Property Graphs

The Property Graph Data Model

- Born in the database community
  - Meant to be queried and processed
  - **<u>THERE IS NOTHING SUCH A STANDARD!</u>**
- Two main constructs: nodes and edges
  - Nodes represent entities,
  - Edges relate pairs of nodes, and may represent different types of relationships
- Nodes and edges might be labeled,
- and may have a set of properties represented as attributes (key-value pairs)***
- Further assumptions:
  - Edges are directed,
  - Multi-graphs are allowed

**** Note: in some definitions (the least) edges are not allowed to have attributes*

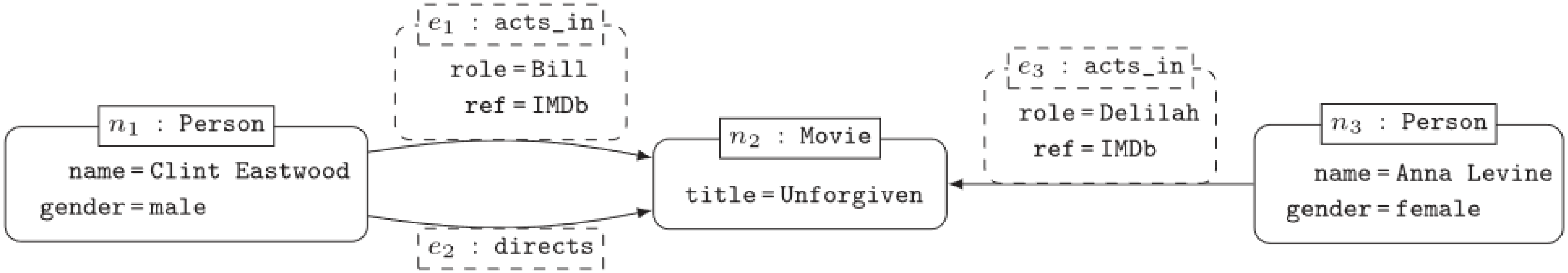
Formal Definition

Definition 2.3 (Property graph). A property graph G is a tuple $(V, E, \rho, \lambda, \sigma)$, where:

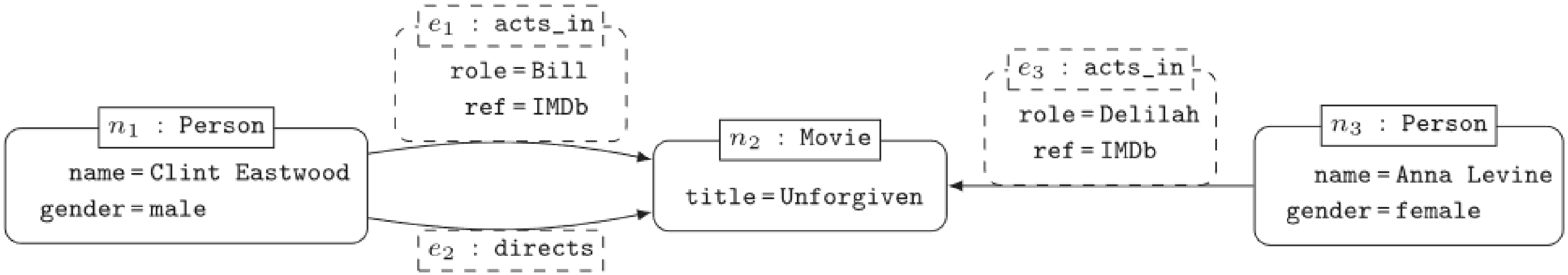
(1) V is a finite set of *vertices* (or *nodes*).
(2) E is a finite set of *edges* such that V and E have no elements in common.
(3) $\rho : E \to (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ indicates that e is a directed edge *from* node v_1 *to* node v_2 in G.
(4) $\lambda : (V \cup E) \to Lab$ is a total function with Lab a set of labels. Intuitively, if $v \in V$ (respectively, $e \in E$) and $\lambda(v) = \ell$ (respectively, $\lambda(e) = \ell$), then ℓ is the label of node v (respectively, edge e) in G.
(5) $\sigma : (V \cup E) \times Prop \to Val$ is a partial function with $Prop$ a finite set of properties and Val a set of values. Intuitively, if $v \in V$ (respectively, $e \in E$), $p \in Prop$ and $\sigma(v, p) = s$ (respectively, $\sigma(e, p) = s$), then s is the value of property p for node v (respectively, edge e) in the property graph G.

