```

On create set will be use only if the merge create the node, if you run this query before all the queries above the facebook parameter will not be created.

***Query find on: annDogCreation***

---

## Application - Movie Graph

***on neof4j browser run the command*** `:play movie graph`

Let's create a more complex example with a mini application containing actors and directors that are related through the movies they've collaborated on.

We can create more than one nodes and relationships with a single block query statement composed of multiple **CREATE** clauses:

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to
the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne', born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (LillyW:Person {name:'Lilly Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
```

```
CREATE
  (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix),
  (LillyW)-[:DIRECTED]->(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),
  (JoelS)-[:PRODUCED]->(TheMatrix)
```

***You can find all the query block here: Cypher/MovieGraph/createGraph.cql***

***Source query of movie graph -> here***

The script add lot of nodes with relative relationships and properties, with this we can test some query:

1. Find the actor named "TOM Hanks"

```
MATCH (tom {name: "Tom Hanks"})
RETURN tom
```

or

```
MATCH (tom:Person)
WHERE tom.name="Tom Hanks"
RETURN tom
```

2. Find the movie with title "Cloud Atlas"

```
MATCH (cloud:Movie)
WHERE cloud.title="Cloud Atlas"
RETURN cloud
```

or

```
MATCH (cloud:Movie {title: "Cloud Atlas"})
RETURN cloud
```

3. Find 10 people

```
MATCH (people:Person)
RETURN people.name
LIMIT 10
```

The ***LIMIT*** clause is use to limitate the number of nodes in output.

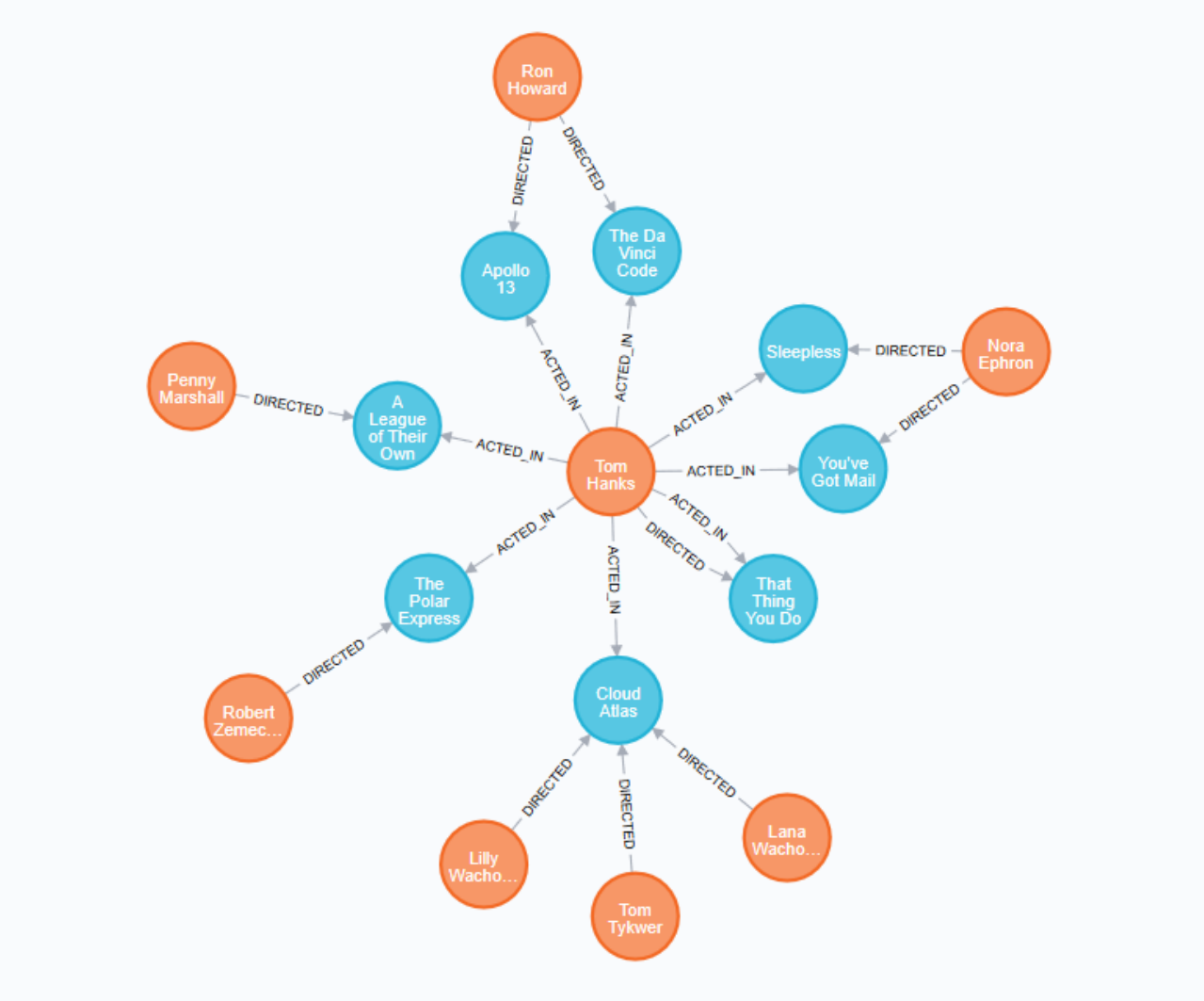
4. Find movies released in the 1990s

```
MATCH (movie:Movie)
WHERE movie.released >= 1990 AND movie.released < 2000
RETURN movie.title
```

5. Find the films where "Tom Hanks" acted and show the director of the film

```
MATCH (tom:Person)-[:ACTED_IN]->(movie)<-[:DIRECTED]-(dir)
WHERE tom.name = "Tom Hanks"
RETURN tom,movie,dir
```

graph result:




6. List of Tom Hanks movies

```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(movie)
RETURN tom,movie
```

graph result:


7. Who directed "Cloud Atlas"

```
MATCH (dir:Person)-[:DIRECTED]->(movie)
WHERE movie.title = "Cloud Atlas"
RETURN dir
```

8. Tom Hanks co-actor (actor in the same film)

```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(movie)
WHERE film.title = "Cloud Atlas"
RETURN dir
```

9. How people are related to "Cloud Atlas"

```
MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"})
RETURN people.name,Type(relatedTo),relatedTo
```

The *Type()* is use to get the type of nodes or relationships.

