Introduction to Neo4j

and Cypher Query Language

Joseph Azar

CP58 - UTBM Sevenans

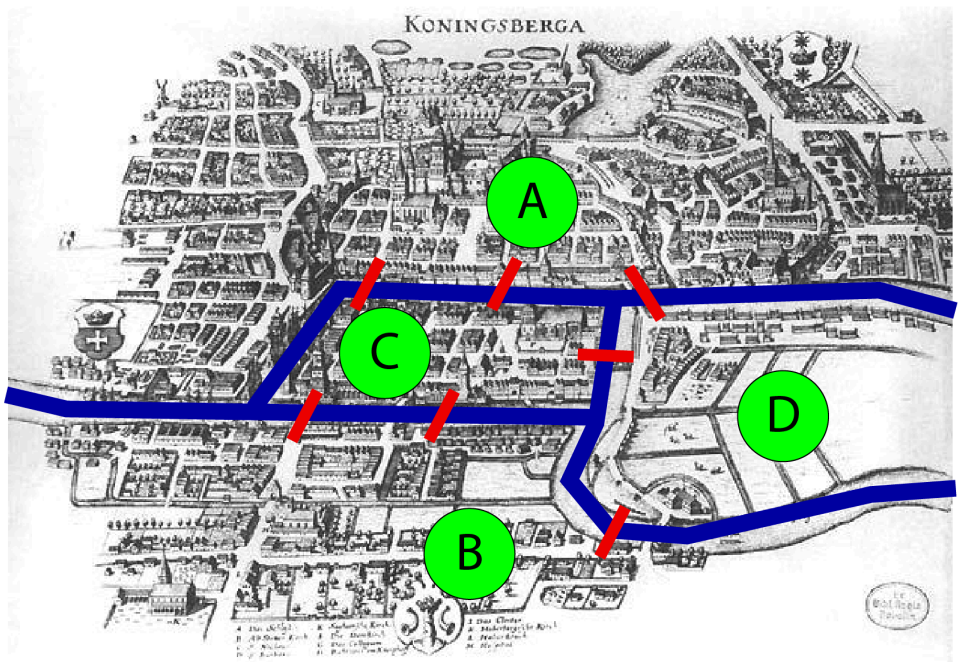
Quick Recap: Session 1

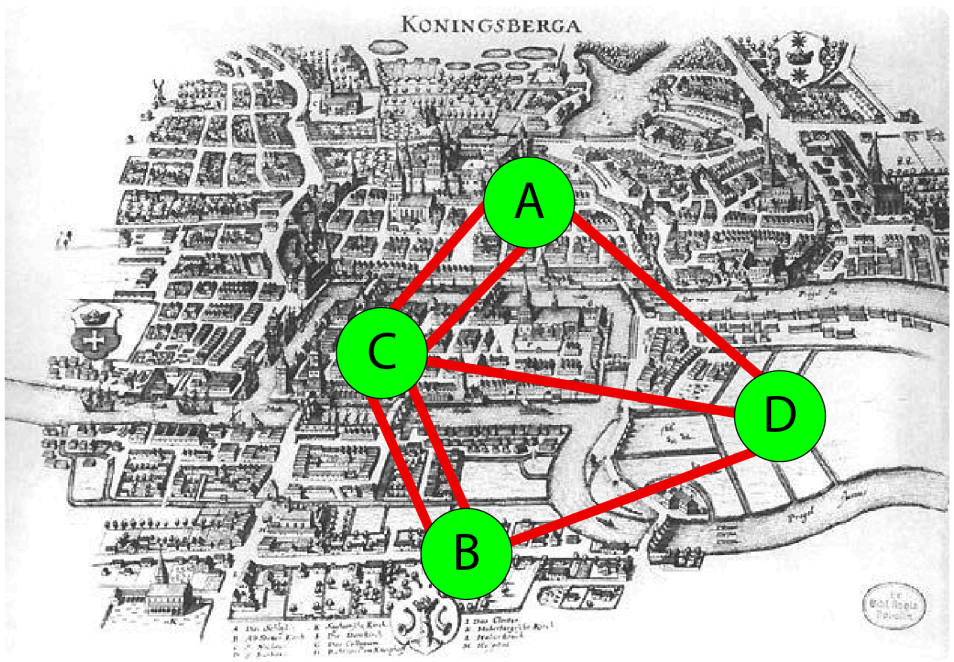
What we learned:

- Knowledge = Information + Context + Relationships
- Graphs = Nodes + Edges
- Graphs are perfect for modeling CONNECTIONS

Today: Learn how to build and query graphs with Neo4j!

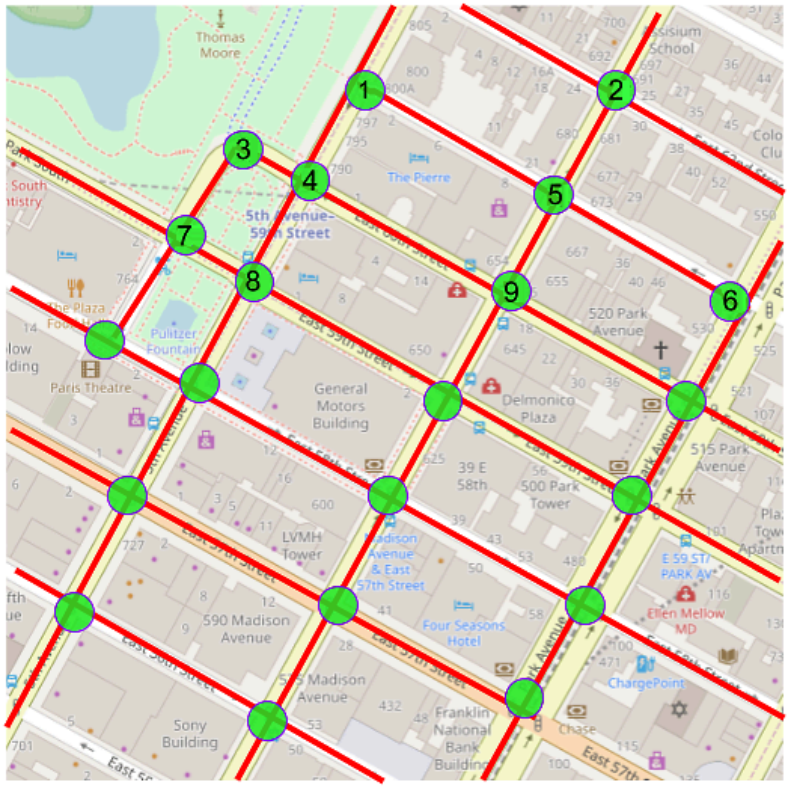
History: The Seven Bridges of Königsberg





Euler's Problem (1736): Can you cross all 7 bridges exactly once?

