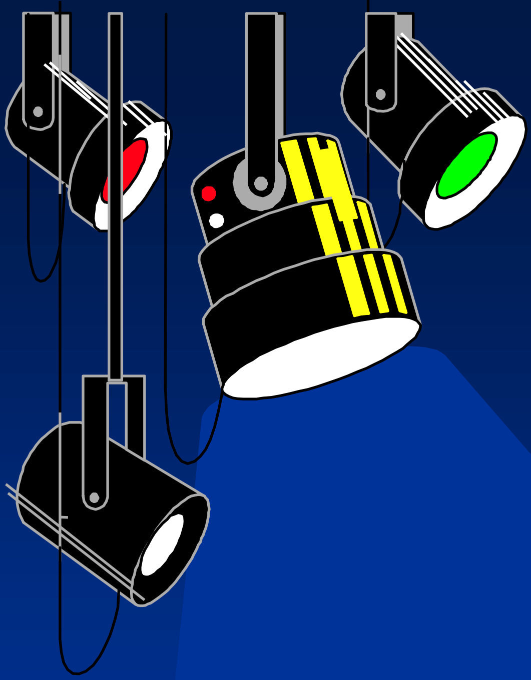


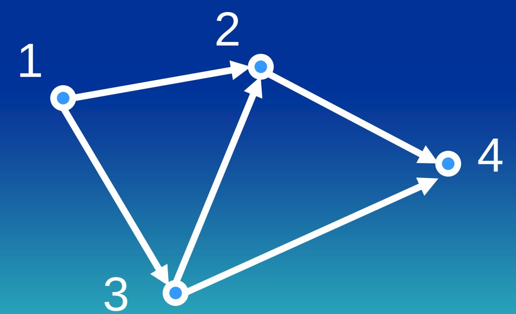
Outline

- Graph Databases
  - Graph database types
  - Graph storage on disk
  - Retrieval of a graph database
  - Maintenance of a graph database
    - Adding a new node

- **What is a graph database**
  - A graph database is defined as a specialized, single-purpose platform for creating and manipulating graphs.
  - Graphs contain nodes, edges, and properties, all of which are used to represent and store data in a way that relational databases are not equipped to do.

In fact, a graph can be stored as a relation. But some operations cannot be efficiently supported.

G:



$\Rightarrow$

G:

parent	child
1	2
1	3
2	4
3	2
3	4

- Considering a reachability query, by which we will check whether a node *v* is reachable from another node *u* through a path.
- For this, we will make a series of join operations. This is very time consuming.
- For example, to check whether node 4 is reachable from node 1, we will do a join between table *G* and itself.

parent	child
1	2
1	3
2	4
3	2
3	4

$\bowtie$
Child = parent

parent	child
1	2
1	3
2	4
3	2
3	4

=

ancestor	desc.
1	4
1	2
1	4
3	4

- In general, it is not efficient since to find a path containing *k* edges *k* joins have to be conducted.

- It is very time consuming for a large graph.

- **What is a graph database**
  - Graph analytics is another commonly used term, and it refers specifically to the process of analyzing data in a graph format using data points as nodes and relationships as edges.
  - Graph analytics requires a database that can support graph formats: this could be a dedicated graph database, or a converged database that supports multiple data models, including graphs.

- **Graph database types**
  - There are two popular models of graph databases: property graphs and RDF graphs (Resource Description Framework).
  - The property graph focuses on analytics and querying.
  - The RDF graph emphasizes data integration.
  - Both types of graphs consist of a collection of points (vertices) and the connections between those points (edges). But there are differences as well.

- **Property graphs**
  - Property graphs are used to model relationships among data, and they enable query and data analytics based on these relationships.
  - A property graph has vertices that can contain detailed information about a subject, and edges that denote the relationship between the vertices. The vertices and edges can have attributes, called properties, with which they are associated.