10. Movies and actors up to 4 "hops" away from Kevin Bacon

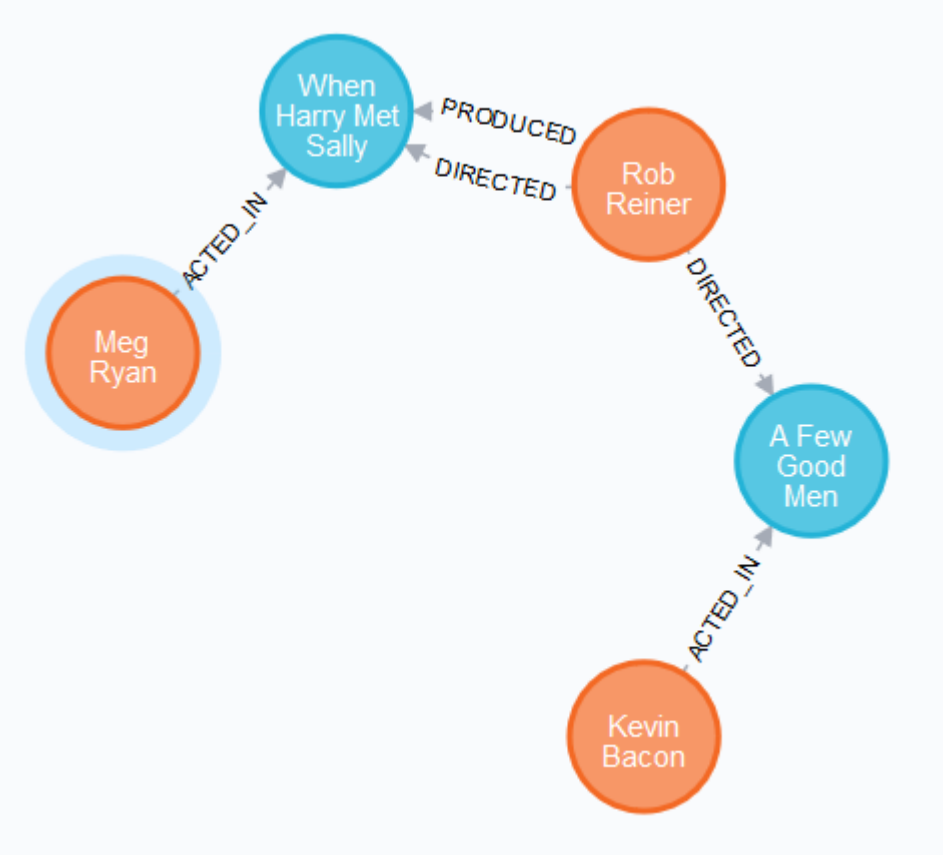
```
MATCH (bacon:Person {name: "Kevin Bacon"})-[*1..4]-(person)
RETURN DISTINCT person
```

The result of this query block count 135 nodes and 180 relationships.

11. Kevin Bacon the shortest path of any relationships to Meg Ryan

```
MATCH p=shortestPath(
    (bacon:Person {name: "Kevin Bacon"})-[*]-(meg:Person {name: "Meg Ryan"})
)
RETURN p
```

Result graph:




The **shortestPath({}-[]-{})** is a function that take a relation of 2 nodes and return the shortest path from them.

***To know more about the shortest path problem you can visit the wikipedia page: https://en.wikipedia.org/wiki/Shortest_path_problem***

12. Extend Tom Hanks co-actors, to find co-co-actors who haven't worked with Tom Hanks

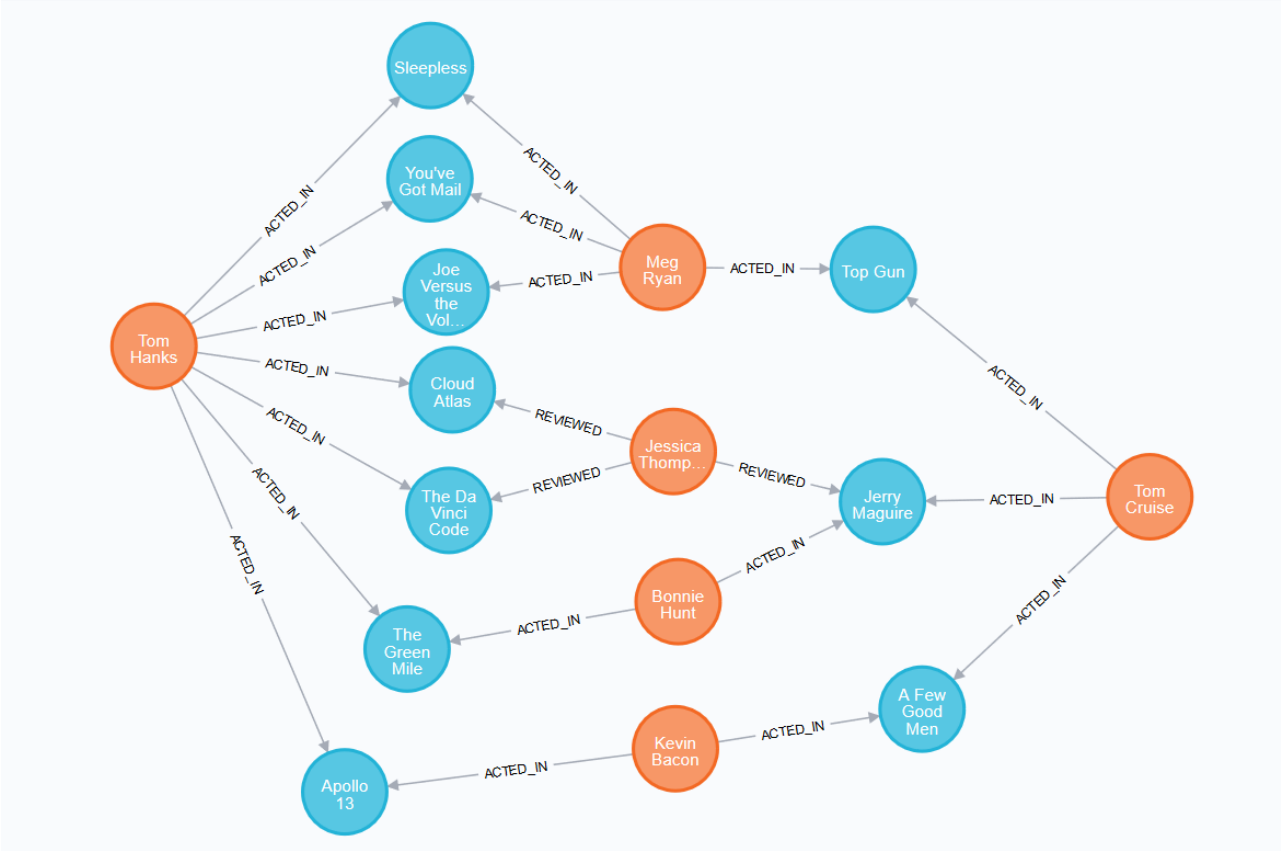
```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->()<-[:ACTED_IN]-
(coactor:Person),
    (coactor)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoactor:Person)
```

```
WHERE NOT (tom)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoactor) AND tom <>
cocoactor
RETURN cocoactor.name AS Racommended, count(*) AS Strength ORDER BY Strength
DESC
```