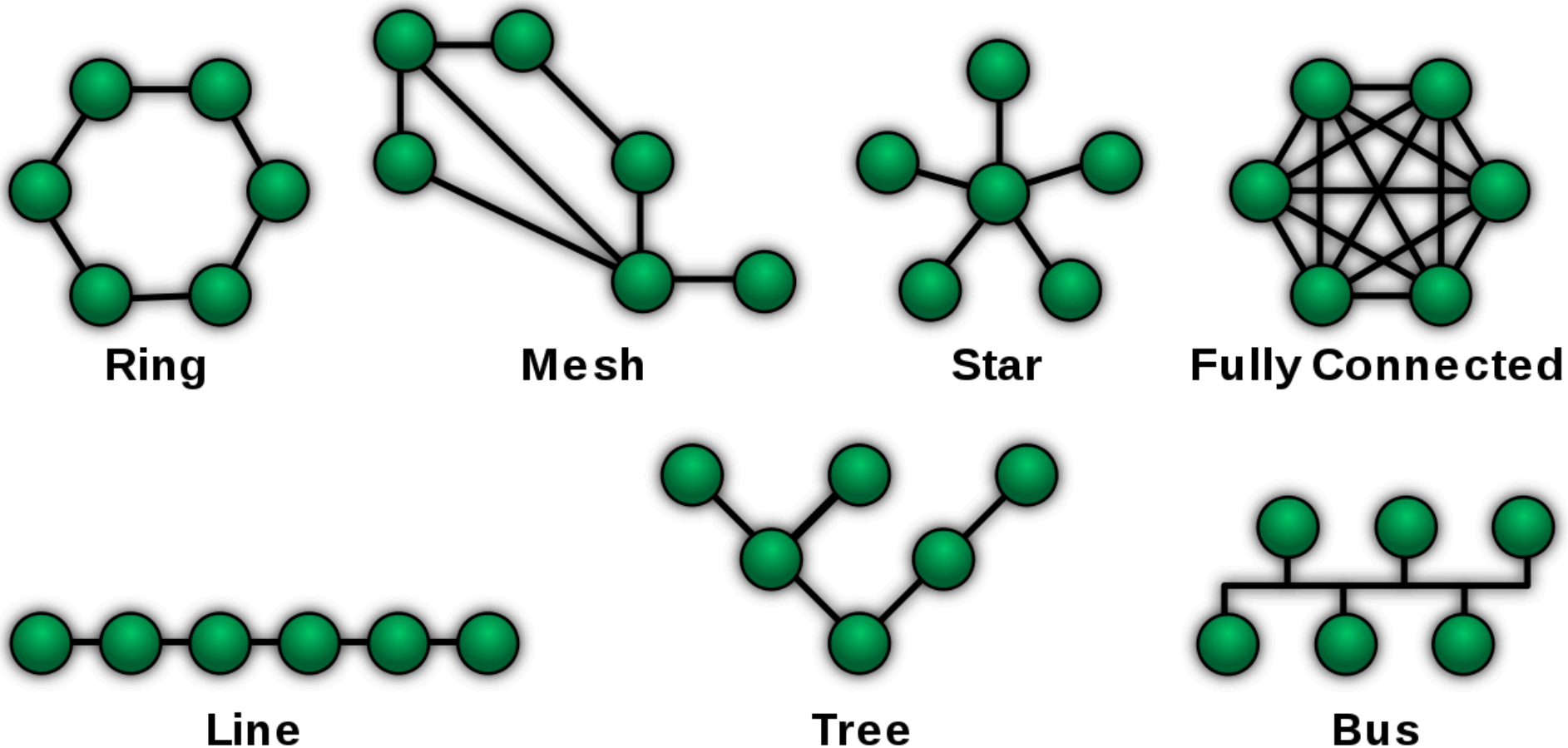
Real Example: Road Network



Nodes: Street intersections (numbered)

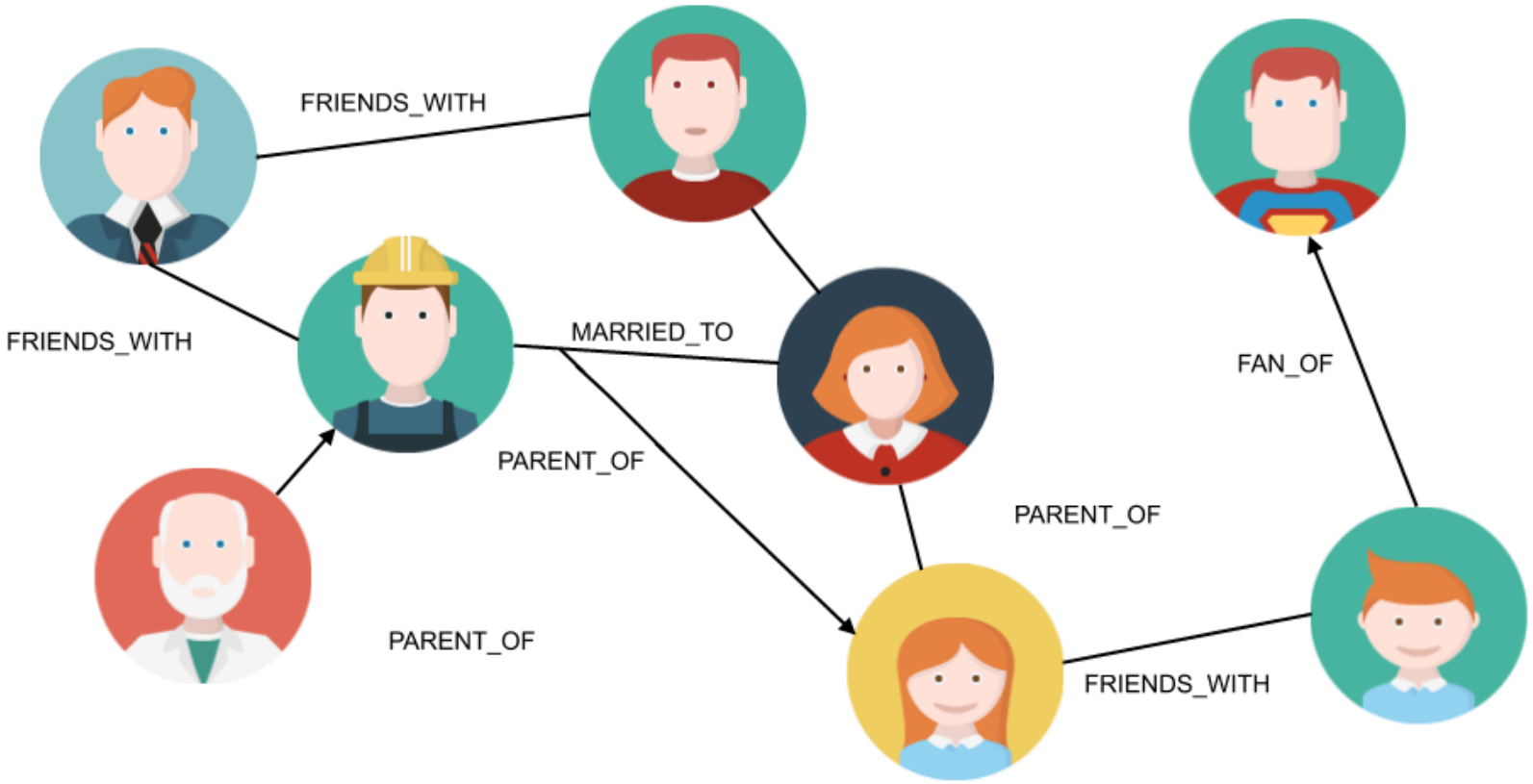
Edges: Streets connecting intersections

Common Graph Patterns



Different structures for different uses!

