Graph Data Management

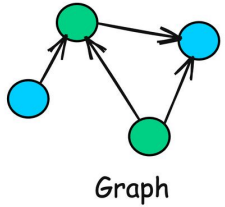
NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Graph Data Management



- Also: property graph databases, graph stores
- Store data as **nodes** and **relationships**
  - Entities, relationships, properties
  - Data are naturally modeled as a graph
- Schema is often flexible (lightweight)
- Typical operations:
  - Find a node or neighborhood
  - Traverse relationships
  - Match structural patterns
  - …
- Joins are replaced by traversals

JanusGraph

NebulaGraph

TigerGraph

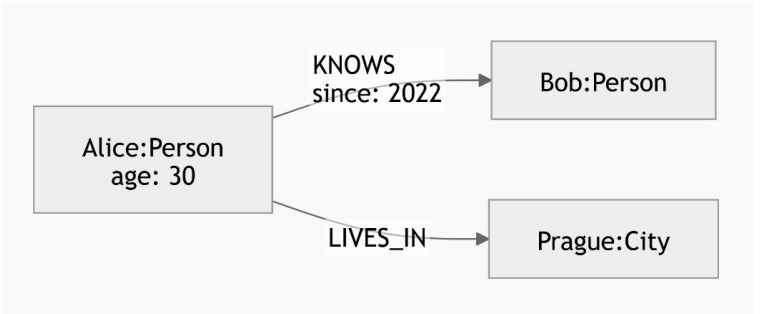
neo4j

Property Graph Model

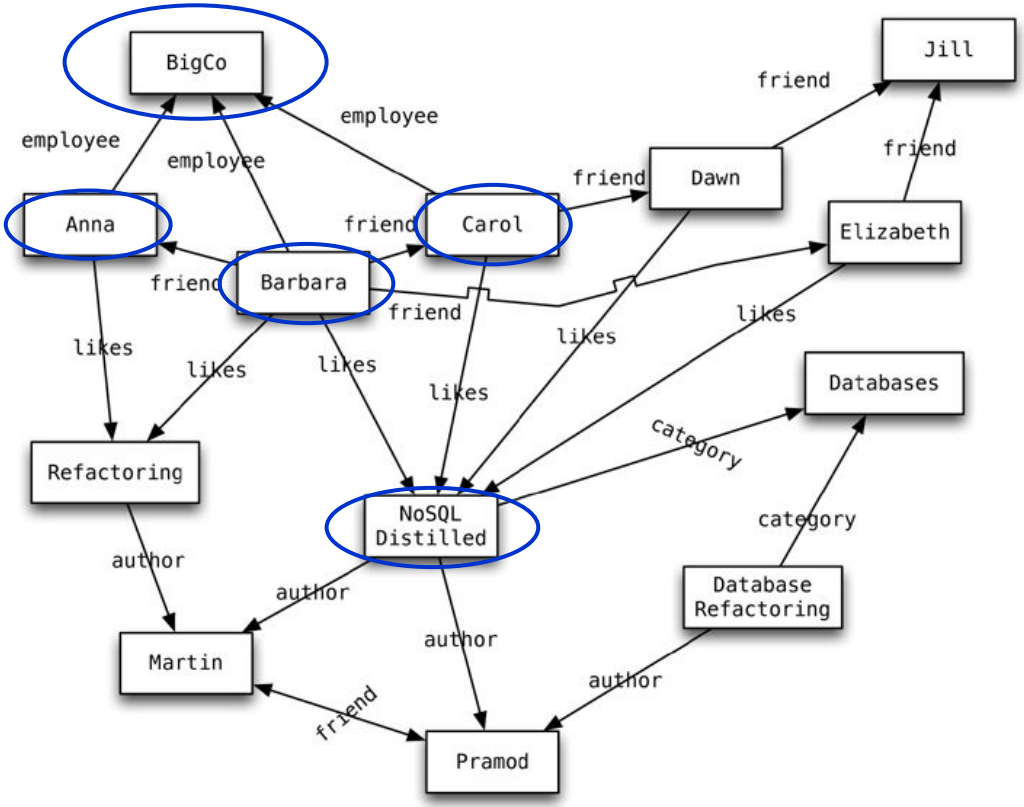
Two variants:
- **Labeled Property Graph** (LPG) – schema-later (Neo4j)
- **Typed Property Graph** (TPG) – schema-first

```
(Alice:Person {age: 30})-[:KNOWS {since: 2022}]->(Bob:Person)
(Alice)-[:LIVES_IN]->(Prague:City)
```