13. Find someone to introduce Tom Hanks to Tom Cruise

```
MATCH (tomh:Person {name: "Tom Hanks"})-[]->(movie)<-[]-(someone:Person),
    (someone)-[]->(movie2)<-[]-(tomc:Person {name: "Tom Cruise"})
RETURN tomh,movie, someone,movie2, tomc
```

Result graph:




Let's now clean the graph by delete all the nodes and relationships:

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

With this the engine take all the nodes, for every one it detach from any relationships and then delete it. To be sure and prove that the graph is gone the query is:

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

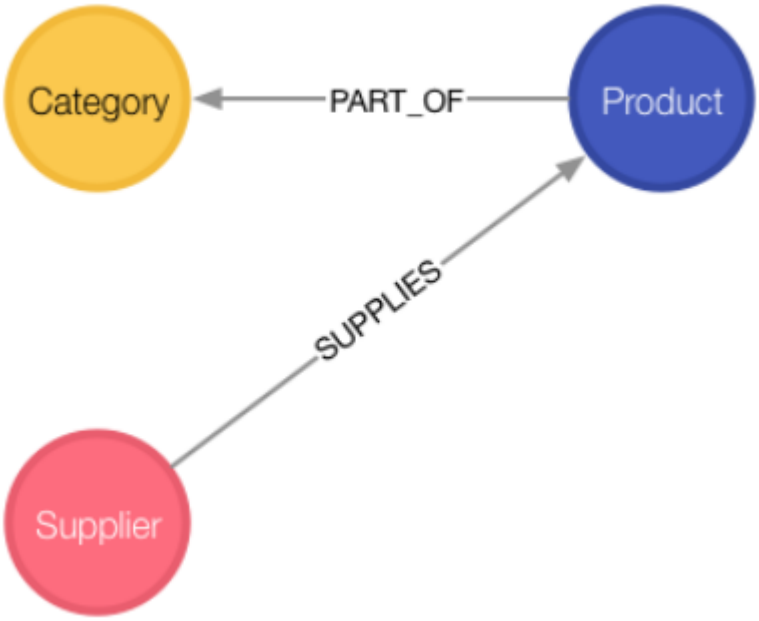
***All the query above is on the source directory of Cypher here***

---

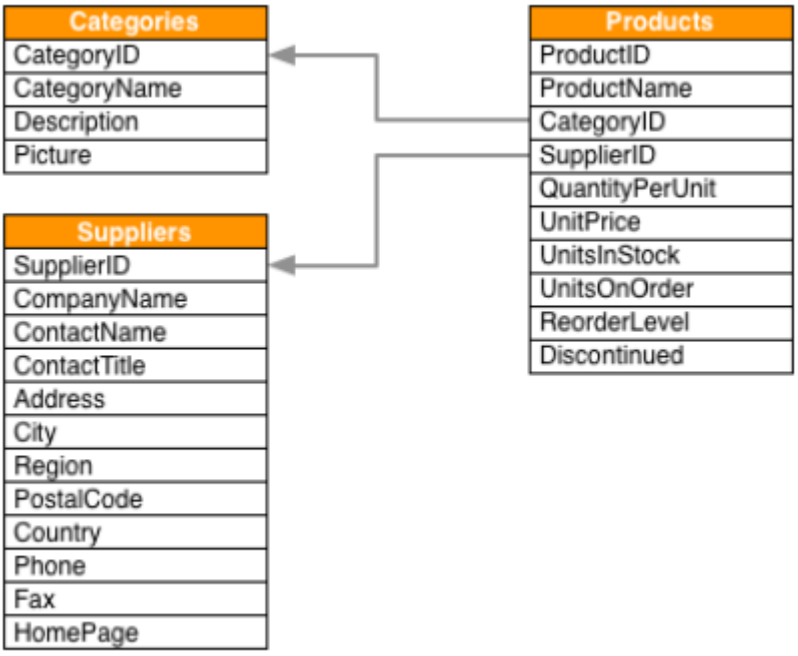
## Application - Northwind Graph

***on neof4j browser run the command*** `:play northwind graph`

This application demostrates how to migrate from a relational database to Neo4j, to do this we need to transform all the data on th relational tables to the nodes and relationships of a graph. Pratically how to get this:




From this:



This example is a sellers of food products for a few categories provided by suppliers. The database actually in use is a relational table of product catalog. The first step to transform this is with the `LOAD CSV` clause that retrive a CSV file from a valid URL and create a named map:

- Products nodes:

```
LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/northwind/products.csv" AS
row
CREATE (n:Product)
SET n = row,
    n.unitPrice = toFloat(row.unitPrice),
    n.unitinStock = toInteger(row.unitsInStock), n.unitOnOrder =
toInteger(row.unitsOnOrder),
    n.reorderLevel = toInteger(row.reorderLevel), n.discontinued =
(row.discontinued <> "0")
```

- Categories nodes:

```
LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/northwind/categories.csv"
AS row
CREATE (n:Category)
SET n = row
```

- Suppliers nodes:

```
LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/northwind/suppliers.csv"
AS row
CREATE (n:Supplier)
SET n = row
```

It's important to define an index for a node with the purpose of making searches of related data more efficent:

- Product index:

```
CREATE INDEX ON :Product(productID)
```

- Category index:

```
CREATE INDEX ON :Category(productID)
```

- Supplier index:

```
CREATE INDEX ON :Supplier(productID)
```

This 3 type of nodes are related one to another with foreign keys references we can use this properties to create the relationships from node to node. A product is a part of a category and a supplier supplies product, let's create this two relationship:

- Product-part_of->Category:

```
MATCH (p:Product), (c:Category)
WHERE p.categoryID = c.categoryID
CREATE (p)-[:PART_OF]->(c)
```

- Supplier-supplies->Product:

```
MATCH (p:Product), (s:Supplier)
WHERE p.supplierID = s.supplierID
CREATE (s)-[:SUPPLIES]->(p)
```

We can now test what we crated:

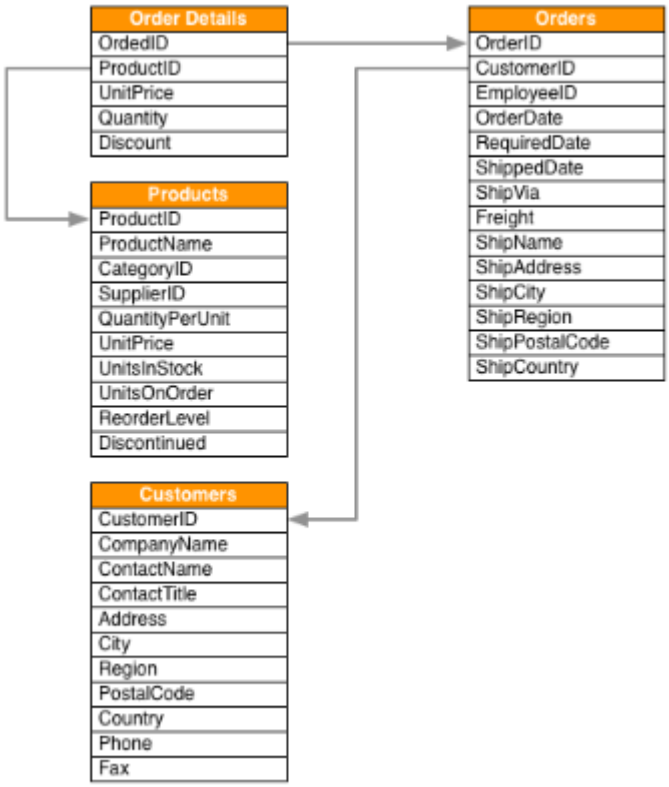
1. List the product categories proviced by each supplier:

```
MATCH (s:Supplier)-[*]->(c:Category)
RETURN
    s.companyName as Company,
    collect(distinct c.categoryName) as Categories
ORDER BY Company DESC
```

2. Find the produce suppliers:

```
MATCH (s:Supplier)-[*]->(c:Category {categoryName: "Produce"})
RETURN DISTINCT s.companyName as ProduceSuppliers
```

We can expand this graph with more datas and infos about Orders and Costumers:




We can do the same thing we had before to load and index the records:

- Customers nodes

```
LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/northwind/customers.csv"
AS row
CREATE (n:Customer)
SET n = row
```

- Orders nodes

```
LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/northwind/orders.csv" AS
row
CREATE (n:Order)
SET n = row
```

- Index customer

```
CREATE INDEX ON :Customer(customerID)
```

- Index order

```
CREATE INDEX ON :Order(orderID)
```

- Create relationship customer-purchased-order

```
MATCH
    (c:Customer),
    (o:Order)
WHERE c.customerID = o.customerID
CREATE (c)-[:PURCHASE]->(o)
```

Notice that the Order Details are always part of an Order and are the description of the relation between an order and a product, this is a sign of a data relationship indicating shared information between two other records:

```
LOAD CSV WITH HEADERS FROM "http://data.neo4j.com/northwind/order-details.csv" AS
row
MATCH (p:Product), (o:Order)
WHERE
    p.productID = row.productID AND
    row.orderID = o.orderID
CREATE (o)-[details:ORDERS]->(p)
set details = row,
    details.quantity = toInteger(row.quantity)
```

Example of query:

- List of costumers with the number of products purchased for the Produce category:

```
MATCH
    (c:Customer)-[:PURCHASED]->(:Order)-[o:ORDERS]->(p:Product),
    (p)-[:PART_OF]->(cat:Category {categoryName: "Produce"})
RETURN DISTINCT
    c.contactName AS CustomerName,
    SUM(o.quantity) AS TotalProductsPurchased
```