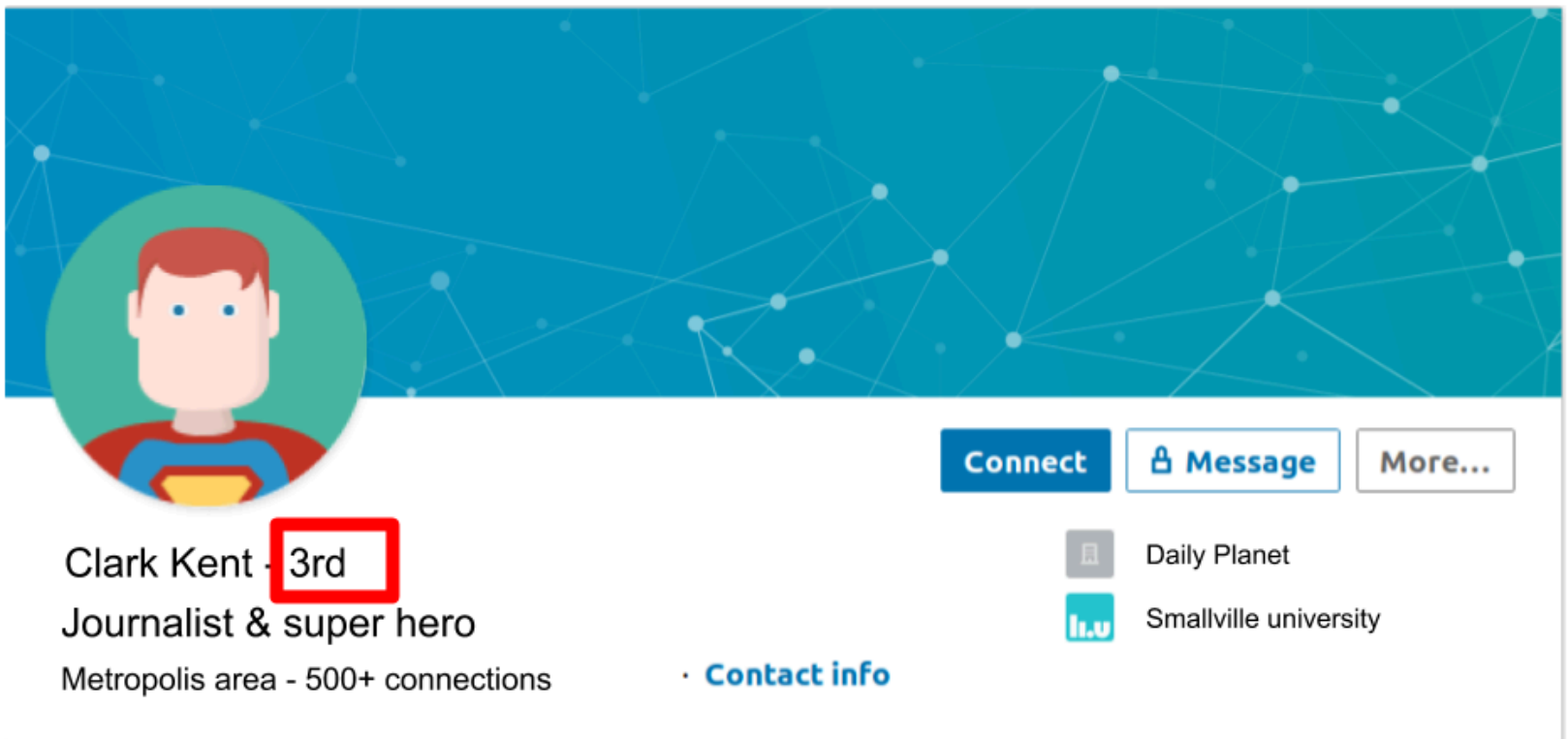
Social Network Example



Real-world relationships: FRIENDS_WITH, MARRIED_TO,

PARENT_OF, FAN_OF

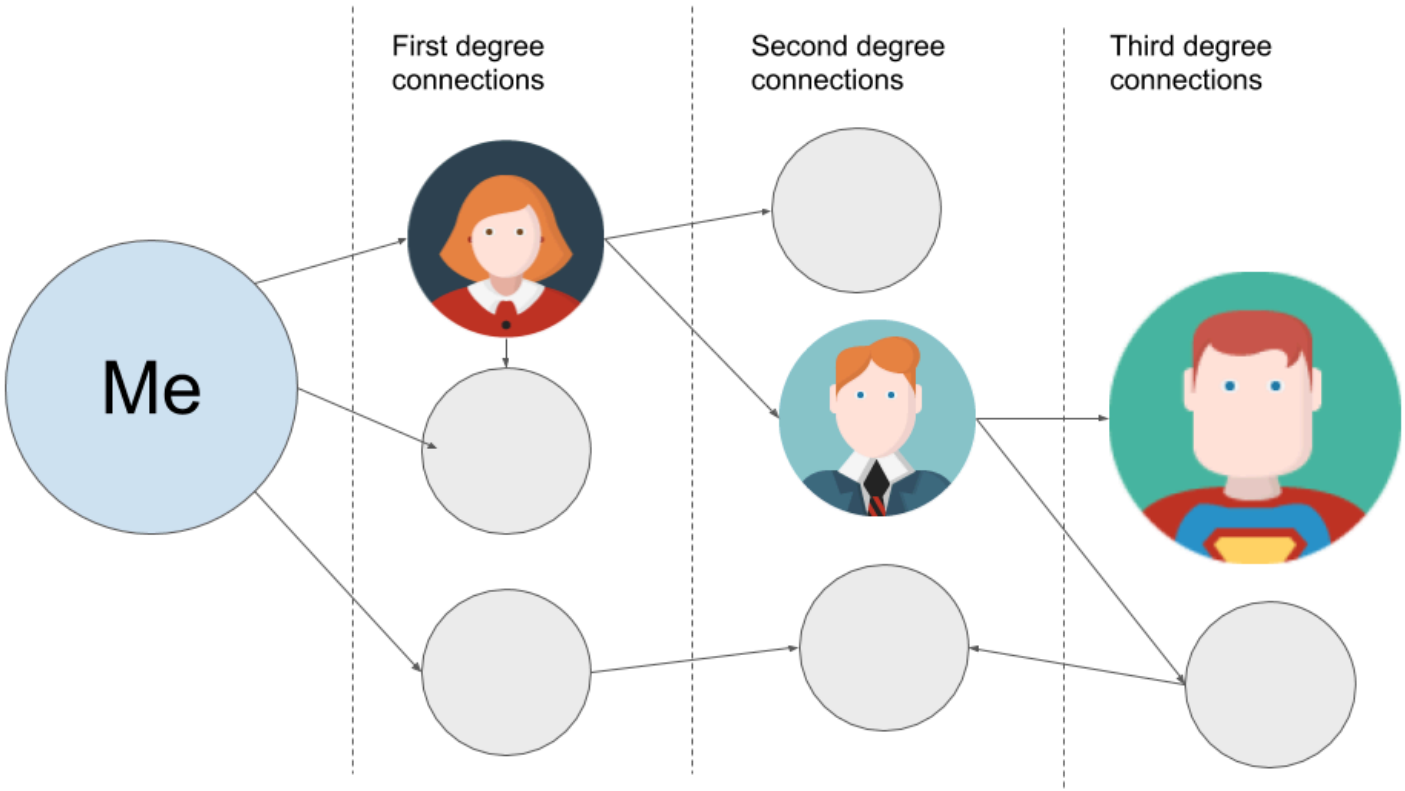
Degrees of Separation



Clark Kent is "3rd" degree connection

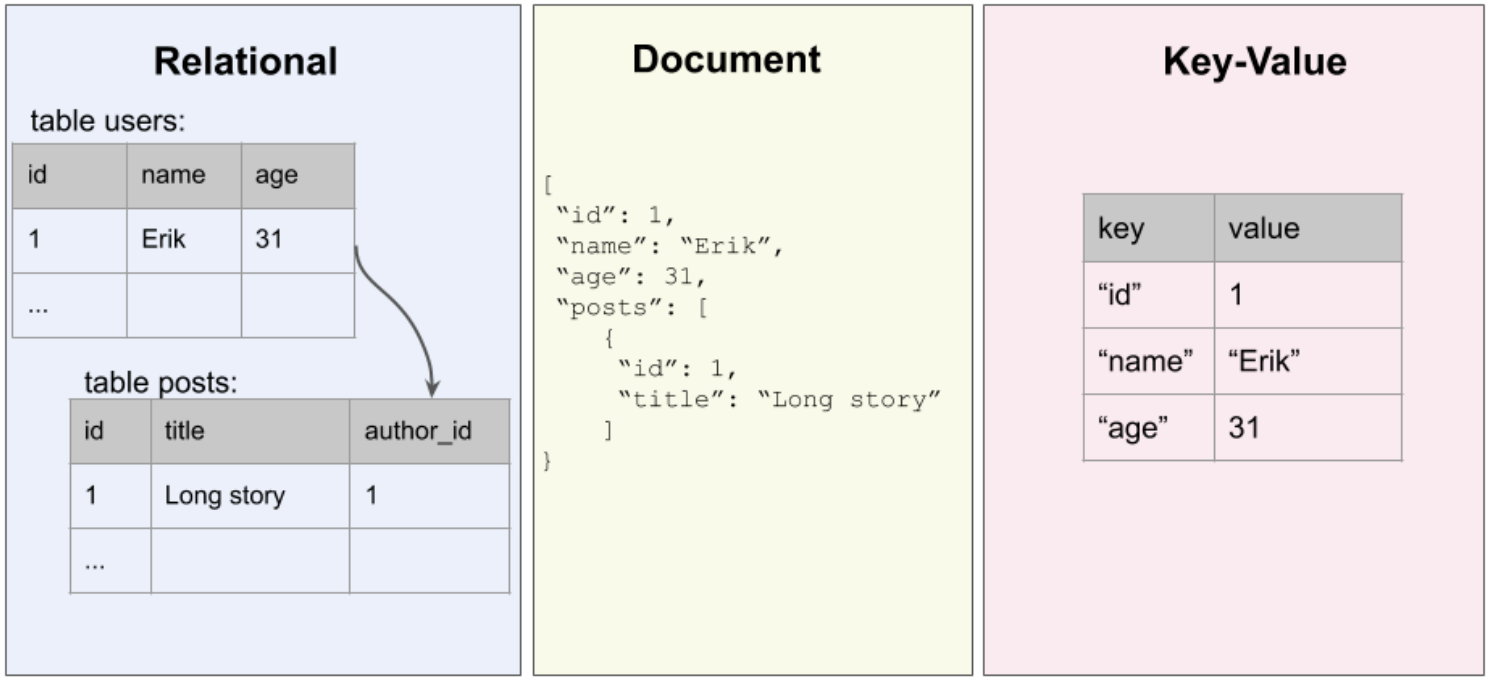
Me → Friend → Friend's Friend → Clark Kent

Visualizing Connection Degrees



Six Degrees of Separation: Any 2 people are connected

Different Database Models

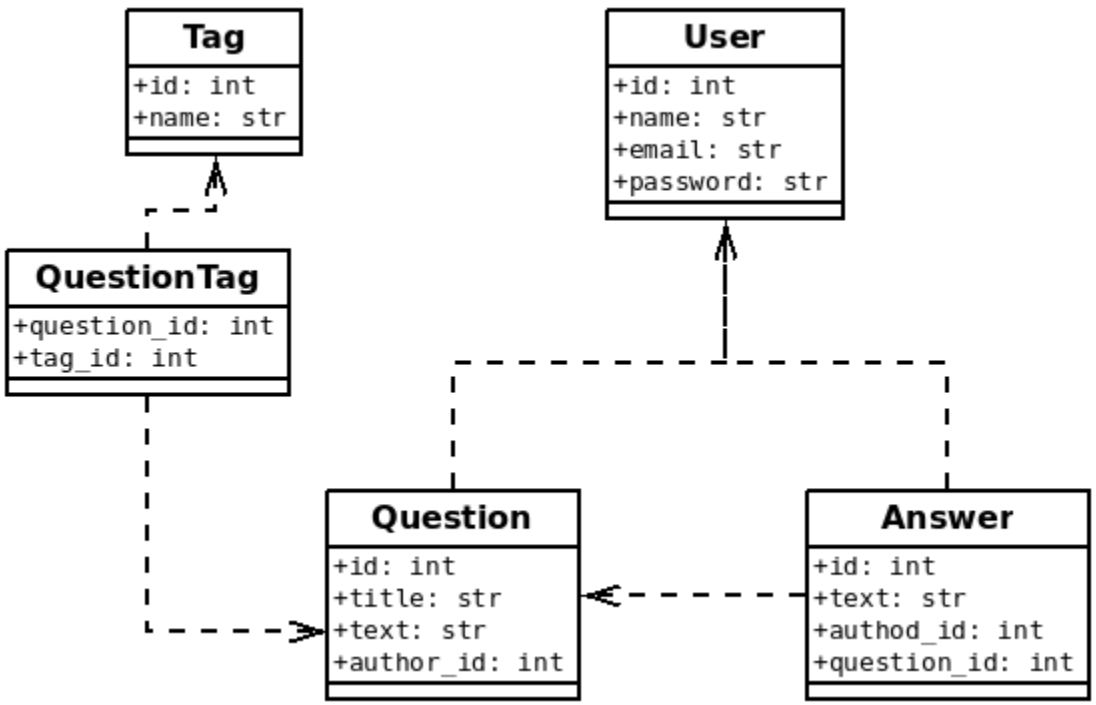


Relational: Tables with rows/columns

Document: JSON-like documents

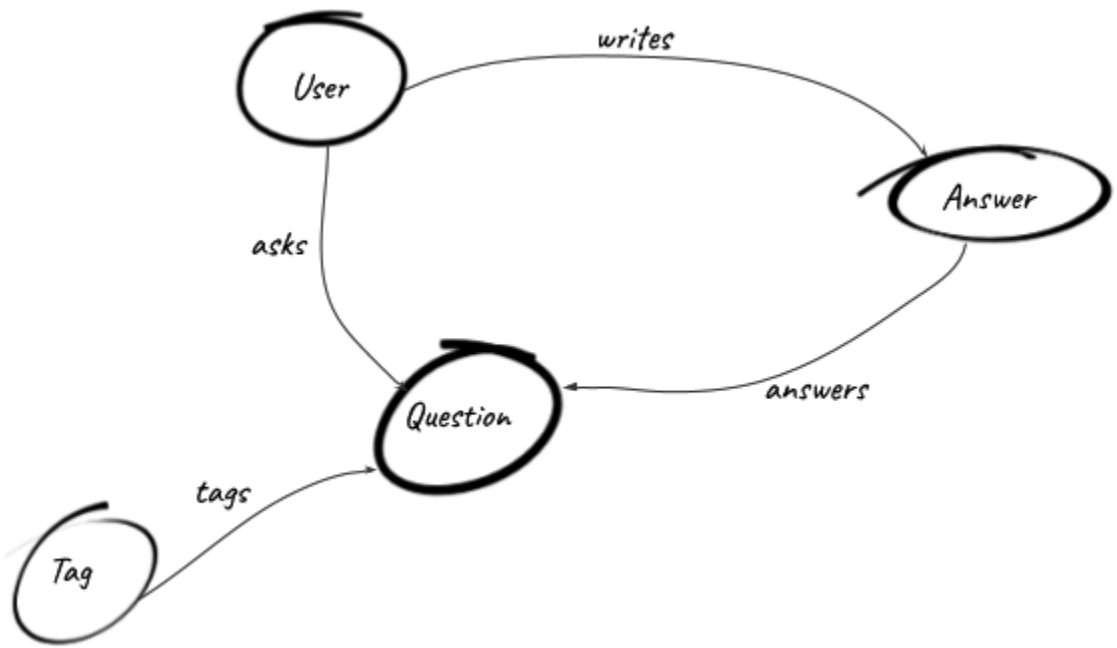
Key-Value: Simple key-value pairs

From Tables to Graphs



Traditional: Multiple tables with foreign keys

From Whiteboard to Graph



Your sketch IS your data model!

User → asks → Question → has → Answer

What is Neo4j?

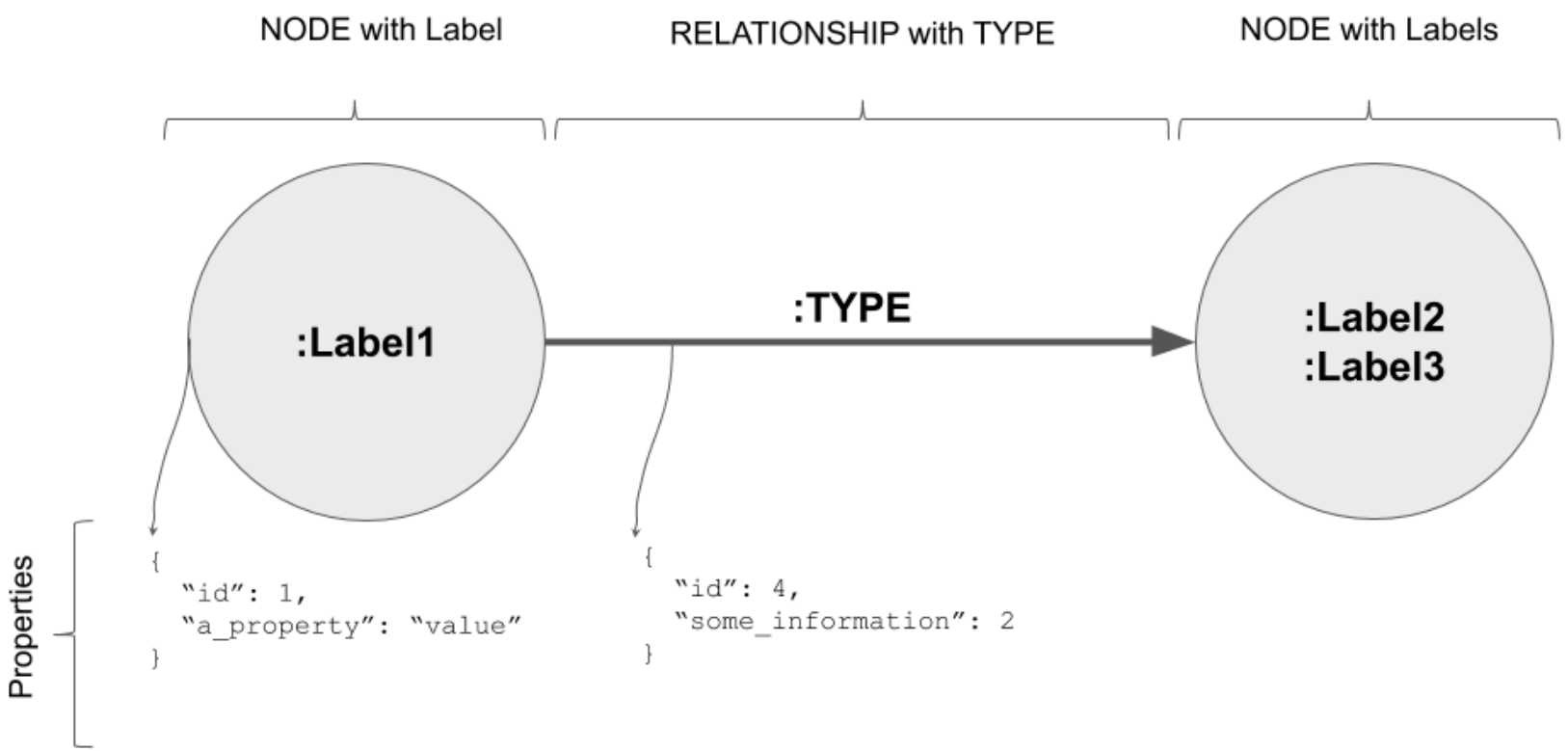
> **Neo4j = A Graph Database**

Just like:

- 📁 Excel stores data in **tables**
- 💾 Neo4j stores data in **graphs**

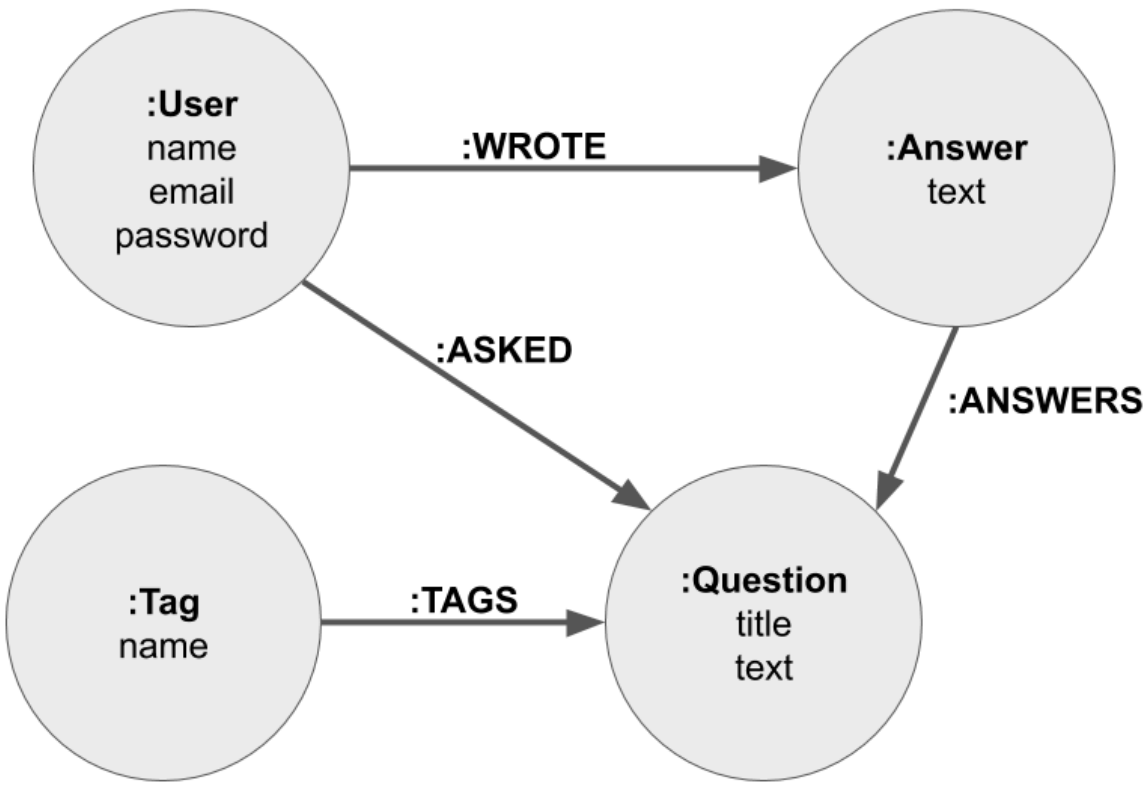
> **Why "Neo4j"?**
>
> From the movie "The Matrix" (1999) - Neo is the main character!

Neo4j Building Blocks



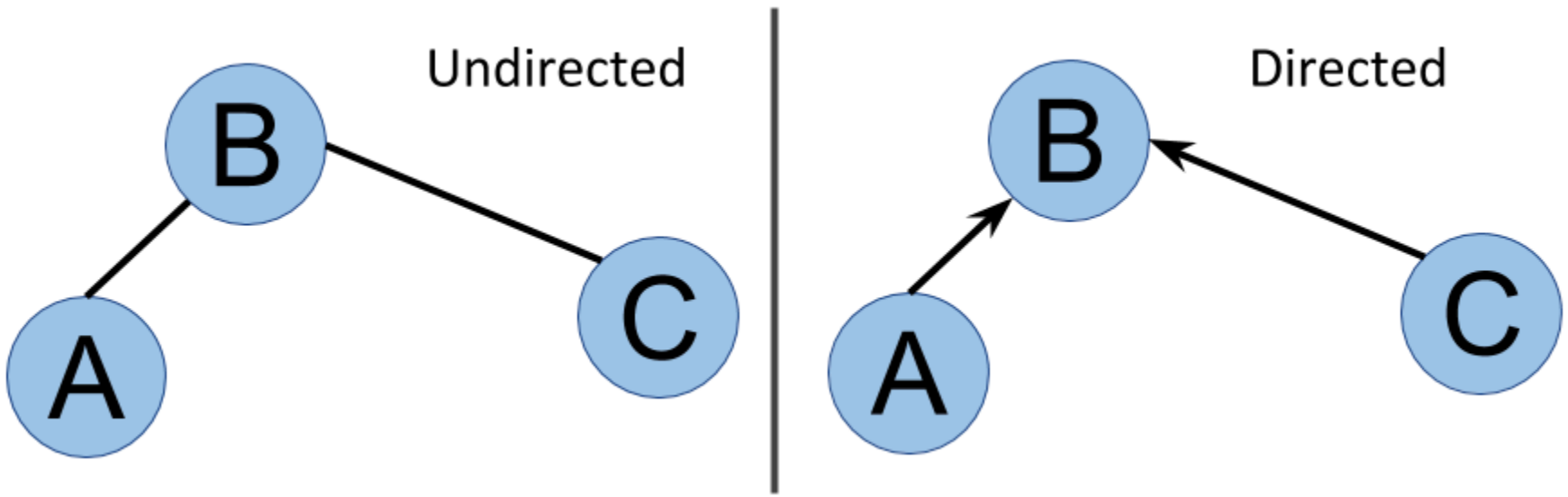
Nodes: Can have labels (types) and properties (data)

Complete Graph Model Example



Q&A System: Users, Questions, Answers, Tags

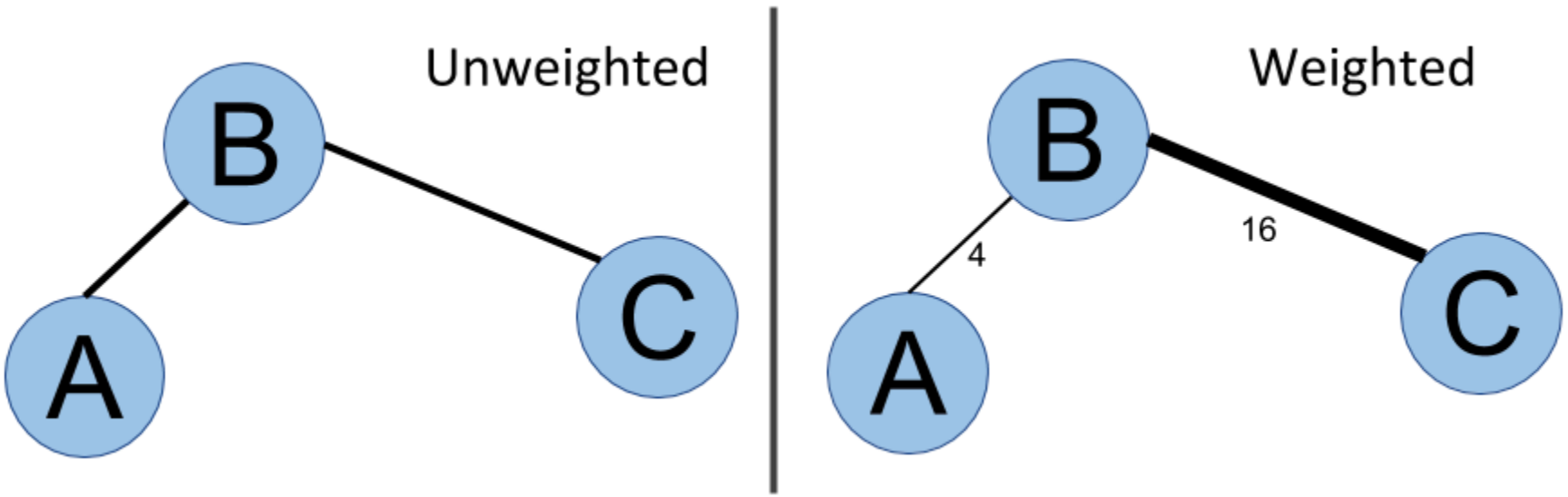
Directed vs Undirected



Neo4j: All relationships are directed!