- **Property graphs**
  - Example

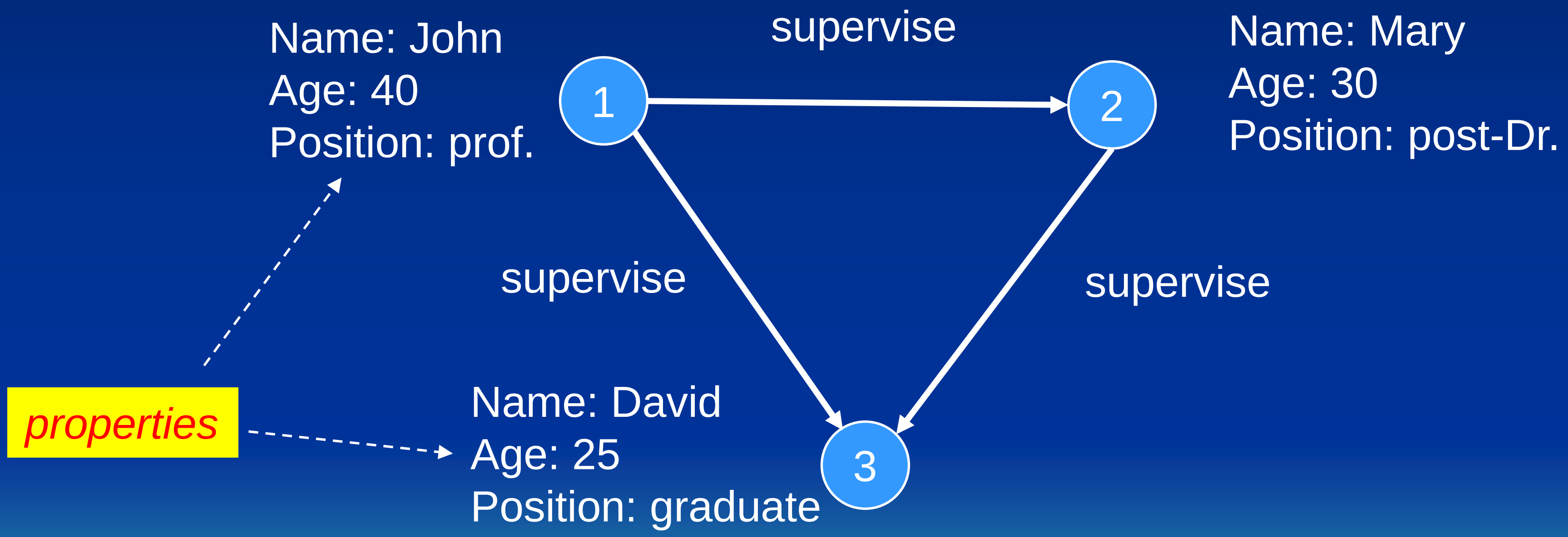


Figure 1

- **RDF**

- RDF graphs (RDF stands for Resource Description Framework) conform to a set of W3C (Worldwide Web Consortium) standards designed to represent statements and are best for representing complex metadata and master data.
- They are often used for linked data, data integration, and knowledge graphs.
- They can represent complex concepts in a domain, or provide rich semantics and inferencing on data.

- **RDF**

- In the RDF model a statement is represented by three elements: two vertices connected by an edge reflecting the subject, predicate and object of a sentence:

  <subject><predicate><object>.

  This is known as an *RDF triple*.

- Every vertex and edge is identified by a unique URI, or Unique Resource Identifier.

- The RDF model provides a way to publish data in a standard format with well-defined semantics, enabling information exchange.

- Government statistics agencies, pharmaceutical companies, and healthcare organizations have adopted RDF graphs widely.

- **RDF**
  - Example

In an RDF graph, each edge represents a statement.