- Node
  - identifier
  - **labels**
    - Group nodes of similar role
    - LPG-specific concept
  - properties
- Relationship
  - directed edge between two nodes
  - **type**
  - properties
- Property
  - key-value pair, e.g. age: 30, since: 2022



Example - Graph Traversal



Relational vs. Graph Databases

Relational DBs	Graph DBs
data in tables	data as nodes and relationships
relationships via foreign keys	relationships stored directly
multi-hop queries often need joins	multi-hop queries use traversals
good for structured tabular data	good for highly interconnected data

Graph databases are especially useful when relationships are central

Person

id	name	age
1	Alice	30
2	Bob	NULL

City

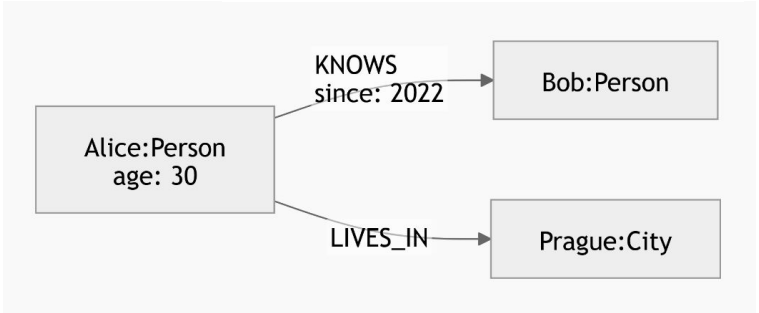
id	name
1	Prague

Knows

person1_id	person2_id	since
1	2	2022

LivesIn

person_id	city_id
1	1

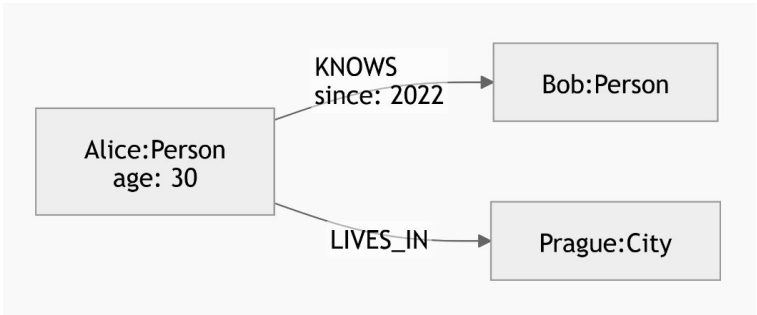


Document vs. Graph Databases

Document DBs	Graph DBs
data stored as documents	data stored as nodes and relationships
good for hierarchical and nested data	good for highly interconnected data
natural representation as a tree	natural representation as a graph
references are possible, but not central	relationships are first-class

```
{
  "name": "Alice",
  "age": 30,
  "lives_in": {
    "name": "Prague"
  },
  "knows": [
    {
      "name": "Bob",
      "since": 2022
    }
  ]
}
```