---

## Recommendations

***This paragraph is referred to recommedation application on neo4j browser.***

**It is possible to recreate this database using the open movie database here ->** www.omdbapi.com

**for more ->** https://movielens.org/ **and** https://grouplens.org/datasets/movielens/

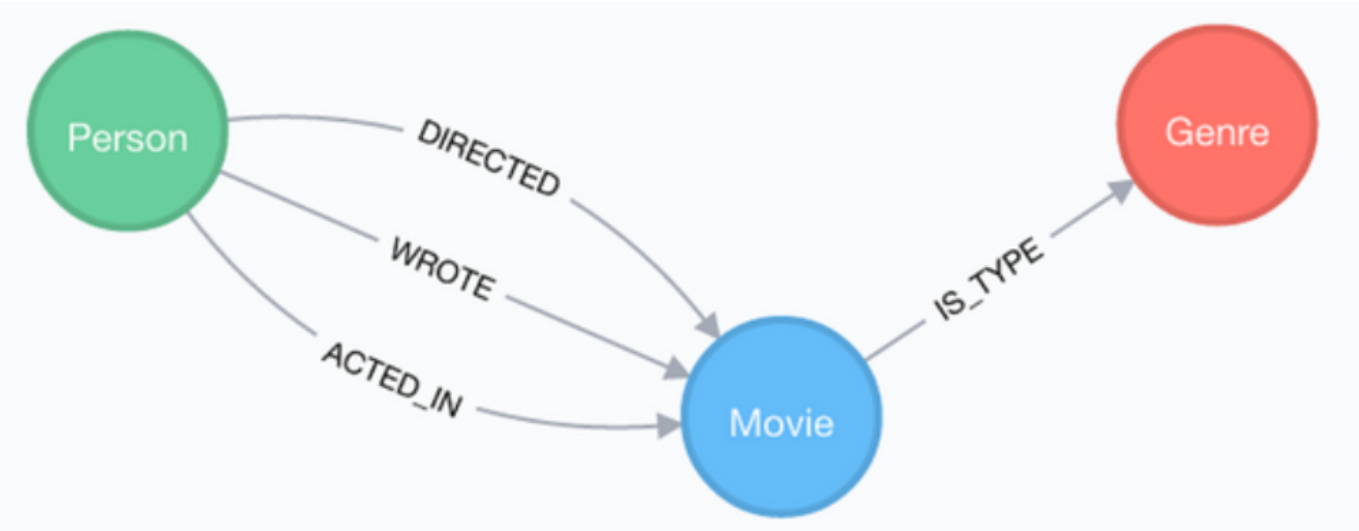
Personalized product recommendations can increase conversions, improve sales rates and provide a better experice for users. This paragraph take a look at how generate graph-based real-time personalized product

recommendations using a dataset of movies and movie ratings. All of this techniques can also apply through many different types of products or content.

Generating personalized recommendations is one of the most common use cases for a graph database and there are some benefits of using a graph to generate racommendations:

- Performance -> index-free adjacency allows for calculating recommendations in real time, ensuring the recommendation is always relevant and reflecting up-to-date information.
- Data model -> The labeled property graph model allows for easily combining datasets from multiple sources allowing enterprises to unlock value from previously separated data silos.

For this paragraph will use a graph movie with default node-relationships-node template:



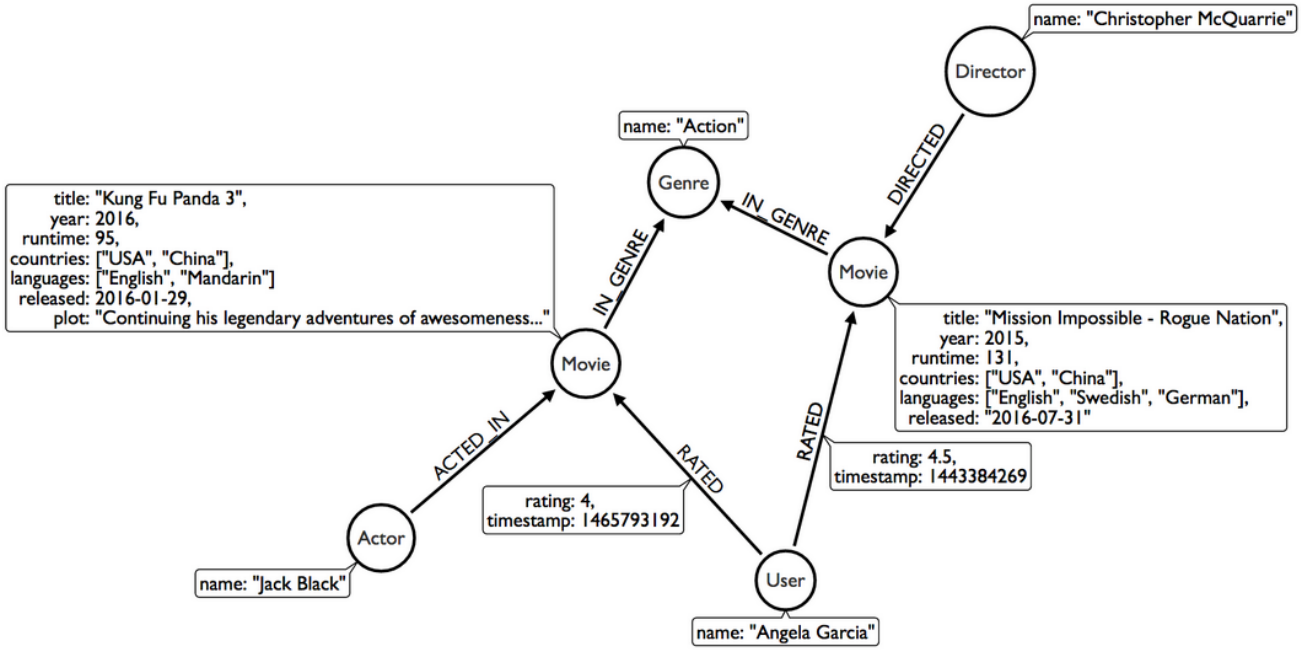

In this use case, we are using graphs to combine data from multiple silos:

- Product catalog -> data describing movies comes from the product catalog silo
- User Purchases and Reviews -> data on user purchases and reviews comes from the user or transaction silo

By combining these two containers in one graph, we are able to query across datasets to generate personalized product recommendations.

The graph result is made by:

- Labels nodes -> Movie, Actor, Director, User, Genre are the
- Relationships -> ACTED_IN, IN_GENERE, DIRECTED, RATED
- Properties -> title, name, year, rating




**From now on i will use Cypher -> Query Language**

Let's look at a Cypher query that answers the question "How many reviews does each Matrix movie have?":

```
MATCH (movie:Movie)<-[:RATED]-(user:User)
WHERE movie.title CONTAINS "Matrix"
WITH
    movie.title AS movie,
    COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5
```

Dissection of the query:

- Search for an existing graph pattern -> `MATCH (movie:Movie)<-[:RATED]-(user:User)` -> find all Movie-Rated-User on the graph
- Filter the match result -> `WHERE movie.title CONTAINS "Matrix"` -> remove useless Movie-Rated-User
- Aggregate users with movie -> `WITH movie.title AS movie, COUNT(*) AS reviews` -> count number of paths matched for each movie
- Return for each movie the number of reviews -> `RETURN movie, reviews`
- Order return by number of reviews in descending order -> `ORDER BY reviews DESC`
- Limit the number of records to find and return -> `LIMIT 5`