But you can query them in any direction or ignore direction

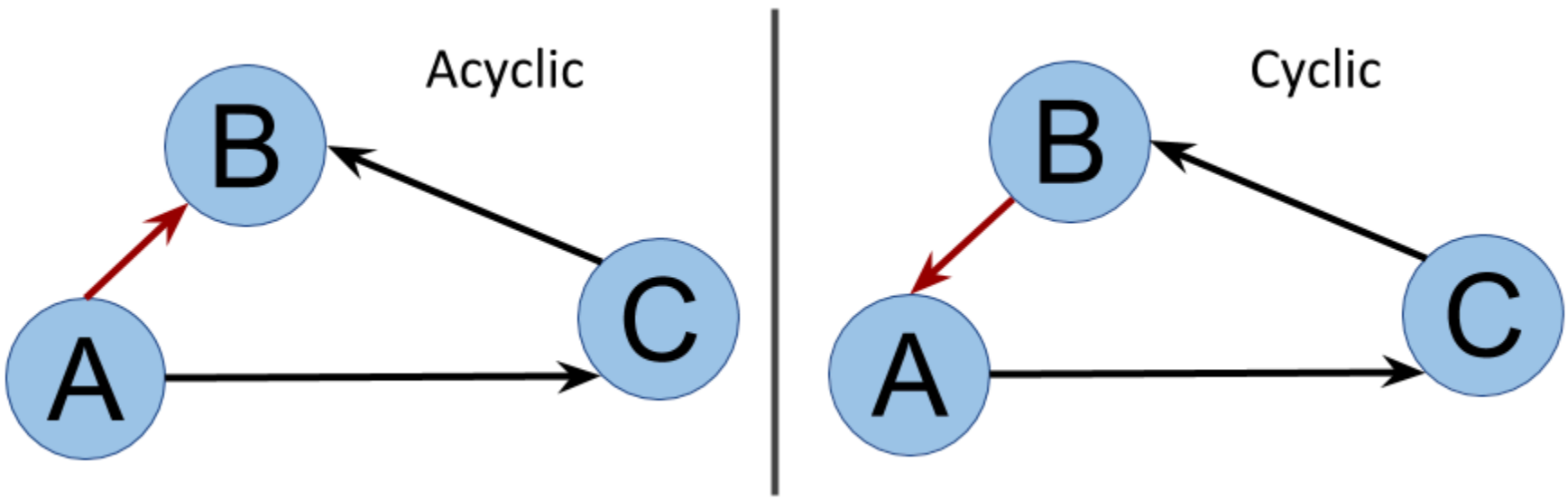
Weighted vs Unweighted



Weights: Distances, costs, times, strengths

Stored as relationship properties in Neo4j

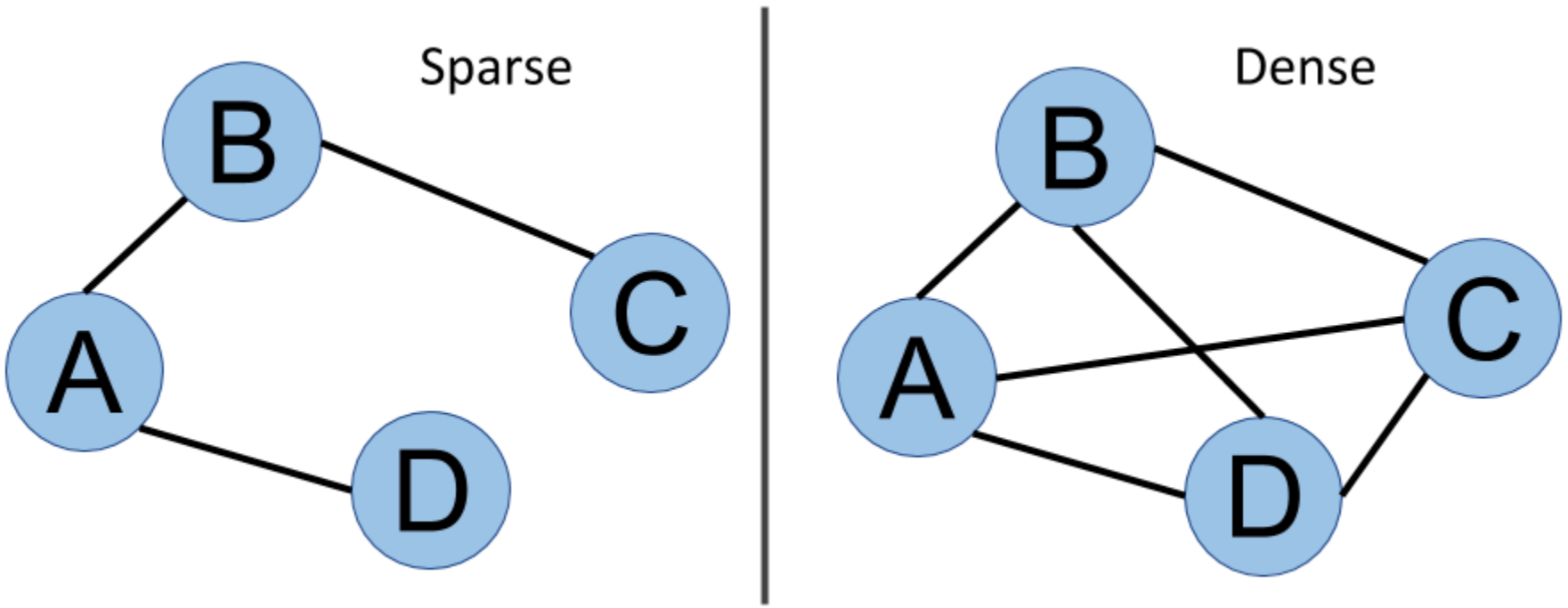
Cyclic vs Acyclic



Cycle: Path that returns to starting point (A→B→C→A)

Important to detect to avoid infinite loops!

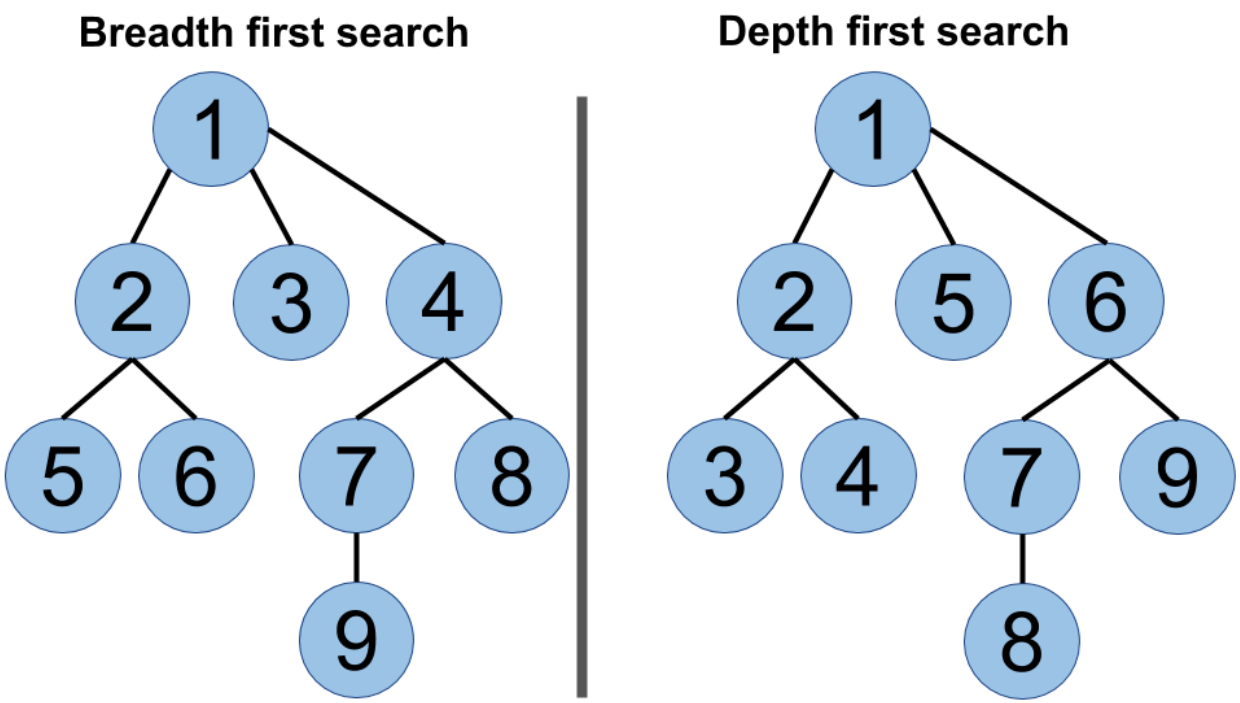
Sparse vs Dense Graphs



Density: How many connections vs how many possible?

Dense graphs take longer to traverse

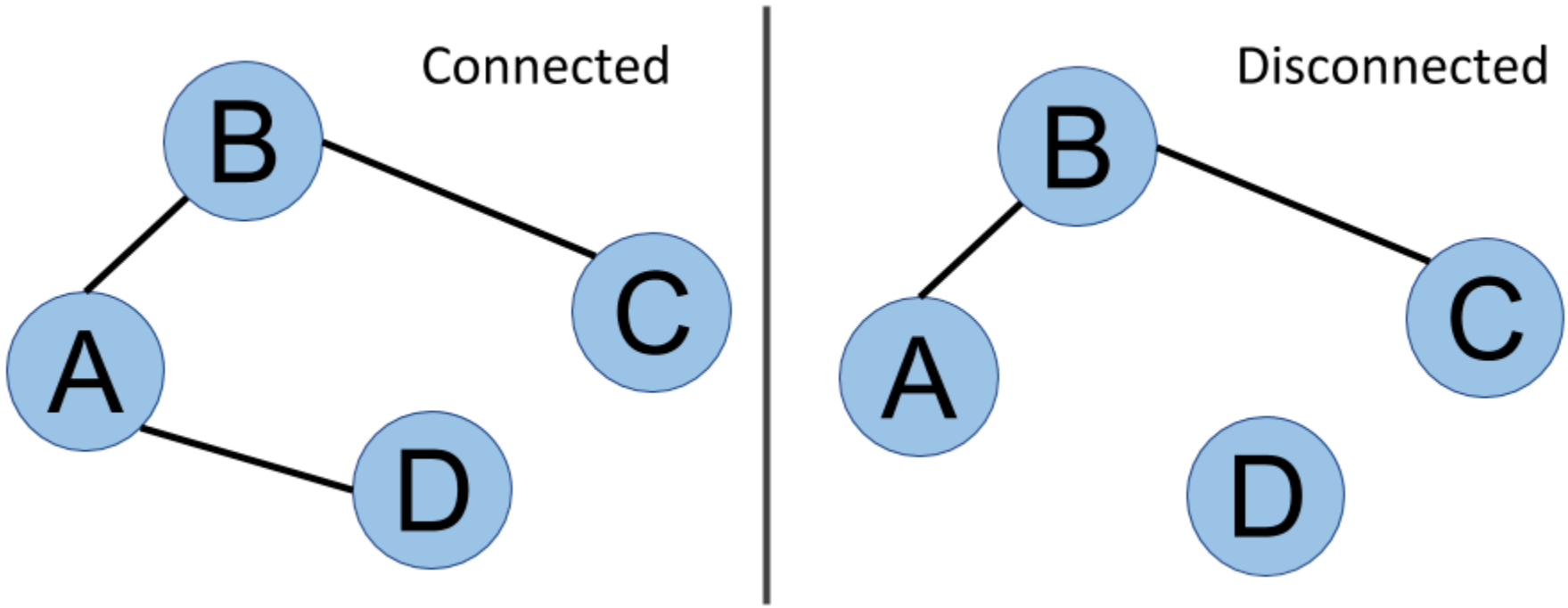
Graph Traversal: BFS vs DFS



Breadth-First: Visit all neighbors before going deeper

Depth-First: Go deep before exploring neighbors

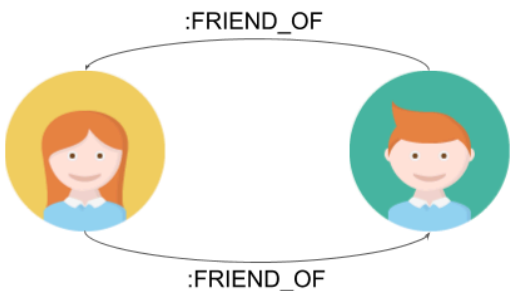
Connected vs Disconnected



Connected: Can reach any node from any other node

Disconnected: Has isolated components

Neo4j: Relationship Best Practices



❌ **Wrong:** Creating relationships in both directions

Introducing Cypher

The Language of Neo4j

What is Cypher?

> **Cypher = Language to talk to Neo4j**

Just like:

- 🗣️ You speak **English** to communicate with people
- 💬 You write **Cypher** to communicate with Neo4j

> **Special feature:** Cypher uses VISUAL patterns!
>
> It looks like the graphs you draw!

Cypher Pattern: Nodes

> **Nodes are written with ( )**

```
Examples:
()                    ← A node (any node)
(p)                   ← A node we call "p"
(:Person)             ← A Person node
(p:Person)            ← A Person node we call "p"
```

Think of ( ) as drawing a circle!

Cypher Pattern: Relationships

> **Relationships are written with -[ ]->**

```
Examples:
-[]->;                ← Any relationship
-[r]->;               ← Relationship we call "r"
-[:KNOWS]->;          ← A KNOWS relationship
-[r:KNOWS]->;         ← A KNOWS relationship we call "r"
```

Think of --> as drawing an arrow!

Cypher Pattern: Complete Example

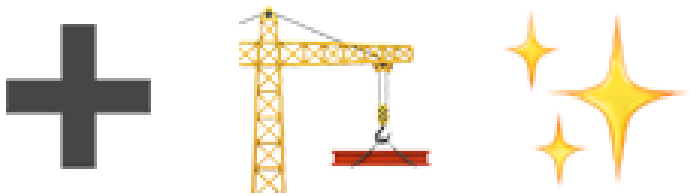
```
(alice:Person)-[:KNOWS]->(bob:Person)
```

Reads as:

"Alice (a Person) KNOWS Bob (a Person)"

> **See how visual it is?**
>
> The code looks like the graph!

Creating Data in Neo4j



Creating Your First Node

> **Use CREATE to make things**

Example:

```
CREATE ()
```

This creates an **empty node** (not very useful!)

Creating a Node with a Label

```
CREATE (:Person)
```

This creates a **Person** node

> **Label** = Type of thing (Person, Part, City, etc.)

Creating a Node with Properties

```
CREATE (:Person {name: "Alice", age: 25})
```

This creates a Person named Alice, age 25

> **Properties** = Details about the node
>
> Written in { } with `key: value` pairs

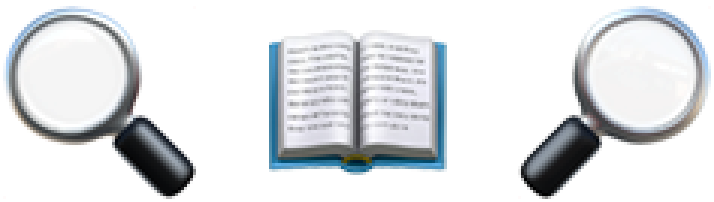
Creating Relationships

```
CREATE (alice:Person {name: "Alice"})
       -[:KNOWS]->
       (bob:Person {name: "Bob"})
```

Creates Alice, Bob, and the KNOWS relationship in one go!

> **Important:** Relationships MUST have a type!

Reading Data from Neo4j



Finding Data with MATCH

Use MATCH to FIND things

```
MATCH (n)
RETURN n
LIMIT 10
```

This finds all nodes (limited to 10)

Always use LIMIT when exploring!

Finding by Property

```
MATCH (p:Person {name: 'Alice'})
RETURN p
```

This finds the Person named Alice

> **Like searching:**
>
> "Find me the person called Alice"

Using WHERE for Filters

```
MATCH (p:Person)
WHERE p.age > 20
RETURN p.name, p.age
```

Finds all People older than 20

> **WHERE allows:** `>`, `<`, `=`, `AND`, `OR`

Finding Relationships

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
RETURN a.name, b.name
```

Finds all Person-KNOWS-Person connections

> **Reads as:** "Find person A who KNOWS person B"

Multi-Hop Queries

```
MATCH (a:Person)-[:KNOWS*2]->(c)
RETURN a.name, c.name
```

Finds "friends of friends" (2 hops away)

> ***2** = exactly 2 hops
>
> ***1..3** = between 1 and 3 hops

Updating Data

Updating with SET

```
MATCH (p:Person {name: "Alice"})
SET p.age = 26
RETURN p
```

Changes Alice's age to 26

1. MATCH (find it)
2. SET (change it)

MERGE: Create or Update

MERGE = CREATE if not exists, else find it

```
MERGE (p:Person {name: "Alice"})
RETURN p
```

Prevents duplicates! Very useful for data imports

Deleting Data

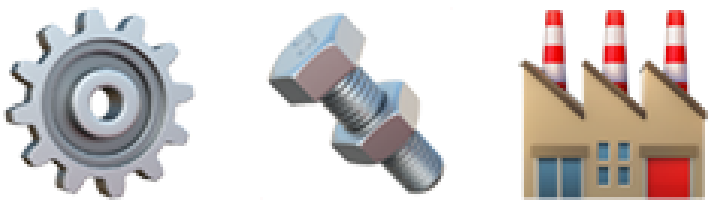
Deleting Nodes

```
MATCH (p:Person {name: "Bob"})
DETACH DELETE p
```

Deletes Bob AND all his relationships

> **DETACH DELETE** removes the node + relationships
>
> Without DETACH, you'll get an error!

Practical Example: Bill of Materials



Creating a Simple BOM

```
CREATE (engine:Part {name: "Engine"})
       -[:CONTAINS]->
       (block:Part {name: "Cylinder Block"})
       -[:CONTAINS]->
       (cylinder:Part {name: "Cylinder"})
       -[:CONTAINS]->
       (piston:Part {name: "Piston"})
```

Creates Engine → Block → Cylinder → Piston hierarchy!

Query: What's in the Engine?