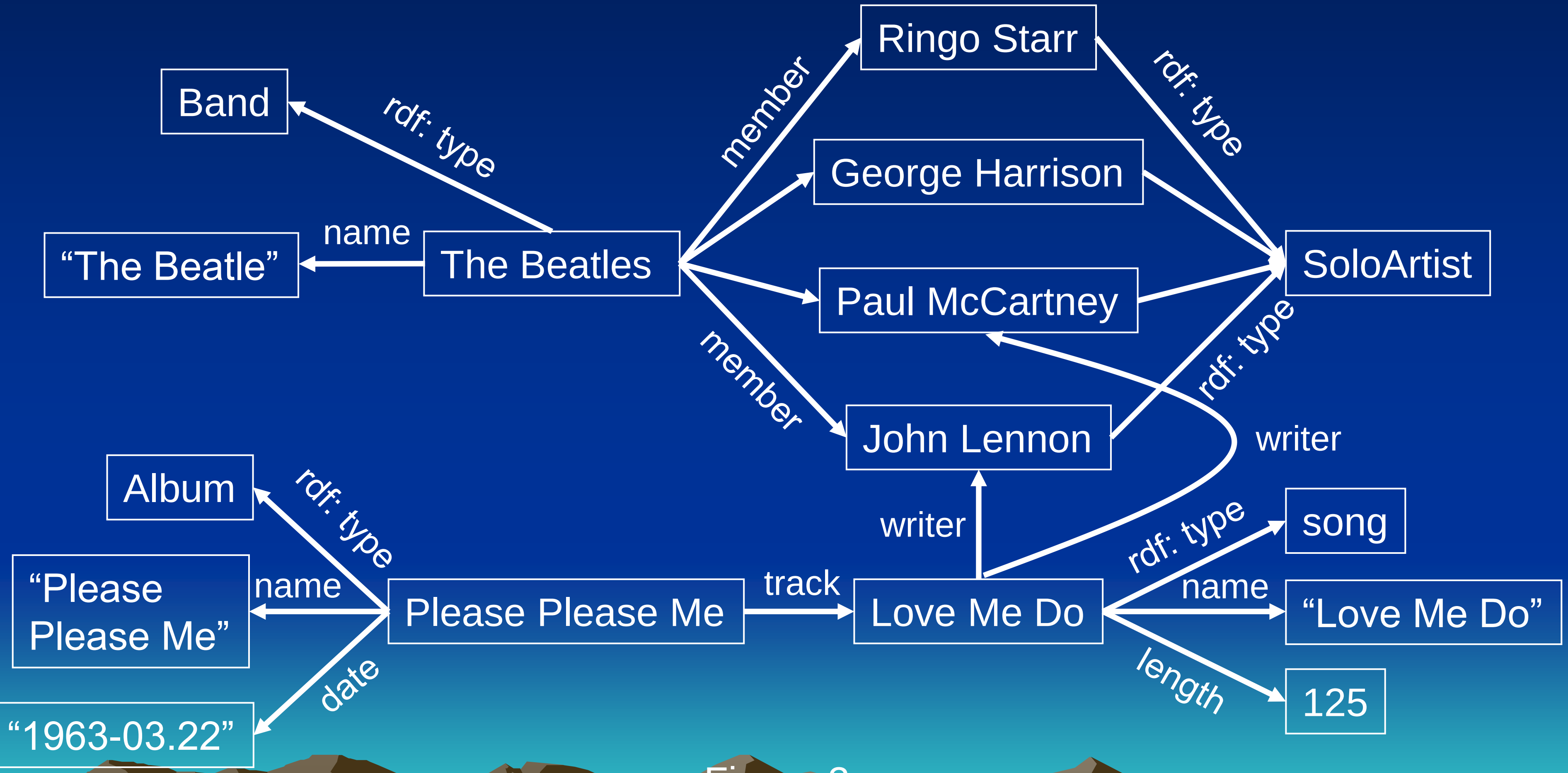
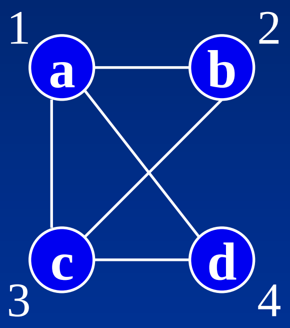


Figure 2

- This graph shows several nodes that represent entities such as the Beatles band and one of their studio albums. Each edge has an identifier that tells us what relationship holds between those nodes. For example, the member edge links bands to its members. The rdf: type edge represents a special kind of relationship.
- In this graph we also have nodes representing datatype values (i.e., “literals”) such as strings, numbers, dates. The corresponding edges are sometimes called “attributes” of the node and are often used to represent the characteristics of the nodes.
- This simple, flexible data model has a lot of expressive power to represent complex situations, relationships, and other things of interest, while also being appropriately abstract, i.e., it does not expose very much implementation detail in the same way as, say, the relational data model used with SQL.

- **Graph storage in main memory**

  - Storing a graph as a matrix

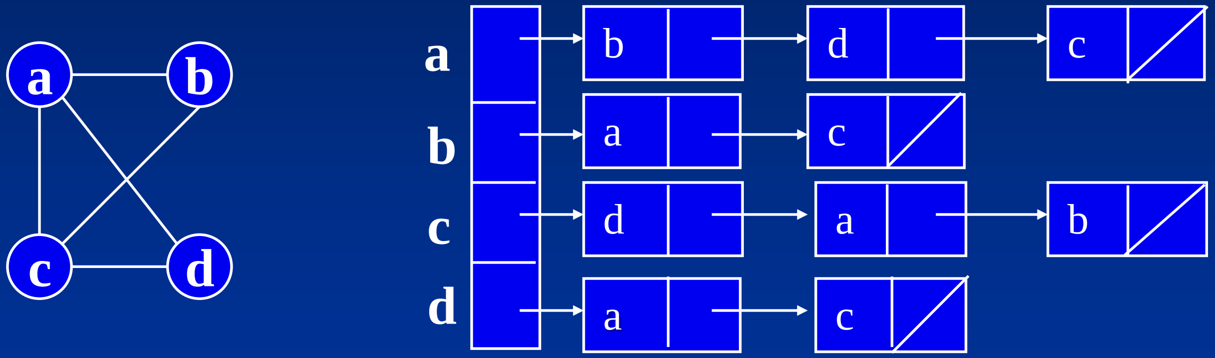


$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 1 & 0 & 1 & 0 \end{array}$$

