# GRAPH DATABASES:
# RDF AND PROPERTY GRAPHS

Ioana Manolescu
Inria and Ecole polytechnique
https://pages.saclay.inria.fr/ioana.manolescu
Ioana.manolescu@inria.fr

# Plan

1. Why do we need graph databases?

2. RDF graph databases

3. Property graph databases

# Data models, query languages, and data management systems

**Data model**: abstraction (usually with clear mathematical semantics) used to represent the data

- E.g., relational model: relation=set (or bag) of tuples

**Query language**: language (with completely specified grammar) used to express information needs to be answered over a dataset

- E.g., SQL
- Data Manipulation Language: query language + updates. Also part of SQL

**Data management system**: a system that provides CRUD (via DML) over data of a specific model

- CRUD: create, retrieve (=query), update, delete
- E.g., PostgreSQL, MySQL, Oracle, Microsoft, Amazon*, Google*, Snowflake, etc.

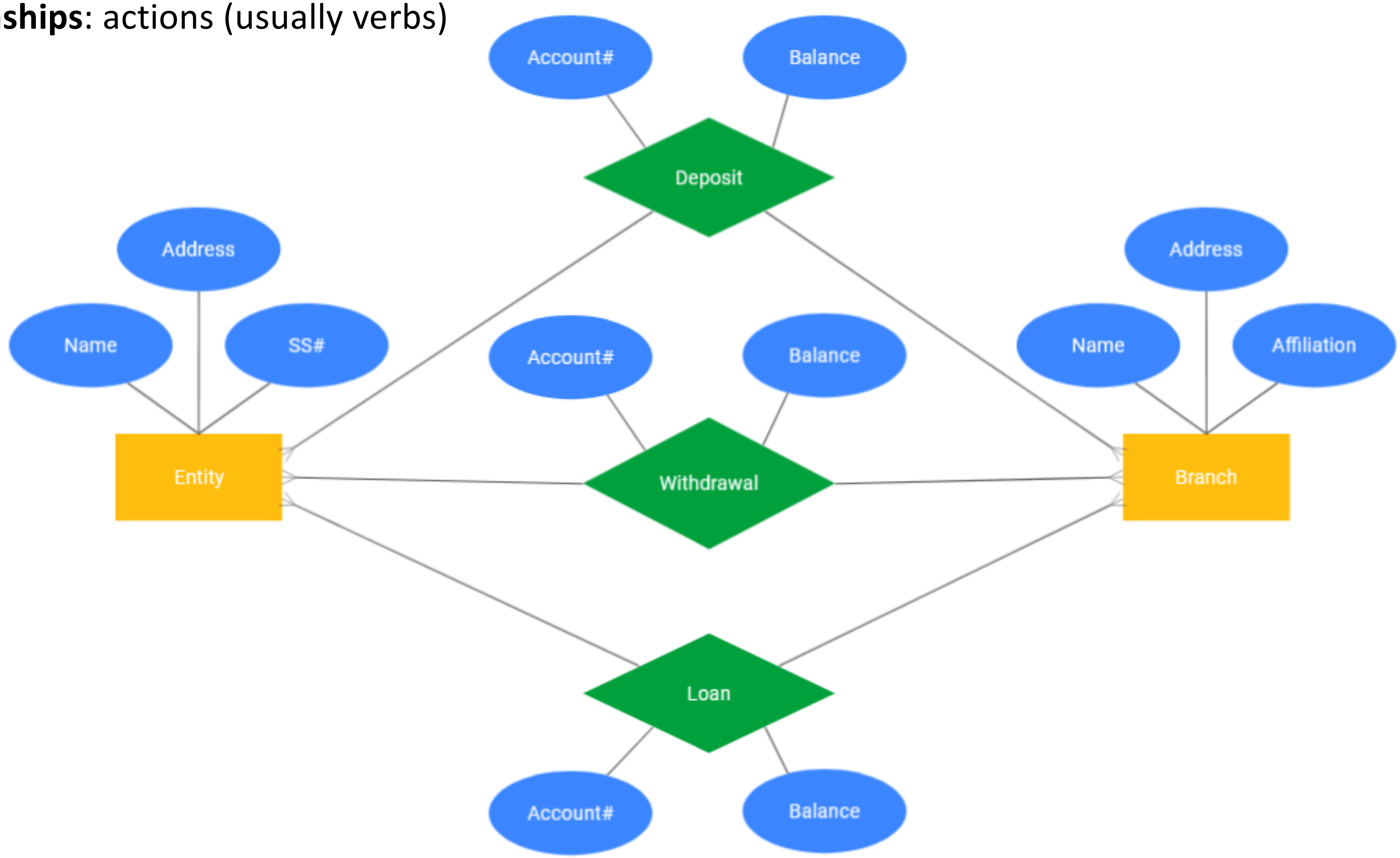
Today: graph data models and query languages. Next week: Neo4J lab.

# Why graph databases?

The real world consists of interconnected entities and relationships

**Entities**: nouns. One Entity in E-R diagram for each kind of real-world entity.

**Relationships**: actions (usually verbs)



# Why graph databases?

The real world consists of interconnected entities and relationships

**Entities**: nouns. One Entity in E-R diagram for each kind of real-world entity.

- Client, Bank, Branch, Product, Review, Song, …

**Relationships**: actions (usually verbs)

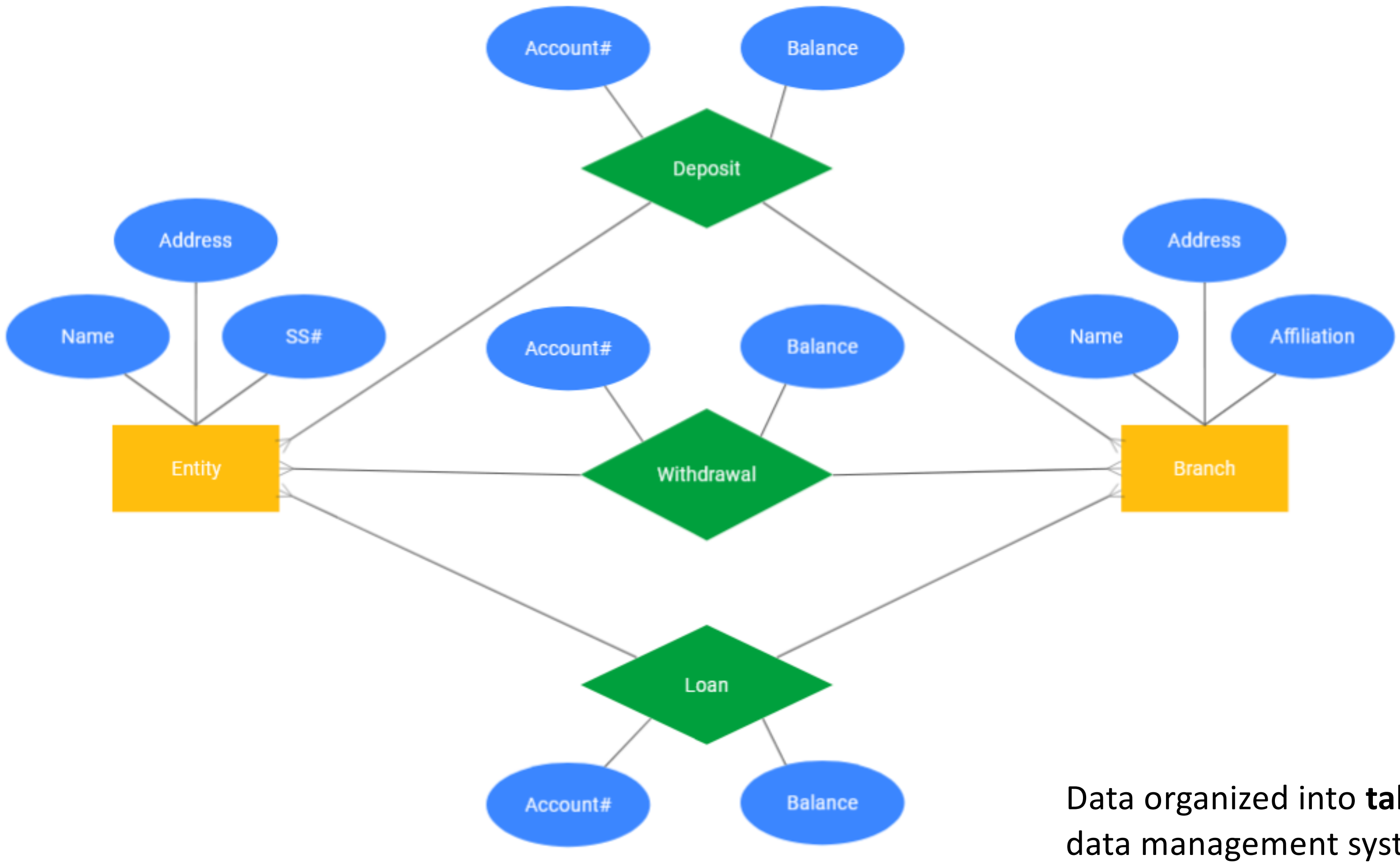
- E.g., Buys, Borrows, Writes, Likes, …
- Higher-arity relationships can connect more than two entities
  - E.g., buysHouseFrom between Buyer, Seller, Notarybuyer, NotarySeller

The first **scalable databases** have been relational: data stored in **tables**.