Extracted from: R. Angles et al. Foundations of Modern Query Languages for Graph Databases

Example of Property Graph



Example of Property Graph



Formal definition:

$$V = \{n_1, n_2, n_3\} \qquad E = \{e_1, e_2, e_3\}$$

$$\rho(e_1) = (n_1, n_2) \qquad \rho(e_2) = (n_1, n_2)$$
$$\rho(e_3) = (n_3, n_2)$$

$$\lambda(n_1) = \texttt{Person} \qquad \lambda(n_2) = \texttt{Movie}$$
$$\lambda(n_3) = \texttt{Person} \qquad \lambda(e_1) = \texttt{acts\_in}$$
$$\lambda(e_2) = \texttt{directs} \qquad \lambda(e_3) = \texttt{acts\_in}$$

$$\begin{aligned}
\sigma(n_1, \texttt{name}) &= \texttt{Clint Eastwood} \\
\sigma(n_1, \texttt{gender}) &= \texttt{male} \\
\sigma(n_2, \texttt{title}) &= \texttt{Unforgiven} \\
\sigma(n_3, \texttt{name}) &= \texttt{Anna Levine} \\
\sigma(n_3, \texttt{gender}) &= \texttt{female} \\
\sigma(e_1, \texttt{role}) &= \texttt{Bill} \\
\sigma(e_1, \texttt{ref}) &= \texttt{IMDb} \\
\sigma(e_3, \texttt{role}) &= \texttt{Delilah} \\
\sigma(e_3, \texttt{ref}) &= \texttt{IMDb}
\end{aligned}$$

Traversal Navigation

- We define the graph traversal pattern as: *“the ability to rapidly traverse structures to an arbitrary depth (e.g., tree structures, cyclic structures) and with an arbitrary path description (e.g. friends that work together, roads below a certain congestion threshold)”* [Marko Rodriguez]

Traversal Navigation

- We define the graph traversal pattern as: *"the ability to* ***rapidly*** *traverse structures to an* ***arbitrary depth*** *(e.g., tree structures, cyclic structures) and with an* ***arbitrary path description*** *(e.g. friends that work together, roads below a certain congestion threshold)"* [Marko Rodriguez]

Traversal Navigation

- We define the graph traversal pattern as: *"the ability to **rapidly** traverse structures to an **arbitrary depth** (e.g., tree structures, cyclic structures) and with an **arbitrary path description** (e.g. friends that work together, roads below a certain congestion threshold)"* [Marko Rodriguez]
- Totally opposite to set theory (on which relational databases are based on)
  - Sets of elements are operated by means of the relational algebra

Traversing Data in a RDBMS

- In the relational theory, it is equivalent to joining data (schema level) and select data (based on a value)

User	Address	Phone	Email	Alternate
Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
Bob	456 Bar Ave		bob@example.org	
...	...	...	...	...
Zach	99 South St.		zach@example.org	

User	OrderId
Alice	1234
Alice	5678
Alice	9012
...	...
Zach	5588

OrderId	LineItemId
1234	abcd
1234	efab
...	...
5588	bafd

LineItemId	Description	Handling
abcd	strawberry ice cream	freezer
efab	potatoes	
...	...	
bafd	dried spaghetti	

Traversing Data in a RDBMS

- In the relational theory, it is equivalent to joining data (schema level) and select data (based on a value)

User	Address	Phone	Email	Alternate
Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
Bob	456 Bar Ave		bob@example.org	
…	…	…	…	…
Zach	99 South St.		zach@example.org	

User	OrderId
Alice	1234
Alice	5678
Alice	9012
…	…
Zach	5588

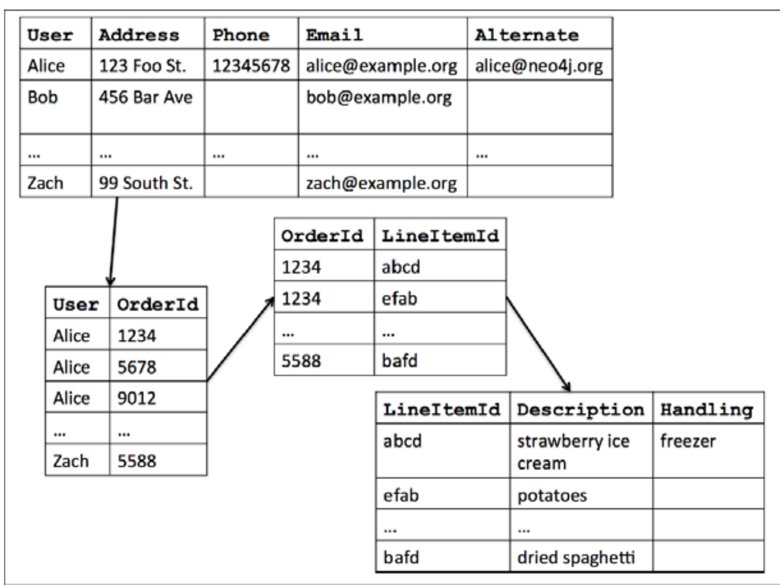
OrderId	LineItemId
1234	abcd
1234	efab
…	…
5588	bafd

LineItemId	Description	Handling
abcd	strawberry ice cream	freezer
efab	potatoes	
…	…	
bafd	dried spaghetti	

```
SELECT *
FROM user u, user_order uo,
orders o, items i
WHERE u.user = uo.user AND
uo.orderId = o.orderId AND
i.lineItemId = i.LineItemId
AND u.user = 'Alice'
```

Traversing Data in a RDBMS

- In the relational theory, it is equivalent to joining data (schema level) and select data (based on a value)



User	Address	Phone	Email	Alternate
Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
Bob	456 Bar Ave		bob@example.org	
...	...	...	...	...
Zach	99 South St.		zach@example.org	

