ADBMS- Graph Database

Shabana A T

NITC

May 7, 2020

Outline

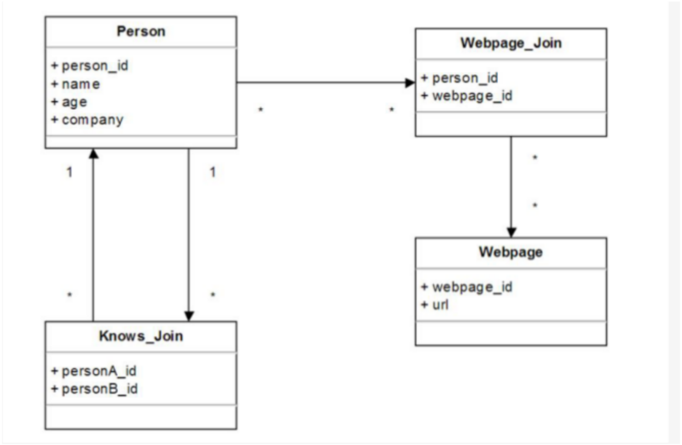
Why Graphs are important?

- Graphs are useful for representing real world data
- Therefore it is relavent that large graphs can be represented in a database. The current leading model is the relational model. While graph data can be stored efficiently in relational databases, but many of the more powerful graph operations are either hard to implement, or execute inefficiently.
- Therefore, the concept of graph database was introduced to solve this problem.

What is Graph Database

- A graph database is simply a database that is built on top of a graph data structure.
- Like in a graph, graph databases can store nodes and edges between nodes.
- Each node and edge is uniquely identified and may contain properties.
- For example, a node may contain the properties such as name, occupation, age, etc.
- An edge also has a label that defines the relationship between two nodes.
- A graph database will support graph operations such as traversals and shortest path calculations. Because a lot of the data we are interested in can be represented by graphs.
- Graph databases are very good at representing data that has a lot of many-to-many relationships.

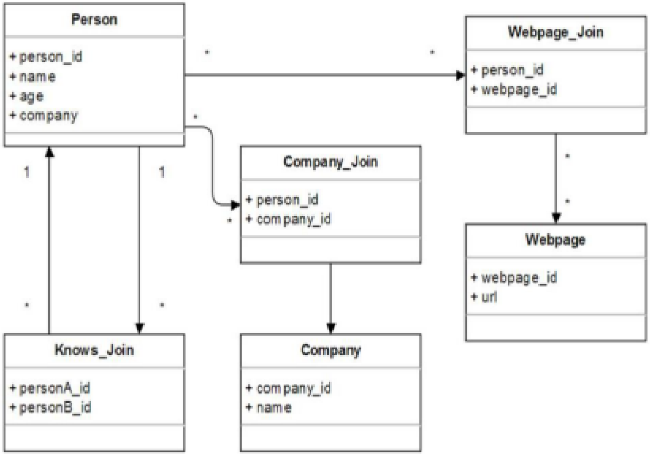
Relational Database Example



Relational Database Example(Continue...)

- We have a schema of a relational database that stores people, their name, age, the company they work at, who they know, and webpages that they like.
- The knows relation and likes relation is represented by join tables
- What if we wanted to allow multiple companies? We could either allo duplicate rows or create another join table.

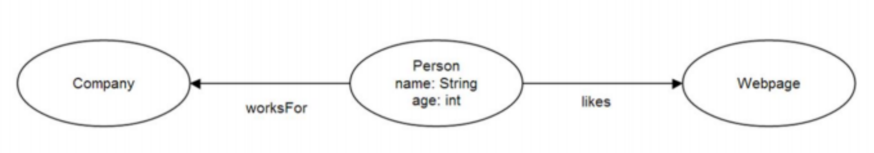
Relational Database Example(Continue...)



Relational Database Example(Continue...)

- Company_Join.
- It is important to note that for each relationship we wish to add, we must create another join table.
- However, a problem arises when the number of join tables increase to a large number.
- Because join tables represent edges, to do any meaningful work that requires traversals would require the relational database to execute a lot of joins.
- This is known as a **join bomb** and leads to reduced query performance.

Graph Database Example



- This diagram is the graph database schema that represents the same people data that was presented before.
- The circles represent nodes, and the solid lines represent relationships.
- To solve the same problem in a graph database, we need only create a new edge from the Person node to the Company node. This is a much simpler solution. An important thing to note is that graph databases do not have to execute joins for each edge traversal, and avoid join bombs.

- We live in a connected world! There are no isolated pieces of information, but rich, connected domains all around us. Only a database that natively embraces relationships is able to store, process, and query connections efficiently. While other databases compute relationships at query time through expensive JOIN operations, a graph database stores connections alongside the data in the model.
- Accessing nodes and relationships in a native graph database is an efficient, constant-time operation and allows you to quickly traverse millions of connections per second per core.

- **Nodes** are the entities in the graph. They can hold any number of attributes (key-value pairs) called properties. Nodes can be tagged with labels, representing their different roles in your domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes.
- **Relationships** provide directed, named, semantically-relevant connections between two node entities (e.g. Employee WORKS_FOR Company). A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. Due to the efficient way relationships are stored, two nodes can share any number or type of relationships without sacrificing performance. Although they are stored in a specific direction, relationships can always be navigated efficiently in either direction.

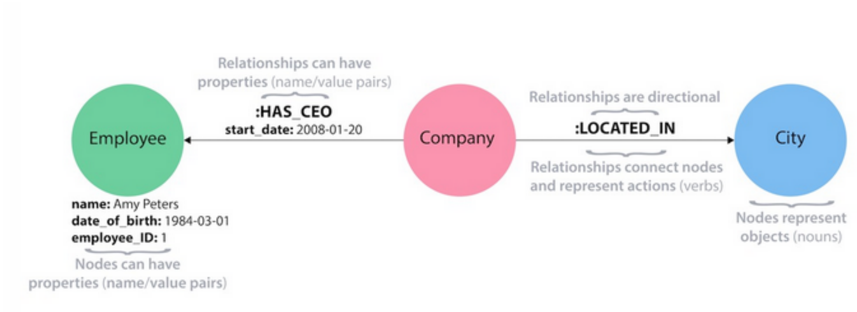

Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
Relationships connect nodes and represent actions (verbs)
City
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)
Nodes represent objects (nouns)

Figure: Graph Database

Query Processing in Graph Database

- There are several questions you might want to ask of data stored in a graph.
  - 1. List for me all the nodes or edges that have a certain property.
  - 2. Find subgraphs that match a given set of relationships.
  - 3. Can these two nodes reach each other?
  - 4. How many hops does it take for two nodes to connect?
- Is easy to think of in a graph model.
- How do we translate these questions into actionable queries?

Neo4j

- Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for your applications.
- Neo4j is referred to as a native graph database because it efficiently implements the property graph model down to the storage level. This means that the data is stored exactly as you whiteboard it, and the database uses pointers to navigate and traverse the graph.
- **Cypher**, a declarative query language similar to SQL, but optimized for graphs.
- **Constant time traversals** in big graphs for both depth and breadth due to efficient representation of nodes and relationships. Enables scale-up to billions of nodes on moderate hardware.
- **Flexible** property graph schema that can adapt over time, making it possible to materialize and add new relationships later to shortcut and speed up the domain data when the business needs change.

Neo4j Graph model

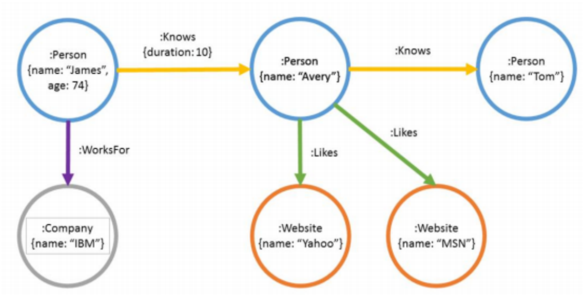
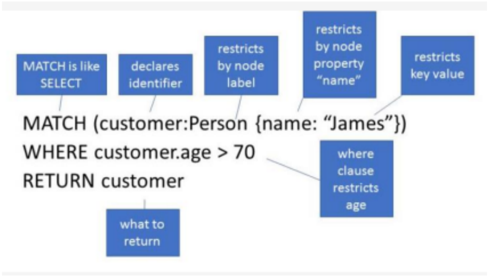


Figure: Neo4j Graph Database

- The Neo4j graph is composed of nodes and edges, with an unlimited number of edges between nodes.
- Nodes and edges can have properties, which are key-value pairs.
- They can also be given labels, which define the type of each node or edge.

Example queries

- Give me people named James older than 70.

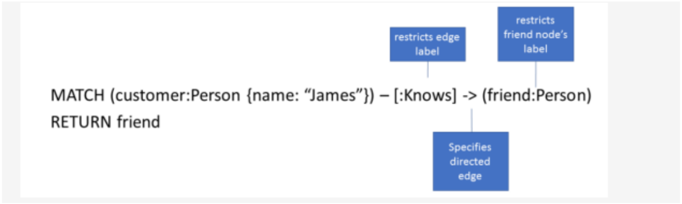


```
MATCH (customer:Person {name: “James”})
WHERE customer.age > 70
RETURN customer
```

- A term after the colon indicates that we want the node to have a Person label in effect, we are looking only for person nodes. (If we didnt specify this, wed be searching through all kinds of nodes.)
- necessary property key-value pairs.

Example queries

- Give me James friends.

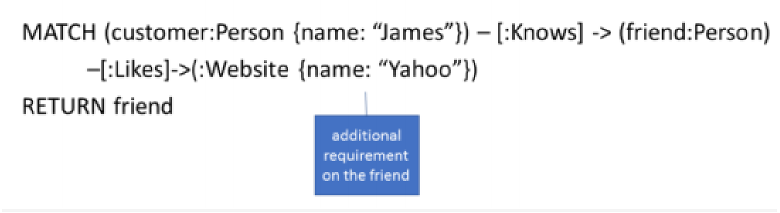


```
MATCH (customer:Person {name: “James”}) – [:Knows] -> (friend:Person)
RETURN friend
```

- Restrict the edge between the two nodes to directed edges that are of the Knows type.
- Search for another node, a node of type Person, which is assigned the identifier friend.
- Finally return friends, with all the nodes properties.

Example queries

- Give me James friends who likes Yahoo.

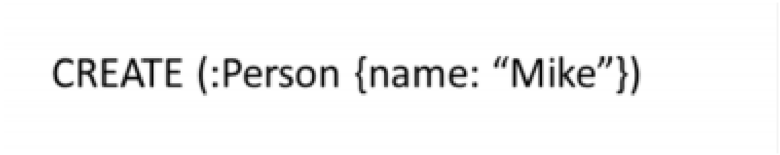


```
MATCH (customer:Person {name: “James”}) – [:Knows] -> (friend:Person)
        –[:Likes]->(:Website {name: “Yahoo”})
RETURN friend
```

additional requirement on the friend

Example queries

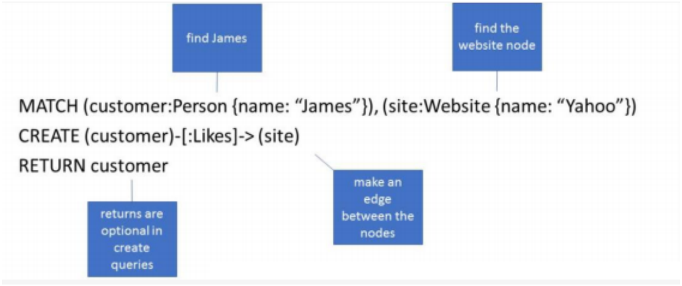
- Create a person named Mike



```
CREATE (:Person {name: “Mike”})
```

Example queries

- Stores that James like Yahoo(Add a new relationship)



```
MATCH (customer:Person {name: "James"}), (site:Website {name: "Yahoo"})
CREATE (customer)-[:Likes]-> (site)
RETURN customer
```

find James
find the website node
make an edge between the nodes
returns are optional in create queries

- To add an edge between two existing nodes, we have to MATCH to find the nodes first, and then use the nodes identifiers to create a new edge between them.
- Returns are optional.