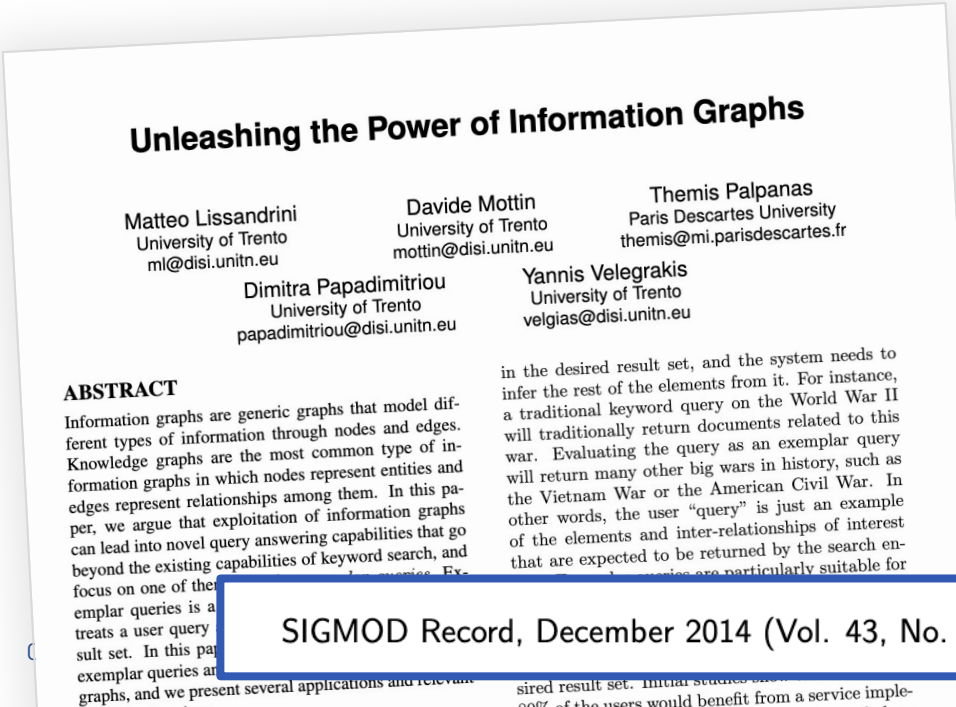
Data Science with Graphs

— Modelling & Querying Graphs —

Matteo Lissandrini – University of Verona

Hello! Καλημερα! Buongiorno!



Unleashing the Power of Information Graphs

Matteo Lissandrini
University of Trento
ml@disi.unitn.eu

Davide Mottin
University of Trento
mottin@disi.unitn.eu

Themis Palpanas
Paris Descartes University
themis@mi.parisdescartes.fr

Dimitra Papadimitriou
University of Trento
papadimitriou@disi.unitn.eu

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

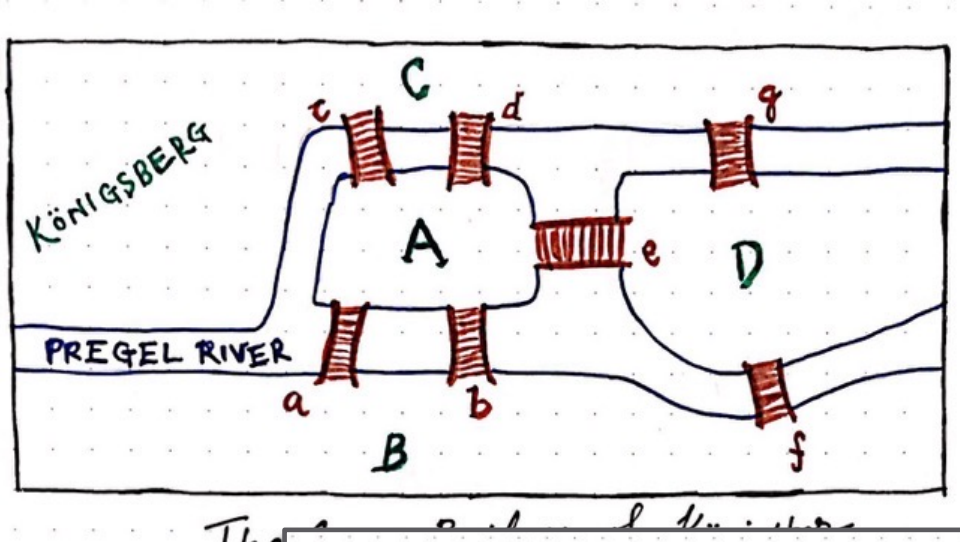
Information graphs are generic graphs that model different types of information through nodes and edges. Knowledge graphs are the most common type of information graphs in which nodes represent entities and edges represent relationships among them. In this paper, we argue that exploitation of information graphs can lead into novel query answering capabilities that go beyond the existing capabilities of keyword search, and focus on one of the[r] emplar queries is a[n] treats a user query sult set. In this pa[r] exemplar queries ar[e] graphs, and we present several applications and relevant

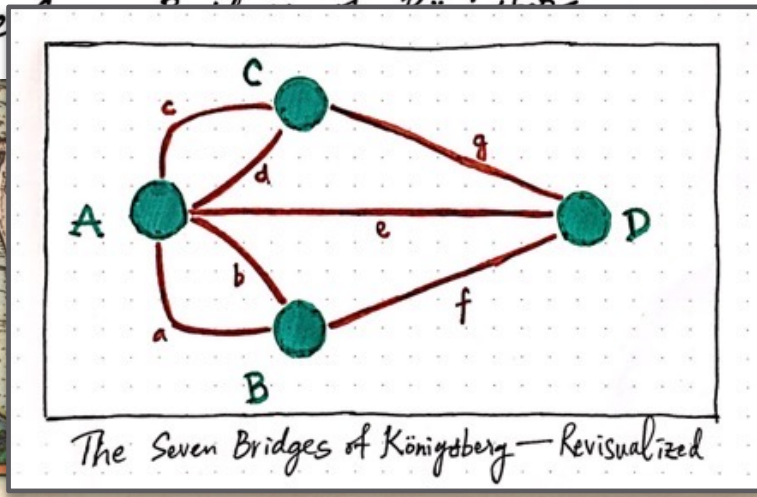
in the desired result set, and the system needs to infer the rest of the elements from it. For instance, a traditional keyword query on the World War II will traditionally return documents related to this war. Evaluating the query as an exemplar query will return many other big wars in history, such as the Vietnam War or the American Civil War. In other words, the user "query" is just an example of the elements and inter-relationships of interest that are expected to be returned by the search engine. [...] are particularly suitable for

[...] sired result set. Initial studies [...] [...]% of the users would benefit from a service imple-

SIGMOD Record, December 2014 (Vol. 43, No. 4)

Associate Professor
University of Verona (Italy)
https://lissandrini.com/





Welcome to **Königsberg**

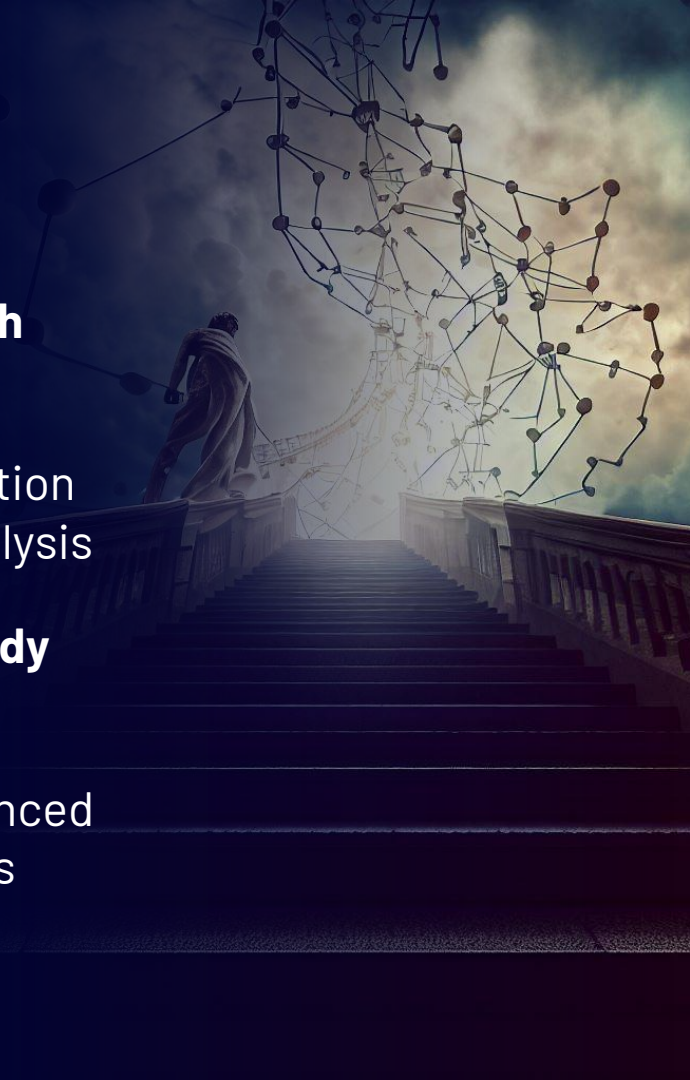
– Leonhard Euler, 1735

https://medium.com/basecs/k%C3%B6nigsberg-seven-small-bridges-one-giant-graph-problem-2275d1670a12

Course Objectives:

at the end of the course

1. You understand the different ways in which the **graph model can be adopted** in different domains
2. You are familiar with **core graph terminology** in relation to **challenges, methods, and solutions** for graph analysis
3. You can identify core methods and challenges **to study the content and structure** of a large graph
4. You have **concrete pointers and references** of advanced methods of data analysis and exploration with graphs



Agenda

- **Modelling & Querying Graphs**
- **Graph Structure Analysis**
- **Graph Exploration**
- **Data Exploration with Graphs**

Extra Materials:
slides contain extra materials that we will not be able to cover today. Feel free to ask questions about those.

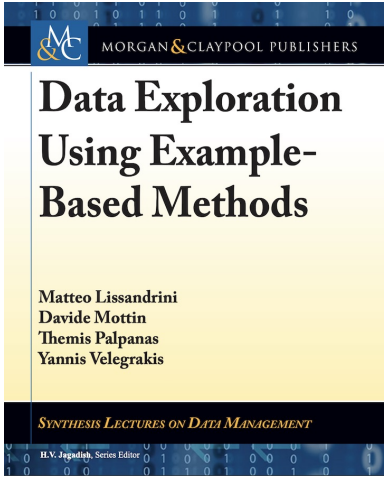
On References

Slides contain pointers to relevant materials

- Many slides have been adapted from existing courses and presentations; they are referenced whenever possible
- Some slides point to other online documentation, relevant Wikipedia pages (when sufficient), published papers, to expand when/if needed

Further Based on chapter & online slides:

- from Mining of Massive Datasets; Leskovec, Rajaraman, Ullman (3rd edition)
- from Web Data Mining; Bing Liu, Second Edition (July 2011)

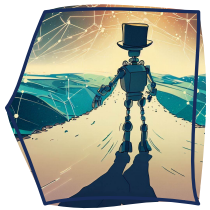


Data Exploration
https://data-exploration.eu

More references also at the end of the slides

Outline

1. **Graphs are Everywhere**
   - The Web-Link structure
   - The Query-Log graph
   - The Social network
   - The Knowledge graph

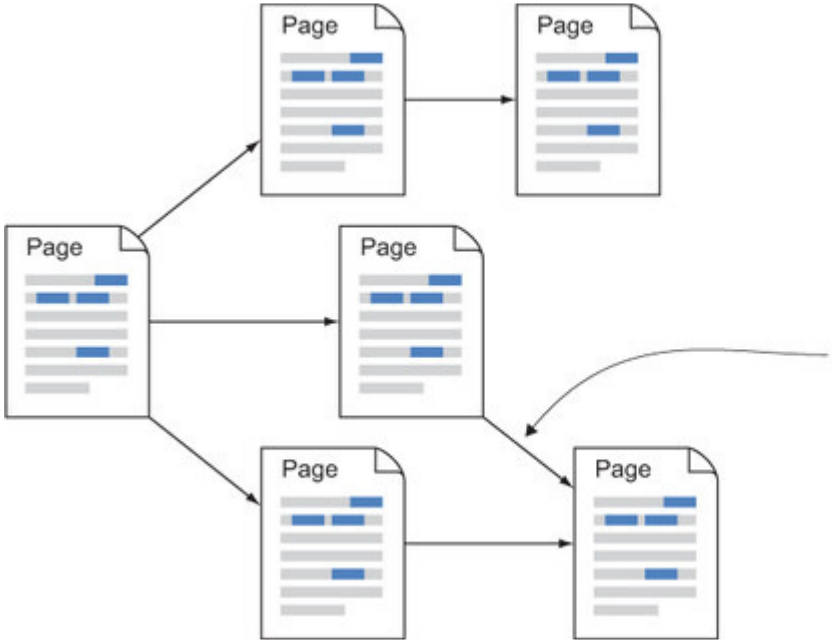


2. **The Graph Model**
   - Undirected/Directed graphs
   - Labelled/Unlabeled graphs
   - N-partite graphs
   - Heterogeneous Information Networks
   - RDF graphs
   - Property Graph
   - Graph Database vs. Database of Graphs
   - Knowledge Graphs

3. **Representing Graphs**
   - Adjacency matrix
   - Adjacency List
   - Triples & Storage for Triplestore
   - Property graph storage models

4. **Graph Navigation**
   - Breadth-First Search / Depth-First Search
   - Connected Components
   - Paths & Shortest path
   - CYPHER
   - SPARQL
   - Gremlin

Webpages and Links



WorldWideWeb is a hypertext browser/editor which allows one to read information from local files and remote servers. It allows **hypertext links to be made and traversed**, and also remote indexes to be interrogated for lists of useful documents. Local files may be edited, and **links made from areas of text to other files**, remote files, remote indexes, remote index searches, internet news groups and articles. All these sources of information are presented in a consistent way to the reader. For example, an index search returns a **hypertext document with pointers to documents** matching the query. Internet news articles are displayed with **hypertext links to other referenced articles** and groups.

– Tim Berners-Lee, 20 Aug 1991

Hyper-Links:
Text in pages links
to other pages containing
relevant information

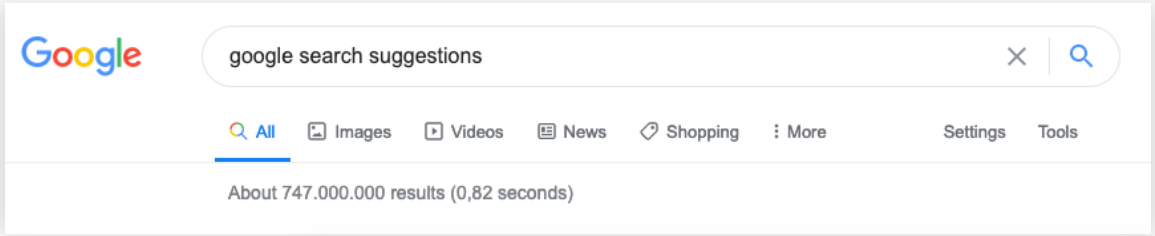
Traversing a Link:

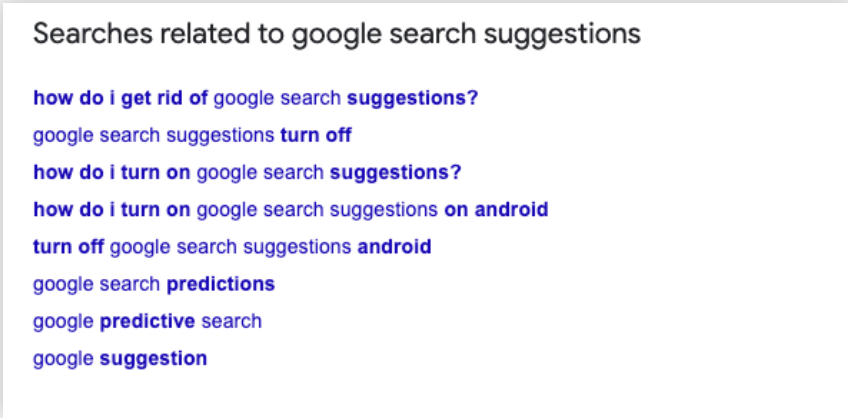
A link from Page A to Page B tells us that there is a relationship between the two documents.
Page A mentions something for which Page B contains additional relevant information

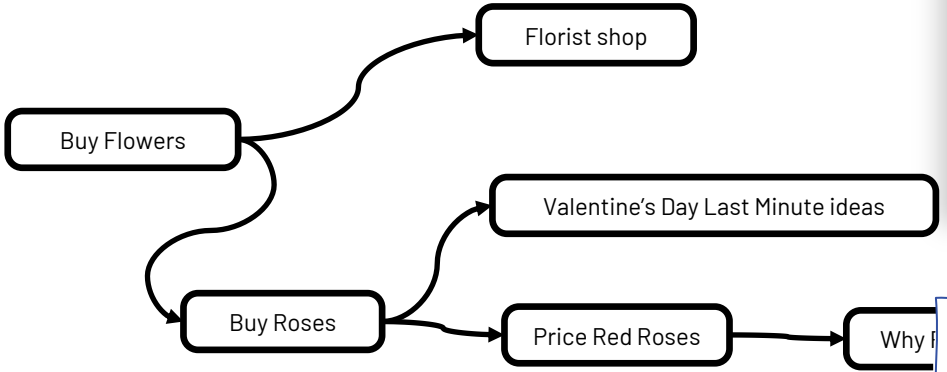
Query Logs

Monitoring Search Activites

1. Record Search query + User Click: Click Log
2. Record for the same user keyword search happening one after the other: Search Session







Session Links:
The user search for B after searching for A

Implicit Links:

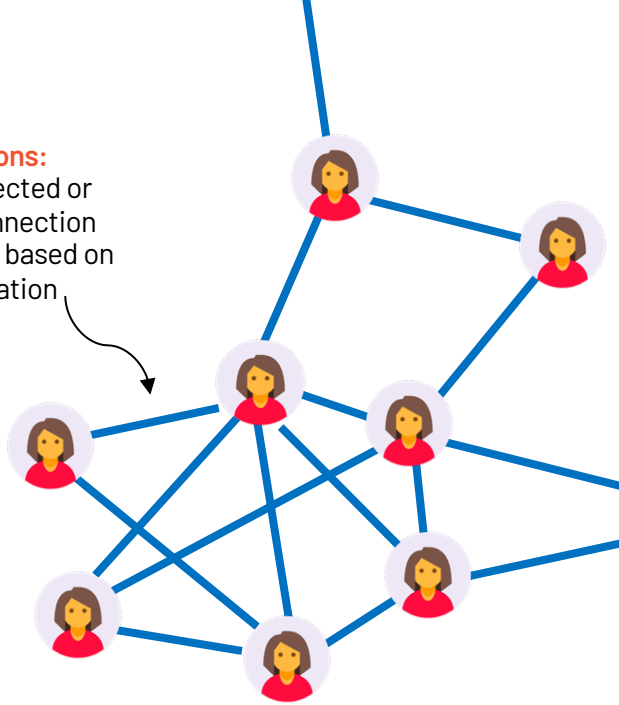
We infer a link from the behaviour of the user

http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/

Social Networks

User connections

- On a social media platform users can express explicitly their connection with other users
- Connection can be:
    1. Undirected: friendship, colleague
    2. Directed: Follow



Up to here, objects and links are all of the same type:

Pages, Search queries, Users...

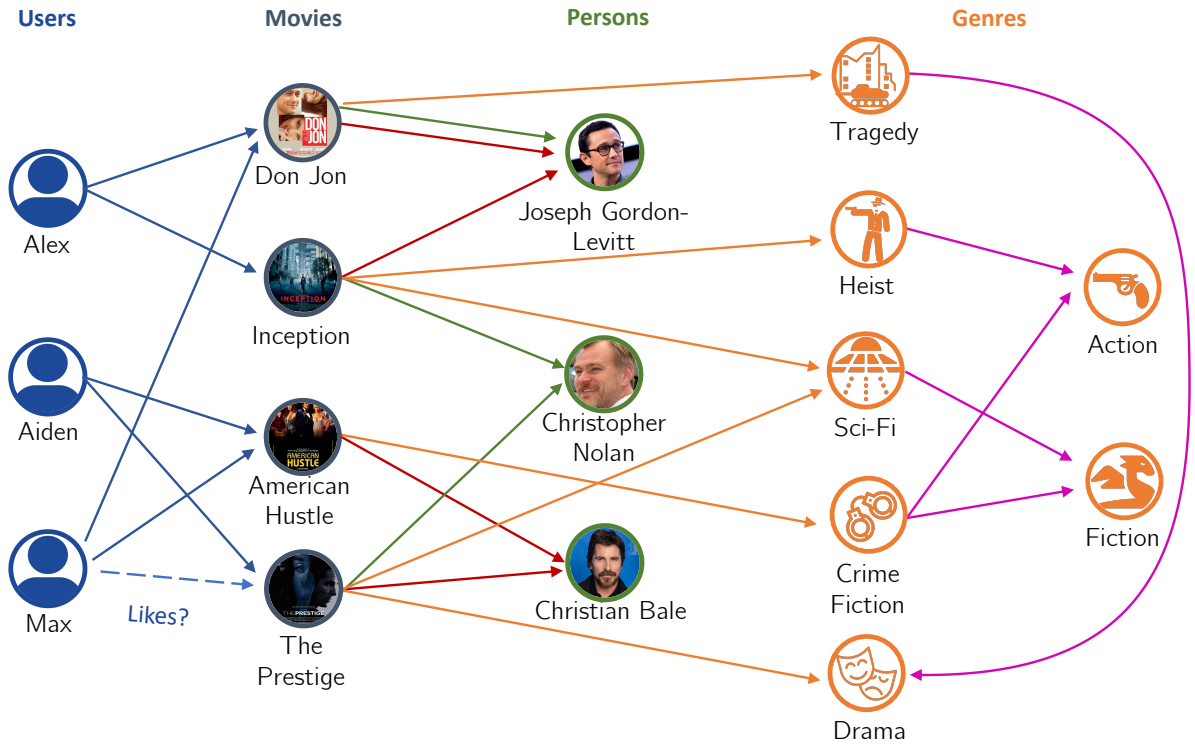
Product & Customer Networks

Heterogenous Information Networks:

Nodes are of different types

User-product & product-product connections

- Main nodes are customers & products, edges are transactions or interactions
- Other nodes can describe products



→ Likes

→ Starring

→ Directed by

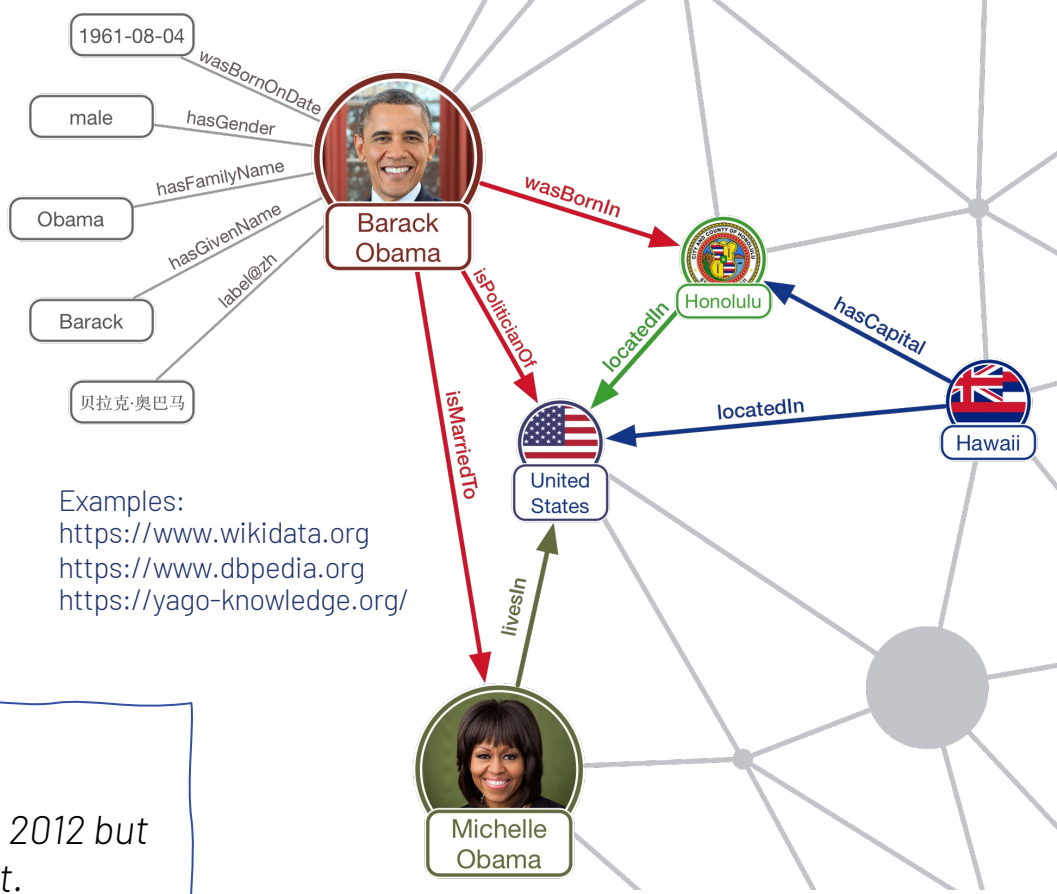
→ Has genre

→ Subgenre of

“Concept Networks”

Abstract model of Knowledge

- Links represent «facts»
- Facts connect different objects
    1. Real Entities
    2. Abstract Concepts
    3. Pieces of Data
- Facts are of different type
  they have different meaning



Examples:
https://www.wikidata.org
https://www.dbpedia.org
https://yago-knowledge.org/

This model is called “Knowledge Graph”:

The term has been popularized by Google in 2012 but it existed in different forms earlier than that.

https://blog.google/products/search/introducing-knowledge-graph-things-not/

Knowledge Graph Adoption

Existing Open Knowledge Graphs

1.9B Facts

210M Facts

52M Facts

http://linkedlifedata.com/sources.html

6.7B Facts

https://pubchem.ncbi.nlm.nih.gov/docs/rdf

132B Facts

The Growing Role of Graphs & Knowledge Graphs

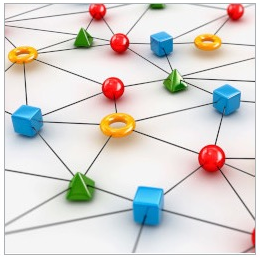
COMMUNICATIONS OF THE ACM

Home / Magazine Archive / August 2019 (Vol. 62, No. 8) / Industry-Scale Knowledge Graphs: Lessons and

PRACTICE

Industry-Scale Knowledge Graphs: Lessons and Challenges

By Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, Jamie Taylor
Communications of the ACM, August 2019, Vol. 62 No. 8, Pages 36-43
10.1145/3331166
Comments



Credit: Adempercem / Stutterstock

Knowledge graphs are critical to many enterprises today provide the structured data and factual knowledge that many products and make them more intelligent and "m

In general, a knowledge graph describes objects of inter connections between them. For example, a knowledge g have nodes for a movie, the actors in this movie, the dir so on. Each node may have properties such as an actor's and age. There may be nodes for multiple movies involv particular actor. The user can then traverse the knowled to collect information on all the movies in which the act appeared or, if applicable, directed.

Many practical implementations impose constraints on in knowledge graphs by defining a *schema* or *ontology*. For example, a link from a movie to its direct connect an object of type Movie to an object of type Person. In some cases the links themselves might have their own properties: a link connecting an actor and a movie might have the name of the specific role the actor played. Similarly, a link connecting a politician with a specific role in government might have the time period

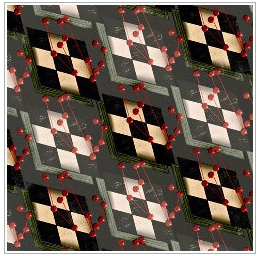
COMMUNICATIONS OF THE ACM

Home / Magazine Archive / September 2021 (Vol. 64, No. 9) / The Future Is Big Graphs: A Community View on Graph...

CONTRIBUTED ARTICLES

The Future Is Big Graphs: A Community View on Graph Processing Systems

By Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Hasihofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Ka Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Juan F. Sequeda, Joshua Shinavier
Communications of the ACM, September 2021, Vol. 64 No. 9, Pages 62-71
10.1145/3434642
Comments



Credit: Alli Torban

Graphs are, by nature, 'unifyi interconnectedness to repres real- and digital-world phenc consumers of graph instances these abstractions, future pro and systems. What needs to graph processing to continue

Back to Top

Key Insights

- Graphs ar enabling graph pro every dom
- Diverse w languages suitable a metrics w processin decade.

COMMUNICATIONS OF THE ACM

Home / Magazine Archive / March 2021 (Vol. 64, No. 3) / Knowledge Graphs / Full Text

REVIEW ARTICLES

Knowledge Graphs

By Claudio Gutierrez, Juan F. Sequeda
Communications of the ACM, March 2021, Vol. 64 No. 3, Pages 96-104
10.1145/3418294
Comments

"Those who cannot remember the past are condemned to repeat it."
—George Santayana

Back to Top

Key Insights

- Data was traditionally considered a material object, tied to bits, with no semantics per se. Knowledge was traditionally conceived as the immaterial object, living only in people's minds and language. The destinies of data and knowledge became bound together, becoming almost inseparable, by the emergence of digital computing in

.. a next-generation knowledge platform for **continuously integrating billions of facts** about real-world entities and powering experiences across a variety of production use cases.

[...]

The entries of data sources used to construct the KG **are continuously changing...**

[...]

Self-serve data onboarding: Low-effort **onboarding of new data sources** is important to ensure consistent growth of the KG.

Apple

Saga: A Platform for Continuous Construction and Serving of Knowledge At Scale

Ihab F. Ilyas, Theodoros Rekatsinas, Vishnu Konda
Jeffrey Pound, Xiaoguang Qi, Mohamed Soliman
Apple

ABSTRACT

We introduce Saga, a next-generation knowledge construction and serving platform for powering knowledge-based applications at industrial scale. Saga follows a hybrid batch-incremental design to continuously integrate billions of facts about real-world entities and construct a central knowledge graph that supports multiple production use cases with diverse requirements around data freshness, accuracy, and availability. In this paper, we discuss the unique challenges associated with knowledge graph construction at industrial scale, and review the main components of Saga and how they address these challenges. Finally, we share lessons-learned from a wide array of production use cases powered by Saga.

CCS CONCEPTS

• **Computer systems organization** → *Neural networks*; **Data flow architectures**; **Special purpose systems**; • **Information systems** → **Deduplication**; **Extraction, transformation and loading**; **Data cleaning**; **Entity resolution**.

KEYWORDS

knowledge graphs, knowledge graph construction, entity resolution, entity linking

ACM Reference Format:
Ihab F. Ilyas, Theodoros Rekatsinas, Vishnu Konda, Jeffrey Pound, Xiaoguang Qi, Mohamed Soliman. 2022. Saga: A Platform for Continuous Construction and Serving of Knowledge At Scale. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526049

1 INTRODUCTION

Accurate and up-to-date knowledge about real-world entities is needed in many applications. Search and assistant services require open-domain knowledge to power question answering. Other applications need rich entity data to render entity-centric experiences. Many internal applications in machine learning need training data sets with information on entities and their relationships. All of these applications require a broad range of knowledge that is accurate and continuously updated with facts about entities.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

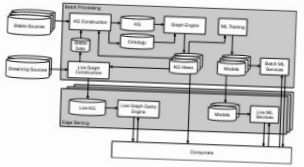


Figure 1: Overview of the Saga knowledge platform.

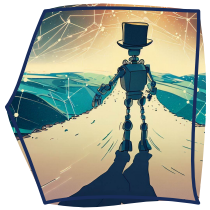
Constructing a central knowledge graph (KG) that can serve these needs is a challenging problem, and developing a KG construction and serving solution that can be shared across applications has obvious benefits. This paper describes our effort in building a next-generation knowledge platform for continuously integrating billions of facts about real-world entities and powering experiences across a variety of production use cases.

Knowledge can be represented as a graph with edges encoding *facts* amongst *entities* (nodes) [61]. Information about entities is obtained by integrating data from multiple structured databases and data records that are extracted from unstructured data [19]. The process of cleaning, integrating, and fusing this data into an accurate and canonical representation for each entity is referred to as *knowledge graph construction* [80]. Continuous construction and serving of knowledge plays a critical role as access to up-to-date and trustworthy information is key to user engagement. The entries of data sources used to construct the KG are continuously changing: new entities can appear, entities might be deleted, and facts about existing entities can change at different frequencies. Moreover, the set of input sources can be dynamic. Changes to licensing agreements or privacy and trustworthiness requirements can affect the set of admissible data sources during KG construction. Such data feeds impose unique requirements and challenges that a knowledge platform needs to handle:

(1) *Hybrid batch and stream construction:* Knowledge construction requires operating on data sources over heterogeneous domains. The update rates and freshness requirements can differ across sources. Updates from streaming sources with game scores need to be reflected in the KG within seconds but sources that focus on verticals such as songs can provide batch updates with millions of entries on a daily basis. Any platform for constructing and

Outline

1. **Graphs are Everywhere**
   - The Web-Link structure
   - The Query-Log graph
   - The Social network
   - The Knowledge graph



2. **The Graph Model**
   - Undirected/Directed graphs
   - Labelled/Unlabeled graphs
   - N-partite graphs
   - Heterogeneous Information Networks
   - RDF graphs
   - Property Graph
   - Graph Database vs. Database of Graphs

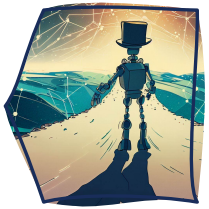
3. **Representing Graphs**
   - Adjacency matrix
   - Adjacency List
   - Triples & Storage for Triplestore
   - Property graph storage models

4. **Graph Navigation**
   - Breadth-First Search / Depth-First Search
   - Connected Components
   - Paths & Shortest path
   - CYPHER
   - SPARQL
   - Gremlin

Outline

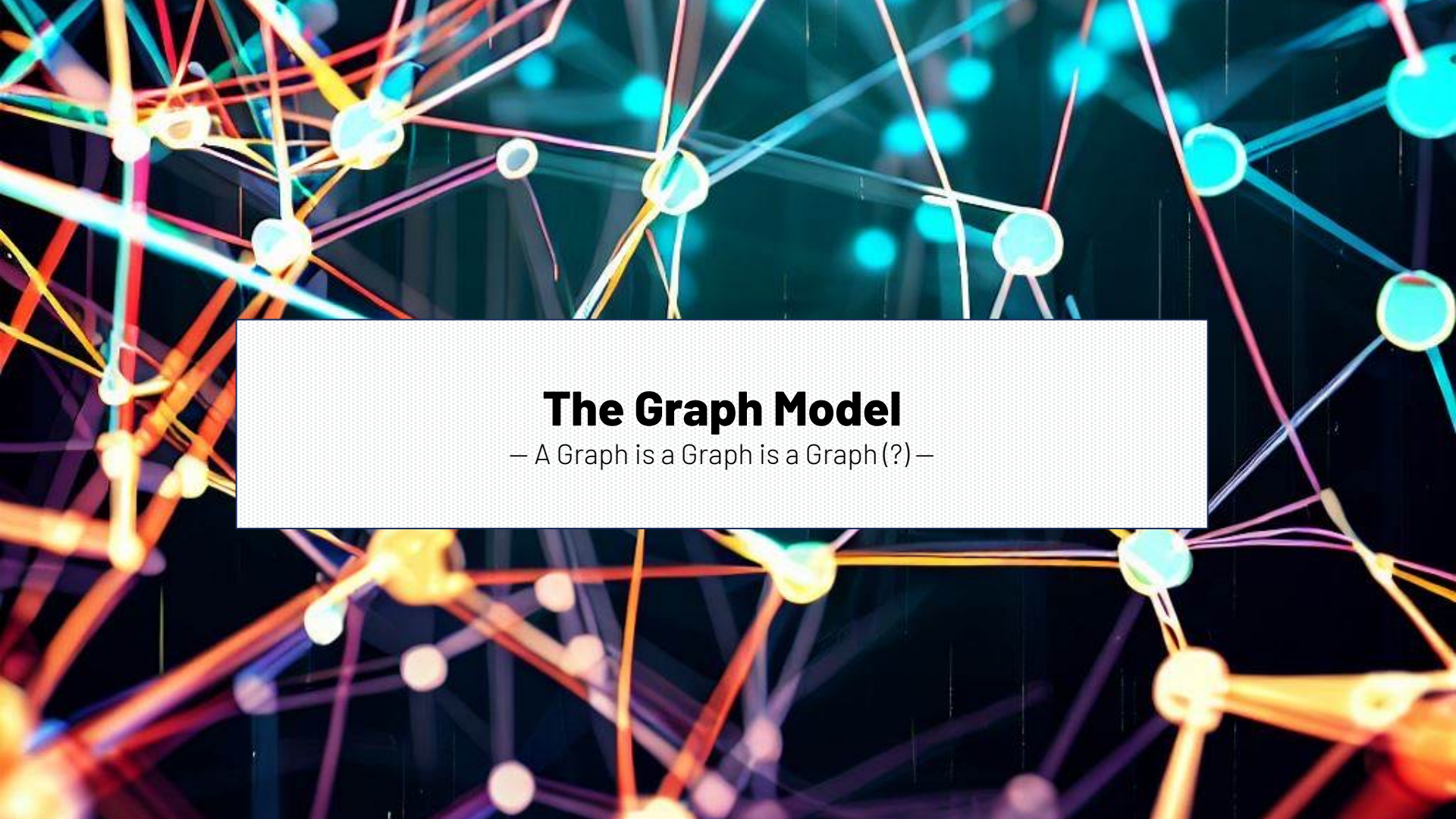
1. **Graphs are Everywhere**
   - The Web-Link structure
   - The Query-Log graph
   - The Social network
   - The Knowledge graph

2. **The Graph Model**
   - Undirected/Directed graphs
   - Labelled/Unlabeled graphs
   - N-partite graphs
   - Heterogeneous Information Networks
   - RDF graphs
   - Property Graph
   - Graph Database vs. Database of Graphs



3. **Representing Graphs**
   - Adjacency matrix
   - Adjacency List
   - Triples & Storage for Triplestore
   - Property graph storage models

4. **Graph Navigation**
   - Breadth-First Search / Depth-First Search
   - Connected Components
   - Paths & Shortest path
   - CYPHER
   - SPARQL
   - Gremlin



The Graph Model

— A Graph is a Graph is a Graph (?) —

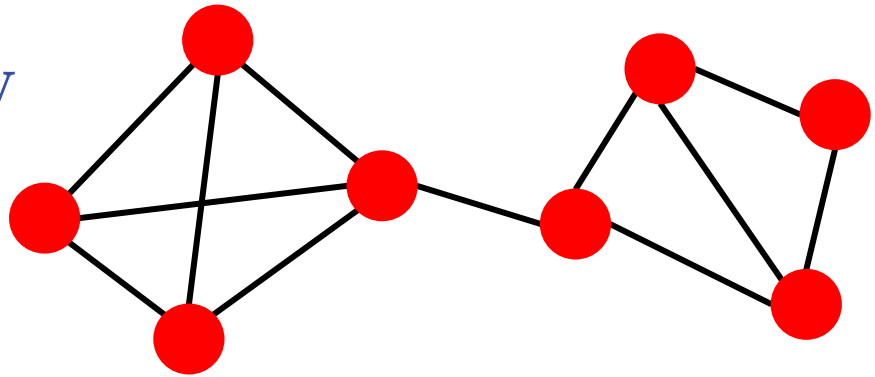
The Graph Model: Formalized

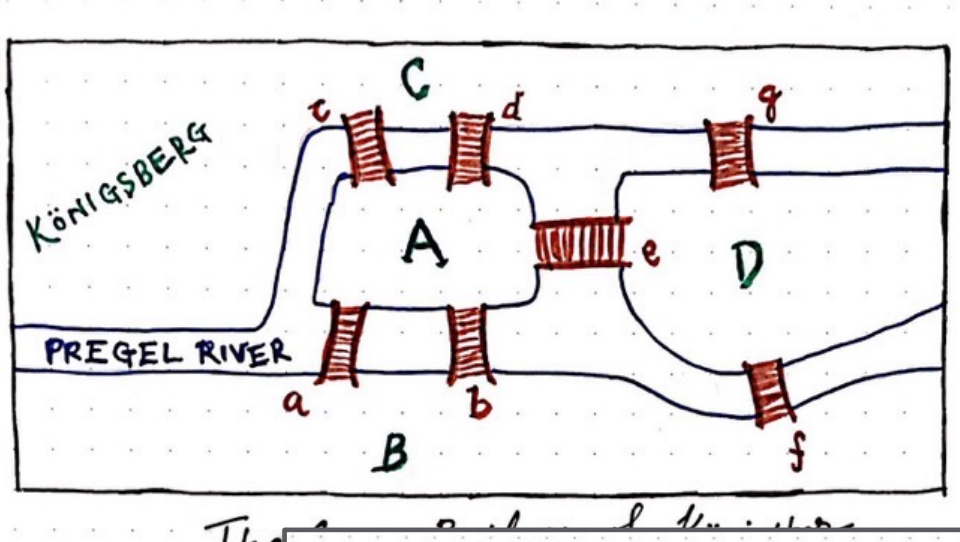
$G: \langle \text{Nodes} ; \text{Edges} \rangle$ a tuple with 2 countable sets

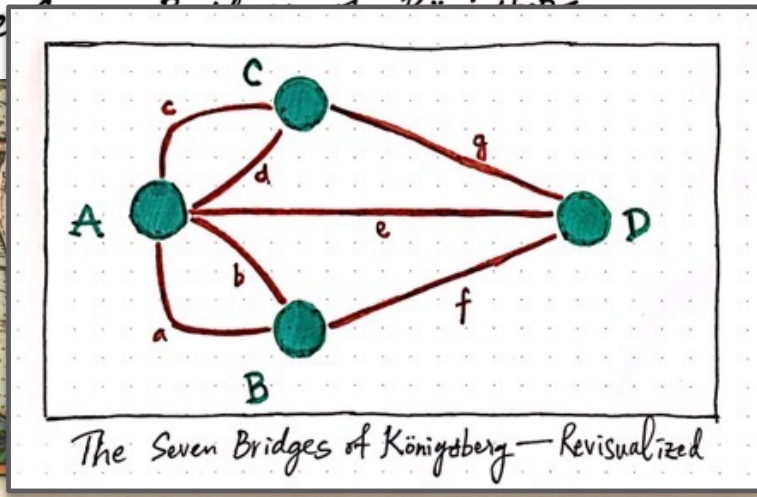
These are "simplistic" formalization, we will see better versions

- **Nodes** N : identified by some ID
- **Edges** E : $E \subseteq N \times N \rightarrow$ identified by pair of nodes

Also called Vertices V







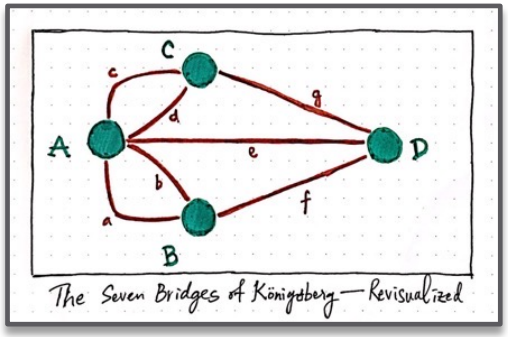
Welcome to **Königsberg**

– Leonhard Euler, 1735

https://medium.com/basecs/k%C3%B6nigsberg-seven-small-bridges-one-giant-graph-problem-2275d1670a12