Thus, to set up a data management application:

1. Design the conceptual Entity-Relationship model
2. Turn each Entity into a **table**
3. Turn each Relationship into a **table**

# From a graph-structured model to a relational database

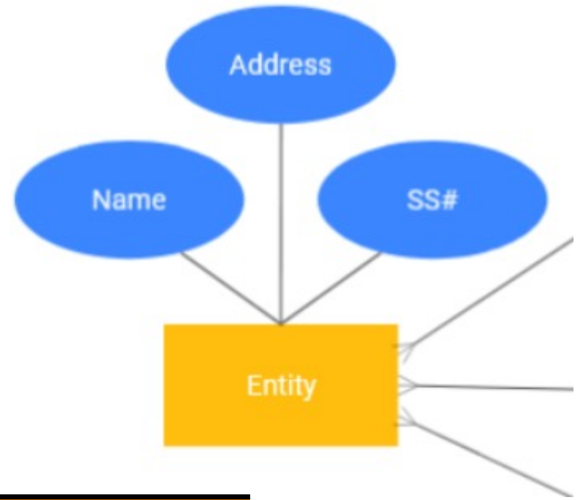



- **Entity**(Name, Address, SS#)
- **Branch**(Name, Address, Affiliation)
- **Deposit**(eName, bName, Account#, Balance)
- **Withdrawal**(eNo, bNo, Account#, Balance)
- **Loan**(eNo, bNo, Account#, Balance)

Data organized into **tables** because this is what data management systems handled best.

# What do we miss in relational databases? (1)

**Limited support for data heterogeneity**




- If different instances of an entity have different attributes, e.g., People have SS# and birthDate, but Companies have none of these and have registrationNo ?
- Add all attributes to the schema; use NULLs in the data

| Name | Address | SS# | Birthdate | Registration# |
|---|---|---|---|---|
| John | 1 main street | 123456 | 19/3/2001 | null |
| ACME | 10 main street | null | null | 98765 |
| | | | | |

- The attributes must still be known in advance

# What do we miss in relational databases? (2)

**Hard to write in SQL path queries: for each chain of money transfers from the entity named “Alice” to the entity named “Xing”, find the time and the amount**

Entity(ID, name), Account(eID, aNo), Transfer(a1, a2, date, amount)

```
WITH RECURSIVE chain(date, amount) AS (
   SELECT t.date, t.amount, a2.aNo
   FROM Entity ea, Account a1, Transfer t, Account a2
   WHERE ea.name='Alice' and ea.ID=a.eID and and a.aNo=t.a1 and t.a2=a2.aNo

   UNION ALL

   SELECT t2.date, t2.amount, t2.a2
   FROM chain c, Transfer t2
   WHERE c.a2=t2.a1 and c.date<=t2.date)

SELECT sum(amount), min(date), max(date)
FROM chain c, Account ax, Entity ex
WHERE c.a2No=ax.aNo and ax.eID=ex.ID and ex.name='Xing'
```

# What do we miss in relational databases? (3)

**Hard to write in SQL queries such as: for each chain of money transfers from the entity named "Alice" to the entity named "Xing", find the time and the amount**

Entity(ID, name), Account(eID, aNo), Transfer(a1, a2, date, amount)

WITH RECURSIVE chain(date, amount) AS ( … )  SELECT … FROM chain…

Note: we avoided cycles by asking that the dates be increasing

"When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. Sometimes, using UNION instead of UNION ALL can accomplish this by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are completely duplicate: it may be necessary to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the already-visited values."

# What do we miss in relational databases? (4)

**Impossible to write queries over the schema and the data**

- "For each table and each of their attributes, if the attribute name starts with 'Pers', show the value of the attribute"

**Dataset interoperability**

- If two databases have Entities numbered 1, 2, 3, … these IDs are local to each database.
- If each database contains company 'ACME', is it really the same?

**Databases store data, not knowledge**

- **Knowledge**: any Student is a Person; if X teaches a class, then X is an Instructor
- Therefore, databases do not **reason**
  - A query asking for Person instances will not return Students
  - Unless we explicitly copy every instance of Student in Person… and even that does not always work (which attributes are present in both tables?)
  - Distinguish from *enforcing constraints* (that is a strength of relational DBMSs)

# Recap: main limitations in relational databases

1. **Limited support for path queries**
2. **Impossible to write queries over schema and data**
3. **Database interoperability not clear**
4. **No support for knowledge and reasoning**

# Enter graph databases

| | Relational databases | RDF databases | Property graph databases |
|---|---|---|---|
| Path queries | Barely (hard) | ✓ | ✓ |
| Query schema and data | — | ✓ | ✓ |
| Database interoperability | — | ✓ | — |
| Reasoning | — | ✓ | — |

RDF has lead to wide-scale interoperability of data and knowledge.

Property graphs are more attractive in a single-organization (company) setting, in particular because they facilitate **efficient querying**.

# Graph database ranking

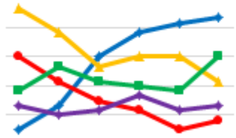
Ranking > Graph DBMS  RSS Feed

## DB-Engines Ranking of Graph DBMS

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

This is a partial list of the complete ranking showing only graph DBMS.

Read more about the method of calculating the scores.



trend chart

☐ include secondary database models  43 systems in ranking, November 2024

| Rank Nov 2024 | Rank Oct 2024 | Rank Nov 2023 | DBMS | Database Model | Score Nov 2024 | Score Oct 2024 | Score Nov 2023 |
|---|---|---|---|---|---|---|---|
| 1. | 1. | 1. | Neo4j | Graph | 42.70 | +0.19 | -7.00 |
| 2. | 2. | 2. | Microsoft Azure Cosmos DB | Multi-model | 23.95 | -0.55 | -10.16 |
| 3. | 3. | 3. | Aerospike | Multi-model | 5.32 | -0.25 | -1.90 |
| 4. | 4. | 4. | Virtuoso | Multi-model | 3.87 | -0.04 | -1.75 |
| 5. | 5. | 5. | ArangoDB | Multi-model | 3.09 | -0.35 | -1.45 |
| 6. | 6. | 6. | OrientDB | Multi-model | 2.97 | -0.06 | -0.81 |
| 7. | ↑ 8. | ↑ 8. | GraphDB | Multi-model | 2.89 | +0.12 | +0.14 |
| 8. | ↓ 7. | ↓ 7. | Memgraph | Graph | 2.72 | -0.09 | -0.37 |
| 9. | 9. | ↑ 10. | Amazon Neptune | Multi-model | 2.17 | 0.00 | -0.32 |
| 10. | ↑ 11. | ↓ 9. | NebulaGraph | Graph | 1.79 | -0.08 | -0.75 |
| 11. | ↓ 10. | 11. | Stardog | Multi-model | 1.78 | -0.13 | -0.52 |
| 12. | 12. | 12. | JanusGraph | Graph | 1.78 | 0.00 | -0.32 |
| 13. | 13. | ↑ 15. | Fauna | Multi-model | 1.43 | -0.07 | -0.35 |
| 14. | 14. | ↓ 13. | TigerGraph | Graph | 1.40 | -0.05 | -0.58 |

RDF

Strong, complete suite of standards

Property Graphs (some flavor of)

# RDF databases

# RDF graph: basic concepts

**RDF: Resource Description Framework**

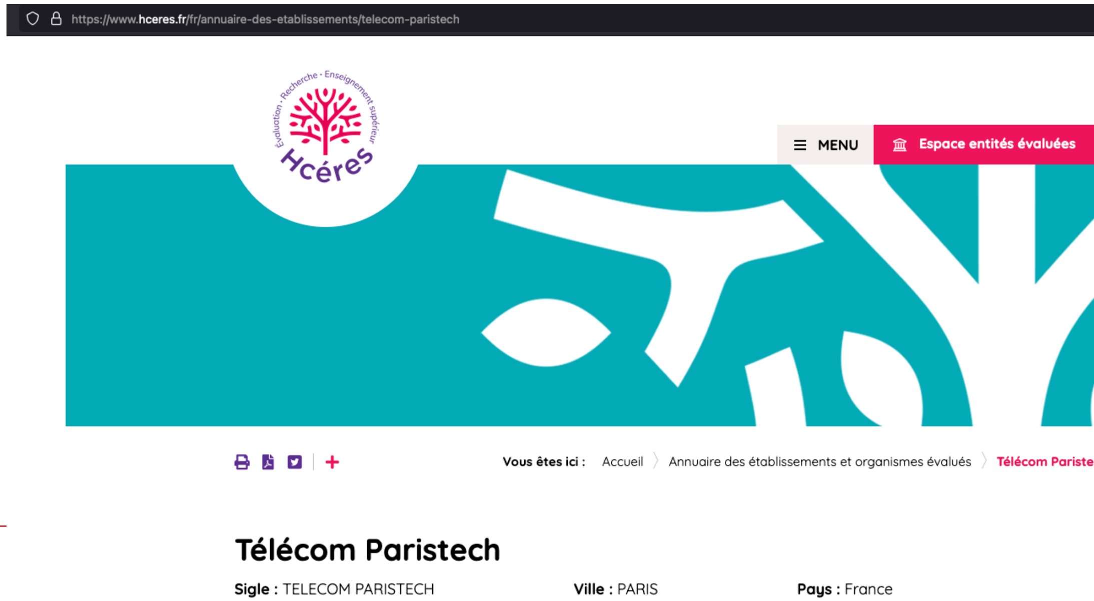
Each piece of data states a value for a property that a resource (subject) has

(subject, property, object) or (subject, property, value)

An RDF database, or graph, is a set of **triples**.

In each triple, the subject is an **International Resource Identifier** (or IRI), whose format is standardized. Previously known as URIs.

E.g., https://www.hceres.fr/fr/annuaire-des-etablissements/telecom-paristech : identifier of Télécom Paristech according to HCERES



# RDF graph: basic concepts

**RDF: Resource Description Framework**

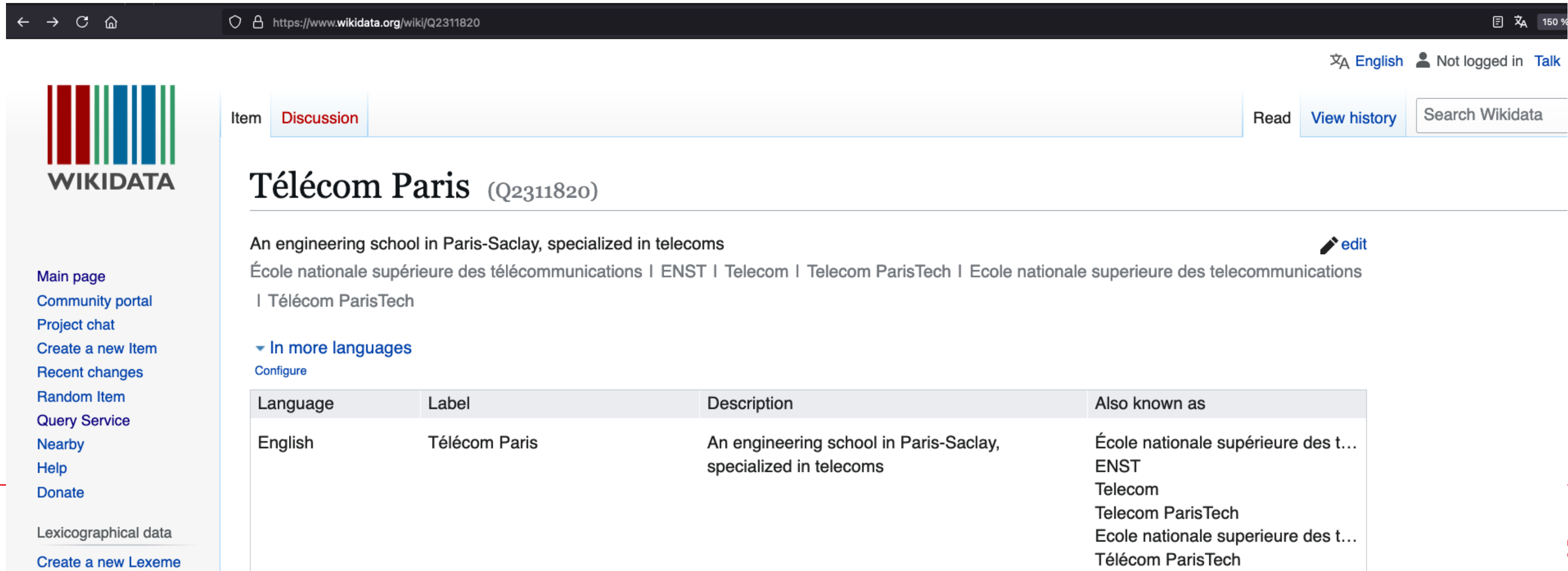
Each piece of data states a value for a property that a resource (subject) has

(subject, property, object) or (subject, property, value)

An RDF database, or graph, is a set of **triples**.

In each triple, the subject is an **International Resource Identifier** (or IRI), whose format is standardized. Previously known as URIs.

E.g., https://www.wikidata.org/wiki/Q2311820 : identifier of Télécom Paristech according to Wikidata, a large RDF graph online



# RDF graph: basic concepts

**RDF: Resource Description Framework**

Each piece of data states a value for a property that a resource (subject) has

(subject, property, object) or (subject, property, value)

An RDF database, or graph, is a set of **triples**.

In each triple, the subject is an **International Resource Identifier** (or IRI), whose format is standardized. Previously, Universal Resource Identifiers (URIs) .

- https://www.hceres.fr/fr/annuaire-des-etablissements/telecom-paristech is more readable… for those that use a Latin alphabet
  - IRIs allow to use many more alphabets

- URIs are sometimes informative for a human.
  - When they are not, e.g., https://www.wikidata.org/wiki/Q2311820, usually the label or name of the resource is described in the RDF graph

# IRIs enable interoperability between datasets!

**RDF: Resource Description Framework**

Each piece of data states a value for a property that a resource (subject) has

(subject, property, object) or (subject, property, value)

An RDF database, or graph, is a set of **triples**.

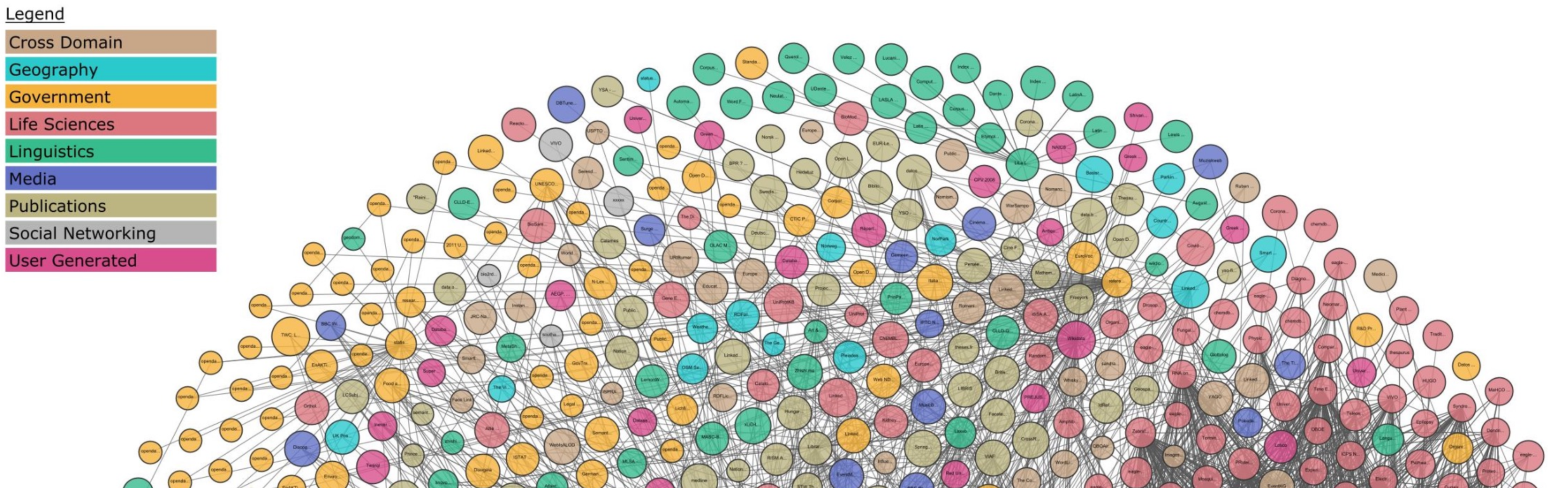
In each triple, the subject is an **International Resource Identifier** (or IRI), whose format is standardized. Previously, Universal Resource Identifiers (URIs).

All properties are also IRIs.

Values may also be IRIs.

IRIs ensure interoperability!

The Linked Open Data Cloud



# RDF graphs: IRIs vs. literals

Subjects, and properties, must be IRIs. Objects can be IRIs or values.

Below, we use *N-triples* syntax for RDF: `s p o .`

```
<http://data.kasabi.com/dataset/nasa/launchsite/canaryislands>
<http://purl.org/net/schemas/space/country> "Spain" .

<http://data.kasabi.com/dataset/nasa/launchsite/capecanaveral>
<http://purl.org/net/schemas/space/country> "United States" .

<http://data.kasabi.com/dataset/nasa/launchsite/kourou>
<http://purl.org/net/schemas/space/country> "France" .


<http://nasa.dataincubator.org/launch/1998-003>
<http://purl.org/net/schemas/space/launchsite>
<http://data.kasabi.com/dataset/nasa/launchsite/capecanaveral> .
```

# RDF: Practical details

Literals may have a **type** attached: uriJohn foaf:age "42"^^xsd:integer

- "42"^^xsd:integer is not the same as "42”

In an RDF graph description, one may introduce **IRI prefixes** and associate them short names, leading to a more compact syntax:

```
@prefix dt: <http://example.org/datatype#> .
@prefix ns: <http://example.org/ns#> .
@prefix : <http://example.org/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
:x ns:p "cat"@en .
:y ns:p "42"^^xsd:integer .
:z ns:p "abc"^^dt:specialDatatype .
```

This is the **Turtle** syntax for RDF
https://www.w3.org/TR/turtle/

Has exactly the same content as

```
<http://example.org/ns#x> <http://example.org/ns#p> "cat"@en .
<http://example.org/ns#y> <http://example.org/ns#p> "42"^^ <http://www.w3.org/2001/XMLSchema#integer> .
<http://example.org/ns#z> <http://example.org/ns#p> "abc"^^ <http://example.org/datatype#specialDatatype> .
```

An **RDF/XML** syntax for RDF also exists https://www.w3.org/TR/rdf-syntax-grammar/

# RDF graphs: types

Pre-defined `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` property allows attaching types to resources, e.g.:

```
<http://data.kasabi.com/dataset/nasa/launchsite/capecanaveral>
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
 <http://purl.org/net/schemas/space/LaunchSite> .
```

A resource may have several types, e.g.:

```
<http://data.kasabi.com/dataset/nasa/launchsite/capecanaveral>
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
 <https://www.wikidata.org/wiki/Q194188> .
```

spaceport

# RDF: Practical details

Very frequently used IRI prefixes from W3C standards:

| Prefix | IRI |
|---|---|
| rdf: | http://www.w3.org/1999/02/22-rdf-syntax-ns# |
| rdfs: | http://www.w3.org/2000/01/rdf-schema# |
| xsd: | http://www.w3.org/2001/XMLSchema# |
| fn: | http://www.w3.org/2005/xpath-functions# |

Other common namespaces:

| Prefix | Vocabulary description |
|---|---|
| foaf: | "Friend of a Friend", describing people and their relationships |
| dc: | "Dublin Core", metadata about creative works: title, year, author, license, … |
| schema: | Schema.org, a repository of type definitions for commonly encountered entities |

# RDF graphs: blank nodes

They correspond to entities for whom the IRI is not known.
They are of the form _:label

```
_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://purl.org/net/schemas/space/LaunchSite> .
<http://nasa.dataincubator.org/launch/2024-003>
<http://purl.org/net/schemas/space/launchsite> _:b1 .
```

**Within** an RDF graph, all occurrences of a BN denote the same “unknown” node.

**Across** RDF graphs, BN labels denote different nodes.
If graphs are **unioned**, the blank nodes are automatically relabeled:

| **Graph G1** | | **Graph G2** | | **Graph G3=G1 U G2** |
|---|---|---|---|---|
| _:b1, _:b2 | U | _:b1, _:b2, _:b3 | = | _b1, _b2, _b3, _b4, _b5 |

Blank nodes can be seen as “local IDs”

# Sample RDF graph (abridged)

```
uriJohn foaf:name “John” .
uriJohn nasa:crewOf nasa:apollo13 .
nasa:apollo13 rdf#type nasa:Spaceship .
uriJohn nasa:experimentAuthor _:exp1 .
_:exp1 rdf#type nasa:RadiologyExperiment .
```

We only know nasa:Spaceship and nasa:RadiologyExperiment are types (or classes) because they appear as values of rdf#type.

Only a few other predefined properties: rdf#comment, rdf#label.

Types can also have properties, e.g.:

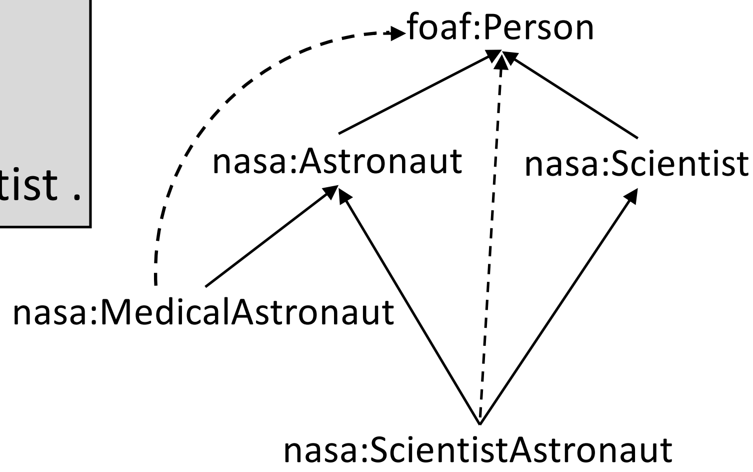
```
nasa:Spaceship schema:author uriAliceJones .
nasa:Spaceship schema:creationDate ”1/1/1998” .
```

# Adding semantics (knowledge) to RDF graphs: type hierarchies

**RDF Schema** (RDFS, in short) is the simplest language for describing knowledge that holds in RDF graphs.

RDFs defines the property rdfs:subclassOf:

nasa:Spaceship rdfs:subclassOf nasa:Vehicle .
nasa:Astronaut rdfs:subclassOf foaf:Person .
nasa:Scientist rdfs:subclassOf foaf:Person .
nasa:ScientistAstronaut rdfs:subclassOf nasa:Scientist .




subclassOf is naturally transitive. This leads to a first type of **reasoning**:

(t1, subclassOf, t2), (t2, subclassOf, t3) → (t1, subclassOf, t3)

**Implicit triples:**

nasa:MedicalAstronaut rdfs:subclassOf foaf:Person .

nasa:ScientistAstronaut rdfs:subclassOf foaf:Person .

# Sample graph enriched by reasoning with type hierarchies

**Explicit** triples:

uriJohn foaf:name “John” .

uriJohn nasa:crewOf nasa:apollo13 .

nasa:apollo13 rdf#type nasa:Spaceship .

uriJohn nasa:experimentAuthor _:exp1 .

_:exp1 rdf#type nasa:RadiologyExperiment .

nasa:Spaceship rdfs:subclassOf nasa:Vehicle .

nasa:Astronaut rdfs:subclassOf foaf:Person .

nasa:Scientist rdfs:subclassOf foaf:Person .

nasa:ScientistAstronaut rdfs:subclassOf nasa:Astronaut .

nasa:ScientistAstronaut rdfs:subclassOf nasa:Scientist .

**Implicit** triples:

nasa:ScientistAstronaut rdfs:subclassOf foaf:Person .

nasa:apollo13 rdf#type nasa:Vehicle .

**Data triples:** the property is not from an ontology language (the first five explicit triples)

**Schema** triples (the five rdfs:subclassOf explicit triples and the implicit nasa:ScientistAstronaut rdfs:subclassOf foaf:Person)

**Data** triple (nasa:apollo13 rdf#type nasa:Vehicle)

# More reasoning on RDF graphs: subproperty

Properties can be specializations of each other, just like classes:

```
nasa:experimentAuthor rdfs:subpropertyOf nasa:participatedTo .
```

Just like subclassOf, subPropertyOf is transitive (reasoning identical; examples ommitted).

From the explicit triples:

```
uriJohn nasa:experimentAuthor _:exp1 .
nasa:experimentAuthor rdfs:subpropertyOf nasa:participatedTo .
```

**Data** triple

**Schema** triple

We get the implicit triple:

```
uriJohn nasa:participatedTo _:exp1 .
```

**Data** triple

# More reasoning on RDF graphs: property domain and range

Some properties are naturally associated to some types

nasa:crewOf rdfs:domain nasa:Astronaut

nasa:crewOf rdfs:range nasa:Spaceship

**Semantics**: any subject of nasa:crewOf is automatically an astronaut; any object of nasa:crewOf is automatically a spaceship

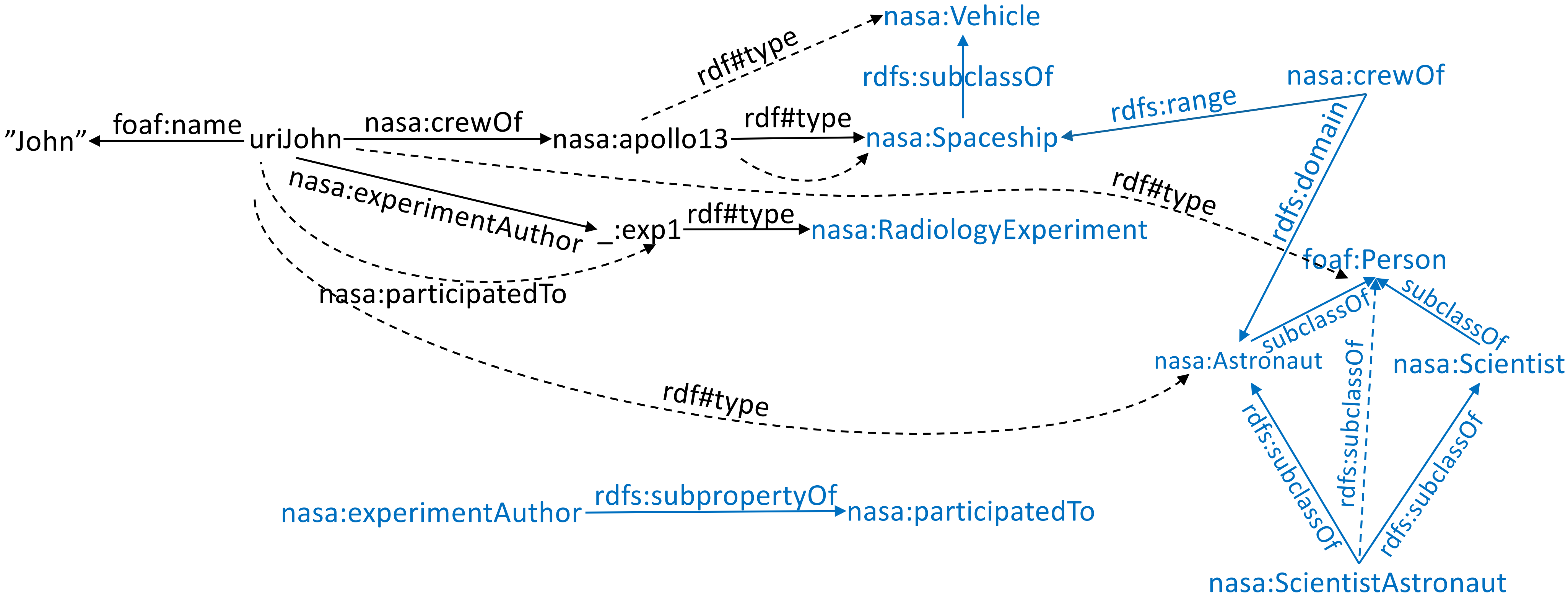
- Thanks to rdfs:domain and rdfs:range, the subject and object of every nasa:crewOf triple *gain more (implicit) type information*

Together, subclassOf, subpropertyOf, domain and range make up the **RDFS ontology language**, a small but useful language for expressing knowledge.

**Reasoning** with an ontology: enumerating all consequences (implicit triples) based on the data and schema triples, until no new triple can be inferred.

- For RDFS ontologies, this process terminates and runs in polynomial time in the number of triples from the graph + ontology.

# Sample reasoning on our NASA graph




Classes, properties, and schema triples in blue.

Explicit triples are shown by full-line arrows.
Implicit triples are shown by dashed lines.

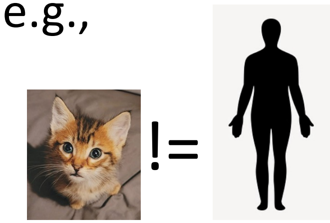
A given triple, explicitly present in the graph, may also be implicitly present. (They only "count as one".)

One method known to produce complete results: 1. Saturate the ontology. 2. Saturate the data graph with the (enriched) ontology.

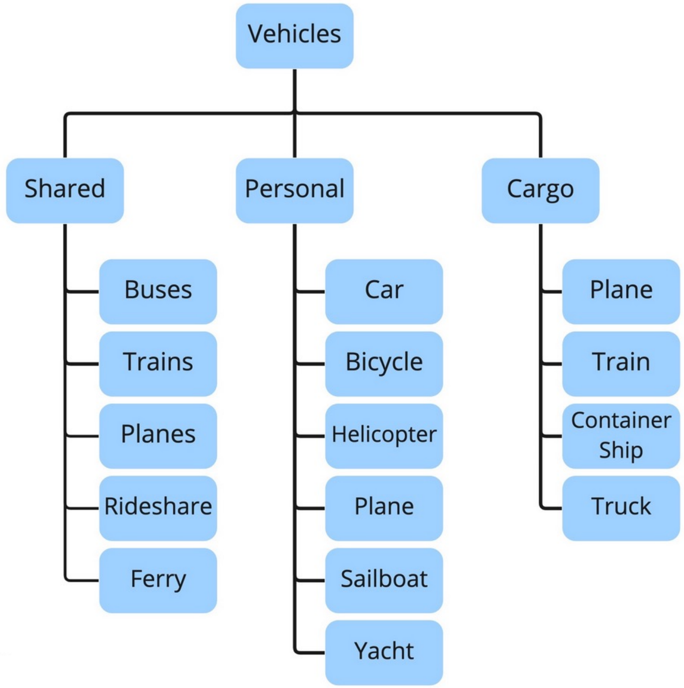
# Wrap-up on RDF ontologies and reasoning

RDFS is a <u>small</u> ontology language.

- Too small even to declare constraints, e.g., one cannot be a Human and a Cat; or, only Humans have VoterIDs; or, a person has at most two parents




Even <u>smaller</u>: use just subclassOf → taxonomy




<u>Larger</u> language used in more "industrial-strengh" applications: W3C OWL (Web Ontology Language)

OWL allows declaring:

- Cardinality constraints
- Class disjointness
- Constraints between classes and intersections/unions of classes/property domains, or property ranges…

# Wrap-up on RDF ontologies, reasoning, shape constraints

We have seen: Reasoning on an RDF graph with an RDF ontology, aka graph **saturation**

If we don't saturate, how to get complete results? **Reformulate** the query.

- If Astronaut subclassOfPerson, query asks for Person instances?

In the presence of an OWL ontology, the graph may be inconsistent (not satisfy rules).
We then look for repairs (minimal modifications to the graph).

The need for structure (shape) constraints has also been felt.
The **SHACL** language has been proposed for that.

- a stands for rdf:type; sh is SHACL namespace
- ReviewShape describes the expected shape of resources of type Review, which should have a rating.
- ratingShape describes the expected shape of the rating property
- Property values can also be constrained

```
ex:ReviewShape
  a sh:NodeShape ;
  sh:targetClass ex:Review ;
  sh:property ex:ratingShape .
ex:ratingShape
   a sh:PropertyShape ;
   sh:path ex:rating ;
   sh:datatype xsd:integer ;
   sh:minInclusive 1 ;
   sh:maxInclusive 5 ;
   sh:minCount 1 ;
   sh:maxCount 1 .
```

# RDF: what about relationships of higher arity?

E.g., buysHouseFrom between Buyer, Seller, NotaryBuyer, NotarySeller

How do we represent that ns:b1 buys house ns:h1 from seller ns:s1 with the help of buyer’s notary ns:n1 and seller’s notary ns:n2?

# Querying RDF databases with SPARQL

# SPARQL: the standard query language for RDF

**SPARQL allows to:**

1. Return values from one or several RDF graphs, matching a certain structural pattern and certain conditions
2. Build new graphs
3. Aggregate information
4. Check for the presence of complex paths
5. Update RDF graphs

# SPARQL: basic graph pattern queries

**Data:**

```
<http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> "SPARQL Tutorial" .
```

**Query:**

```
SELECT ?title
WHERE { <http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> ?title . }
```

**Result:**

| ?title |
| --- |
| "SPARQL Tutorial" |

**Data:**

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
```

**Query:**

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name . ?x foaf:mbox ?mbox }
```

**Result:**

| ?name | ?mbox |
| --- | --- |
| "Johnny Lee Outlaw" | <mailto:jlow@example.com> |
| "Peter Goodguy" | <mailto:peter@example.org> |

# SPARQL: basic graph pattern queries

**Data:**

```
@prefix dt: <http://example.org/datatype#> .
@prefix ns: <http://example.org/ns#> .
@prefix : <http://example.org/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
:x ns:p "cat"@en .
:y ns:p "42"^^xsd:integer .
:z ns:p "abc"^^dt:specialDatatype .
```

**Query:**

```
SELECT ?v WHERE { ?v ?p "cat" }
```

**Result:**

| ?v |
| --- |

**Query:**

```
SELECT ?v WHERE { ?v ?p "cat"@en }
```

**Result:**

| ?v |
| --- |
| <http://example.org/ns#x> |

**Query:**

```
SELECT ?v WHERE { ?v ?p 42 }
```

**Result:**

| ?v |
| --- |
| <http://example.org/ns#y> |

**Query:**

```
SELECT ?v
WHERE { ?v ?p "abc"^^<http://example.org/datatype#specialDatatype> }
```

**Result:**

| ?v |
| --- |
| <http://example.org/ns#z> |

# SPARQL: OPTIONAL patterns in graph pattern queries

**Data:**

```
@prefix foaf:     <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

**Query:**

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
         OPTIONAL { ?x foaf:homepage ?hpage }
       }
```

**Result:**

| ?name | ?mbox | ?hpage |
|---|---|---|
| "Alice" | | <http://work.example.org/alice/> |
| "Bob" | <mailto:bob@work.example> | |

# How are SPARQL basic pattern queries evaluated?

1. Because RDF graphs are very general, the only “regularity” we can assume is: **triple(s, p, o)**
2. Because URIs (and often literals) are long, storage typically **dictionary-encodes** them as integers.
3. The evaluation of an n-triple pattern requires **n-1 joins**.
   Join <u>estimation errors multiply</u> → bad optimizer choices!
   The triple table is typically large → <u>errors are costly</u>
4. We may store **one table per class**, and **one table per property**
   Depending on the graph, this may lead to many tables!
5. Or, we may store one table per **frequent class** and property, and store the rest in a triples table.
6. Still n-1 joins needed. No magic bullet.

triples

| s | p | o |
|---|---|---|
| | | |

enc_triples

| s | p | o |
|---|---|---|
| | | |

dictionary

| id | IRI_or_lit |
|---|---|
| | |

isCrewOf

| s | o |
|---|---|
| | |

Person

| s |
|---|
| |

# SPARQL: FILTERing basic graph pattern query matches

FILTER can express complex conditions over one or more variables.

**Data:**

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
@prefix ns:   <http://example.org/ns#> .
:book1  dc:title  "SPARQL Tutorial" .
:book1  ns:price  42 .
:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
```

**Query:**

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title
WHERE { ?x dc:title ?title FILTER regex(?title, "^SPARQL") }
```

**Result:**

| ?title |
|---|
| "SPARQL Tutorial" |

**Query:**

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?title ?price
WHERE { ?x ns:price ?price . FILTER (?price < 30.5)
          ?x dc:title ?title . }
```

**Result:**

| ?title | ?price |
|---|---|
| ”The Semantic Web" | 23 |

# SPARQL: blank nodes in query results; implicit variables

Because blank node labels don't matter much, blank nodes can be renamed before returning them.

**Data:**

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Alice" .
_:b  foaf:name   "Bob" .
```

**Query:**

```
PREFIX foaf:   http://xmlns.com/foaf/0.1/
SELECT ?x ?name
WHERE  { ?x foaf:name ?name }
```

**Result:**

| ?x | ?name |
|---|---|
| _:c | "Alice" |
| _:d | "Bob" |

**Query with explicit variables:**

```
PREFIX foaf:   http://xmlns.com/foaf/0.1/
SELECT ?x ?y
WHERE  { ?x foaf:name ?y }
```

**Equivalent query with implicit variables:**

```
PREFIX foaf:   http://xmlns.com/foaf/0.1/
{ ?x foaf:name ?y }
```

# SPARQL: creating new graphs

CONSTRUCT queries return new graphs (triples), not tables!
CONSTRUCT ensures the RDF data model is closed under SPARQL

- The set of all integers (Z) is closed under (+, -)
- The relational data model is closed under SQL

**Data:**

```
@prefix org:   <http://example.com/ns#> .

_:a  org:employeeName   "Alice" .
_:a  org:employeeId     12345 .

_:b  org:employeeName   "Bob" .
_:b  org:employeeId     67890 .
```

**Query:**

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX org: <http://example.com/ns#>
CONSTRUCT { ?x foaf:name ?name }
WHERE { ?x org:employeeName ?name }
```

**Result:**

```
@prefix org: <http://example.com/ns#> .
_:x foaf:name "Alice" .
_:y foaf:name "Bob" .
```

# SPARQL: aggregation

Very much like SQL

**Data:**

```
@prefix : <http://books.example/> .

:org1 :affiliates :auth1, :auth2 .
:auth1 :writesBook :book1, :book2 .
:book1 :price 9 .
:book2 :price 5 .
:auth2 :writesBook :book3 .
:book3 :price 7 .
:org2 :affiliates :auth3 .
:auth3 :writesBook :book4 .
:book4 :price 7 .
```

**Query:**

```
PREFIX : <http://books.example/>
SELECT (SUM(?lprice) AS ?totalPrice)
WHERE {?org :affiliates ?auth .
       ?auth   :writesBook ?book .
       ?book :price ?lprice . }
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

**Result:**

| ?totalprice |
|---|
| 21 |

# SPARQL: path expressions

For cases where there is some flexibility in the pattern that we want to match

**Alternative:**

{ :book1 dc:title|rdfs:label ?displayString }

**Concatenation:**

{?x foaf:knows/foaf:knows/foaf:name ?name . }

**Inverse:**

{ <mailto:alice@example> ^foaf:mbox ?x } which is the same as {?x foaf:mbox <mailto:alice@example>}

**Repeated labels along a path (at least once)**

```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows+/foaf:name ?name .
}
```

**Repeated labels along a path (zero or more least once)**

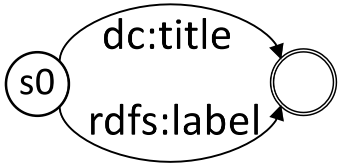
```
{
  ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows*/foaf:name ?name .
}
```

**Label negation**

{ ?x !(rdf:type|^rdf:type) ?y }

SPARQL enables checking the existence of paths whose labels match a given regular expression.
It does not enable (a) finding arbitrary paths, (b) returning paths, (c) limiting the path length.

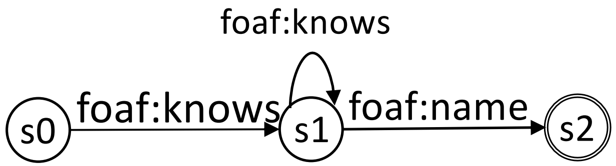
# SPARQL: path expressions and equivalent automata




{ :book1 dc:title|rdfs:label ?displayString }

{?x foaf:knows/foaf:knows/foaf:name ?name . }


{ ?x foaf:knows+/foaf:name ?name . }



# SPARQL allows to query the data together with the ontology

**Graph:**

```
@prefix nasa:      <http://nasa.org/knowledge/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
nasa:Neil_Armstrong rdf:type nasa:Astronaut.
nasa:Neil_Armstrong nasa:crewOf nasa:Apollo_13
```

**Query:**

```
PREFIX nasa: <http://nasa.org/knowledge/> . , rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?x, ?y, ?p, ?o
WHERE { ?x rdf:type ?y .  ?x p ?o . FILTER ?p != rdf:type}
```

**Result:**

| ?x | ?y |
|---|---|
| <http://www.nasa.org/knowledge/#Neil_Armstrong> | <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> |

# Property graphs databases

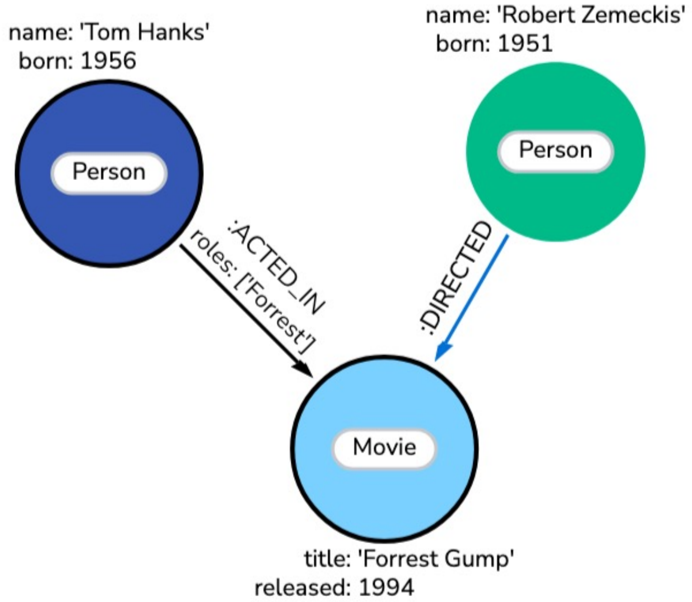
# Property graphs

**Nodes** have

- An id
- Zero or more attributes (name=value)
- Zero or more labels

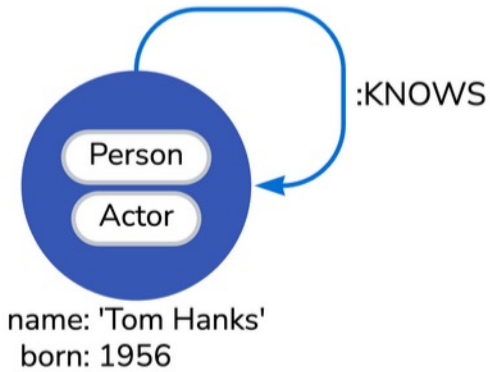
**Edges** (relationships) have

- A type
- A source and a target nodes
- Zero or more attributes (name=value)
- Some edges can be undirected!



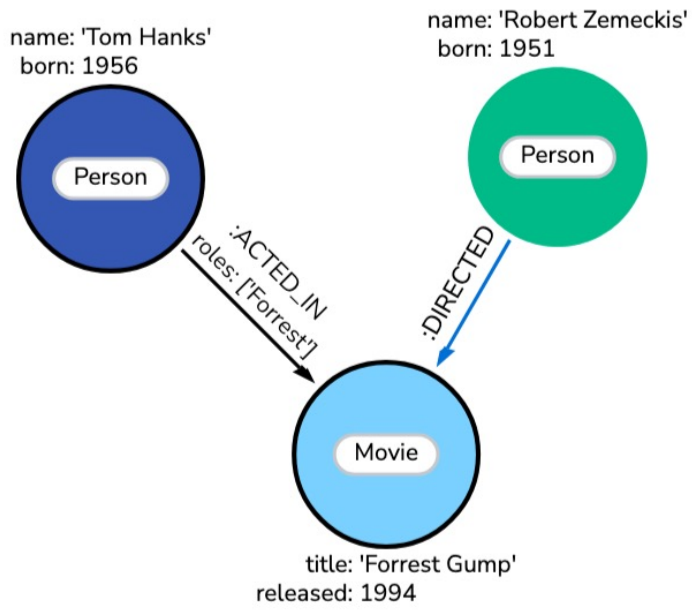

1-node graph

Can't do this in RDF!

# Property graphs: what about higher arity relationships?

How do we model that buyer Alice bought a house H from seller Bob with the notaries Jones and Thomson?



# Querying property graphs

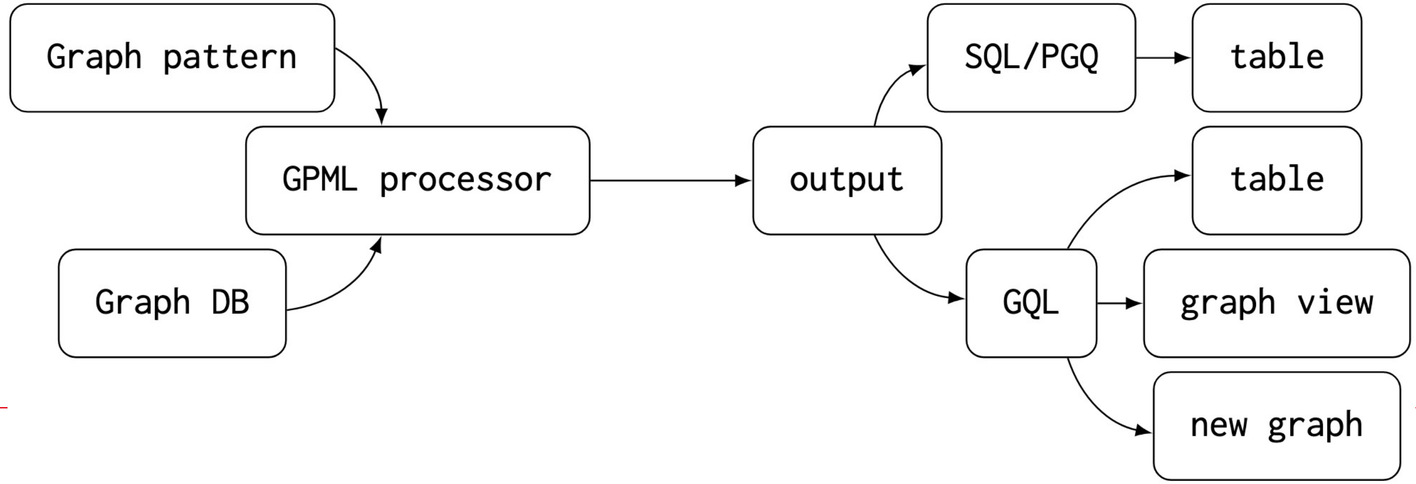
The Neo4J company put out “their own query language”: **Cypher**

- Since 2014 → implemented! Lab on Cypher

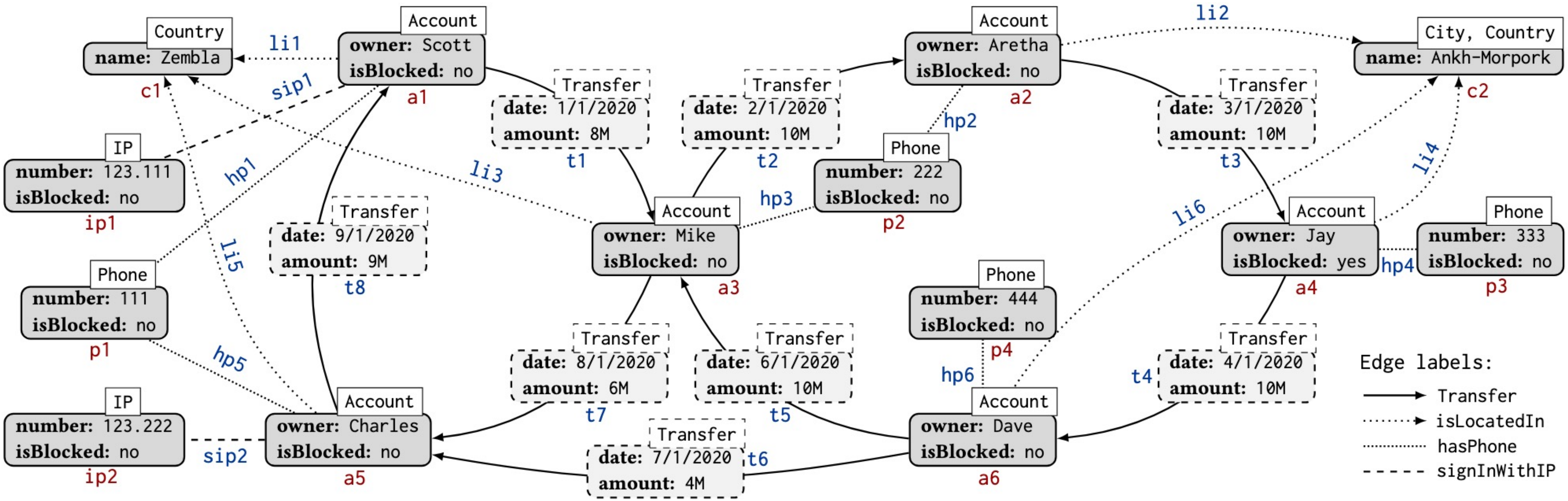
In 2019, ISO (the International Standard Organization, that standardizes SQL) has taken up the task of standardizing property graph querying. It has created:

1. **SQL/PGQ**, an extension of SQL to query graphs stored in relational tables
2. **GQL**, a query language completely separate from the relational model

- The graph pattern matching is the same in SQL/PGQ, and GQL.

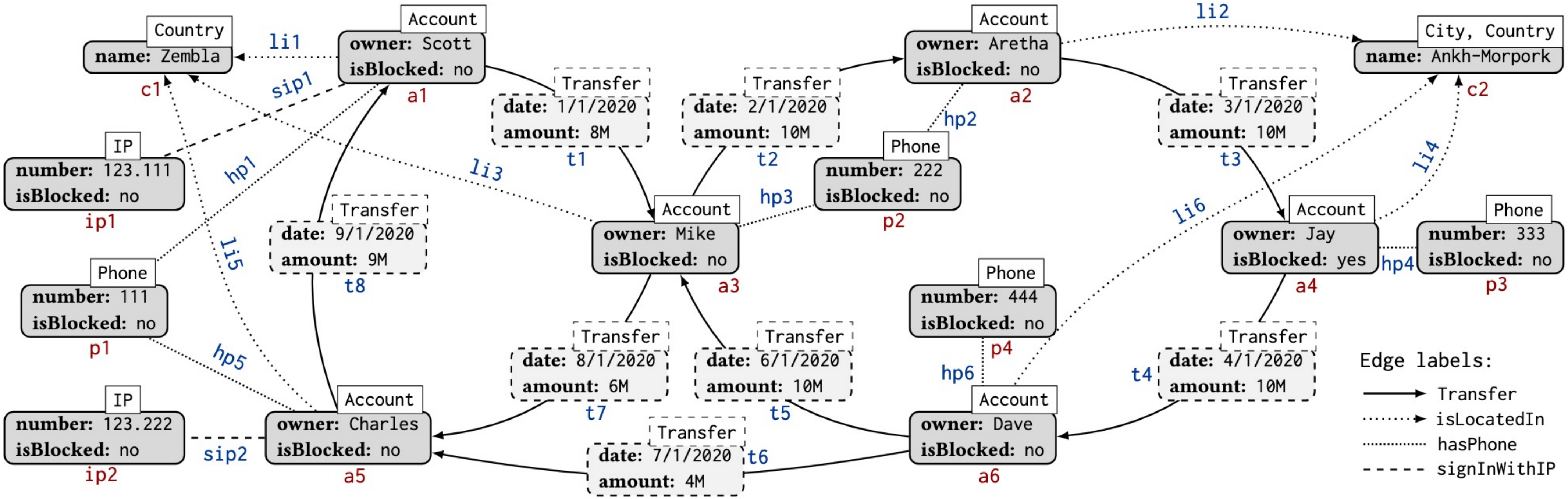


# Sample property graph to illustrate queries




From: Deutsch et al., Graph Pattern Matching in GQL and SQL/PGQ, SIGMOD 2022

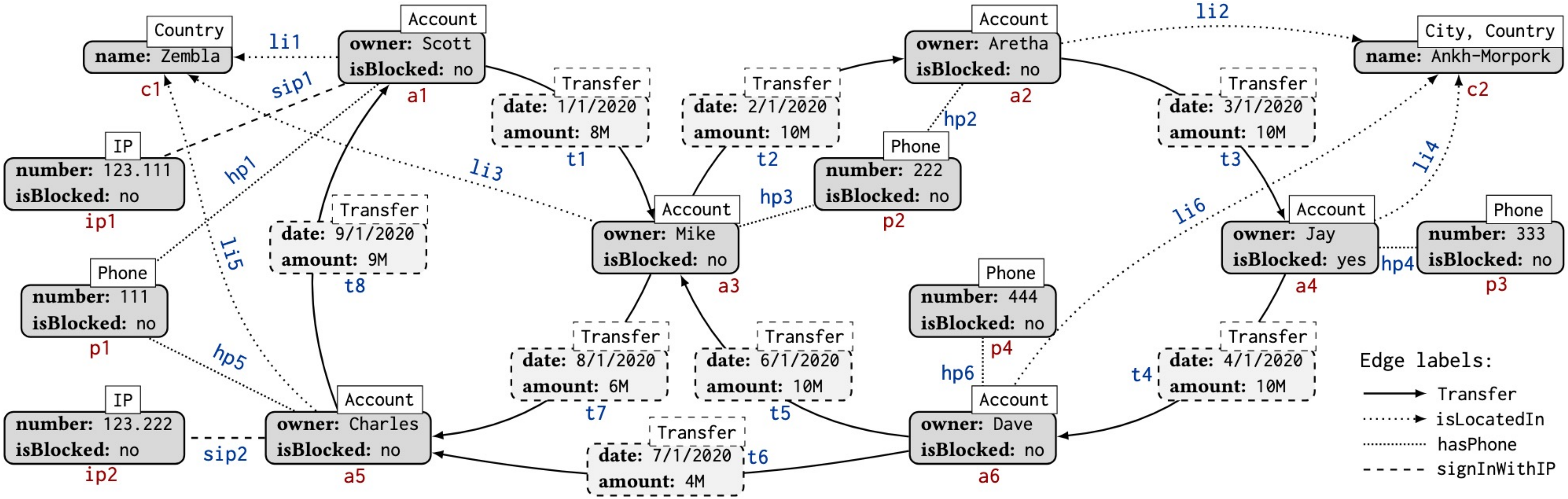
# Cypher pattern matching




**MATCH** (a:Account {isBlocked:'no'})−[:isLocatedIn]−>
(g:City {name:'Ankh−Morpork'})<−[:isLocatedIn]−(b:Account {isBlocked:'yes'}),
p = (a)−[:Transfer*1..] −>(b)
**RETURN** a.owner, b.owner, p

a, g, b, p: variables. ASCII art for edges. Regular path expressions (here: one or more Transfer edges)
Returns tables. Also: returns paths!

# Cypher pattern matching



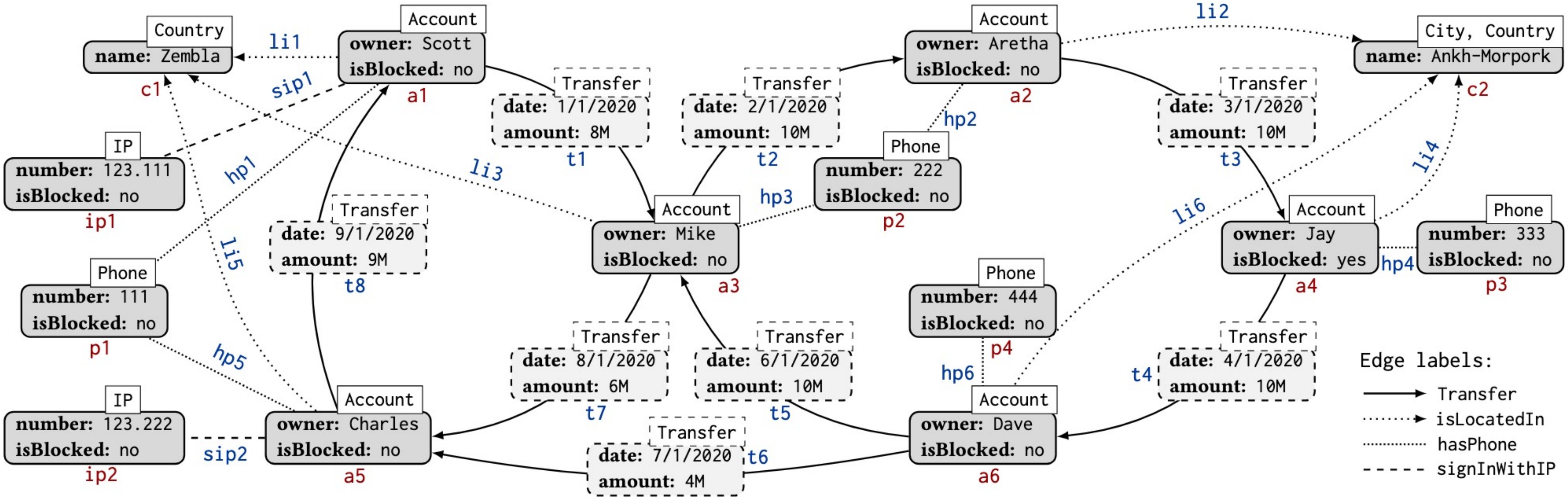

**We can put conditions on several attributes of the same node**

**MATCH** (a:Account {isBlocked:'no', owner: 'Scott'})… → **Less joins** needed to evaluate queries!

We can ask for the label(s) of a node or edge:

MATCH (a {number: '123.222'}) RETURN a.labels() → Querying the data together with the schema

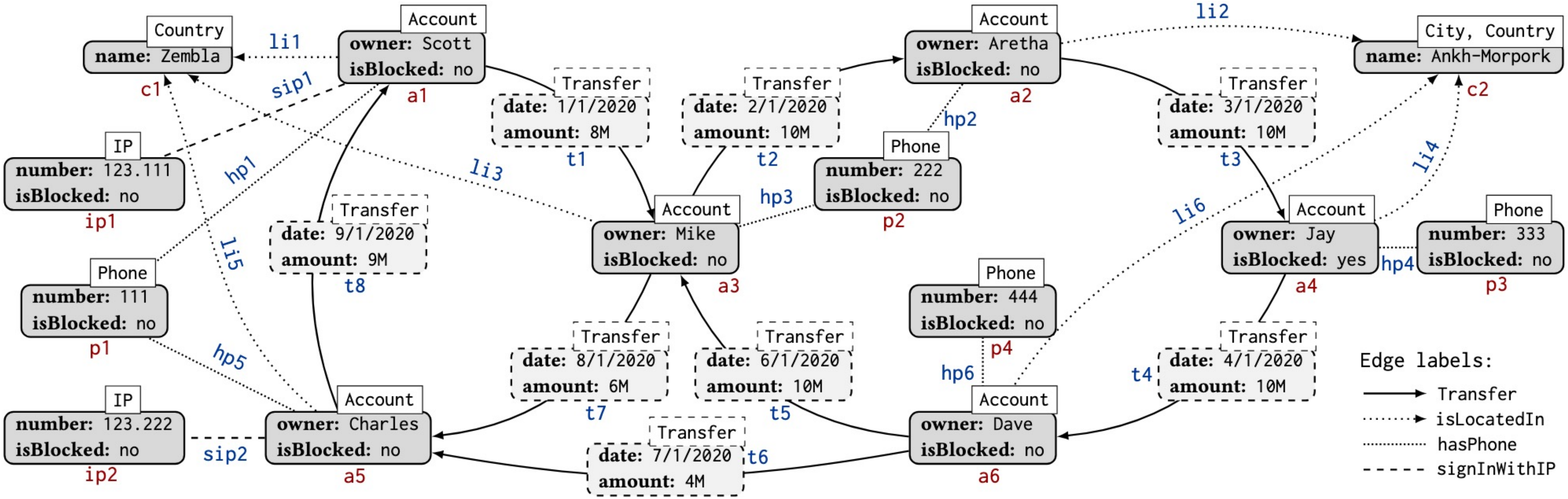
# Cypher pattern matching




**MATCH** (a:Account {isBlocked:'no'})−[:isLocatedIn]−>
(g:City {name:'Ankh−Morpork'})<−[:isLocatedIn]−(b:Account {isBlocked:'yes'}),
p = (a)−[*1..] −>(b)
**RETURN** a.owner, b.owner, p

Regular path expressions (here: one or more edges between a and b)

# Cypher pattern matching




**MATCH** p=shortestpath(a1:Account {owner:'Aretha'})−[:Transfer*]−> a2:Account{owner:’Charles’} )
**RETURN** p

A few more graph exploration algorithms (BFS, …) available as libraries one can call.

# More about Cypher and Neo4J

Cypher: Dominating industrial standard.

Neo4J strives to keep customers in by adding libraries for plenty of tasks:
Graph operations (BFS traversals, PageRank…)

Multiple parallel computation libraries (Map/Reduce…)

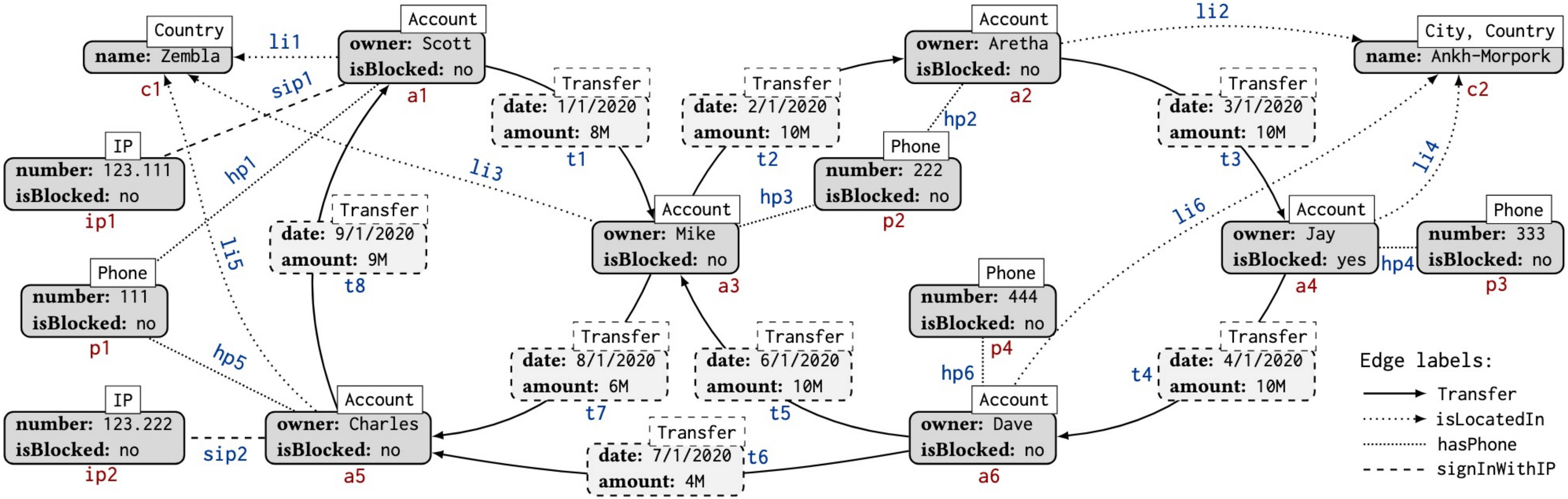
Large library of operations: APOC (Awesome Procedures on Cypher)
https://neo4j.com/docs/apoc/current/

Possibly fragilized by the standard imposed by multi B$ database companies

- Slight variations in data model (does every node need to have a label? Etc.)
- Path matching modes much richer in PGQL

The most widely implemented language so far → the lab next week.

# GQL pattern matching




Also “ASCII art”
SELECT-FROM-WHERE ☺
Richer path semantics (see next)

**SELECT** x.owner AS A, y.owner AS B
**FROM**
**MATCH** (x:Account)−[:isLocatedIn]−> (g:City)<−[:isLocatedIn]−(y:Account),
**MATCH ANY** (x)−[e:Transfer]−>+(y)
**WHERE** x.isBlocked='no' AND y.isBlocked='yes’ AND g.name='Ankh−Morpork'

# GQL pattern matching

Matching a **directed edge**: MATCH −[e]−>

Matching an **undirected edge**: MATCH ~[e]~

**Undirected or directed right to left**: MATCH <~[e]~

Find large transfers from accounts into which a login attempt was made from a blocked phone:

```
MATCH (p:Phone WHERE p.isBlocked='yes')~[e:hasPhone]
        ~(a1:Account)−[t:Transfer WHERE t.amount>1M]−>(a2)
```

To **constrain the length** of a path:

```
MATCH (a:Account) [()−[t:Transfer]−>() WHERE t.amount>1M] {2,5} (b:Account)
```

# Matching paths in GQL

**Path restrictors: what do we call a path?**

| | |
|---|---|
| TRAIL | No repeated edges (may repeat nodes!) |
| ACYCLIC | No repeated nodes (thus, cannot repeat edges) |
| SIMPLE | No repeated nodes, except that the first and last nodes may be the same. |

**Path selectors: which path(s) to return among those that match?**

| | |
|---|---|
| ANY SHORTEST | One path with shortest length for each (s, d). Non-deterministic |
| ALL SHORTEST | All paths with shortest length for each (s, d). |
| ANY | One path in each partition arbitrarily. Non-deterministic |
| ANY $k$ | Arbitrary $k$ paths in each partition (if fewer than $k$, return all). Non-deterministic. |
| SHORTEST $k$ | The shortest $k$ paths (if fewer than $k$, return all). Non-deterministic |
| SHORTEST $k$ GROUP | Group paths with same source and destination. For each (s, d), group paths again by length. Return $k$ shortest-length groups. |

***Questions?***