Representation of Graphs and Storage

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실 NETWORK SCIENCE LAB

가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA

Contents

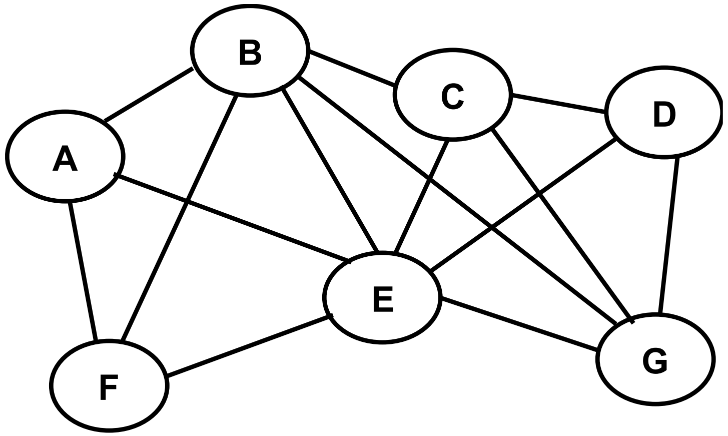
- Graph representation
  - Sparse matrix representations
  - Adjacency matrix representations
  - Adjacency matrix properties
  - Adjacency list
- Graph databases and storage systems
  - Several graph storage techniques
  - Popular GraphDB engines
  - Pros/Cons of GraphDB engines
  - Example GraphDB databases

Graph Representation

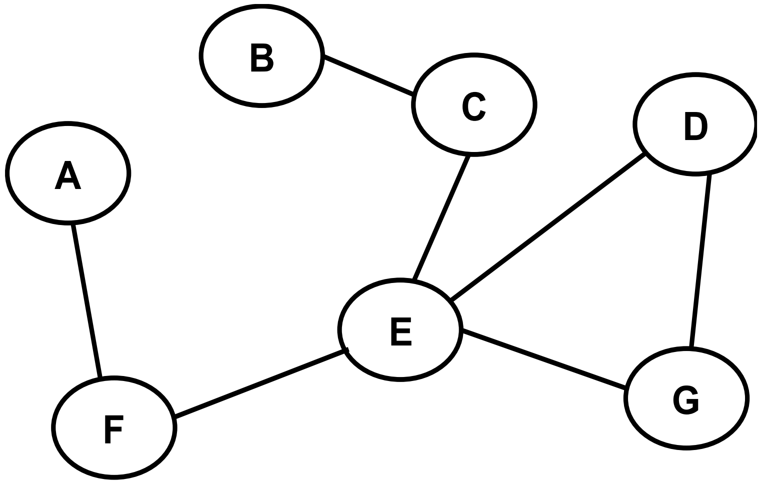
- Graphs are often useful for lots of data and questions
  - E.g., "What's the lowest-cost path from A to B?"
- We need a data structure that represents graphs
- Which data structure is "best" can depend on:
  - Properties of the graph (e.g., dense versus sparse)
  - The common queries about the graph
    - E.g., ("is (u ,v) an edge?" vs "what are the neighbors of node u?")
- We will discuss two standard graph representations
  - Adjacency Matrix and Adjacency List
  - Different trade-offs, particularly time versus space

Sparse Graph Representation

- If the graph has n vertices (nodes)
  - Maximum # of edges is n^2
- In dense graphs number of edges is close to n^2
- In sparse graphs number of edges is close to n



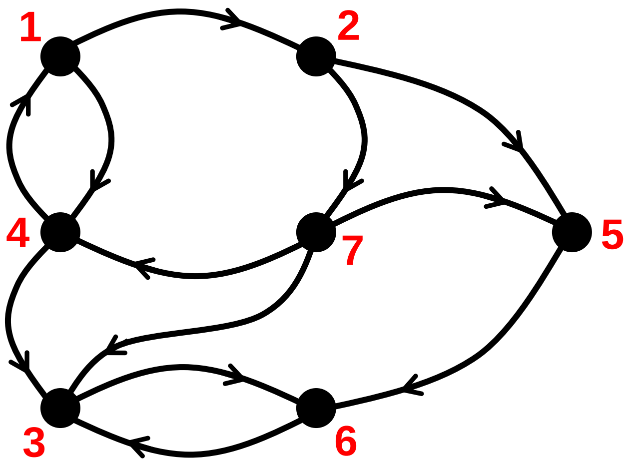
Dense graphs
(many edges between nodes)



Sparse graphs
(few edges between nodes)

Sparse Graph Representation

- Example: a web page network



	1	2	3	4	5	6	7
1		●		●			
2					●		●
3						●	
4	●		●				
5						●	
6			●				
7			●	●	●		

- Web page = Node
- Link = Directed edge
- Link matrix: $A_{ij} = 1$ if page i links to page j

Web Page Matrix

- n = 2 billion (and growing by 1 million a day)
- n x n array of ints => 16 * 1018 bytes (16 * 109 GB)
- Each page links to 10 (say) other pages on average
- On average there are 10 nonzero entries per row
- Space needed for nonzero elements is approximately 20 billion x 4 bytes = 80 billion bytes (80 GB)

Representation of Unstructured Sparse Matrix

- Single linear list in row-major order.
  - Scan the nonzero elements of the sparse matrix in row-major order
  - Each nonzero element is represented by a triple (row, column, value)
  - The list of triples may be an array list or a linked list (chain)

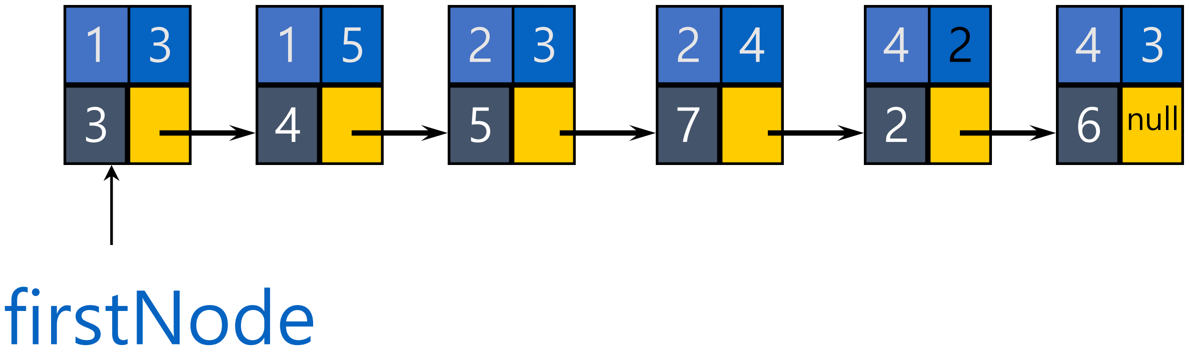
- 0 0 3 0 4
- 0 0 5 7 0
- 0 0 0 0 0
- 0 2 6 0 0

list =

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

list =

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6



- Sample code:

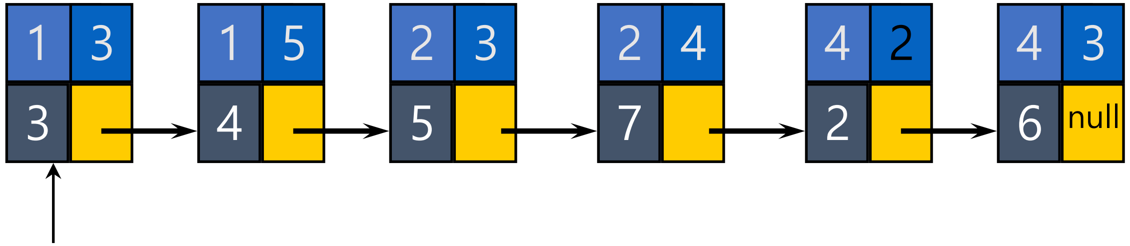
```python
import numpy as np
# Instantiate the graph
G = nx.DiGraph()
G.add_nodes_from([1,2,3,4,5])

# Define rows, columns and weighted values of the graph
rows    = [1, 1, 2, 2, 4, 4]
columns = [3, 5, 3, 4, 2, 3]
values  = [3, 4, 5, 7, 2, 6]

edges = zip(rows, columns, values)

G.add_weighted_edges_from(list(edges))


# 2D array adjacency matrix
A = nx.adjacency_matrix(G)
A_dense = A.todense()
print(A_dense)
# Pandas format of adjacency matrix
nx.to_pandas_adjacency(G)
```

list =

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

```
[[0 0 3 0 4]
 [0 0 5 7 0]
 [0 0 0 0 0]
 [0 2 6 0 0]
 [0 0 0 0 0]]
```

	1	2	3	4	5
1	0.0	0.0	3.0	0.0	4.0
2	0.0	0.0	5.0	7.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	2.0	6.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0

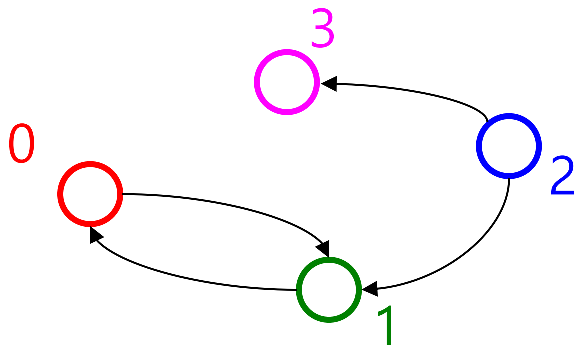


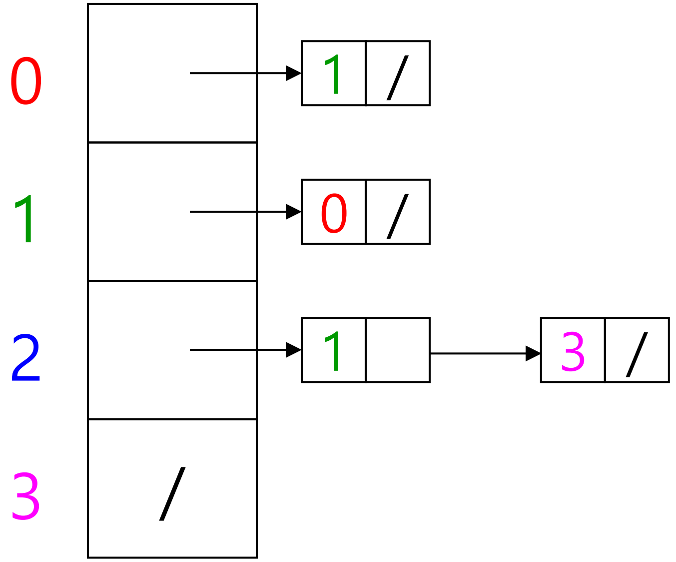
firstNode

- Sample code:

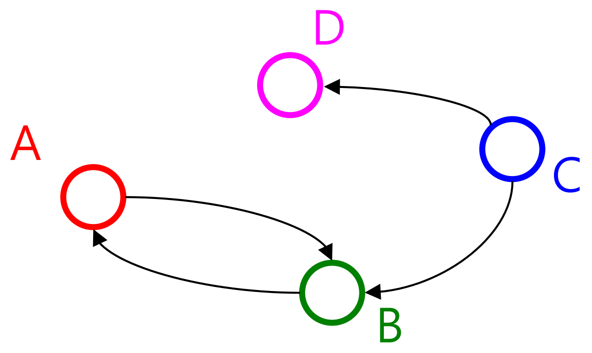
```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([(0, 1), (1, 0), (2, 1), (2, 3)])

A = nx.adjacency_matrix(G)
print(A)
```

```
  (0, 1)	1
  (1, 0)	1
  (2, 1)	1
  (2, 3)	1
```





- Assign each node a number from 0 to |V|-1
- A |V| x |V| matrix of Booleans (0 or 1)
  - Then A[i][j] == 1 if there's an edge from node i to node j; 0 otherwise.



	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

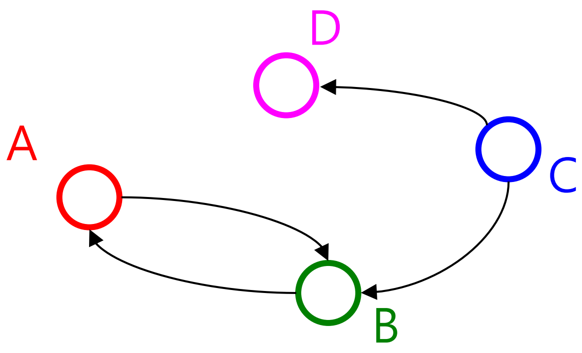
Adjacency Matrix

- Sample code:

```
import networkx as nx
import numpy as np
import pandas as pd

# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])
# 2D array adjacency matrix
A = nx.adjacency_matrix(G)
A_dense = A.todense()
print(A_dense)
# Pandas format
nx.to_pandas_adjacency(G)
```

```
[[0 1 0 0]
 [1 0 0 0]
 [0 1 0 1]
 [0 0 0 0]]
```

	A	B	C	D
A	0.0	1.0	0.0	0.0
B	1.0	0.0	0.0	0.0
C	0.0	1.0	0.0	1.0
D	0.0	0.0	0.0	0.0



	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:

- Space requirements:

- Best for sparse or dense graphs?

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges: O(|V|)
  - Get a vertex's in-edges: O(|V|)
  - Decide if some edge exists: O(1)
  - Insert an edge: O(1)
  - Delete an edge: O(1)
- Space requirements: $O(|V|^2)$
- Best for sparse or dense graphs?
  - dense

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

Adjacency Matrix Properties

- Sample code: checking connection between nodes

```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])

# Get a vertex's out-edges:
print(f"OUT-edges of node B: {G.out_degree('B')}")
# Get a vertex's in-edges:
print(f"IN-edges of node B: {G.in_degree('B')}")
# Decide if some edge exists:
print(f"Check an edge from A to C: {G.has_edge('A', 'C')}")
# Insert an edge:
G.add_edge("A", "C")
# OR
G.add_edges_from([("A", "D")])
print(f"Check an edge from A to C: {G.has_edge('A', 'C')}")
print(f"Check an edge from A to D: {G.has_edge('A', 'D')}")
# Delete an edge:
G.remove_edge("A", "C")
# OR
G.remove_edges_from([("A", "D")])
print(f"Check an edge from A to C: {G.has_edge('A', 'C')}")
print(f"Check an edge from A to D: {G.has_edge('A', 'D')}")
```

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

```
OUT-edges of node B: 1
IN-edges of node B: 2
Check an edge from A to C: False
Check an edge from A to C: True
Check an edge from A to D: True
Check an edge from A to C: False
Check an edge from A to D: False
```