Graph databases are a better fit when links between entities are important



Wide-Column vs. Graph Databases

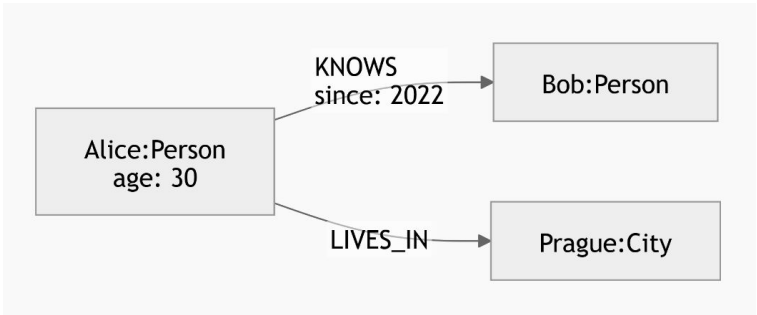
Wide-Column DBs	Graph DBs
data grouped by row key and columns	data stored as nodes and relationships
optimized for lookup by key	optimized for traversal of connected data
good for large-scale sparse data	good for relationship-centric queries
weaker support for explicit relationships	relationships stored directly

Wide-column databases optimize access by key, while graph databases optimize access by relationships

Wide-Column Table

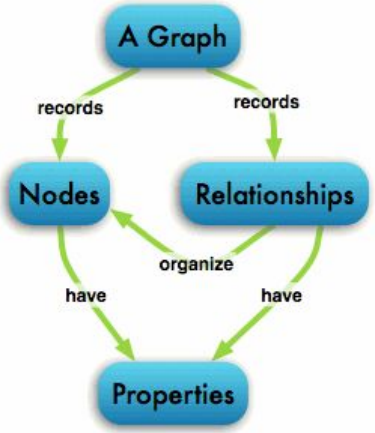
row key	age	lives_in	knows:Bob
Alice	30	Prague	2022

row key	name
Bob	Bob



Neo4j

- One of the best-known property graph databases
  - Open source
  - Written in: Java
    - Cross-platform
- Initial release: 2007
- Data: Nodes, relationships, labels, and properties
- Data model: labeled property graph (schema-optional)
- Queries: mainly using **Cypher**
- Supports official drivers for multiple languages
  - Including Python



https://neo4j.com/

Data Model

- Fundamental units: **nodes** and **relationships**
- Both can have **properties**
  - Stored as key–value pairs
    - Value - simple type or array of simple types
- Relationships are directed
  - Have type, start node, and end node
  - Can be recursive
- In queries, direction can be used or ignored

Missing value is usually represented by an absent property

Type	Description	Value range
boolean		true/false
byte	8-bit integer	-128 to 127, inclusive
short	16-bit integer	-32768 to 32767, inclusive
int	32-bit integer	-2147483648 to 2147483647, inclusive
long	64-bit integer	-9223372036854775808 to 9223372036854775807, inclusive
float	32-bit IEEE 754 floating-point number	
double	64-bit IEEE 754 floating-point number	
char	16-bit unsigned integers representing Unicode characters	u0000 to uffff (0 to 65535)
String	sequence of Unicode characters	

Node Labels/Edge Types

- Nodes can have zero, one, or multiple labels
  - Used for logical grouping and for selecting nodes in queries
  - Examples: `:Person`, `:City`, `:Student`
- Relationships always have exactly one type
  - Examples: `:KNOWS`, `:LIVES_IN`

```
(Alice:Person:Student)-[:LIVES_IN]->(Prague:City)
(Alice)-[:KNOWS]->(Bob:Person)
```

Cypher

- Query language for Neo4j
- Used for querying and updating
- Declarative
  - Describe what to find
  - Inspired by SQL and SPARQL
- Based on **graph patterns**
- Designed to be readable

https://neo4j.com/docs/getting-started/cypher/

Basic Cypher Clauses

- `MATCH` — find a pattern
- `WHERE` — add conditions
- `RETURN` — return results
- `CREATE` — create nodes/relationships
- `SET` — update properties/labels
- `DELETE` / `DETACH DELETE` — remove data
- `REMOVE` — removes label/property
- `WITH` — pass intermediate results