User	OrderId
Alice	1234
Alice	5678
Alice	9012
...	...
Zach	5588

OrderId	LineItemId
1234	abcd
1234	efab
...	...
5588	bafd

LineItemId	Description	Handling
abcd	strawberry ice cream	freezer
efab	potatoes	
...	...	
bafd	dried spaghetti	

```
SELECT *
FROM user u, user_order uo,
orders o, items i
WHERE u.user = uo.user AND
uo.orderId = o.orderId AND
i.lineItemId = i.LineItemId
AND u.user = ‘Alice’
```

Cardinalities:

|User|: 5.000.000
|UserOrder|: 100.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Traversing Data in a RDBMS

- In the relational theory, it is equivalent to joining data (schema level) and select data (based on a value)

User	Address	Phone	Email	Alternate
Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
Bob	456 Bar Ave		bob@example.org	
...	...	...	...	...
Zach	99 South St.		zach@example.org	

User	OrderId
Alice	1234
Alice	5678
Alice	9012
...	...
Zach	5588

OrderId	LineItemId
1234	abcd
1234	efab
...	...
5588	bafd

LineItemId	Description	Handling
abcd	strawberry ice cream	freezer
efab	potatoes	
...	...	
bafd	dried spaghetti	

```
SELECT *
FROM user u, user_order uo,
orders o, items i
WHERE u.user = uo.user AND
uo.orderId = o.orderId AND
i.lineItemId = i.LineItemId
AND u.user = ‘Alice’
```

Cardinalities:

|User|: 5.000.000
|UserOrder|: 100.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Query Cost?!

Activity

- *Wear your data steward hat and discuss in pairs the database tuning that would guarantee the most efficient access plan for this query*
  - *What join algorithm would you take? Why?*

User	Address	Phone	Email	Alternate
Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
Bob	456 Bar Ave		bob@example.org	
...	...	...	...	...
Zach	99 South St.		zach@example.org	

User	OrderId
Alice	1234
Alice	5678
Alice	9012
...	...
Zach	5588

OrderId	LineItemId
1234	abcd
1234	efab
...	...
5588	bafd

LineItemId	Description	Handling
abcd	strawberry ice cream	freezer
efab	potatoes	
...	...	
bafd	dried spaghetti	

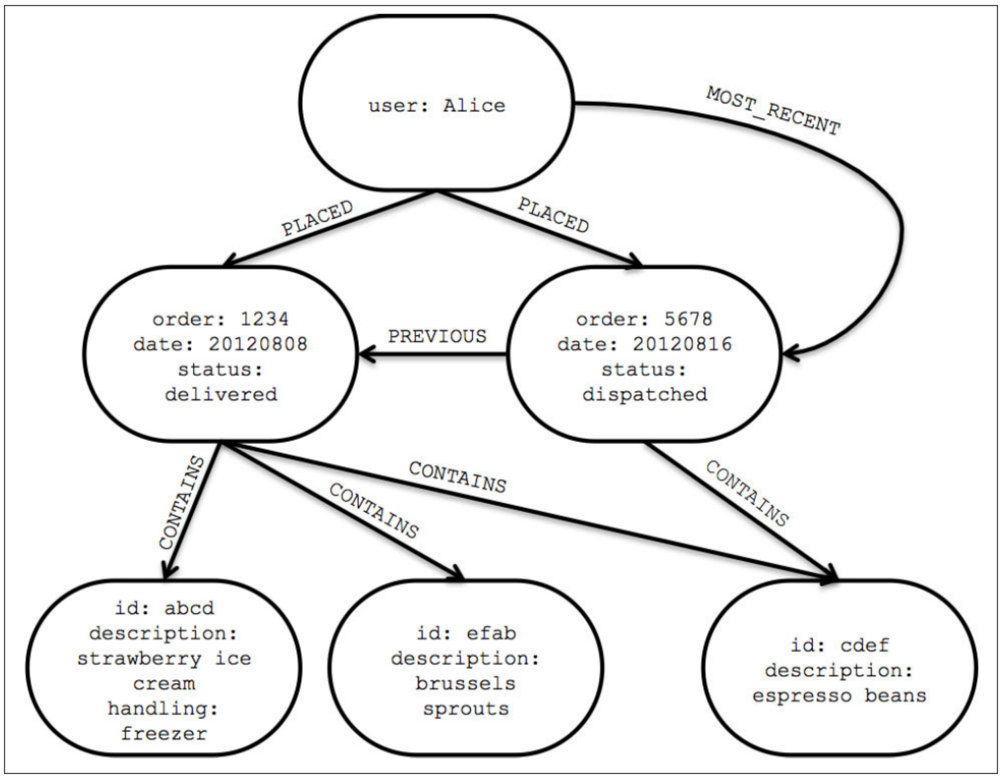
```
SELECT *
FROM user u, user_order uo,
orders o, items i
WHERE u.user = uo.user AND
uo.orderId = o.orderId AND
i.lineItemId = i.LineItemId
AND u.user = 'Alice'
```

Cardinalities:

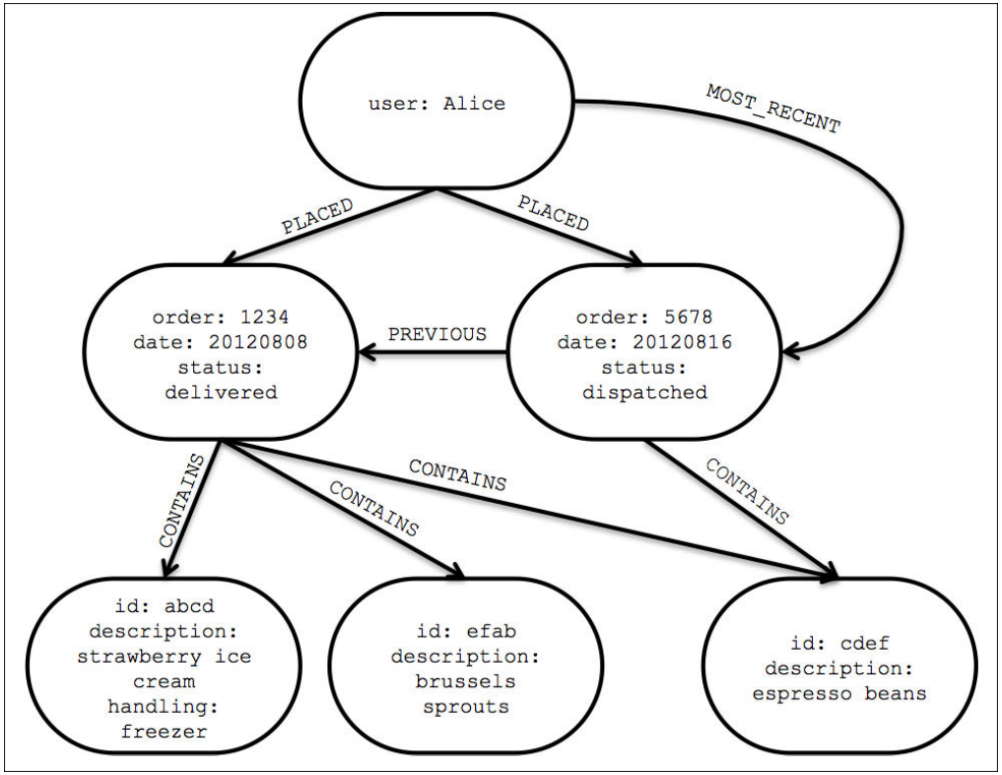
|User|: 5.000.000
|UserOrder|: 100.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Query Cost?!

Traversing Data in a Graph Database



Traversing Data in a Graph Database

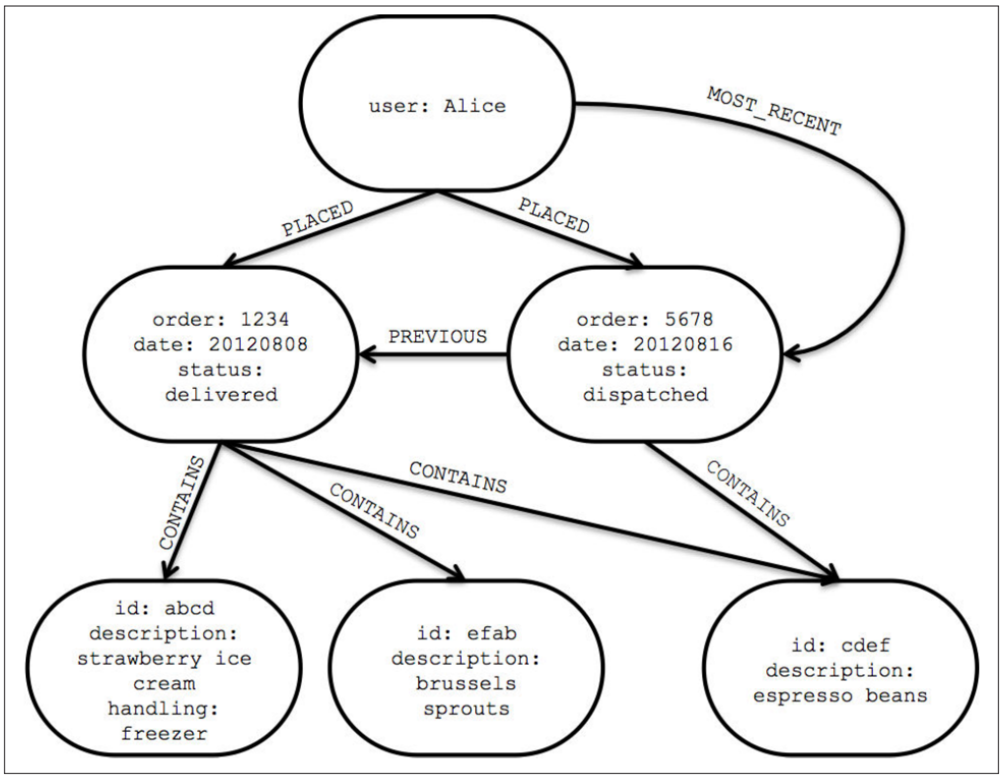


Cardinalities:

|User|: 5.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Activity

- *What would be the cost of this query in a graph database?*

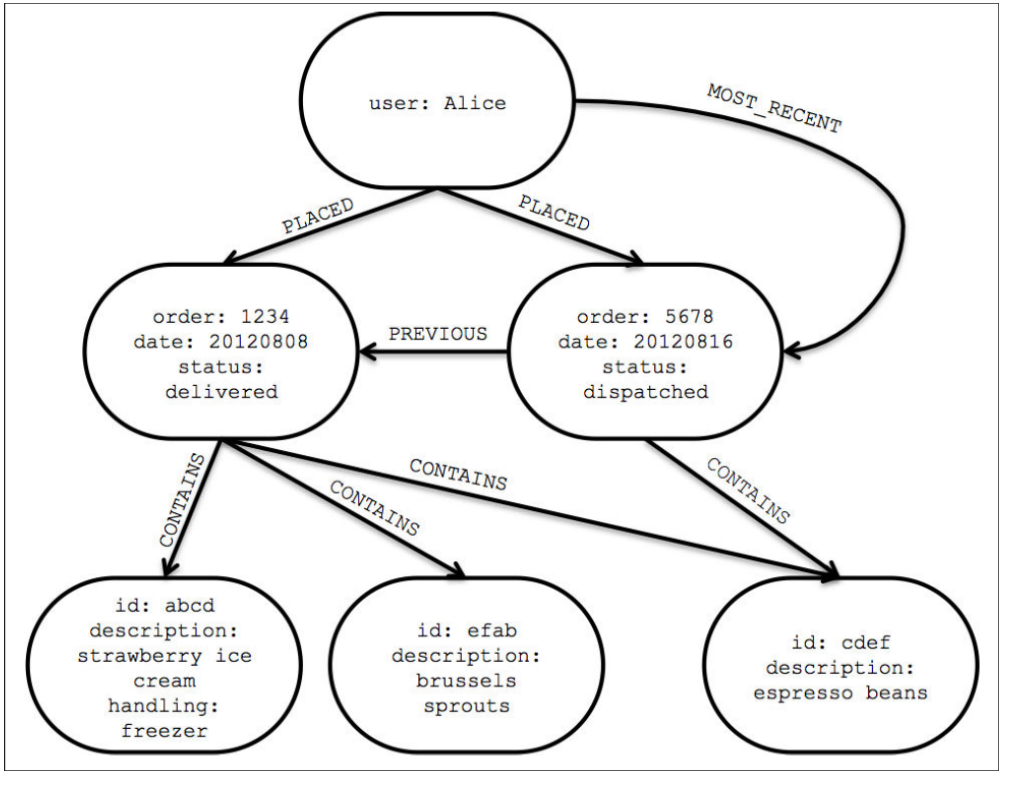


Cardinalities:

|User|: 5.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Activity

- *What would be the cost of this query in a graph database?*



Cardinalities:

|User|: 5.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Query Cost?!

Traversing Property Graphs

- Traversing graph data depends on two main variables
  - The size of the graph (i.e., #edges),
  - The topology of the graph,
  - The query topology

REFRESHING SOME BASICS ON GRAPHS

GRAPH OPERATIONS

Graph Operations

- Content-based queries
  - The value is relevant
    - Get a node, get the value of a node / edge attribute, etc.
    - A typical case are summarization queries (i.e., aggregations)
- Topological queries
  - Only the graph topology is considered
  - Typically, several business problems (such as fraud detection, trend prediction, product recommendation, network routing or route optimization) are solved using graph algorithms exploring the graph topology
    - Computing the betweenness centrality of a node…
      - in a social network, an analyst can detect influential people or groups for targeting a marketing campaign audience.
      - in a telecommunication operator, an analyst may detect central nodes of an antenna network and optimize the routing and load balancing across the infrastructure accordingly
- Hybrid approaches

Topological Queries

- Divided in three (four) main categories
  - Adjacency,
  - Reachability,
  - Pattern Matching,
  - [Graph metrics]

Adjacency Queries

- Formalized as node adjacency or edge incidence
  - Node adjacency
  - Edge incidence (node degree, out-degree, in-degree)
  - K-neighbourhood of a node
- <u>Computational cost</u>: linear cost on the number of edges to visit
- <u>Examples</u>:
  - Find all friends of a person
  - Airports with a direct connection
  - Movies watched by a person
  - Products bought by a customer
  - ...

Reachability Queries

- Formalized as traversal queries
  - Fixed-length paths (fixed #edges and nodes)
  - Regular simple paths (restrictions as regular expressions)
    - Hybrid if the restriction is in the *content*
  - Shortest path
- <u>Computational cost</u>: hard to compute for large graphs
  - Shortest-path (Dijkstra's algorithm): **O(|V²|)**
    - Smarter implementations based on priority queues yield **O(|E|*|V|log|V|)** complexity
- <u>Examples</u>:
  - Friend-of-a-friend
  - Flight connections
  - Logistics (goods distribution)
  - Items bought in a user orders
  - ...

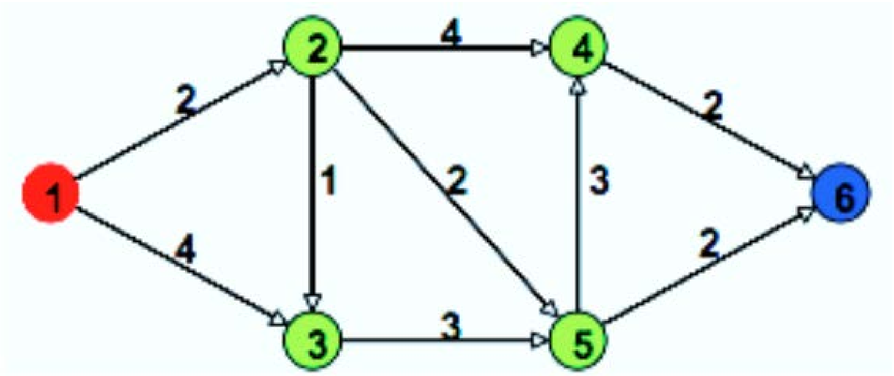
Single-Source Shortest-Path

- Dijkstra's algorithm
  - Main idea:
    - Optimal substructure: The subpath of any shortest path is itself a shortest path
    - Triangle inequality: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$, if u,v is the shortest path
  - Input:
    - A weighted graph **G = (V,E),**
    - A source vertex $\mathbf{V_s} \in \mathbf{V}$
  - Internal structures:
    - **S**: Set of vertices whose shortest paths from the source have already been determined,
    - **Q (V-S)**: Remaining vertices,
    - **d**: current estimated shortest paths to each vertex,
    - [**pre**: array de predecessors for each vertex] – traceback
  - Output:
    - The graph representing all the paths from one vertex to all the others must be a spanning tree
    - There will be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex

Pseudocode

```
shortest_paths( Graph g, Node s )
    initialise_single_source( g, s )
    S := { 0 }          /* Make S empty */
    Q := Vertices( g ) /* Put the vertices in a PQ */
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )

 relax( Node u, Node v, double w[][] )
     if d[v] > d[u] + w[u,v] then
        d[v] := d[u] + w[u,v]
        pi[v] := u
```



Label-constrained Reachability

- Definition:

$$G_L^* = \{(s,t) \mid \text{there is a path in } G \text{ from } s \text{ to } t \text{ using only edges with labels in } L\}$$

Label-constrained Reachability

- Definition:

$$G_L^* = \{(s,t) \mid \text{there is a path in } G \text{ from } s \text{ to } t \text{ using only edges with labels in } L\}$$

- It is equivalent to determine whether or not there is a path in G from s to t such that the concatenation of the edge labels along the path forms a string in the language denoted by the regular expression $(\ell_1 \cup \cdots \cup \ell_n)^*$

where: $L = \{\ell_1, \ldots, \ell_n\}$, U *disjunction and * the Kleene star*

Label-constrained Reachability

- Definition:

$$G_L^* = \{(s,t) \mid \text{there is a path in } G \text{ from } s \text{ to } t \text{ using only edges with labels in } L\}$$

- It is equivalent to determine whether or not there is a path in G from s to t such that the concatenation of the edge labels along the path forms a string in the language denoted by the regular expression $(\ell_1 \cup \cdots \cup \ell_n)^*$

  where: $L = \{\ell_1, \ldots, \ell_n\}$, U *disjunction and * the Kleene star*

- Typically, the allowed topology and labels involved are expressed as a regular expression
  - In general, the cost of regular label-constrained queries is known to be NP-complete

Pattern Matching

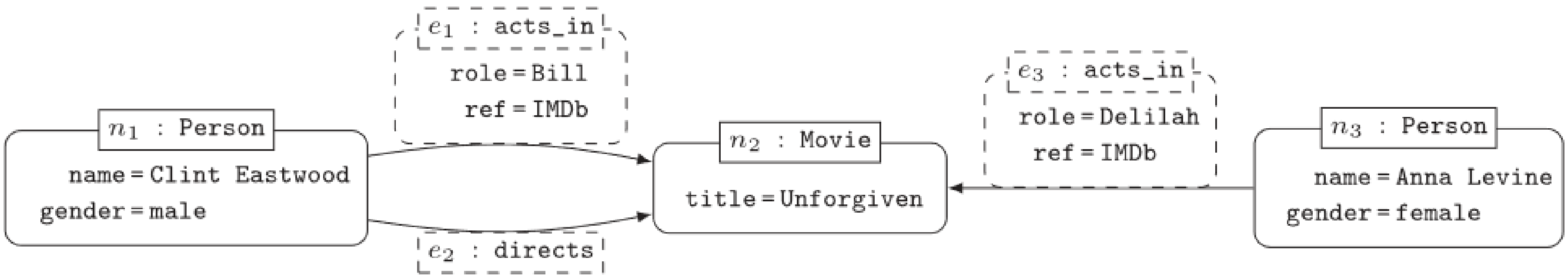
- Formalized as the graph isomorphism problem
  - Input: property graph G, and a pattern graph P
  - Output: all sub-graphs of G that are isomorphic to P
- <u>Computational cost</u>: hard to compute, in general, NP-complete
- Examples:
  - Group of cities all of them directly connected by flights
  - People without telephone
  - People who have watched sci-fi movies in the last month
  - …

Property Graph Patterns

- Based on *basic graph patterns* (bgps)
  - Equivalent to conjunctive queries
- A *bgp* for querying property graphs is a property graph where variables can appear in place of any constant (labels / properties)
- A **match** for a *bgp* is a mapping from variables to constants such that when the mapping is applied to the *bgp*, the result is *contained* within the original graph
- The **results** for a *bgp* are then all mappings from variables in the query to constants that comprise a match

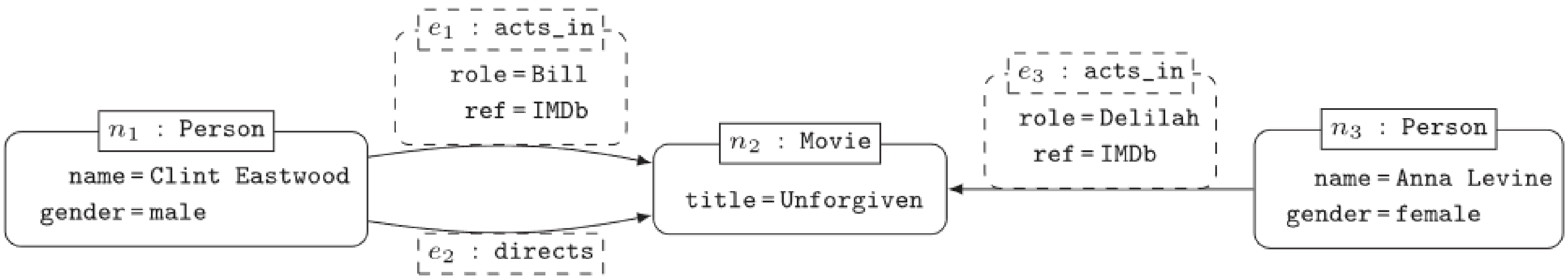
Example of Graph Pattern

Graph:

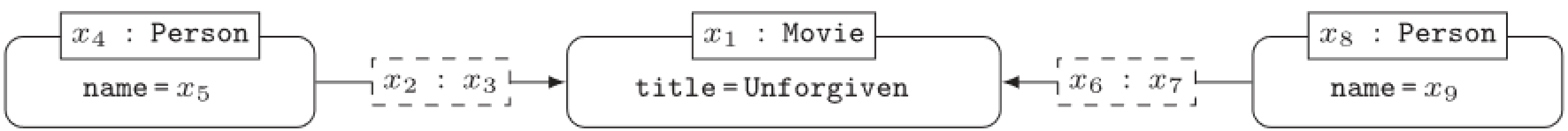


Example of Graph Pattern

Graph:



BGP:



Evaluating Graph Patterns

- Evaluating a *bgp* **Q** against a graph database **G** corresponds to listing all possible matches of **Q** with respect to **G**
- Formally:

Definition 3.5 (Match). Given an edge-labelled graph $G = (V, E)$ and a bgp $Q = (V', E')$, a *match* h of Q in G is a mapping from $Const \cup Var$ to $Const$ such that:

(1) for each constant $a \in Const$, it is the case that $h(a) = a$; that is, the mapping maps constants to themselves; and
(2) for each edge $(b, l, c) \in E'$, it holds that $(h(b), h(l), h(c)) \in E$; this condition imposes that (a) each edge of Q is mapped to an edge of G, and (b) the structure of Q is preserved in its image under h in G (that is, when h is applied to all the terms in Q, the result is a sub-graph of G).

Extracted from: R. Angles et al. Foundations of Modern Query Languages for Graph Databases

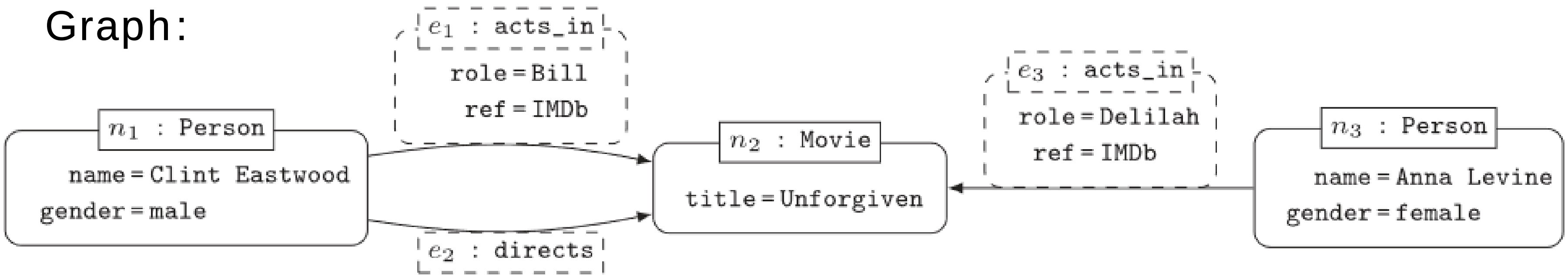
Semantics of a Match

- **Homomorphism-based semantics**: the previous definition maps to a homomorphism from **Q** to **G**
  - Multiple variables in **Q** can map to the same term in **G**
  - Corresponds to the familiar semantics of select-from-where queries (conjunctive queries) in relational databases
- **Isomorphism-based semantics**: an additional constraint is added, the match function *h* must be injective. Still, different semantics can be applied:
  - Strict isomorphism (no-repeated-anything): *h* is injective
  - No repeated-node semantics: *h* is only injective for nodes
  - No repeated-edge semantics: h is only injective for edges

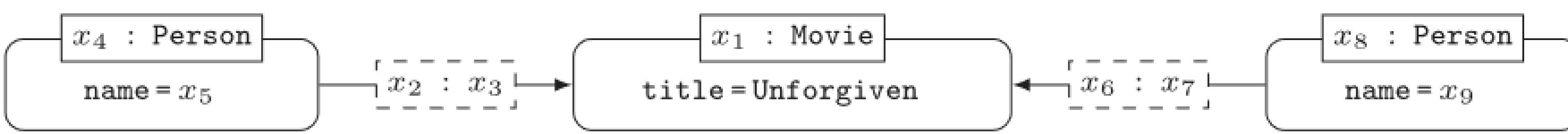
Activity

- Objective: Understand the differences between graph matching isomorphism-based and homomorphism-based semantics
  - Given the following graph, bgp and potential results...

Graph:



BGP:



Activity

- *Objective: Understand the differences between graph matching isomorphism-based and homomorphism-based semantics*
  - *Given the following graph, bgp and potential results...*

Results:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
n_2	e_2	directs	n_1	Clint Eastwood	e_3	acts_in	n_3	Anna Levine
n_2	e_3	acts_in	n_3	Anna Levine	e_2	directs	n_1	Clint Eastwood
n_2	e_1	acts_in	n_1	Clint Eastwood	e_3	acts_in	n_3	Anna Levine
n_2	e_3	acts_in	n_3	Anna Levine	e_1	acts_in	n_1	Clint Eastwood
n_2	e_2	directs	n_1	Clint Eastwood	e_1	acts_in	n_1	Clint Eastwood
n_2	e_1	acts_in	n_1	Clint Eastwood	e_2	directs	n_1	Clint Eastwood
n_2	e_1	acts_in	n_1	Clint Eastwood	e_1	acts_in	n_1	Clint Eastwood
n_2	e_2	directs	n_1	Clint Eastwood	e_2	directs	n_1	Clint Eastwood
n_2	e_3	acts_in	n_1	Anna Levine	e_3	acts_in	n_1	Anna Levine

Circle what results would be obtained if applying isomorphism-based or homomorphism-based semantics

From Intractable to Tractable Matching

- These results apply to property graphs:
  - Graph isomorphism is known to be NP-complete in the worst case
  - Graph homomorphism is also known to be NP-complete in the worst case
  - However, *graph simulation and bi-simulation,* a relaxed form of graph homomorphism, can be computed within polynomial time
    - It might still hard to compute for large graphs
    - New iterative algorithms allow to scale-well

Fan et al. Graph Pattern Matching: From Intractable to Polynomial Time

Graph Metrics

- They can be formalized either as adjacency, reachability or pattern matching
  - Thus, the cost depends on how the metric is formalized
- Given their relevance, they are typically provided as built-in functions
- <u>Examples</u>:
  - Graph node order,
  - the min / max degree in the graph,
  - the length of a path,
  - the graph diameter,
  - the graph density,
  - closeness / betwenness of a node,
  - the pageRank of a node,
  - ...

Topological Queries

- Support provided by current GBDs

	Adjacency		Reachability				
Graph Database	Node/edge adjacency	k-neighborhood	Fixed-length paths	Regular simple paths	Shortest path	Pattern matching	Summarization
Allegro	•		•			•	
DEX	•		•	•	•	•	
Filament	•		•			•	
G-Store	•		•	•	•	•	
HyperGraph	•					•	
Infinite	•		•	•	•	•	
Neo4j	•		•	•	•	•	
Sones	•					•	
vertexDB	•		•	•		•	

R. Angles. A Comparison of Current Graph Database Models (as of 2012)

Implementation of the Operations

- Note that the operations presented are conceptual: agnostic of the technology
- The implementation of the ops depends on:
  - The graph database implementation,
  - The operation implementation,
  - The pattern (in case of pattern matching)

Implementation of the Operations

- Note that the operations presented are conceptual: agnostic of the technology
- The implementation of the ops depends on:
  - The graph database implementation,
  - The operation implementation,
  - The pattern (in case of pattern matching)

Summary

- Property graphs do not have a defined standard, but there is a de-facto standard
  - Nodes / edges may have labels (equivalent to the concept of typing) and / or properties (to the concept of attributes)
- The basic operations on graphs are:
  - Adjacency queries,
  - Reachability queries and
  - Pattern matching
- However, their traditional definition needs to be redefined for property graphs, yielding different computational complexities
  - It basically depends on the graph topology, graph pattern and internal structures the graph database is implemented on