Adjacency Matrix Properties

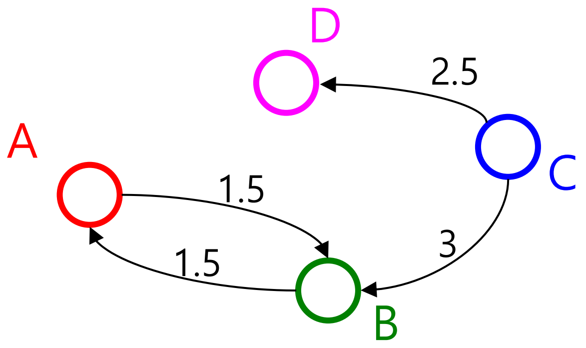
- How can we adapt the representation for weighted graphs?
  - Instead of Boolean, store a number in each cell
  - Need some value to represent ‘not an edge’
    - 0, -1, or some other value based on how you are using the graph
    - Might need to be a separate field if no restrictions on weights

- Sample code for weighted graph:

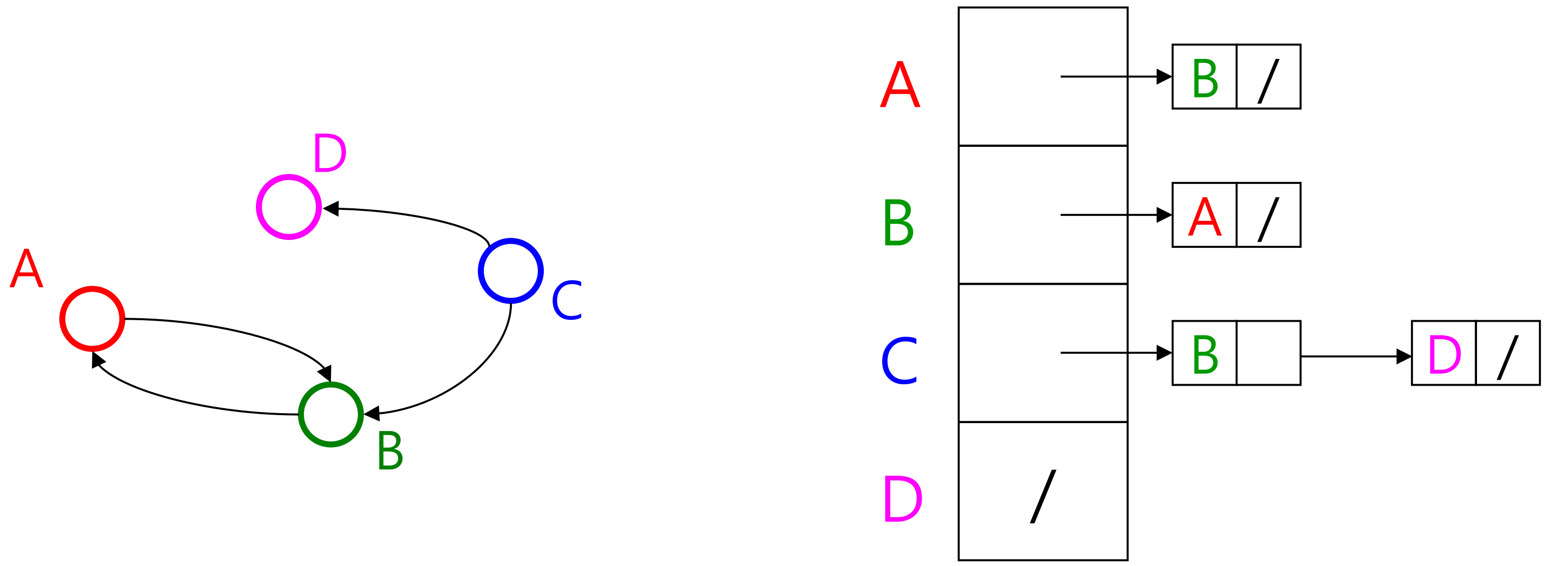
```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_weighted_edges_from([("A", "B", 1.5), ("B", "A", 1.5), ("C", "B", 3), ("C", "D", 2.5)])
# 2D array adjacency matrix
A = nx.adjacency_matrix(G)
A_dense = A.todense()
print(A_dense)
# Pandas format of adjacency matrix
nx.to_pandas_adjacency(G)
```

```
[[0.  1.5 0.  0. ]
 [1.5 0.  0.  0. ]
 [0.  3.  0.  2.5]
 [0.  0.  0.  0. ]]
```

	A	B	C	D
A	0.0	1.5	0.0	0.0
B	1.5	0.0	0.0	0.0
C	0.0	3.0	0.0	2.5
D	0.0	0.0	0.0	0.0



- Assign each node a number from 0 to |V|-1
  - A list (or array) of length |V| in which each entry stores a list of all adjacent vertices (e.g., linked list)

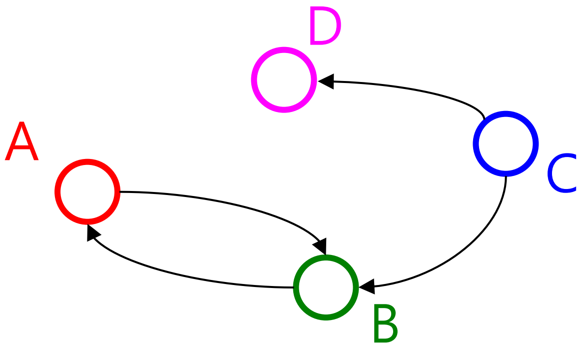


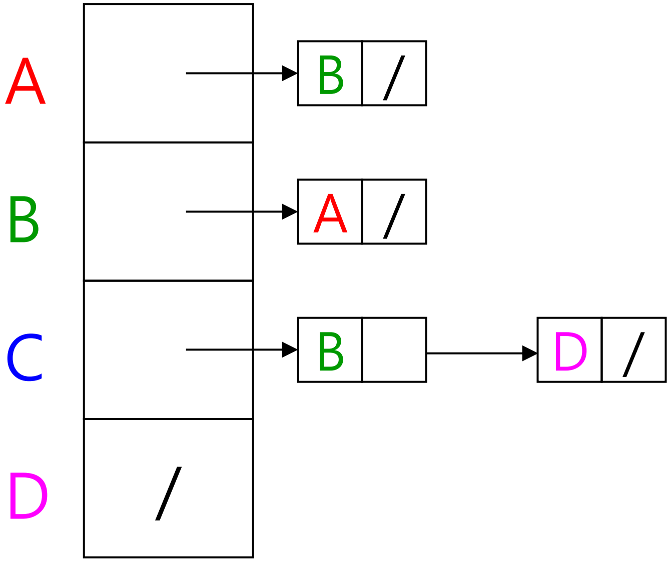
Adjacency List

➢ Sample Code

```
# Instantiate the graph
G = nx.DiGraph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])
adjacency_list = nx.generate_adjlist(G)
for line in adjacency_list:
    print(line)
```

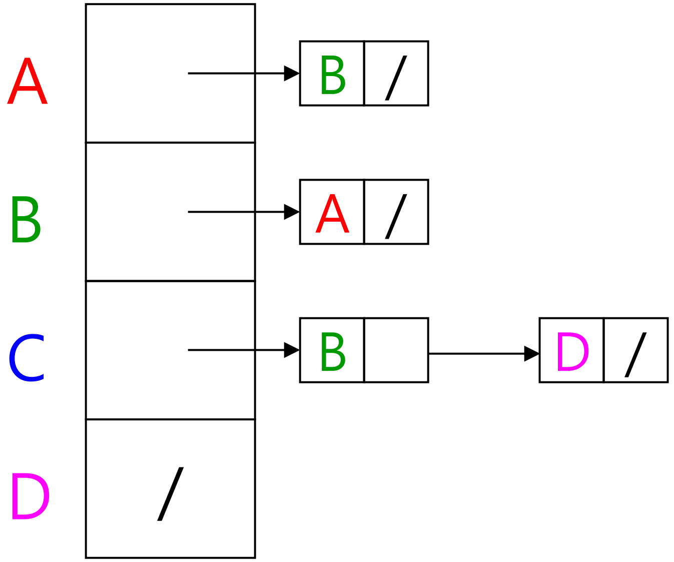
```
A B
B A
C B D
D
```





Adjacency List Properties

- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for sparse or dense graphs?



