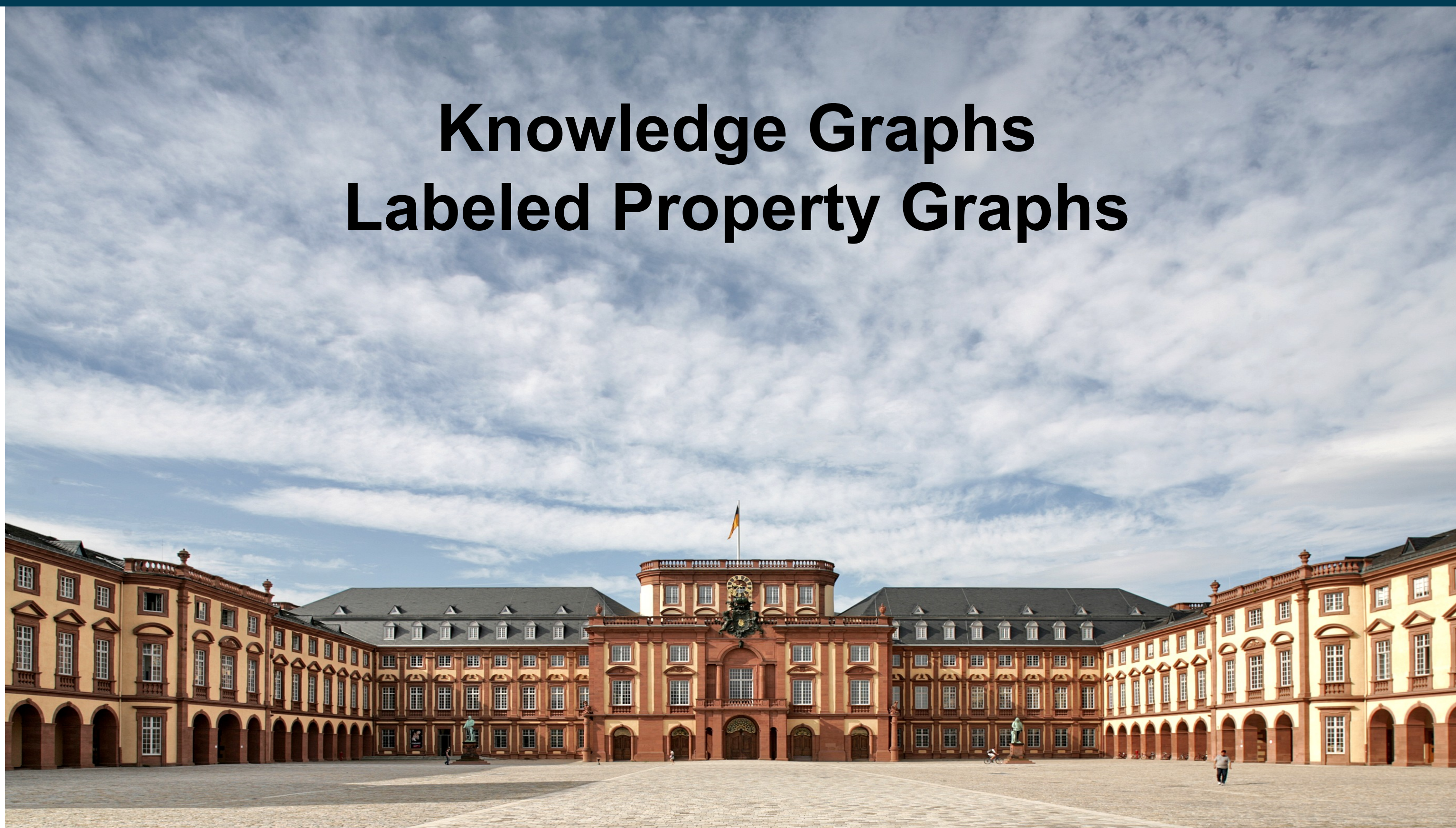
UNIVERSITÄT
MANNHEIM



Knowledge Graphs
Labeled Property Graphs

Heiko Paulheim

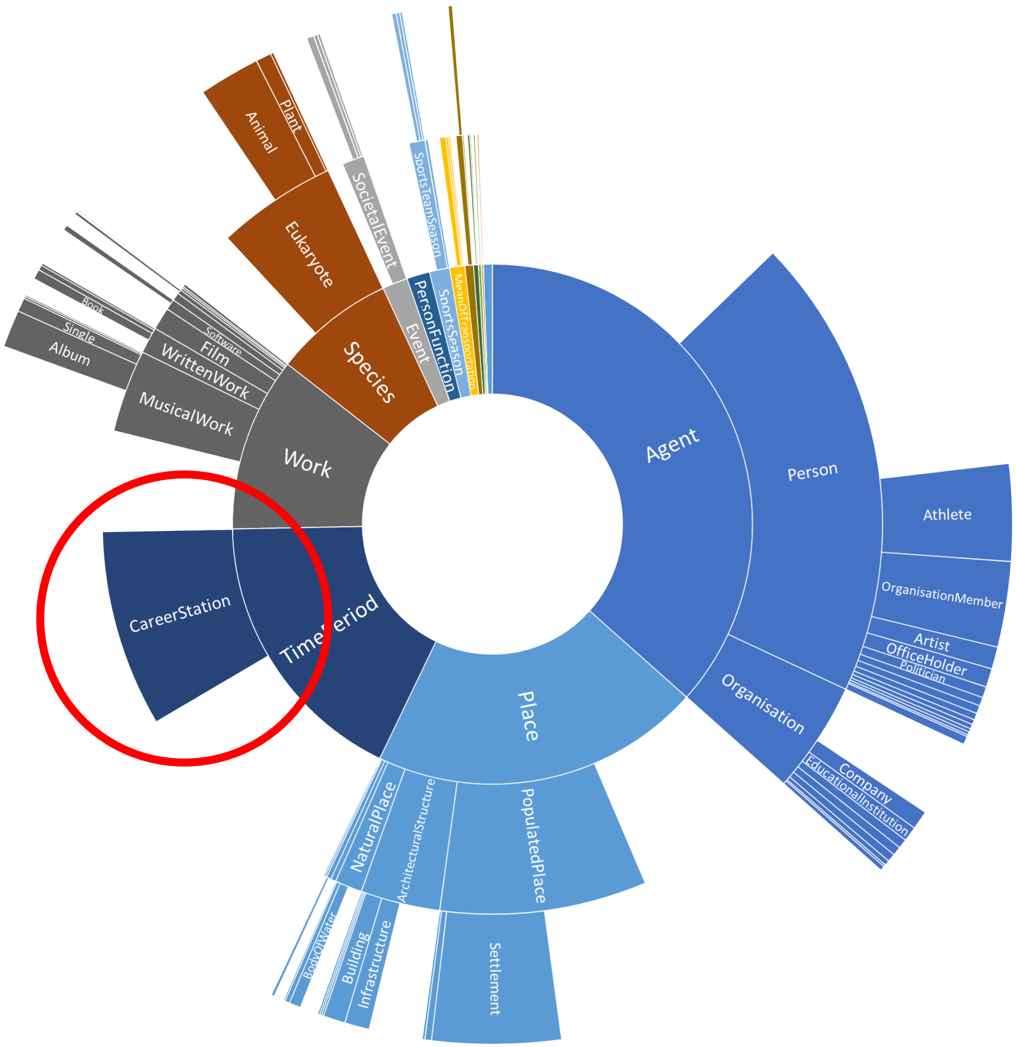
Previously on “Knowledge Graphs”

- Principles:
  - RDF, RDF-S, SPARQL & co
  - Public Knowledge Graphs

- Today:
  - Some modeling shortcomings of RDF
  - Labeled Property Graphs as an alternative
  - RDF*/SPARQL*
  - Cypher

Previously on “Knowledge Graphs”

- Classes in DBpedia
  - What’s a CareerStation?



Verbosity of RDF Graphs

- Example from DBpedia:
  - Modeling careers of athletes

- Observation:
  - The information is more complex than pure triples

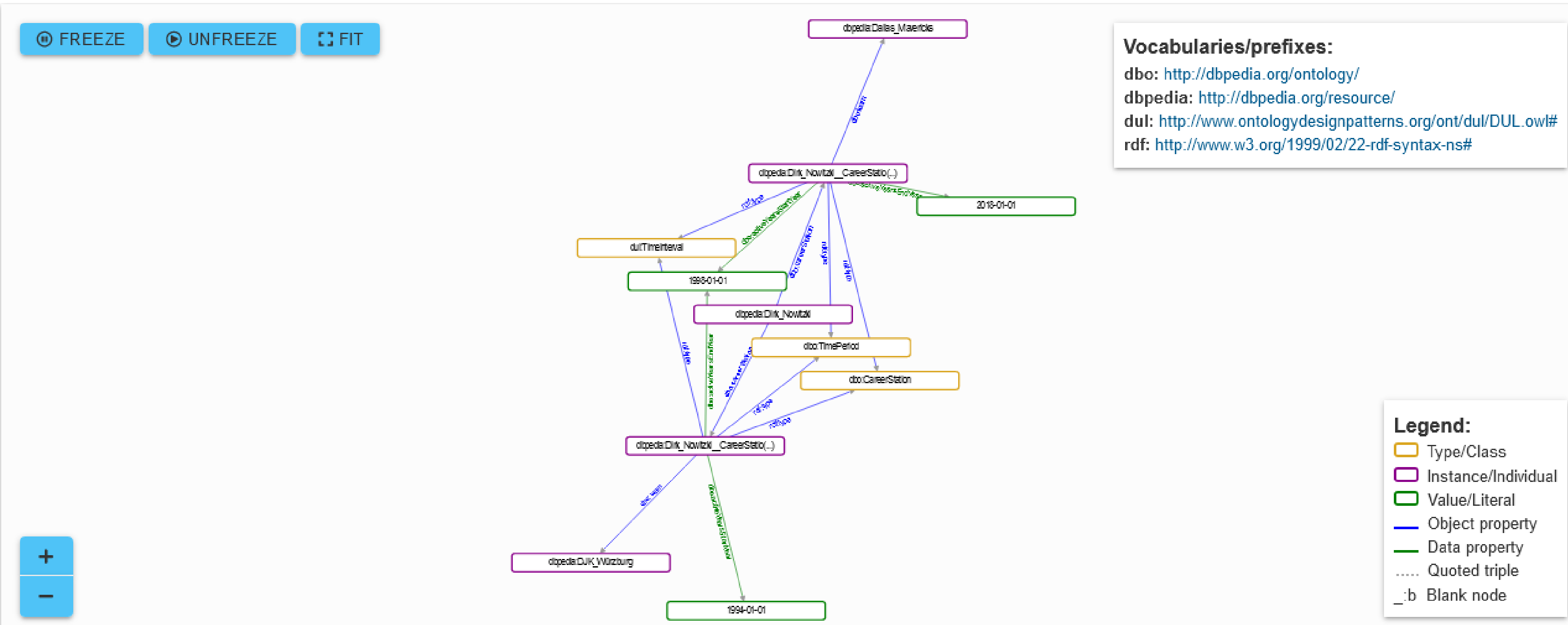
Verbosity of RDF Graphs

- Each career station adds one entity and ~seven statements

```
dbr:Dirk_Nowitzki dbo:careerStation dbr:Dirk_Nowitzki__CareerStation__1 .
dbr:Dirk_Nowitzki__CareerStation__1
        rdf:type dbo:CareerStation,
                dbo:TimePeriod,
                dul:TimeInterval ;
        dbo:activeYearsEndYear    "1998"^^xsd:gYear ;
        dbo:activeYearsStartYear  "1994"^^xsd:gYear ;
        dbo:team dbr:DJK_Würzburg .
dbr:Dirk_Nowitzki dbo:careerStation dbr:Dirk_Nowitzki__CareerStation__2 .
dbr:Dirk_Nowitzki__CareerStation__2
        rdf:type dbo:CareerStation,
                dbo:TimePeriod,
                dul:TimeInterval ;
        dbo:activeYearsEndYear    "2018"^^xsd:gYear ;
        dbo:activeYearsStartYear  "1998"^^xsd:gYear ;
        dbo:team dbr:Dallas_Mavericks .
```

