MI-PDB, MIE-PDB: **Advanced Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/

Lecture 9:

# Graph Databases, Neo4j, Cypher

19. 4. 2016

Lecturer: **Martin Svoboda**
svoboda@ksi.mff.cuni.cz

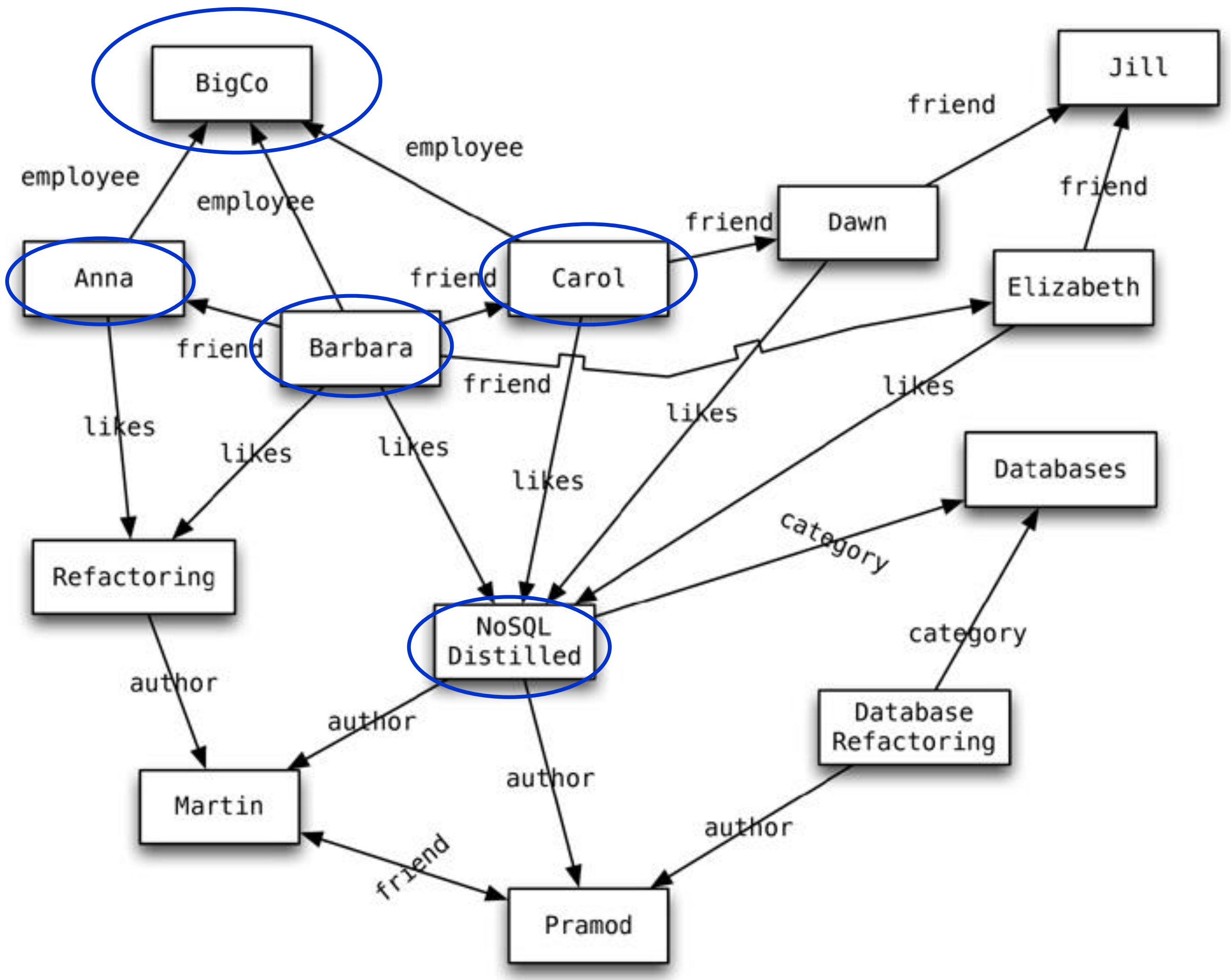
Author: **Irena Holubová**
Faculty of Mathematics and Physics, Charles University in Prague
Course NDBI040: **Big Data Management and NoSQL Databases**

# Graph Databases

## Basic Characteristics

- To store entities and relationships between these entities
  - Node is an instance of an object
  - Nodes have properties
    - e.g., name
  - Edges have directional significance
  - Edges have types
    - e.g., likes, friend, …
- Nodes are organized by relationships
  - Allow to find interesting patterns
  - e.g., "Get all nodes employed by Big Co that like NoSQL Distilled"

# Example:



BigCo
Jill
friend
employee
employee
employee
friend
Dawn
friend
Anna
friend
Carol
Elizabeth
friend
Barbara
friend
likes
likes
likes
likes
likes
likes
likes
Databases
Refactoring
category
NoSQL Distilled
category
author
author
Database Refactoring
Martin
author
author
friend
Pramod

# Graph Databases

## RDBMS vs. Graph Databases

- When we store a graph-like structure in RDBMS, it is for a single type of relationship
  - "Who is my manager"
  - Adding another relationship usually means schema changes, data movement etc.
  - In graph databases relationships can be dynamically created / deleted
    - There is no limit for number and kind
- In RDBMS we model the graph beforehand based on the Traversal we want
  - If the Traversal changes, the data will have to change
  - We usually need a lot of join operations
- In graph databases the relationships are not calculated at query time but persisted
  - Shift the bulk of the work of navigating the graph to inserts, leaving queries as fast as possible

# Graph Databases

## Suitable Use Cases

**Connected Data**

- Social networks
- Any link-rich domain is well suited for graph databases

**Routing, Dispatch, and Location-Based Services**

- Node = location or address that has a delivery
- Graph = nodes where a delivery has to be made
- Relationships = distance

**Recommendation Engines**

- “your friends also bought this product”
- “when invoicing this item, these other items are usually invoiced”

# Graph Databases

## When Not to Use

- When we want to update all or a subset of entities
  - Changing a property on all the nodes is not a straightforward operation
  - e.g., analytics solution where all entities may need to be updated with a changed property
- Some graph databases may be unable to handle lots of data
  - Distribution of a graph is difficult or impossible

# Graph Databases

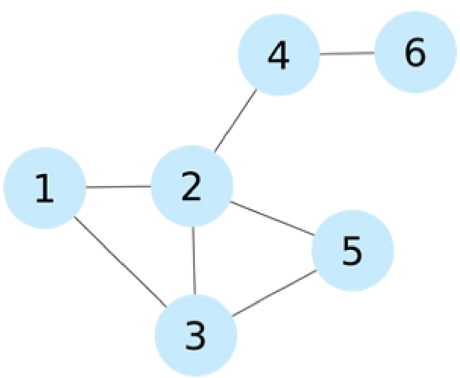
## Data structures and queries

- Data: a set of entities and their relationships
  - e.g., social networks, travelling routes, …
  - We need to efficiently represent graphs
- Basic operations: finding the neighbours of a node, checking if two nodes are connected by an edge, updating the graph structure, …
  - We need efficient graph operations
- $G = (V, E)$ is commonly modelled as
  - set of nodes (vertices) $V$
  - set of edges $E$
  - $n = |V|$, $m = |E|$
- Which data structure should be used?
  - Adjacency matrix, adjacency list, incidence matrix, Laplacian matrix

# Adjacency Matrix

- Bi-dimensional array *A* of *n x n* Boolean values
  - Indexes of the array = node identifiers of the graph
  - The Boolean junction $A_{ij}$ of the two indices indicates whether the two nodes are connected
- Variants
  - Directed graphs, weighted graphs, …

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



# Adjacency List

- A set of lists where each accounts for the neighbours of one node
  - A vector of *n* pointers to adjacency lists
- Often compressed
  - Exploitation of regularities in graphs, difference from other nodes, …

N1 → {N2, N3}

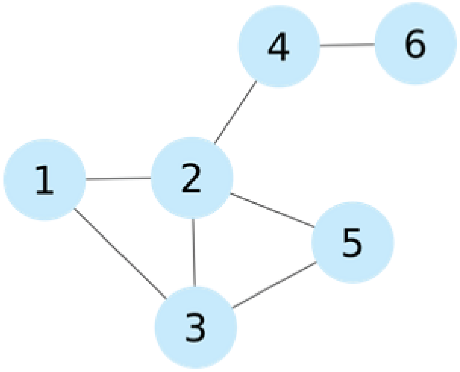
N2 → {N1, N3, N5}

N3 → {N1, N2, N5}

N4 → {N2, N6}

N5 → {N2, N3}

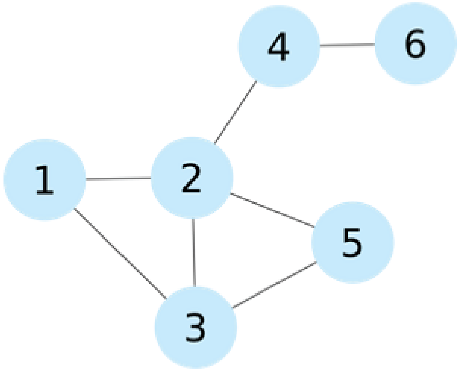
N6 → {N4}



# Incidence Matrix

- Bi-dimensional Boolean matrix of *n* rows and *m* columns
  - A column represents an edge
  - A row represents a node

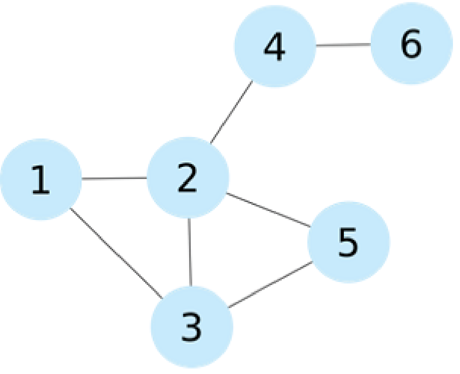
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$



# Laplacian Matrix

- Bi-dimensional array of *n x n* integers
  - Diagonal of the Laplacian matrix indicates the degree of the node
  - The rest of positions are set to -1 if the two vertices are connected, 0 otherwise

$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$



# Graph Databases
## Graph and database types

- A graph database = a set of graphs
- Types of graphs:
  - Directed-labeled graphs
    - e.g., XML, RDF, traffic networks
  - Undirected-labeled graphs
    - e.g., social networks, chemical compounds
- Types of graph databases:
  - Non-transactional = few numbers of very large graphs
    - e.g., Web graph, social networks, …
  - Transactional = large set of small graphs
    - e.g., chemical compounds, biological pathways, linguistic trees each representing the structure of a sentence…

# Graph Databases

## Representatives

Neo4j
the graph database

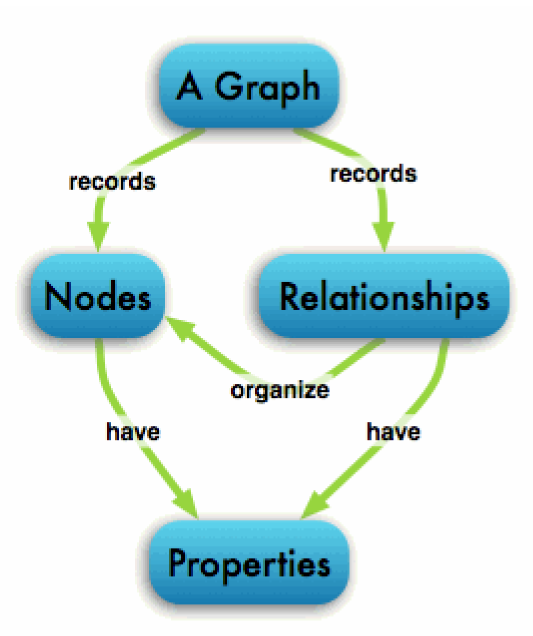
InfiniteGraph
The Distributed Graph Database
for the Cloud and Beyond

OrientDB®

**FlockDB**

# Neo4j

- Open source graph database
  - The most popular
- Initial release: 2007
- Written in: Java
- OS: cross-platform
- Stores data in nodes connected by directed, typed relationships
  - With properties on both
  - Called property graph




http://www.neo4j.org/

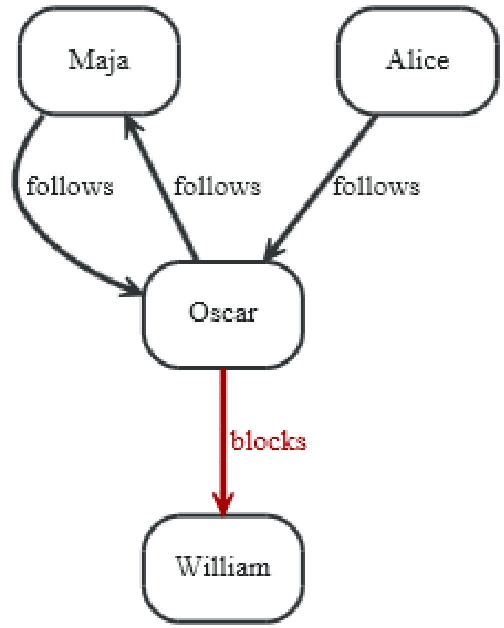
# Neo4j

## Main Features (according to Authors)

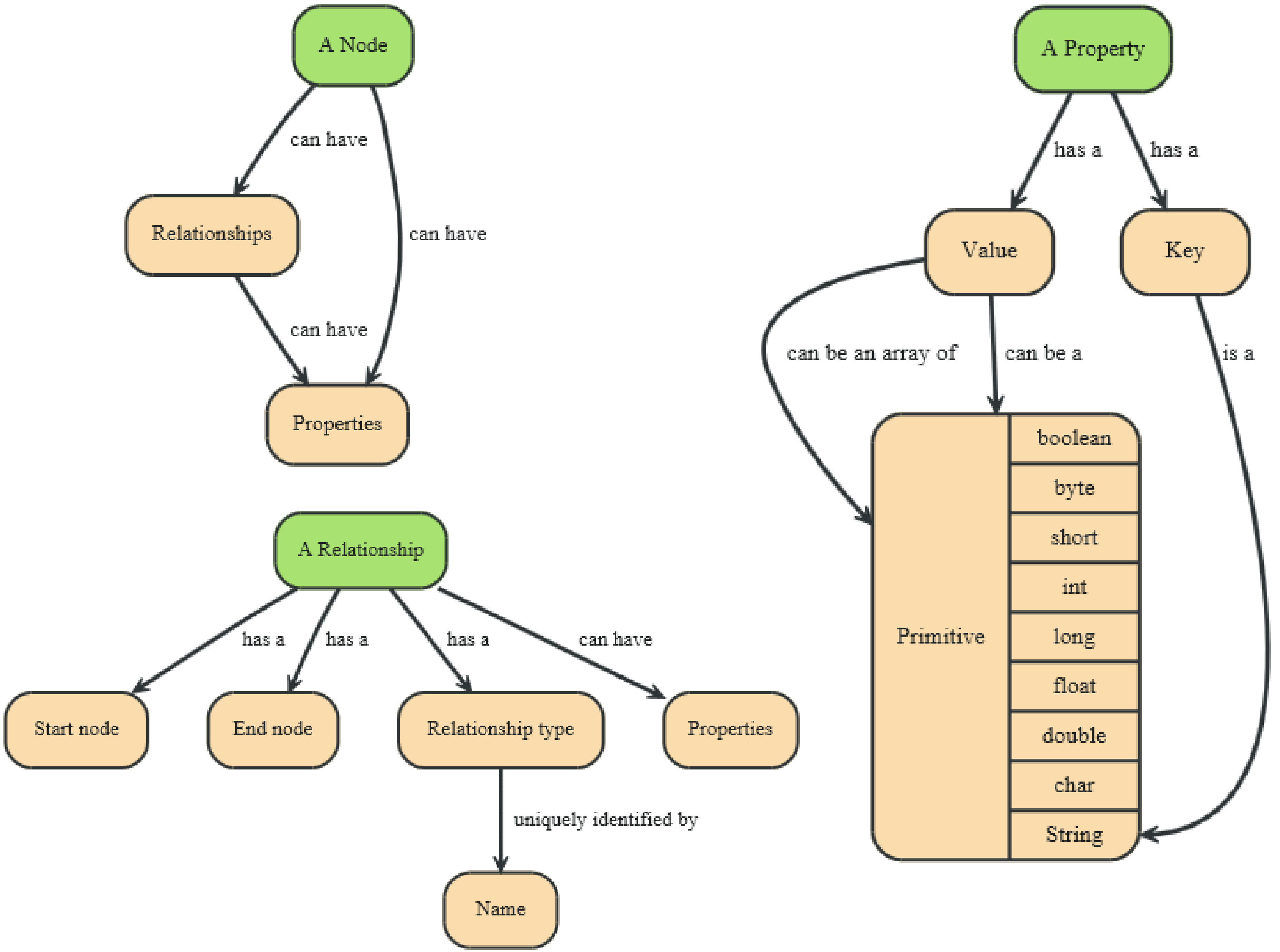
- intuitive – a graph model for data representation
- reliable – with full ACID transactions
- durable and fast – disk-based, native storage engine
- massively scalable – up to several billions of nodes / relationships / properties
- highly-available – when distributed across multiple machines
- expressive – powerful, human readable graph query language
- fast – powerful traversal framework
- embeddable
- simple – accessible by REST interface / object-oriented Java API

# Neo4j

## Data Model – Node, Relationship, Property




- Fundamental units: nodes + relationships
- Both can contain properties
  - Key-value pairs where the key is a string
  - Value can be primitive or an array of one primitive type
    - e.g., `String`, `int`, `int[]`, ...
  - `null` is not a valid property value
    - nulls can be modelled by the absence of a key
- Relationships
  - Directed (incoming and outgoing edge)
    - Equally well traversed in either direction = no need to add both directions to increase performance
    - Direction can be ignored when not needed by applications
  - Always have start and end node
  - Can be recursive


A Node
can have
Relationships
can have
can have
Properties
A Relationship
has a
has a
has a
can have
Start node
End node
Relationship type
Properties
uniquely identified by
Name
A Property
has a
has a
Value
Key
can be an array of
can be a
is a
Primitive
boolean
byte
short
int
long
float
double
char
String

| 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 | |

# Neo4j

## “Hello World” Graph – Java API

```
// enum of types of relationships:
private static enum RelTypes implements RelationshipType
{
    KNOWS
};

GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;

// starting a database (directory is created if not exists):
graphDb = new
     GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);

// …
```

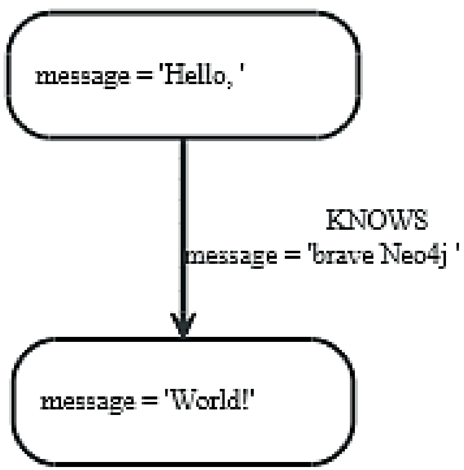
# Neo4j

## "Hello World" Graph

```
// create a small graph:
firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = graphDb.createNode();
secondNode.setProperty( "message", "World!" );

relationship = firstNode.createRelationshipTo
   (secondNode, RelTypes.KNOWS);
relationship.setProperty
   ("message", "brave Neo4j ");

// …
```



# Neo4j
## “Hello World” Graph

```
// print the result:
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );

// let's remove the data:
firstNode.getSingleRelationship
   (RelTypes.KNOWS, Direction.OUTGOING).delete();
firstNode.delete();
secondNode.delete();

// shut down the database:
graphDb.shutdown();
```

# Neo4j

## “Hello World” Graph – Transactions

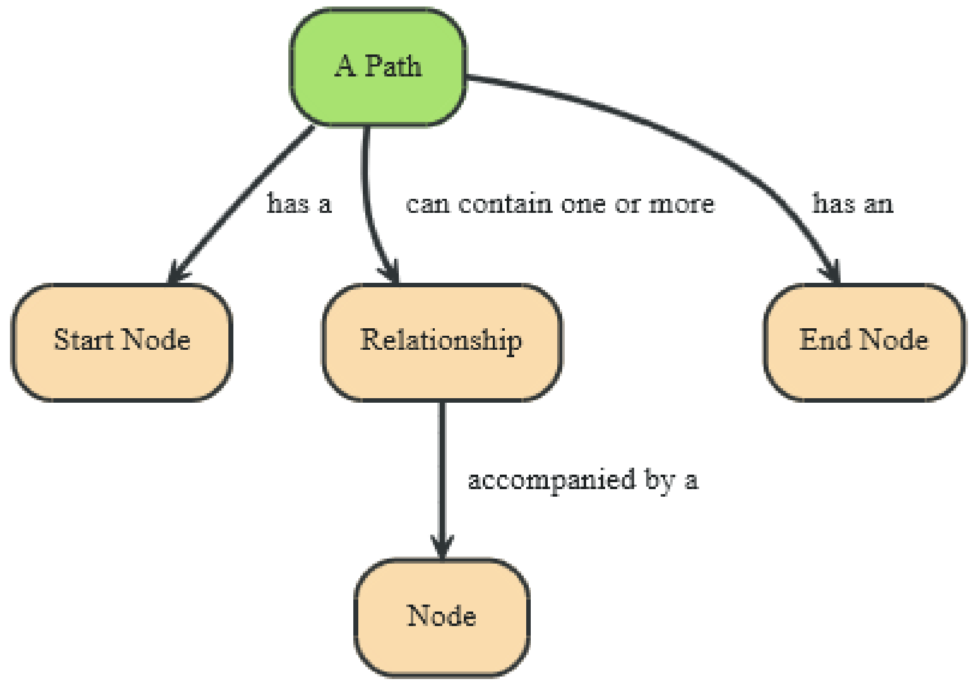
```
// all writes (creating, deleting and updating any data)
// have to be performed in a transaction,
// otherwise NotInTransactionException

Transaction tx = graphDb.beginTx();
try
{
    // updating operations go here
    tx.success();           // transaction is committed on close
}
catch (Exception e)
{
    tx.failure();           // transaction is rolled back on close
}
finally
{
    tx.close();             // or deprecated tx.finish()
}
```

# Neo4j

## Data Model – Path, Traversal

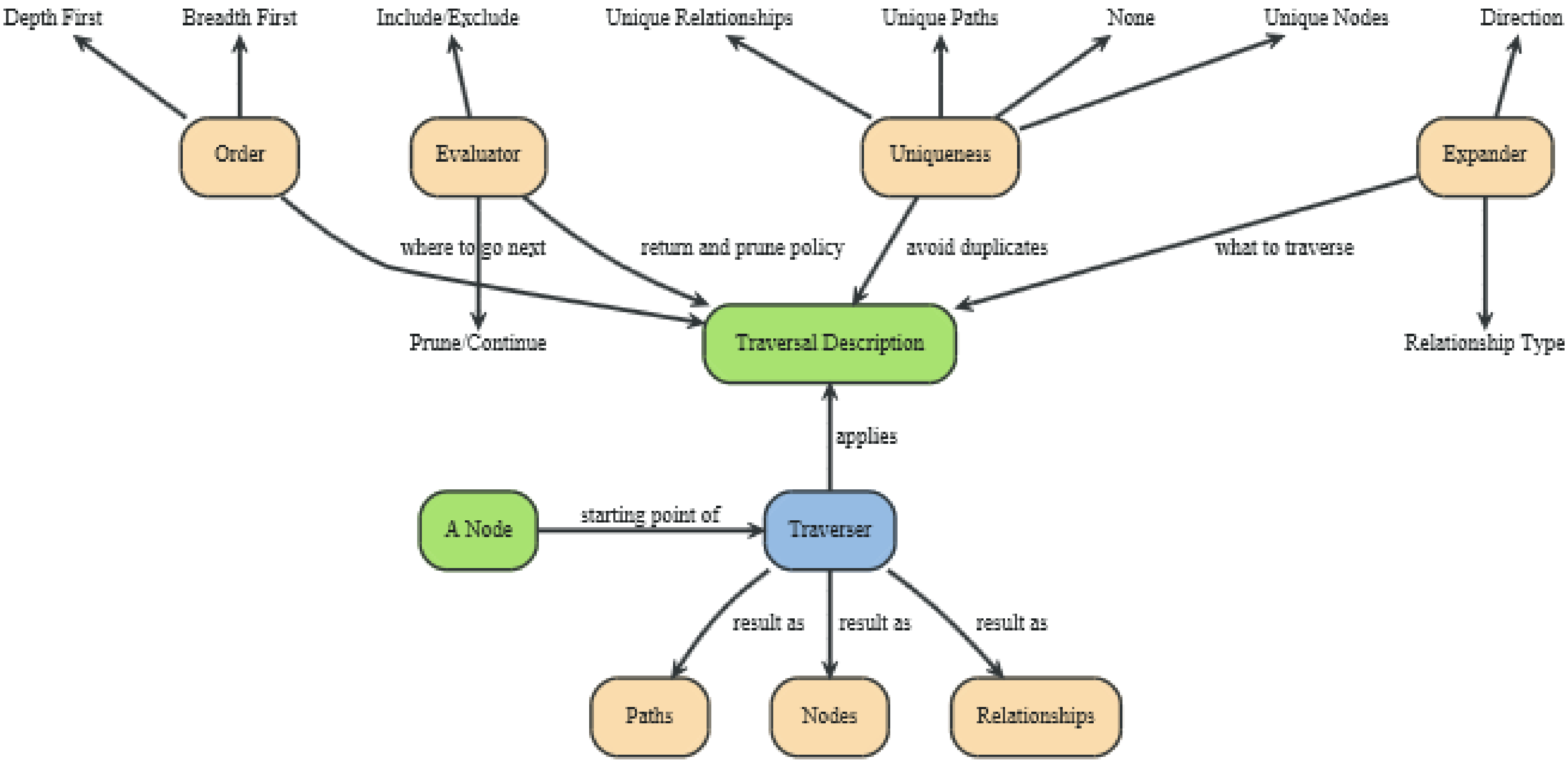
- Path = one or more nodes with connecting relationships
  - Typically retrieved as a query or traversal result
- Traversing a graph = visiting its nodes, following relationships according to some rules
  - Mostly a subgraph is visited
  - Neo4j: Traversal framework + Java API, Cypher, Gremlin



# Neo4j

## Traversal Framework

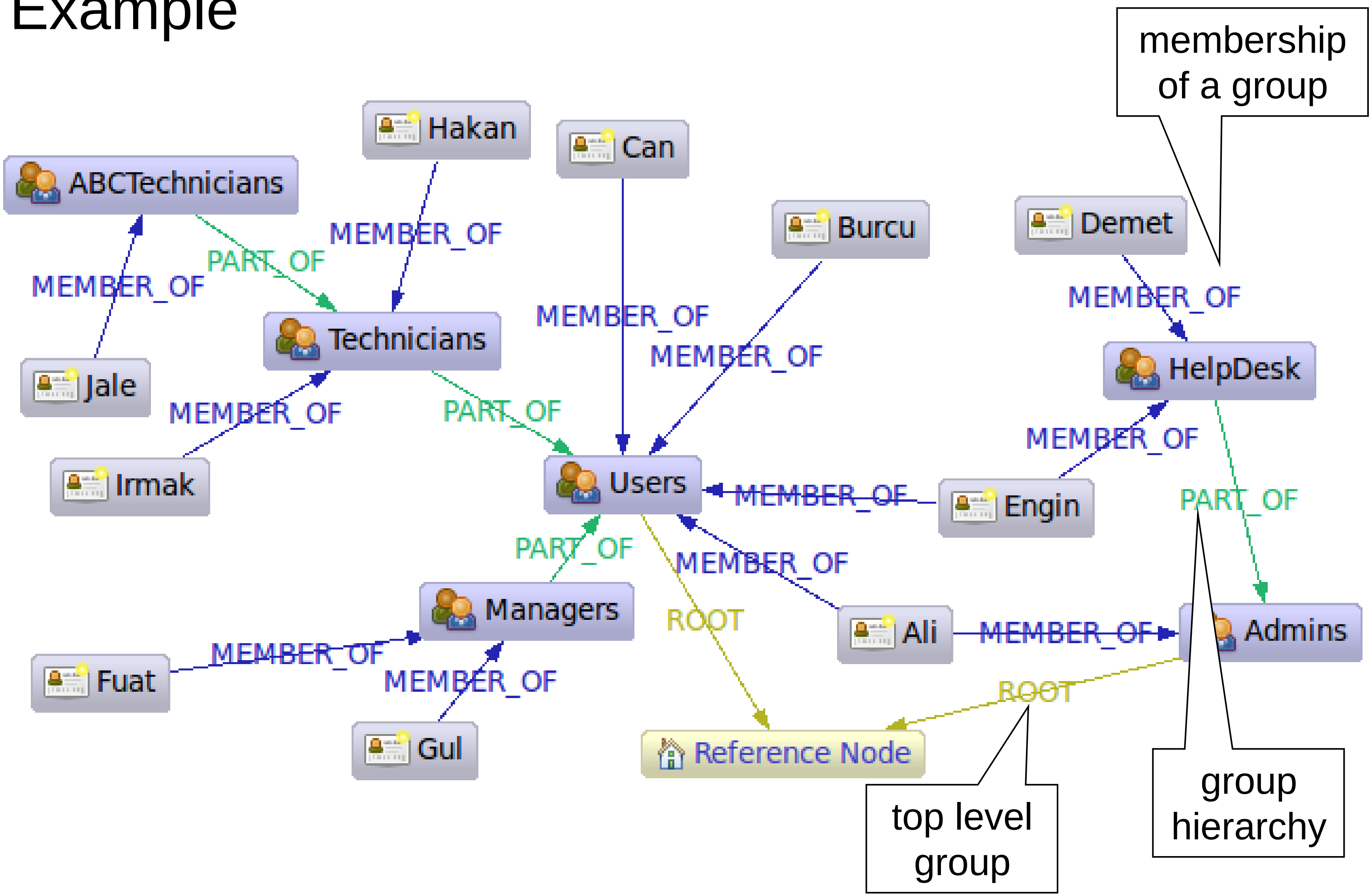
- A traversal is influenced by
  - Expanders – define what to traverse
    - i.e., relationship direction and type
  - Order – depth-first / breadth-first
  - Uniqueness – visit nodes (relationships, paths) only once
  - Evaluator – what to return and whether to stop or continue traversal beyond a current position
  - Starting nodes where the traversal will begin



Depth First
Breadth First
Include/Exclude
Unique Relationships
Unique Paths
None
Unique Nodes
Direction
Order
Evaluator
Uniqueness
Expander
where to go next
return and prune policy
avoid duplicates
what to traverse
Prune/Continue
Traversal Description
Relationship Type
applies
A Node
starting point of
Traverser
result as
result as
result as
Paths
Nodes
Relationships

# Neo4j

## Example



# Neo4j

## Task 1. Get the Admins

```
Node admins = getNodeByName( "Admins" );
TraversalDescription traversalDescription = Traversal.description()
        .breadthFirst()
        .evaluator( Evaluators.excludeStartPosition() )
        .relationships( RoleRels.PART_OF, Direction.INCOMING )
        .relationships( RoleRels.MEMBER_OF, Direction.INCOMING );
Traverser traverser = traversalDescription.traverse( admins );

String output = "";
for ( Path path : traverser )
{
    Node node = path.endNode();
    output += "Found: "
              + node.getProperty( NAME ) + " at depth: "
              + (path.length()) + "\n";
}
```

```
Found: HelpDesk at depth: 1
Found: Ali at depth: 1
Found: Engin at depth: 2
Found: Demet at depth: 2
```

# Neo4j
## Traversal Framework – Java API

- **TraversalDescription**
  - The main interface used for defining and initializing traversals
  - Not meant to be implemented by users
    - Just used
  - Can specify branch ordering
    - **`breadthFirst()`** / **`depthFirst()`**
- **Relationships**
  - Adds a relationship type to traverse
    - Empty (default) = traverse all relationships
    - At least one in the list = traverse the specified ones
  - Two methods: including / excluding direction
    - **`Direction.BOTH`**
    - **`Direction.INCOMING`**
    - **`Direction.OUTGOING`**

# Neo4j

## Traversal Framework – Java API

- **Evaluator**
  - Used for deciding at each position: should the traversal continue, and/or should the node be included in the result
  - Actions:
    - **`Evaluation.INCLUDE_AND_CONTINUE`**: Include this node in the result and continue the traversal
    - **`Evaluation.INCLUDE_AND_PRUNE`**: Include this node in the result, but do not continue the traversal
    - **`Evaluation.EXCLUDE_AND_CONTINUE`**: Exclude this node from the result, but continue the traversal
    - **`Evaluation.EXCLUDE_AND_PRUNE`**: Exclude this node from the result and do not continue the traversal
  - Pre-defined evaluators:
    - **`Evaluators.excludeStartPosition()`**
    - **`Evaluators.toDepth(int depth)`** / **`Evaluators.fromDepth(int depth)`**
    - …

# Neo4j

## Traversal Framework – Java API

- **Uniqueness**
  - Can be supplied to the `TraversalDescription`
  - Indicates under what circumstances a traversal may revisit the same position in the graph
    - **`NONE`**: Any position in the graph may be revisited.
    - **`NODE_GLOBAL`**: No node in the graph may be re-visited (default)
    - …
- **Traverser**
  - Traverser which is used to step through the results of a traversal
  - Steps can correspond to
    - **`Path`** (default)
    - **`Node`**
    - **`Relationship`**

# Neo4j

## Task 2. Get Group Membership of a User

```
Node jale = getNodeByName( "Jale" );
traversalDescription = Traversal.description()
        .depthFirst()
        .evaluator( Evaluators.excludeStartPosition() )
        .relationships( RoleRels.MEMBER_OF, Direction.OUTGOING )
        .relationships( RoleRels.PART_OF, Direction.OUTGOING );
traverser = traversalDescription.traverse( jale );
```

```
Found: ABCTechnicians at depth: 1
Found: Technicians at depth: 2
Found: Users at depth: 3
```

# Neo4j

## Task 3. Get All Groups

```
Node referenceNode = getNodeByName( "Reference_Node" ) ;
traversalDescription = Traversal.description()
        .breadthFirst()
        .evaluator( Evaluators.excludeStartPosition() )
        .relationships( RoleRels.ROOT, Direction.INCOMING )
        .relationships( RoleRels.PART_OF, Direction.INCOMING );
traverser = traversalDescription.traverse( referenceNode );
```

```
Found: Admins at depth: 1
Found: Users at depth: 1
Found: HelpDesk at depth: 2
Found: Managers at depth: 2
Found: Technicians at depth: 2
Found: ABCTechnicians at depth: 3
```

# Neo4j

## Task 4. Get All Members of a Group

```
Node referenceNode = getNodeByName( "Reference_Node" ) ;
traversalDescription = Traversal.description()
        .breadthFirst()
        .evaluator(
                Evaluators.includeWhereLastRelationshipTypeIs
                        ( RoleRels.MEMBER_OF ) );
traverser = traversalDescription.traverse( referenceNode );
```

```
Found: Ali at depth: 2
Found: Engin at depth: 2
Found: Burcu at depth: 2
Found: Can at depth: 2
Found: Demet at depth: 3
Found: Gul at depth: 3
Found: Fuat at depth: 3
Found: Hakan at depth: 3
Irmak at depth: 3
Found: Jale at depth: 4
```

# Cypher

- Neo4j graph query language
  - For querying and updating
- Still growing = syntax changes are probable
- Declarative – we describe what we want, not how to get it
  - Not necessary to express traversals
- Human-readable
  - Inspired by SQL and SPARQL

http://docs.neo4j.org/chunked/stable/cypher-query-lang.html

# Cypher Clauses

- START: Starting points in the graph, obtained via index lookups or by element IDs.
- MATCH: The graph pattern to match, bound to the starting points in START.
- WHERE: Filtering criteria.
- RETURN: What to return.
- CREATE: Creates nodes and relationships.
- DELETE: Removes nodes, relationships and properties.
- SET: Set values to properties.
- FOREACH: Performs updating actions once per element in a list.
- WITH: Divides a query into multiple, distinct parts.

# Cypher Examples

## Creating Nodes

```
CREATE n

(empty result)
Nodes created: 1
```

```
CREATE (a {name : 'Andres'})
RETURN a

a
Node[2]{name:"Andres"}
1 row
Nodes created: 1
Properties set: 1
```

```
CREATE (n  {name : 'Andres', title : 'Developer'})

(empty result)
Nodes created: 1
Properties set: 2
```

# Cypher Examples

## Creating Relationships

```
START a=node(1), b=node(2)
CREATE a-[r:RELTYPE]->b
RETURN r

r
:RELTYPE[1] {}
1 row
Relationships created: 1
```

```
START a=node(1), b=node(2)
CREATE a-[r:RELTYPE {name : a.name + '<->' + b.name }]->b
RETURN r

r
:RELTYPE[1] {name:"Andres<->Michael"}
1 row
Relationships created: 1
Properties set: 1
```

# Cypher Examples

## Creating Paths

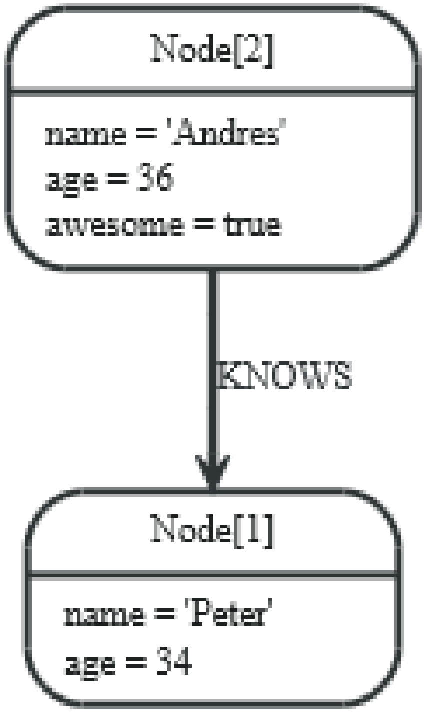
```
CREATE p = (andres {name:'Andres'})-[:WORKS_AT]->neo<-
   [:WORKS_AT]-(michael {name:'Michael'})
RETURN p
```

```
p
[Node[4]{name:"Andres"},:WORKS_AT[2]
   {},Node[5]{},:WORKS_AT[3] {},Node[6]{name:"Michael"}]
1 row
Nodes created: 3
Relationships created: 2
Properties set: 2
```

all parts of the pattern not already in scope are created

# Cypher Examples

## Changing Properties




```
START n = node(2)
SET n.surname = 'Taylor'
RETURN n

n
Node[2]{name:"Andres",age:36,awesome:true,surname:"Taylor"}
1 row
Properties set: 1
```

```
START n = node(2)
SET n.name = null
RETURN n

n
Node[2]{age:36,awesome:true}
1 row
Properties set: 1
```

# Cypher Examples

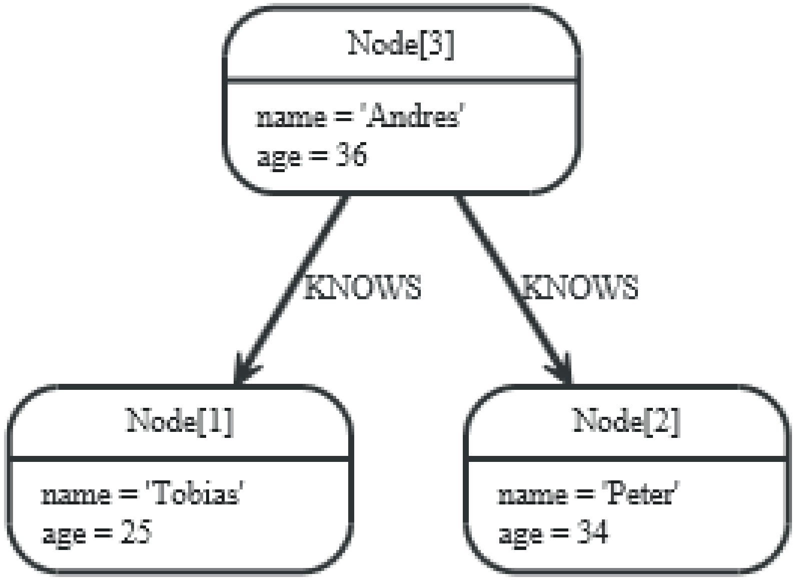
## Delete

```
START n = node(4)
DELETE n
```

```
(empty result)
Nodes deleted: 1
```

```
START n = node(3)
MATCH n-[r]-()
DELETE n, r
```

```
(empty result)
Nodes deleted: 1
Relationships deleted: 2
```



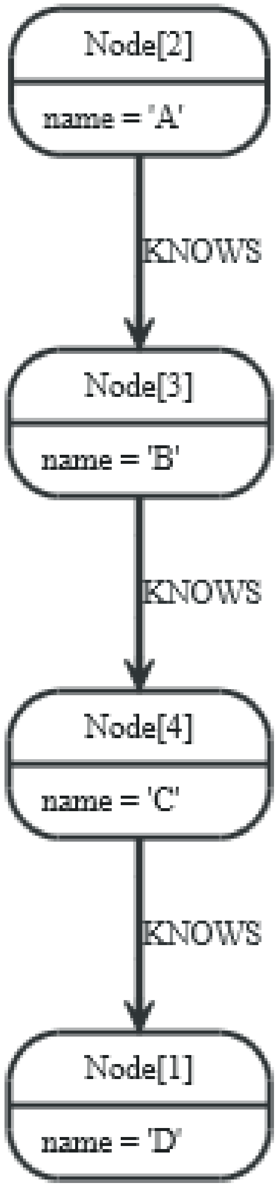
# Cypher Examples

## Foreach

```
START begin = node(2), end = node(1)
MATCH p = begin -[*]-> end
FOREACH(n in nodes(p) | SET n.marked = true)
```

```
(empty result)
Properties set: 4
```

can be combined with any update command



# Cypher Examples

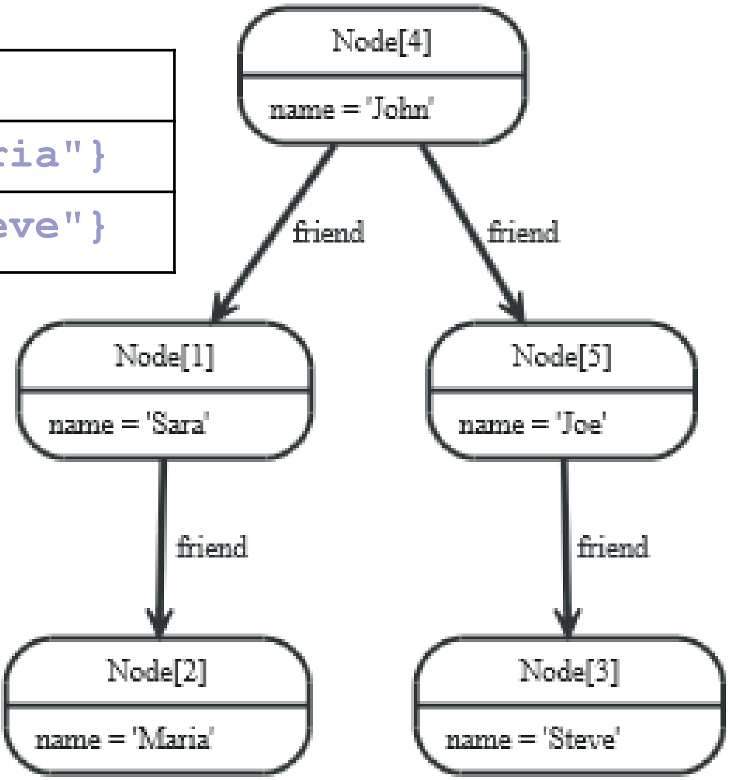
## Querying

in general: **node:index-name(key = "value")**

```
START john=node:node_auto_index(name = 'John')
MATCH john-[:friend]->()-[:friend]->fof
RETURN john, fof
```

| john | fof |
|---|---|
| Node[4]{name:"John"} | Node[2]{name:"Maria"} |
| Node[4]{name:"John"} | Node[3]{name:"Steve"} |

```
neo4j.properties file:
...
node_auto_indexing=true
relationship_auto_indexing=true
node_keys_indexable=name,phone
relationship_keys_indexable=since
...
```



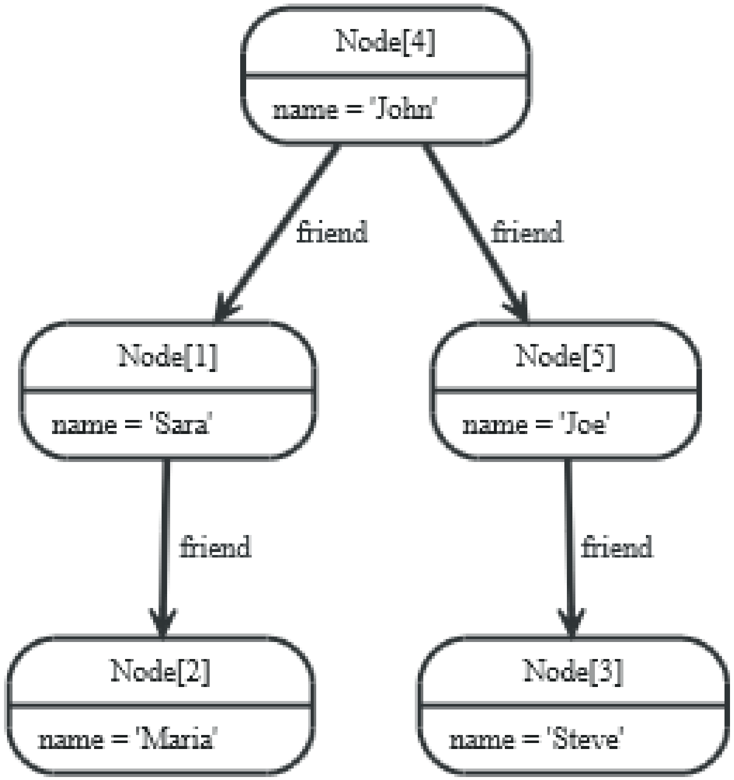
# Cypher Examples

## Querying

List of users

```
START user=node(5,4,1,2,3)
MATCH user-[:friend]->follower
WHERE follower.name =~ 'S.*'
RETURN user, follower.name
```

| user | follower.name |
|---|---|
| Node[5]{name:"Joe"} | "Steve" |
| Node[4]{name:"John"} | "Sara" |



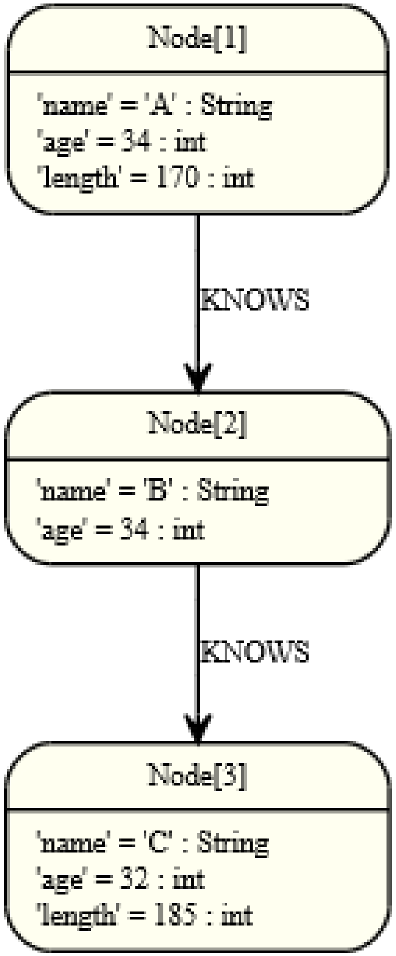
# Cypher Examples

## Order by

```
START n=node(3,1,2)
RETURN n
ORDER BY n.name
```

We can use:
- multiple properties
- asc/desc

| n |
| --- |
| Node[1]{name->"A",age->34,length->170} |
| Node[2]{name->"B",age->34} |
| Node[3]{name->"C",age->32,length->185} |



# Cypher Examples

## Count

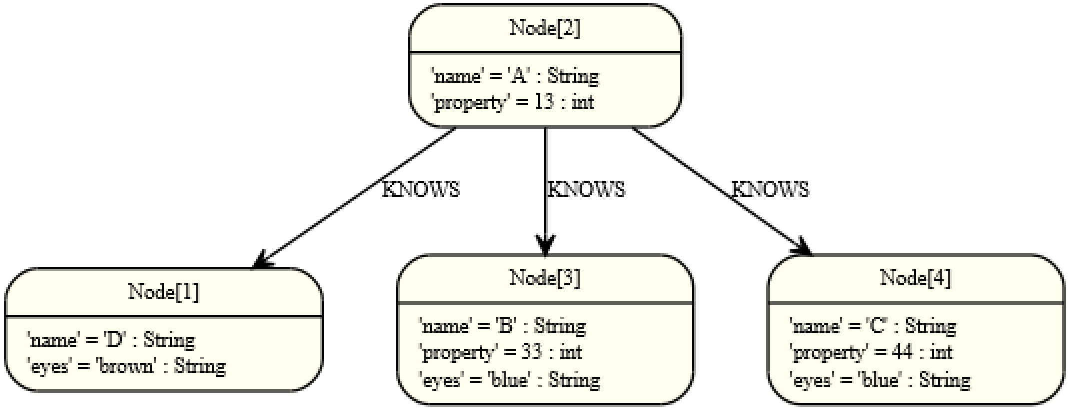
```
START n=node(2)
MATCH (n)-->(x)
RETURN n, count(*)
```

| n | count(*) |
| --- | --- |
| Node[2]{name->"A",property->13} | 3 |

```
START n=node(2)
MATCH (n)-[r]->()
RETURN type(r), count(*)
```

| TYPE(r) | count(*) |
| --- | --- |
| "KNOWS" | 3 |

count the groups of relationship types



# Cypher

- And there are many other features
  - Other aggregation functions
    - Count, sum, avg, max, min
  - LIMIT n - returns only subsets of the total result
    - SKIP n = trimmed from the top
    - Often combined with order by
  - Predicates ALL and ANY
  - Functions
    - LENGTH of a path, TYPE of a relationship, ID of node/relationship, NODES of a path, RELATIONSHIPS of a path, …
  - Operators
  - …