  - Advantage: determining whether an edge $\in G$ is efficient
  - Disadvantage: space requirement - $\Theta(|V|^2)$ (not space-efficient)

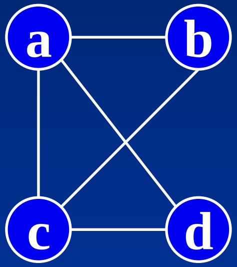
- **Graph storage in main memory**

  - Storing a graph as a linked list



  - Advantage: space requirement - **Θ(|*E*|)** (space-efficient)
  - Disadvantage: determining whether an edge ∈ *G* is not efficient

- **Graph storage in main memory**
  - Storing a graph as a two-dimensional array

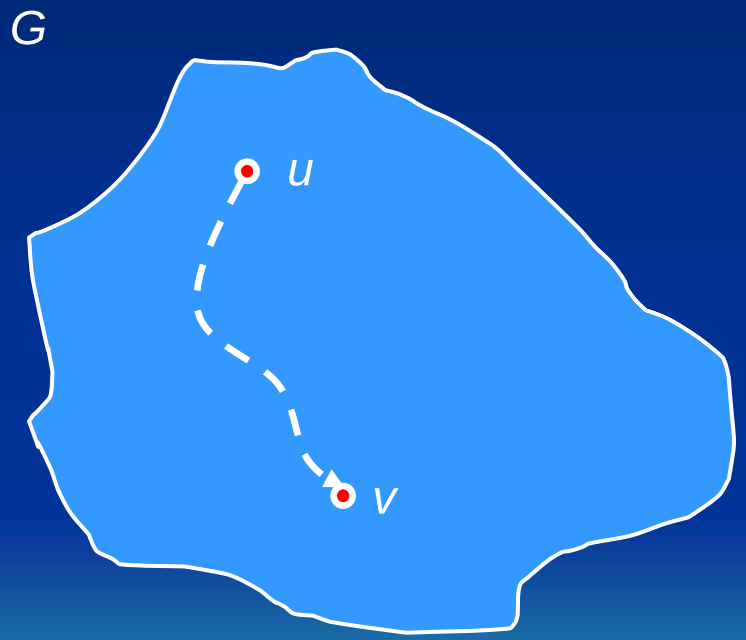


Start node	End node
a	b
a	c
a	d
b	c
c	d

  - Advantage: space requirement - $\Theta(|E|)$ (space-efficient)
  - Disadvantage: determining whether an edge $\in G$ is not efficient

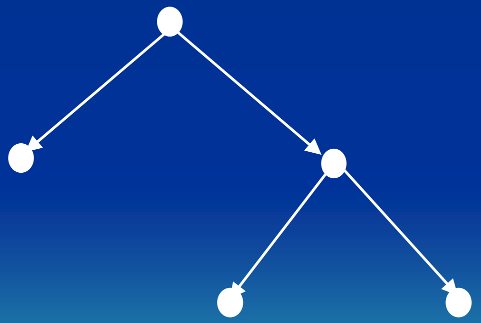
- **Reachability queries**

  - Given a directed graph *G*, check whether a node *v* is reachable from another node *u* through a path in *G*.

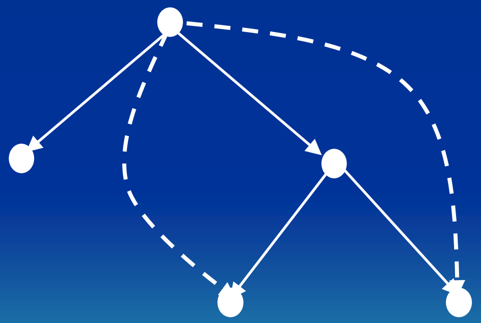


- **Reachability queries**
  - To evaluate reachability queries, we need to compute the transitive closure of *G*.
  - The transitive closure *G** of a graph *G* is a graph such that there is an edge (*u*, *v*) in *G** iff there is path from *u* to *v* in *G*.

G:



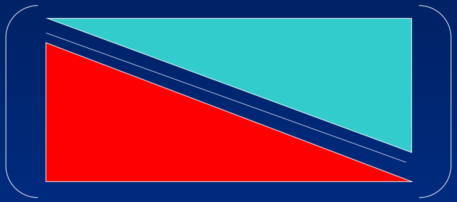
*G**:



- **If *G* is stored as a matrix, we can use the Warren's algorithm to compute *G**.**

Warren's algorithm is a quite simple way to generate a boolean matrix to represent the transitive closure of a graph *G*. Assume that *G* is represented by a boolean matrix *M* in which *M*(*i*, *j*) = 1 if edge (*i*, *j*) is in *G*, and *M*(*i*, *j*) = 0 if (*i*, *j*) is not in *G*. Then, the matrix *M'* for the transitive closure of *G* can be computed from *M*, in which *M'*(*i*, *j*) = 1 if there exits a path from *i* to *j* in *G*, and *M'*(*i*, *j*) = 0 if there is no path from *i* to *j* in *G*.

Warren's algorithm is given below:



```
Algorithm Warren
for $i$ = 2 to $n$ do
    for $j$ = 1 to $i$ - 1 do
    {if $M(i, j) = 1$ then set $M(i, *) = M(i, *) \vee M(j, *)$;}
for $i$ = 1 to $n$ - 1 do
    for $j$ = $i$ + 1 to $n$ do
    {if $M(i, j) = 1$ then set $M(i, *) = M(i, *) \vee M(j, *)$;}
```

In the algorithm, $M(i, *)$ denotes row i of M.
The theoretic time complexity of Warren's algorithm is $O(n^3)$.

- If *G* is stored as a two-dimensional array, a series of join operations have to be conducted to compute *G**.

Start node	End node
a	b
a	c
a	d
b	c
c	d

$\bowtie$

Start node	End node
a	b
a	c
a	d
b	c
c	d

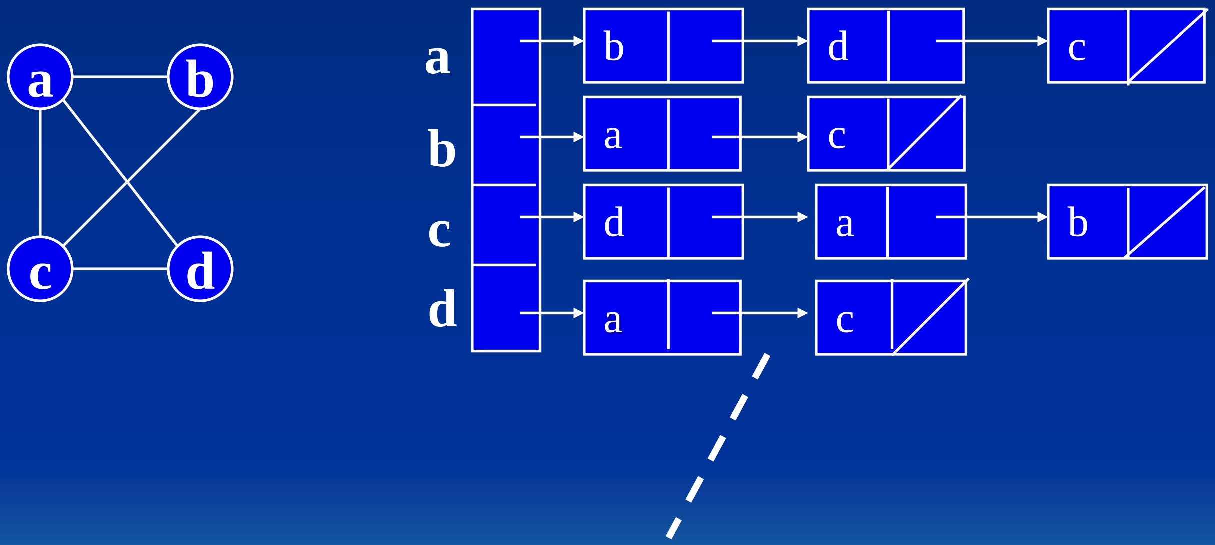
$\bowtie$

Start node	End node
a	b
a	c
a	d
b	c
c	d

… …

The theoretic time complexity of the operation is $O(n^k)$, where k is the number of joins. In the worst case, $k = O(n)$.

- **If *G* is stored as a linked list, it can be only kept in main memory. Not permanent storage. That is, if the computer turns off, the linked list disappears.**



Each link is just a memory unit address.

How can we store a graph as a linked list permanently on hard disk?