The Graph Model: Formalized

$G: \langle N ; E \rangle$



- **Nodes** N : identified by some ID
- **Edges** E : $E \subseteq N \times N \rightarrow$ identified by pair of nodes

Options: 1) Undirected $e_{ij}: \langle ID_i, ID_j \rangle \equiv e_{ji}: \langle ID_j, ID_i \rangle$

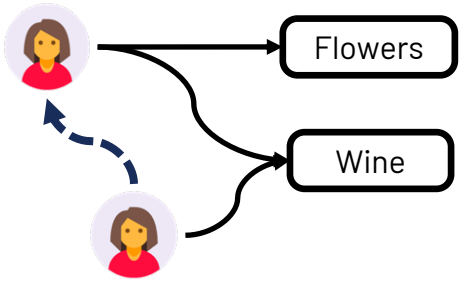
2) Directed $e_{ij}: \langle ID_i, ID_j \rangle \neq e_{ji}: \langle ID_j, ID_i \rangle$

Source Destination

Multigraph: If E can contain duplicates

$E : E \subseteq \mathbb{N} \times N \times N \rightarrow$ we assign IDs to edges

The Graph Model: Formalized



$G: \langle N ; E ; L ; f_{(L)} \rangle$

- **Nodes** N : identified by some ID
- **Edges** E : $E \subseteq N \times N \rightarrow$ identified by pair of nodes
- **Labels** L : special values that describe the type of a node or edge
- **Labeling Function** $f_{(L)} : N \cup E \rightarrow L$

$L: \{$  ;  ;  ;  $\}$

User Follows Product Buys

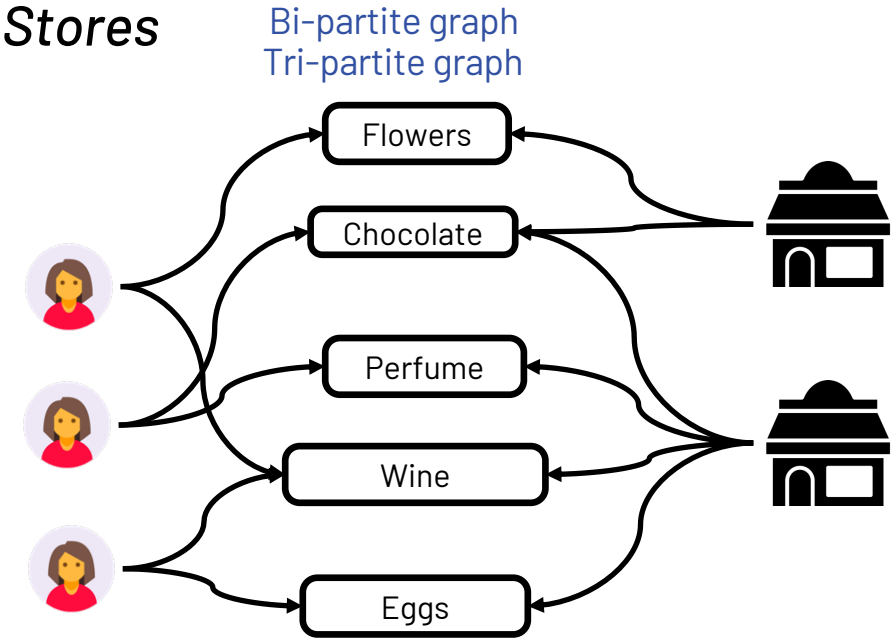
The Graph Model: N-partite graphs

Assume the following Graph

- **Nodes** N : *Users + Products + Stores*
- **Edges** E : *User Buys Product*
  + Store sells product

N-partite graphs:

(a) Nodes are divided in subsets.
(b) Connections exists only from one subset to another, and never within the same subset

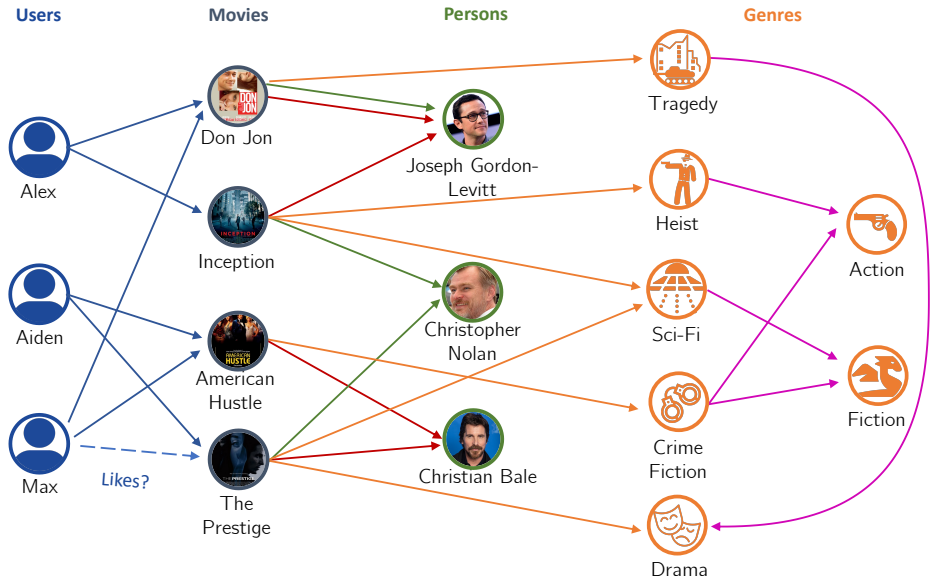


Heterogeneous Information Networks (HIN)

HINs

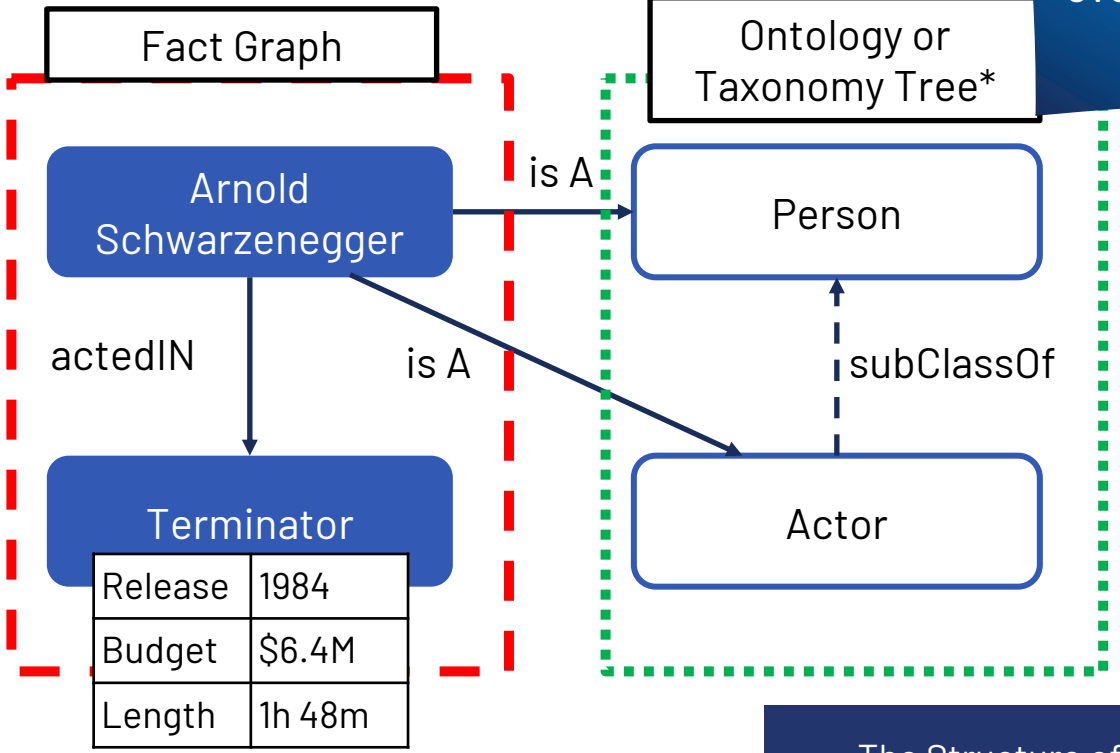
- Nodes and Edges are of different types
- There are limited (<<20) node/edge types
- Usually, no node attributes are considered
- Connections follow predefined "schema", i.e., a high-level structure of the graph

Meta-path: *A meta path is a path defined on a schema, e.g., user–[likes]→movie –[starring]→ actor*



C. Shi, Y. Li, J. Zhang, Y. Sun and P. S. Yu, "A survey of heterogeneous information network analysis," in IEEE Transactions on Knowledge and Data Engineering, 2017, doi: 10.1109/TKDE.2016.2598561.

Knowledge Graphs



There is not a concept of "node labels"
Nodes are described by other nodes

The Structure of the Graph
Is as important as the Data values

RDF & Triples

Representing a KG as a Collection of Facts

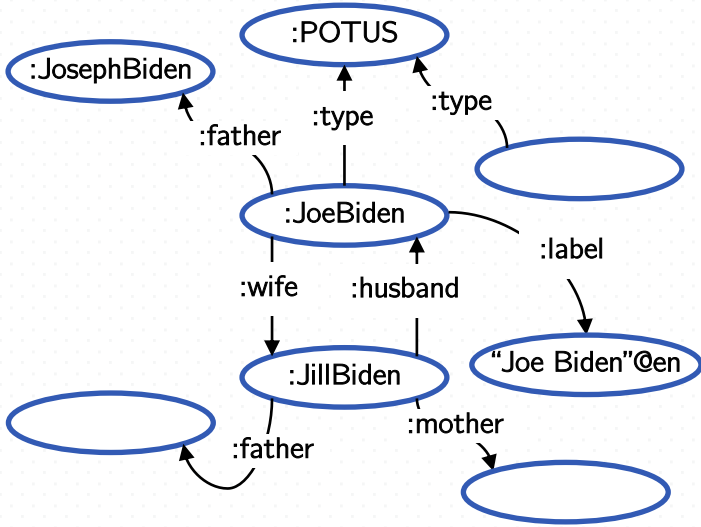
the RDF Model

- **Nodes are either:**
  - Entities (resources identified by IRI)
  - Literals (values as strings, integers, dates)
  - Blank Nodes (special kind of nodes without IRI)

- **Edges are statements**
  (Subject, Predicate, Object)

- Edge types are resources

That's a triple!

