TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

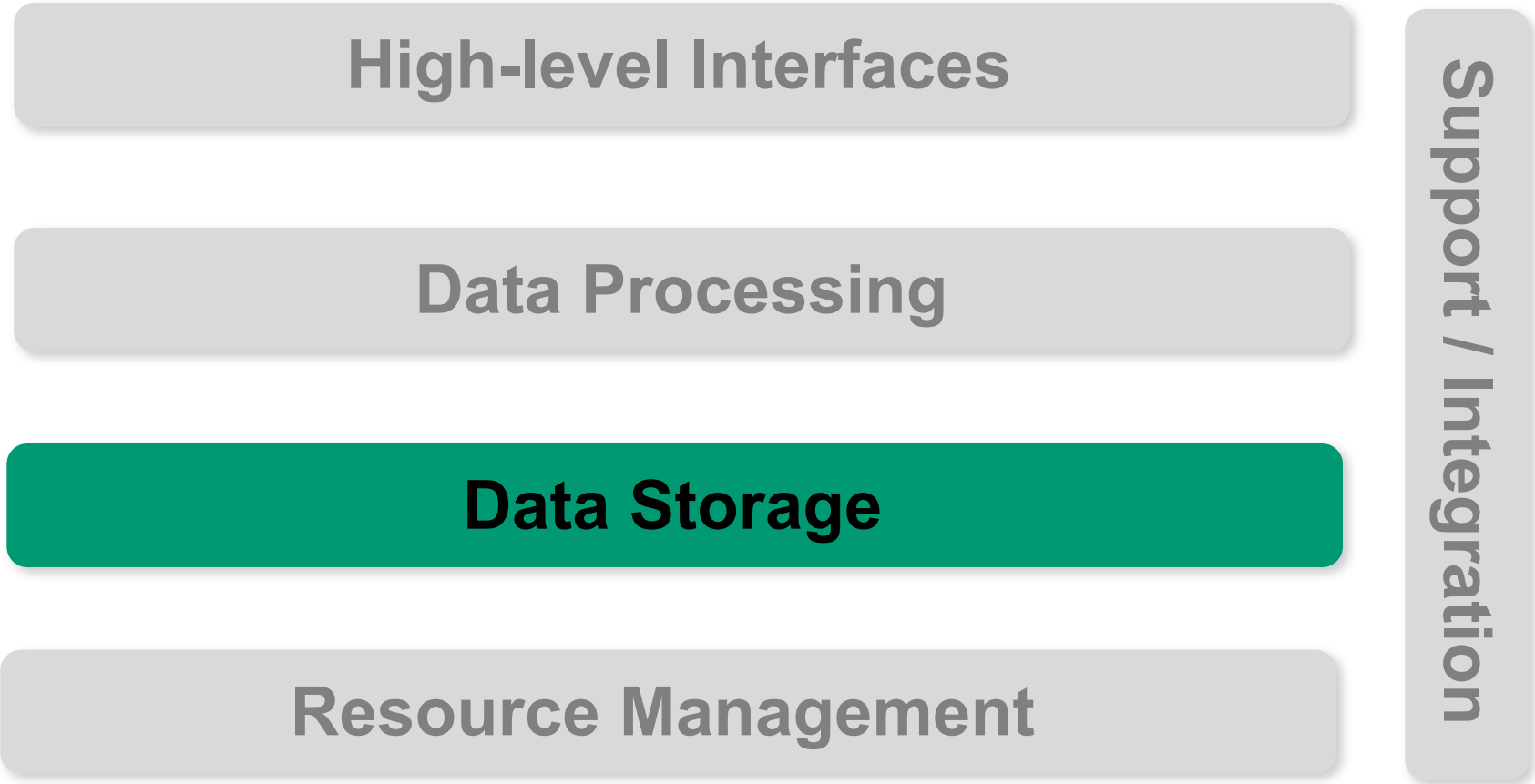
NoSQL: Neo4j

A.A. 2025/26

Matteo Nardelli

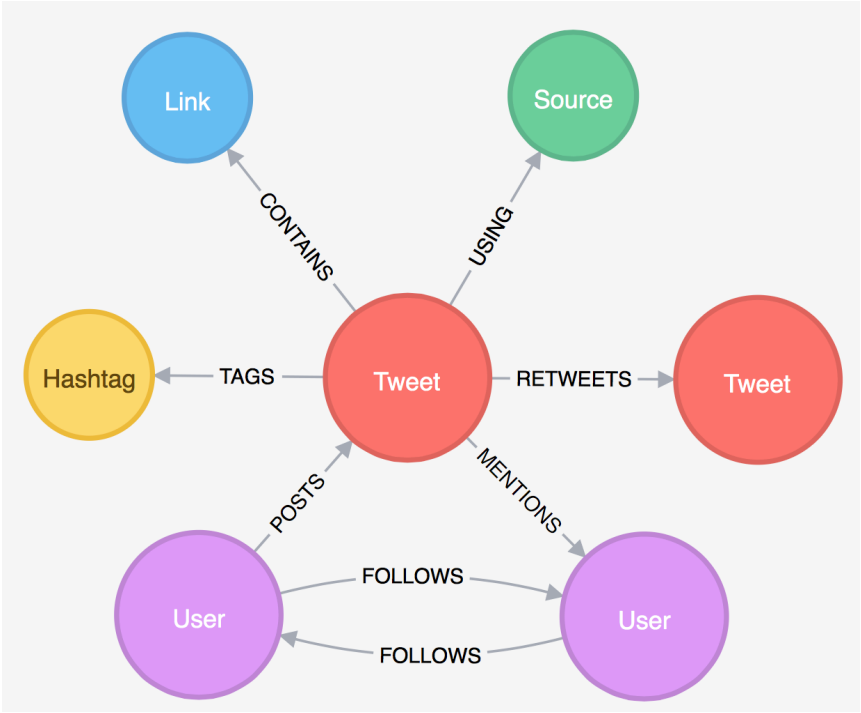
Laurea Magistrale in Ingegneria Informatica - II anno

The reference Big Data stack



Graph data model

- Uses graph structures
  - Nodes are the entities and have a set of attributes
  - Edges are the relationships between the entities
    - E.g.: an author writes a book
  - Edges can be directed or undirected
  - Nodes and edges also have individual properties consisting of key-value pairs



Graph data model

- Powerful data model
  - Differently from other types of NoSQL stores, it concerns itself with relationships
  - Focus on visual representation of information (more human-friendly than other NoSQL stores)
  - Other types of NoSQL stores are poor for interconnected data

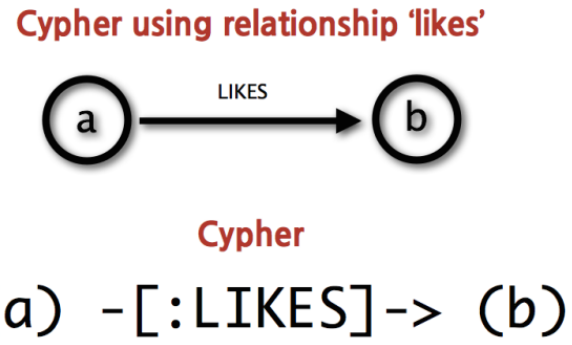
- Cons:
  - Sharding: data partitioning is difficult
  - Horizontal scalability
    - When related nodes are stored on different servers, traversing multiple servers is not performance-efficient
  - Requires rewiring your brain

Suitable use cases for graph databases

- Good for applications where you need to model entities and relationships between them
  - Social networking applications
  - Pattern recognition
  - Dependency analysis
  - Recommendation systems
  - Solving path finding problems raised in navigation systems
  - …
- Good for applications in which the focus is on querying for relationships between entities and analyzing relationships
  - Computing relationships and querying related entities is simpler and faster than in RDBMS

Neo4j: data model

- A graph records data in nodes and relationships
- Nodes are often used to represent entities
  - A node can have properties, relationships, and can also be labeled with one or more labels
  - Note that a node can have relationships to itself
- Relationships organize nodes by connecting them
  - A relationship connects two nodes; a start node and an end node
  - A relationship can have properties



Neo4j: data model

- Properties (both nodes and relationships) can be of different type:
  - Numeric values
  - String values
  - Boolean values
  - Lists of any other type of value
- Labels assign roles or types to nodes
  - A label is a named graph construct that is used to group nodes into sets
  - All nodes labeled with the same label belong to the same set
  - Labels can be added and removed at runtime
  - A node can have multiple labels

Neo4j: Cypher

- A traversal navigates through a graph to find paths;
  - starts from starting nodes to related nodes, finding answers to questions
- Cypher provides a declarative way to query the graph powered by traversals and other techniques
- A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result

- Cypher: is a textual declarative query language
  - It uses a form of ASCII art to represent graph-related patterns

Cypher

```
(a) -[:LIKES]-> (b)
```

Hands-on Neo4j
(Docker image)

Neo4j with Dockers

- We use the official neo4j container

```
$ docker pull neo4j:5.6.0
```

- Create a container with Neo4j and forward its ports

```
$ docker run
        --publish=7474:7474
        --publish=7687:7687
        --volume=$HOME/neo4j/data:/data
        neo4j:5.6.0
```

- We will interact with Neo4j using its webUI

```
http://localhost:7474
```

Cypher syntax

- Cypher uses a pair of parentheses (usually containing a text string) to represent a node

```
(varname:Label { p_name: p_value, ... } )
```

  - () represents a node
  - varname (optional) assigns a name to the node that can be used elsewhere within a single statement.
  - the Label (prefixed with a colon ":") declares the node's type (or label).
  - the node's properties are represented as a list of key/value pairs, enclosed within a pair of braces

Cypher syntax

- Cypher uses a pair of dashes (--) to represent an undirected relationship. Directed relationships have an arrowhead at one end (<--, -->).
  - It is possible to create only directed relationship, although they can be queried as undirected

```
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Bracketed expressions ([...]) are used to add details:

- a variable (e.g., role) can be defined, to be used elsewhere in the statement.
- the relationship's type (e.g., :ACTED_IN) is analogous to the node's label. A relationship can have at most one type.
- the properties (e.g., roles) are entirely equivalent to node properties.

Cypher syntax

Variables:

To increase modularity and reduce repetition, Cypher allows patterns to be assigned to variables

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

https://neo4j.com/developer/cypher-query-language/

Cypher syntax: Create

Create a node with label Person and property name with value "you":

```
CREATE (you:Person {name:"You"})
RETURN you
```

Create a more complex structure: add a new node and a new relationship with the existing one

```
MATCH  (you:Person {name:"You"})
CREATE (you)-[like:LIKE]->(neo:Database {name:"Neo4j"})
RETURN you, like, neo
```

Cypher syntax: Find, Update and Remove

Find a node (basic syntax)

```
MATCH (you {name:"You"})-[:FRIEND]->(yourFriends)
RETURN you, yourFriends
```

Update an existing node (similarly, to update a relationship)

```
MATCH (n {property:value})
SET n :NewLabel
RETURN n
```

Remove a property (or a Label) from a node (or relationship)

```
MATCH (b {name: "Bruce Springsteen"})
REMOVE b.nickname RETURN b
```

Cypher syntax: Delete

Delete a node:

```
MATCH (a:ToDel)
DELETE a
```

Note that a node cannot be deleted if it participates in a relationship. To remove also relationships, we need to detach the node, delete it and its relationships:

```
MATCH (b {name: "Bruce Springsteen"})
DETACH DELETE b;
```

Cypher syntax: Read Clauses

These clauses read data from the data store:

- MATCH Specify the patterns to search for in the database
- OPTIONAL MATCH Specify the patterns to search for in the database while using nulls for missing parts of the pattern
- WHERE Adds constraints to the patterns in a MATCH or OPTIONAL MATCH clause or filter the results of a WITH clause
- START Find starting points through legacy indexes

Read more: http://neo4j.com/docs/developer-manual/current/cypher/clauses/

Cypher syntax: Write Clauses

These clauses write data to the data store:

- CREATE Create nodes and relationships
- MERGE Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
- ON CREATE (used with MERGE) it specifies the actions to take if the pattern needs to be created.
- SET Update labels on nodes and properties on nodes and relationships.
- DELETE Delete graph elements (nodes, relationships or paths).
- REMOVE Remove properties and labels from nodes and relationships.

Cypher syntax: General Clauses

These comprise general clauses that work in conjunction with other clauses:

- RETURN Defines what to include in the query result set.
- ORDER BY A sub-clause following RETURN or WITH, specifying that the output should be sorted in particular way.
- LIMIT Constrains the number of rows in the output.
- SKIP Defines from which row to start including the rows in the output
- WITH Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
- UNION Combines the result of multiple queries.

Cypher syntax: Operators

Within clauses, we often rely on operators to combine and compare nodes/relationships or access to their properties

General operators:

DISTINCT, . for property access,

[] for dynamic property access

Mathematical operators:

+, -, *, /, %, ^

Comparison operators:

=, <>, <, >, <=, >=, IS NULL, IS NOT NULL

Cypher syntax: Operators

String-specific comparison operators:

STARTS WITH, ENDS WITH, CONTAINS

Boolean operators

AND, OR, XOR, NOT

String operators

+ for concatenation, =~ for regex matching

List operators

+ for concatenation,

IN to check existence of an element in a list,

[] for accessing element(s)

Cypher syntax: Relationship pattern length

Relationship pattern length:

```
(a)-[*2]->(b)
```

It is possible to specify a length (2 in the example) in the relationship description of a pattern.

It can be a variable length:

*3..5 (between 3 and 5),
*3.. (greater than 3),
*..5 (less than 5),
* (any length)

Read more: http://neo4j.com/docs/developer-manual/current/cypher/functions/

Cypher syntax: Relationship pattern

Relationship pattern:

- nodes and relationship expressions are the building blocks for more complex patterns;
- patterns can be written continuously or separated with commas

Examples:

- friend-of-a-friend:

```
(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
```

- shortest path:

```
path = shortestPath( (user)-[:KNOWS*..5]-(other) )
```

http://neo4j.com/docs/developer-manual/current/cypher/clauses/match/

Neo4j's Native Graph Processing

- Neo4j utilizes index-free adjacency:
  - Each node maintains direct references to its adjacent nodes.
  - Each node acts as a micro-index of other nearby nodes
    - This is much cheaper than using global indexes
    - Also, query times are independent of the total size of the graph, but are proportional to the amount of the graph searched!
  - In contrast, non-native graph databases use (global) indexes to link nodes together.

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

Neo4j's Data on Disk

- Database files are persisted for long term durability;
  - data/databases/neo4j/neostore*
  - data stored as linked lists of fixed size records;
  - follow offsets to know how to fetch data;

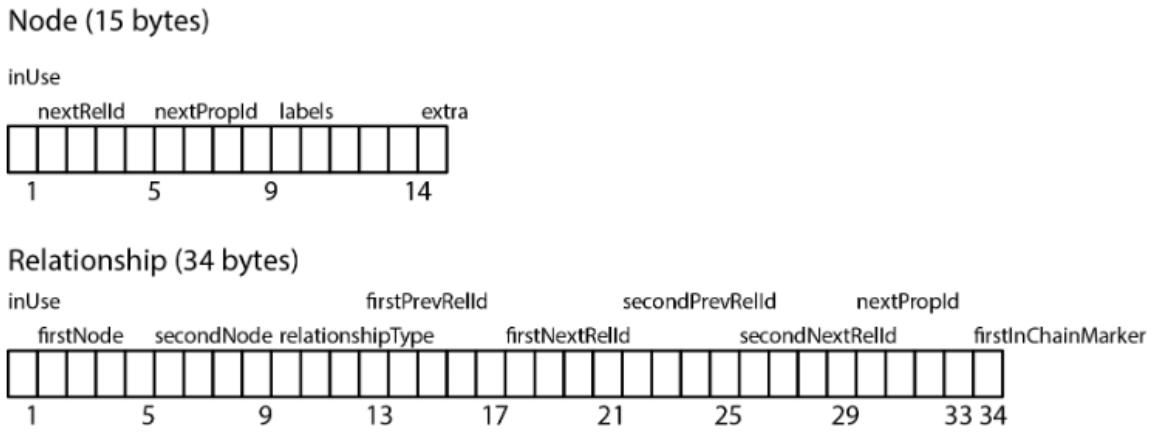
Store File	Record size	Contents
neostore.nodestore.db	15 B	Nodes
neostore.relationshipstore.db	34 B	Relationships
neostore.propertystore.db	41 B	Properties for nodes and relationships
neostore.propertystore.db.strings	128 B	Values of string properties
neostore.propertystore.db.arrays	128 B	Values of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

- **Properties**: linked list holding a key and value and pointing to the next property.
- Each **node** and **relationship** references its first property record;
- **Nodes** also reference the first relationship in its relationship chain;
- Each **relationship** references its start and end node and the previous and next relationship record for the start and end node.

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

Neo4j's Data on Disk

- Details of node and relationship records:

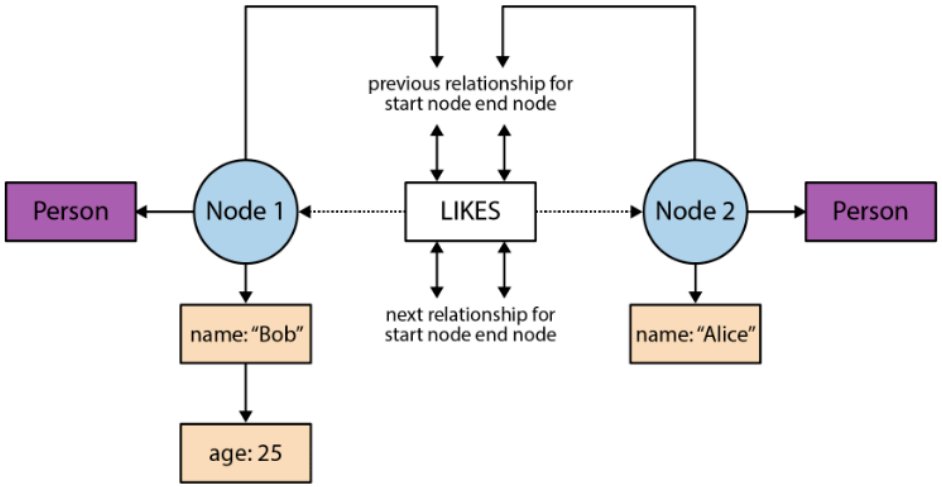


- Fixed-size records enable fast lookups for nodes in the store file.
  - Based on this format, knowing the record id, the database can directly compute a record's location, at cost O(1)
  - More efficient than, e.g., performing a search, which would cost O(log n).

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

Neo4j's Data on Disk

- How data is retrieved?
  - Each node record contains a pointer to its first property and relationship;
  - To read a node's properties, we follow the singly linked list structure beginning with the pointer to the first property.
  - To find a relationship for a node, we follow the node's relationship pointer to its first relationship.
    - From here, we follow the doubly linked list of relationships for that particular node until we find the relationship we're interested in.
  - We can read the relationship properties using the singly linked list structure (also used for node properties).



Store files:

- nodestore.db
- relationshipstore.db
- propertystore.db