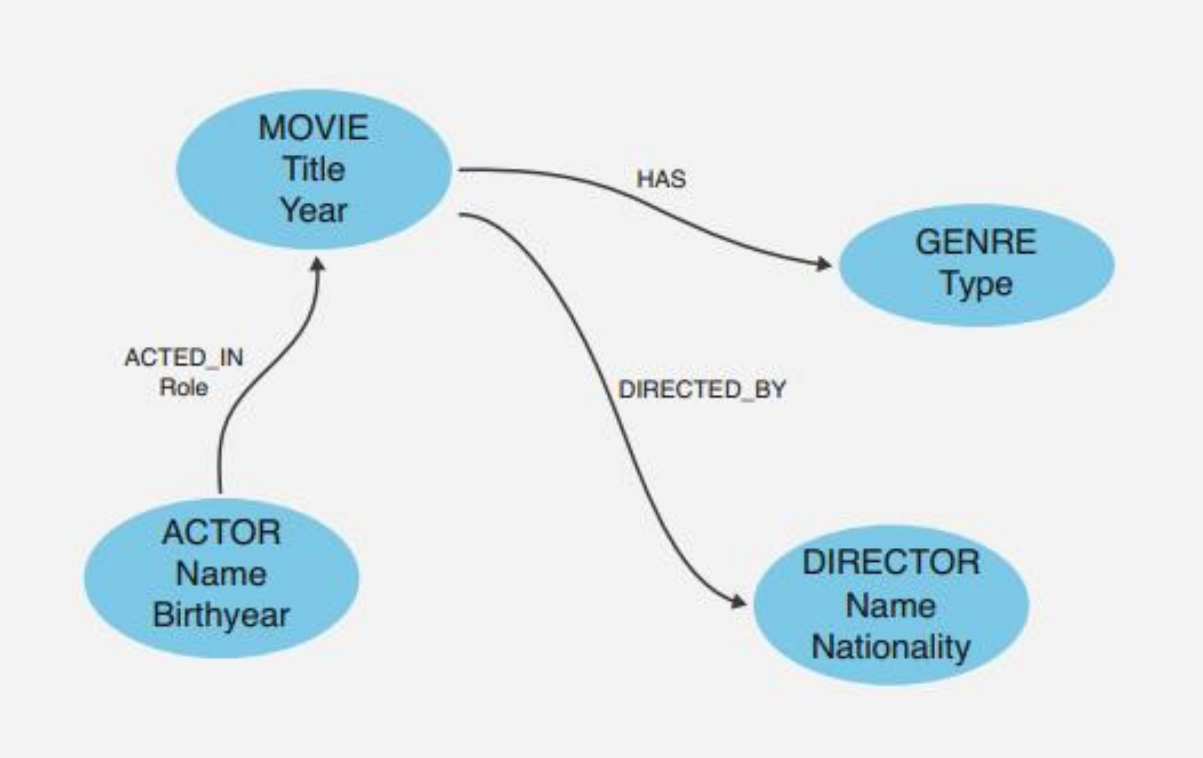
Lecture 6: Graphical databases

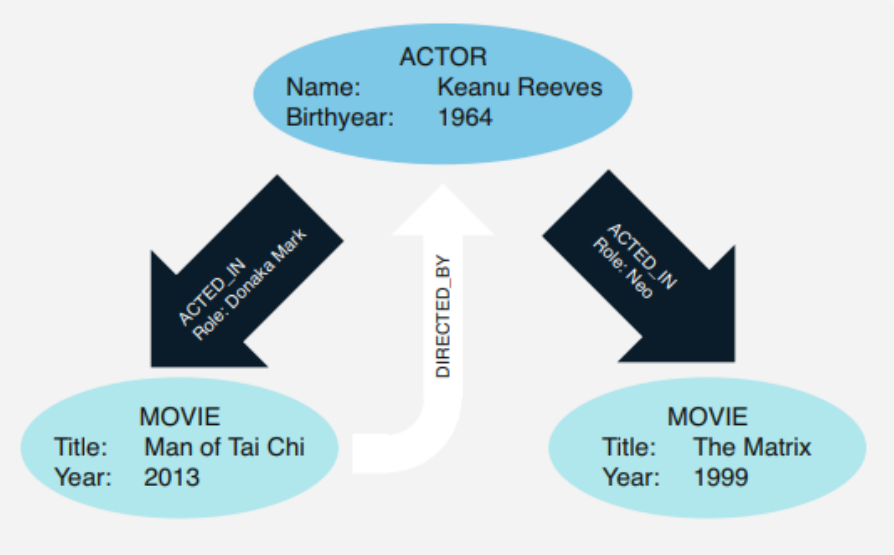
Dr Anesu Nyabadza



Property graphs consist of nodes (concepts, objects) and directed edges (relationships) connecting the nodes. Both nodes and edges are given a label and can have properties. Properties are given as attribute-value pairs with the names of attributes and the respective values


IS FRIEND OF
LIKES
name:
Zush1 Zam
neo4j
SERVES
LOCATED IN
cuisine:
1Sush1
IS FRIEND OF
LIKES
SERVES
location:
New York
name:
1Sush1
LOCATED IN

A graph abstractly presents the nodes and edges with their properties. Consider the movie example → It contains the nodes MOVIE with attributes Title and Year (of release), GENRE with the respective Type (e.g., crime, mystery, comedy, drama, thriller, western, science fiction, documentary, etc.), ACTOR with Name and Year of Birth, and DIRECTOR with Name and Nationality.
The example uses three directed edges: The edge ACTED_IN shows which artist from the ACTOR node starred in which film from the MOVIE node.
This edge also has a property, the Role of the actor in the movie. The other two edges, HAS and DIRECTED_BY, go from the MOVIE node to the GENRE and DIRECTOR node, respectively.
In the manifestation level, i.e., the graph database, the property graph contains the concrete values see **figure below**. For each node and for each edge, a separate record is stored.
Thus, in contrast to relational databases, the connections between the data **are not stored and indexed as key references**, but as **separate records**. This leads to **efficient processing** of network analyses.



Neo4J- Applications

1. Neo4j's graph database model excels in **social network analysis**, which relies heavily on relationships between users, posts, comments, and likes. It can quickly navigate complex social graphs to find connections, influencers, and communities. Traditional SQL databases would necessitate several JOIN operations and sophisticated queries to produce comparable results, resulting in lower speed.

2. **Recommendation systems** rely on understanding user preferences and item relationships. Neo4j's ability to model and traverse complicated relationships makes it ideal for developing recommendation engines. Neo4j, which analyzes user interactions and item similarities inside a graph structure, can provide personalized suggestions with more accuracy and efficiency than SQL databases.

3. Detecting and investigating fraud, including financial and identity theft, requires studying complex transaction networks and relationships. Neo4j's graph database is excellent at describing and querying complicated networks, making it perfect for fraud detection and investigation. It outperforms SQL databases in terms of detecting suspicious patterns, revealing hidden relationships, and visualizing fraud networks.

4. Network and IT Operations Management entails examining dependencies, configurations, and performance data for interconnected systems. Neo4j's graph database is ideal for modeling network topologies, relationships, and configurations, allowing for effective root cause investigation, impact assessment, and capacity planning. When it comes to network operations management, Neo4j outperforms SQL databases in terms of visibility and query perfo

Neo4J- Applications

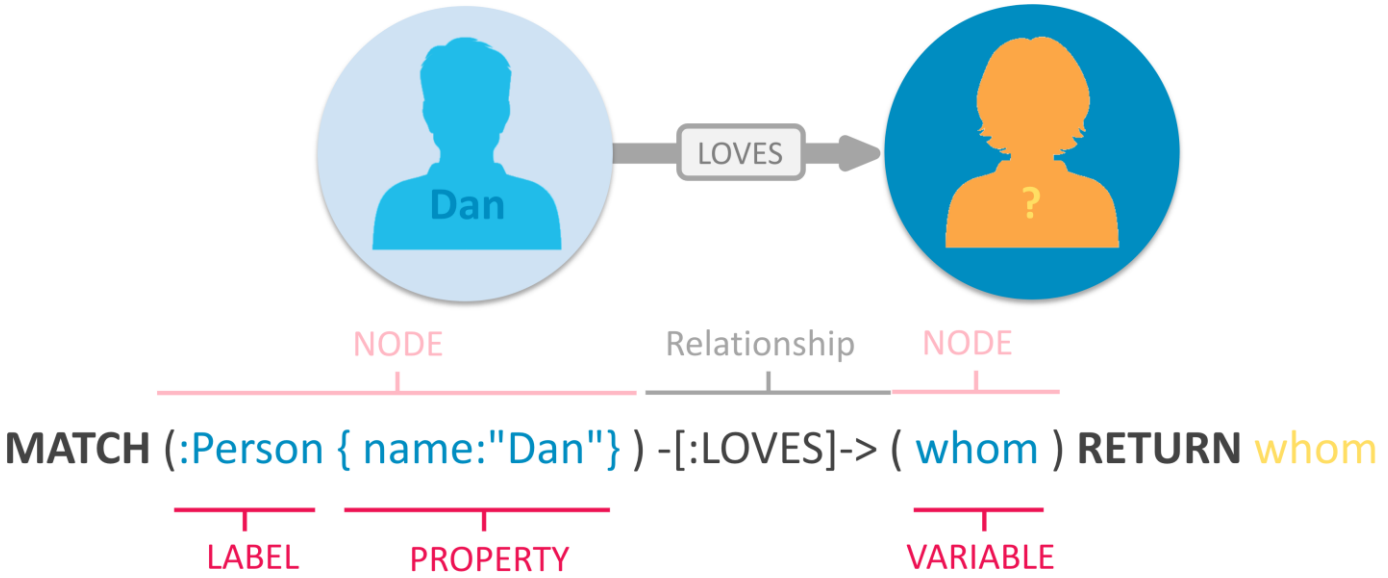
5. In genomics and bioinformatics research, evaluating biological data frequently entails investigating complex interactions between genes, proteins, pathways, and illnesses. Neo4j's graph database is perfect for describing and querying biological networks, allowing researchers to discover complicated biological relationships, find possible therapeutic targets, and better understand disease mechanisms. Unlike SQL databases, which struggle to model and query biological interactions well, Neo4j offers a more user-friendly and efficient platform for genomic and bioinformatics analysis.

6. Neo4j's graph database is ideal for **highly interconnected data models**, including social networks, recommendation systems, and fraud detection applications. Neo4j's inherent support for graph topologies and interactions enables the efficient traversal and analysis of complicated networks. In contrast, **MongoDB's document-oriented model is better suited to semi-structured** or hierarchical data where relationships are less important. Attempting to represent and query heavily interconnected data in MongoDB may result in more difficult and inefficient queries than Neo4j.

7. Neo4j has tremendous features for **complicated queries, pattern matching, and graph analytics**, which MongoDB may not be able to efficiently provide. Neo4j's query language, Cypher, is specifically built for traversing and querying network structures, making it ideal for tasks like pathfinding, community detection, and centrality analysis. In comparison, while MongoDB offers basic querying and aggregating operations, running complicated graph-related queries frequently necessitates additional application logic and can be less performant than Neo4j.

Cypher - Graph Query Language

Cypher is a declarative query language for extracting patterns from graph databases.



Users define their query by specifying **nodes and edges**. The database management system then calculates all patterns meeting the criteria by analyzing the possible paths (connections between nodes via edges). The user declares the structure of the desired pattern, and the database management system's algorithms traverse all necessary connections (paths) and assemble the results

- *MATCH: Specification of nodes and edges, as well as declaration of search patterns*
- *WHERE: Conditions for filtering results*
- *RETURN: Specification of the desired search result, aggregated if necessary*

the Cypher query for the year the movie "The Matrix" was released would be:

```
MATCH (m: Movie {Title: "The Matrix"})
RETURN m.Year
```

***MATCH**: Specifies the pattern to match in the graph database.*
*(**m:Movie {Title: "The Matrix"}**): Defines a node pattern with the label "Movie" and properties matching the condition **{Title: "The Matrix"}**. This pattern represents finding a node labeled "Movie" with the title property equal to "The Matrix". The variable m is assigned to represent nodes matching this pattern.*
***RETURN m.Year**: Specifies that the query should return the "Year" property of the nodes matched by the pattern. The variable m is used to refer to these nodes.*

```
MATCH (m: Movie {Title: "The Matrix"})
RETURN m.Year
```

The query sends out the variable m for the movie “The Matrix” to return the movie’s year of release by m.Year. In Cypher, parentheses always indicate nodes, i.e., (m: Movie) declares the control variable m for the MOVIE node. In addition to control variables, individual attribute-value pairs can be included in curly brackets. Since we are specifically interested in the movie “The Matrix,” we can add {Title: “The Matrix”} to the node (m: Movie)

Queries regarding the relationships within the graph database are a bit more complicated. Relationships between two arbitrary nodes (a) and (b) are expressed in Cypher by the arrow symbol “->,” i.e., the path from (a) to (b) is declared as “(a) - > (b).”

If the specific relationship between (a) and (b) is of importance, the edge [r] can be inserted in the middle of the arrow.
The square brackets represent edges, and r is our variable for relationships.
Now, if we want to find out who played Neo in “The Matrix,” we use the following query to analyze the ACTED_IN path between ACTOR and MOVIE

```
MATCH (a: Actor) –[: Acted_In {Role: "Neo"}] –> (: Movie {Title: "The Matrix"}])
RETURN a.Name
```

For a list of movie titles (m), actor names (a), and respective roles (r), the query would have to be

```
MATCH (a: Actor) –[r: Acted_In] –> (m: Movie)
RETURN m.Title, a.Name, r.Role
```

Since our example graph database only contains one actor and two movies, the result would be the movie “Man of Tai Chi” with actor Keanu Reeves in the role of Donaka Mark and the movie “The Matrix” with Keanu Reeves as Neo.
In real life, however, such a graph database of actors, movies, and roles has countless entries. A manageable query would therefore have to remain limited, e.g., to actor Keanu Reeves, and would then look like this:

```
MATCH (a: Actor) –[r: Acted_In] –> (m: Movie)
WHERE (a.Name = "Keanu Reeves")
RETURN m.Title, a.Name, r.Role
```

Similar to SQL, Cypher uses declarative queries where the user specifies the desired properties of the result pattern (Cypher) or results table (SQL) and the respective database management system then calculates the results.
However, analyzing relationship networks, using recursive search strategies, or analyzing graph properties is hardly possible with SQL.

Graph databases are even more relationship-oriented than relational databases. Both nodes and edges of the graph are independent data sets. This allows efficient traversal of the graph for network-like information. However, there are applications that focus on structured objects as a unit. **Document databases** are suitable for this purpose.

Neo4J- Syntax

Exam type question:
Detail the Cypher queries (neo4j) needed to store details of cars and the country of origin. Each car has a name and each country has a name.

Solution:

Each car node will have a property for the car's name.

```
CREATE (:Car {name: "Toyota Camry"}),
(:Car {name: "Honda Civic"}),
(:Car {name: "Ford Mustang"}),
(:Car {name: "BMW 3 Series"})
```

CREATE: This keyword is used to create nodes in Neo4j.
(:Car {name: "Toyota Camry"}): This syntax creates a node labeled "Car" with a property "name" set to "Toyota Camry". Similarly, nodes for other car models are created in the same manner.

Create nodes to represent the countries of origin for the cars. Each country node will have a property for the country's name.

```
CREATE (:Country {name: "Japan"}),
(:Country {name: "USA"}),
(:Country {name: "Germany"})
```

Neo4J- Syntax

Exam type question:

Detail the Cypher queries (neo4j) needed to store details of cars and the country of origin. Each car has a name and each country has a name.

Solution cont....:

establish relationships between the car nodes and the country nodes to indicate the country of origin for each car.

```
MATCH (car:Car),(country:Country)
WHERE car.name = "Toyota Camry" AND country.name = "Japan"
CREATE (car)-[:MADE_IN]->(country)
MATCH (car:Car),(country:Country)
WHERE car.name = "Honda Civic" AND country.name = "Japan"
CREATE (car)-[:MADE_IN]->(country)
MATCH (car:Car),(country:Country)
WHERE car.name = "Ford Mustang" AND country.name = "USA"
CREATE (car)-[:MADE_IN]->(country)
MATCH (car:Car),(country:Country)
WHERE car.name = "BMW 3 Series" AND country.name = "Germany"
CREATE (car)-[:MADE_IN]->(country)
```

```
MATCH (car:Car),(country:Country)
WHERE car.name = "BMW 3 Series" AND country.name = "Germany"
CREATE (car)-[:MADE_IN]->(country)
```

MATCH: This keyword is used to find nodes that match specific criteria.
(car:Car), (country:Country): These clauses define new **variables car and country** and specify that they **represent nodes labeled as "Car" and "Country",** respectively.
WHERE: This clause filters nodes based on specified conditions.
CREATE: This keyword is used to create relationships between nodes.
(:MADE_IN): This syntax defines a relationship type **labeled** "MADE_IN".
(:Car)-[:MADE_IN]->(:Country): This pattern represents a relationship between a car node and a country node, indicating that the car is made in that country. We specify the car and country names in the **WHERE clause** to match the specific car-country pairs.
Now query the data using the return clause to verify that cars are associated with their respective countries.

```
MATCH (car:Car)-[:MADE_IN]->(country:Country)
RETURN car.name, country.name
```

MATCH: This keyword is used to find patterns in the graph.
(car:Car)-[:MADE_IN]->(country:Country): This pattern matches car nodes connected to country nodes via the "MADE_IN" relationship.
RETURN: This keyword is used to specify what data to return from the query.
car.name, country.name: These expressions specify that we want to return the **names of cars and their respective countries**.
Executing this query will return a list of car names along with their corresponding countries of origin.

Another **EXAM type** example where we store details of movies and their genres. Each movie will have a title, and each genre will have a name

Take 10 min to write down the code and the query to present the results

Solution

```
//Create movie Node
CREATE (:Movie {title: "Iron Man"}),
(:Movie {title: "The Avengers"}),
(:Movie {title: "Guardians of the Galaxy"}),
(:Movie {title: "Black Panther"})

//Create genre node
CREATE (:Genre {name: "Action"}),
(:Genre {name: "Adventure"}),
(:Genre {name: "Sci-Fi"}),
(:Genre {name: "Superhero"})

//Create relationships
MATCH (movie:Movie),(genre:Genre)
WHERE movie.title = "Iron Man" AND genre.name = "Action"
CREATE (movie)-[:BELONGS_TO]->(genre)

MATCH (movie:Movie),(genre:Genre)
WHERE movie.title = "The Avengers" AND genre.name IN ["Action", "Adventure", "Sci-Fi", "Superhero"]
CREATE (movie)-[:BELONGS_TO]->(genre)

MATCH (movie:Movie),(genre:Genre)
WHERE movie.title = "Guardians of the Galaxy" AND genre.name IN ["Action", "Adventure", "Sci-Fi", "Superhero"]
CREATE (movie)-[:BELONGS_TO]->(genre)

MATCH (movie:Movie),(genre:Genre)
WHERE movie.title = "Black Panther" AND genre.name IN ["Action", "Adventure", "Sci-Fi", "Superhero"]
CREATE (movie)-[:BELONGS_TO]->(genre)
```

Solution cont....

```
// get the movies and associated genres
// is used to comment code, similar to MongoDB
MATCH (movie:Movie)-[:BELONGS_TO]->(genre:Genre)
RETURN movie.title, genre.name
```

Homework

Let's consider an example where we store details of students, courses, and their enrollments.