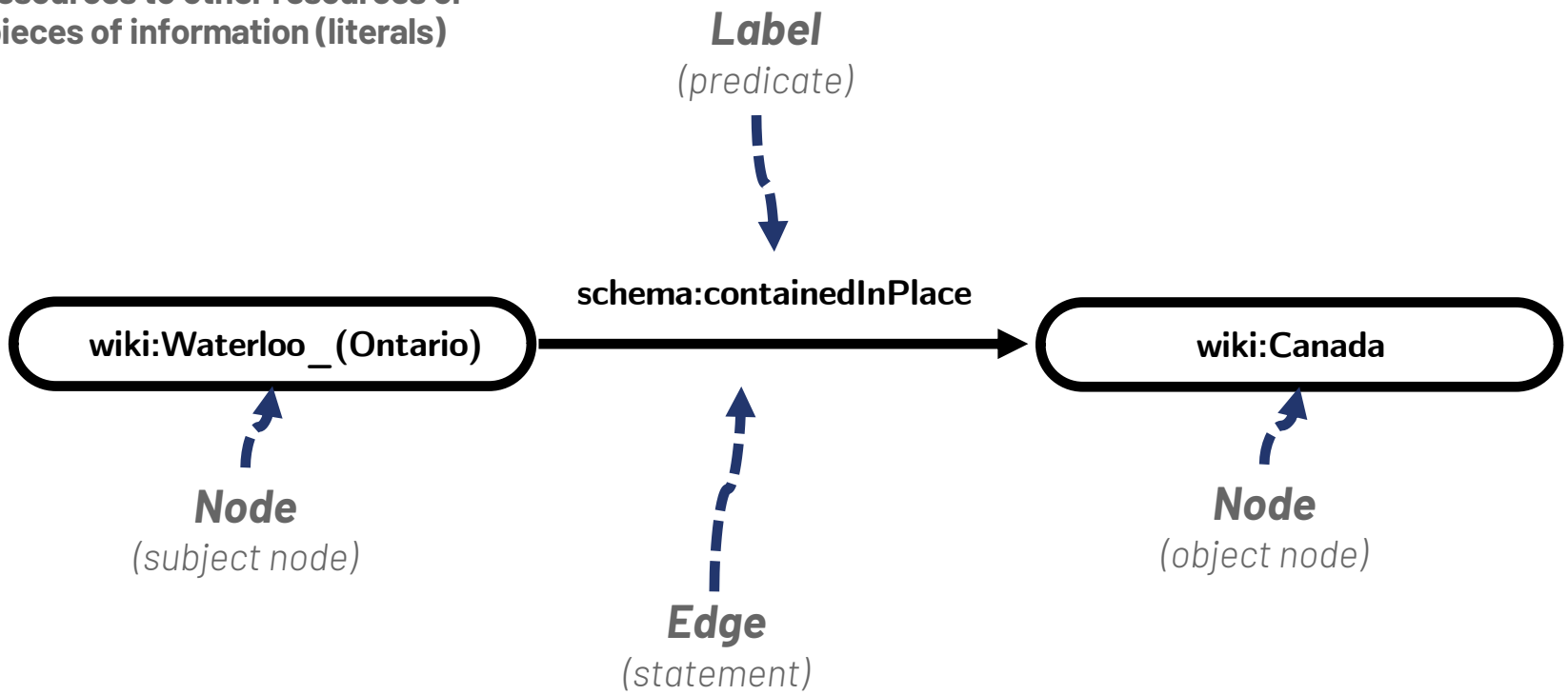



```
@prefix : <http://www.example.kg/> .
:JoeBiden :label "Joe Biden"@en .
:JoeBiden :type  :POTUS .
:JoeBiden :wife :JillBiden .
:JillBiden :husband :JoeBiden .
```

RDF Graphs: Edges

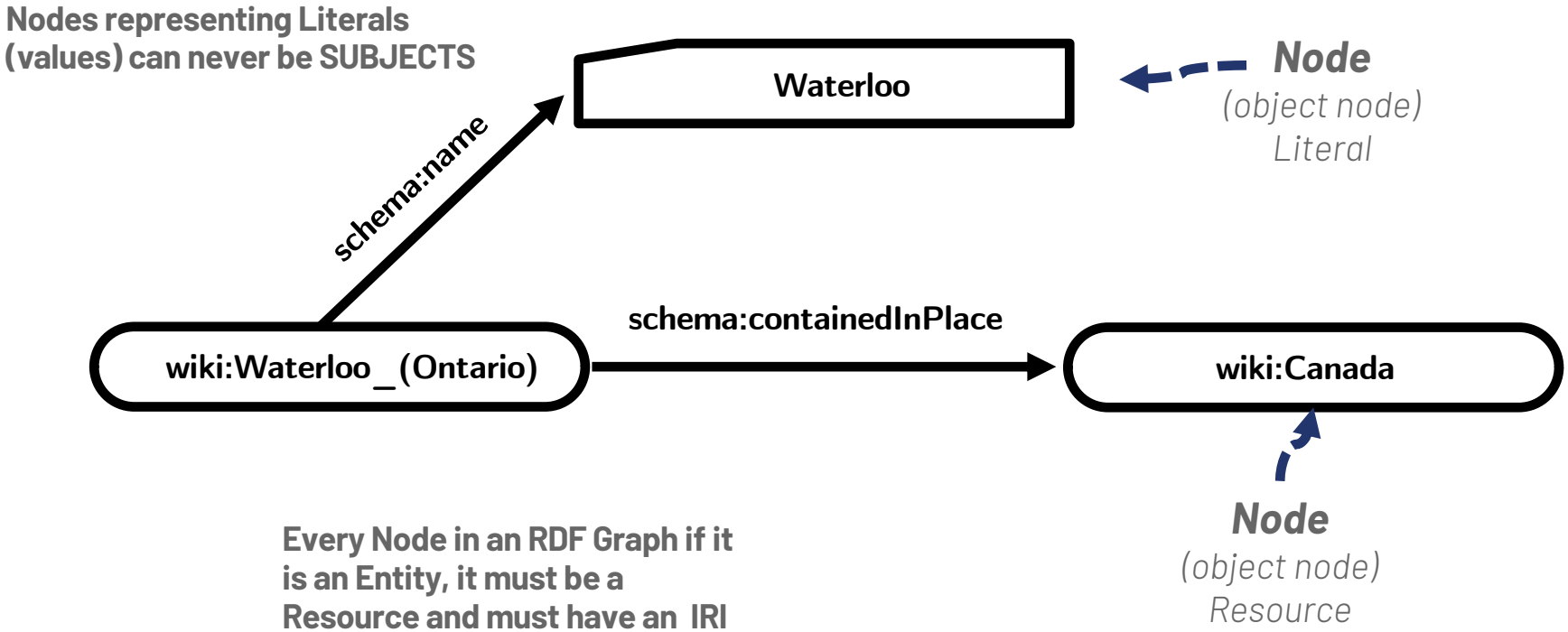
```
wiki: https://en.wikipedia.org/wiki/
schema: https://schema.org/
```

An edge is a statement connecting resources to other resources or pieces of information (literals)



RDF Graphs: Literals

```
wiki: https://en.wikipedia.org/wiki/
schema: https://schema.org/
```



Readable URIs ?

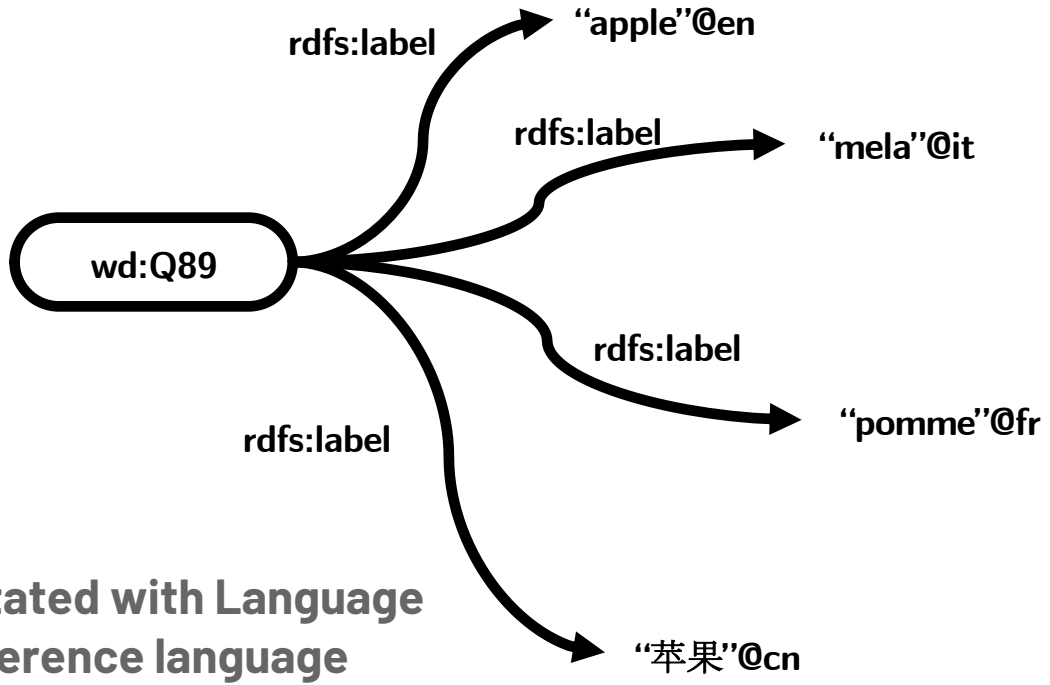
The spelling of a given URI is not supposed to provide clues about what the URI represents.

Still, it is very common as a strategy, for example on DBPedia
https://dbpedia.org/resource/Apple_Inc.
https://dbpedia.org/resource/Apple

Instead on WikiData
https://www.wikidata.org/entity/Q312
https://www.wikidata.org/entity/Q89

Annotated Literals

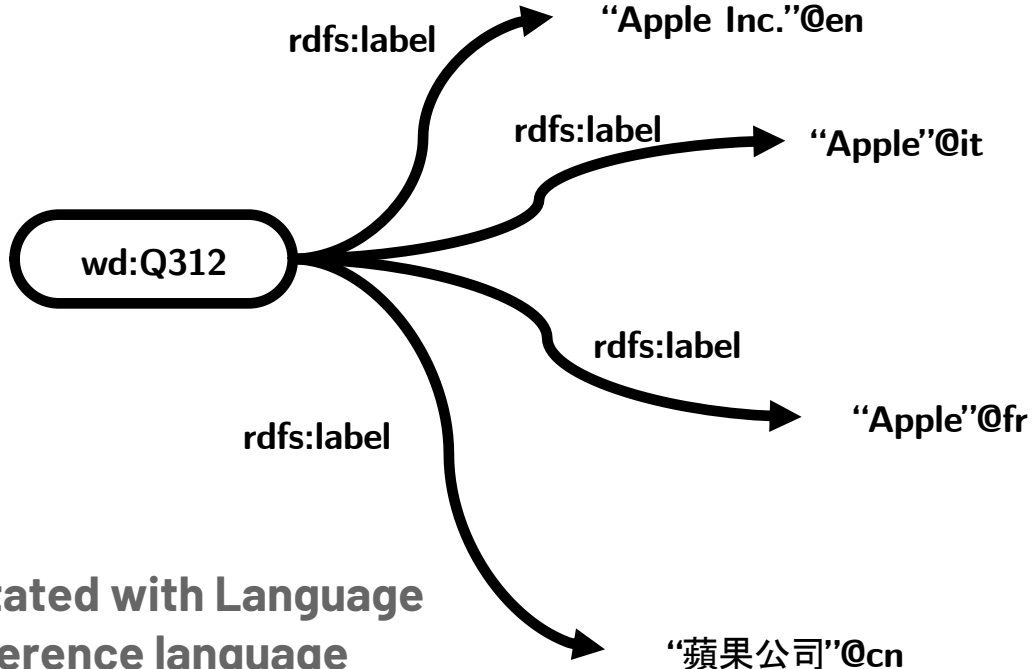
```
wd: https://www.wikidata.org/entity/
rdfs: http://www.w3.org/2000/01/rdf-schema#
```



