Last Words

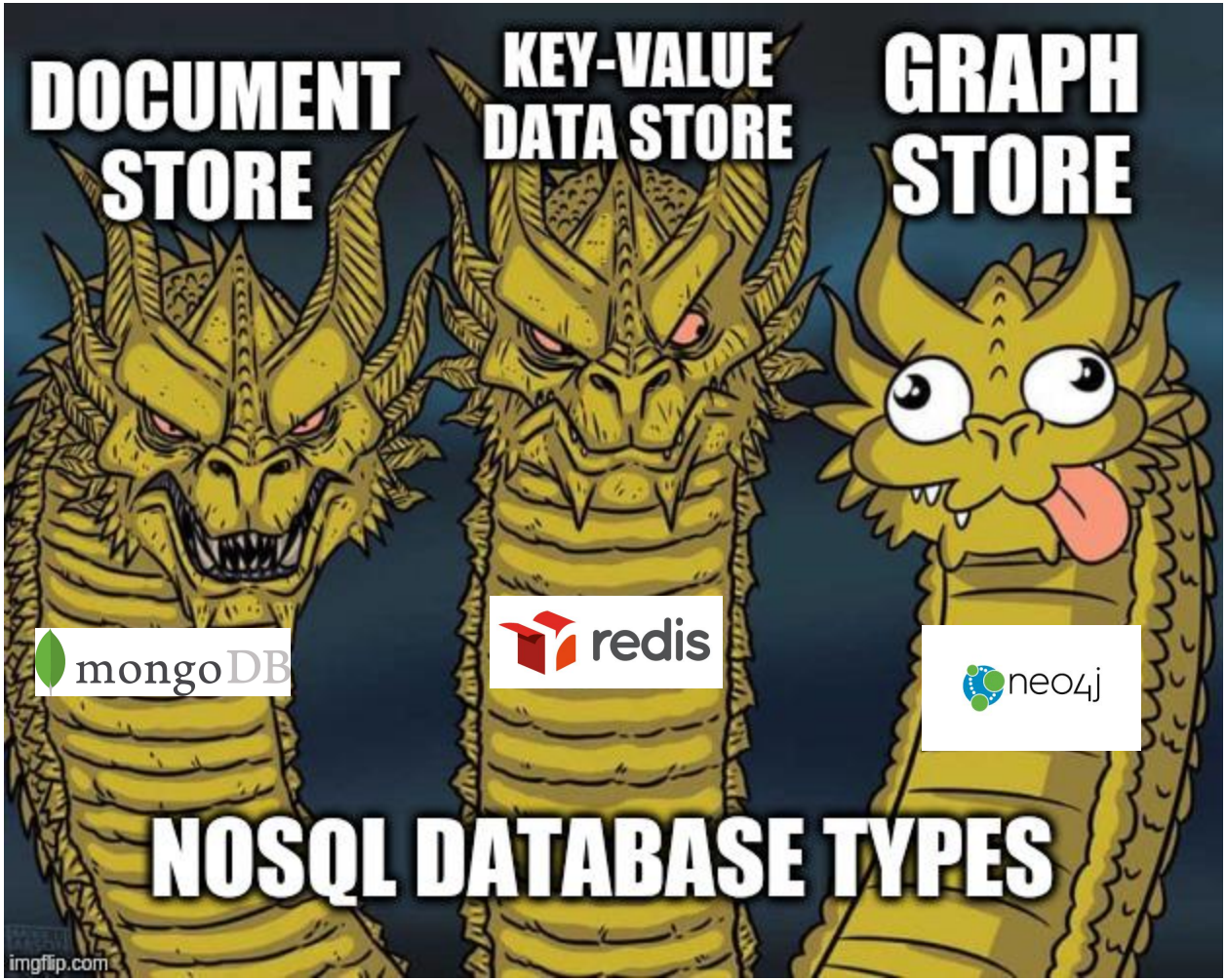
- Try to solve your issue Alone. YES, Alone!!?
- Missing/Steps configuration.
- Happy to everyone when they do their tasks.
- Find Work/life balance.
- Prepare yourself to companies Jobs/Tasks.