The result is:

| movie | reviews |
|---|---|

| movie | reviews |
|---|---|
| "Matrix, The" | 259 |
| "Matrix Reloaded, The" | 82 |
| "Matrix Revolutions, The" | 54 |

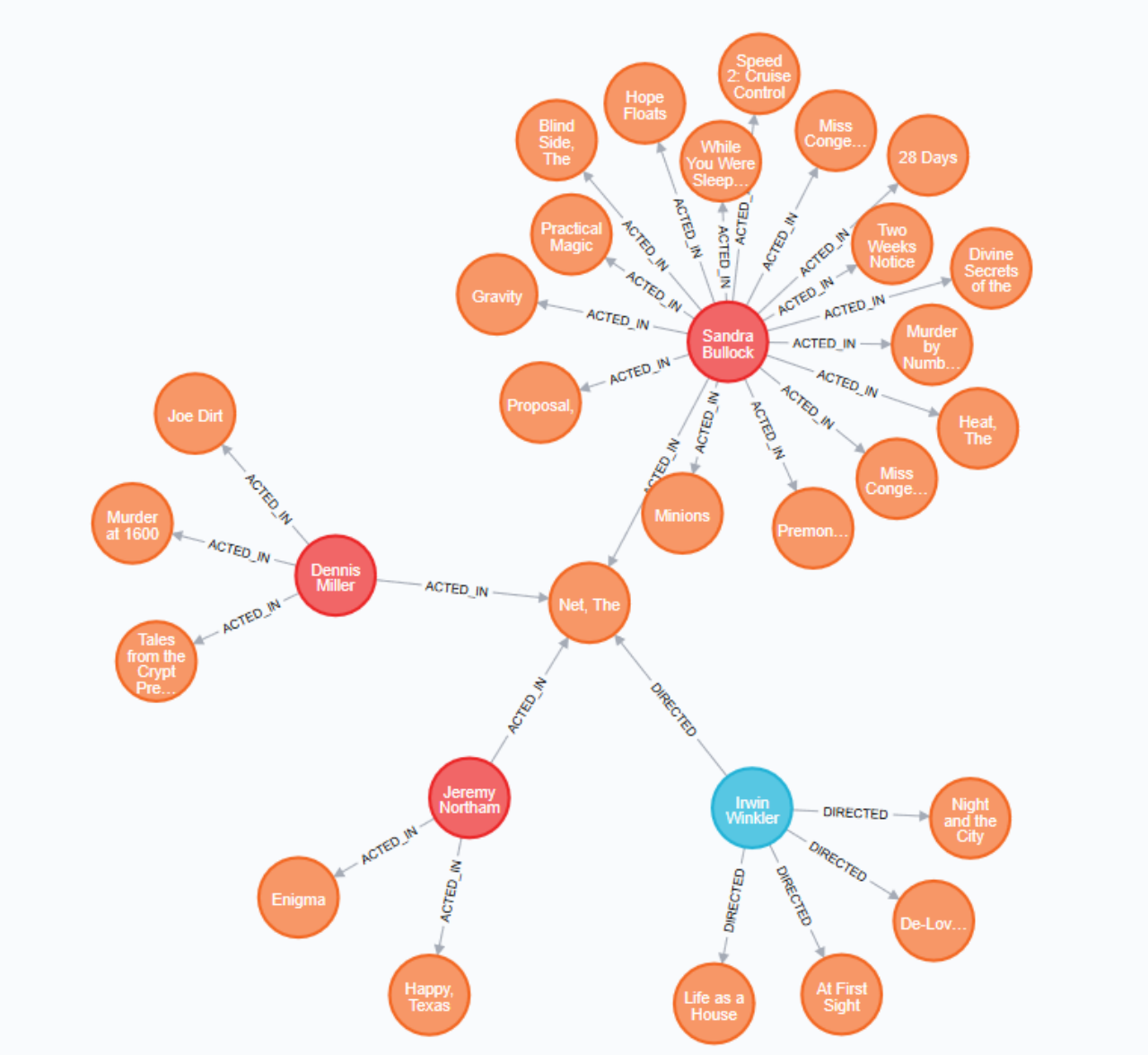
**Personalized Reccomendations**

There are two basic approaches to recommendation algorithms:

- **Content-Based filtering**: recommend intems that are similar to those that a user is viewing, rated highly or purchased previously.

  An example can be "Products similar to the product you are looking at now":

```
MATCH p=(m:Movie {title: "Net, the"})-[:ACTED_IN|:IN_GENRE|:DIRECTED*2]-()
RETURN p LIMIT 25
```



All of that result can be a movie to recommend.

The goal of content-based filtering is to find similar items, usign attributes (or traits) of the item. Using our movie data, one way we could define similarity is movies that have common genres.

Examples:

1. Find movies most similar to Inception based on shared generes:

```
MATCH (inc:Movie {title: "Inception"})-[:IN_GENRE]->(g:Genre)<-
[:IN_GENRE]-(rec:Movie)
WITH rec.title, COLLECT(g.name) AS genres, COUNT(*) AS commonGenres
RETURN rec.title AS title, genres, commonGenres
ORDER BY commonGenres DESC
LIMIT 10;
```

| title | genres | commonGenres |
|---|---|---|
| "Patlabor: The Movie (Kidô keisatsu patorebâ: The Movie)" | ["Drama", "Action", "Crime", "Thriller", "Mystery", "Sci-Fi"] | 6 |
| "Strange Days" | ["Drama", "Action", "Crime", "Thriller", "Mystery", "Sci-Fi"] | 6 |
| "Watchmen" | ["Drama", "Action", "Thriller", "Mystery", "Sci-Fi", "IMAX"] | 6 |
| "Girl Who Played with Fire, The (Flickan som lekte med elden)" | ["Drama", "Action", "Crime", "Thriller", "Mystery"] | 5 |
| "Fast Five (Fast and the Furious 5, The)" | ["Drama", "Action", "Crime", "Thriller", "IMAX"] | 5 |
| "Cellular" | ["Drama", "Action", "Crime", "Thriller", "Mystery"] | 5 |
| "Rubber" | ["Drama", "Action", "Crime", "Thriller", "Mystery"] | 5 |
| "Negotiator, The" | ["Drama", "Action", "Crime", "Thriller", "Mystery"] | 5 |
| "X-Files: Fight the Future, The" | ["Action", "Crime", "Thriller", "Mystery", "Sci-Fi"] | 5 |
| "Source Code" | ["Drama", "Action", "Thriller", "Mystery", "Sci-Fi"] | 5 |

Remember to use the LIMIT clause to prevent the output of too many rows.