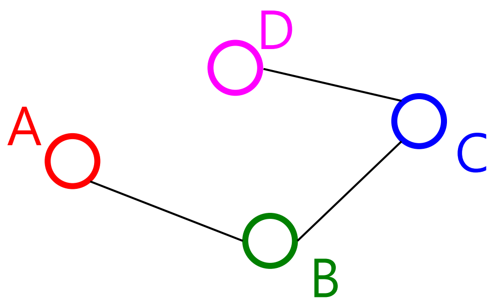
Adjacency List Properties

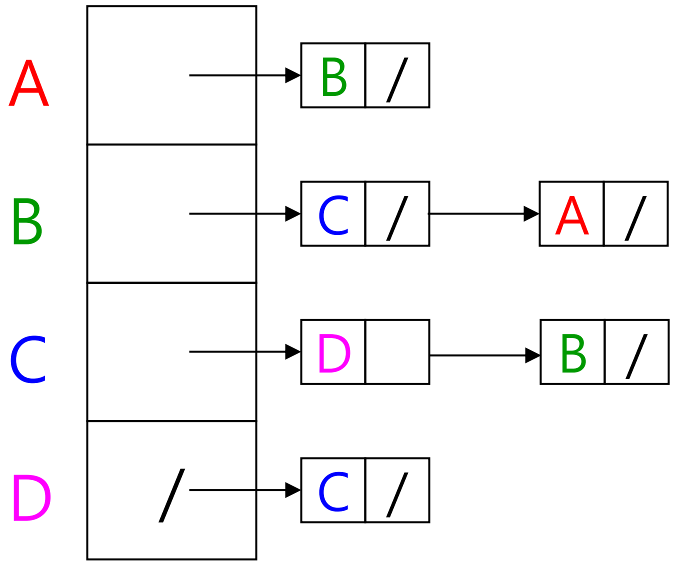
- Running time to:
  - Get a vertex's out-edges: O(d) where d is out-degree of vertex
  - Get a vertex's in-edges: O(|E|) (could keep a second adjacency list for this!)
  - Decide if some edge exists: O(d) where d is out-degree of source
  - Insert an edge: O(1) (unless you need to check if it's already there)
  - Delete an edge: O(d) where d is out-degree of source

- Space requirements: O(|V|+|E|)

- Best for sparse or dense graphs? sparse

- Adjacency lists also work well for undirected graphs
  - Put each edge in two lists to support efficient "get all neighbors"



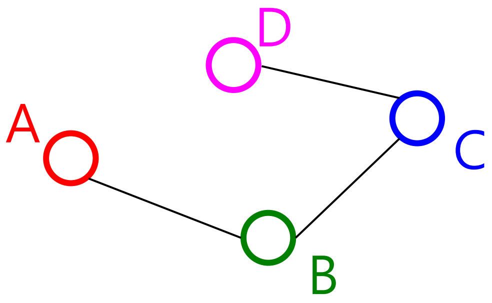


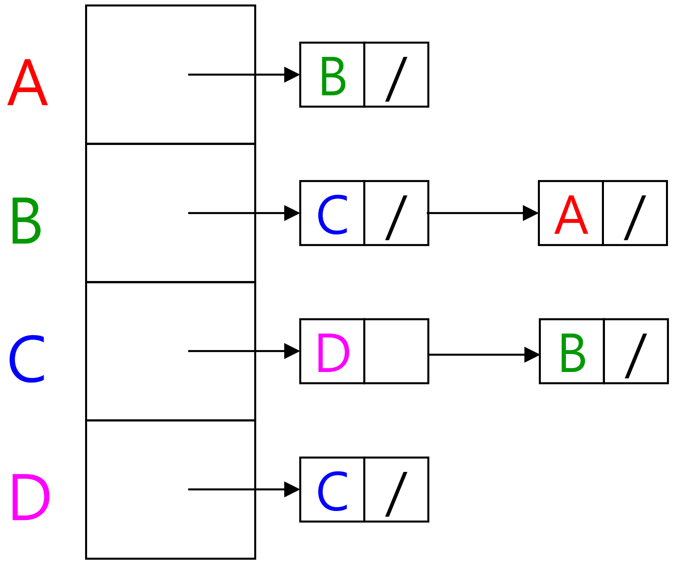
➤ Sample code:

```
# Instantiate the graph
G = nx.Graph()
# add node/edge pairs
G.add_edges_from([("A", "B"), ("B", "A"), ("C", "B"), ("C", "D")])
# Adjency list
adjacency_list = nx.generate_adjlist(G)
for line in adjacency_list:
    print(line)

nx.to_pandas_adjacency(G)
```

```
A B
B C
C D
D
```

	A	B	C	D
A	0.0	1.0	0.0	0.0
B	1.0	0.0	1.0	0.0
C	0.0	1.0	0.0	1.0
D	0.0	0.0	1.0	0.0





Which is better?

- Graphs are often sparse
  - Streets form grids
  - Airlines rarely fly to all cities
- Adjacency list should generally be your default choice
  - Slower performance compensated by greater space savings

	Dense Matrix	Sparse Matrix
Representation	2D (V x V)	Store only non-zero elements
Implementation	Simple	Complex
Memory Usage	High	Low
Suitable Graph Size	Small	Large
Use Case	Small, fully connected graphs	Social networks, web graphs, road networks

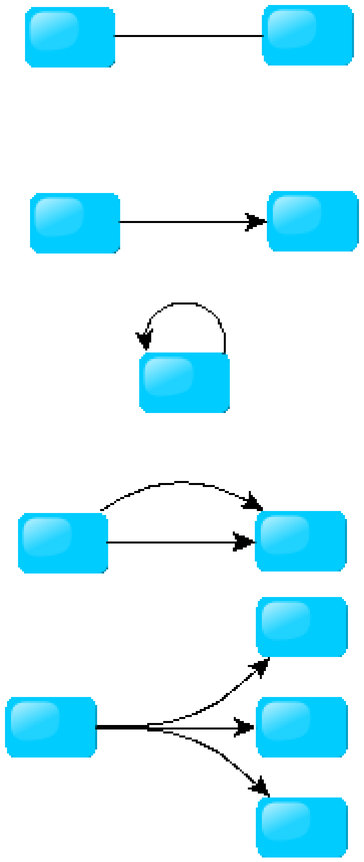
Which is better?

- Adjacency matrix: use when graph is dense (many edges)
- Adjacency list: use when graph is sparse (few edges) to save memory

	Adjacency Matrix	Adjacency List
Space Complexity	O(V^2)	O(V + E)
Edge Lookup	O(1)	O(degree(V))
Neighbour Iteration	O(V)	O(degree(V))
Best	Dense graph	Sparse graph

Different types of graph

- A database should store and present all types of graph
  - Undirected Graph
  - Directed Graph
  - Pseudo Graph
  - Multi Graph
  - Hyper Graph

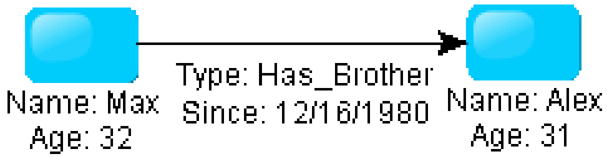


More kinds of graph

- Weighted Graph

- Labeled Graph

- Property Graph

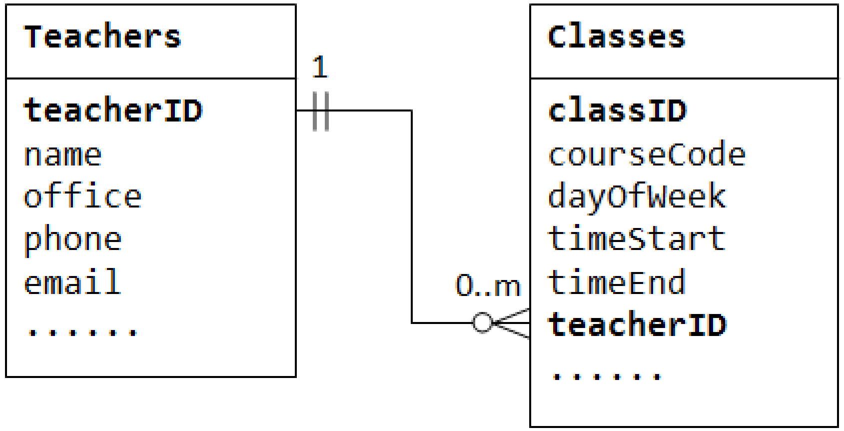


What is a graph database?

- A database with an explicit graph structure
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Plus, an Index for lookups

Relational database

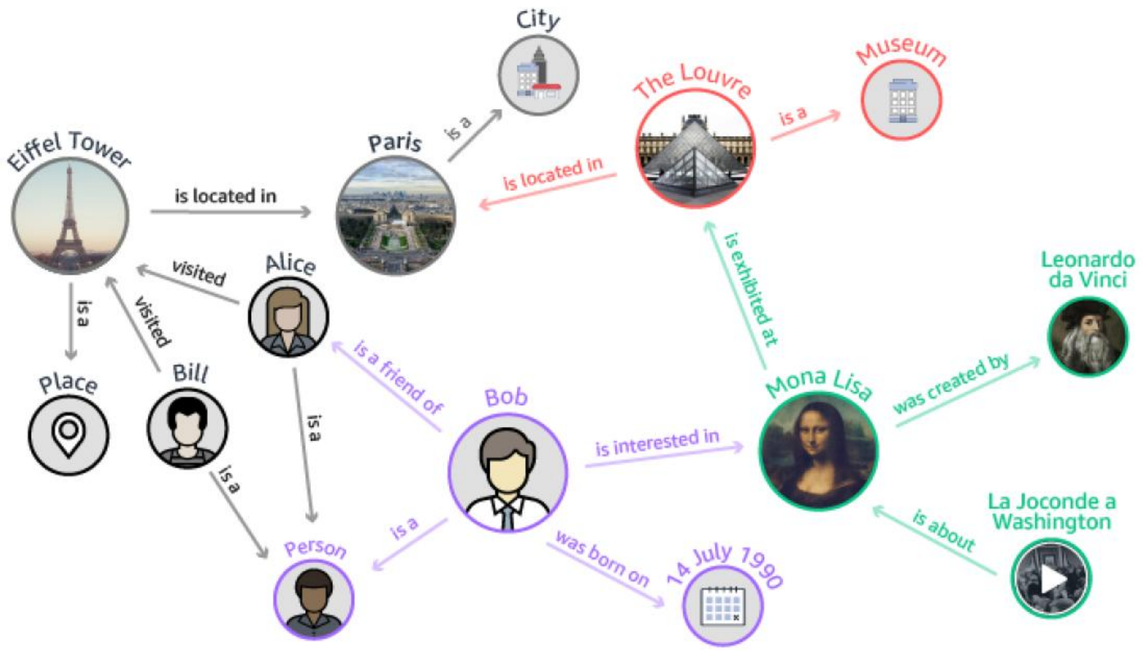
- A relational database is a collection of information that organizes data in predefined relationships where data is stored in one or more tables (or "relations") of columns and rows, making it easy to see and understand how different data structures relate to each other.
- Relationships are a logical connection between different tables, established on the basis of interaction among these tables.



https://www3.ntu.edu.sg/home/ehchua/programming/sql/Relational_Database_Design.html

Graph database

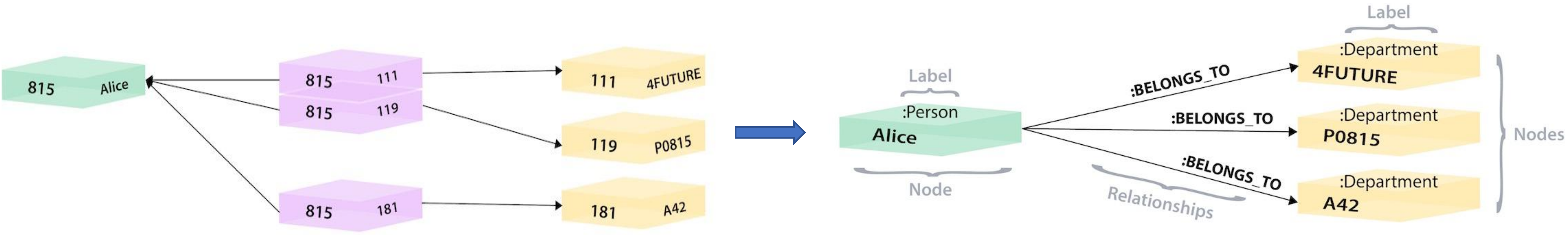
- A graph database stores nodes and relationships instead of tables, or documents. Data is stored just like you might sketch ideas on a whiteboard.
- The data is stored without restricting it to a pre-defined model, allowing a very flexible way of thinking about and using it.



