Advanced Databases

Graph databases

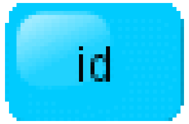
Giulia Vilone

giulia.vilone@tudublin.ie

What is a graph?

What is a graph?

An abstract representation of a set of objects where some pairs are connected by links.

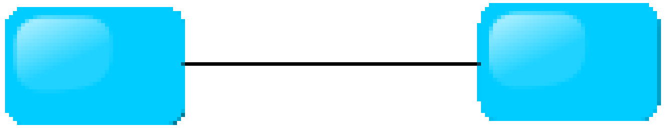


Object (Vertex, Node)

Link (Edge, Arc, Relationship)

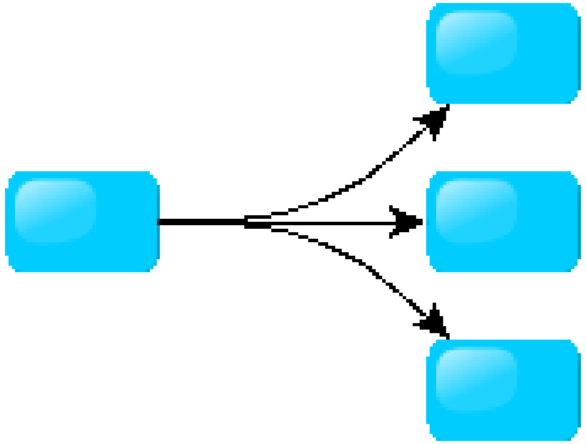
Different types of graphs

Undirected Graph

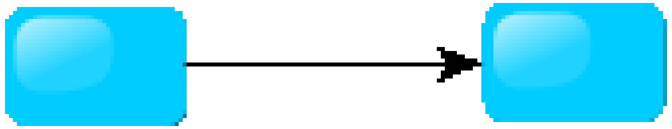


Pseudo Graph

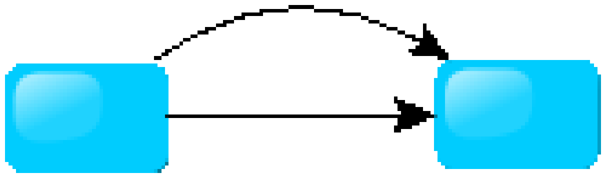
Hyper Graph



Directed Graph

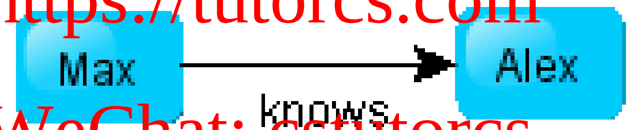


Multi Graph



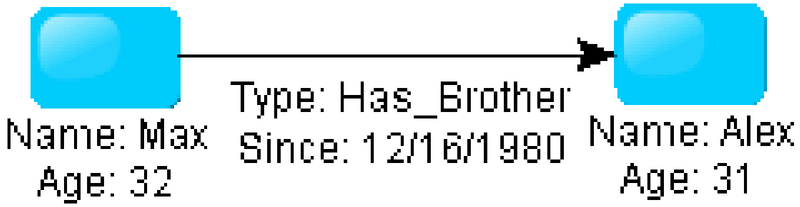
More types of graphs

Labelled Graph



Weighted Graph

Property Graph



What is a graph database?

- A database with an explicit graph structure with nodes, edges and properties to store data

- Each node knows its adjacent nodes

- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Plus, an index for look-ups
- Provides index-free adjacency

Graph databases

Data model

- Nodes with properties.
- Named relationships with properties.

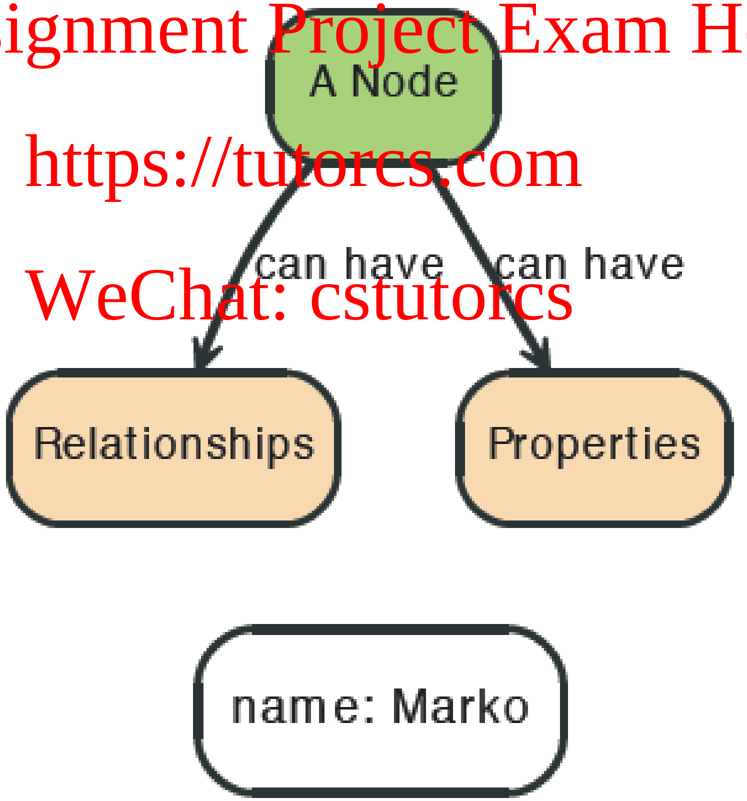
Examples

- Neo4j, Sones GraphDB, OrientDB, InfiniteGraph, AllegroGraph

Graph databases

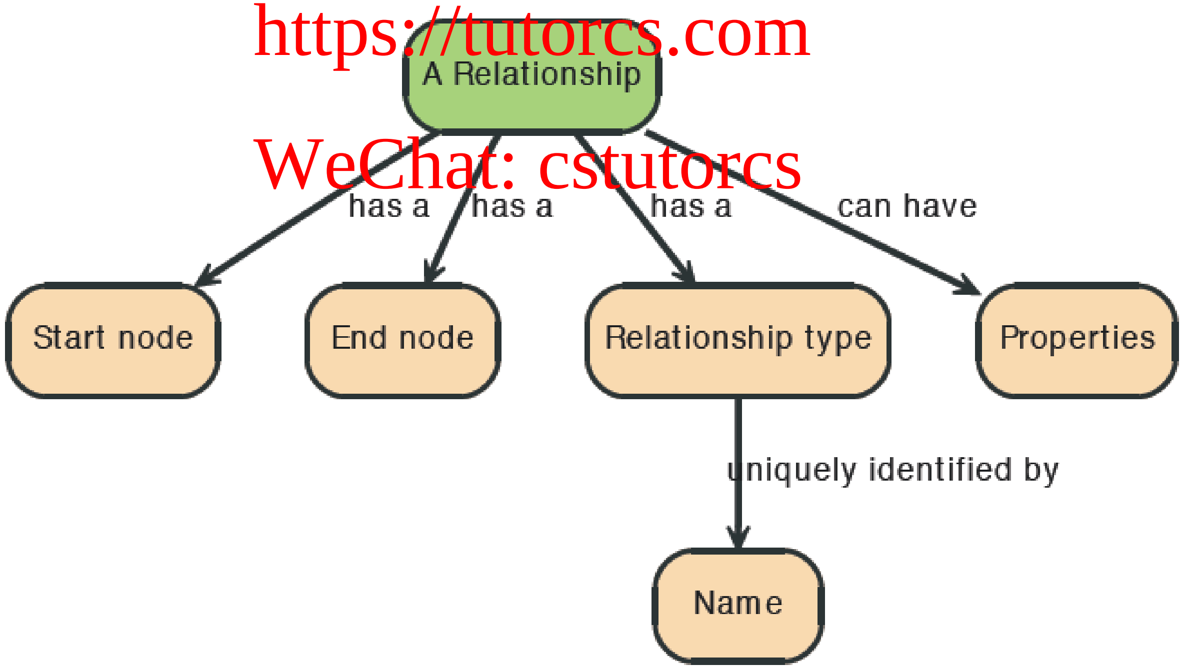
- Nodes represent entities
- Edges represent relationships, hold most of the important information and connect
  - nodes to other nodes
  - nodes to properties
- Connections between data are explored
- Faster for associative data sets
- Intuitive
- Optimal for searching social network data

Node in graph databases



Relationships in graph databases

Relationships between nodes are a key part of Neo4j.



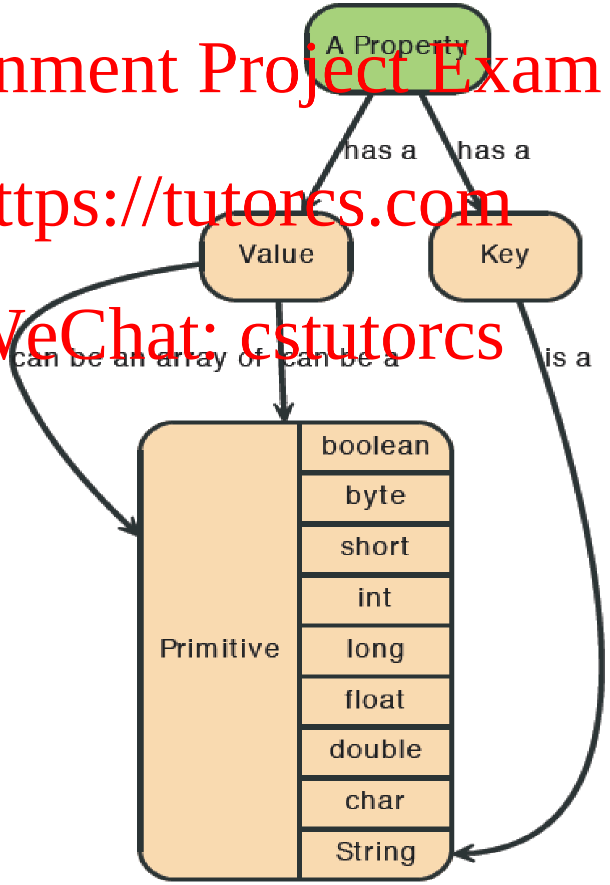
Properties

- Both nodes and relationships can have properties.

