Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

CSEN 903: Advanced Computer Lab, Winter 2025
Lab 6 Manual: Knowledge Graphs

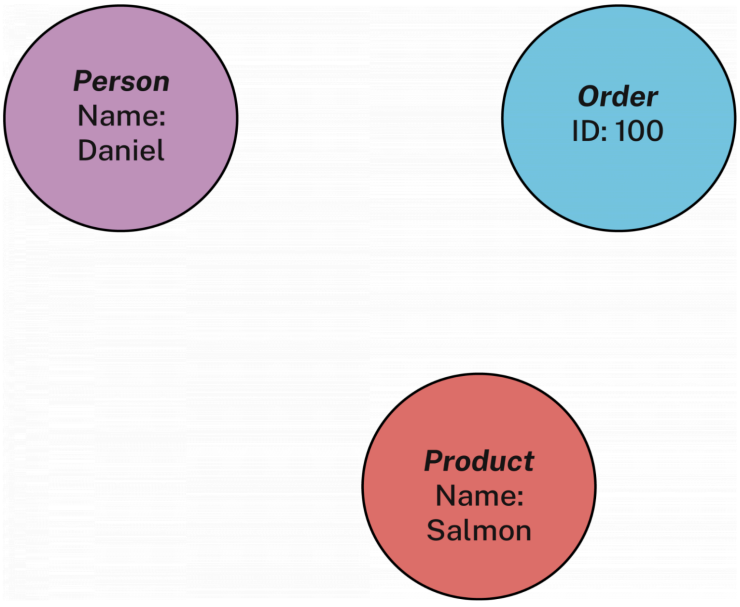
Part 1: Read and Watch Before the Lab

A **knowledge graph (KG)** is a structured way of representing knowledge using **entities** (nodes) and **relationships** (edges). Unlike tabular data, KGs capture **semantic meaning**, making them ideal for reasoning, recommendation, and search systems.

In a KG, entities can represent things, occasions, circumstances, or ideas. The context and significance of these entities' connections are captured by their relationships.

What makes a KG helpful is the way it organizes the principles, data, and relationships to surface new knowledge for your user or business. The design is useful for many use cases, including real-time applications, search problems, and grounding generative AI for question-answering.

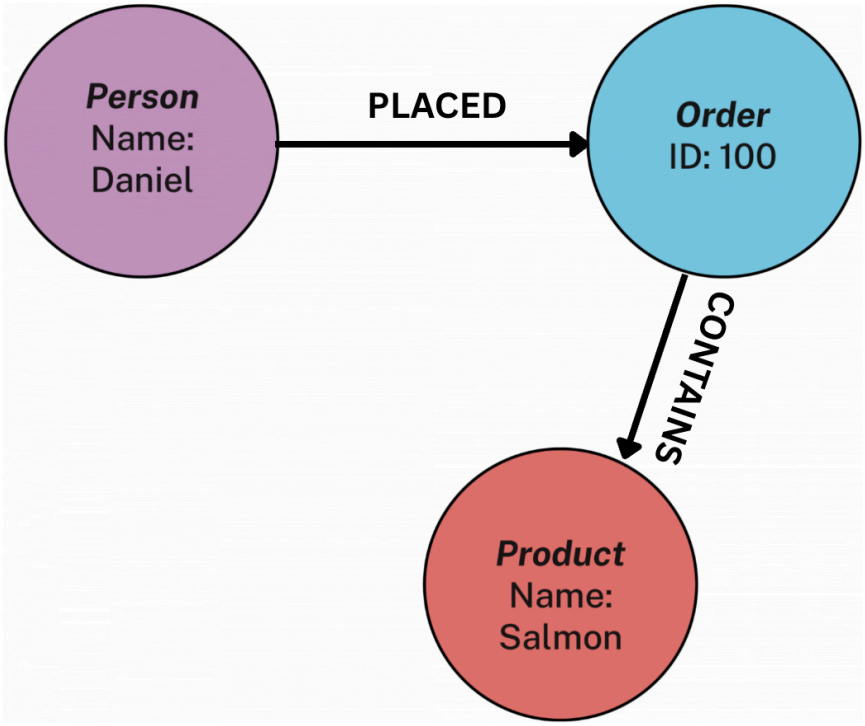
Nodes:



Nodes represent and store details about entities, such as people, places, objects, or institutions. Each node has a (or sometimes several) labels to identify the node type and may optionally have one or more properties (attributes) to add more details about the entity. Nodes are also referred to as *vertices*.

In the figure above, the nodes in an e-commerce knowledge graph represent entities such as people (customers and prospects) through the Person Node, products through the Product node, and orders through the Order node.

Relationships:



Relationships are used to link two nodes together: they show how the entities are related with respect to each other. Same as nodes, each relationship has a label identifying the relationship type and may optionally have one or more properties to add more details about the relationship. Relationships can also be referred to as *edges*.

In the same e-commerce example, a relationship PLACED exists between the customer and order nodes, to demonstrate the “placed order” relationship between customers and their orders. Similarly the CONTAINS relationship shows the link between the Order

and the Product. One would say now we can read this simple graph very smoothly: A Person named Daniel placed the 100th order which contains the Salmon product.

Why Graph Databases matter?

To store and query knowledge graphs efficiently, specialized databases such as property graph databases (like **Neo4j**) are ideal. Neo4j is commonly used to model richer relationships and it is more intuitive for most enterprise scenarios.
They represent data exactly as it appears conceptually: nodes, relationships, and properties, which allows for:

- **Easy modeling:** Natural translation from real-world data to graph form.
- **Flexibility:** Add new entities or relations without heavy restructuring.
- **Performance:** Fast traversal across deeply connected data.

Cypher

Cypher is the **query language** used in Neo4j to interact with knowledge graphs. It was designed to be intuitive and visually close to the way graphs are drawn, making it easier to express complex patterns of connected data. Instead of using joins like in SQL, Cypher uses syntax with parentheses for nodes and arrows for relationships, mirroring the way we naturally think about connections. With Cypher, you can easily create, query, and update data in a graph by describing what pattern you want to match, rather than how to traverse it. This makes it ideal for exploring relationships such as customer behavior, product dependencies, or supplier networks in a more human-readable way.

Let's say we have customers, orders, and products connected in the graph as follows:

```
(Customer)-[:PLACED]->(Order)-[:CONTAINS]->(Product)
```

To find **all the products purchased by a specific customer**, say *"Sara"*, you could write:

```
MATCH (c:Customer {name:
"Sara"})-[:PLACED]->(o:Order)-[:CONTAINS]->(p:Product)
RETURN p.name AS Product, p.category AS Category, p.price AS Price;
```

Knowledge Graphs Use Cases:

You can consider using knowledge graphs when structuring the data and caring about the relationships between entities is crucial. Here are some examples in the industry where knowledge graphs are commonly considered:

- **Generative AI and Enterprise Search**
  Used to ground large language models with factual, domain-specific data to reduce hallucination and improve explainability.
- **Fraud Detection and Risk Analysis**
  Graphs reveal suspicious transaction patterns, hidden links between accounts, or shared entities across networks.
- **Customer 360 and Master Data Management**
  Combine customer data from multiple systems to form a unified view of interactions and relationships.
- **Supply Chain Management**
  Map suppliers, logistics, and materials to improve visibility and resilience in operations.
- **Healthcare and Research**
  Connect genes, drugs, diseases, and trials to accelerate discovery and understanding.

In this lab, you'll:

1. Understand the structure of a graph database.
2. Create nodes and relationships manually using **Neo4j.**
3. Query the graph using **Cypher**, Neo4j's query language.
4. Apply these concepts to a mini domain (e.g., movies, people, or diseases)

Required Readings:

1. What is a Knowledge Graph?
2. Intro to Graph Databases (Neo4j Docs)
3. Cypher Query Language Basics

Required Watch List:

1. Knowledge Graphs and Neo4j
2. Neo4j in 100 Seconds
3. Neo4j (Graph Database) Crash Course

Part 2: In-Lab Task

1. Visit https://neo4j.com/cloud/aura/
2. Click **"Start Free"** and sign up or log in
3. Create a **free database**:
    - Click **"Create Instance"**
    - Select **"AuraDB Free"** option
    - By default, the database name is set to **Instance01** in the online free version.
    - Credentials for the new instance are autogenerated. Make sure to **Download the text file** as **the password will not be available after this point.**
    - The creation of the instance will take a few minutes. **It will be ready once a running** sign is displayed.
4. Click **"Connect"** once the database is ready and launch the **Query Editor**
5. Copy and paste this Cypher block in the Query Editor:

```
CREATE
  (TheMatrix:Movie {title:'The Matrix', released:1999}),
  (Keanu:Person {name:'Keanu Reeves'}),
  (Carrie:Person {name:'Carrie-Anne Moss'}),
  (Laurence:Person {name:'Laurence Fishburne'}),
  (SciFi:Genre {name:'Science Fiction'}),

  (Keanu)-[:ACTED_IN]->(TheMatrix),
```

```
(Carrie)-[:ACTED_IN]->(TheMatrix),
(Laurence)-[:ACTED_IN]->(TheMatrix),
(TheMatrix)-[:HAS_GENRE]->(SciFi)
```

You should receive a success message stating: Created 5 nodes, created 4 relationships, set 6 properties, added 5 labels.

6. Try these queries in the editor:
   - Find all people who acted in movies:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, m.title
```

   - Find movies in a genre:

```
MATCH (m:Movie)-[:HAS_GENRE]->(g:Genre)
RETURN m.title, g.name
```

   - **List all nodes and relationships:**

```
MATCH (n)-[r]->(m)
  RETURN n, r, m
```

7. Extend the graph by adding another movie and actor (who acted in both films). Next, add a director node and add a :DIRECTED relationship.

```
CREATE (Reloaded:Movie {title:'Matrix Reloaded', released:2003}),
       (Lana:Director {name:'Lana Wachowski'}),
       (Lana)-[:DIRECTED]->(Reloaded),
       (Keanu)-[:ACTED_IN]->(Reloaded)
```

In the above snippet, we can see that the new Movie node with the relationships got created. However, we run into a cypher scoping issue, as in Neo4j, variables don't persist across different queries.

   - We can delete the bad created node for Keanu(replace elementID with yours displayed on the new unwanted node)

```
MATCH (n)
WHERE elementId(n) = '4:5b1c1915-fdf6-4a09-9fe3-c505fbb33dd7:7'
DETACH DELETE n;
```

- Link the existing actor to the new movie

```
MATCH (keanu:Person {name:'Keanu Reeves'})
MATCH (m:Movie {title:'Matrix Reloaded'})
MERGE (keanu)-[:ACTED_IN]->(m);
```

At any point, to see the changes you made you can rerun the cell that lists all nodes and relationships (Check step 6 again if you can't find the query)

8. Try the following queries on the extended KG.
   - Who co-acted with Keanu Reeves?

```
MATCH (keanu:Person {name:'Keanu
Reeves'})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coactor)
RETURN DISTINCT coactor.name
```

   - Shortest path between two actors:

```
MATCH p = shortestPath((a:Person {name: 'Keanu
Reeves'})-[*]-(b:Person {name: 'Laurence Fishburne'}))
RETURN p
```

Part 3: Graded Task

Task Description

Note: AuraDB free version only allows the creation of 1 instance DB. Hence before you start the task, make sure to delete the one you created and recreate a new instance for this task.

You will build a small **Books Knowledge Graph** using **Neo4j Aura Free**. This graph will include **authors**, **books**, **genres**, and **publishers**, and the relationships among them. You will then write Cypher queries to retrieve information from the graph.

1. You must create the following **nodes**, each with the specified **labels and properties**:
    a. Create the following `Author` nodes: J.K. Rowling, George Orwell, Yuval Noah Harari.
    b. Create the following `Book` nodes:*Harry Potter and the Philosopher's Stone* (Year: 1997), *Harry Potter and the Chamber of Secrets* (Year: 1998), *1984* (Year: 1949), *Sapiens* (Year: 2011).
    c. Create the following `Genre` nodes: Fantasy, Science Fiction, Non-Fiction.
    d. Create the following `Publisher` nodes:Bloomsbury, Penguin Books, Harvill Secker.
    e. Use the following relationships:
        - J.K. Rowling `WROTE` *Harry Potter and the Philosopher's Stone* and *Harry Potter and the Chamber of Secrets.*
        - George Orwell `WROTE` *1984.*
        - Yuval Noah Harari `WROTE` *Sapiens.*
        - *Harry Potter and the Philosopher's Stone* `HAS_GENRE` → Fantasy
        - *Harry Potter and the Chamber of Secrets* `HAS_GENRE` → Fantasy
        - *1984* `HAS_GENRE` → Science Fiction
        - *Sapiens* `HAS_GENRE` → Non-Fiction
        - *Harry Potter and the Philosopher's Stone* `PUBLISHED_BY` → Bloomsbury
        - *Harry Potter and the Chamber of Secrets* `PUBLISHED_BY` → Bloomsbury
        - *1984* `PUBLISHED_BY` → Penguin Books
        - *Sapiens* `PUBLISHED_BY` → Harvill Secker

2. Write Cypher queries for the following.
    a. **List all books written by each author.**
    Return the author's name and the title of each book.

    b. **Find all books in the "Fantasy" genre.**
    Return only the titles of the books.

    c. **Find all authors who have books published by "Bloomsbury."**
    Return the author's name and the title of each book they published with Bloomsbury.

    d. **List all books published after the year 1950.**
    Return the title and publication year of each book.

    e. **Find the shortest path between J.K. Rowling and George Orwell.**
    Return the full path that connects them through any node types (books, genres, publishers, etc.).

Deliverables:

Submit an exported text file with all the completed queries above to the form. The submission **deadline** is by the end of your lab.