https://olab.ca/heterogeneous-graphs-and-graph-neural-networks/

Graph database: An example

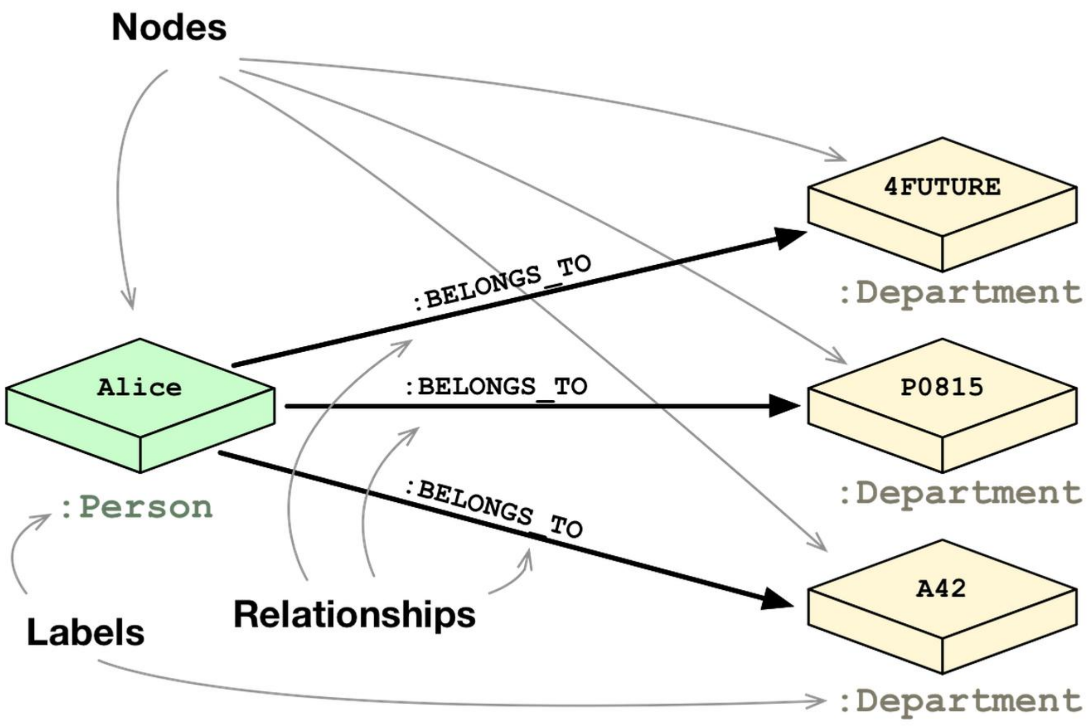
- To find the user Alice and her person ID of 815.
  - we search the Person-Department table (orange middle table) to locate all the rows that reference Alice's person ID (815).
- Once we retrieve the 3 relevant rows, we go to the Department table on the right to search for the actual values of the department IDs (111, 119, 181).
- Now we know that Alice is part of the 4Future, P0815, and A42 departments.



Relational database

Graph database

- Nodes: Alice, 4FUTURE, P0815, A42
- Lables: Person, Department
- Relationships: BELONGS_TO



Popular GraphDB Engines

IBM Graph™

DATASTAX

Cayley

OrientDB®

neo4j

The #1 Database for Connected Data

ORACLE

Dgraph

ArangoDB

Graph Engine
SERVING BIG GRAPHS IN REAL-TIME

TITAN

- **Pros:**
  - Runs complex distributed queries
  - Scales out through sharded storage
  - Returns data natively in JSON, making it ideally suited for web development
  - Written on top of GraphQL

- **Cons:**
  - No native windows installation
  - No support for windows in a production environment

OrientDB

- **Pros:**
  - Multi model DB – both graph and document DB
  - Easily add users/roles
  - Supports multiple databases

- **Cons:**
  - No native windows service installation
  - Requires more schema design up front

- **Pros:**
  - Runs on Windows natively - in either a console or as a service
  - 24/7 production support since 2003 – Mature
  - Large and active user community

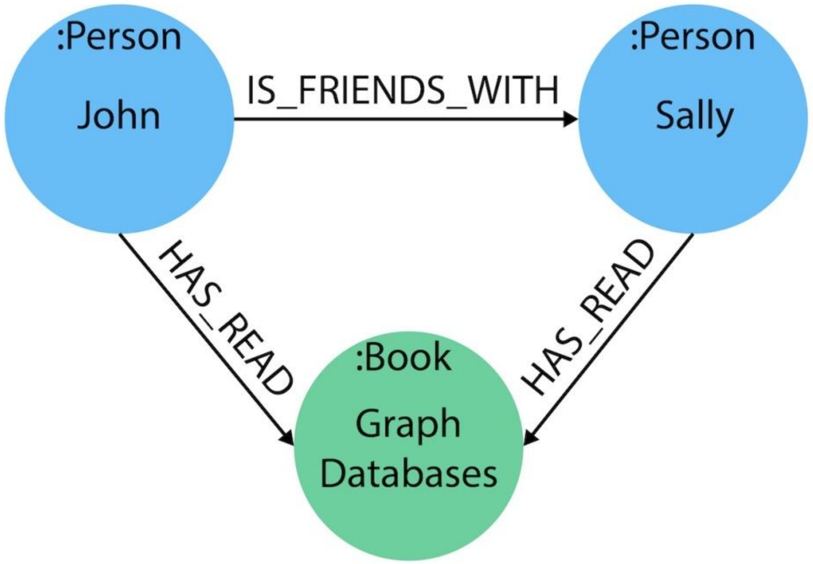
- **Cons:**
  - Only one Project can be running on one port at a time

What does Neo4j provide?

- Full ACID (atomicity, consistency, isolation, durability)
- REST API
- Property Graph
- Lucene Index
- High Availability (with Enterprise Edition)

You can't model it in SQL

- The SQL becomes more complex as the length of the relationships increase
- Performance on the joins becomes an issue quickly
- SQL is not well-suited to model rich domains
- It's not easy to start at one row and follow relevant relationships along a path

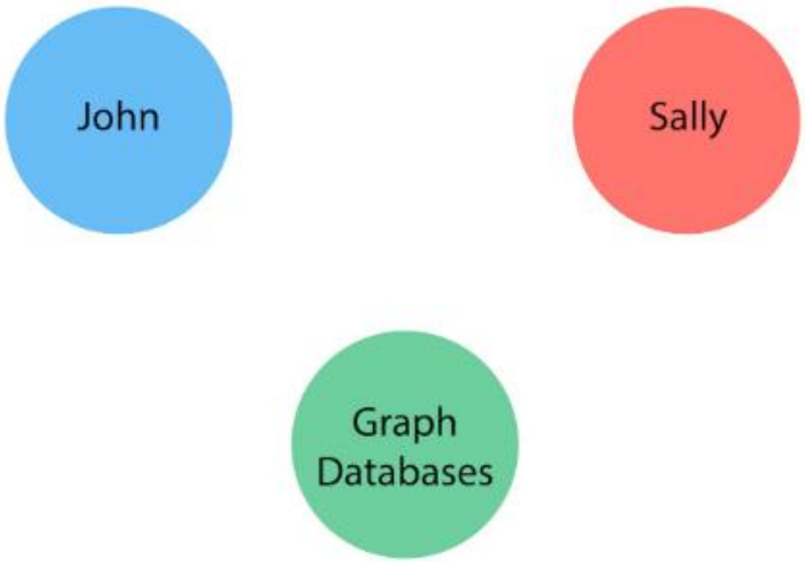


Two people, Sally and John, are friends.
Both John and Sally have
read the book, Graph Databases.

https://neo4j.com/developer/guide-data-modeling/

Neo4J Model: Example

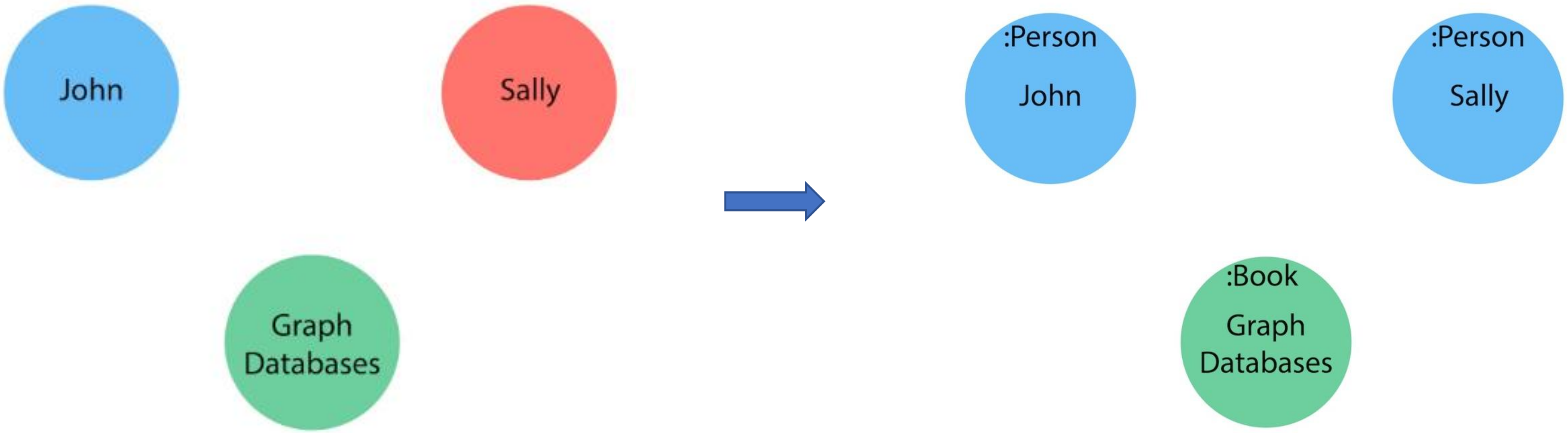
- Two *people*, **Sally** and **John**, are friends. Both **John** and **Sally** have read the *book*, **Graph Databases**.
  - Extracting the nodes: John, Sally, Graph Databases



https://neo4j.com/developer/guide-data-modeling/

Neo4J Model: Example

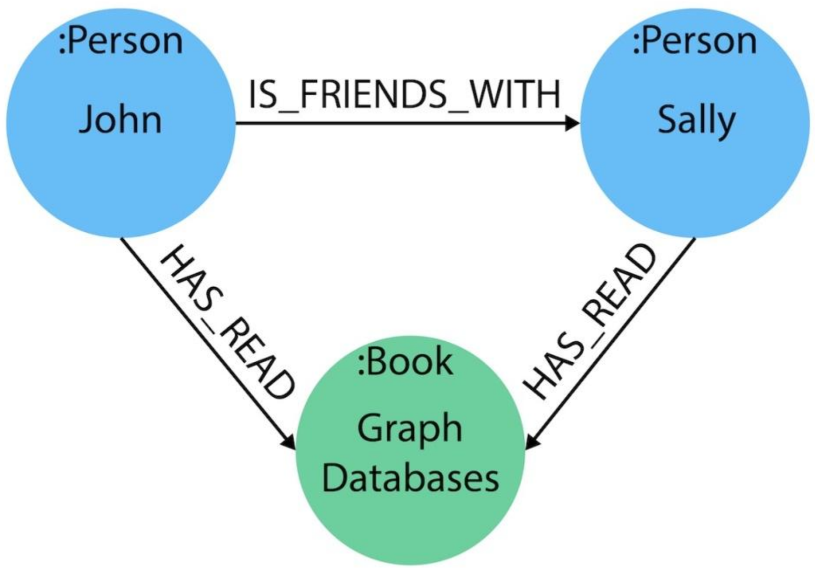
- Two *people*, **Sally** and **John**, are friends. Both **John** and **Sally** have read the *book*, **Graph Databases**.
  - Extracting the nodes: John, Sally, Graph Databases
  - Extracting the labels



https://neo4j.com/developer/guide-data-modeling/

Neo4J Model: Example

- Two *people*, **Sally** and **John**, <u>are friends</u>. Both **John** and **Sally** <u>have read</u> the *book*, **Graph Databases**.
  - Extracting the nodes: John, Sally, Graph Databases
  - Extracting the labels
  - Defining Relationships:
    - John <u>is friends with</u> Sally
    - Sally <u>is friends with</u> John
    - John <u>has read</u> Graph Databases
    - Sally <u>has read</u> Graph Databases



- Sample scripts:
  - Two people, John and Sally, are friends. Both John and Sally have read the book, Graph Databases.
  - Create Nodes:

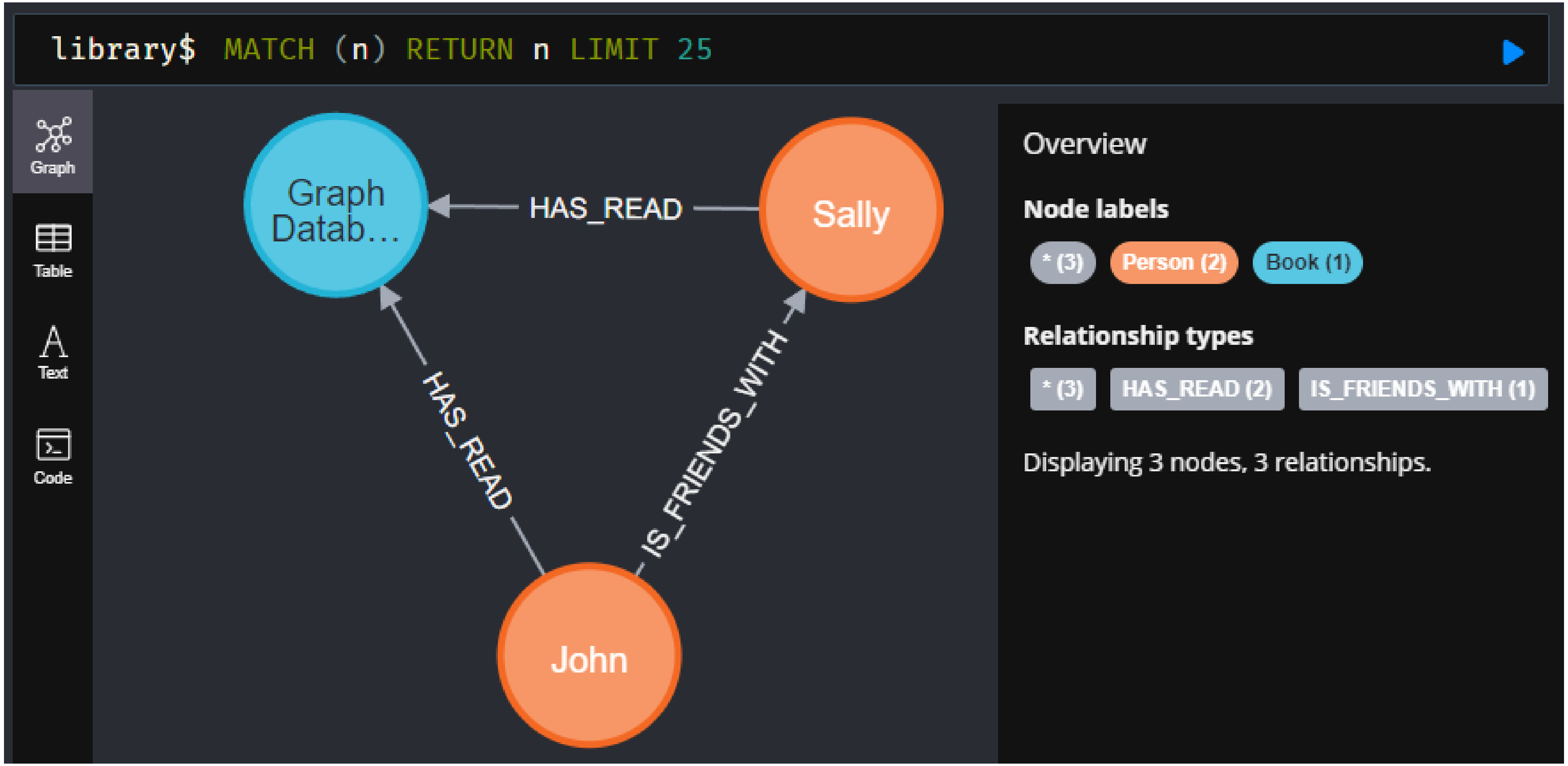
```
CREATE(:Person{name:'John',born:'Mar 8, 1998',linkedin:'@john'})
CREATE(:Person{name:'Sally',born:'Oct 16, 1997',linkedin:'@sali'})
CREATE(:Book{name:'GraphDatabases',published_date:'Nov 16, 2015'})
```

  - Generate relationships:

```
MATCH (a:Person), (b:Person) WHERE (a.name = 'John' AND b.name = 'Shally') CREATE (a)-[r
1:IS_FRIENDS_WITH]->(b);
MATCH (a:Person), (b:Book) WHERE (a.name = 'John' OR a.name = 'Shally') AND b.name = '
GraphDatabases' CREATE (a)-[r2:HAS_READ]->(b);
```

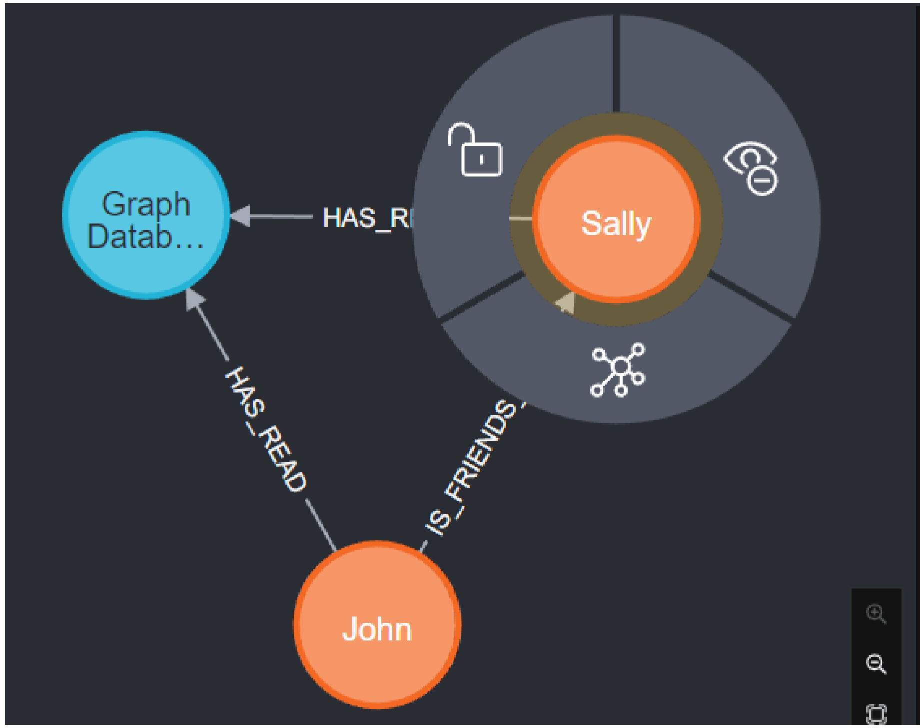
Example: Neo4j database

➢ Show database.



Example: Neo4j database

- Node properties.



Node properties

Person

<id>	2
born	Oct 16, 1997
linkedin	@sali
name	Sally

Person

<id>	0
born	Mar 8, 1998
linkedin	@john
name	John

Book

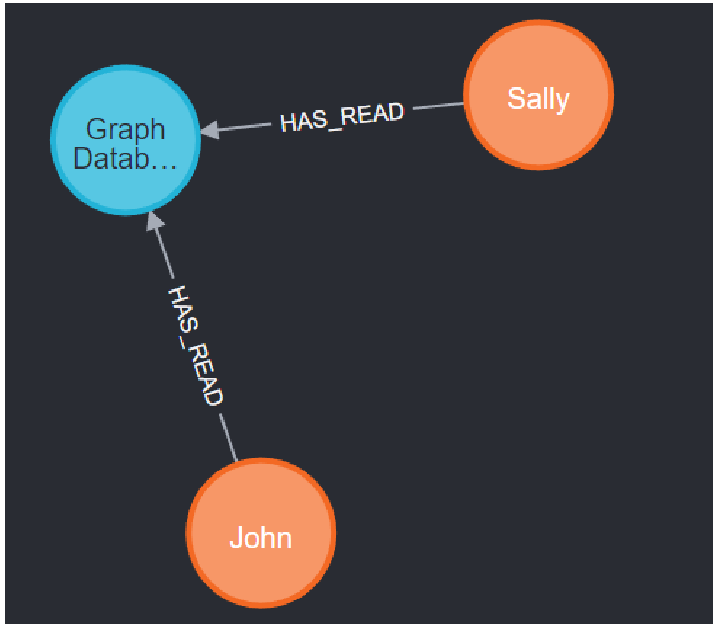
<id>	4
name	Graph Databases
published_date	Nov 16, 2015

- Delete relationships

```
MATCH (n:Person {name: 'John'})-[r:IS_FRIENDS_WITH]→()
DELETE r
```

Table
Code

Deleted 1 relationship, completed after 27 ms.



Example: Neo4j database

- Delete a node: to delete a node, all the relationships have to be removed.

```
MATCH (n:Person {name: 'John'})-[r:HAS_READ]->()
DELETE r
```

Table

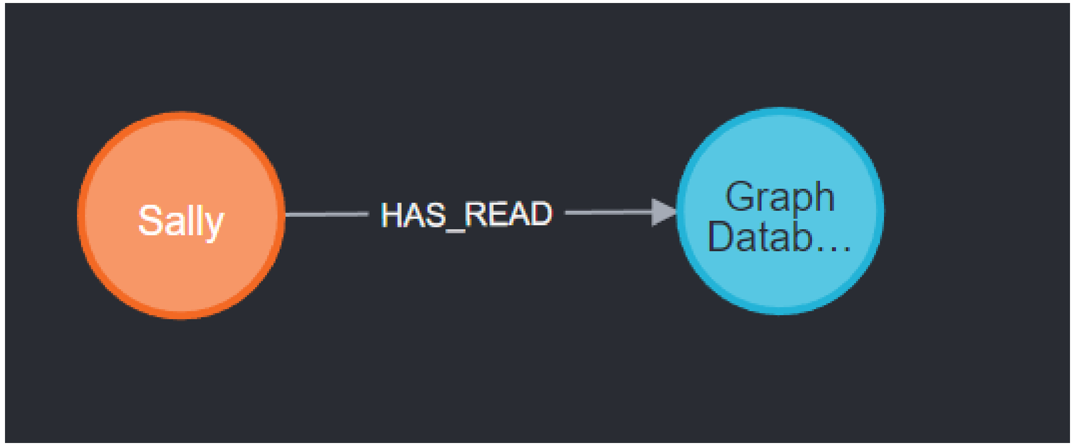
Deleted 1 relationship, completed after 2 ms.

- Remove a node:

```
MATCH (n:Person {name: 'John'})
DELETE n
```

Table

Deleted 1 node, completed after 1 ms.



네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA