

Advanced Graph Modeling Patterns

Session 3






Joseph Azar

CP58 - UTBM Sevenans

Quick Recap

What we learned so far:

-  Nodes and Relationships (basic patterns)
-  Cypher basics (CREATE, MATCH, SET, DELETE)
-  Simple queries and traversals

Today: Learn advanced patterns for real-world complexity!

The Story

A Manufacturing Challenge



Meet AutoMeca Industries

Company: AutoMeca Industries

Business: Automotive parts manufacturing

Challenge: Managing complex supplier relationships, equipment maintenance, and product traceability

Their pain points:

- Suppliers both provide parts AND maintain equipment
- Equipment needs scheduled maintenance over time
- Parts have quantities and specifications
- Production batches need full traceability

Why Simple Graphs Don't Work

Reality is more complex than simple arrows!

Example Problems:

- "Supplier X provides 500 bolts per order, but also services our CNC machines"
- "Maintenance contract valid from Jan 2024 to Dec 2024"
- "Engineer manages 3 projects while also being a team member on 2 others"
- "Part A contains 4 units of Part B"

We need advanced patterns!

Pattern 1

Multiple Relationships Between Nodes



Multiple Relationships: The Concept

Two nodes can have MULTIPLE types of relationships!



Same supplier, different roles!

When to Use Multiple Relationships?

Perfect for:

- Different roles (supplier + maintainer)
- Different contexts (manager + team member)
- Different time periods (current + historical)
- Different interaction types (collaborates + reports to)

Key principle: Each relationship represents a DIFFERENT type of connection!

Hands-On: Multiple Relationships

Scenario: Model a supplier that both provides parts AND maintains equipment

Step 1: Create the factory and supplier

```
CREATE (factory:Factory {name: "AutoMeca Plant 1"})
CREATE (supplier:Supplier {
  name: "TechSupply Co",
  contact: "tech@supply.com"
})
```

 Run this query first!

Step 2: Create Equipment and Parts

```
CREATE (machine:Equipment {  
  name: "CNC Mill 3000",  
  serialNumber: "CNC-2024-001"  
})  
  
CREATE (bolts:Part {  
  name: "M8 Bolts",  
  specification: "ISO 4017"  
})
```

✅ Now we have all the entities!

Step 3: Create Multiple Relationships

```
MATCH (s:Supplier {name: "TechSupply Co"})
MATCH (f:Factory {name: "AutoMeca Plant 1"})
MATCH (m:Equipment {name: "CNC Mill 3000"})
MATCH (b:Part {name: "M8 Bolts"})

CREATE (s)-[:PROVIDES {
  quantity: 5000,
  pricePerUnit: 0.15,
  leadTimeDays: 7
}]->(b)

CREATE (s)-[:MAINTAINS {
  contractStart: date('2024-01-01'),
  contractEnd: date('2024-12-31'),
  responseTimeHours: 4
}]->(m)
```

✅ One supplier, TWO different relationships!

Step 4: Query Multiple Relationships

Question: What does TechSupply Co do for us?

```
MATCH (s:Supplier {name: "TechSupply Co"})-[r]->(target)
RETURN type(r) as relationship_type,
       labels(target) as target_type,
       target.name as target_name,
       properties(r) as details
```

Expected Result:

- PROVIDES → M8 Bolts (qty: 5000, price: 0.15, lead: 7 days)
- MAINTAINS → CNC Mill 3000 (contract: 2024, response: 4 hrs)

Practical Query: Find All Maintainers

Question: Which suppliers maintain our equipment?

```
MATCH (s:Supplier)-[m:MAINTAINS]->(e:Equipment)
WHERE m.contractEnd >= date()
RETURN s.name as supplier,
       e.name as equipment,
       m.responseTimeHours as response_time
ORDER BY m.responseTimeHours
```

✓ Find active maintenance contracts, sorted by response time!

Pattern 2

Rich Relationship Properties



Relationship Properties: The Power

Relationships aren't just arrows—they carry DATA!

What can we store on relationships?

- **Quantities:** How many parts in assembly
- **Dates:** When relationship is valid
- **Costs:** Price, lead time, shipping cost
- **Quality metrics:** Rating, reliability score
- **Status:** Active, pending, expired

Real Example: Bill of Materials

Challenge: An engine contains 4 cylinders, each cylinder contains 1 piston with 3 rings

Without properties: Can't represent quantities!

With properties: Store quantity on CONTAINS relationship!

Hands-On: BOM with Quantities

Let's build: Engine → Cylinder Block → Cylinders → Pistons → Rings

```
CREATE (engine:Part {name: "V4 Engine", partNumber: "ENG-V4-2024"})
CREATE (block:Part {name: "Cylinder Block", partNumber: "BLK-001"})
CREATE (cylinder:Part {name: "Cylinder", partNumber: "CYL-001"})
CREATE (piston:Part {name: "Piston", partNumber: "PST-001"})
CREATE (ring:Part {name: "Piston Ring", partNumber: "RNG-001"})

CREATE (engine)-[:CONTAINS {quantity: 1}]->(block)
CREATE (block)-[:CONTAINS {quantity: 4}]->(cylinder)
CREATE (cylinder)-[:CONTAINS {quantity: 1}]->(piston)
CREATE (piston)-[:CONTAINS {quantity: 3}]->(ring)
```

Calculate Total Parts Needed

Question: How many piston rings does the engine need?

```
MATCH path = (e:Part {name: "V4 Engine"})
              -[:CONTAINS*]->(r:Part {name: "Piston Ring"})
WITH relationships(path) as rels
RETURN reduce(total = 1, rel IN rels |
             total * rel.quantity) as total_rings
```

Result: 12 rings

(1 engine × 1 block × 4 cylinders × 1 piston × 3 rings = 12)

Let's Break Down This Query

This query looks complex! Let's understand it step by step:

```
MATCH path = (e:Part {name: "V4 Engine"})
             -[:CONTAINS*]->(r:Part {name: "Piston Ring"})
WITH relationships(path) as rels
RETURN reduce(total = 1, rel IN rels |
            total * rel.quantity) as total_rings
```

Step 1: Find the Path

```
MATCH path = (e:Part {name: "V4 Engine"})  
-[:CONTAINS*]->(r:Part {name: "Piston Ring"})
```

What this does:

- `path =` → Save the entire path (not just the nodes)
- `[:CONTAINS*]` → Follow CONTAINS relationships ANY number of times
- `->` → In the forward direction
- Finds: Engine → Block → Cylinder → Piston → Ring

✓ We now have the complete chain from engine to rings!

Step 2: Extract Relationships

WITH relationships(path) as rels

What this does:

- **WITH** → Pass data to the next part of the query
- **relationships(path)** → Extract just the relationships from the path
- **as rels** → Store them in a variable called "rels"

We now have:

- Relationship 1: {quantity: 1} (engine → block)
- Relationship 2: {quantity: 4} (block → cylinders)
- Relationship 3: {quantity: 1} (cylinder → piston)
- Relationship 4: {quantity: 3} (piston → rings)

Step 3: Multiply All Quantities

```
RETURN reduce(total = 1, rel IN rels |  
total * rel.quantity) as total_rings
```

What `reduce()` does:

- `total = 1` → Start with value 1
- `rel IN rels` → Loop through each relationship
- `total * rel.quantity` → Multiply total by the quantity

The calculation:

Start: `total = 1`

Step 1: `total = 1 × 1 = 1`

Step 2: `total = 1 × 4 = 4`

Key Concept: reduce()

Think of reduce() like a loop that accumulates a result

Like this in programming:

```
total = 1
for rel in rels:
    total = total * rel.quantity
return total
```

Cypher version:

```
reduce(
  total = 1,
  rel IN rels |
  total * rel.quantity
)
```

Use reduce() when you need to: Calculate totals, multiply values, concatenate strings, find max/min across a path

Find All Parts in Engine

```
MATCH (e:Part {name: "V4 Engine"})  
      -[c:CONTAINS*]->(part:Part)  
RETURN part.name as part_name,  
       part.partNumber as part_number
```

Returns:

- Cylinder Block (BLK-001)
- Cylinder (CYL-001)
- Piston (PST-001)
- Piston Ring (RNG-001)

Pattern 3

Time-Based Relationships



Time-Based Relationships: Why?

Relationships change over time!

Real-world scenarios:

- Maintenance contracts (start date, end date)
- Employee assignments (joined project, left project)
- Supplier agreements (contract period)
- Equipment ownership (purchase date, disposal date)
- Part revisions (effective from date)



Hands-On: Maintenance Schedule

Scenario: Track equipment maintenance over time

```
CREATE (machine:Equipment {
  name: "Press Machine PM-500",
  purchaseDate: date('2020-03-15')
})

CREATE (tech1:Technician {
  name: "Marie Durant",
  specialty: "Hydraulics"
})

CREATE (tech2:Technician {
  name: "Pierre Martin",
  specialty: "Electronics"
})

CREATE (tech1)-[:MAINTAINS {
  validFrom: date('2024-01-01'),
  validTo: date('2024-06-30'),
  maintenanceType: "Hydraulic system"
}]-(machine)

CREATE (tech2)-[:MAINTAINS {
  validFrom: date('2024-07-01'),
```

Query: Who Maintains This NOW?

```
MATCH (t:Technician)-[m:MAINTAINS]->(e:Equipment)
WHERE e.name = "Press Machine PM-500"
      AND m.validFrom <= date()
      AND m.validTo >= date()
RETURN t.name as technician,
       m.maintenanceType as type,
       m.validTo as contract_end
```

✓ This finds ONLY current maintenance contracts!

Query: Maintenance History

Question: Who maintained this machine in the past?

```
MATCH (t:Technician)-[m:MAINTAINS]->(e:Equipment)
WHERE e.name = "Press Machine PM-500"
      AND m.validTo < date()
RETURN t.name as technician,
       m.maintenanceType as type,
       m.validFrom as started,
       m.validTo as ended
ORDER BY m.validFrom DESC
```

✓ Shows all expired maintenance contracts!

Pattern 4

Self-Referencing Relationships



Self-Referencing: The Concept

A node can have relationships to OTHER NODES of the SAME TYPE!

Common examples:

- **Employees:** Manager → Employee (both are Employees)
- **Parts:** Assembly → Sub-assembly (both are Parts)
- **Tasks:** Task → Prerequisite Task (both are Tasks)
- **Documents:** Document → Referenced Document

Hands-On: Organizational Structure

Build: Company hierarchy with managers and employees

```
CREATE (ceo:Employee {name: "Claire Dubois", title: "CEO"})
CREATE (cto:Employee {name: "Thomas Bernard", title: "CTO"})
CREATE (coo:Employee {name: "Sophie Martin", title: "COO"})
CREATE (dev1:Employee {name: "Lucas Petit", title: "Developer"})
CREATE (dev2:Employee {name: "Emma Leroy", title: "Developer"})
CREATE (ops1:Employee {name: "Marc Durand", title: "Ops Manager"})

CREATE (ceo)-[:MANAGES]->(cto)
CREATE (ceo)-[:MANAGES]->(coo)
CREATE (cto)-[:MANAGES]->(dev1)
CREATE (cto)-[:MANAGES]->(dev2)
CREATE (coo)-[:MANAGES]->(ops1)
```

✓ All nodes are Employees, but related hierarchically!

Query: Who Reports to the CTO?

```
MATCH (manager:Employee {title: "CTO"})  
      -[:MANAGES]->(employee:Employee)  
RETURN employee.name as employee,  
       employee.title as title
```

Result:

- Lucas Petit - Developer
- Emma Leroy - Developer

Query: All People Under CEO

Challenge: Find CEO's direct reports AND their reports (all levels)

```
MATCH (ceo:Employee {title: "CEO"})
      -[:MANAGES*]->(employee:Employee)
RETURN employee.name as employee,
        employee.title as title
ORDER BY employee.title
```

Result: All 5 employees below CEO!

The `*` finds relationships at ANY depth

Query: Who is Lucas's Boss's Boss?

```
MATCH (lucas:Employee {name: "Lucas Petit"})  
      <-[:MANAGES*2]-(boss:Employee)  
RETURN boss.name as boss,  
       boss.title as title
```

Result: Claire Dubois - CEO

The `*2` goes exactly 2 levels up!

Putting It All Together

Combined Real-World Example





Final Challenge

Mission: Build a complete production tracking system

Requirements:

- Products with BOM (quantities)
- Suppliers (provide parts + maintain equipment)
- Time-based contracts
- Employee hierarchy

Use ALL 4 patterns we learned!

Step 1: Create Product & Parts (ALL AT ONCE)

```
// Create all parts and relationships in ONE query
CREATE (motor:Product {
  name: "Electric Motor EM-1000",
  productCode: "EM1000"
}),
(housing:Part {
  name: "Motor Housing",
  partNumber: "HSG-001",
  material: "Aluminum"
}),
(rotor:Part {
  name: "Rotor Assembly",
  partNumber: "ROT-001"
}),
(bearing:Part {
  name: "Ball Bearing",
  partNumber: "BRG-001",
  specification: "6205-2RS"
}),
(motor)-[:CONTAINS {quantity: 1}]->(housing),
(motor)-[:CONTAINS {quantity: 1}]->(rotor),
(rotor)-[:CONTAINS {quantity: 2}]->(bearing)
```

✓ Create everything in one statement to keep variables available!

Step 2: Add Suppliers with MATCH

```
// First find the parts we created
MATCH (housing:Part {partNumber: "HSG-001"})
MATCH (bearing:Part {partNumber: "BRG-001"})

// Create suppliers and link them
CREATE (supplier1:Supplier {
  name: "MetalParts SA",
  location: "Lyon"
}),
(supplier2:Supplier {
  name: "BearingTech GmbH",
  location: "Stuttgart"
}),
(supplier1)-[:PROVIDES {
  quantity: 100,
  pricePerUnit: 45.00,
  leadTimeDays: 14
}]->(housing),
(supplier2)-[:PROVIDES {
  quantity: 1000,
  pricePerUnit: 3.50,
  leadTimeDays: 7
}]->(bearing)
```

Step 3: Add Equipment & Maintenance

```
// Find supplier and create equipment with maintenance
MATCH (s:Supplier {name: "MetalParts SA"})

CREATE (machine:Equipment {
  name: "Assembly Line A1",
  type: "Automated Assembly"
}),
(s)-[:MAINTAINS {
  validFrom: date('2025-01-01'),
  validTo: date('2026-12-31'),
  responseTimeHours: 8,
  contractValue: 15000
}]-(machine)
```

✅ MetalParts now PROVIDES parts AND MAINTAINS equipment!

Final Query: Complete Supplier Info

Question: What does MetalParts SA do for us?

```
MATCH (s:Supplier {name: "MetalParts SA"})-[r]->(target)
RETURN type(r) as relationship,
       labels(target)[0] as target_type,
       target.name as target_name,
       properties(r) as details
```

Shows:

- PROVIDES → Motor Housing (qty: 100, price: €45, lead: 14 days)
- MAINTAINS → Assembly Line A1 (contract: 2024, response: 8hrs, value: €15k)

Real Query: Total Cost Calculation

Business Question: What's the total parts cost for one motor?

```
MATCH (motor:Product {name: "Electric Motor EM-1000"})
  -[c:CONTAINS*]->(part:Part)
  <-[p:PROVIDES]-(supplier:Supplier)
WITH part,
  reduce(qty = 1, rel IN c | qty * rel.quantity) as total_qty,
  p.pricePerUnit as unit_price
RETURN part.name as part,
  total_qty as quantity_needed,
  unit_price as price_per_unit,
  total_qty * unit_price as total_cost
```

Breaking Down: Total Cost Query

This query combines multiple advanced concepts!

```
MATCH (motor:Product {name: "Electric Motor EM-1000"})
      -[c:CONTAINS*]->(part:Part)
      <-[p:PROVIDES]-(supplier:Supplier)
WITH part,
      reduce(qty = 1, rel IN c | qty * rel.quantity) as total_qty,
      p.pricePerUnit as unit_price
RETURN part.name as part,
       total_qty as quantity_needed,
       unit_price as price_per_unit,
       total_qty * unit_price as total_cost
```

Step 1: Bidirectional Pattern

```
MATCH (motor:Product {name: "Electric Motor EM-1000"})  
  -[c:CONTAINS*]->(part:Part)  
  <-[p:PROVIDES]-(supplier:Supplier)
```

What's happening:

- `-[c:CONTAINS*]->` → Forward: Motor to Parts (variable depth)
- `<-[p:PROVIDES]-` → Backward: Part from Supplier
- This finds: Motor → Part ← Supplier connections

Example paths found:

- Motor → Housing ← MetalParts SA
- Motor → Rotor → Bearing ← BearingTech GmbH

Step 2: Calculate Quantities

WITH part,
reduce(qty = 1, rel IN c | qty * rel.quantity) as total_qty,
p.pricePerUnit as unit_price

Breaking it down:

- `part` → Keep the part node for later
- `reduce(...)` → Multiply all quantities in path `c`
- `p.pricePerUnit` → Get supplier's price

Example calculation:

For Bearing: Motor(1) × Rotor(1) × Bearing(2) = **2 bearings needed**

Step 3: Calculate Total Cost

```
RETURN part.name as part,  
       total_qty as quantity_needed,  
       unit_price as price_per_unit,  
       total_qty * unit_price as total_cost
```

Example result:

Part	Qty Needed	Price/Unit	Total Cost
Motor Housing	1	€45.00	€45.00
Ball Bearing	2	€3.50	€7.00

✓ Instant BOM cost analysis!

Real Query: Critical Suppliers

Risk Question: Which suppliers are critical (both provide parts AND maintain equipment)?

```
MATCH (s:Supplier)-[:PROVIDES]->(p:Part)
MATCH (s)-[m:MAINTAINS]->(e:Equipment)
WHERE m.validTo >= date('2025-01-01')
RETURN s.name as supplier,
       s.location as location,
       COUNT(DISTINCT p) as parts_provided,
       COUNT(DISTINCT e) as equipment_maintained
ORDER BY parts_provided DESC
```

✓ Identifies suppliers you really can't afford to lose!

Summary: 4 Advanced Patterns

1. Multiple Relationships: Same nodes, different connection types

2. Relationship Properties: Store data ON the relationship (quantities, dates, costs)

3. Time-Based Relationships: ValidFrom/ValidTo for temporal data

4. Self-Referencing: Nodes of same type connected (hierarchies)

Key Takeaways

 **Reality is complex—your graph can handle it!**

Remember:

- Use multiple relationship TYPES for different roles
- Put data ON relationships when it describes the connection
- Use dates on relationships for time-based tracking
- Don't fear self-references—they're powerful!

Practice Tips

How to master these patterns:

1. **Start simple:** Build basic model first
2. **Add complexity:** Gradually add relationships
3. **Query often:** Verify your model works
4. **Iterate:** Refine based on queries you need

Common mistake: Making model too complex too soon!

Quick Reference

Multiple Relationships:

```
(a)-[:TYPE1]->(b), (a)-[:TYPE2]->(b)
```

Relationship with Properties:

```
(a)-[:TYPE {prop1: value1, prop2: value2}]->(b)
```

Time-Based Query:

```
WHERE r.validFrom <= date() AND r.validTo >= date()
```

Variable Depth Traversal:

```
(a)-[:TYPE*]->(b) // any depth  
(a)-[:TYPE*2]->(b) // exactly 2 hops  
(a)-[:TYPE*1..3]->(b) // 1 to 3 hops
```

What's Next?

Next Session:

Graph Algorithms for Engineering

- Shortest path finding
- Community detection
- Centrality analysis
- Impact analysis

Homework: Build a graph for YOUR engineering domain using these patterns!

Questions?



Thank you!

joseph.azar@utbm.fr