```
MATCH (e:Part {name: "Engine"})
      -[:CONTAINS*]->
      (p:Part)
RETURN p.name
```

Results:

- Cylinder Block
- Cylinder
- Piston

Counting with COUNT

```
MATCH (e:Part {name: "Engine"})
      -[:CONTAINS*]->
      (p:Part)
RETURN COUNT(p) as total_parts
```

Result: total_parts: 3

Importing from CSV

```
LOAD CSV WITH HEADERS
FROM 'file:///parts.csv' AS row
CREATE (:Part {
  name: row.name,
  weight: toFloat(row.weight)
})
```

Imports parts from CSV file!

Sorting Results

```
MATCH (p:Part)
RETURN p.name, p.weight
ORDER BY p.weight DESC
LIMIT 5
```

ORDER BY = Sort

DESC = High to low, **ASC** = Low to high

Cypher Cheat Sheet

Create: `CREATE (:Label {property: value})`

Find: `MATCH (n:Label) RETURN n`

Update: `MATCH (n) SET n.property = value`

Delete: `MATCH (n) DETACH DELETE n`

Find relationship: `MATCH (a)-[r:TYPE]->(b) RETURN a, b`

Count: `RETURN COUNT(n)`

🛠️ Hands-On Exercise

10-Minute Practice

Factory Assembly Line Problem

📋 Problem: Assembly Line Management

Scenario: Small factory producing electric motors

- **Workstations:** Winding, Assembly, Testing
- **Parts:** Coil (at Winding), Housing (supplier), Motor (assembled), Tested Motor (final)
- **Operators:** Marie (Winding + Assembly), John (Testing)
- **Times:** Winding 30min, Assembly 45min, Testing 15min

Your task: Model this as a graph and query it!

🎯 Exercise Objectives

You will learn to:

1. Model an unstructured problem as a graph
2. Create nodes (workstations, operators, parts)
3. Create relationships (PROCESSES, WORKS_AT, SENDS_TO)
4. Search the graph (find paths, operators, process times)
5. Update data (change process times)
6. Add new data (new operator, new station)
7. Delete data (remove a node)
8. Clean up (delete entire graph)

Let's start! 🚀

Step 1: Create Workstations

Task: Create the 3 workstation nodes

```
CREATE (:Workstation {
  name: "Winding Station",
  processTime: 30
})
```

Your turn! Create the other 2 workstations:

- "Assembly Station" with processTime: 45
- "Testing Station" with processTime: 15

Step 1: Solution ✅

> 💡 **Solution:**

```
CREATE (:Workstation {
  name: "Assembly Station",
  processTime: 45
})

CREATE (:Workstation {
  name: "Testing Station",
  processTime: 15
})
```

Step 2: Create Operators

Task: Create the 2 operator nodes

```
CREATE (:Operator {
  name: "Marie",
  experience: 5
})

CREATE (:Operator {
  name: "John",
  experience: 3
})
```

✅ You now have 2 operators and 3 workstations!

Step 3: Operators → Workstations

Task: Marie works at Winding and Assembly, John works at Testing

```
MATCH (marie:Operator {name: "Marie"})
MATCH (winding:Workstation {name: "Winding Station"})
CREATE (marie)-[:WORKS_AT]->(winding)
```

Your turn! Create:

- Marie WORKS_AT Assembly Station
- John WORKS_AT Testing Station

Step 3: Solution ✅

> 💡 **Solution:**

```
MATCH (marie:Operator {name: "Marie"})
MATCH (assembly:Workstation {name: "Assembly Station"})
CREATE (marie)-[:WORKS_AT]->(assembly)

MATCH (john:Operator {name: "John"})
MATCH (testing:Workstation {name: "Testing Station"})
CREATE (john)-[:WORKS_AT]->(testing)
```

Step 4: Create Parts

> **Task:** Create the 4 part types

```
CREATE (:Part {name: "Coil", cost: 5.0})
CREATE (:Part {name: "Housing", cost: 12.0})
CREATE (:Part {name: "Motor", cost: 50.0})
CREATE (:Part {name: "Tested Motor", cost: 50.0})
```

Simple! All parts created in one go!

Step 5: Workstations → Parts

> **Task:** Connect workstations to the parts they produce

```
MATCH (winding:Workstation {name: "Winding Station"})
MATCH (coil:Part {name: "Coil"})
CREATE (winding)-[:PRODUCES]->(coil)

MATCH (assembly:Workstation {name: "Assembly Station"})
MATCH (motor:Part {name: "Motor"})
CREATE (assembly)-[:PRODUCES]->(motor)

MATCH (testing:Workstation {name: "Testing Station"})
MATCH (tested:Part {name: "Tested Motor"})
CREATE (testing)-[:PRODUCES]->(tested)
```

Step 6: Process Flow Between Stations

Task: Create the flow: Winding → Assembly → Testing

```
MATCH (w:Workstation {name: "Winding Station"})
MATCH (a:Workstation {name: "Assembly Station"})
CREATE (w)-[:SENDS_TO]->(a)

MATCH (a:Workstation {name: "Assembly Station"})
MATCH (t:Workstation {name: "Testing Station"})
CREATE (a)-[:SENDS_TO]->(t)
```

✅ Your assembly line flow is complete!

Step 7: View Your Graph

Task: See everything you created!

```
MATCH (n)
RETURN n
LIMIT 25
```

What you should see:

Step 8: Search - Who Works Where?

> **Question:** Which workstations does Marie work at?

```
MATCH (marie:Operator {name: "Marie"})
      -[:WORKS_AT]->
      (station:Workstation)
RETURN station.name
```

Expected Result:

Step 9: Search - Complete Process Flow

Question: What's the complete path from Winding to Testing?

```
MATCH path = (start:Workstation {name: "Winding Station"})
             -[:SENDS_TO*]->
             (end:Workstation {name: "Testing Station"})
RETURN start.name, end.name
```

This shows: Winding connects to Testing (through Assembly)

Step 10: Calculate Total Process Time

> **Question:** What's the total time for the entire process?

```
MATCH (s:Workstation)
RETURN SUM(s.processTime) as totalTime
```

Result: totalTime: 90 minutes

(30 + 45 + 15 = 90)

Step 11: Update - Improve Process Time

> **Task:** Assembly Station got upgraded! Reduce time to 35 min

```
MATCH (s:Workstation {name: "Assembly Station"})
SET s.processTime = 35
RETURN s.name, s.processTime
```

> ✅ Process time updated from 45 to 35 minutes!

Step 12: Add New Operator

Task: Hire a new operator "Lisa" who will work at Assembly

```
CREATE (lisa:Operator {name: "Lisa", experience: 2})

MATCH (lisa:Operator {name: "Lisa"})
MATCH (assembly:Workstation {name: "Assembly Station"})
CREATE (lisa)-[:WORKS_AT]->(assembly)

RETURN lisa, assembly
```

✅ Lisa is now working at Assembly Station!

Step 13: Add Packaging Station

Task: Add a new "Packaging Station" after Testing

```
CREATE (pkg:Workstation {
  name: "Packaging Station",
  processTime: 10
})

MATCH (testing:Workstation {name: "Testing Station"})
MATCH (pkg:Workstation {name: "Packaging Station"})
CREATE (testing)-[:SENDS_TO]->(pkg)

RETURN pkg
```

✅ New station added to the line!

Step 14: Count What We Have

> **Task:** How many workstations and operators do we have now?

```
MATCH (s:Workstation)
RETURN COUNT(s) as totalStations

MATCH (o:Operator)
RETURN COUNT(o) as totalOperators
```

Results:

Stations: 4 (Winding, Assembly, Testing, Packaging)

Operators: 3 (Marie, John, Lisa)

Step 15: Remove Packaging Station

Task: Packaging station didn't work out. Remove it!

```
MATCH (pkg:Workstation {name: "Packaging Station"})
DETACH DELETE pkg
```

DETACH DELETE: Removes node + relationships

✅ Packaging station removed!

Step 16: Clean Up - Delete Everything

⚠️ **Warning:** This deletes the ENTIRE graph!

```
MATCH (n)
DETACH DELETE n
```

Verify it's empty:

```
MATCH (n) RETURN COUNT(n)
```

🎉 Exercise Complete!

What you just did:

- ✅ Modeled a real factory problem as a graph
- ✅ Created 9 nodes (3 stations, 2 operators, 4 parts)
- ✅ Created relationships (WORKS_AT, PRODUCES, SENDS_TO)
- ✅ Queried the graph (operators, flow, calculations)
- ✅ Updated data (changed process time)
- ✅ Added new nodes (Lisa, Packaging)
- ✅ Deleted nodes (removed Packaging)
- ✅ Cleaned up (deleted entire graph)

You now know how to use Neo4j! 🚀

Key Takeaways

1. Cypher is Visual

Patterns look like graphs: `(a)-[:REL]->(b)`

2. Simple Operations

CREATE, MATCH, SET, DELETE, MERGE

3. Graph Traversal is Powerful

Multi-hop queries are easy and fast!

Neo4j Tools

Neo4j Desktop

Manage databases locally

Neo4j Browser

Write Cypher queries, visualize results

Neo4j Bloom

Visual exploration without code

Next Steps

Practice!

- Install Neo4j Desktop
- Try the examples from today
- Build your own graphs

Free Resources:

- Neo4j Sandbox (online, no install!)
- Neo4j GraphAcademy (free courses)
- neo4j.com/docs

Summary

- ✅ Neo4j is a graph database
- ✅ Cypher is the query language
- ✅ Use ( ) for nodes, -[]-> for relationships
- ✅ CREATE to make, MATCH to find, SET to update
- ✅ Graph properties: directed, weighted, cyclic, dense
- ✅ Graph traversal is powerful and fast!

You're ready to build with Neo4j! 🎉

Questions?

Thank you!

joseph.azar@utbm.fr