Life is Harder

Databases - Tutorial 12
Graph Database -Neo4j

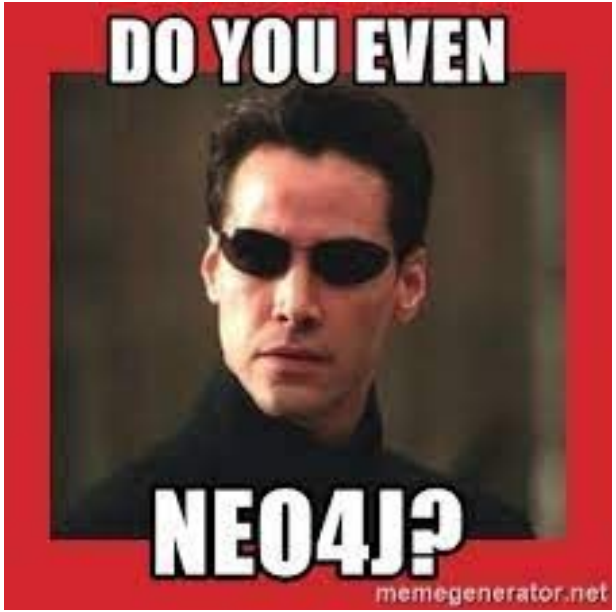
Hamza Salem - Innopolis University

4 **Carlo strozzi**


DOCUMENT STORE
KEY-VALUE DATA STORE
GRAPH STORE
mongoDB
redis
neo4j
NOSQL DATABASE TYPES
imgflip.com

Content

- Neo4J Introduction



Graph Fundamentals

Basic concepts to get you going.

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

1. Nodes - graph data records
2. Relationships - connect nodes
3. Properties - named data values

A *Graph* Database

Neo4j stores data in a Graph, with records called Nodes.

The simplest graph has just a single node with some named values called Properties. Let's draw a social graph of our friends on the Neo4j team:

1. Start by drawing a circle for the node
2. Add the name Emil
3. Note that he is from Sweden

- Nodes are the name for data records in a graph
- Data is stored as Properties
- Properties are simple name/value pairs

Labels

Associate a set of nodes.

Nodes can be grouped together by applying a Label to each member. In our social graph, we'll label each node that represents a Person.

1. Apply the label "Person" to the node we created for Emil
2. Color "Person" nodes red

- A node can have zero or more labels
- Labels do not have any properties

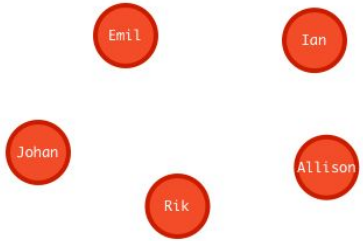
More Nodes

Schema-free, nodes can have a mix of common and unique properties.

Like any database, storing data in Neo4j can be as simple as adding more records. We'll add a few more nodes:

1. Emil has a klout score of 99
2. Johan, from Sweden, who is learning to surf
3. Ian, from England, who is an author
4. Rik, from Belgium, has a cat named Orval
5. Allison, from California, who surfs

- Similar nodes can have different properties
- Properties can be strings, numbers, or booleans
- Neo4j can store billions of nodes



Consider Relationships

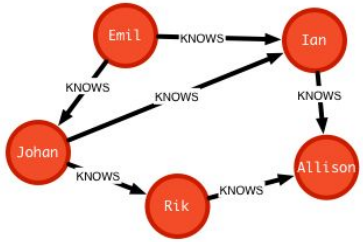
Connect nodes in the graph

The real power of Neo4j is in connected data. To associate any two nodes, add a Relationship which describes how the records are related.

In our social graph, we simply say who KNOWS whom:

1. Emil KNOWS Johan and Ian
2. Johan KNOWS Ian and Rik
3. Rik and Ian KNOWS Allison

- Relationships always have direction
- Relationships always have a type
- Relationships form patterns of data

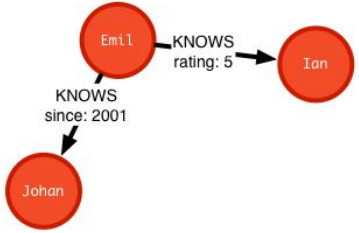


Relationship properties

Store information shared by two nodes.