CREATE and SET

```
CREATE (a:Person {name: 'Alice', age: 30})
CREATE (b:Person {name: 'Bob'})
CREATE (c:City {name: 'Prague'})
CREATE (a)-[:KNOWS {since: 2022}]->(b)
CREATE (a)-[:LIVES_IN]->(c)

MATCH (a:Person {name: 'Alice'})
SET a.age = 31
RETURN a
```

… updates an existing property (upsert)

```
MATCH (a:Person {name: 'Alice'})
SET a:Student
RETURN a
```

… adds a new label

Cypher uses patterns to create and update graph data

CREATE

```
CREATE (a:Person {name: 'Alice', age: 30})
       -[:LIVES_IN]->
       (:City {name: 'Prague'})

MATCH (a:Person {name: 'Alice'}),
      (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2022}]->(b)

MATCH (a:Person {name: 'Alice'})
CREATE (a)-[:LIVES_IN]->(c:City {name: 'Brno'})
```

`MATCH` can find existing nodes, while `CREATE` can add both new nodes and new relationships.

MERGE

```
MERGE (a:Person {name: 'Alice'})

MATCH (a:Person {name: 'Alice'}),
      (c:City {name: 'Prague'})
MERGE (a)-[:LIVES_IN]->(c)

MERGE (a:Person {name: 'Alice'})-[:LIVES_IN]->(c:City {name: 'Prague'})

MERGE (a:Person {name: 'Alice', age: 30})

MERGE (a:Person {name: 'Alice'})
SET a.age = 30
```

`CREATE (a:Person {name: 'Alice'})` **creates a new node every time the query is executed,** `MERGE` **only if it does not exist**

(DETACH) DELETE, REMOVE

```
MATCH (a:Person {name: 'Alice'})-[r:KNOWS]->(b:Person {name: 'Bob'})
DELETE r

MATCH (b:Person {name: 'Bob'})
DETACH DELETE b

MATCH (a:Person {name: 'Alice'})
REMOVE a:Student
RETURN a

MATCH (a:Person {name: 'Alice'})
REMOVE a.age
RETURN a
```

A node with existing relationships cannot be deleted by `DELETE` alone

MATCH, WHERE, RETURN

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.name = 'Alice'
RETURN b
```

```
+--------------------------------+
| b                              |
+--------------------------------+
| (:Person {name: 'Bob'})        |
+--------------------------------+
```

```
MATCH (a:Person)-[:KNOWS]->(:Person)-[:LIVES_IN]->(c:City)
WHERE a.name = 'Alice'
RETURN c.name
```

```
+----------+
| c.name   |
+----------+
| Prague   |
| Brno     |
+----------+
```

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.name = 'Alice'
RETURN a.name, b.name AS friend
```

```
+--------+--------+
| a.name | friend |
+--------+--------+
| Alice  | Bob    |
| Alice  | Carol  |
+--------+--------+
```

DISTINCT, ORDER BY, LIMIT, SKIP

```
MATCH (a:Person)-[:KNOWS]->(:Person)-[:LIVES_IN]->(c:City)
WHERE a.name = 'Alice'
RETURN DISTINCT c.name AS city
ORDER BY city ASC
SKIP 1
LIMIT 5
```

Cypher can remove duplicates, sort results, skip selected rows, and limit the output size

Aggregation Functions

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

- Grouping is implicit
  - Non-aggregated expressions in `RETURN` define the groups
  - `a.name` is the grouping key
  - `count(b)` is evaluated for each value of the key

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
RETURN count(b) AS number_of_friends
```

WITH

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WITH b, count(*) AS popularity
WHERE popularity >= 2
MATCH (b)-[:LIVES_IN]->(c:City)
RETURN b.name AS person, popularity, c.name AS city
```

- Used to pass **intermediate results** to the next part of a query
- It allows us to:
  - Carry selected variables forward
  - Rename values with AS
  - Use aggregation results in later steps

FOREACH

```
MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.name = 'Alice'
WITH collect(b) AS friends
FOREACH (f IN friends | SET f.visited = true)
```

- `MATCH` returns multiple rows with different values of `b`
- `collect(b)` combines them into a single list
- `FOREACH` iterates over that list