Verbosity of RDF Graphs

- Each career station adds one entity and ~seven statements



Visualization: https://issemantic.net/rdf-visualizer

Verbosity of RDF Graphs

- Example from DBpedia:
  - ~2.7M nodes of type dbo:CareerStation*
    - ~45% of all entities!
  - 13.5M RDF statements describe those nodes

* As of October 2023

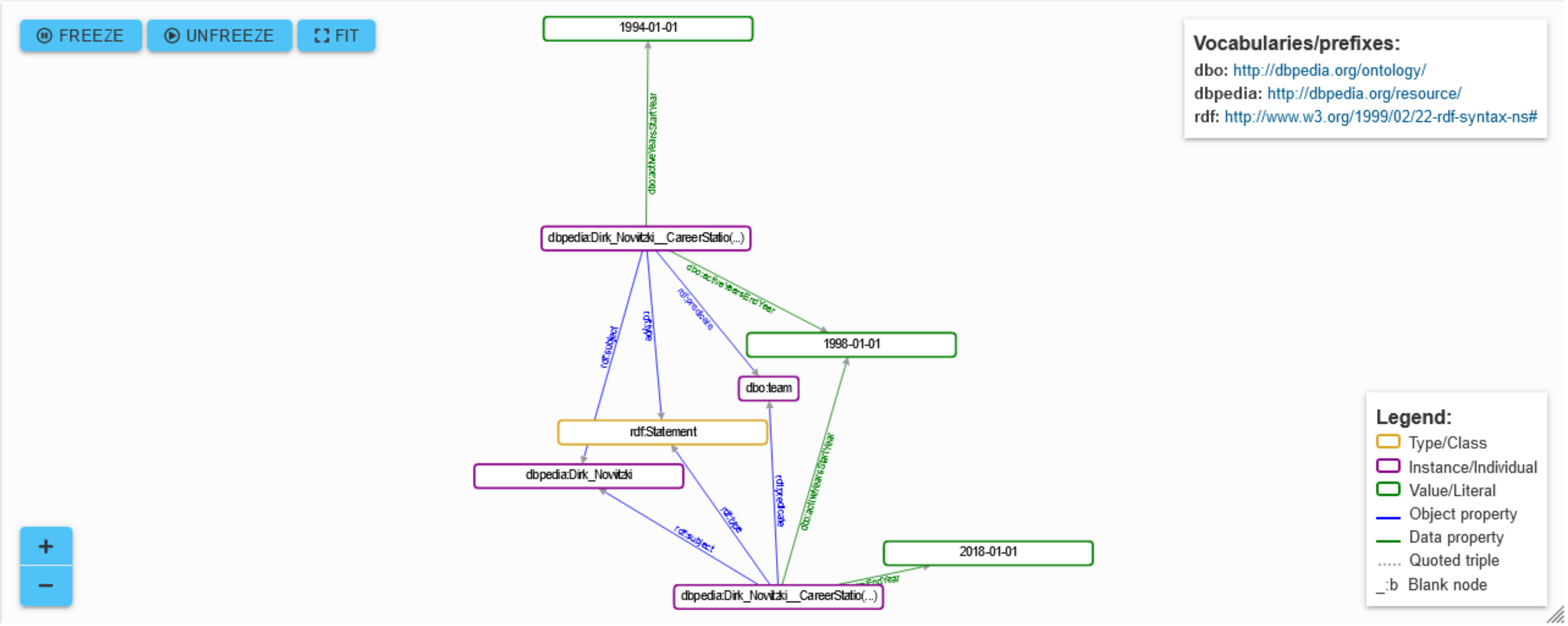
Verbosity of RDF Graphs

- Alternatives:
  - RDF Reification

```
dbr:Dirk_Nowitzki__CareerStation__1
        rdf:type rdf:Statement ;
        rdf:subject dbr:Dirk_Nowitzki ;
        rdf:predicate dbo:team ;
        rdf:object dbr:DJK_Würzburg .
dbr:Dirk_Nowitzki__CareerStation__1
        dbo:activeYearsEndYear   "1998"^^xsd:gYear ;
        dbo:activeYearsStartYear "1994"^^xsd:gYear .
dbr:Dirk_Nowitzki__CareerStation__2
        rdf:type rdf:Statement ;
        rdf:subject dbr:Dirk_Nowitzki ;
        rdf:predicate dbo:team ;
        rdf:object dbr:Dallas_Mavericks .
dbr:Dirk_Nowitzki__CareerStation__2
        dbo:activeYearsEndYear   "2018"^^xsd:gYear ;
        dbo:activeYearsStartYear "1998"^^xsd:gYear .
```

Verbosity of RDF Graphs

- Alternatives:
  - RDF Reification



Visualization: https://issemantic.net/rdf-visualizer

Verbosity of RDF Graphs

- Alternatives:
  - RDF Named Graphs (e.g., TriG)

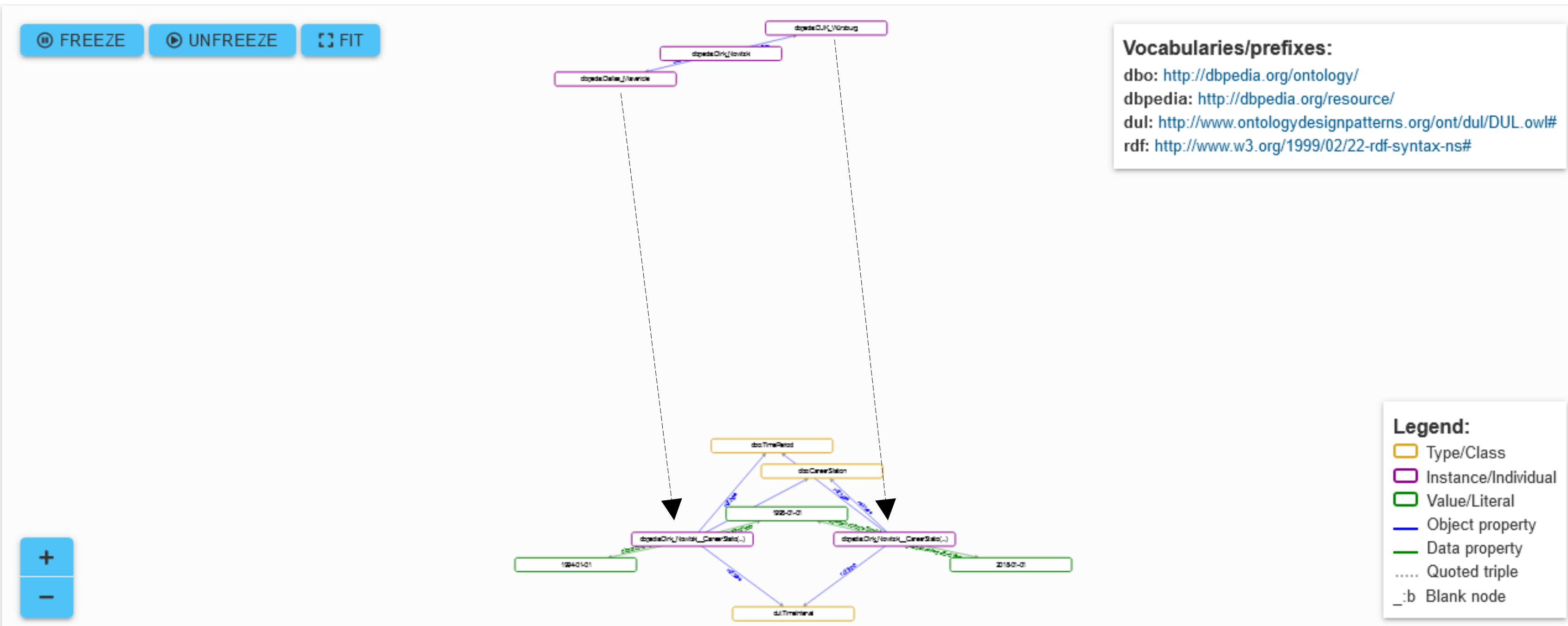
```
dbr:Dirk_Nowitzki__CareerStation_1 {
        dbr:Dirk_Nowitzki dbo:team dbr:DJK_Würzburg .
}

dbr:Dirk_Nowitzki__CareerStation_2 {
        dbr:Dirk_Nowitzki dbo:team dbr:Dallas_Mavericks     .
}

dbr:Dirk_Nowitzki {
        dbr:Dirk_Nowitzki__CareerStation_1
                dbo:activeYearsEndYear    "1998"^^xsd:gYear ;
                dbo:activeYearsStartYear  "1994"^^xsd:gYear .
        dbr:Dirk_Nowitzki__CareerStation__2
                dbo:activeYearsEndYear    "2018"^^xsd:gYear ;
                dbo:activeYearsStartYear  "1998"^^xsd:gYear ;
}
```

Verbosity of RDF Graphs

- Alternative: Named Graphs



Visualization: https://issemantic.net/rdf-visualizer

Verbosity of RDF Graphs

- Intermediate summary:
  - RDF seems particularly bad at representing non-triple information
  - Choice:
    - Blow up RDF graph (like DBpedia)
    - Use non-straightforward representation
      - Reification
      - Named Graphs
    - Other approaches in academia (singleton property, NDFluents, …)
      - Not very handy either
      - Little adoption
    - In any case:
      - Querying gets harder

Verbosity of RDF Graphs

- Motivation for labeled property graphs
- Modeling would be much easier
  - If we could simply attach information to edges

- Attempt in the Semantic Web Technologies Toolstack:
  - RDF* / SPARQL*

Hello RDF*

- RDF:
  - Subjects are URIs or blank nodes
  - Predicates are URIs
  - Objects are URIs, blank nodes, or literals

- RDF*:
  - Subjects are URIs, blank nodes, or quoted statements
  - Predicates are URIs
  - Objects are URIs, blank nodes, literals, or quoted statements

Hello RDF*

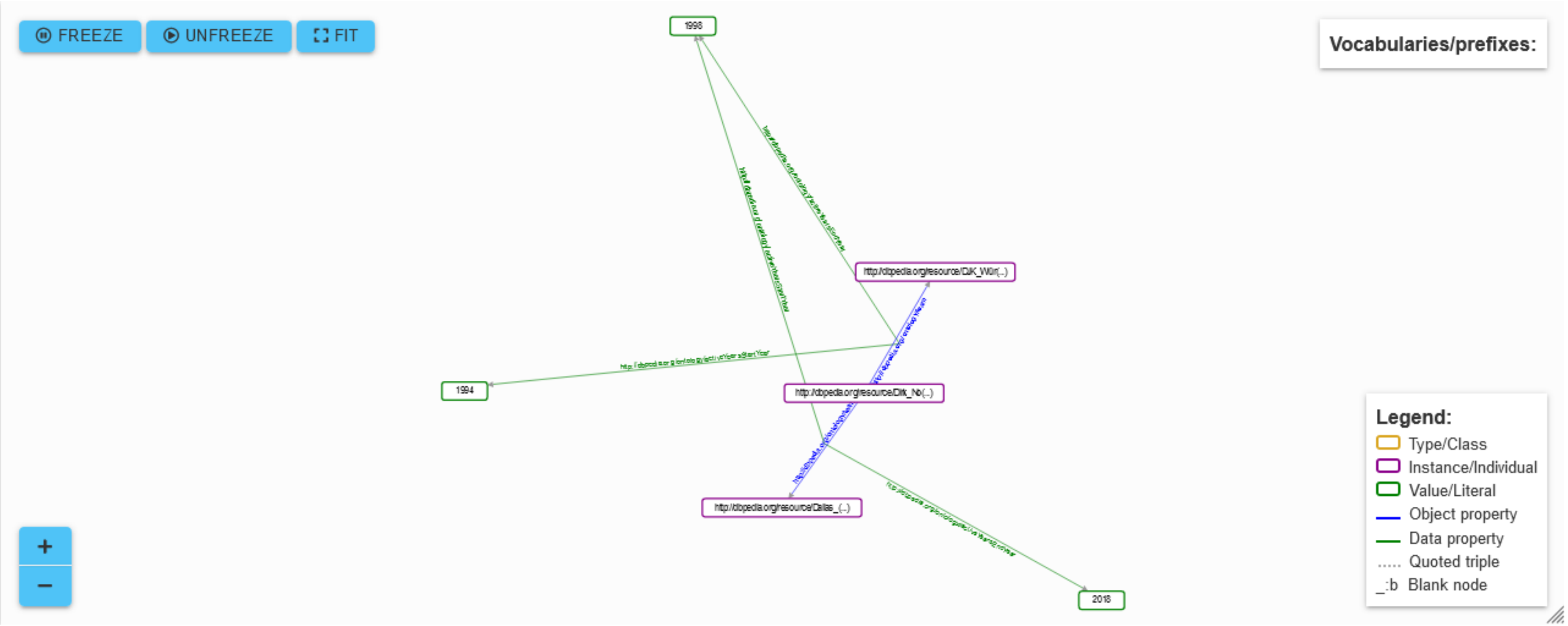
- Quoting triples

```
<<dbr:Dirk_Nowitzki dbo:team dbr:DJK_Wuerzburg>>
    dbo:activeYearsStartYear 1994 ;
    dbo:activeYearsEndYear 1998 .
```

- In this example, the subject of the statement is a triple.

The CareerStation Example in RDF*

- Annotations are added to edges



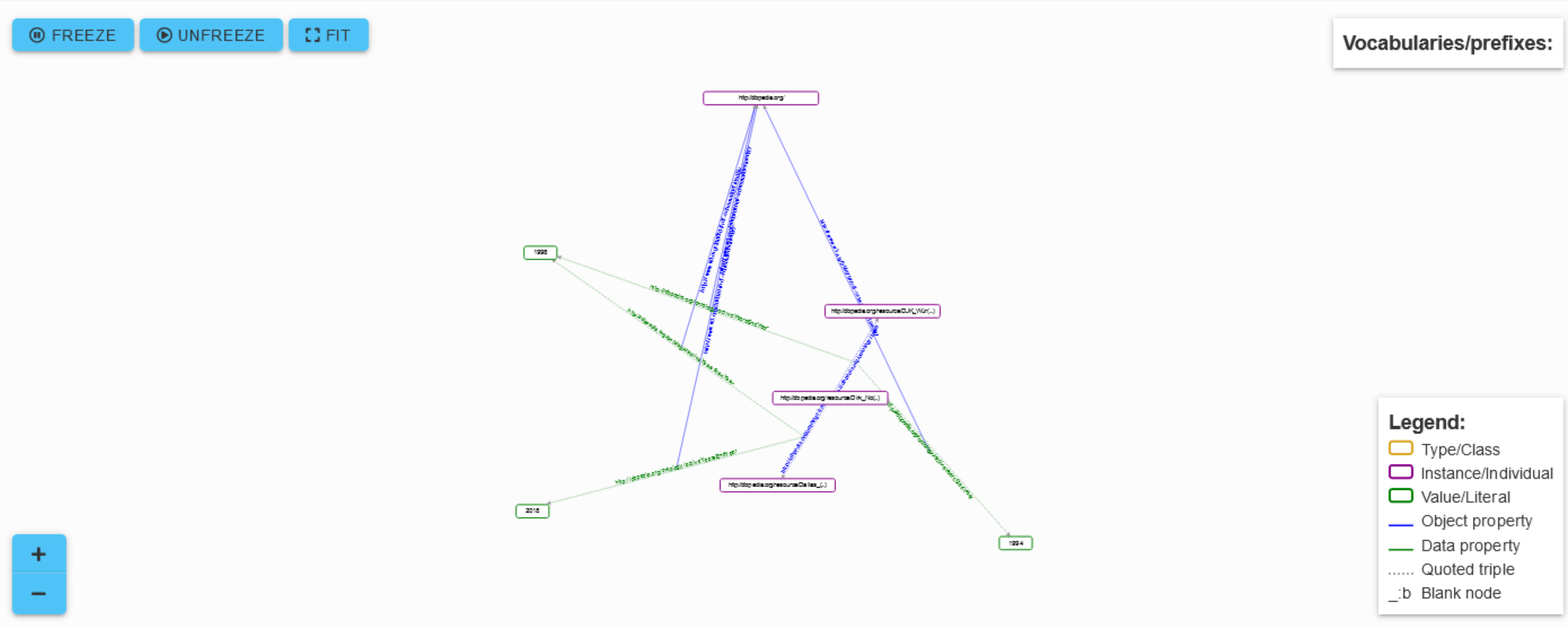
Nesting in RDF*

- RDF* statements can be subjects and objects themselves

```
<<
<<dbr:Dirk_Nowitzki dbo:team dbr:DJK_Wuerzburg>>
    dbo:activeYearsStartYear 1994 ;
    dbo:activeYearsEndYear 1998 .
>>
rdfs:definedBy
<http://dbpedia.org/>
```

Nesting in RDF*

- Visualized:



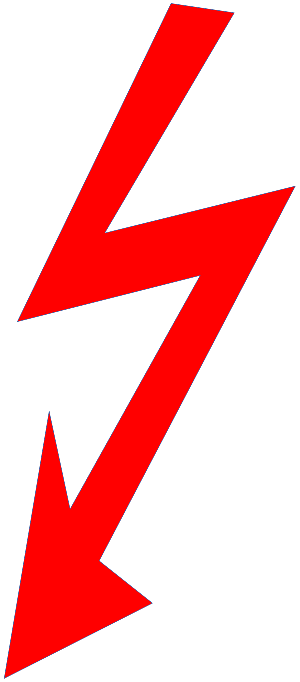
Interpretation of RDF* Graphs

- Or: is RDF* just syntactic sugar
  for representing reification more nicely?

Interpretation of RDF* vs. RDF

- RDF example

```
:s1 a rdf:Statement ;
    rdf:subject :Hamburg ;
    rdf:predicate rdf:type ;
    rdf:object :City .
:s2 a rdf:Statement ;
    rdf:subject :Hamburg ;
    rdf:predicate rdf:type ;
    rdf:object :Country .
:Peter :says :s1 .
:Mary :says :s2 .
```



```
:City owl:disjointWith :Country .
```

Interpretation of RDF* vs. RDF

- Observation
  - In RDF, we cannot make statements
    about two contradictory statements A and B
  - ...without the entire graph being contradictory

- This is not in line with “everyday semantics”. Compare
  - *Hamburg is a city and a country,*
    and nothing is a city and a country at the same time.
- to
  - *Peter says Hamburg is a city, Mary says Hamburg is a country,*
    and nothing is a city and a country at the same time.

Interpretation of RDF* vs. RDF

- Observation:
  - In RDF, when we make a statement about a statement S, S is automatically assumed to be true.

- In RDF*, this is not the case:

```
:Peter :says <<:Hamburg rdf:type :City >> .
:Mary :says <<:Hamburg rdf:type :Country >> .

:City owl:disjointWith :Country .
```

RDF*: Quoted vs. Asserted Triples

- Quoted triples are not automatically true
- If we want to make them true (asserted), we have to do so explicitly:

```
dbr:Dirk_Nowitzki dbo:team dbr:DJK_Wuerzburg .
<<dbr:Dirk_Nowitzki dbo:team dbr:DJK_Wuerzburg>>
    dbo:activeYearsStartYear 1994 ;
    dbo:activeYearsEndYear 1998 .
```

- For this, there is a syntactic shortcut:

```
dbr:Dirk_Nowitzki dbo:team dbr:DJK_Wuerzburg
{|  dbo:activeYearsStartYear 1994 ;
    dbo:activeYearsEndYear 1998 |} .
```

SPARQL*: Querying RDF* Graphs

- SPARQL*:
  - Just like ordinary SPARQL
  - Triple patterns can contain
    - Quoted triples
    - Triple annotations
  - Plus a few more builtin functions

- SPARQL* Results:
  - A few devils in the details

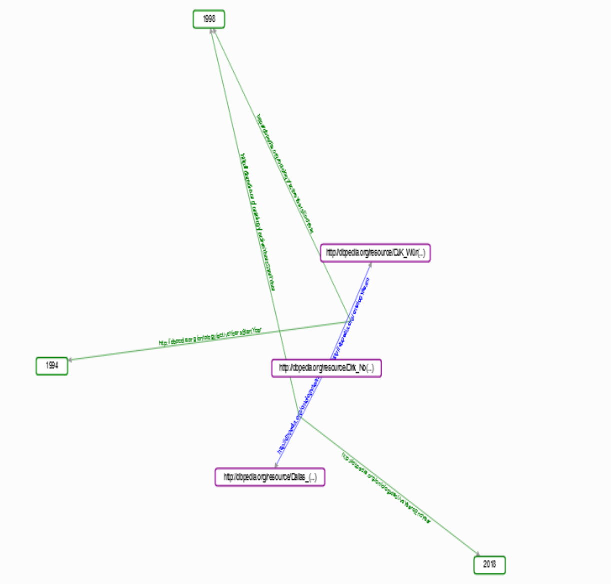
Hello SPARQL*

- When did Dirk Nowitzki play for DJK Würzburg?

```
SELECT ?startyear ?endyear WHERE {
    dbr:Dirk_Nowitzki dbo:team :dbr:DJK_Würzburg
    {| dbo:activeYearsStartYear ?startyear ;
       dbo:activeYearsEndYear ?endyear |} }
```

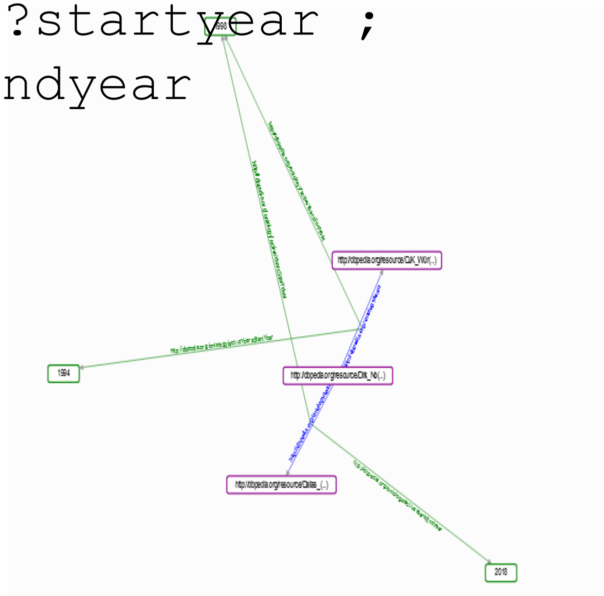
- Returns

```
{(?startyear=1994; ?endyear=1998)}
```



Hello SPARQL*

- When did Dirk Nowitzki play for DJK Würzburg?
- ```
  SELECT ?startyear ?endyear WHERE {
      dbr:Dirk_Nowitzki dbo:team :dbr:DJK_Würzburg .
      <<dbr:Dirk_Nowitzki
                  dbo:team :dbr:DJK_Würzburg>>
            dbo:activeYearsStartYear ?startyear ;
            dbo:activeYearsEndYear ?endyear
  }
  ```
- Returns
  `{(?startyear=1994; ?endyear=1998)}`
- Note: these are the same short/longhand notations as for RDF*


  ```

SPARQL* Return Types

- Consider the following RDF* graph:

```
:Julia :loves :Peter .
:Jane :knows :Julia .
:Jane :knows <<:Julia :loves :Peter>> .
```

- We can query with SPARQL*

```
SELECT ?x WHERE {:Jane :knows ?x}
```

- Results:

```
{(?x = :Julia), (?x = <<:Julia :loves :Peter>>)}
```

SPARQL* Return Types

- SPARQL return types:
  - Resource with URI `isURI`
  - Blank node `isBLANK`
  - Literal `isLITERAL`
  - Number `isNUMERIC`
- SPARQL* adds a fifth return type:
  - Triple `isTRIPLE`

SPARQL* Return Types

- Consider the following RDF* graph:

```
:Julia :loves :Peter .
:Jane :knows :Julia .
:Jane :knows <<:Julia :loves :Peter>> .
```

- We can query with SPARQL*

```
SELECT ?x WHERE {:Jane :knows ?x .
                     FILTER(isTRIPLE(?x)}
```

- Results:

```
{(?x= <<:Julia :loves :Peter>>)}
```

Other Query Types with SPARQL*

- ASK and DESCRIBE: work as in SPARQL
- CONSTRUCT: can also construct RDF*

```
CONSTRUCT {<<?x ?y ?z>> :definedIn :myDataSet}
WHERE {?x ?y ?z}
```

- Result on this example:

```
<<:Julia :loves :Peter >> :definedIn :myDataSet .
<<:Jane :knows :Julia >> :definedIn :myDataSet .
<<:Jane :knows <<:Julia :loves :Peter>> >>
         :definedIn :myDataSet .
```

Mind the Assertion Gap

- Remember: not all *quoted* triples are *asserted*
- The default graph of SPARQL results is only *asserted* triples

- Consider the following RDF* graph:

```
:Julia :loves :Peter .
:Jane :knows :Julia .
:Jane :knows <<:Julia :loves :Peter>> .
:Julia :thinks <<:Jane :loves :Peter>> .
```

- Query:

```
SELECT ?x WHERE {?x :loves :Peter}
```

- Result:

```
{(?x = :Julia)}
```

Mind the Assertion Gap

- Remember: not all *quoted* triples are *asserted*
- The default graph of SPARQL results is only *asserted* triples

- Consider the following RDF* graph:

```
:Julia :loves :Peter .
:Jane :knows :Julia .
:Jane :knows <<:Julia :loves :Peter>> .
:Julia :thinks <<:Jane :loves :Peter>> .
```

- On the other hand:

```
SELECT ?x WHERE {:Julia :thinks ?x}
```

- Result:

```
{(?x = <<:Jane :loves :Peter>>)}
```

RDF*/SPARQL*: Not (yet) a standard, but...

- Lots of tools support RDF* and/or SPARQL*:

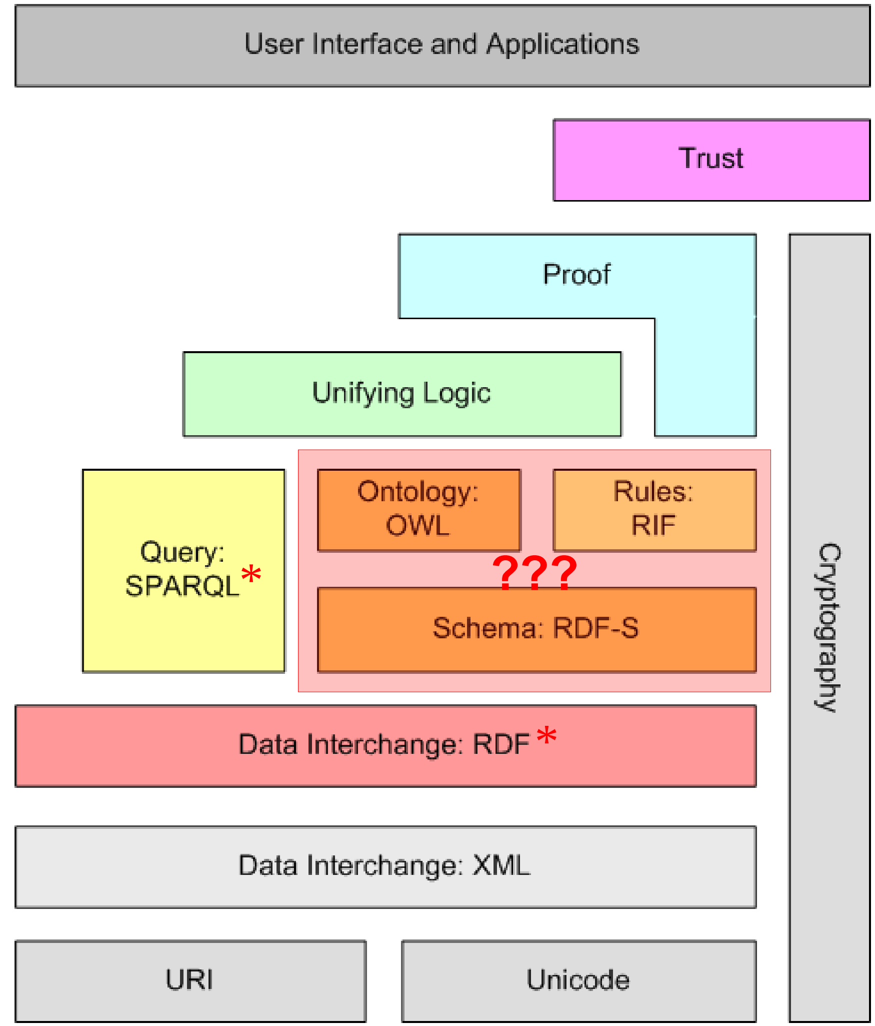
Implementation	Source	Notes
AllegroGraph	mailing list	PG mode, in the works
AnzoGraph	documentation	PG mode
BlazeGraph	documentation	PG mode
Corese	documentation	PG mode
EYE	implementation report	
GraphDB	documentation	
Apache Jena	implementation report, documentation	
Eclipse rdf4j	documentation	
Morph-KGC	github, documentation	RML-star
Oxigraph	implementation reports: Rio Turtle, SPARQL	
RDF.ex	implementation report, documentation	
rdfjs/N3.js	github	
RubyRDF	implementation reports: RDF::TriG, SPARQL	
Stardog	documentation	PG mode
TopBraid EDG	blog post	PG mode with custom annotation syntax

Semantic Web Technology Stack (revisited)

here be dragons...

Knowledge Graph
Technologies
(This lecture)

Technical
Foundations



Berners-Lee (2009): *Semantic Web and Linked Data*
http://www.w3.org/2009/Talks/0120-campus-party-tbl/

RDF* and Inference

- Consider the following RDF* graph and RDFS schema:

```
<<:Berlin :capitalOf :Germany>>
 {| :statedBy :Wikipedia |}
```

```
:capitalOf rdfs:subpropertyOf :locatedIn
```

- Would you consider the following inference legit?

```
<<:Berlin :locatedIn :Germany>>
 {| :statedBy :Wikipedia |}
```

RDF* and Inference

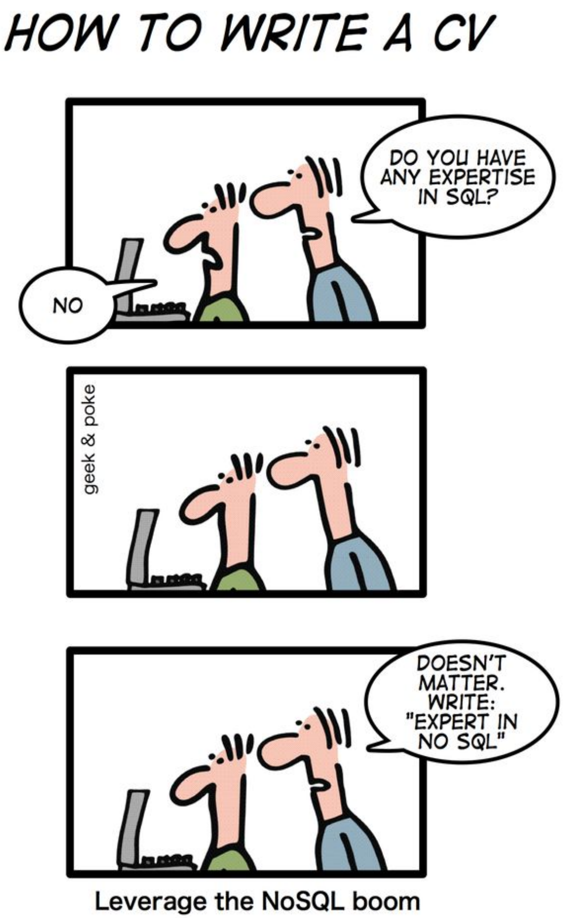
- OK, so what about

```
<<:Bonn :capitalOf :Germany>>
{| :from "1949" ; :until "1990" |}
:capitalOf rdfs:subpropertyOf :locatedIn
```

- RDF* and inference is still an open research topic

Labeled Property Graphs in the Industry

- For a while, RDF had little adoption in the industry
  - Perceived as too verbose and cumbersome
    - We saw that earlier today, too
  - Underlying semantic properties impractical in many cases
- Meanwhile, NoSQL gained a lot of traction
  - i.e., property/value stores
- Labeled Property graphs
  - A combination of property/value stores and graphs

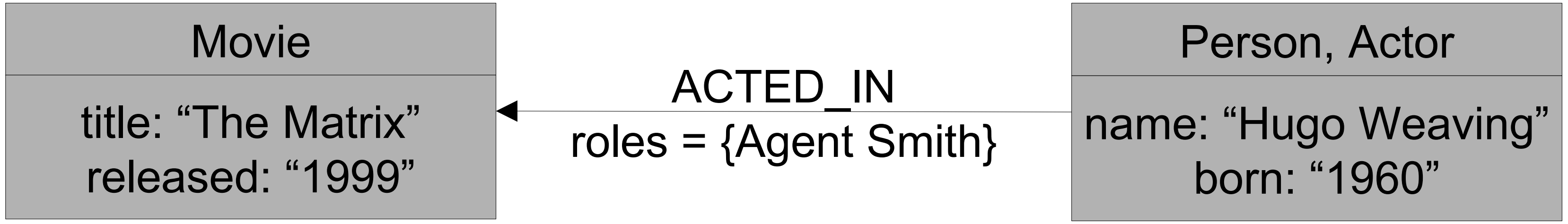


A Brief History of Cypher

- Started as a proprietary query language for the graph database system neo4j in 2011
- Since 2015: Open Cypher
  - Most recent version: Cypher v9, 2018
- Wider adoption, e.g.,
  - Amazon Neptune
  - SAP HANA Graph
  - ...and many others

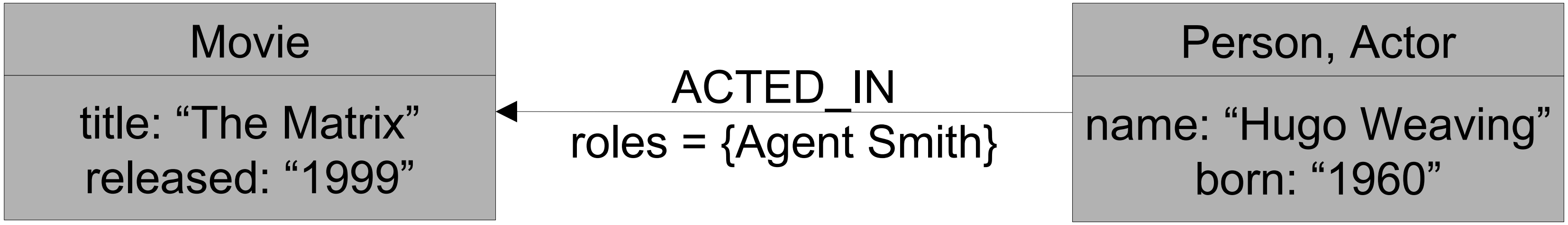
Labeled Property Graphs – Definition

- A graph consists of
  - Entities (with one or more labels)
  - Property keys
  - Property values
  - Relations (with exactly one type)
- Entities and relations can have property key/value pairs



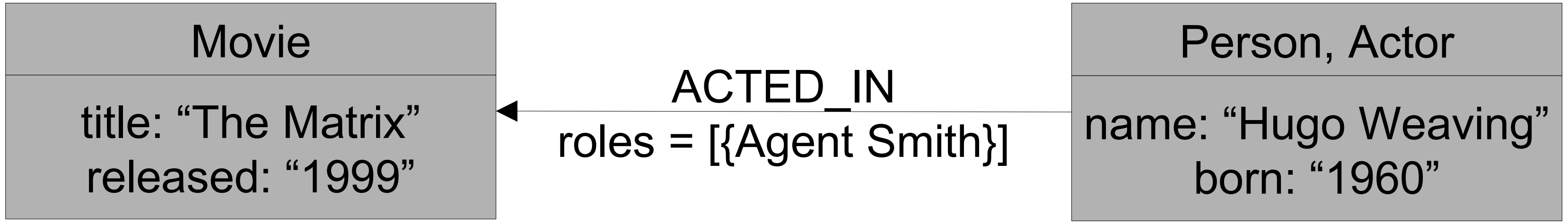
Basics of Cypher

- Like SPARQL, Cypher is based on pattern matching
  - `()` denotes a node
  - `[]` denotes a relation
  - `()-[]->()` denotes a directed path
  - `()-[]-()` denotes an undirected path



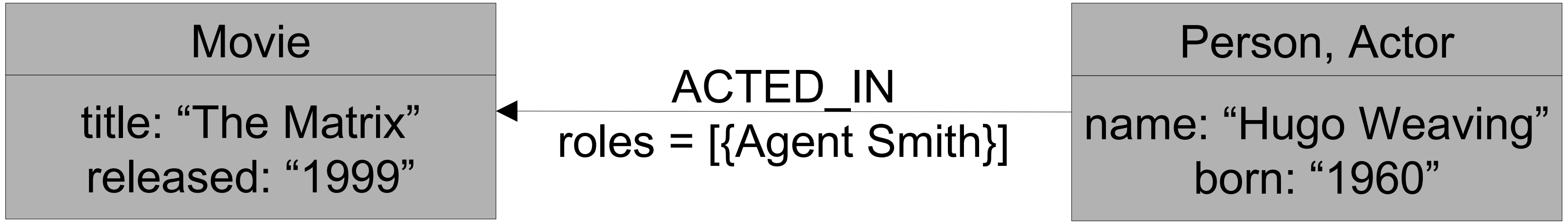
Hello Cypher!

- Simple query: matching any node
  - `MATCH (n) return n`
- Would return all nodes



Hello Cypher!

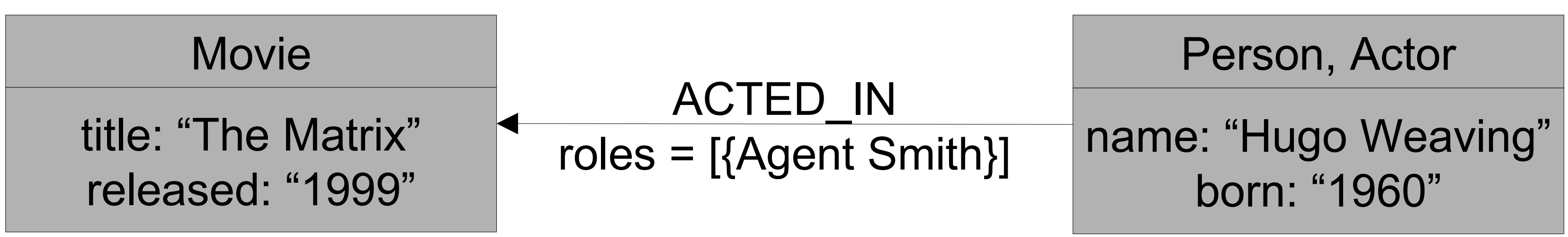
- Simple query: matching nodes with labels
  - `MATCH (n:Movie) return n`
- Would return only movie nodes



Restrictions on Keys

- Simple query: matching any node
  - `MATCH (n:Movie {title: “The Matrix”}) return n`
- Would return only the specific movie

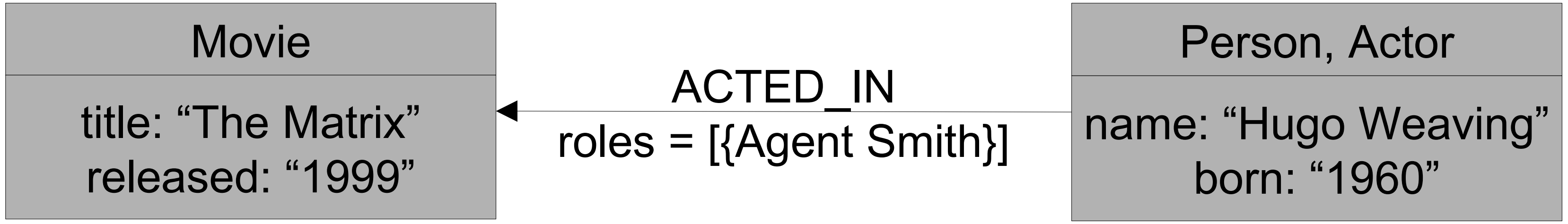
- Also possible:
  - `MATCH (n {title: “The Matrix”) return n`
- Would return any node with a title “The Matrix”



Querying for Node Types

- What kind of node is “The Matrix”?

```
match(m {title:"The Matrix"}) return labels(m)
```

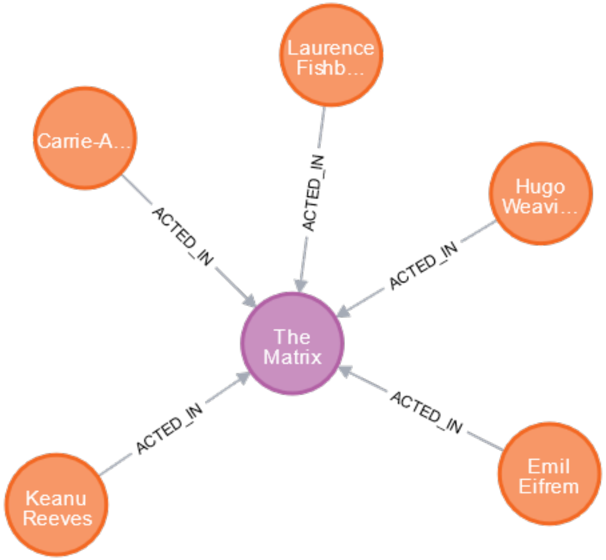


Path Expressions

- Using paths in patterns
  - `MATCH (m:Movie {title: "The Matrix")-[r]-(e) return m,r,e`
- All ingoing and outgoing edges

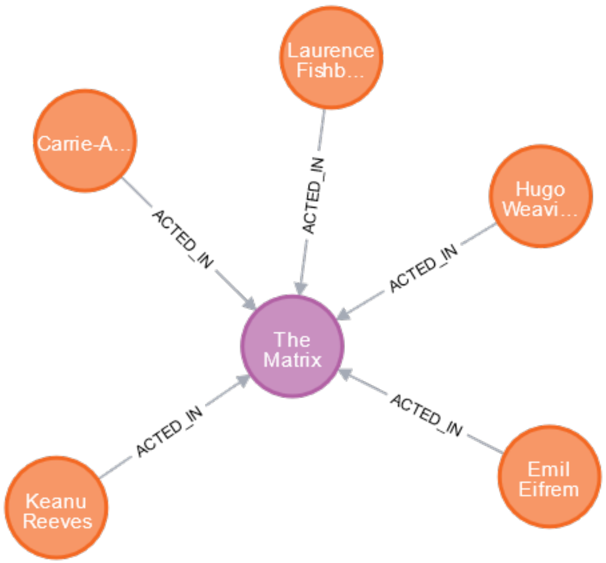
Path Expressions

- Combining restrictions on labels
  - `MATCH (m:Movie {title: “The Matrix”)-[r:ACTED_IN]-(e) return m,r,e`
- All ingoing and outgoing edges with a particular label



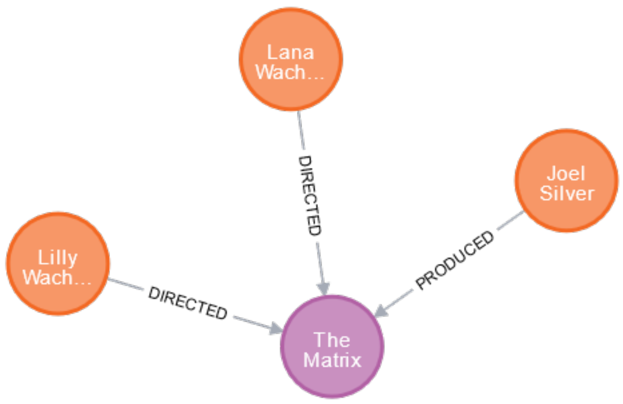
Path Expressions

- Combining restrictions on labels
  - `MATCH (m:Movie {title: "The Matrix")-[r:ACTED_IN]-(e) return m,r,e`
- All ingoing and outgoing edges with a particular label



Path Expressions

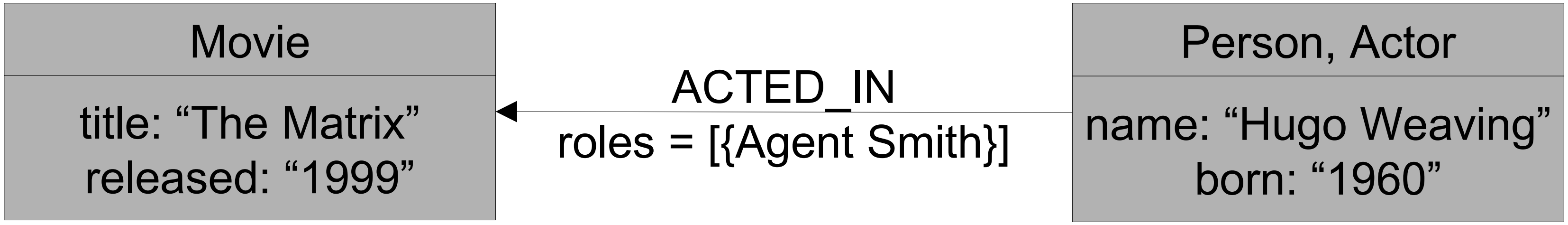
- Combining restrictions on labels
  - ```
    MATCH (m:Movie {title:"The Matrix"})
      <-[r:PRODUCED|DIRECTED]-(e)
    return m,r,e
    ```
- All ingoing and outgoing edges with a particular label


    ```

Querying for Relation Types

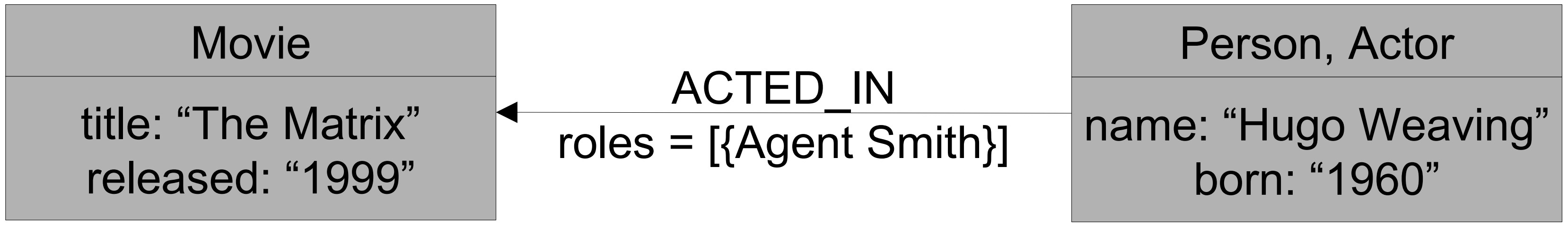
- What kind of relation does Hugo Weaving have to the Matrix?

```
Match
 (Movie {title:"The Matrix"})
   <-[r]-(Person {name:"Hugo Weaving"})
return type(r)
```



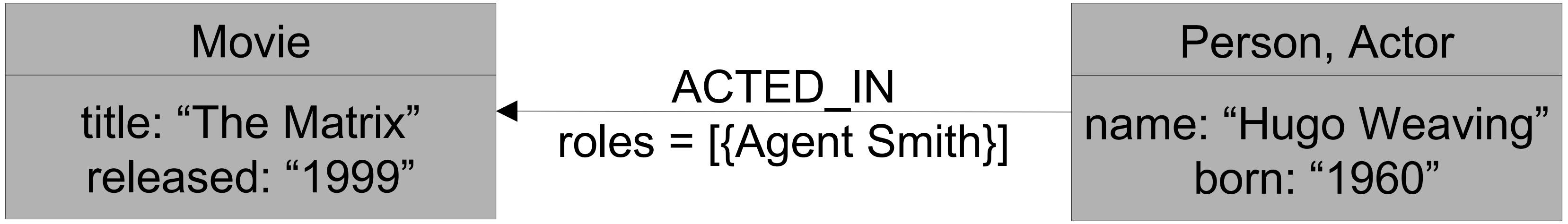
Path Expressions

- Combining restrictions on properties
- Who played Agent Smith in The Matrix?
  - `match({title: "The Matrix"})`
    `<-[ACTED_IN {roles:["Agent Smith"]}]-(e) return e`
- All ingoing and outgoing edges with a particular label



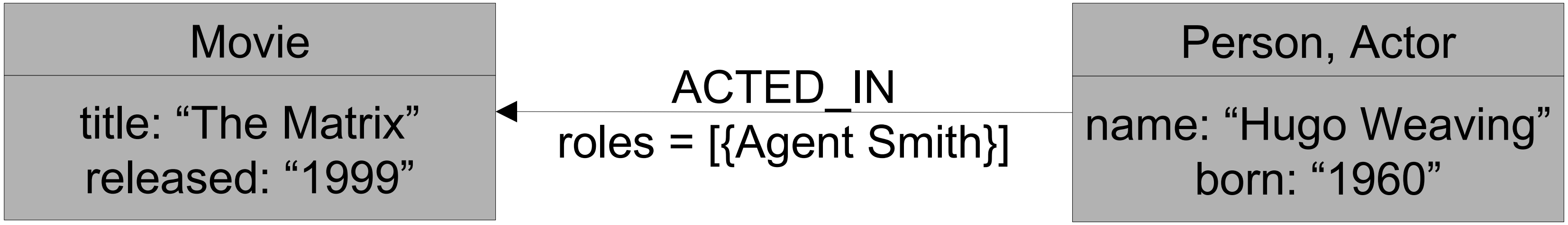
Return Types in Cypher

- So far, our return types were nodes or relations
- We can also query for specific properties:
  - `match(m:Movie {title: "The Matrix"}) return m.released`



Querying for Property Values

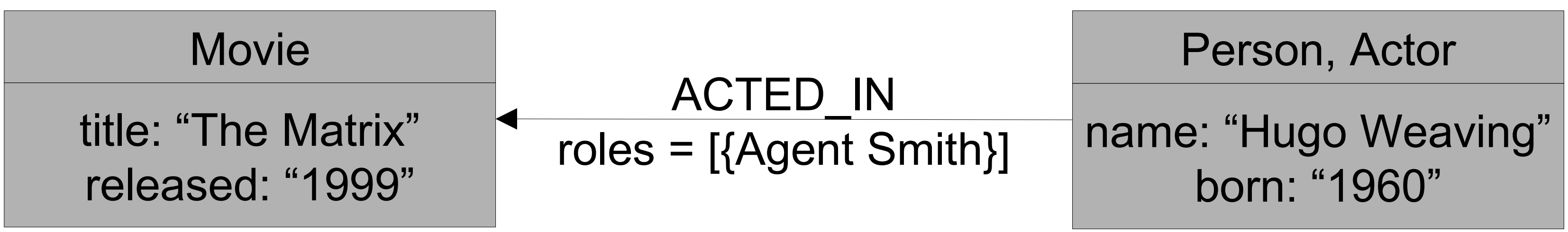
- The return value can also be a property of a relation:
- Which role(s) did Hugo Weaving play in The Matrix?
  - ```
    match(Movie {title: "The Matrix"})
      <-[r:ACTED_IN]-(Person {name:"Hugo Weaving"})
    return r.roles
    ```


    ```

Complex Paths

- So far, we have only considered one hop paths
- Which movies did both Hugo Weaving and Keanu Reeves act in?
  - ```
    match
    (p1:Person {name:"Hugo Weaving"})-[r1:ACTED_IN]->
      (m:Movie)
    <-[r2:ACTED_IN]-(p2:Person {name:"Keanu Reeves"})
    return m
    ```

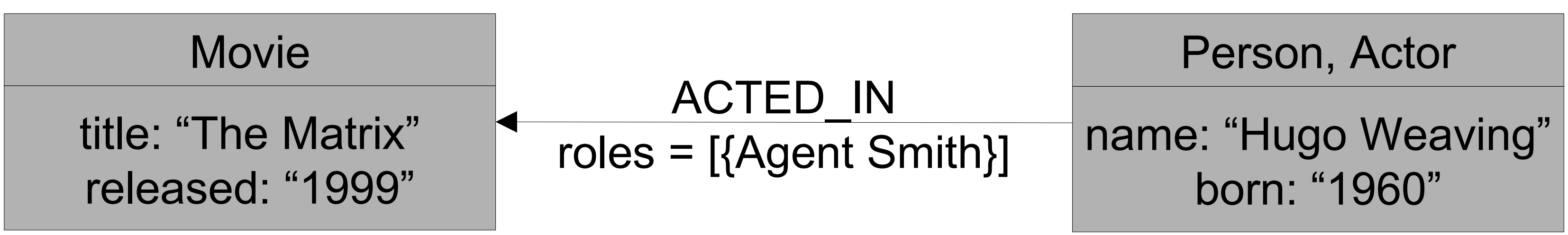
OK, but how about three actors?


    ```

Combining Match Clauses

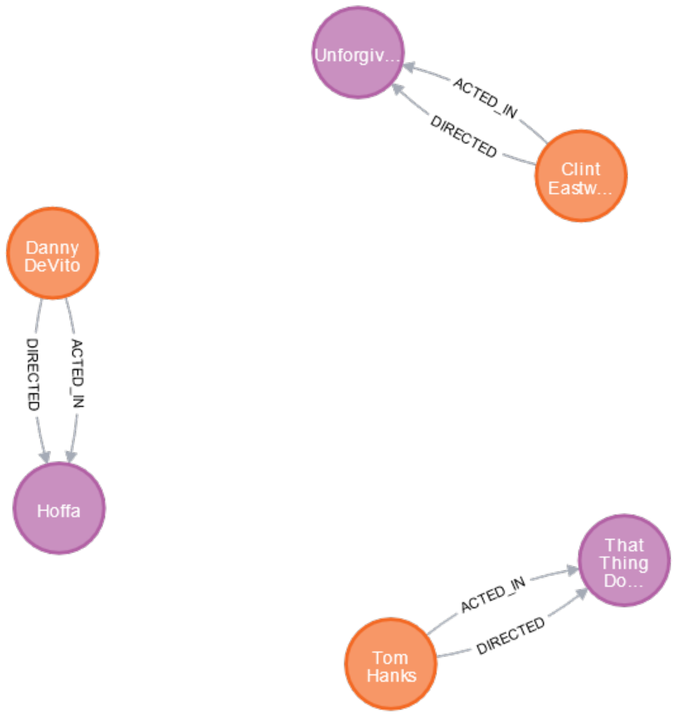
- We can have multiple match clauses
  - By default, they are conjunctive
- Which movies did Hugo Weaving, Keanu Reeves, and Carrie-Anne Moss act in?
  - ```
    match (p1:Person {name:"Hugo Weaving"})
      -[r1:ACTED_IN]->(m:Movie)
    match (p2:Person {name:"Keanu Reeves"})
      -[r2:ACTED_IN]->(m:Movie)
    match (p3:Person {name:"Carrie-Anne Moss"})
      -[r3:ACTED_IN]->(m:Movie)
    return m
    ```

Common variable in the clauses


    ```

Combining Match Clauses

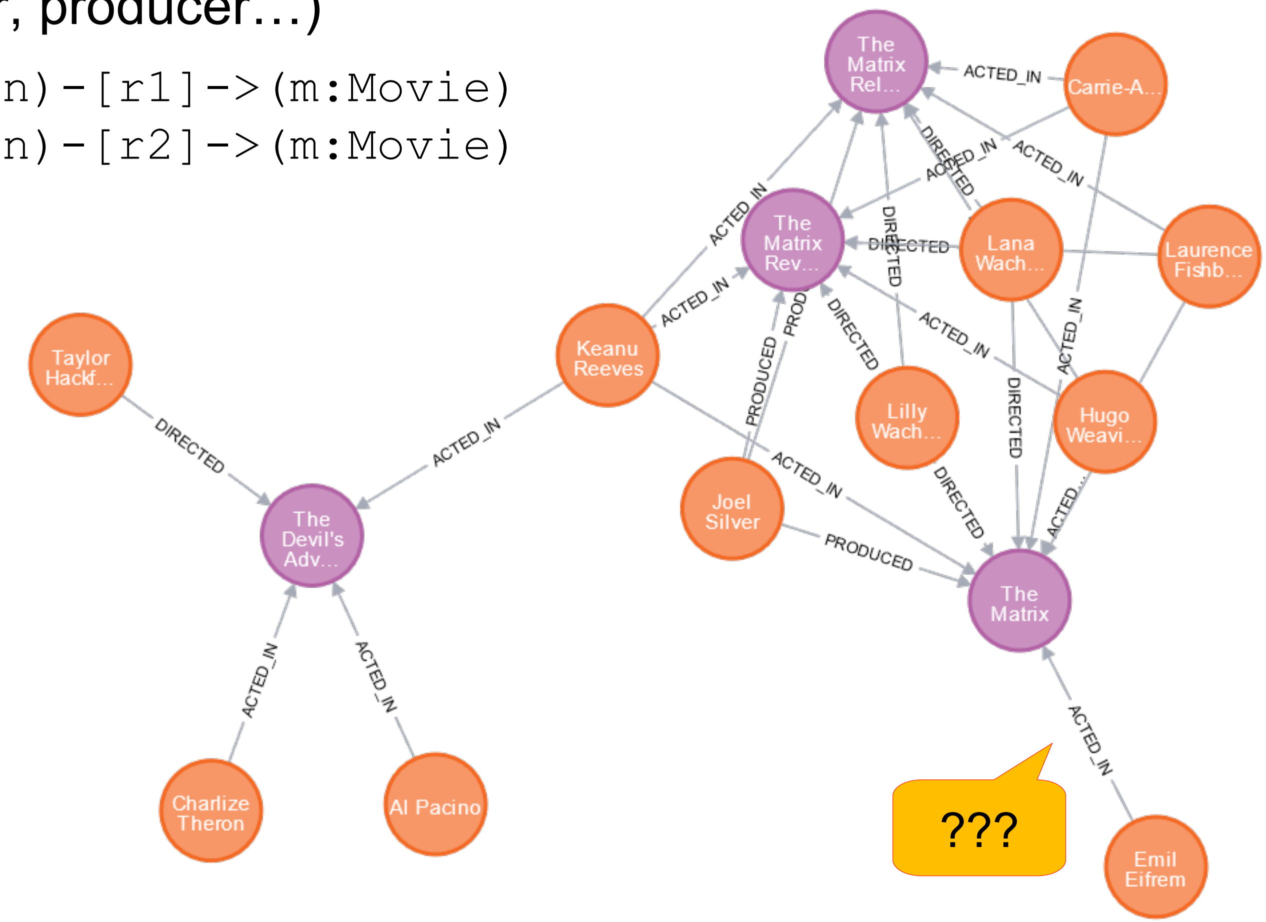
- There can also be more than one common variable
- Which movies where directed by people who also acted in them?
  - ```
    match(p:Person)-[r1:ACTED_IN]->(m:Movie)
    match(p:Person)-[r2:DIRECTED]->(m:Movie)
    return p,m
    ```


    ```

Variable Binding

- Let's try to find people who have at least two relations to a movie (e.g., director, actor, producer…)
  - ```
    match(p:Person)-[r1]->(m:Movie)
    match(p:Person)-[r2]->(m:Movie)
    return p,m
    LIMIT 25
    ```

We haven't seen LIMIT for Cypher yet, but it's straight forward


    ```

Variable Binding

- Let us investigate this more closely
  - ```
    match(p:Person)-[r1]->(m:Movie)
    match(p:Person)-[r2]->(m:Movie)
    return p,m,r1,r2
    LIMIT 25
    ```

r1 and r2 have the same binding!

p


```
{
  "identity": 8,
  "labels": [
    "Person"
  ],
  "properties": {
"born": 1978,
"name": "Emil Eifrem"
  }
}
```


m


```
{
  "identity": 0,
  "labels": [
    "Movie"
  ],
  "properties": {
"tagline": "Welcome to the Real World",
"title": "The Matrix",
"released": 1999
  }
}
```


r1

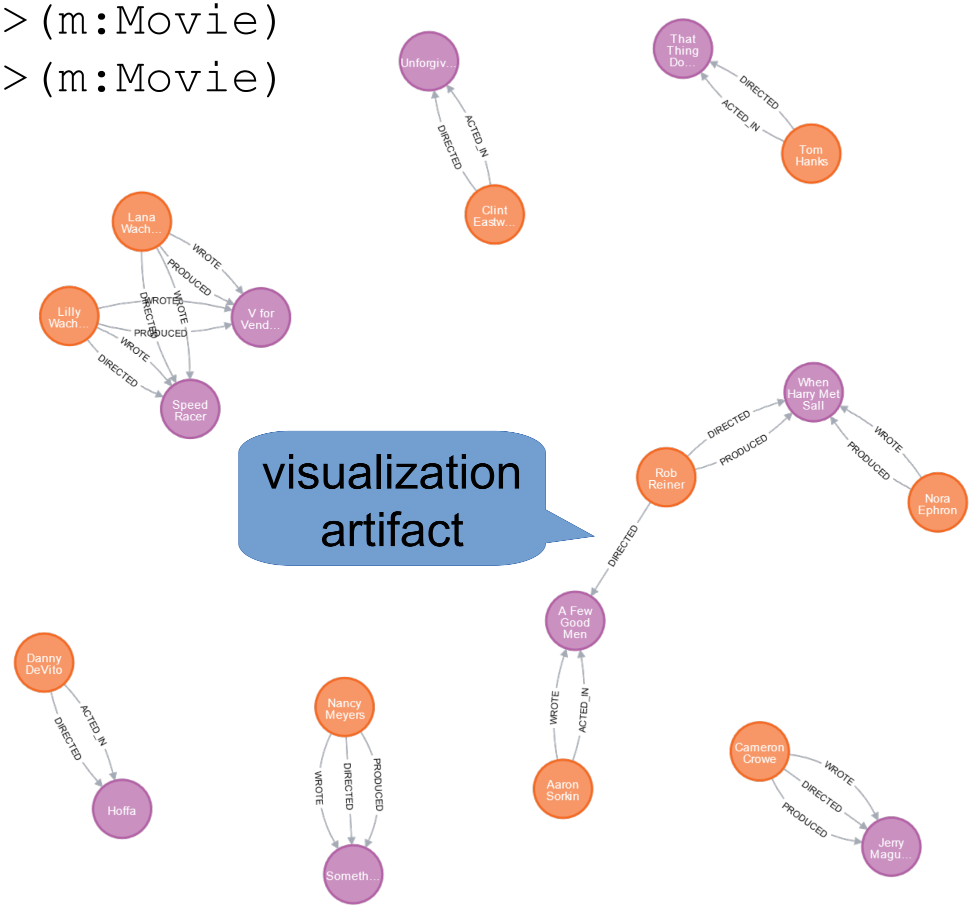

```
{
  "identity": 7,
  "start": 8,
  "end": 0,
  "type":
"ACTED_IN",
  "properties": {
"roles": [
      "Emil"
    ]
  }
}
```


r2


```
{
  "identity": 7,
  "start": 8,
  "end": 0,
  "type":
"ACTED_IN",
  "properties": {
"roles": [
      "Emil"
    ]
  }
}
```
```

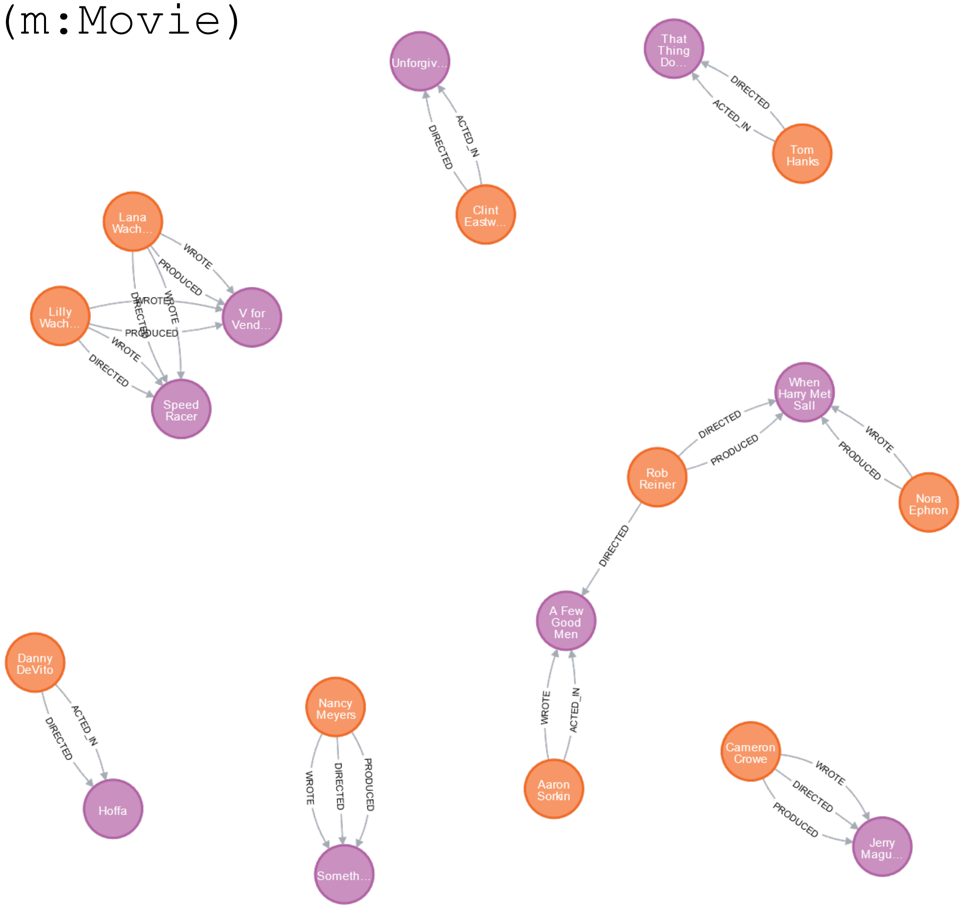
WHERE Clauses

- Used to impose additional restrictions (like in SQL, SPARQL, …)
  - ```
    match(p:Person)-[r1]->(m:Movie)
    match(p:Person)-[r2]->(m:Movie)
    where(r1<>r2)
    return p,m
    ```


    ```

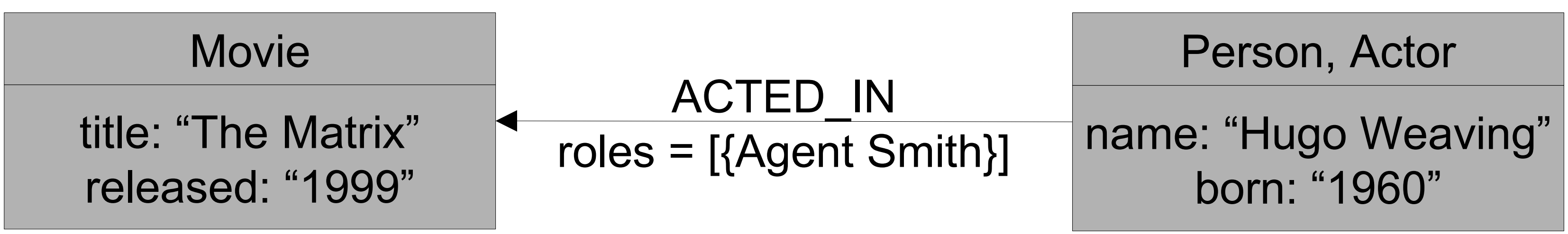
WHERE Clauses

- Used to impose additional restrictions (like in SQL, SPARQL, …)
  - ```
    match(p:Person)-[r]->(m:Movie)
    with p,m,count(r)as c
    where c>=2
    return p,m,r
    ```


    ```

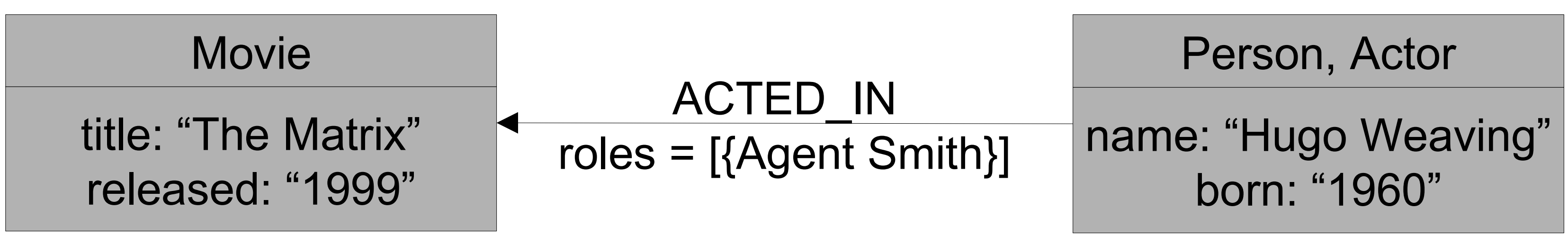
WHERE Clauses

- Numeric comparisons
- All movies starring Hugo Weaving released in the 1990s
  - ```
    Match
    (m:Movie)←[ACTED_IN]-
      (p:Person {name:"Hugo Weaving"})
    where m.released>1990 and m.released<2000
    return m
    ```


    ```

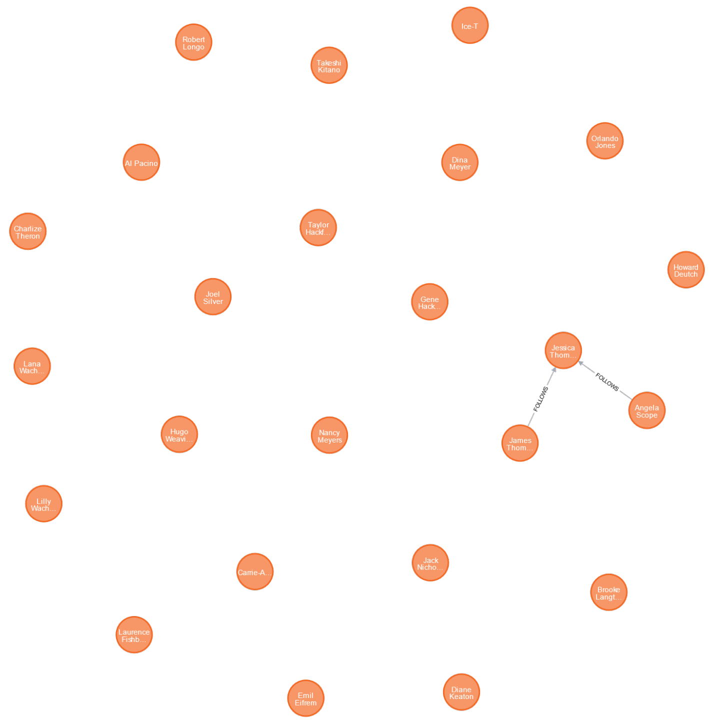
WHERE Clauses

- String comparisons
- All actors whose first name is “Hugo”
  (approximate solution: name starts with “Hugo”)
  - ```
    match(Movie)<-[ACTED_IN]-(p:Person)
    where (p.name STARTS WITH ("Hugo"))
    return p
    ```


    ```

Path Quantifiers

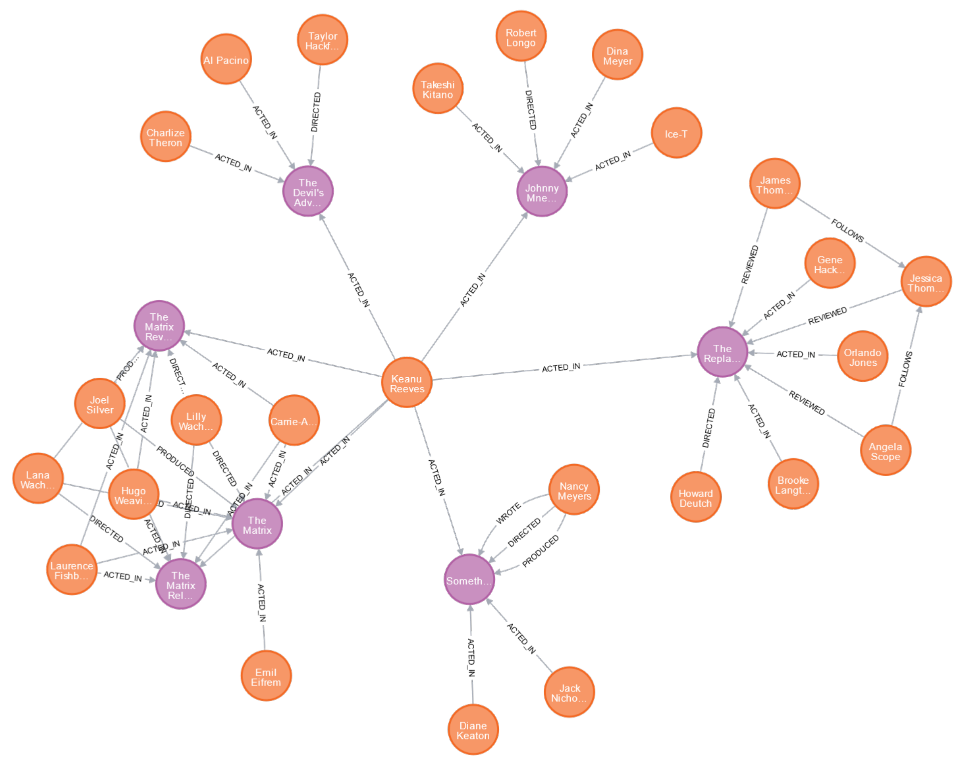
- Find all people connected via two ACTED_IN relations to Keanu Reeves (i.e., all people who co-starred with Keanu Reeves)
  - ```
    match
    (p1:Person {name: "Keanu Reeves"})
     -[ACTED_IN*2]-(p2:Person)
    return p2
    ```


    ```

Path Quantifiers

- Extract find all one and two hop neighbors of Keanu Reeves (no particular edge type)

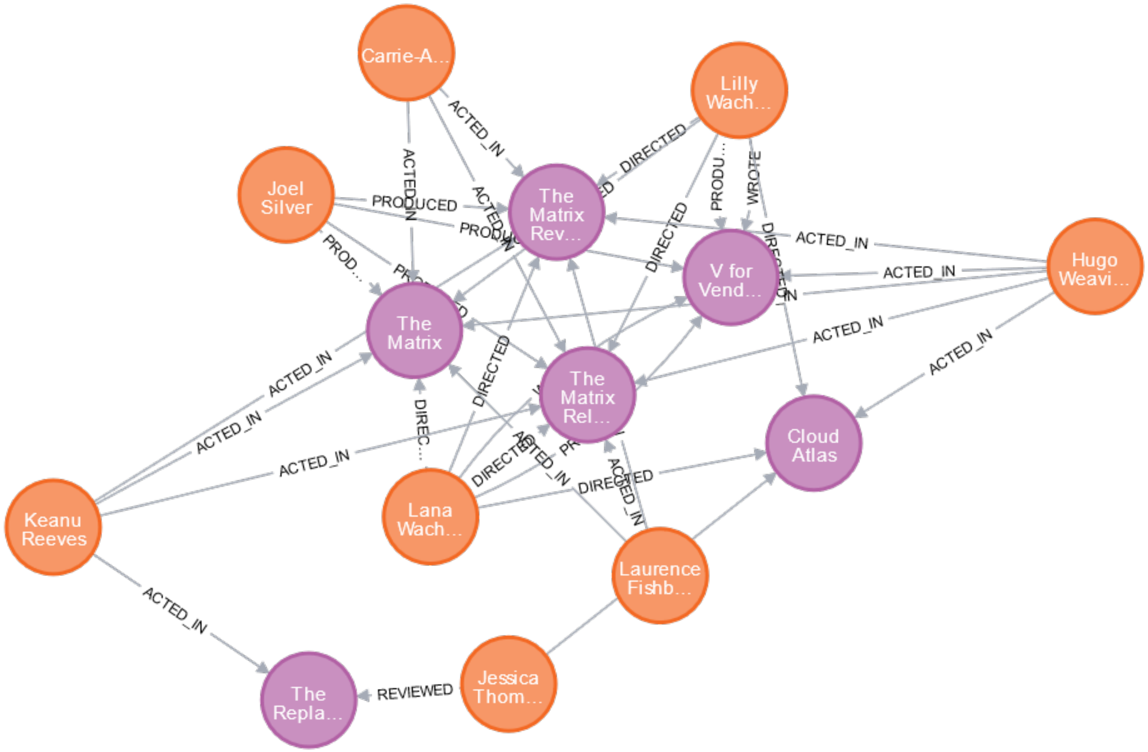
```
match
(p:Person {name: "Keanu Reeves"})-[*1..2]-(e)
return p,e
```



Pathfinding with Quantifiers

- Find all paths of length up to 4 between Keanu Reeves and Hugo Weaving

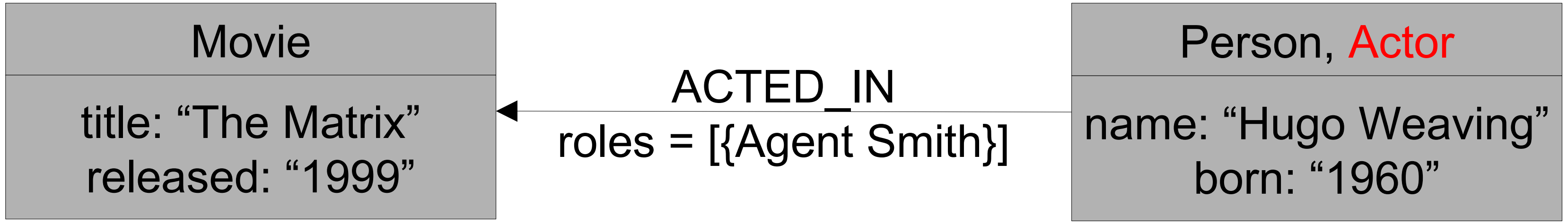
```
match p = (p1:Person {name: "Keanu Reeves"})
  -[*1..4]-(p2:Person {name: "Hugo Weaving"})
return p
```



Graph Updates

- Cypher also allows for adding and deleting information
- This requires a set instead of a return statement, e.g.,

```
match (p:Person)-[ACTED_IN]->(m:Movie)
set p:Actor
```



Graph Updates

- Cypher also allows for adding and deleting properties
- This requires a set instead of a return statement, e.g.,

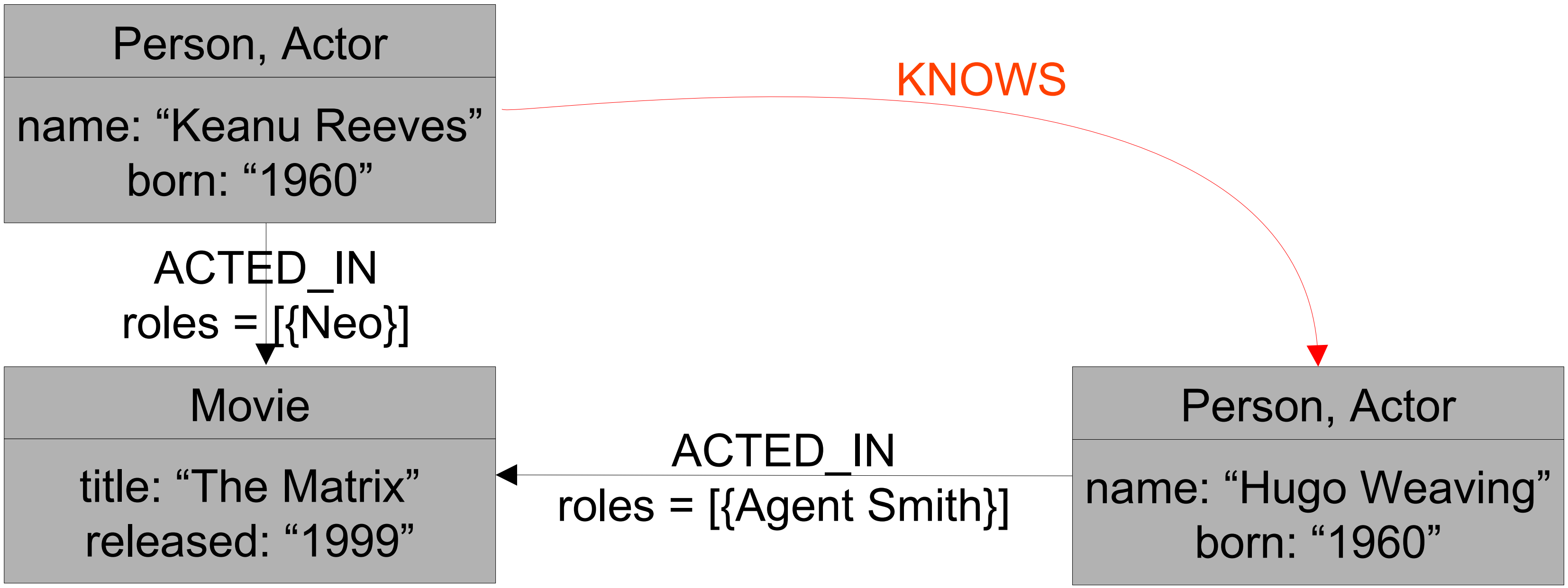
```
match(p:Person)-[ACTED_IN]->(m:Movie)
with p,count(m) as moviecount
where (moviecount>10)
set p.famous="true"
```

- Notes on this query:
  - Cipher allows counting (closed world semantics)
  - The `with` construct is used for variable scoping
    - Compute `with` first
    - Compute `where` second
      - cf. `having` in SQL

Graph Updates

- Cypher also allows for adding and deleting nodes and edges
- This requires a create instead of a return statement, e.g.,

```
match (p1:Person)-[r1:ACTED_IN]->(m:Movie)
match (p2:Person)-[r2:ACTED_IN]->(m:Movie)
create (p1)-[:KNOWS]->(p2)
```



Graph Updates vs. Reasoning

- Inference in Cipher
  - We can infer additional edges using SET/CREATE commands
  - Those only apply for the current state of the graph
  - i.e., later changes are not respected
- Consider again

```
match (p:Person)-[ACTED_IN]->(m:Movie)
set p:Actor
```

  - Here, a later addition of a person acting in a movie would not get the Actor label!

- Inference in RDF/S
  - Can be updated and/or evaluated at query time

Comparison LPG+Cypher vs. RDF*/SPARQL*

- Semantics
  - Open vs. closed

- Expressivitiy
  - Cypher: does not support quoted statements
  - Cypher: only simple properties (literal valued) on the edges, no relations from edges to entities

    → RDF*: slightly better support for n-ary relations
  - SPARQL*: limited support for path queries (e.g., no quantifiers)

- Inference
  - LPG: only graph updates
  - RDF*: subject to ongoing research

Summary

- Labeled Property Graphs
  - Close some modeling gaps of RDF
  - In particular: complex relations, properties on relations

- RDF*/SPARQL*
  - Quoted vs. asserted statements

- LPG/Cipher:
  - Pattern based graph language
  - Querying and manipulating LPGs

Questions?