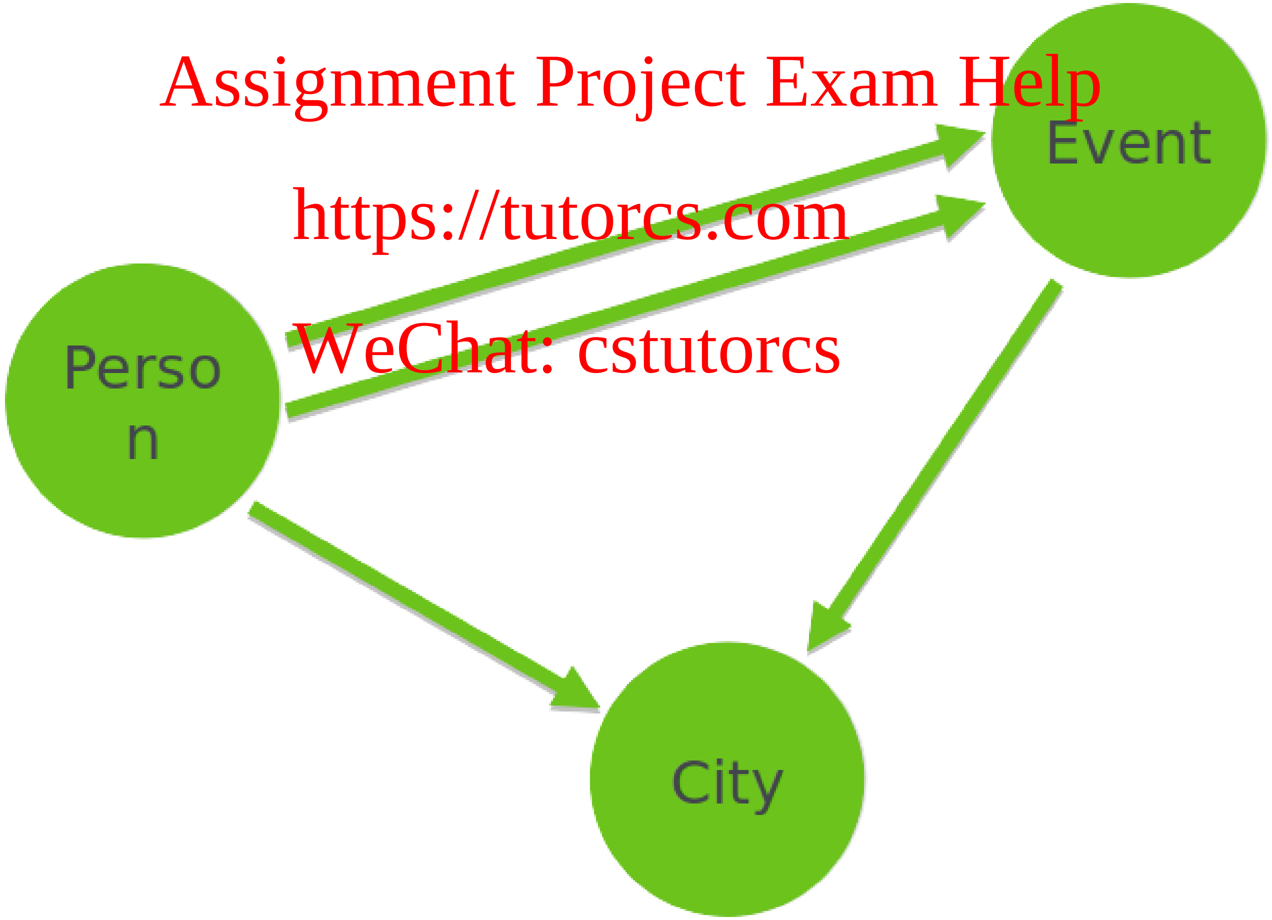
- Properties are key-value pairs where the key is a string.

- Property values can be either a primitive or an array of one primitive type. For example, String, Int and Int[] values are valid for properties.

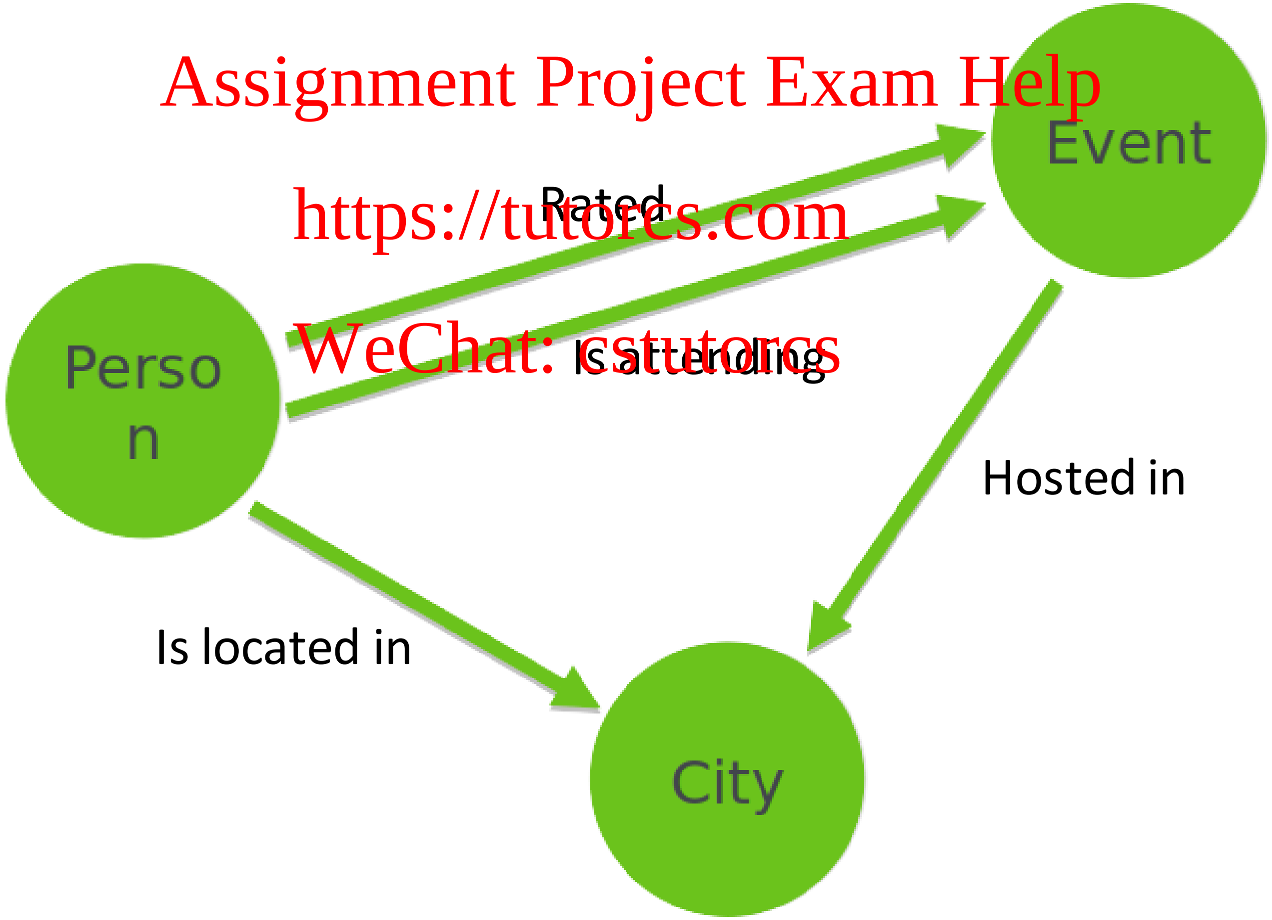
Properties


A Property
has a
has a
Value
Key
can be an array of
can be a
is a
Primitive
boolean
byte
short
int
long
float
double
char
String

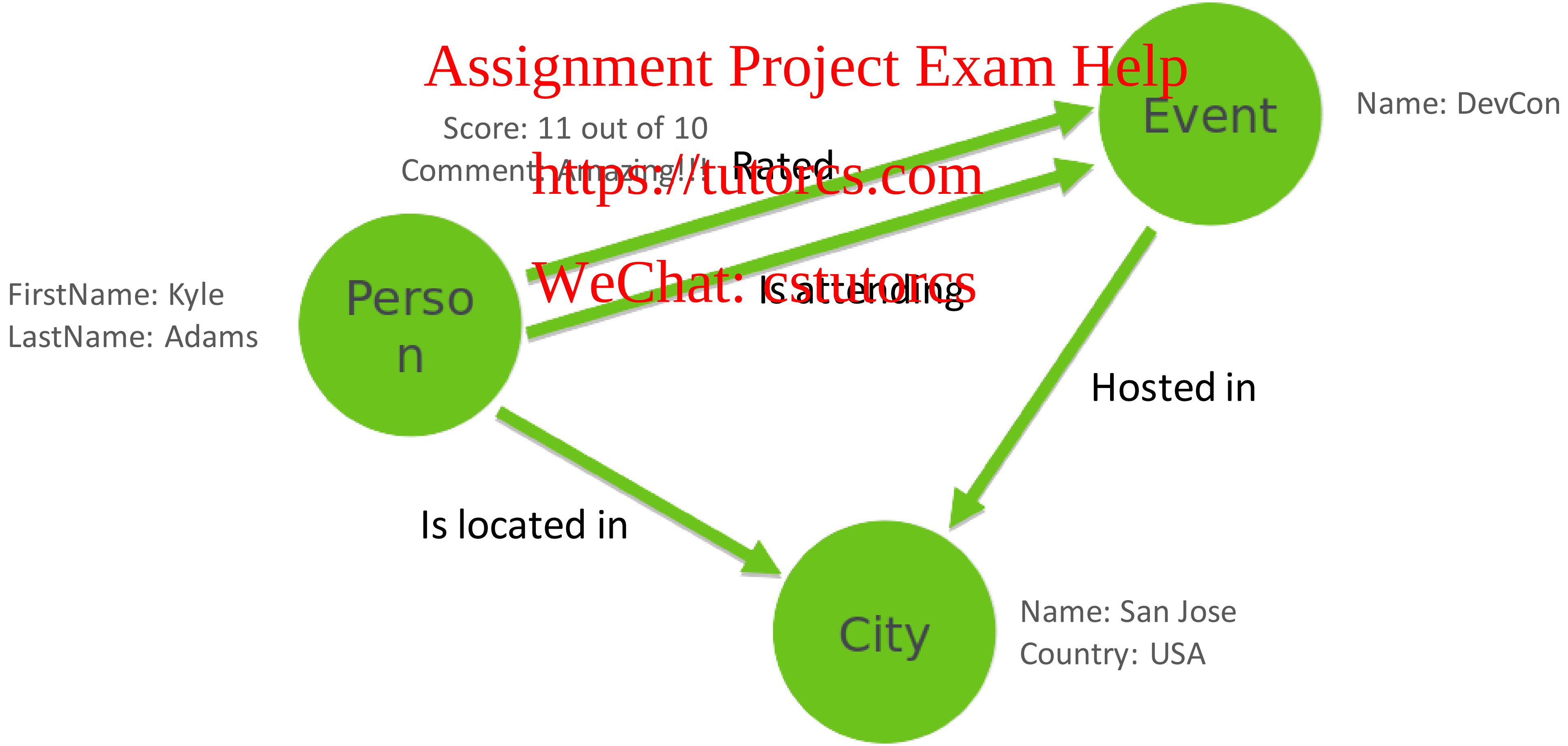
Graph data model



Graph data model



Graph data model



Why using graph database

SQL

- It is hard to represent highly connected data in a graph-like structure.
- They are good for “1 step” relations.

NOSQL

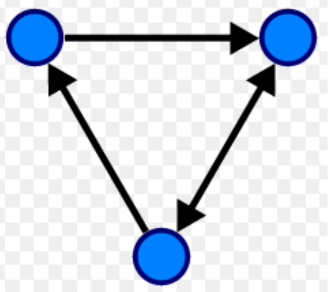
- Most NOSQL databases store sets of disconnected documents/values/columns.
- This makes it difficult to use them for connected data and graphs. One well-known strategy for adding relationships to such stores is to embed an aggregate’s identifier inside the field belonging to another aggregate—effectively introducing foreign keys. But this requires joining aggregates at the <u>application level</u>, which quickly becomes prohibitively expensive.

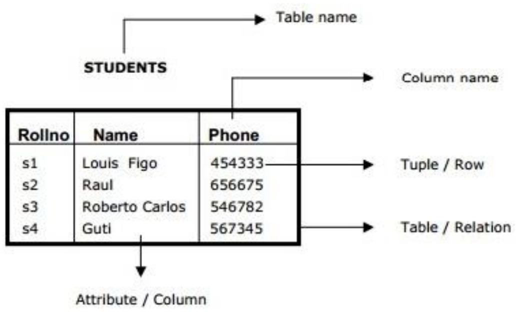
Advantages of graph databases

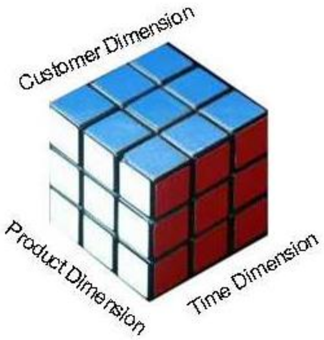
- Extremely powerful data model. When there are relationships that you want to analyse, graph databases become a very nice fit because of the data structure
- Graph databases are very fast for associative data sets
  - Like social networks
- Map more directly to object-oriented applications
  - Object classification and Parent->Child relationships
- Performant when querying interconnected data
- Easily to query

Disadvantages of graph databases

- If data is just tabular with not much relationship between the data, graph databases do not fare well
- OLAP (*online analytical processing*) support for graph databases is not well developed
  - Lots of research happening in this area







Ease of aggregation

Disadvantages of graph databases

- Is it easy to break a graph in pieces? No, partitioning a graph is very hard!

- A distributed graph-database is challenging
- Sharding
- Not everything is a graph

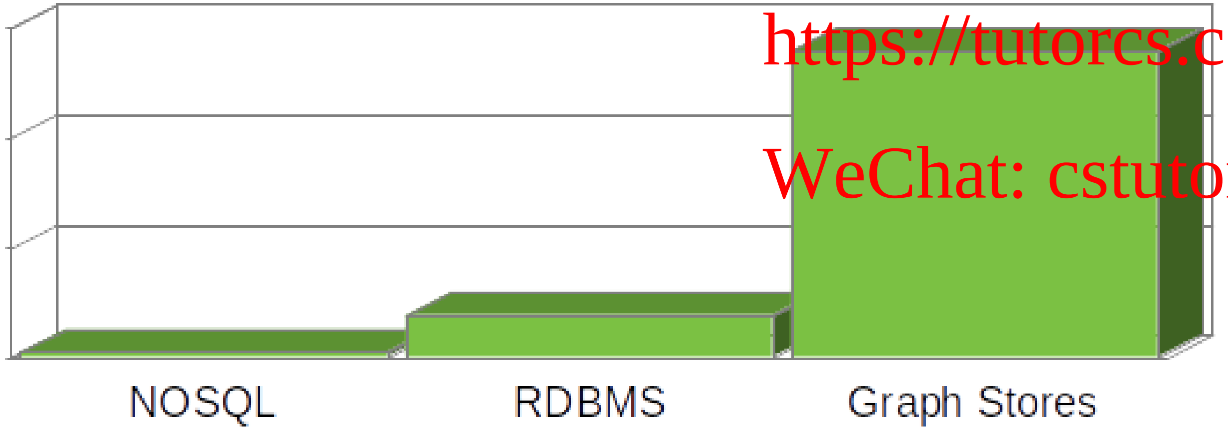
Typical use cases for graph databases

- Recommendations
- Business Intelligence
- Social Computing
- Geospatial

- Genealogy
- Time Series Data
- Web Analytics
- Fraud Detection

Have a look here: https://neo4j.com/use-cases/

Maturity of data models

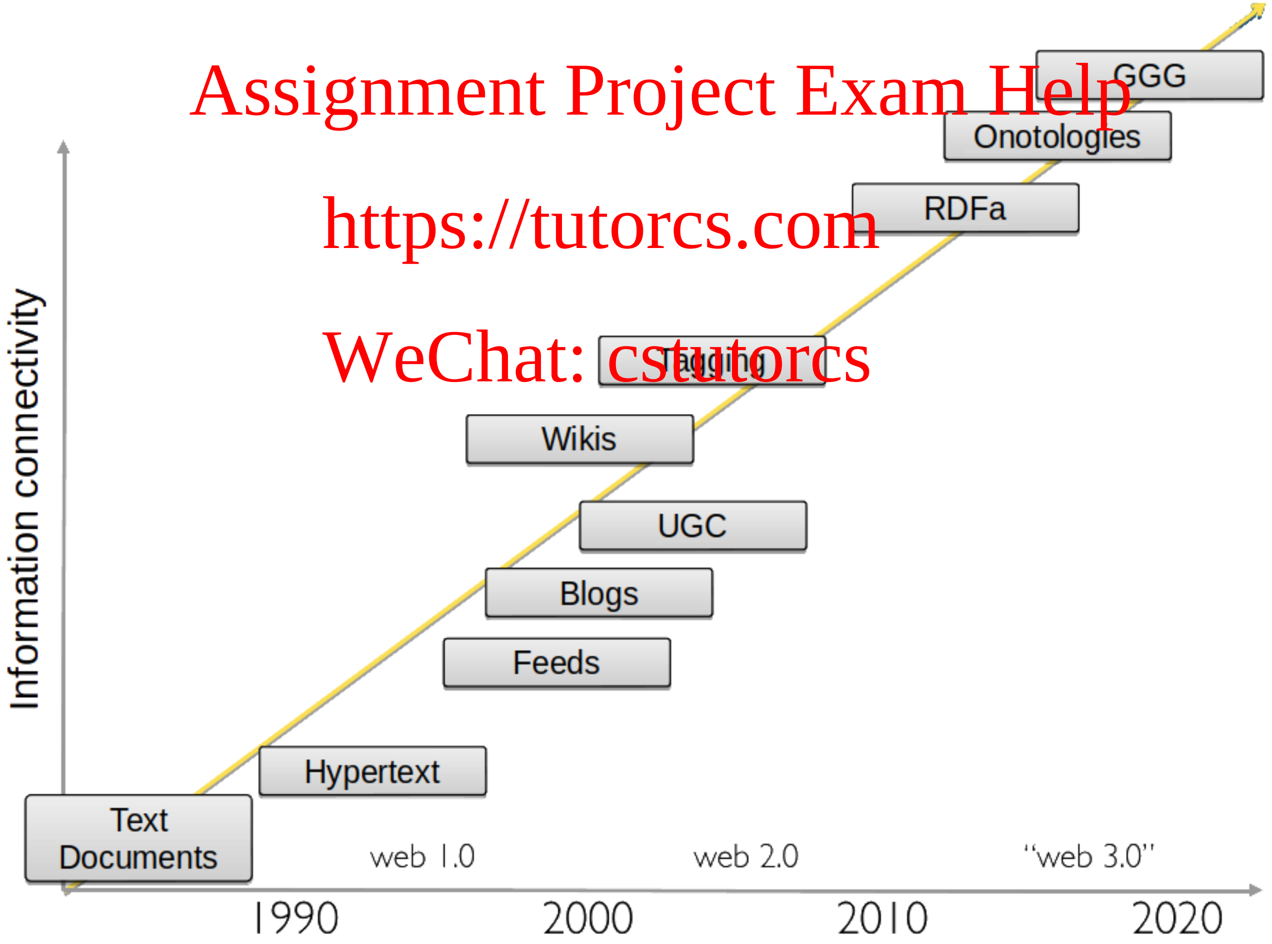


- Most NOSQL: ~8 years
- Relational: 42 years
- Graph Theory: 276 years

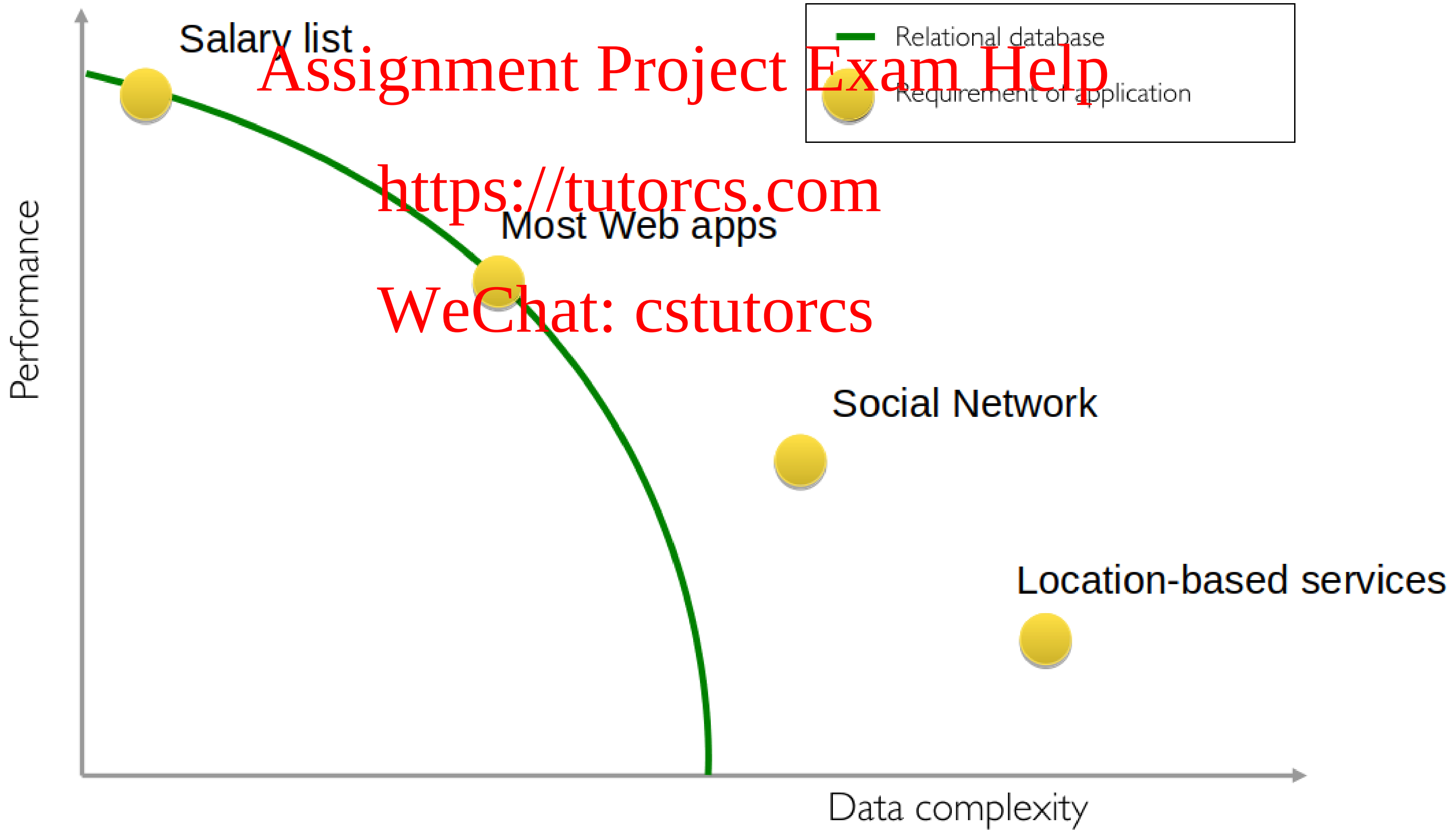
Data is more connected

- Text
- HyperText
- RDF Site Summary or Really Simple Syndication (RSS)
- Blogs
- Tagging
- Resource Description Framework (RDF)

Trend 2 - connectedness



Side node – RDBMS performance



Application domains

Neo4j

Introducing Neo4j

- Introduced in 2010
- Developed by Neo Technologies
- Most Popular Graph Database
- Open source
- Java-based
- NoSQL Graph Database

Salient features of Neo4j

- Neo4j is schema free – Data does not have to adhere to any convention
- ACID – atomic, consistent, isolated and durable for logical units of work
- Easy to get started and use
- Well documented and large developer community
- Support for wide variety of languages
  - Java, Python, Perl, Scala, Cypher, etc

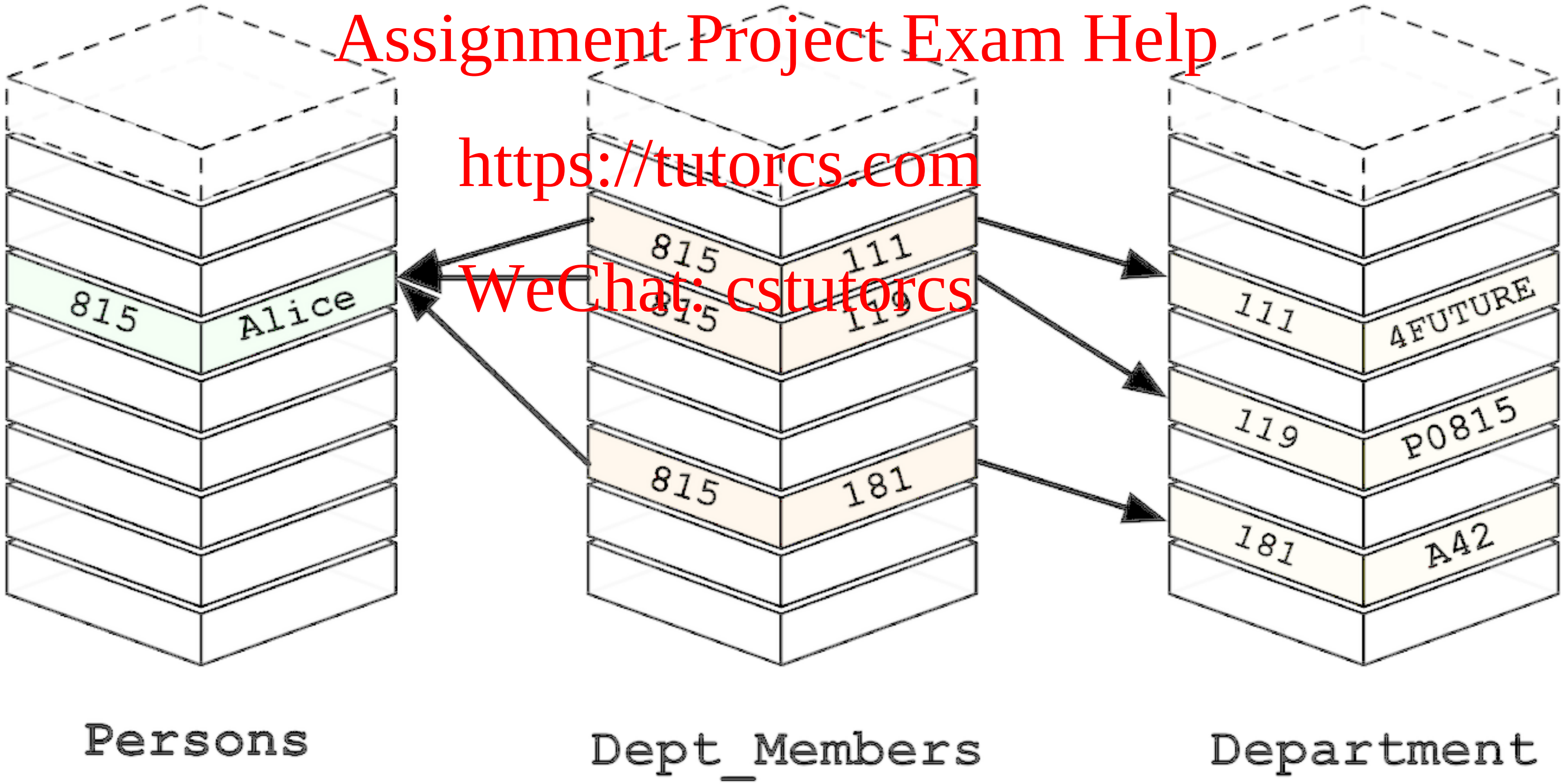
More reasons to use Neo4j

- High performance graph operations

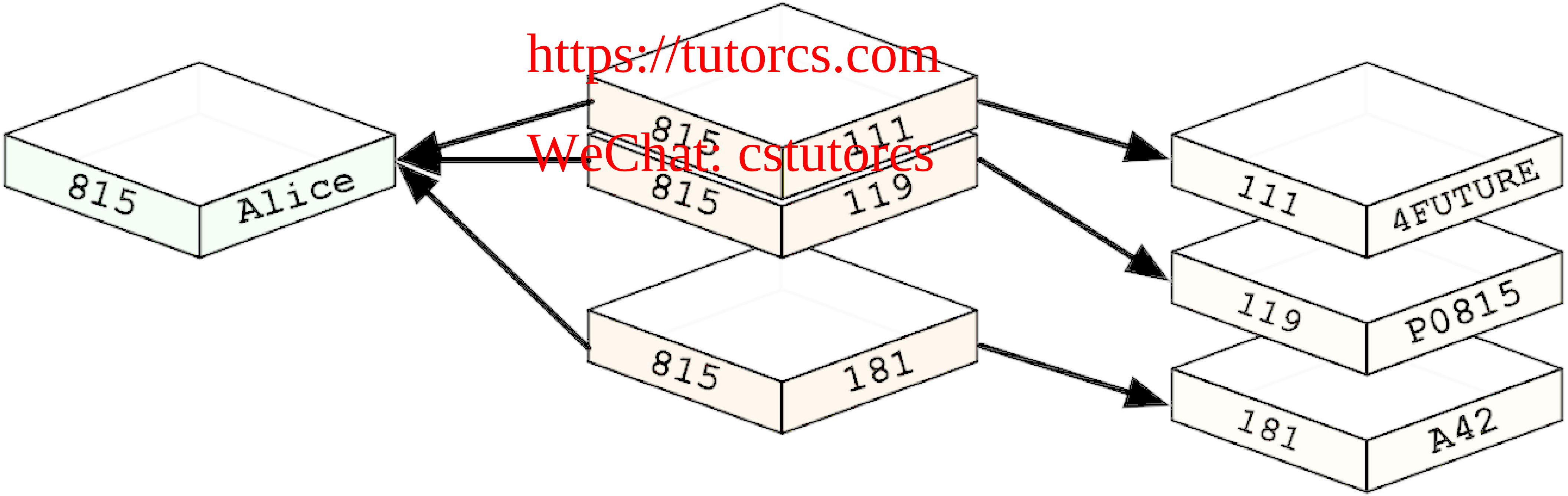
  - Traverse 1,000,000+ relationships/sec on commodity hardware

- 32 billion nodes & relationships per Neo4j instance
- 64 billion properties per Neo4j instance
- Small footprint
- Standalone server is ~65mb

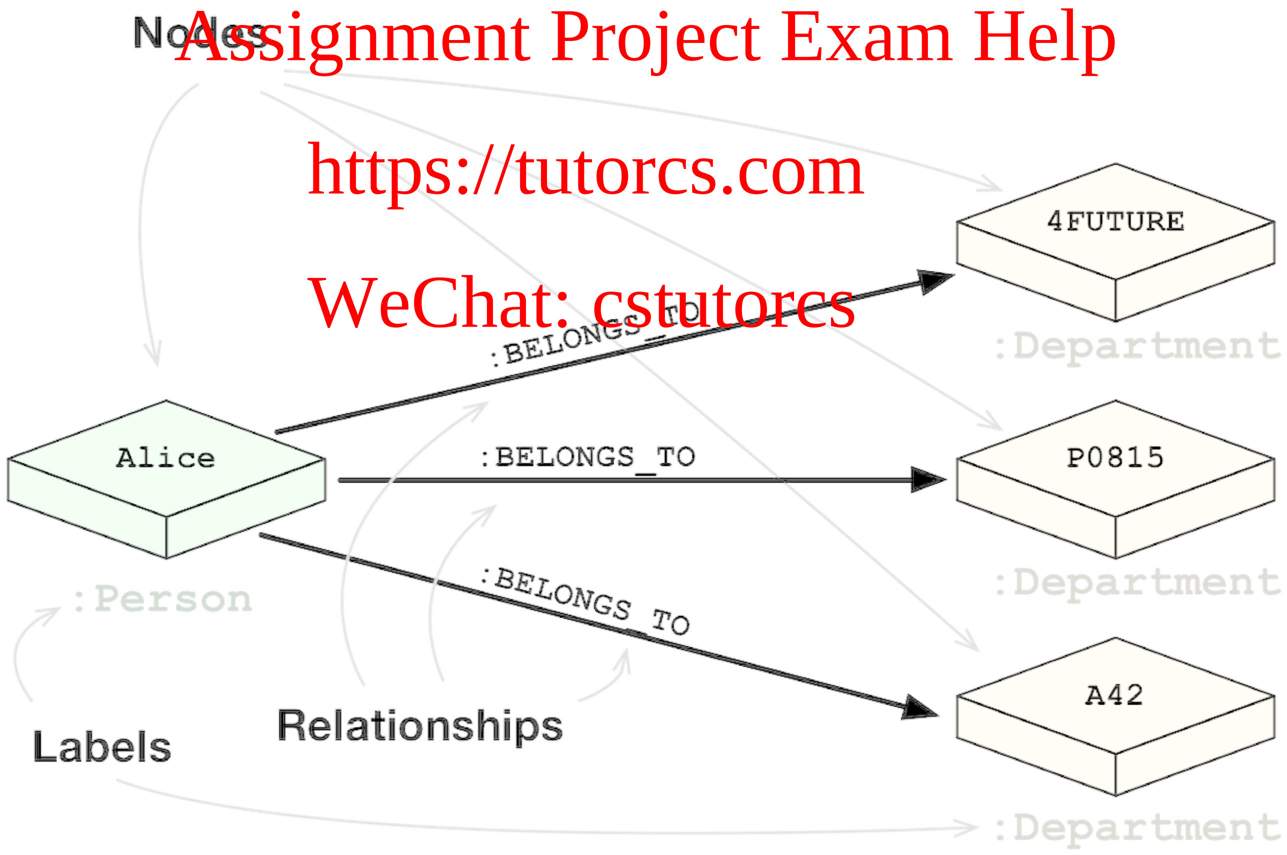
Relational databases



Graph databases



Graph databases



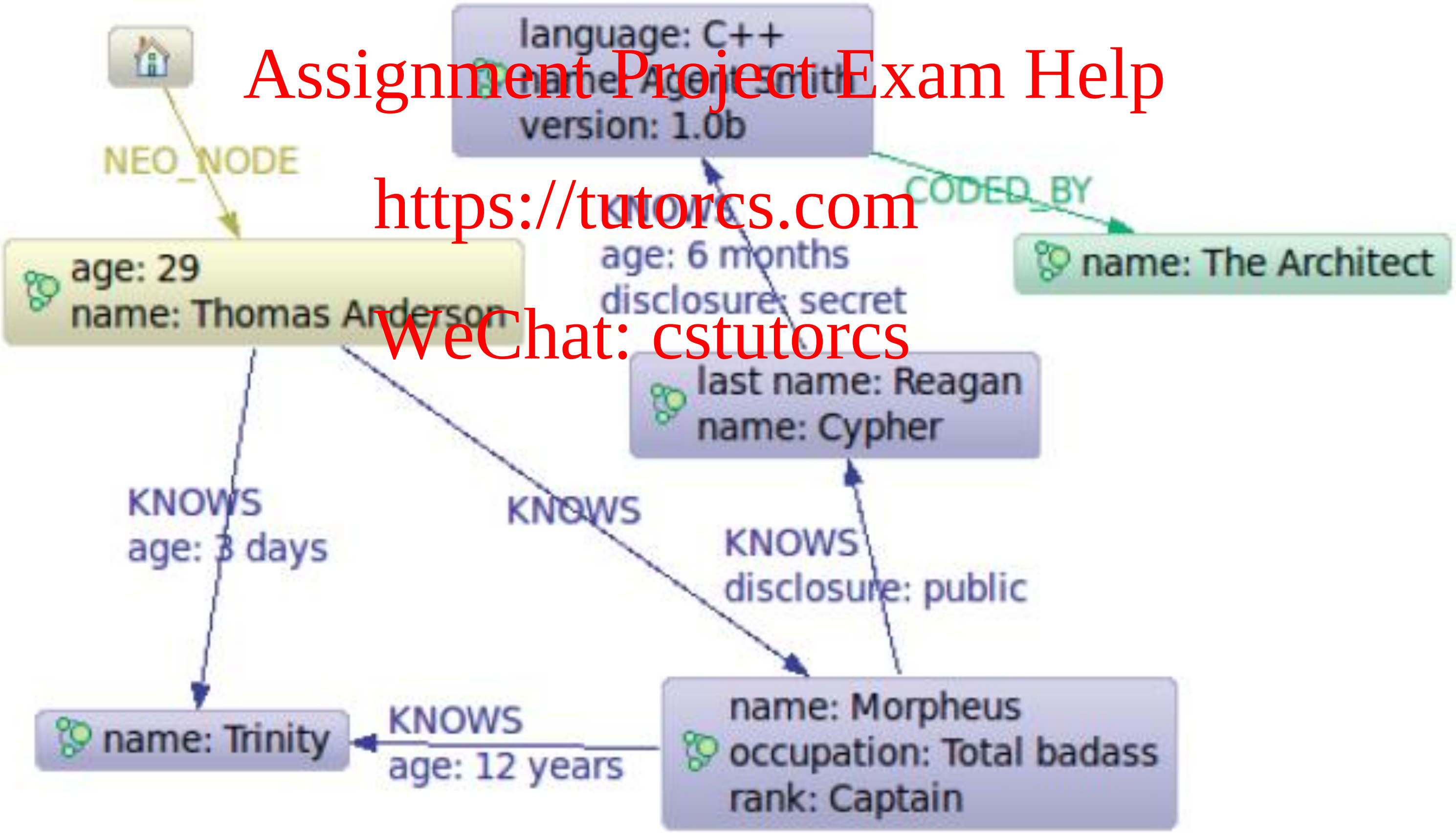
Neo4j design tips: from ER to graph

- **Table to Node Label** – each entity table in the relational model becomes a label on nodes in the graph model.
- **Row to Node** – each row in a relational entity table becomes a node in the graph.
- **Column to Node Property** – columns (fields) on the relational tables become node properties in the graph.
- **Business primary keys only** – remove technical primary keys, keep business primary keys.
- **Add Constraints/Indexes** – add unique constraints for business primary keys, add indexes for frequent lookup attributes.

Neo4j design tips: from ER to graph

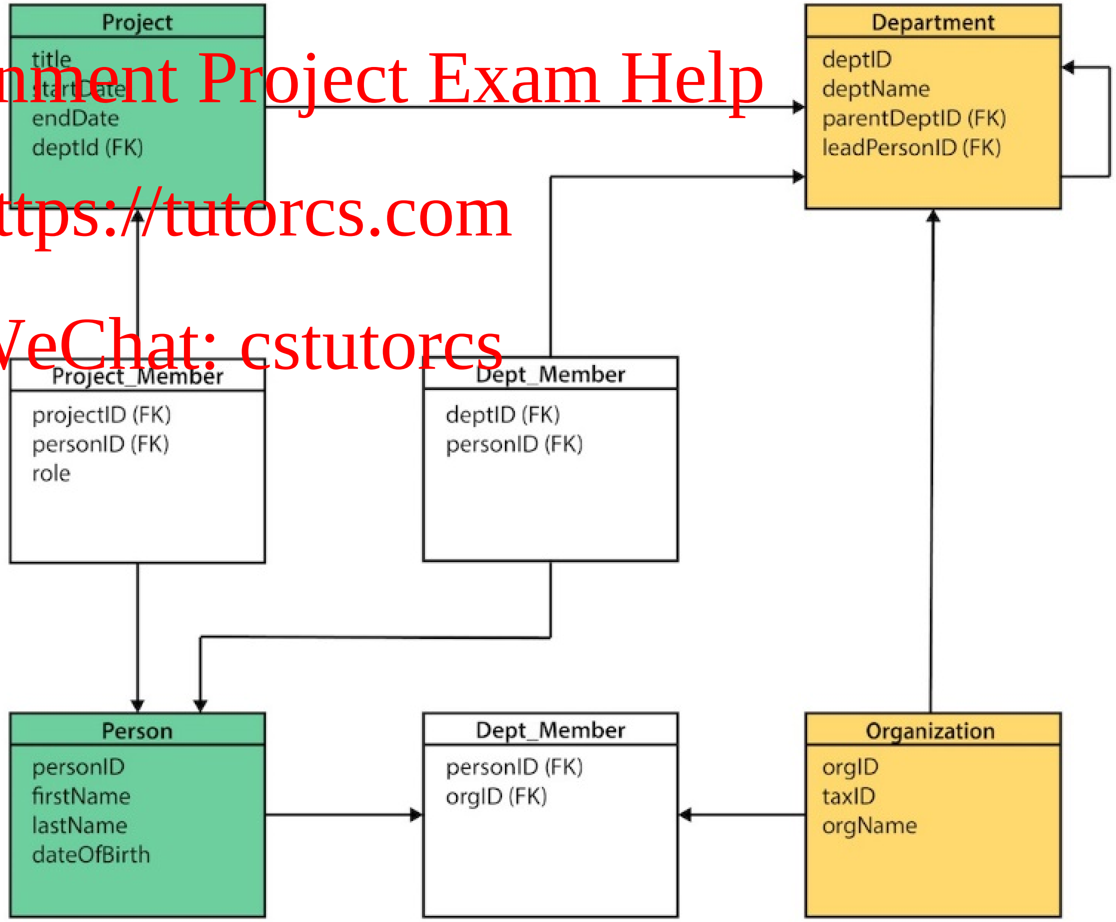
- **Foreign keys to relationships** – replace foreign keys to the other table with relationships, remove them afterwards.
- **No defaults** – remove data with default values, no need to store those.
- **Clean up data** – duplicate data in denormalized tables might have to be pulled out into separate nodes to get a cleaner model.
- **Index columns to array** – indexed column names (like email1, email2, email3) might indicate an array property.
- **Join tables to relationships** – join tables are transformed into relationships, columns on those tables become relationship properties

The matrix graph database

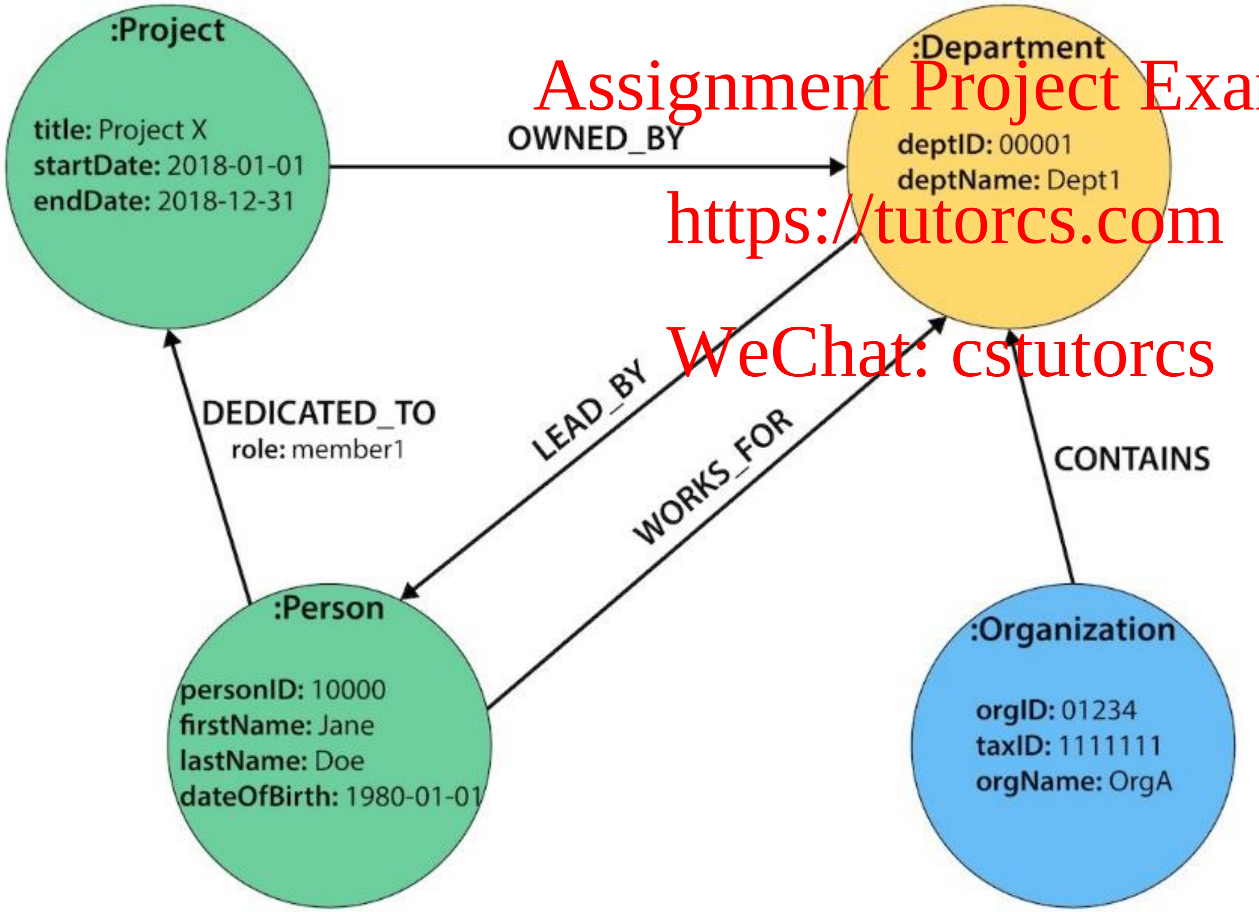


Exercise

Convert this relational model into a graph database

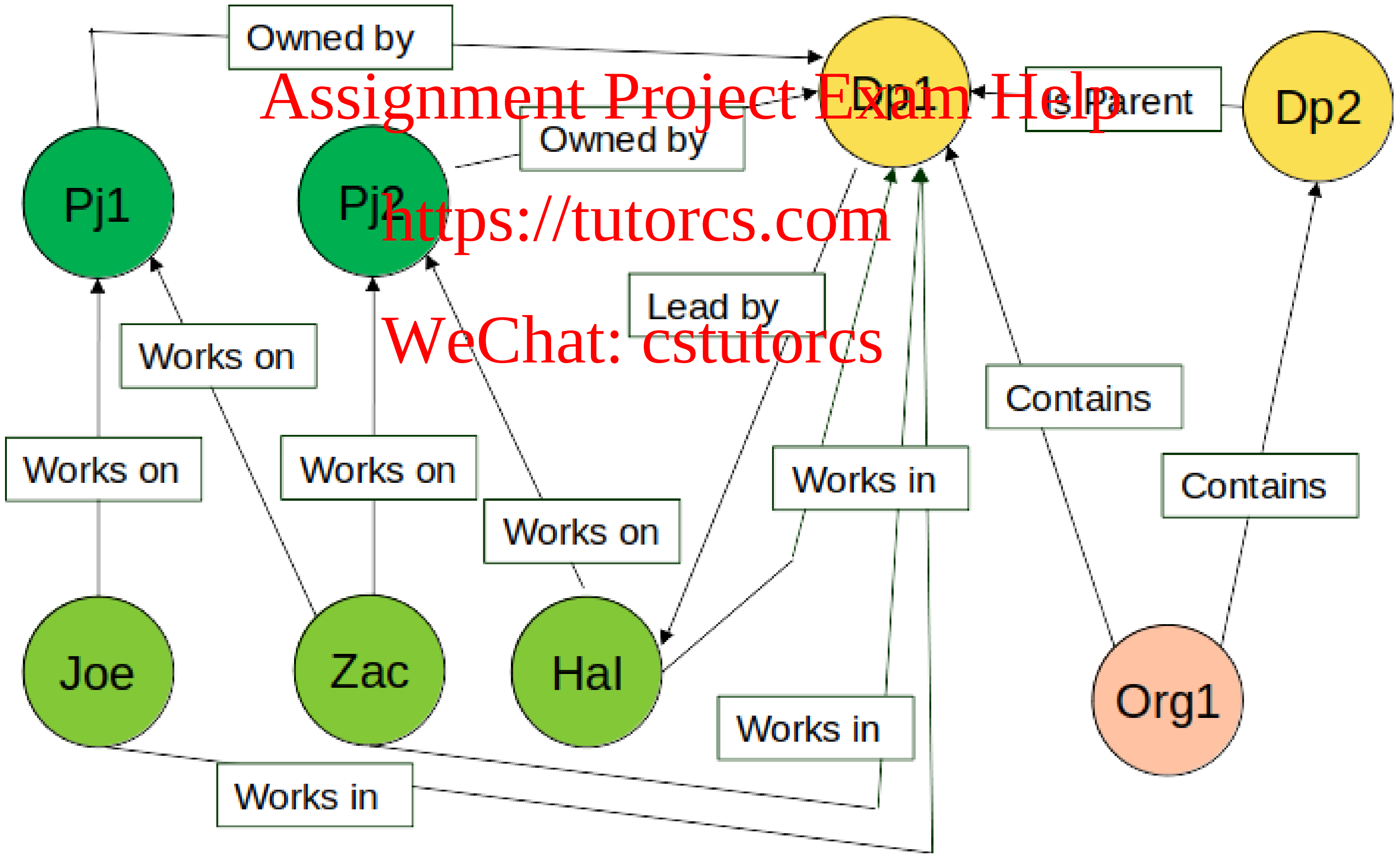


Graph layout



Good Tutorial:
https://dzone.com/refcardz/from-relational-to-graph-a-developers-guide?chapter=1

More detailed solution



Social network performance (traversals)

Why a graph database? MySQL vs Neo4j

Neo4j design tips: from ER to graph

The experiment: round 1

- 1st rule of fight club: Run a friends of friends query
- 2nd rule of fight club: 1,000 users
- 3rd rule of fight club: Average of 50 friends per user
- 4th rule of fight club: Limit the depth of 5
- 5th rule of fight club: Intel i7 commodity laptop w/8GB RAM

Social network performance

RDBMS schema

T_USER

id	name
1	John S
2	Kate H
3	Aleksa V
4	Jack T
5	Jonas P
5	Anne P

T_USER_FRIEND

id	user_1	user_2
1000	1	2
1001	3	5
1002	4	1
1003	6	2
1004	4	5
1005	1	4

Social network performance

SQL: friends of friends at depth 3


```
SELECT distinct uf3.*
FROM t_user_friend uf1
INNER JOIN t_user_friend uf2 on uf1.user_1 = uf2.user_2
INNER JOIN t_user_friend uf3 on uf2.user_1 = uf3.user_2
WHERE uf1.user_1 = user_id
```

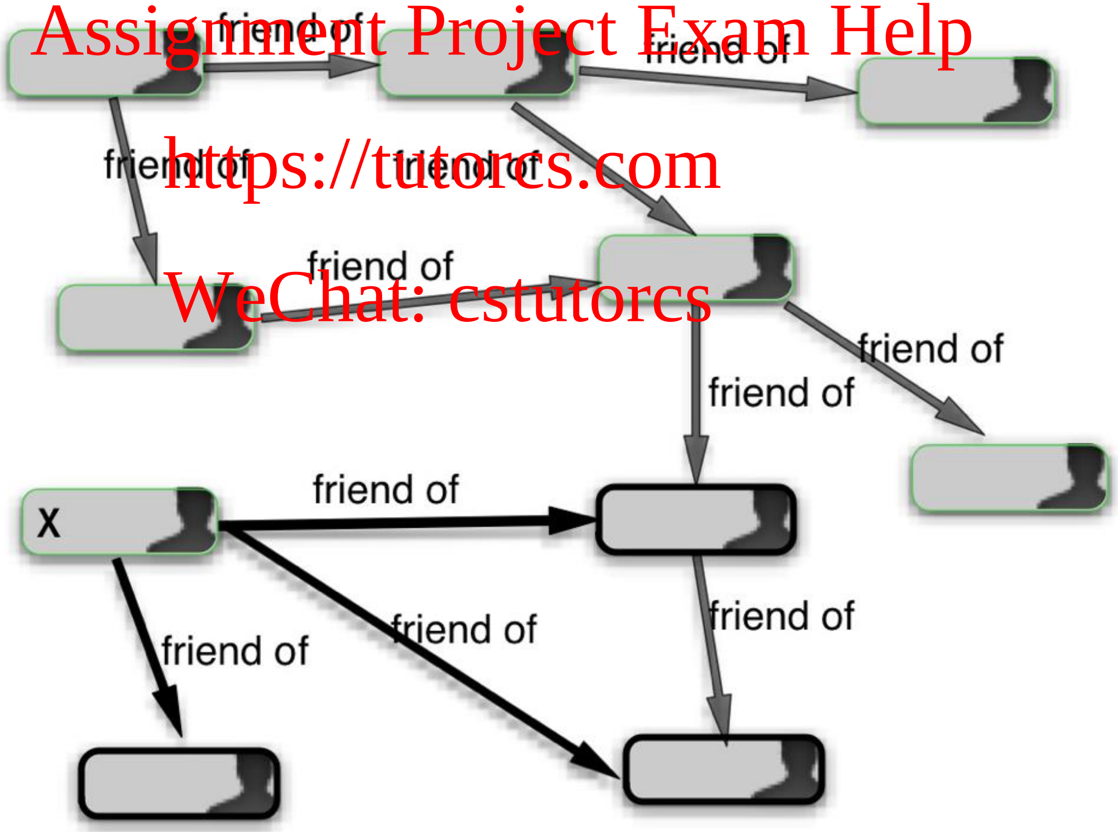
Social network performance

MySQL results: round 1-1,000 users

Depth	Execution time (sec)	Records returned
2	0.028	~900
3	0.213	~999
4	10.273	~999
5	2,613.15	~999

Social network performance

Social graph



Social network performance

Neo4j results: round 1–1,000 users

Depth	Execution time (sec)	Records returned
2	0.04	~900
3	0.06	~999
4	0.07	~999
5	0.07	~999

Neo4j design tips: from ER to graph

The experiment: round 2

- 1st rule of fight club: Run a friends of friends query
- 2nd rule of fight club: 1,000,000 users
- 3rd rule of fight club: Average of 50 friends per user
- 4th rule of fight club: Limit the depth of 5
- 5th rule of fight club: Intel i7 commodity laptop w/8GB RAM

Social network performance

MySQL results: round 1 - 1,000,000 users

Depth	Execution time (sec)	Records returned
2	0.016	~2,500
3	30.267	~125,000
4	1,543.505	~600,000
5	Did not finish after an hour	N/A

Social network performance

Neo4j results: round 1-1,000,000 users

Depth	Execution time (sec)	Records returned
2	0.01	~2,500
3	0.168	~110,000
4	1.359	~600,000
5	2.132	~800,000

Conclusions

- Key questions to ask yourself
  - Is my data going to have a lot of relationships?
  - What sort of questions would I like to ask my database?
- Neo4j is a performant graph database

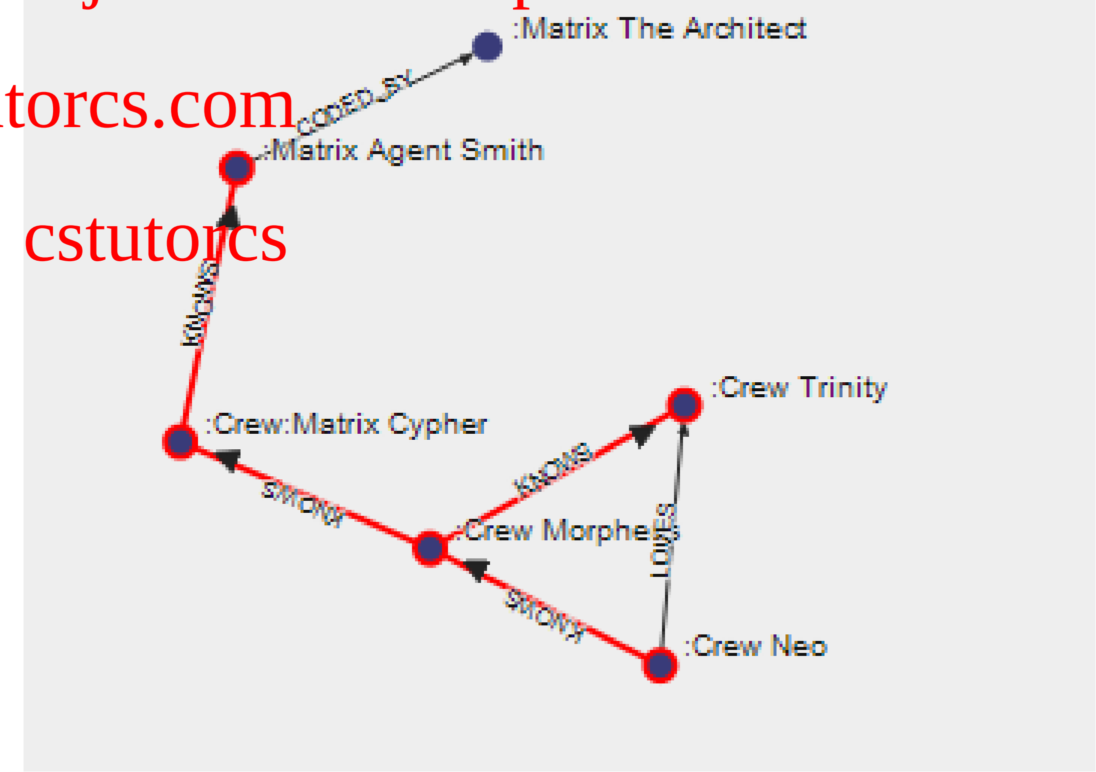
Cypher

- Query Language for Neo4j
- Easy to formulate queries based on relationships
- Many features stem from improving on pain points with SQL such as join tables

Cypher

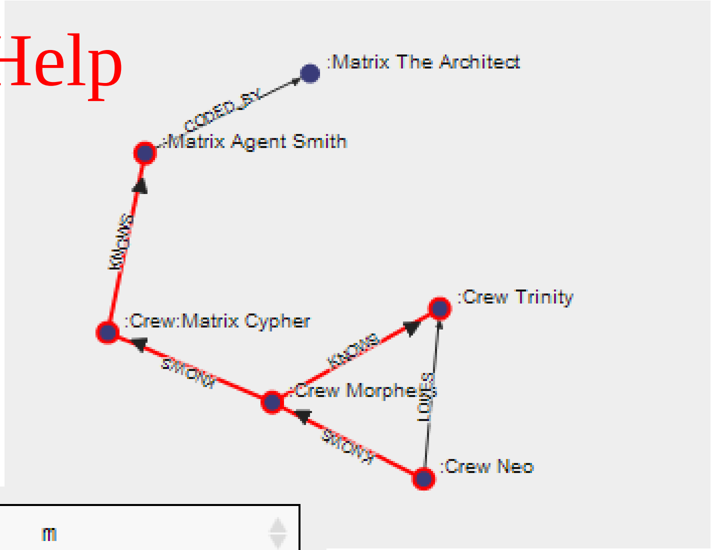
```
CREATE (Neo:Crew { name:'Neo' })
```

```
(Neo)-[:KNOWS]->(Morpheus)
```

Cypher


```
Query:
MATCH (n:Crew)-[r:KNOWS*]-m
WHERE n.name='Neo'
RETURN n AS Neo,r,m
```



Neo	r	m
{name:"Neo"}	[(0)-[0:KNOWS]->(1)]	(1:Crew {name:"Morpheus"})
{name:"Neo"}	[(0)-[0:KNOWS]->(1), (1)-[2:KNOWS]->(2)]	(2:Crew {name:"Trinity"})
{name:"Neo"}	[(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3)]	(3:Crew:Matrix {name:"Cypher"})
{name:"Neo"}	[(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3), (3)-[4:KNOWS]->(4)]	(4:Matrix {name:"Agent Smith"})

Demo

- Create nodes with properties
- Match nodes
- Create relationships between nodes
- Traverse the graph
- Show paths
- Accumulators

Create nodes

- Create nodes

CREATE (e1:Student { name: "Emil", from: "Sweden", age: 29 })

CREATE (e2:Student { name: "Paul", from: "Sweden", age: 29, gender: “m” })

CREATE (s1:Subject { name: “Maths", lecturer: “Julia", age: 29 })

Primary keys / constraints


```
CREATE CONSTRAINT ON (book:Book)
ASSERT book.isbn IS UNIQUE

CREATE CONSTRAINT ON (book:Book)
ASSERT EXISTS book.isbn
```

Match nodes

- Match is used to select nodes. A match query must be ended by a Return statement.

MATCH (e:Student) return e;

MATCH (e:Student {age < 25}) return e;

MATCH (e:Student {name: “Emil”});

Create relationships

- Nodes are connected by relationships. A relationship can also have nodes.


```
MATCH (e:Student {name:'Emil'}), (r:Student {name:'Joe'}) CREATE (e)-[:FRIEND_OF]->(r)
```

```
MATCH (s:Student {name: "Emil"}),(s1:Subject {name: "math"})
CREATE (s)-[r:MARK{date:"12/12/2014", mark:55}]->(s1)
```

Navigate the graph

- Neo4j allow to easily navigate the graph


```
MATCH (e:Student {name:"Emil"})-[:FRIEND_OF*1..3]-(e2:Student)
RETURN e2
```

```
MATCH (e:Student {name:"Emil"})-[:FRIEND_OF*1..3]-(e2:Student)
RETURN e2.from, COUNT(e2.from) as numfriends
```

```
MATCH (e:Student {name:"Alice"})
MATCH path = shortestPath( (e)-[:FRIEND_OF*..5]-(m:Student {name:"Mary"}) )
RETURN path
```

Navigate the graph


```
MATCH (you {name:"You"})
MATCH (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"})
MATCH path = shortestPath( (you)-[:FRIEND*..5]-(expert) )
RETURN db, expert, path
```

Reduce function

It will go through a list, run an expression on every element, storing the partial result in the accumulator.

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

Arguments:

- accumulator: A variable that will hold the result and the partial results as the list is iterated
- initial: An expression that runs once to give a starting value to the accumulator
- list: An expression that returns a list
- expression: This expression will run once per value in the list and produces the result value.

Reduce function

MATCH (e:Student {name:"Alice"})

MATCH path = shortestPath((e)-[:FRIEND_OF*..5] -(m:Student {name:"Mary"}))

RETURN

reduce(tot = 0, n IN nodes(path) | tot + n.age) as tot_age

Sample code

```
CREATE (shakespeare:Author {firstname:'William', lastname:'Shakespeare'}),
       (juliusCaesar:Play {title:'Julius Caesar'}),
       (shakespeare)-[:WROTE_PLAY {year:1599}]->(juliusCaesar),
       (theTempest:Play {title:'The Tempest'}),
       (shakespeare)-[:WROTE_PLAY {year:1610}]->(theTempest),
       (rsc:Company {name:'RSC'}),
       (production1:Production {name: 'Julius Caesar'}),
       (rsc)-[:PRODUCED]->(production1),
       (production1)-[:PRODUCTION_OF]->(juliusCaesar),
       (performance1:Performance {date:20120729}),
       (performance1)-[:PERFORMANCE_OF]->(production1),
       (production2:Production {name:'The Tempest'}),
       (rsc)-[:PRODUCED]->(production2),
       (production2)-[:PRODUCTION_OF]->(theTempest),
       (performance2:Performance {date:20061121}),
       (performance2)-[:PERFORMANCE_OF]->(production2),
       (performance1)-[:VENUE]->(theatreRoyal),
       (performance2)-[:VENUE]->(theatreRoyal),
       (newcastle:City {name:'Newcastle'}),
       (theatreRoyal)-[:CITY]->(newcastle),
```

Sample query


```
MATCH (theatre: Venue {name: 'Theatre Royal'}), (newcastle: City {name: 'Newcastle'}), (bard:
Author {lastname: 'Shakespeare'}),
(newcastle)<-[:CITY]-(theatre)<-[:VENUE]-()-[:PERFORMANCE_OF]->()
-[:PRODUCTION_OF]->(play)<-[:WROTE_PLAY]-(bard)

RETURN DISTINCT play.title AS play
```


The identifiers newcastle, theatre, and bard are anchored to real nodes in the graph based on the specified label and property values.

The syntax (theatre)<-[:VENUE]-() uses the anonymous node, hence the empty parentheses.