Useful when we already have a list and want to apply an update to each element

Returning Paths

```
MATCH p = (a:Person)-[:KNOWS]->(b:Person)
RETURN p
```

```
+-------------------------------------------------------------+
| p                                                           |
+-------------------------------------------------------------+
| (:Person {name: 'Alice'})-[:KNOWS]->(:Person {name: 'Bob'}) |
| (:Person {name: 'Bob'})-[:KNOWS]->(:Person {name: 'Carol'}) |
+-------------------------------------------------------------+
```

- `RETURN p` returns a path
  - In Neo4j Browser, the result can be visualized as a graph
    - **Graph view** vs **table view**
- Useful for exploring connected data

Cypher results do not have to be only tables; they can also be returned as paths

MATCH vs. OPTIONAL MATCH

- MATCH requires the pattern to exist
- OPTIONAL MATCH keeps the row even if the pattern is missing
- Similar to inner join vs. left outer join
  - Missing values are returned as null

```
MATCH (a:Person {name: 'Alice'})
OPTIONAL MATCH (a)-[:LIVES_IN]->(c:City)
RETURN a.name AS person, c.name AS city
```

Direction in Patterns

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

MATCH (a)-[:KNOWS]-(b)
RETURN a, b
```

- -[]-> uses the stored direction
- -[]- ignores direction in the match
- Useful when direction is not important for the query

Cypher - Advanced Constructs

- More aggregation functions: `sum()`, `avg()`, `min()`, `max()`
- Predicates over lists: `all()`, `any()`, `none()`, `single()`
- Path functions: `nodes(p)`, `relationships(p)`, `length(p)`
- Relationship metadata: `type(r)`
- Node metadata: `labels(n)`, `keys(n)`

… and many others

https://neo4j.com/docs/cypher-manual/current/functions/

Python + Cypher

- Python sends Cypher queries to Neo4j
- Query parameters can be passed from application code
- Results can be processed programmatically
- Useful for scripts, web applications, and data pipelines

```
from neo4j import GraphDatabase

URI = "neo4j://localhost:7687"
AUTH = ("neo4j", "password")

driver = GraphDatabase.driver(URI, auth=AUTH)

driver.execute_query("""
     CREATE (a:Person {name: $name, age: $age})
     CREATE (b:Person {name: $friend})
     CREATE (c:City {name: $city})
     CREATE (a)-[:KNOWS {since: $since}]->(b)
     CREATE (a)-[:LIVES_IN]->(c)
""",
name="Alice",
age=30,
friend="Bob",
city="Prague",
since=2022,
database_="neo4j"
)

driver.close()
```

Python + Cypher

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
     "neo4j://localhost:7687",
     auth=("neo4j", "password")
)

records, summary, keys = driver.execute_query(
     """
     MATCH (a:Person)-[:LIVES_IN]->(c:City)
     RETURN a.name AS person, c.name AS city
     """,
     database_="neo4j"
)

for record in records:
     print(record["person"], record["city"])

driver.close()
```

Result: `Alice Prague`

neomodel

```
from neomodel import StructuredNode, StringProperty, RelationshipTo

class Person(StructuredNode):
    name = StringProperty()
    knows = RelationshipTo('Person', 'KNOWS')

alice = Person(name="Alice").save()
bob = Person(name="Bob").save()
alice.knows.connect(bob)
```

Low-level:

- Cypher via driver (full control)
- Explicit queries

High-level:

- OGM (Object Graph Mapping)
  - Similar to ORM
- Work with Python objects instead of queries
  - maps objects ↔ graph
  - less control, more convenience

https://neomodel.readthedocs.io/en/latest/

Path and Traversal

- **Path** = one or more nodes connected by relationships
  - Can be returned directly as a query result
- **Traversal** = visiting nodes by following relationships
  - Follows some rules:
    - relationship type
    - relationship direction
    - traversal order (e.g., BFS / DFS)
  - depth or stopping condition

Traversal Framework

- Not a second query language
  - A framework for explicit graph traversal
- Defines how the graph is explored:
  - start node(s)
  - relationship type(s)
  - relationship direction
  - traversal order - BFS / DFS
  - stopping condition / depth
  - uniqueness - whether a node / relationship / path may be visited again
  - evaluator - whether to include the current node in the result, and whether to continue or stop from it
- Cypher describes what to find
- Traversal framework controls how to traverse the graph

Cypher = what to find

Traversal framework = how to explore

BFS Traversal

- BFS = breadth-first search
  - Explores the graph **level by level**
  - One possible traversal order
- First visits direct neighbors
- Then neighbors of neighbors
- …
- Useful for finding the closest reachable nodes
- BFS vs. DFS
  - BFS is useful for nearest nodes
  - DFS is useful for deep exploration of one path

```
Start at Alice
follow KNOWS
direction: outgoing
order: BFS
depth: up to 2
uniqueness: NODE_GLOBAL
evaluator: include nodes up to depth 2
```

Different traversal orders may visit the same graph in a different sequence

Neo4j Traversal Framework – Java API

- Explicit traversal in Java
- Define how the graph should be explored
- Configure:
  - start node
  - relationship type + direction
  - order (BFS / DFS)
  - evaluator
  - uniqueness
- Then explicitly call `traverse(startNode)`

```
Node alice = getNodeByName("Alice");

TraversalDescription td = tx.traversalDescription()
     .breadthFirst()
     .relationships(Rels.KNOWS, Direction.OUTGOING)
     .evaluator(Evaluators.toDepth(2))
     .uniqueness(Uniqueness.NODE_GLOBAL);

Traverser traverser = td.traverse(alice);

for (Path path : traverser) {
     System.out.println(
     path.endNode().getProperty("name") +
          " at depth " + path.length()
     );
}
```

Unlike Python + Cypher, here we do not just submit a query — we directly control the traversal strategy

Limits of Cypher

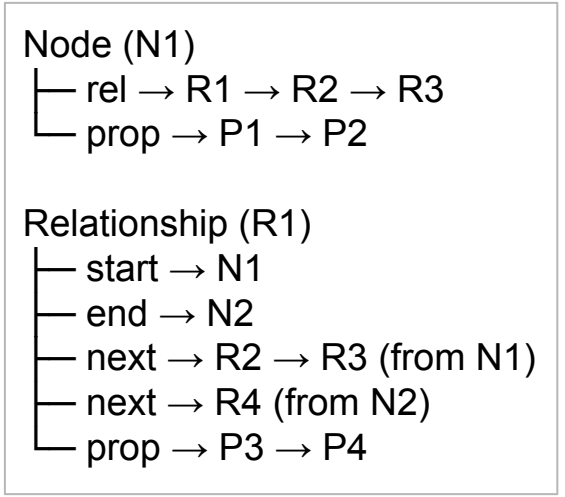
- Cypher cannot easily express:
  - Explicit BFS vs DFS
  - Custom pruning strategies
  - Fine-grained early stopping
  - Complex stateful traversal logic
- Example limitation
  - Prune at hub nodes: stop expanding from nodes with degree >100
  - Path-dependent traversal: continue only while accumulated path cost remains below a threshold
- This is where traversal API is needed

Cypher is weaker when traversal decisions depend on the current path or intermediate state

More on Internals of 

Data on Disk

- Nodes, relationships, properties stored as records
  - Have fixed size
  - Connected using references (pointers)
- Node contains:
  - pointer to first relationship
  - pointer to first property
- Relationship contains:
  - pointer to start node
  - pointer to end node
  - pointer to next relationship (for both nodes)
  - pointer to first property



Traversal = following pointers

Why is Graph Traversal Fast?