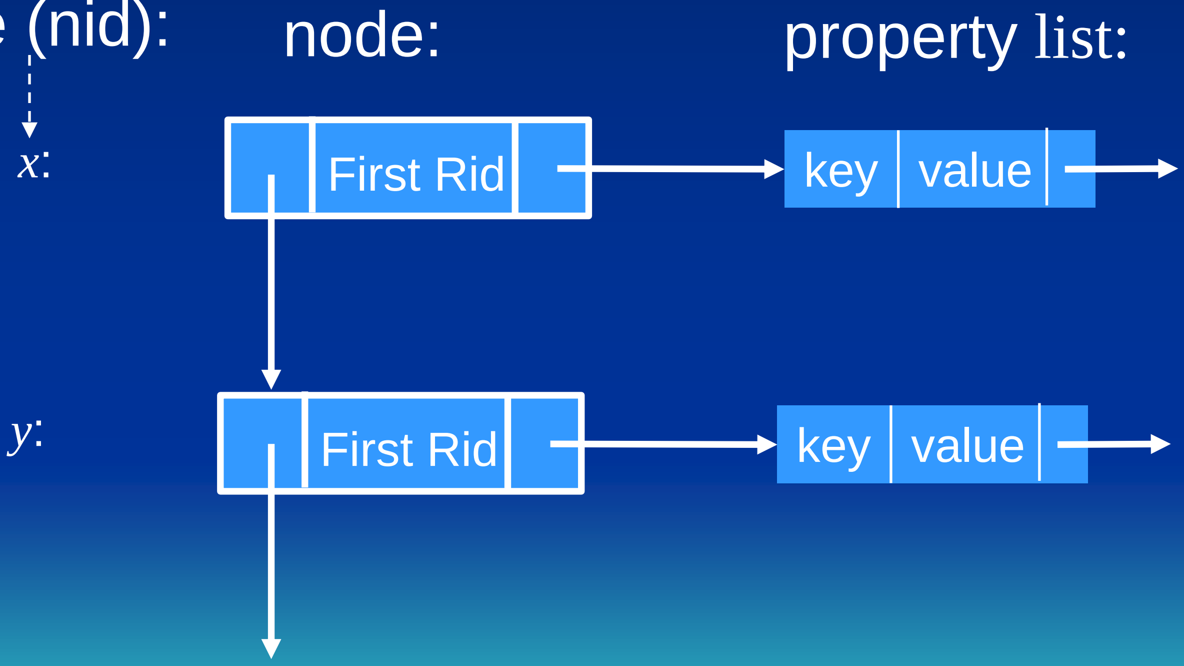
- **Graph storage on disk (Neo4j)**
  - Since a graph database is a schema-less database, we use fixed record lengths to represent persist data and follow offsets in these files to know how to fetch data to answer queries. The following table illustrates the fixed sizes Neo4j uses for the type of Java objects being stored:

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

- **Graph storage on disk**
  - It all boils down to linked lists of fixed size records on disk.
  - Properties are stored as a linked list of property records, each holding key+value.
  - Each node/relationship references its first property record.
  - The Nodes also reference the first relation instance in its relationship chain.
  - Each Relationship references its start and end node. It also references the prev/next relationship record for the start/end node respectively.

- **Node storage on disk**
  - In the file for storing nodes, each node storage is of the same length.

node address
in file (nid):



- **Relation storage on disk**
  - In the file for storing relationships, each relationship storage is of the same length.

relationship address
in file (rid):

r:	src	dest	src_prev_rid	src_next_rid	dst_prev_rid	dst_next_rid

- **Relation storage on disk**
  - Example

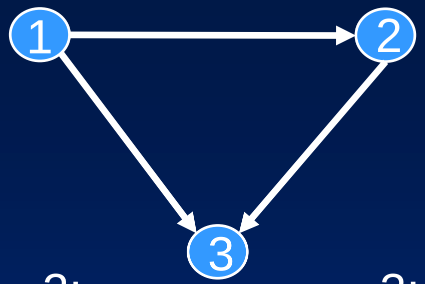
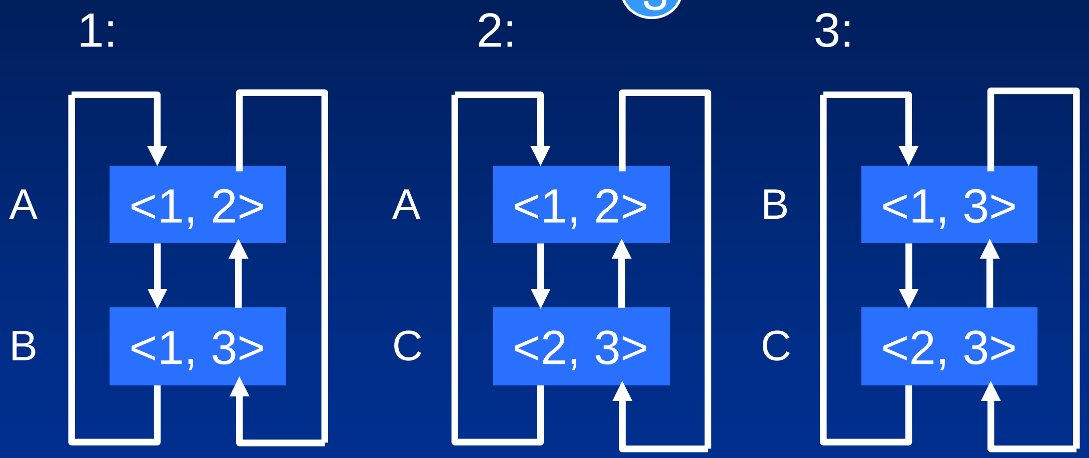


Figure 3

	src	dest
A	1	2
B	1	3
C	2	3

(a)



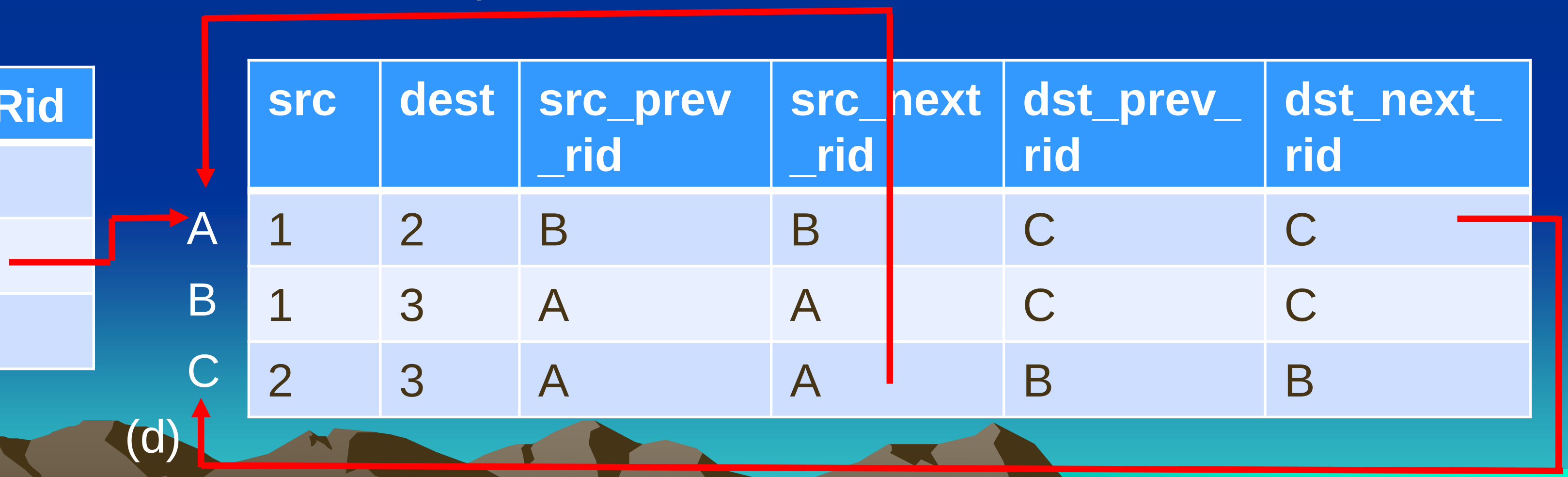
(b)

Node:

	First-Rid
1:	A
2:	A
3:	B

(c)

Relationship:



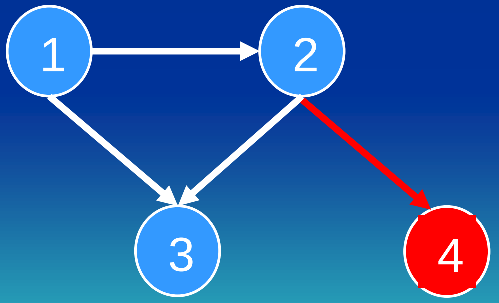
	src	dest	src_prev_rid	src_next_rid	dst_prev_rid	dst_next_rid
A	1	2	B	B	C	C
B	1	3	A	A	C	C
C	2	3	A	A	B	B

(d)

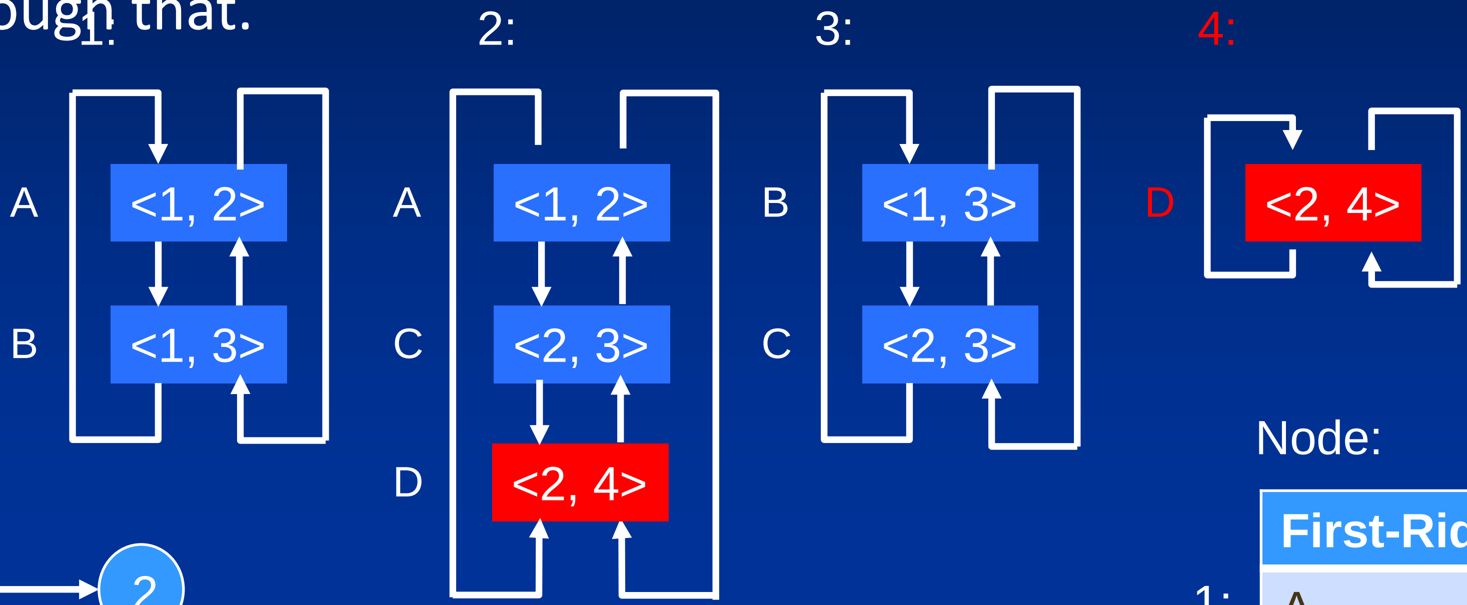
- Figure 3 (b) is a conceptual model. It lists all of the edges associated with each node and organizes them as doubly linked lists. An edge is associated with a node if the node is either the source or the destination of that edge. For example, edge A <1,2> is associated with both node 1 and node 2.
- Figure 3 (c) is the actual nodes storage. Each record stores the first relationship ID (first_rid) of a node. For example, the first relationship for both node 1 and 2 is A.
- Figure 3 (d) is the actual relationships storage. Each record stores the source (src) And destination (dest) of a relationship. In addition, it also stores the previous and next relationship IDs for both the source and destination nodes (src_prev_rid, src_next_rid, dst_prev_rid, dst_next_rid). This essentially preserves the conceptual doubly linked lists for both the source and destination but only stores a given edge once.

- **Retrieve a graph database**
  - Say you want to retrieve all the outbound relationships of node 2. See the red arrows in Figure 3 as an illustration. You first go to the nodes storage and see that the first_rid of node 2 is A.
  - Then you go to the relationships storage to look up A. A isn't an outbound relationship of 2 because 2 is the dst for A. But that's fine. You follow the dst_next_rid in A (because 2 is the destination in relationship A). That points to C. C is an outbound relationship for 2 (2 is the source in C) . You continue to go to the src_next_rid, which loops back to A. Then you know you've exhausted all of node 2's relationship.

- **Add a new node**
  - Now let's say you want to add a new node 4 and make 2 follow 4.
  - See Figure 4 for an overview of the change. Again, let's walk through that.



(a)



(b)

Node:

	First-Rid
1:	A
2:	A
3:	B
4:	D

(c)

Figure 4

Relation:

	src	dest	src_prev_rid	src_next_rid	dst_prev_rid	dst_next_rid
A	1	2	B	B	~~C~~ D	C
B	1	3	A	A	C	C
C	2	3	A	~~A~~ D	B	B
D	2	4	C	A	D	D

- A new record is added to the nodes storage to store the first_rid of node 4, which is D. In the relationships storage, a new record D is inserted. Its src and dst are 2 and 4, respectively.
- Its dst_prev_rid and dst_next_rid are trivial because its dst node 4 has has only one relationship, which is D itself. The slightly tricky part is to update the doubly linked list of relationships for node 2. I've highlighted them in red. It should be fairly straightforward to see how the red part for node 2 in Figure 4 (b) is reflected in the changed relation.