2. Recommend movies similar to those the user Amgeòoca Rodriguez has already watched:

The solution can be more tricky than expected, let's analize how to do that. We need to get all the movies that Angelica Rodriguez have rated `MATCH (u:User{name:"Angelica Rodriguez"})-[r:RATED]->(movie:Movie)`. To find the most similar movies we take the genres linked and scan to find all the movies with the same genres `(movie)-[:IN_GENRE]->(genre:Genre)<-[:IN_GENRE]-(other:Movie)` and the result of this match are a graph with all the user recommendations, but some movies can be already rated by the user so we need to put a filter to the movies not yet rated `WHERE NOT EXISTS((user)-[:RATED]->(other))`. To recommend the best element of all, we necessary do some sort of rating: count the common genres between the movies watch by the user and the movie on the list `WITH other, [genre.name, COUNT(*)] AS scores`. The single score of the movie to recommend is the sum of the scores of the genres to do this we can use the REDUCE expression `REDUCE (s=0,x in COLLECT(scores) | s+x[1]) AS score`.

```
MATCH
    (user:User {name: "Angelica Rodriguez"})-[rated:RATED]->
(movie:Movie),
    (movie)-[:IN_GENRE]->(genre:Genre)<-[:IN_GENRE]-(other:Movie)
WHERE NOT EXISTS( (user)-[:RATED]->(other) )
WITH
    other,
    [genre.name, COUNT(*)] AS scores
RETURN
    other.title AS recommendation,
    other.year AS year,
    COLLECT(scores) AS scoreComponents,
    REDUCE (s=0,x in COLLECT(scores) | s+x[1]) AS score
ORDER BY score DESC
LIMIT 20
```

3. Now let's try to create a better recommendation for similar films by give a score and weight for the other relationships: such the similar actors and the same directors.

```
// Find similar movies by common genres
MATCH (m:Movie)
WHERE m.title = "Wizard of Oz, The"
MATCH (m)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(o:Movie)
WITH m, o, COUNT(*) AS gs

// Find similar movies by common actors
OPTIONAL MATCH (m)<-[:ACTED_IN]-(a:Actor)-[:ACTED_IN]->(o)
WITH m, o, gs, COUNT(a) AS as

// Find similar movies by common directors
OPTIONAL MATCH (m)<-[:DIRECTED]-(d:Director)-[:DIRECTED]->(o)
WITH m, o, gs, as, COUNT(d) AS ds

// Then return with a weighted score
RETURN o.title AS recommendation, (5*gs)+(3*as)+(4*ds) AS score
```

```
ORDER BY score DESC
LIMIT 100
```

These method used to find similar nodes are not so consistent and robust, so we need a new way to quantify usign a similarity metric. Similarity metrics are an important component used in generating personalized recommendations that allow us to quantify how similar two items are, the method we used is the Jaccard Index:

$$J(A,B) = \frac{|\ A \cap B\ |}{|\ A \cup B\ |}$$

The output of this function is a number between 0 and 1 that indicates how similar two sets are: 1 for the identical sets and 0 for sets without common element.

We can use this index for the movies to answer at the question: "What movies are most similar to Inception based on genres?

```

```

- **Collaborative Filtering**: use the preferences, ratings and actions of other users in the network to find items to recommend.

  An example can be "Users who bought this thing, also bought that other thing":

```
MATCH (m:Movie {title: "Crimson Tide"})<-[:RATED]-(u:User)-[:RATED]->
(rec:Movie)
RETURN
    rec.title AS recommendation,
    COUNT(*) as usersWhoAlsoWatched
ORDER BY usersWhoAlsoWatched DESC
LIMIT 25
```

| recommendation | usersWhoAlsoWatched |
|---|---|
| "Forrest Gump" | 70 |
| "Dances with Wolves" | 68 |
| "Pulp Fiction" | 68 |
| "Fugitive, The" | 65 |
| "True Lies" | 64 |
| "Jurassic Park" | 63 |
| "Silence of the Lambs, The" | 62 |
| "Apollo 13" | 61 |

| recommendation | usersWhoAlsoWatched |
|---|---|
| "Batman" | 61 |
| "Aladdin" | 58 |
| ... | ... |

All of that result can be a movie to recommend.

---

# Index



## ACID consistency model

The acronym stands for:

- Atomic -> All operations in a transaction succeed or every operation is rolled back
- Consistent -> On the completition of a transaction, the database is structurally sound
- Isolated -> Transactions do not contend with one another. Contentious access to data is moderated by database so that transactions appear to run sequentially.
- Durable -> The results of applying a transaction are permanent, even in the presence of failures

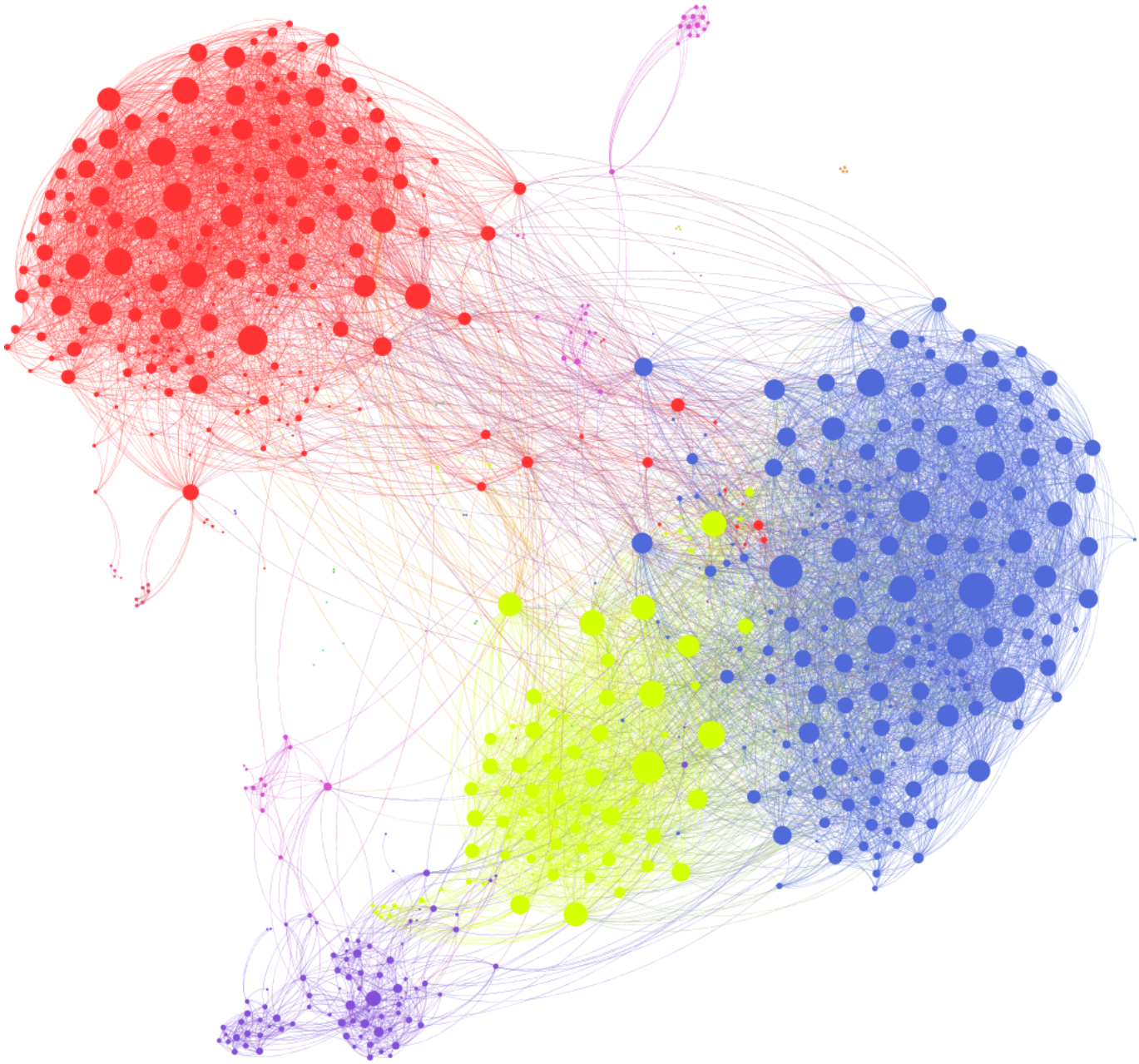
This four properties mean that once a transaction is complete, its data is consistent and stable on disk.

Most of graph databases (Neo4j incuded) use an ACID consistency model to ensure data is safe and consistently stored.

## Cluster

A cluster is when data is assembled around one particular value, on graph usually happens when there are several nodes thet seem to gather in a certain area. In other word a cluster is a group that are placed closely next to each other in a certain area, with a few nodes scattered in other places on the graph.

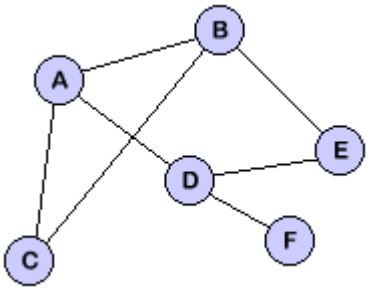
For example this is a social netowork graph:



There are several aggregations of similar nodes that are colored.

## index-free adjacency

A graph database is any storage system that provides index-free adjacency and this means that every element contains a direct pointer to its adjacent element and no index lookups are necessary. This is one of the most element of distinction from relational databases that are based on a key-value store. The idea of an index-free adjacency is analogous to that of a pointer where the time to follow the relationship is O(1) with respect to the size of the graph, and this is not the cost of found a relationship in a relational. To be more precise a relational database use indexes but the aren't used for adjacency or link purposes. With Neo4j tipically a query will find the node to start from with an index and then the resto fo the query simply follows relationships to compute the answer. When create a new set of nodes it's typical to create an index of that type, see northwind graph, this reduce the cost of every query run. Example:




In a relational database if we want to switch from data **A** to data **B** the step are:

1. The db prohibited you to access direct items via pointers, so you need to ask for the permission.
2. The request permission line can be full of other request, wait for your turn.
3. The central index of db search for the data you want in a list of other data , the cost of search is O(n) with **n** number of the data stored.
4. Once you get the memory pointer of that data you can go to **B** by follow the memory allocations by hopping on memory.
5. The cost of this hopping is O(n), but at the end you can switch to **B** and do your operations.

In a graph database like Neo4j all the nodes are connected to each other by relationships, so in order to go from **A** to **B**:

1. Get the relationships pointer
2. Point to node **B** wherever the type of the relationships are
3. Do operations on **B**

The cost in this case is O(1), a big difference in performance between a classical relational db.

## OLTP

This is an acronym that stands for Online transaction processing. The term transaction in the realm of database transactions it denotes an atomic change of state. The goals of OLTP applications are availability, speed, cuncurrency and recoverability.

An OLTP system is a common data processing system like order entry, retail sales and financial transaction systems.

The main characteristics of an OLTP environment are:

- Short response time
- Small transactions
- Data maintenance operations
- Large user populations
- High concurrency
- Large data volumes
- High availability
- Lifecycle-related data usage

The benefits of this environments are:

- Support for bigger databases

- Partition maintenance operations for data maintenance
- Potential higher concurrency through elimination of hot spots

---

## References

Graph database Fundamentals:

- neo4J browser sandbox by type the command `:play concepts`
- graphacademy

Others:

- neo4j online course
- neo4j docs
- neo4j repository
- neo4j browser sandbox
- neo4j developer get started
    - dev-cypher
    - dev-graph-database
    - dev-graph-data-modeling
    - dev-graph-platform
    - dev-graph-visualization-tools
    - dev-language-guide
    - dev-importing-csv-data
    - dev-integration
    - dev-in-production
- neo4j developer manual
- Graphacademy
    - introduction
    - YouTube playlist: