Property Graph Databases

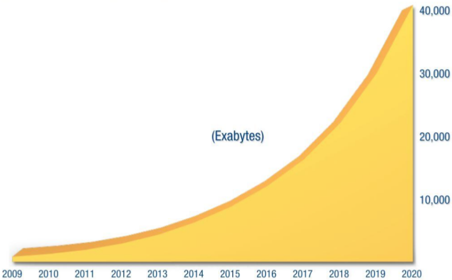
KGA

2018-10-01

Dr. Hamed Shariat Yazdi, Prof. Jens Lehmann

Motivation

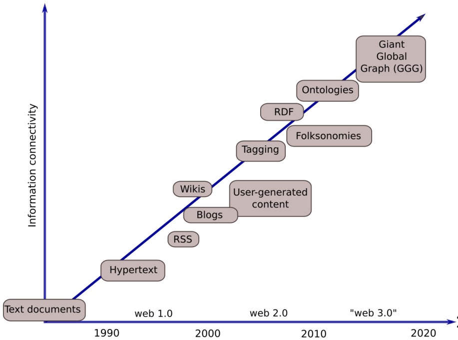
Trend 1: Big Data



The amount of information that we generate and transfer has increased dramatically in the past few years.[1]

[1] Source: IDC's Digital Universe Study, sponsored by EMC, December 2012
http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

Trend 2: Connectedness



[2]

Data is evolving to be more interlinked and connected. Hypertext has links, blogs have pingback, tagging groups all related data.

[2]Figure taken from [3]

Why Relational Databases are not Enough

(Provocative) claim: *Relational databases are not good at handling relationships!*

[3] Table taken from [4]

Why Relational Databases are not Enough

(Provocative) claim: *Relational databases are not good at handling relationships!*

Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

[3]

In this social network database, it is easy to find people Bob considers his friends:

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Result:

[3] Table taken from [4]

Why Relational Databases are not Enough

(Provocative) claim: *Relational databases are not good at handling relationships!*

Person

ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend

PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

[3]

In this social network database, it is easy to find people Bob considers his friends:

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Result: Alice and Zach

[3] Table taken from [4]

Why Relational Databases are not Enough?

- Sad but true: friendship is not always symmetric!
- What if we want to know who considers Bob as friend?

```sql
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.PersonID = p1.ID
JOIN Person p2
  ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```

- User side: still easy to implement
- Database side: Database has to consider all the rows in the **PersonFriend** table $\Rightarrow$ more expensive!

Why Relational Databases are not Enough?

What if we query deeper into the social network?

```sql
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

- ▷ Uses recursive joins ⇝ increase in syntactic, computational, and space complexity of the query

Why Relational Databases are not Enough?

Experimental results: Given any two persons chosen at random, is there a path that connects them that is at most 5 relationships long?

Depth	RDBMS execution time (s)	Neo4j execution time (s)	Records returned
2	0.016	0.01	$\sim$ 2500
3	30.267	0.168	$\sim$ 110,000
4	1543.505	1.359	$\sim$ 600,000
5	Unfinished	2.132	$\sim$ 800,000

- ▷ Comparing relational store and a popular graph database Neo4j
- ▷ Social network consisting of 1,000,000 people
- ▷ Each person with approximately 50 friends

Conclusion: Graph databases outperform relational databases when we are dealing with connected data!

Relational Databases: What exactly is the Problem?

- Relationships exist only between tables $\rightarrow$ problematic for highly connected domains
- Relationship traversal can become very expensive
- Flat, disconnected data structures:
  - Data processing and relationship construction happens in the application layer
- Bound by a previously defined schema

Graph Databases
using the
Labeled Property Graph Model

Graph Databases: Advantages

- ▷ Explicit graph structure:
  - semantic dependencies between entities are made explicit
- ▷ New nodes and new relationships can be easily added into the database without having to migrate data or restructure the existing network
- ▷ Relationships correspond to paths:
  - querying the database = traversing the graph
    $\rightarrow$ this makes certain types of queries simpler
- ▷ Schema-free
- ▷ Index-free adjacency

Index-Free Adjacency

- **Index-Free Adjacency**: each node "knows" (i.e. directly points to) its adjacent nodes
- No explicit index → each node acts as micro-index of nearby nodes → much cheaper than global indices
- Because of this, query times are less/not dependent of the size of the graph → they depend only on the part of the graph that has been searched
- Precomputes and stores bidirectional joins as relationships (i.e. no need to compute them)
- Enables bidirectional traversal of the database:
  we can traverse both incoming and outgoing relationships

Graph Databases: When to Use

When to use:

- ▷ Complex, highly-connected data
- ▷ Dynamic systems with topology that is difficult to predict
- ▷ Dynamic requirements that change over time
- ▷ Cases where relationships in data are themselves meaningful

When not to use:

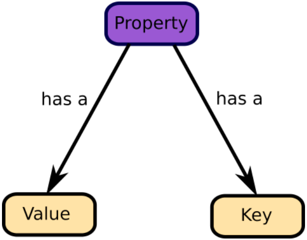
- ▷ Large offline batch analysis tasks of static data
- ▷ Simple, unconnected data structures
- ▷ For certain types of queries:
  - "Graph fishing" (no small set of start nodes)
  - "Bulk Scans" (generic queries not applying to any indexed subset)

Labeled Property Graph Model

Building blocks of a Labeled Property Graph:

- ▷ **Nodes**
  - Can be tagged with one or more labels
  - Can contain properties
- ▷ **Relationships**:
  - Connect nodes and structure the graph
  - Directed
  - Always have a single name, a start node and an end node $\Rightarrow$ no dangling relationships
  - Can have properties
- ▷ **Properties** (= key-value pairs)
- ▷ **Labels**:
  - Group nodes together

Properties

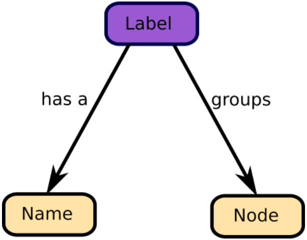


Properties are named values:

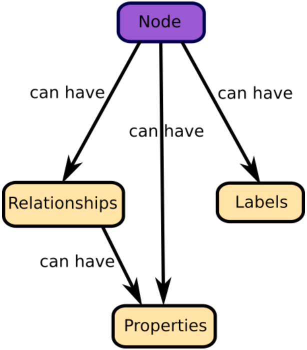
- ▷ Key is the name of the property
  - Always a string
- ▷ Values can be:
  - Numeric values
  - String values
  - Boolean values
  - Lists of any other type of value

Labels

- Used to assign types to nodes
- Groups nodes into sets:
  all nodes with the same label belong to the same set
- Queries can be constrained to these sets instead of the whole graph $\rightsquigarrow$ more efficient queries that are easier to write
- Each node has any number of labels (including none)



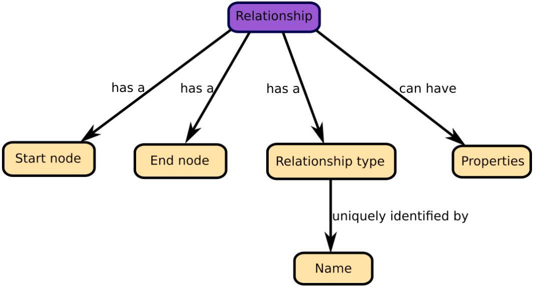
Nodes



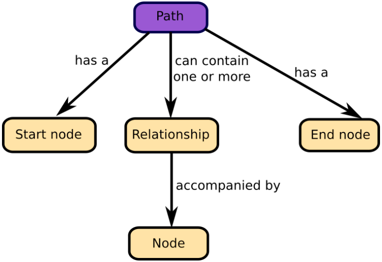
- Together with relationships, fundamental unit of property graph model
  $\rightsquigarrow$ the simplest possible graph is a single node
- Are often used to represent entities
- Can have zero or more properties

Relationships

- Organize the nodes by connecting them
- Always connects a start node to the end node
- A key part of a graph database: allow us to find related data
- Always has a direction

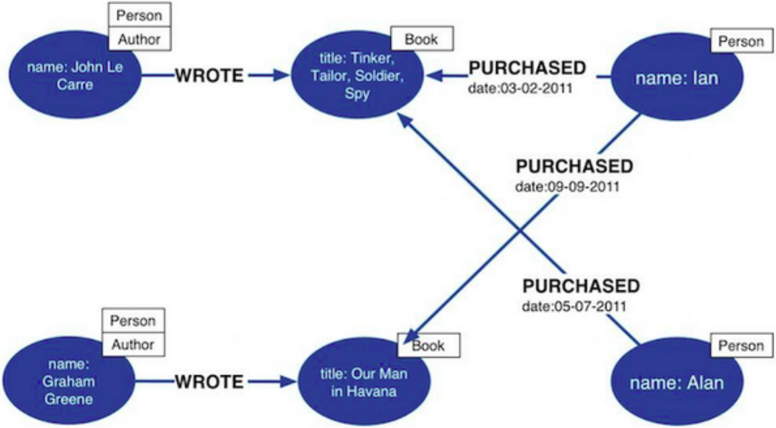


Paths



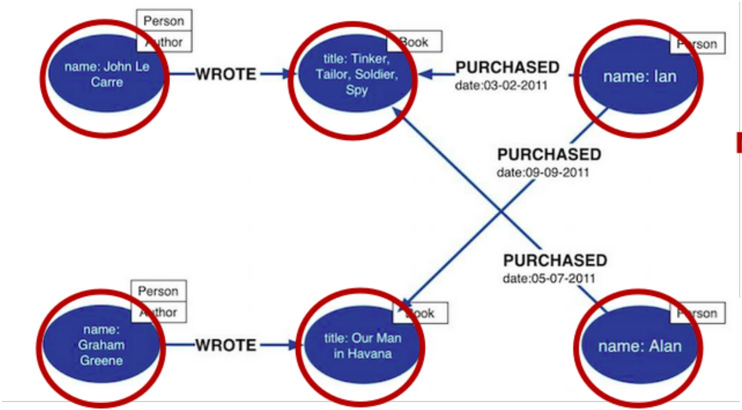
- One or more nodes with connecting relationships
- Typically is a result of a query or a traversal
- Length of a path = number of relationships on that path

Labeled Property Graph Model: Example



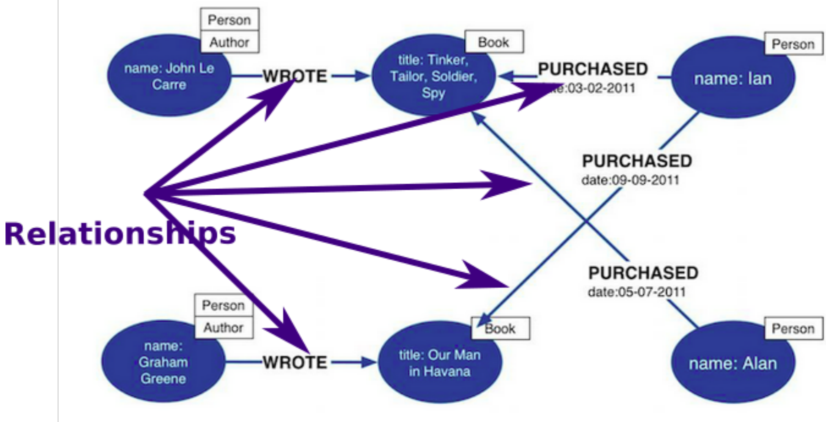
3 Illustration taken from [1]

Labeled Property Graph Model: Example

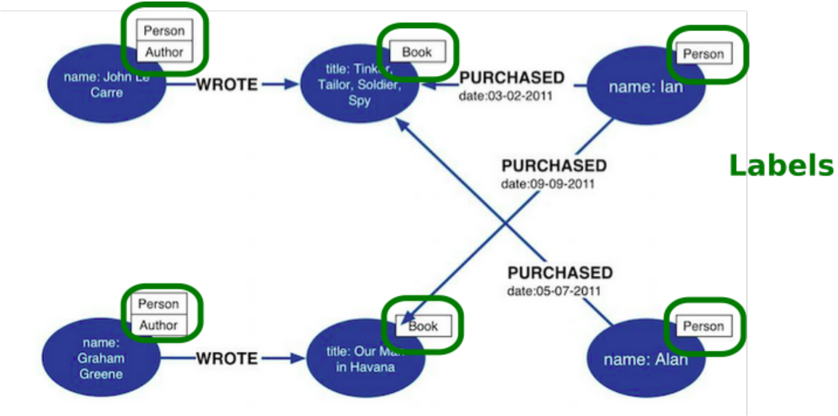


Nodes

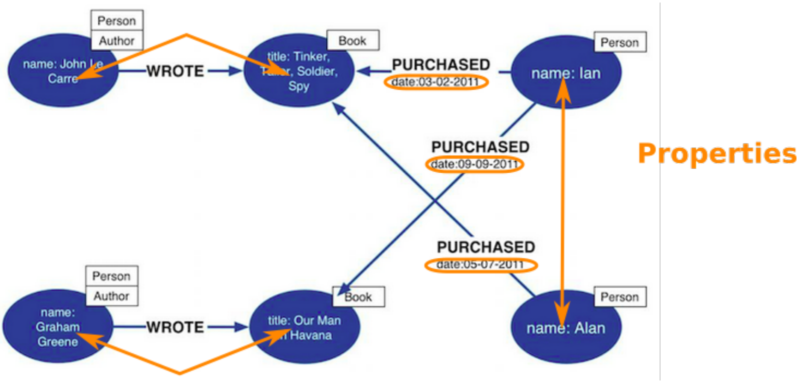
Labeled Property Graph Model: Example



Labeled Property Graph Model: Example



Labeled Property Graph Model: Example



Creating Graph Databases
vs.
Creating Relational Databases

Creating a Relational Database

Step 1: Acquire and improve understanding of data: a **whiteboard sketch** step

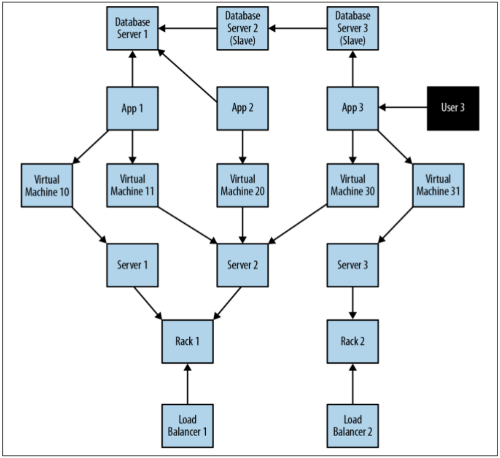


Figure 3-2. Simplified snapshot of application deployment within a data center

[3] Diagram taken from [4]

Creating a Relational Database

Step 2: Construct an entity-relations (E-R) diagram

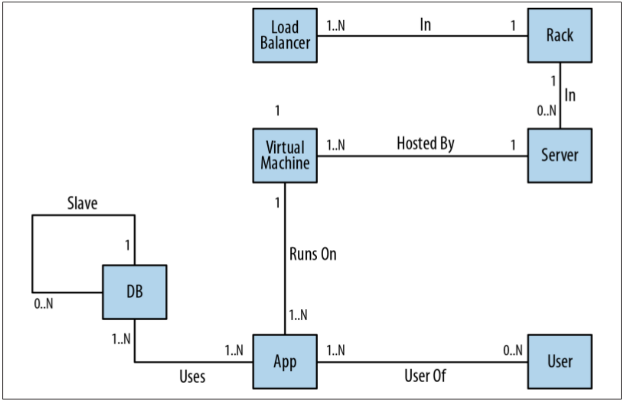


Figure 3-3. An entity-relationship diagram for the data center domain

Note the complexity of the model *before* any data has been added

[3] E-R diagram taken from [4]

Creating a Relational Database

Step 3: Map the E-R diagram into tables and relations and normalize the data

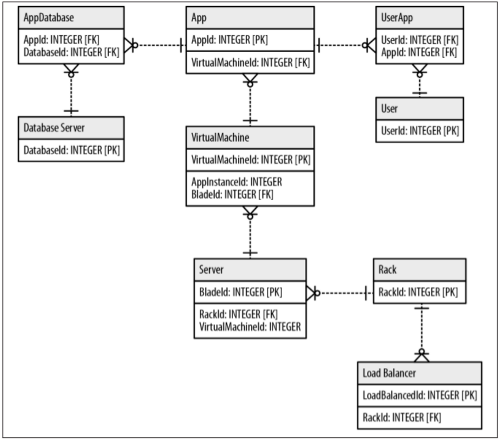


Figure 3-4. Tables and relationships for the data center domain

[3] Diagram taken from [4]

Creating a Graph Database

1. Create a whiteboard sketch
2. ~~Create an E-R diagram~~
3. ~~Map into tables and relations~~

⟵ 2. Domain modeling

- ▷ What you sketch on the whiteboard = what you store in the database
- ▷ No normalization, denormalization or conversion to tables-relations structure necessary
- ▷ No conceptual-relational dissonance: the physical layout of the data is same as it is conceptualized
- ▷ Domain modeling = further graph modeling:
  - Makes sure that every node has the appropriate properties
  - Ensures that every node is in correct semantic context (i.e. add the relations you want to query)

Creating a Graph Database

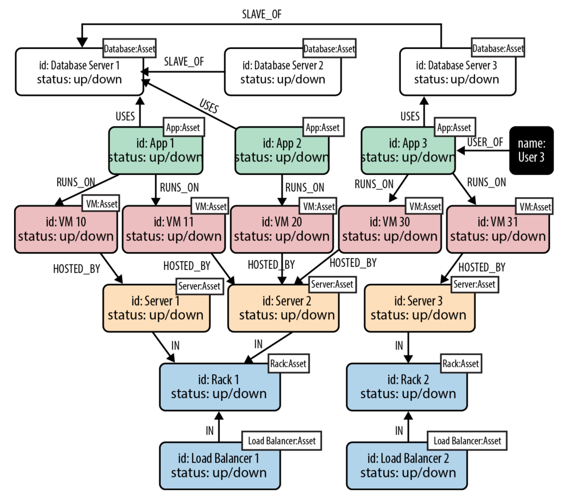


Figure 3-5. Example graph for the data center deployment scenario

3 Image taken from [4]

Graph DBs: Good Design Principles

- ▷ Ensure that later changes are driven by changes in application requirements and not by the need to mitigate bad design decisions
- ▷ There are two techniques to do this:
  - Check that the graph reads well
    - ▷ Pick a node
    - ▷ Follow relationships to other nodes, reading each node's role and each relationship's name as you go
    - ▷ *This should form sensible sentences*
  - *Design for queryability*
    - ▷ Understand end-users' goals: understand the use cases
    - ▷ Try to craft sample queries for the use cases

Design Principles: Reading the Graph

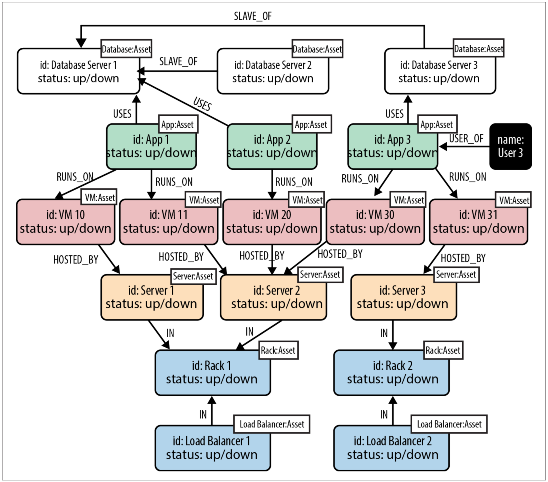


Figure 3-5. Example graph for the data center deployment scenario

"App 1 runs on VM 10, which is hosted by Server 1 in Rack 1."

Neo4J - A Graph Database

Existing Graph Databases

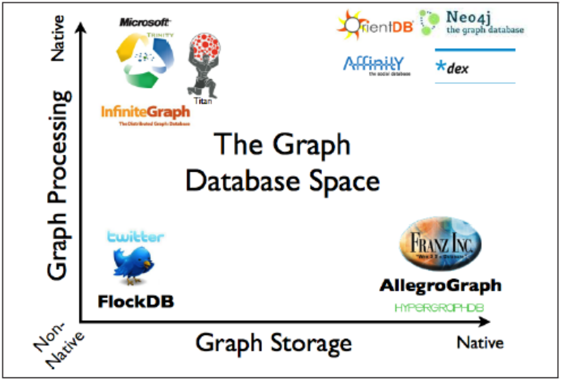


Figure 1-3. An overview of the graph database space

Source: O'Reilly Graph Databases [1]

- ▷ Native graph processing = index-free adjacency = traversal queries work well
- ▷ Native graph storage = storing data in graph shape (e.g. no RDBMS backend)

Neo4j

https://neo4j.com/

- ▷ Probably the most popular graph database today
- ▷ Based on a schema-free labeled property graph model
- ▷ Scales to billions of nodes, relationships and properties

Consists of:

- ▷ Nodes, Relationships, Properties, Labels, Paths, Traversal, and Schema (index and constraints)

Neo4j: Learn more!

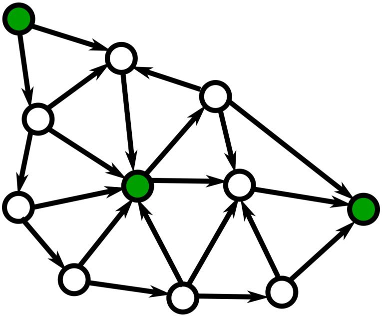
- Open-source
- Extensive support and learning material
- Check out https://neo4j.com/developer/ for further resources (including a sandbox for testing!)
- International events and meetup groups

(Neo4j)-[:LOVES]-(Developers)

Cypher

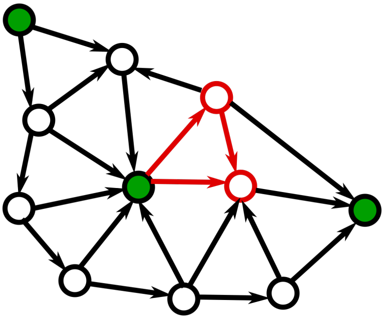
Cypher: Pattern Matching

A pattern matching query language for graphs



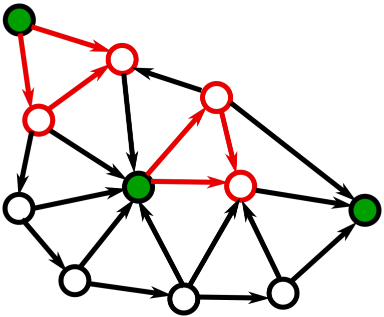
Cypher: Pattern Matching

A pattern matching query language for graphs



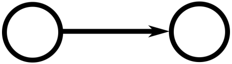
Cypher: Pattern Matching

A pattern matching query language for graphs



Cypher: ASCII Art

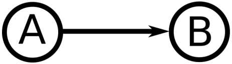
Uses "ASCII art representation"



() --> ()

Cypher: ASCII Art

Directed relationship



(A) --> (B)

Undirected relationship

(A) -- (B)

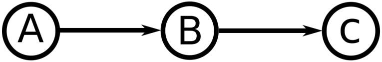
Cypher: ASCII Art

Specific relationships

(A) -[:LOVES]-> (B)

Cypher: ASCII Art

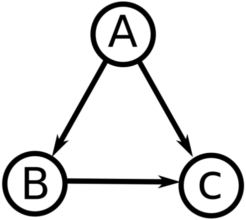
Joined paths



(A) --> (B) --> (C)

Cypher: ASCII Art

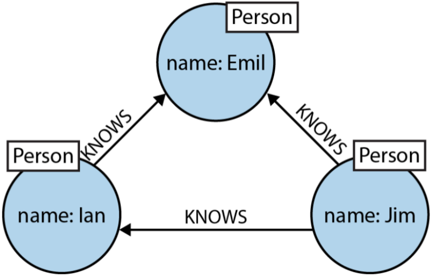
Multiple paths



(A) --> (B) --> (C), (A) --> (C)

(A) --> (B) --> (C) <-- (A)

Cypher - Start



Cypher Patterns

```
(emil:Person {name:'Emil'})
 <-[:KNOWS]- (jim:Person {name:'Jim'})
 -[:KNOWS]-> (ian:Person {name:'Ian'})
 -[:KNOWS]-> (emil)
```

Cypher: `MATCH` Clause

- Specification by example: *draw* data we are looking for
- Used to define a pattern of nodes and relationships that we want to find

MATCH - example

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),
      (a)-[:KNOWS]->(c)
RETURN b, c
```

⇒ Reads: *"Find a node a with label person and name 'Jim'. Starting from a, find a neighbour node b via relation "KNOWS". Then find a neighbour node c of both a and b via relation "KNOWS".*
⇒ Short: Find mutual friends of Jim.

Cypher: MATCH Clause - Anonymous nodes

MATCH - example

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->()-[:KNOWS]->(c),
      (a)-[:KNOWS]->(c)
RETURN c
```

Same as before, but we are not interested in *b* this time.

Cypher: Return Options

- Cypher also provides various options to process the returned results
- They include options to aggregate, order, filter, and limit the returned data
- Example: `count(...)` allows us to return only the number of matched instances

RETURN Options - Example

RETURN Options - example

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (writer:Author {lastname:'Shakespeare'}),
      (theater) <-[:VENUE]- (:Performance) -[:PLAY_OF]-> (writer)
RETURN theater.city
```

Query: Cities with Shakespeare performances in theaters named “Theatre Royal”

RETURN Options - Example

RETURN Options - example

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (writer:Author {lastname:'Shakespeare'}),
      (theater) <-[:VENUE]- (:Performance) -[p:PLAY_OF]-> (writer)
RETURN theater.city AS city, count(p) AS play_count
```

Query: Shakespeare performances in theaters named “Theatre Royal” counting the number of plays
Note: identifiers can also be attached to relations

RETURN Options - Example

RETURN Options - example

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (writer:Author {lastname:'Shakespeare'}),
      (theater) <-[:VENUE]- (:Performance) -[p:PLAY_OF]-> (writer)
RETURN theater.city AS city, count(p) AS play_count
ORDER BY play_count DESC
```

Query: Shakespeare performances in theaters named “Theatre Royal” ordered by the number of plays.
Note: assign/rename variable names with `AS` to use them in `ORDER BY` clause

RETURN Options - Example

RETURN Options - example

```
MATCH (theater:Venue {name:'Theatre Royal'}),
      (writer:Author {lastname:'Shakespeare'}),
      (theater) <-[:VENUE]- (:Performance) -[p:PLAY_OF]-> (writer)
RETURN theater.city AS city, count(p) AS play_count
ORDER BY play_count DESC
LIMIT 1
```

Query: “Theatre Royal” with most Shakespeare plays

Cypher: `WHERE` Clause

▷ `WHERE` constrains graph matches by one/more of the following constraints:

- presence/absence of certain paths in the matched subgraphs
- certain labels for nodes
- certain names for relationships
- presence/absence of specific properties for matched nodes/relationships
- specific values for properties of matched nodes/relationships
- satisfaction of other constraints
  e.g. those performances must have occurred before a certain date

WHERE - Example

For example, we can query specifically for Shakespeare plays that were written *after* 1608 (Shakespeare's final period):

WHERE - example

```
MATCH (bard:Author {lastname:'Shakespeare'}),
      (play) <-[w:WROTE_PLAY]- (bard)
WHERE w.year > 1608
RETURN DISTINCT play.title AS play
```

Other Clauses

Cypher supports a variety of clauses:

- ▷ **CREATE** and **CREATE UNIQUE**
- ▷ **DELETE**
- ▷ **SET**
- ▷ **FOREACH**
- ▷ **UNION**
- ▷ **WITH**

Remember our previous example!

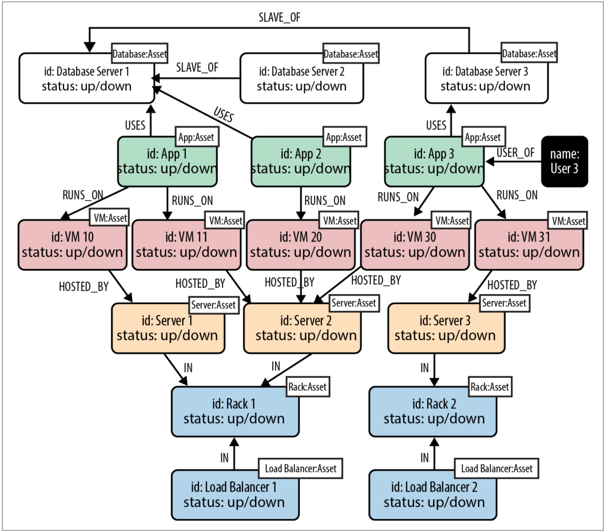


Figure 3-5. Example graph for the data center deployment scenario

Cypher - Design for Queryability Example

Remember the design for queryability design goal!

Goal: Design a query to find the cause behind an unresponsive application or service in our example graph.

Example Query

MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset

Cypher - Design for Queryability Example

The sample query we would need to define is:

Example Query

```
MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset
```

Cypher - Design for Queryability Example

The sample query we would need to define is:

Example Query

MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset

- ▷ Describes a variable length path between one and five relationships long
- ▷ There is no colon or relationship name between the square brackets ⇝ the relationships are unnamed
- ▷ There are no arrow-tips ⇝ relationships are undirected

Cypher - Design for Queryability Example

The sample query we would need to define is:

Example Query

```
MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset
```

Cypher - Design for Queryability Example

The sample query we would need to define is:

Example Query

MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset

- We start with the user who reported a problem
- We add asset nodes that have a status property with a value of 'down'
- Nodes which do not have a status property will not be added to the results

Cypher - Design for Queryability Example

The sample query we would need to define is:

Example Query

```
MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset
```

Cypher - Design for Queryability Example

The sample query we would need to define is:

Example Query

MATCH (user:User)-[*1..5]-(asset:Asset)
WHERE user.name = 'User3' AND asset.status = 'down'
RETURN DISTINCT asset

- Ensures that unique assets are returned in the results, no matter how many times they are matched

Sample Queries

Example 1

MATCH (p:Product {productName: "Chocolade"})
RETURN p.productName, p.unitPrice

Sample Queries

Example 1

MATCH (p:Product {productName: "Chocolade"})
RETURN p.productName, p.unitPrice

- Gets chocolates and their price
- Alternative:

Example 1 Alternative

MATCH (p:Product)
WHERE p.productName = "Chocolade"
RETURN p.productName, p.unitPrice

Sample Queries

Example 2

MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice **DESC**
LIMIT 10

Sample Queries

Example 2

MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice **DESC**
LIMIT 10

- Returns only a subset of attributes, in this case: ProductName and UnitPrice
- Orders by price
- Returns 10 most expensive items

Sample Queries

Example 3

```
MATCH  (p:Product {productName:'Chocolade'})
       <-[:PRODUCT]- (:Order)
       <-[:PURCHASED]- (c:Customer)
RETURN DISTINCT c.name
```

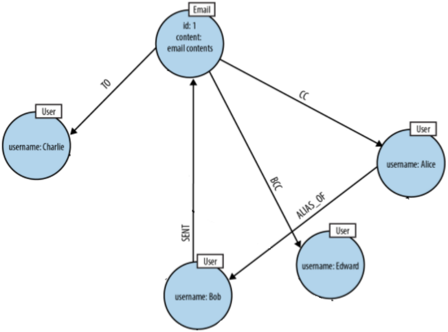
Sample Queries

Example 3

```
MATCH  (p:Product {productName:'Chocolade'})
       <-[:PRODUCT]- (:Order)
       <-[:PURCHASED]- (c:Customer)
RETURN DISTINCT c.name
```

▷ Names of everyone who bought *Chocolade*

Sample Queries

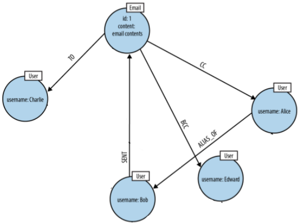


Example 4

MATCH (bob:User {username:'Bob'})-[:SENT]– >(email)-[:CC]– >(alias), (alias)-[:ALIAS_OF]– >(bob)
RETURN email.id

[3] Example and figure taken from [4]

Sample Queries



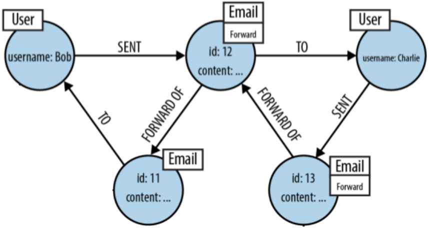
Example 4

MATCH (bob:User {username:'Bob'})-[:SENT]− >(email)-[:CC]− >(alias), (alias)-[:ALIAS_OF]− >(bob)
RETURN email.id

- ▷ Returns all emails that Bob has sent where he's CC'd one of his own aliases
- ▷ Returns 1 result: id:"1", content:"email content"

[3] Example and figure taken from [4]

Sample Queries

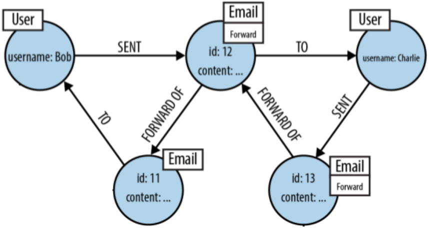


Example 5

MATCH (email:Email {id:'11'})< −[f:FORWARD_OF*]-(:Forward)
RETURN count(f)

[3] Example and figure taken from [4]

Sample Queries



Example 5

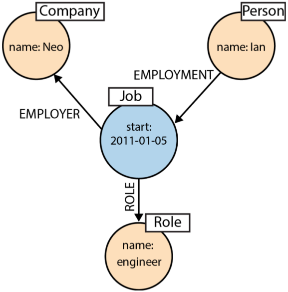
MATCH (email:Email {id:'11'})< –[f:FORWARD_OF*]-(:Forward)
RETURN count(f)

- ▷ This returns the number of times a particular email is forwarded
- ▷ The answer is 2 (count the number of `FORWARD_OF` relationships bound to `f`)

[3] Example and figure taken from [4]

Serialisation

Cypher as a Serialisation Format




```
CREATE (:Person {name:'Ian'})-[:EMPLOYMENT]->
       (employment:Job {start_date:'2011-01-05'})
       -[:EMPLOYER]-> (:Company {name:'Neo'}),
       (employment)-[:ROLE]-> (:Role {name:'engineer'})
```

[3] Image taken from [4]

GEOFF

- GEOFF = **Graph Export Object File Format**
- Is a text representation of a graph
- Based on Cypher
- A GEOFF document consists of a one or more subgraphs, each of which contains one or more paths
- Properties are in JSON syntax

```
(alice {"name":"Alice"})
(bob {"name":"Bob"})
(carol {"name":"Carol"})
(alice)<-[:KNOWS]->(bob)<-[:KNOWS]->(carol)<-[:KNOWS]->(alice)
```

GraphML

- XML-based file format for graphs
- Common format for exchanging graph structure data
- Generic (not limited to graph databases)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
[...]
    <graph id="G" edgedefault="undirected">
        <node id="n0">
            <data key="d0">green</data>
        </node>
        <node id="n1"/>
        <edge id="e0" source="n0" target="n1">
            <data key="d1">1.0</data>
        </edge>
    </graph>
</graphml>
```

Serialisation

- Essentially no standardisation in property graph community
- Serialisation is less important than in semantic technologies:
  - Not much emphasis on data exchange
  - Data publishing not commonly considered (in contrast to Linked Data)
  - Embedding in other formats usually not considered (in contrast to RDFa)

Converting and Comparing the Graph Models

RDF to PGM Transformation

Distinguish between two types of RDF triples:

- ▷ *attribute triples*
  = triples whose object is a literal
- ▷ *relationship triples*
  = triples whose object is an IRI or a blank node

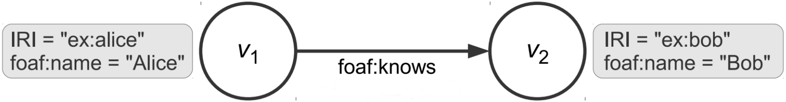
The transformation:

- ▷ Every relationship triple $\Rightarrow$ an edge
- ▷ Every attribute triple $\Rightarrow$ a property of the vertex for the subject of that triple
- ▷ IRI is preserved via its own property

Optional additional conversion:
Triples with predicate `rdf:type` can be used to assign labels to nodes (as the meaning of a type assignment resembles that of a label).

RDF to PGM Transformation

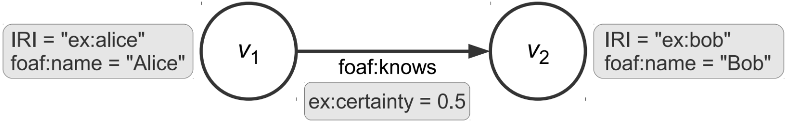
Simple transformation example:




```
ex:alice foaf:knows ex:bob .
ex:alice foaf:name "Alice" .
ex:bob foaf:name "Bob" .
```

RDF to PGM Transformation - Restrictions

Reification transformation example:




```
ex:alice foaf:name "Alice" .
ex:bob foaf:name "Bob" .
<<ex:alice foaf:knows ex:bob>> ex:certainty 0.5 .
<<ex:bob foaf:age 23>> ex:certainty 0.9 .
```

- ▷ We can not have labels for predicates in the RDF format (e.g. `ex:certainty=0.5` as above)
- ▷ A solution is to use Turtle* syntax (an extension of the RDF Turtle format)
- ▷ Turtle* embeds RDF triples into other RDF triples by enclosing the embedded triple in << and >> and use it as subject/object
- ▷ However the transformation of the last sentence is not possible
  See: `https://arxiv.org/pdf/1409.3288v2.pdf` for more details

PGM to RDF Transformation

- Relationships $\Rightarrow$ a (relationship) RDF triple
- Node properties (incl. their labels) $\Rightarrow$ an (attribute) RDF triple
- Relationship properties $\Rightarrow$ metadata triple whose subject is the triple for the corresponding edge
- Patterns for generating IRIs that denote edge labels and properties can be chosen freely

Knowledge Graph Formats

Characteristics	Relational	RDF	PGM
Standardised	yes	yes	no
Traversal performance	$-$	$\sim$	$+$
Large analytical queries	$+$	$\sim$	$-$
Query language	SQL	SPARQL	Cypher & more
Data Publication & Dereferencing	$-$	$+$	$-$
Global Identifiers & Cross dataset fusion	$-$	$+$	$-$

Legend:

$+$: good performance
$\sim$: medium performance
$-$: low performance

Summary

Covered in the lecture:

- Motivation for Model Graph Databases (Big Data, Connectivity)
- Comparing Relational and Property Graph Databases
- Labeled Property Graph Model
- Cypher Query Language
- "Unifying" RDF and Property Graphs
- Not covered: Hypergraph model (edges can have more then two vertices)

References

Neo4j.
https://neo4j.com/developer/graph-database/#property-graph.

O. Hartig.
Reconciliation of rdf* and property graphs.
arXiv preprint arXiv:1409.3288, 2014.

T. Ivarsson.
Graph database and neo4j.
http://www.slideshare.net/thobe/nosqleu-graph-databases-and-neo4j.

I. Robinson, J. Webber, and E. Eifrem.
Graph Databases: New Opportunities for Connected Data.
" O'Reilly Media, Inc.", 2nd edition, 2015.