In a property graph, relationships are data records that can also contain properties. Looking more closely at Emil's relationships, note that:

- Emil has known Johan since 2001
- Emil rates Ian 5 (out of 5)
- Everyone else can have similar relationship properties



Cypher

Neo4j's graph query language

Neo4j's Cypher language is purpose built for working with graph data.

- uses patterns to describe graph data
- familiar SQL-like clauses
- declarative, describing what to find, not how to find it

CREATE

Create a node

Let's use Cypher to generate a small social graph.

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

- `CREATE` clause to create data
- `()` parenthesis to indicate a node
- `ee:Person` a variable 'ee' and label 'Person' for the new node
- `{}` brackets to add properties to the node

MATCH

Finding nodes

Now find the node representing Emil:

```
MATCH (ee:Person) WHERE ee.name = "Emil" RETURN ee;
```

- `MATCH` clause to specify a pattern of nodes and relationships
- `(ee:Person)` a single node pattern with label 'Person' which will assign matches to the variable 'ee'
- `WHERE` clause to constrain the results
- `ee.name = "Emil"` compares name property to the value "Emil"
- `RETURN` clause used to request particular results

CREATE more

Nodes and relationships

`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)
```

Pattern matching

Describe what to find in the graph

For instance, a pattern can be used to find Emil's friends:

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

- `MATCH` clause to describe the pattern from known Nodes to found Nodes
- `(ee)` starts the pattern with a Person (qualified by WHERE)
- `-[:KNOWS]-` matches "KNOWS" relationships (in either direction)
- `(friends)` will be bound to Emil's friends

Recommend

Using patterns

Pattern matching can be used to make recommendations. Johan is learning to surf, so he may want to find a new friend who already does:

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

- `()` empty parenthesis to ignore these nodes
- `DISTINCT` because more than one path will match the pattern
- `surfer` will contain Allison, a friend of a friend who surfs

Analyze

Using the visual query plan

Understand how your query works by prepending `EXPLAIN` or `PROFILE`:

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

Closing words

PROS:

- Easy to use, scale, modify etc ... but only on small scale
- Easy access to data

CONS:

- Application logic degrades quickly with increase in scale
- Neo4j gets slower with the amount of data
- Lack of caching support

Demo

https://18ebe1e0db811eeb04003a5bc1675f73.neo4jsandbox.com/browser/?token=pwfetch:18ebe1e0db811eeb04003a5bc1675f73:eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IlFUbENPRVV4UmtJNFJETkROakpETXpBME5EZzBRelV3UWpNek9UVTVNRFF4TlRKRk56STJOZyJ9.eyJlbWFpbCI6ImVuZ2hhbXphc2FsZW03N0BnbWFpbC5jb20iLCJmYW1pbHlfbmFtZSI6IlNhbGVtIiwiZ2l2ZW5fbmFtZSI6IkhhbXphIiwibG9jYWxlIjoiZW4iLCJuYW1lIjoiSGFtemEgU2FsZW0iLCJuaWNrbmFtZSI6ImVuZ2hhbXphc2FsZW03NyIsInBpY3R1cmUiOiJodHRwczovL2xoMy5nb29nbGV1c2VyY29udGVudC5jb20vYS0vQU9oMTRHamoxTkg3cXgzLUtGRzE3QldUd2hxNm5ZNUlyNFZ2WEI5ZUJvRHRhWkk9czk2LWMiLCJ1c2VyX21ldGFkYXRhIjp7ImNvbXBhbnkiOiJ5b3VyIGNvbXBhbnlZmUifSwiYXBwX21ldGFkYXRhIjp7InNhbmRib3h2MyI6eyJjcmVhdGVkQXQiOjE1ODE0MjIwMzcwNTQsImFncmVlZFRvVGVybXNBdCI6MTU4MTQyMjA3OTc1Niwic2FuZGJveEJyb3dzZXJUb3VyIjoxNjE3ODg4Njc3NTA4LCJzYW5kYm94QnJvd3NlckZlZWRiYWNrIjp7InJhdGluZyI6MSwidGV4dEZlZWRiYWNrIjoidCJ9fX0sInNhbmRib3h2MyI6eyJjcmVhdGVkQXQiOjE1ODE0MjIwMzcwNTQsImFncmVlZFRvVGVybXNBdCI6MTU4MTQyMjA3OTc1Niwic2FuZGJveEJyb3dzZXJUb3VyIjoxNjE3ODg4Njc3NTA4LCJzYW5kYm94QnJvd3NlckZlZWRiYWNrIjp7InJhdGluZyI6MSwidGV4dEZlZWRiYWNrIjoidCJ9fSwiZmlyZWJhc2VfZGF0YSI6eyJ1aWQiOiJnb29nbGUtb2F1dGgyfDExNTgzMjAzMDY2NTg4OTAyMTQ2NiJ9LCJzY29wZXMiOnsic2FuZGJveGVzIjpbInNib3gxIiwic2JveDIiLCJzYm94MyJdfSwiY2xpZW50SUQiOiJEeGhtaUY4VENlem5JN1hvaTA4VXlZU2NMR1puazRrZSIsImNyZWF0ZWRfYXQiOiIyMDIwLTAyLTExVDA3OjQ5OjU1LjU5MVoiLCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiaWRlbnRpdGllcyI6W3sicHJvdmlkZXIiOiJnb29nbGUtb2F1dGgyIiwidXNlcl9pZCI6IjExNTgzMjAzMDY2NTg4OTAyMTQ2NiIsImNvbm5lY3Rpb24iOiJnb29nbGUtb2F1dGgyIiwiaXNTb2NpYWwiOnRydWV9XSwidXBkYXRlZF9hdCI6IjIwMjItMDQtMjdUMTM6Mjg6MjcuNDE3WiIsInVzZXJfaWQiOiJnb29nbGUtb2F1dGgyfDExNTgzMjAzMDY2NTg4OTAyMTQ2NiIsImlzcyI6Imh0dHBzOi8vbG9naW4ubmVvNGouY29tLyIsInN1YiI6Imdvb2dsZS1vYXV0aDJ8MTE1ODMyMDMwNjY1ODg5MDIxNDY2IiwiYXVkIjoiRHhobWlGOFRDZXpuSTdYb2kwOFV5WVNjTEdabms0a2UiLCJpYXQiOjE2NTEwNjYxMTgsImV4cCI6MTY1MTE1MjUxOCwibm9uY2UiOiJOM1ZtYW5CUFMzUkRTMXBHTWw4MlgxVndRbmg0VGxsaVRERnZRbU5wV1dnMGFHZDJPQzVYYTVRRM1pBPT0ifQ.LjVCLcudJFHWAIPvljLwag7s39zk8NTtmZR0VcF3rYXWDQ_yqMMaQWsB9nutUfWqfi-ZbrCNqoPjiOopD0u0gYY23ZrPVIcPlUC3nMCA59D1pVkRAQJsna8Ax0qlX21rdm2VFHGQJDP7EolhidisD-P-u7D4vNQODp_PtHv_8mutrCJa9Dd0_nrBk2-prBJyKzPXpt-LqdBLEOtr9iPklth3P0jN2E1DfK8QUkrbOy0BVEq51Zv82CbcZDaeq8rbBgT4D1BTbG8WYA1TLAOTtZGGaWxSqk3RK-4kJMKrfY4VjmnE82czumuu2Q_CQEqseADdVDQ4xmLCKoCL83O8QA&_ga=2.118155145.1449166741.1651065886-1058097308.1651065883

https://neo4j.com/docs/cypher-manual/current/clauses/create/

NEO4J

- Link to download - https://neo4j.com/download/
- Tutorial - https://neo4j.com/developer/get-started/
- Why we should use the neo4j - https://www.youtube.com/watch?v=_D19h5s73Co
- NEO4J sandbox - https://sandbox.neo4j.com
- https://habr.com/ru/post/219441/
- https://neo4j.com/docs/cypher-manual/current/clauses/create/