- Traversal = pointer chasing
  - No joins
  - No global scans
  - Only local pointer hops
- Cost depends on visited nodes, not on the size of the graph

Relational DB	→ join
Graph DB	→ pointer hop

Relationships are stored, not computed

Data on Disk

- Note: Neo4j is a schema-less database
  - Fixed record lengths + offsets in files → **direct access** (no scanning)
- Several types of files to store the data

Store File	Record size	Contents
neostore.nodestore.db	15 B	Nodes
neostore.relationshipstore.db	34 B	Relationships
neostore.propertystore.db	41 B	Properties for nodes and relationships
neostore.propertystore.db.strings	128 B	Values of string properties
neostore.propertystore.db.arrays	128 B	Values of array properties
Indexed Property	1/3 * AVG(X)	Each index entry is approximately 1/3 of the average property value size

https://neo4j.com/developer/kb/understanding-data-on-disk/

Transaction Management

- Transaction = a unit of work executed atomically
- In neo4j:
  - Every query runs inside a transaction
  - Support for **ACID** properties
- Transactions can be:
  - Implicit (auto-commit)
  - Explicit
- Why transactions matter?
  - Ensure consistency
  - Allow **commit** on success (make changes permanent) and **rollback** on error (discard all changes)
  - Isolate concurrent updates

Either all changes are applied, or none of them are

Implicit vs Explicit Transactions

- Implicit transaction
  - created automatically for a single query
  - usually committed automatically if the query succeeds
- Explicit transaction
  - started and controlled by the application
  - can contain multiple queries / updates
  - committed explicitly
  - rolled back if an error occurs
- Use implicit when one simple query is enough
- Use explicit when several steps must succeed together

```
create Alice
create Bob
create KNOWS
```

If step 3 fails:

- with transaction → nothing is stored
- without transaction semantics → partial update would remain

Implicit = convenience, Explicit = control

Java vs Python

Java

- Explicit transaction object
- Application controls begin / commit / rollback
- Suitable for fine-grained control

Python

- Transaction managed by the driver
- Callback receives the current transaction as `tx`
- Rollback is automatic if the callback fails

```
try (Transaction tx = db.beginTx()) {

    Node alice = tx.createNode(Label.label("Person"));
    alice.setProperty("name", "Alice");

    tx.commit();
}
```

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
    "neo4j://localhost:7687",
    auth=("neo4j", "password")
)

def create_person(tx, name):
    tx.run("CREATE (:Person {name: $name})",name=name)

with driver.session() as session:
    session.execute_write(create_person, "Alice")

driver.close()
```

Java: we create and control the transaction
Python: the driver manages the transaction

Java vs Python

```java
try (Transaction tx = db.beginTx()) {
    Node alice = tx.createNode(Label.label("Person"));
    alice.setProperty("name", "Alice");

    Node bob = tx.createNode(Label.label("Person"));
    bob.setProperty("name", "Bob");

    throw new RuntimeException("Something went wrong");
    // tx.commit();
}
```

```python
from neo4j import GraphDatabase

def create_two_people(tx):
   tx.run("CREATE (:Person {name: 'Alice'})")
   tx.run("CREATE (:Person {name: 'Bob'})")
   raise Exception("Something went wrong")

with driver.session() as session:
   try:
      session.execute_write(create_two_people)
   except Exception:
      print("Transaction rolled back")
```

Because the transaction failed, neither Alice nor Bob is stored

Why Do We Need Indexes?

- Without index: Database may need to scan many nodes
- With index: Database can find matching nodes directly
- Typical use:
  - find a person by name
  - find all movies from a given year
- Index
  - Used to find entry points in the graph
  - Based on property values
- Traversal
  - Used to explore connected data
  - Follows relationships

```
MATCH (p:Person)
WHERE p.name = 'Alice'
RETURN p
```

```
CREATE INDEX person_name_index
FOR (p:Person)
ON (p.name)
```

```
MATCH (p:Person {name: 'Alice'})
   -[:KNOWS]->(f:Person)
RETURN f.name
```

Traversal follows relationships, but an index helps us find the starting node

Neo4j Cluster: Replication, not Sharding

- Basic Neo4j cluster
  - one database is replicated across multiple nodes
    - one node acts as **leader** for writes
    - other nodes (**followers**) keep replicas and can serve read queries
  - main goal: high availability, failover, and read scaling
- What the basic cluster does not provide
  - The graph is not automatically partitioned into independent pieces
  - Clustering means replication, not horizontal partitioning/sharding

Cluster = same data on multiple servers for availability and read scaling

Neo4j Sharding with Composite Databases

- Sharding in Neo4j
  - handled through **composite databases**
  - one logical graph can be split into multiple databases (shards)
    - User-defined shards (e.g., one for each region, year, …)
  - queries can access these shards through a single entry point
- Relationships do not cross database boundaries
  - Cross-database links are modeled via proxy nodes and shared IDs
- Composite database
  - does not store data itself
  - contains aliases to local or remote constituent databases
  - enables a single Cypher query to read across multiple graphs/databases
- Only in Enterprise Edition

Fabric = different parts of the graph in different databases, queried as one logical whole

Why is Graph Sharding Difficult?

- A graph contains vertices, edges, and properties
  - We must decide what to shard:
- Two common approaches:
  - edge cut: vertices are partitioned, edges cross partitions (e.g., NebulaGraph)
  - vertex cut: edges are partitioned, high-degree vertices are shared (e.g., JanusGraph)
- Problem:
  - traversal crosses partitions
  - pointer hops become network hops
  - performance depends on communication

Sharding breaks locality → traversal becomes expensive

Beyond Property Graphs

Graph Query Paradigms

Declarative (Cypher, SPARQL)

- Describe what to find
- Database decides how to execute
- Pattern matching

Procedural (Gremlin)

- Describe how to traverse the graph
- Explicit sequence of steps
- Full control over traversal

Declarative = WHAT
Procedural = HOW

Gremlin

- Gremlin = traversal-based graph query language
  - Part of Apache TinkerPop
    - A vendor-neutral standard for graph computing
  - Works on property graphs
  - Queries are expressed as traversals
- Gremlin is procedural
  - We describe how to traverse the graph step by step
- Gremlin as pipeline (data flows through steps)

https://tinkerpop.apache.org/gremlin.html

Gremlin - Examples


```
g.V().has("name","Alice")

g.V().has("name","Alice").out("KNOWS")

g.V().has("name","Alice").out("KNOWS").values("name")

g.V().has("name","Alice").out("KNOWS").count()

g.V().has("name","Alice").out("KNOWS").out("KNOWS")

g.V().hasLabel("Person").has("age", gt(30)).values("name")
```

- selection
- traversal
- projection
- aggregation
- multi-hop
- filtering

Cypher: declarative (what)
Gremlin: procedural (how)

Property Graph vs RDF

```
SELECT ?f
WHERE {
   :Alice :knows ?f .
}
```

Property Graph (Neo4j)

- Nodes and relationships
- Both can have properties
- Relationships have types and direction
- Nodes have labels

RDF (Resource Description Framework)

- Data as triples
- Subject – predicate – object
- No explicit properties on nodes or edges
  - Uniform representation

```
(Alice)-[:KNOWS {since: 2022}]->(Bob)
```

```
Alice – knows – Bob
Alice – knowsSince – 2022
```

RDF flattens everything into triples; property graphs keep richer structure

When is RDF Useful?

- RDF / triple stores are useful when:
  - Data from many sources must be **integrated**
  - A **standard representation** is needed
  - We work with **knowledge graphs**
  - **Interoperability** is more important than compact structure
- Typical use cases
  - Semantic web
  - Linked open data
  - Enterprise knowledge graphs
- RDF graph: more uniform, more standardised
- Property graph: more natural for traversal and application queries

References

- Neo4j http://www.neo4j.org/
- Neo4j Docs https://neo4j.com/docs/
- RDF https://www.w3.org/RDF/
- SPARQL https://www.w3.org/2001/sw/wiki/SPARQL
- Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications