# Models and Systems for Big Data Management

## Graph Databases - Neo4j

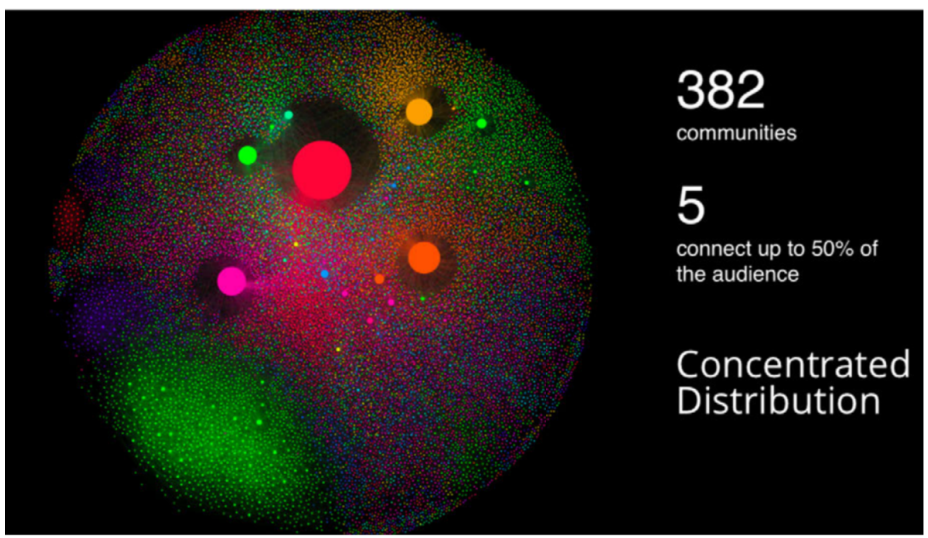
Nacéra Seghouani

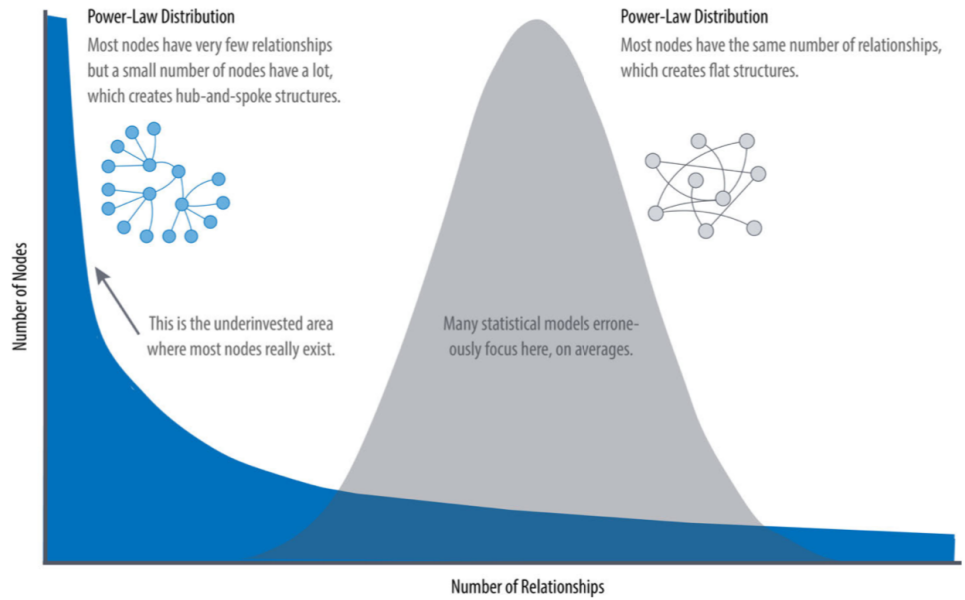
Computer Science Department, CentraleSupélec
Laboratoire de Recherche en Informatique, LRI LaHDAK group
nacera.bennacer@centralesupelec.fr

2019-2020

# Introduction

- Connected information is everywhere in real data. Relationships within real-world systems:
  - from protein interactions to social networks,
  - from communication systems to power grids,
  - and from retail experiences to Mars mission planning.
- Growth of networks that connectivity increases over time, but not uniformly.
  - This gaming community analysis shows a concentration of connections around just 5 of 382 communities.
  - The web, in common with graphs like social networks, has a power-law distribution with a few nodes being highly connected and most nodes being modestly connected.



# Introduction

- Graph algorithms help to make sense of connected data and provides theoretical base for understanding structure/topology revealing patterns in large highly connected datasets.
- The types of questions graph analytics answer

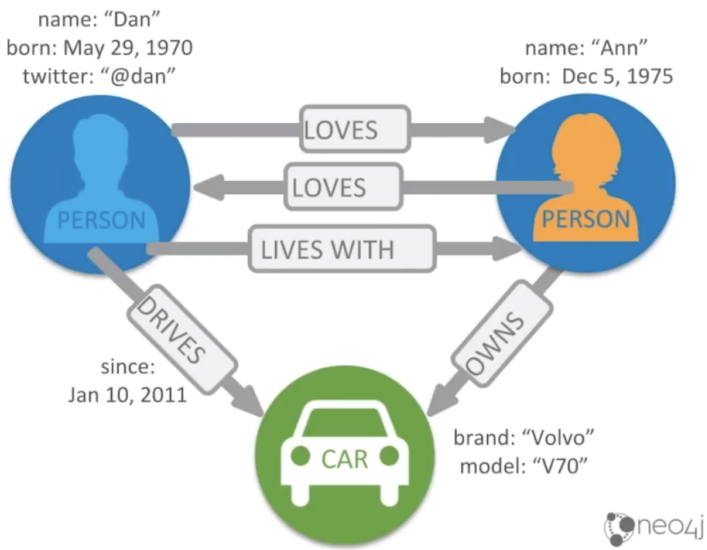

- Investigate the epidemic propagation or a cascading transport failure.
- Identify the most vulnerable, or damaging, components in a network attack.
- Identify the least costly or fastest way to route information or resources.
- Reveal communities based on behavior for personalized recommendations.
- Predict missing links in your data, whether groups will merge or break apart.
- Locate influences in a complex system.
- Visualise, extract more/relevant predictive features for machine learning.
- ...

# Introduction

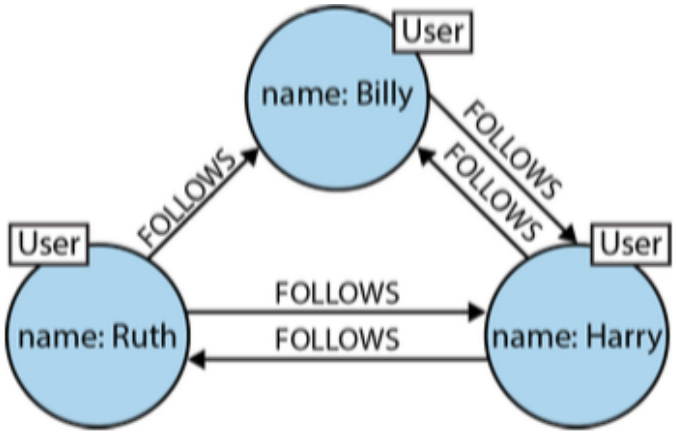
- Relational/SQL databases allows to represent entities using relations/tables, connected using foreign keys, querying need join operations
- Graph-oriented databases handle connected information providing a view on data that fits targeted relationships
- Graph-oriented database natively embraces relationships to be able to store, process, and query connections efficiently
- Graph Databases:
  - Giraph (Apache, on Haddop inspired by Google Pregel),
  - GraphX (Apache Spark's API),
  - GraphDB, OrientDB (SAP), Amazon Neptune, ...
  - **Neo4j**

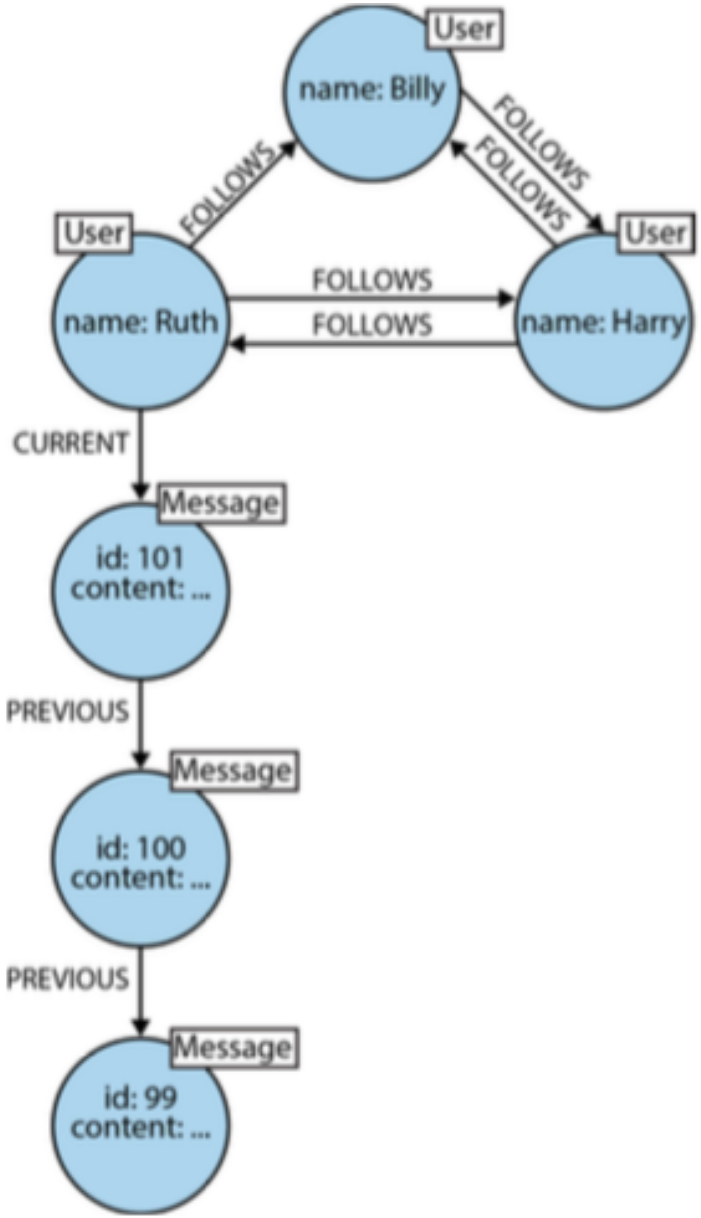
# Neo4J Labeled Property Graph Data Model

- **Nodes**
  - are the name for data records in a graph, grouped together using a **Label**
  - a node can have zero or more labels
  - a label doesn't have any property, has a native indexe
- **Relationships** relate nodes
  - directed, no broken with a start and end node, and of zero or one **Type**. No specified direction means the two direction. A node can have relationships to itself.
- **Properties** are name/value pairs associated to nodes or relationships
  - a variety of data types, from numbers and strings to spatial and temporal data.



# Neo4J Labeled Property Graph Data Model



# Neo4J & Cypher

- ☞ Neo4j is a schema-free graph database, handles complex, densely-connected data.
- ☞ Neo4j provides **Cypher**, a declarative query language to specify patterns to find data that matches these patterns.
- ☞ Cypher provides SQL-like clauses and keywords as: `MATCH`, `WHERE` and `DELETE`.

# Neo4J & Cypher

## Patterns

### ✎ Node pattern syntax

```
()  (m)  // any node
(:Movie) (m:Movie)  (m:Person:Actor) //a node labeled Movie,  Person and Actor
(m:Movie {title: "The Matrix"}) //a node labeled Movie  with properties
(m:Movie {title: "The Matrix", released: 1997})
```

### ✎ Relationship patten syntax

```
-->       -[r]->  // any directed relationship
-[:ACTED_IN]->   -[r:ACTED_IN]->  //a relationship typed ACTED_IN
-[r:ACTED_IN {roles: ['Neo']}]-> //a labeled  relationship with properties
(a)-[r:TYPE1|TYPE2]->(b) //relationship could have any one of a set of types
```

### ✎ Path pattern

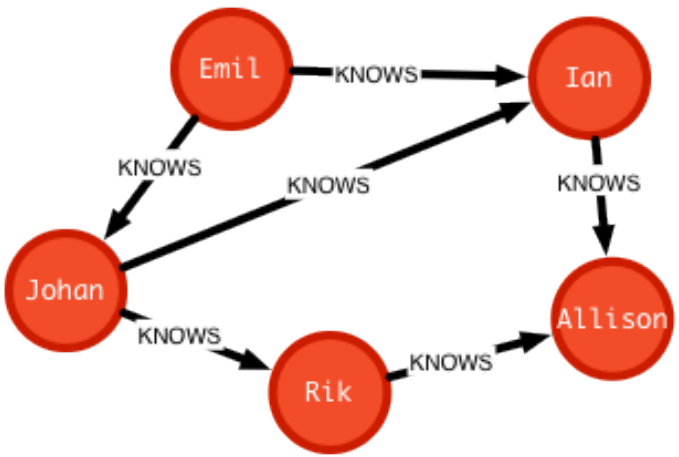
```
(a)->()->(b)    //3 nodes, two relationships, in one path  of length 2
(a)-[*2]->(b)   //specifying a length in the relationship
(a)-[*3..5]->(b)  //variable length relationships
(me)-[:KNOWS*1..2]-(remote_friend)
(a)-[*]->(b)
```

# Neo4J & Cypher

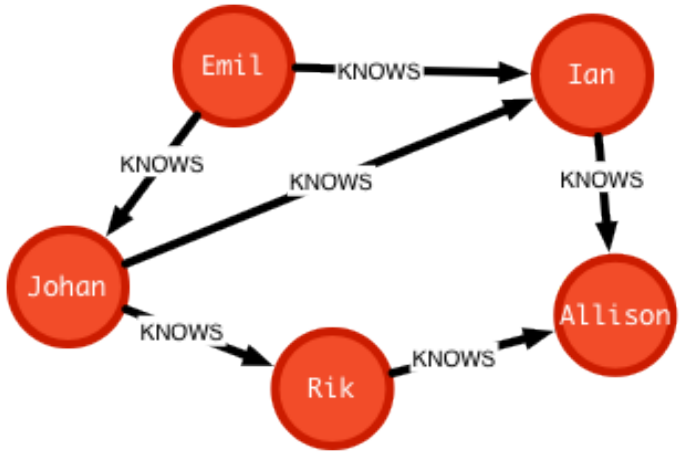
```
CREATE (e:Person { name: "Emil", from: "Sweden"})
```

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

```
MATCH (e:Person) WHERE e.name = "Emil" CREATE
(j:Person { name: "Johan", from: "Sweden", learn: "surfing" }),
(i:Person { name: "Ian", from: "England", title: "author" }),
(r:Person { name: "Rik", from: "Belgium", pet: "cat" }),
(a:Person { name: "Allison", from: "California", hobby: "surfing" }),
(e)-[:KNOWS {since: 2001}]->(j), (e)-[:KNOWS {rating: 5}]->(i),
(j)-[:KNOWS]->(i), (j)-[:KNOWS]->(r),  (i)-[:KNOWS]->(a),  (r)-[:KNOWS]->(a)
```




# Neo4J & Cypher




- Find friends of Emil

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

- Find persons who like surfing that are friends of friends of Johan

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

# Cypher: Data value types

- Property types
  - Can be: returned from Cypher queries, used as parameters, stored as properties, constructed with Cypher literals
  - Number (Integer and Float), String, Boolean, Point, Date, Time, LocalTime, DateTime, LocalDateTime and Duration
  - **Null is not a valid property value.**
- Structural types
  - Can be returned from Cypher queries
  - Nodes, comprising: Id, Label(s), Map (of properties)
  - Relationships, comprising: Id, Type, Map (of properties), Id of the start and end nodes
  - Paths: An alternating sequence of nodes and relationships
- Composite types
  - Can be returned from Cypher queries, used as parameters, constructed with Cypher literals
  - Lists heterogeneous, ordered collections of values, each of which has any property, structural or composite type.
  - Maps are heterogeneous, unordered collections of (key, value) pairs, where: the key is a String the value has any property, structural or composite type

# Cypher: Data value types

- Values

```
boolean (true/false/TRUE/FALSE)
byte, short, int, long, float, double 13, -4,  3.14, 6.022E23, 0xFC3A9
char, String: 'Hello', "World"
Lists of any  type of value [0, "1", [2, 3]]
```

- Mathematical operators

```
+, -, *, / and %, ^
```

- Comparison operators

```
=, <>, <, >, <= , >=, IS NULL, IS NOT NULL
3 < 4, "x" < "xy", false < true
```

- Logical operators

```
AND , OR , XOR , NOT
```

- String operators

```
+, STARTS WITH, ENDS WITH, CONTAINS, =~  (regular expression matching)
MATCH (n) WHERE n.name STARTS WITH {"Micha"} RETURN n.name
```

- Lists can be concatenated using the + operator.
  `IN` operator: checks if an element exists in a list.

# Cypher: Creating Data

- Creating Data: Cypher returns the number of changes or explicit variables.

```
1) CREATE (:Movie {title:'The Matrix', released:1997 })

2) CREATE (p:Person {name:'Keanu Reeves', born:1964}) RETURN p

3) CREATE (a:Person { name:"Tom Hanks",  born:1956 })
-[r:ACTED_IN { roles: ["Forrest"]}]->
(m:Movie { title:"Forrest Gump",released:1994 })

4) CREATE (d:Person { name:"Robert Zemeckis", born:1951 })-[:DIRECTED]->(m)
RETURN a,d,r,m
```

- Use parameters with CREATE.

```
{'props' : { 'name' : 'Andres' , 'position' : 'Developer'}}
CREATE (n:Person $props) RETURN n
```

# Cypher: Matching Data

✎ Matching Patterns: Cypher returns more ...

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

2) MATCH (p:Person { name:'Tom Hanks' })-[r:ACTED_IN]->
(m:Movie) RETURN m.title, r.roles

3) MATCH (john {name: 'John'})-[:friend]->()-[:friend]->(fof)
RETURN  fof.name

4) MATCH (j:Person {name: 'John'})-[:KNOWS]-()-[:KNOWS]-(s)
RETURN DISTINCT s upper(s.name) AS name

5) MATCH (n) RETURN n.name, n.age ORDER BY n.name DESC SKIP {30} LIMIT {10}
```

# Cypher: Filtering Data

- WHERE to filtering results using boolean expressions :
  - on node labels, properties, on relationship properties, on paths
  - on lists, on string regular expressions
  - missing properties evaluated to null

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

2) Match (n) WHERE n:Actor return n.name

3) MATCH (a) WHERE a.name > 'Andy' AND a.name < 'Timothy' RETURN a.name, a.age

4) MATCH (p:Person)-[r:ACTED_IN]->(m:Movie) WHERE
 m.released > 2000 OR 'Neo' IN r.roles RETURN p, r, m

5) MATCH (user)-[:friend]->(follower) WHERE
user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve'] AND follower.name =~ 'S.*'
RETURN user.name, follower.name

6) MATCH (n)-[k:KNOWS]->(f) WHERE k.since < 2000 RETURN f.name, f.age, f.email

7) MATCH (me)-[:KNOWS*1..2]-(remote_friend) WHERE me.name = 'Filipa'
RETURN remote_friend.name
```

# Cypher: Filtering Data

- `type(r)` and `labels(n)` return resp. the type of `r` and labels of `n` if any; otherwise null.

```
1) MATCH (timothy { name: 'Timothy' }),(others)
WHERE others.name IN ['Andy', 'Peter'] AND (timothy)<--(others)
RETURN others.name, others.age

2) MATCH (persons),(peter { name: 'Peter' }) WHERE NOT (persons)-->(peter)
RETURN persons.name, persons.age

3) MATCH (n)-[r]->() WHERE n.name='Andy' AND type(r)=~ 'K.*' RETURN type(r), r.since

4) MATCH (n) WHERE n.city= 'xxx' OR n.belt IS NULL
RETURN n.name, n.age, n.city ORDER BY n.name

5) MATCH (n)-[]-() WHERE labels(n) is not null RETURN n
```

# Cypher: Combinated statements

✎ Subsequent CREATE statements are executed once for each matched row.

```
1) MATCH (p:Person { name:'Tom Hanks'})
CREATE (m:Movie { title:'Cloud Atlas',released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]->(m) RETURN p,r,m

2) MATCH (a:Person),(b:Person) WHERE a.name = 'A' AND b.name = 'B'
CREATE (a)-[r:RELTYPE]->(b) RETURN type(r)
```

# Cypher: Merging nodes

- `MERGE` acts like a combination of `MATCH` or `CREATE`, but checks first for the existence of data before creating it.
- `MERGE ON CREATE SET` merges a node and set properties if the node needs to be created.
- `MERGE ON MATCH SET` merges nodes and setting properties on found nodes.

```
1) MERGE (robert:Critic) RETURN robert, labels(robert)

2) MERGE (charlie { name: 'Charlie Sheen', age: 10 }) RETURN charlie

3) MATCH (person:Person)
MERGE (city:City { name: person.bornIn }) RETURN person.name, person.bornIn, city

4) MERGE (m:Movie { title:"Cloud Atlas" }) ON CREATE SET m.released = 2012 RETURN m

5) MERGE (person:Person)
ON MATCH SET person.found = TRUE , person.lastAccessed = timestamp()
RETURN person.name, person.found, person.lastAccessed

6) MERGE (keanu:Person { name: 'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
ON MATCH SET keanu.lastSeen = timestamp()
RETURN keanu.name, keanu.created, keanu.lastSeen
```

# Cypher: Merging relationships

- `MERGE` can be used to simultaneously create both a new node and a relationship.
- `MERGE` used on a whole pattern, either everything matches, or everything is created.

```
1) MATCH (c:Person { name: 'Charlie Sheen' }),(w:Movie { title: 'Wall Street'
MERGE (c)-[r:ACTED_IN]->(w) RETURN c.name, type(r), w.title

2) MATCH (o:Person { name: 'Oliver Stone' }),(r:Person { name: 'Rob Reiner' })
MERGE (o)-[:DIRECTED]->(movie:Movie)<-[:ACTED_IN]-(r) RETURN movie

3) MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city

4) MATCH (m:Movie { title:"Cloud Atlas" })
MATCH (p:Person { name:"Tom Hanks" })
MERGE (p)-[r:ACTED_IN]->(m) ON CREATE SET r.roles =['Zachry'] RETURN p,r,m
```

# Cypher: Aggregating

- Some aggregation functions: `count , sum , avg, collect, ...`
- The first returned variables which are not aggregated function will be the grouping key

```
1) MATCH (:Person) RETURN count(*) AS people

2) MATCH (a:Person)-[:ACTED_IN]->(m:Movie) RETURN a, count(*) AS appearances
ORDER BY appearances DESC LIMIT 10;

3) MATCH (m:Movie)<-[:ACTED_IN]-(a:Person) RETURN m.title AS movie,
collect(a.name) AS cast, count(*)

4) MATCH (m:Movie)<-[:ACTED_IN]-(a:Person) RETURN m.title AS movie,
collect(a.name) AS cast, count(*), a.name

5) MATCH (u:User)-[r:RATED]->(m:Movie) RETURN count(r) ORDER BY count(r) LIMIT

6) MATCH (u:User)-[r:RATED]->(m:Movie) RETURN u, count(r) ORDER BY  count(r)
LIMIT 10
```

# Cypher: Chaining

- ✎ `WITH` allows query parts to be chained together, piping the results from one to be used as starting point in the next.
- ✎ using `WITH` allows to specify the aggregation, and to filter on aggregated groups.

```
MATCH (david { name: 'David' })-[:followedBy]->(otherPerson)
WITH otherPerson, count(*) AS followers
WHERE followers> 1000
RETURN otherPerson.name

MATCH (n)
WITH n
ORDER BY n.name DESC LIMIT 3
RETURN collect(n.name)
```