String Literals can be annotated with Language tags to distinguish their reference language

Annotated Literals (2)

```
wd: https://www.wikidata.org/entity/
rdfs: http://www.w3.org/2000/01/rdf-schema#
```



String Literals can be annotated with Language tags to distinguish their reference language

Literal Data Types

```
wd: https://www.wikidata.org/entity/
rdfs: http://www.w3.org/2000/01/rdf-schema#
```

Typed literals can be expressed via **XML Schema datatypes.**

https://www.w3.org/TR/rdf11-concepts/#section-Datatypes

Namespace for typed literals:

http://www.w3.org/2001/XMLSchema#

Examples:

"Spock"^^<http://www.w3.org/2001/XMLSchema#string>

"1161.00"^^<http://www.w3.org/2001/XMLSchema#float>

"2023-08-02"^^<http://www.w3.org/2001/XMLSchema#date>

Literal Data Types

```
wd: https://www.wikidata.org/entity/
rdfs: http://www.w3.org/2000/01/rdf-schema#
```

Typed literals can be expressed via **XML Schema dat**

https://www.w3.org/TR/rdf11-concepts/#section-Datatypes

Namespace for typed literals:

http://www.w3.org/2001/XMLSchema#

Examples:

"Spock"^^<http://www.w3.org/2001/XMLSchema#string>

"1161.00"^^<http://www.w3.org/2001/XMLSchema#float>

"2023-08-02"^^<http://www.w3.org/2001/XMLSchema#dat

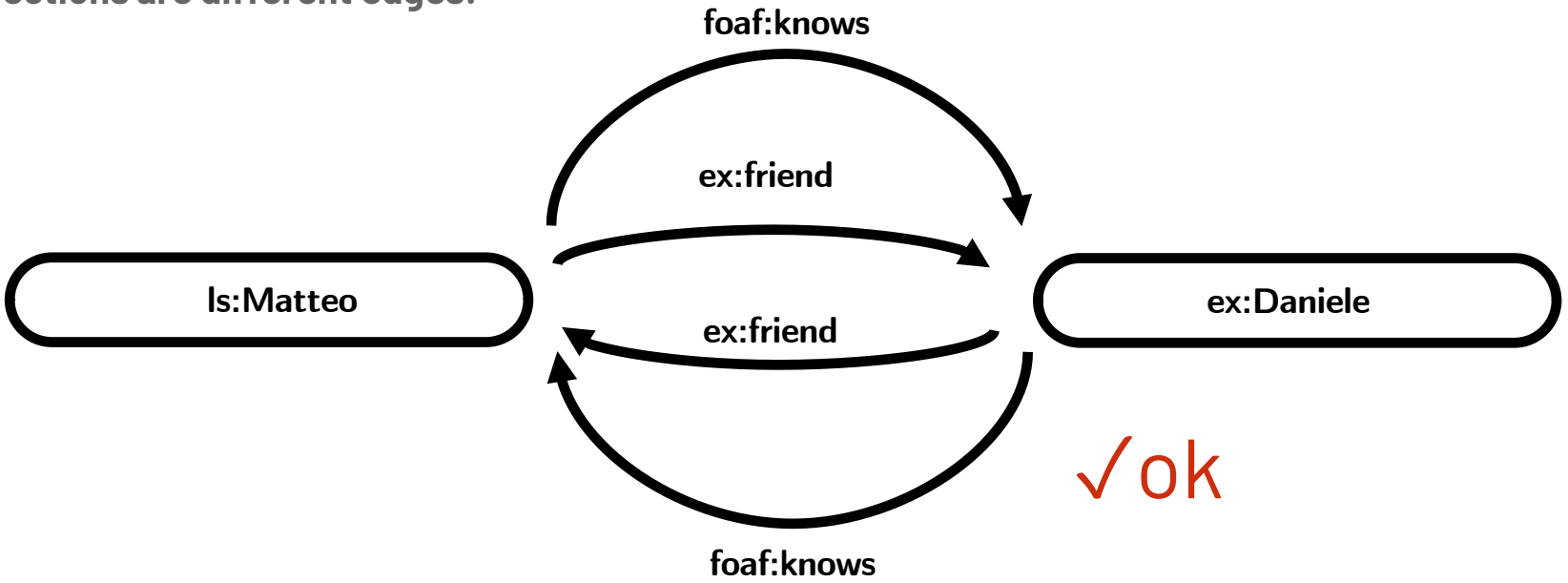
A list of the RDF-compatible XSD types, with short descriptions"

	Datatype	Value space (informative)
Core types	xsd:string	Character strings (but not all Unicode character strings)
	xsd:boolean	true, false
	xsd:decimal	Arbitrary-precision decimal numbers
	xsd:integer	Arbitrary-size integer numbers
IEEE floating-point numbers	xsd:double	64-bit floating point numbers incl. ±Inf, ±0, NaN
	xsd:float	32-bit floating point numbers incl. ±Inf, ±0, NaN
Time and date	xsd:date	Dates (yyyy-mm-dd) with or without timezone
	xsd:time	Times (hh:mm:ss.sss…) with or without timezone
	xsd:dateTime	Date and time with or without timezone
	xsd:dateTimeStamp	Date and time with required timezone
Recurring and partial dates	xsd:gYear	Gregorian calendar year
	xsd:gMonth	Gregorian calendar month
	xsd:gDay	Gregorian calendar day of the month
	xsd:gYearMonth	Gregorian calendar year and month
	xsd:gMonthDay	Gregorian calendar month and day
	xsd:duration	Duration of time
	xsd:yearMonthDuration	Duration of time (months and years only)
	xsd:dayTimeDuration	Duration of time (days, hours, minutes, seconds only)
Limited-range integer numbers	xsd:byte	-128…+127 (8 bit)
	xsd:short	-32768…+32767 (16 bit)
	xsd:int	-2147483648…+2147483647 (32 bit)
	xsd:long	-9223372036854775808…+9223372036854775807 (64 bit)
	xsd:unsignedByte	0…255 (8 bit)
	xsd:unsignedShort	0…65535 (16 bit)
	xsd:unsignedInt	0…4294967295 (32 bit)
	xsd:unsignedLong	0…18446744073709551615 (64 bit)
	xsd:positiveInteger	Integer numbers >0
	xsd:nonNegativeInteger	Integer numbers ≥0
	xsd:negativeInteger	Integer numbers <0
	xsd:nonPositiveInteger	Integer numbers ≤0
Encoded binary data	xsd:hexBinary	Hex-encoded binary data
	xsd:base64Binary	Base64-encoded binary data
Miscellaneous XSD types	xsd:anyURI	Absolute or relative URIs and IRIs
	xsd:language	Language tags per [BCP47]
	xsd:normalizedString	Whitespace-normalized strings
	xsd:token	Tokenized strings
	xsd:NMTOKEN	XML NMTOKENs
	xsd:Name	XML Names
	xsd:NCName	XML NCNames

RDF Graphs: Multiple Edges

```
foaf: http://xmlns.com/foaf/0.1/
ex: http://example.org/
ls: https://lissandrini.com/
```

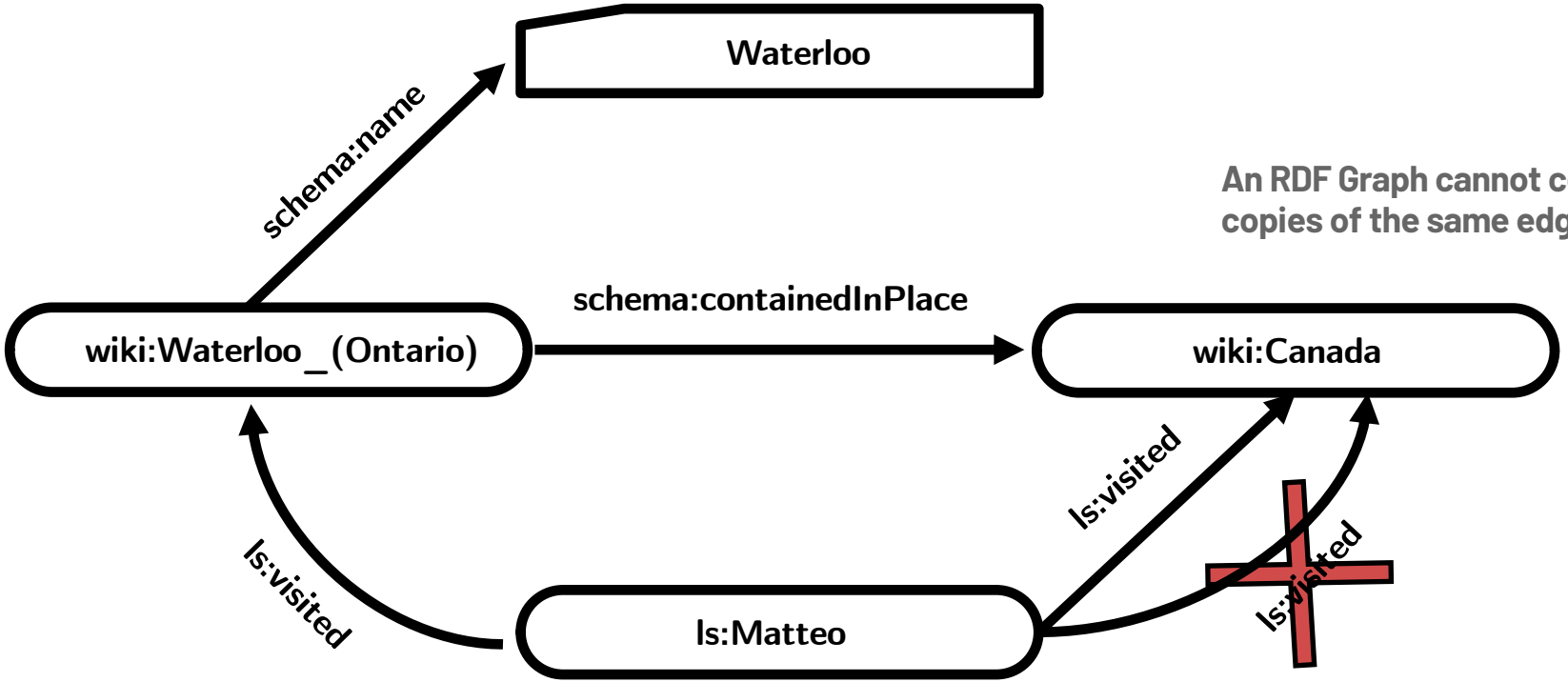
Edges with different predicates (labels) or directions are different edges!



✓ok

RDF Graphs: Limitations

```
wiki: https://en.wikipedia.org/wiki/
schema: https://schema.org/
ls: https://lissandrini.com/
```

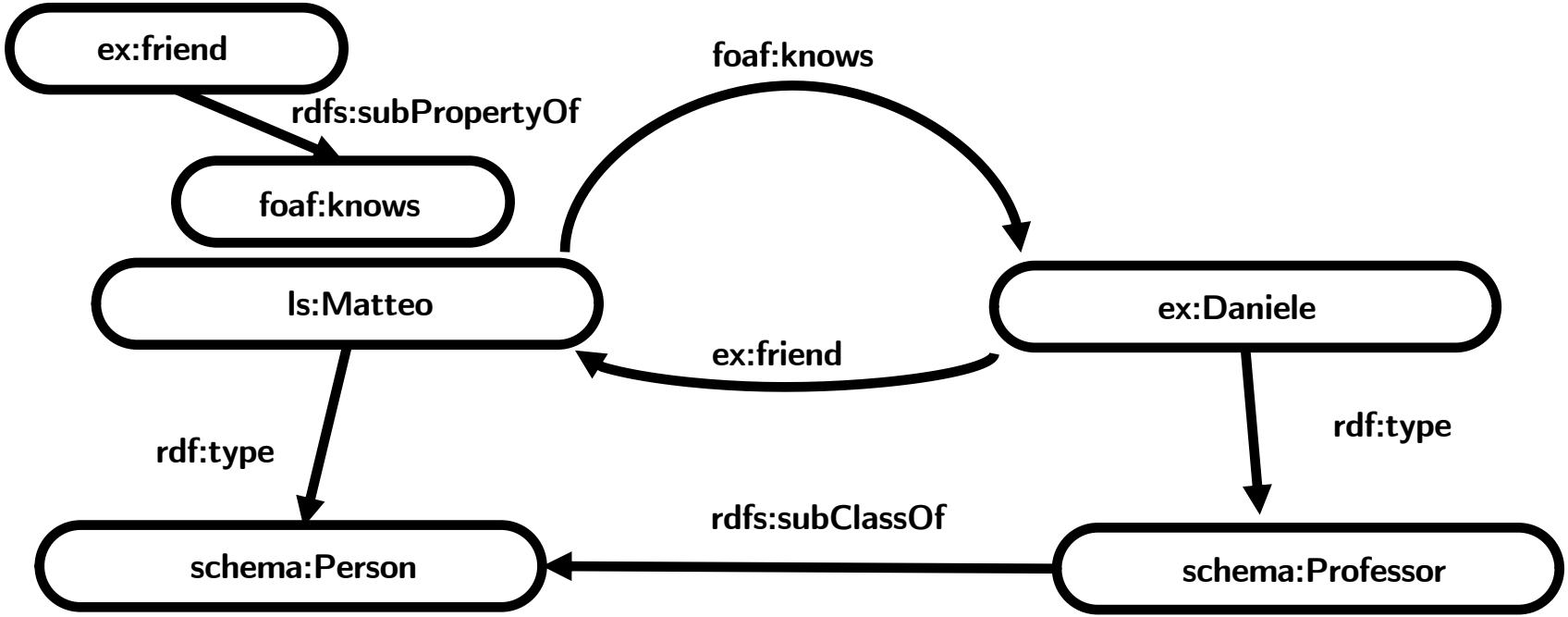


An RDF Graph cannot contain copies of the same edge

RDF Graphs: Types

```
foaf: http://xmlns.com/foaf/0.1/
ex: http://example.org/
ls: https://lissandrini.com/
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
```

Predicates can be nodes too



Entity Types are also nodes

RDF Graph Formal Definition

$\mathcal{I}$: Internationalized Resource Identifiers (IRIs),

$\mathcal{L}$: typed or un-typed literals $\mathcal{L}$ (constants),

$\mathcal{B}$: blank nodes (placeholders for IRIs or literals).

```
@prefix : <http://www.example.kg/> .
:JoeBiden :label "Joe Biden"@en .
:JoeBiden :type  :POTUS .
:JoeBiden :wife :JillBiden .
:JillBiden :husband :JoeBiden .
```

An RDF graph is a labeled directed graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$ with:

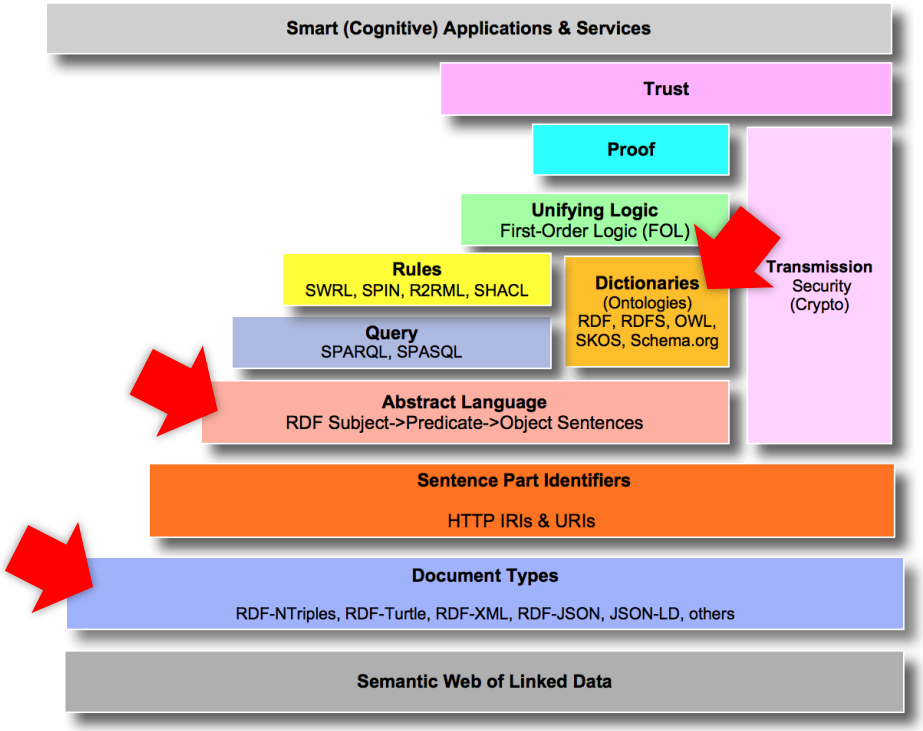
- $\mathcal{N} \subseteq \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ is the set of nodes
  $\mathcal{N}^{>0} = \mathcal{N} \setminus \mathcal{L}$ nodes in $\mathcal{N}$ allowed to have outgoing edges
  (*literals are never subjects!*)
- $\mathcal{E} \subseteq \mathcal{N}^{>0} \times \mathcal{I} \times \mathcal{N}$ is the set of directed edges;
- $\mathcal{P}: \{p \in \mathcal{I} \mid \exists (s, p, o) \in \mathcal{E}\}$ is the set of predicates for G.

It is not properly a multigraph!

W3C RDF Working Group. 2014. Resource description framework. http://www.w3.org/RDF/.

RDF: Data Model & Standard

- RDF allow us to **describe data on the web**
- it also allows to describe **datasets**
- ...but it allows to describe more than that: **Semantics via schemas and ontologies**



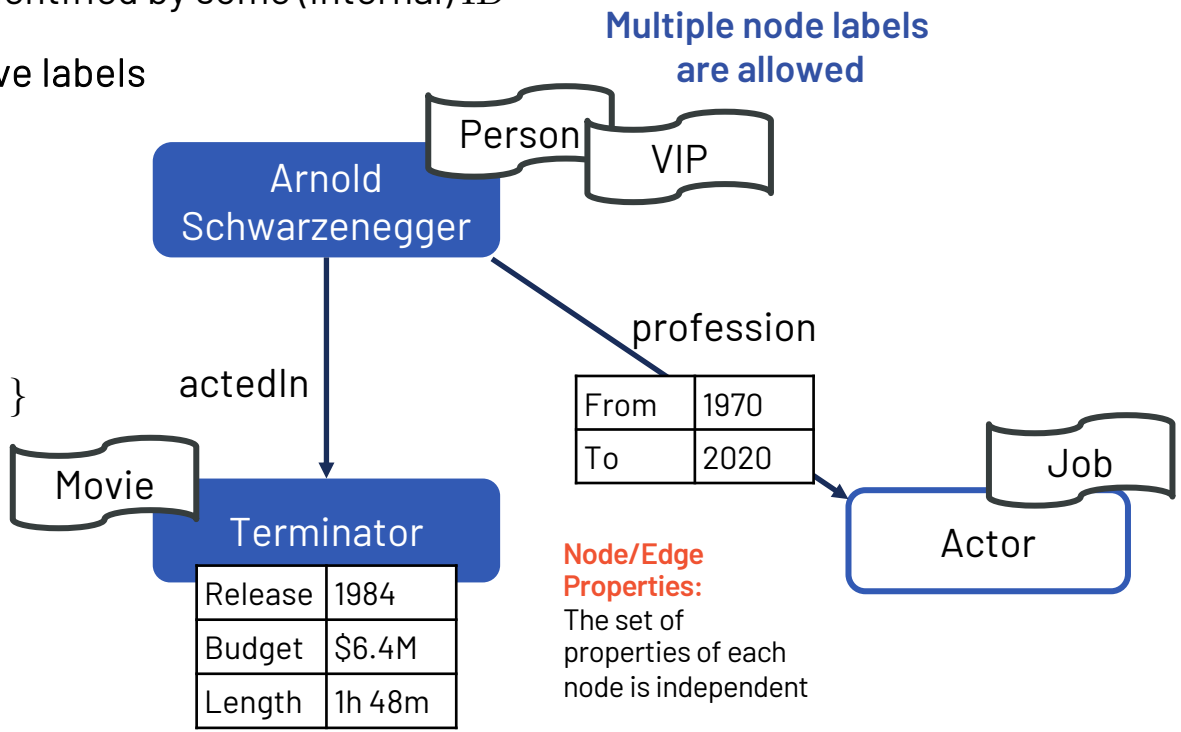
https://medium.com/openlink-software-blog/semantic-web-layer-cake-tweak-explained-6ba5c6ac3fab

The Graph Model Extended: Property Graph

- **Both Nodes N & Edges E** : identified by some (internal) ID
- **Both Nodes N & Edges E** have labels
- Assign to each node & edge

  a dictionary of attributes

  - $\mathrm{ID} \rightarrow \{ <\mathrm{key}_1, \mathrm{value}_1>, \ldots \}$



Formalization of Property Graph

Countable sets $\mathcal{L}$: Labels $\mathcal{K}$: Keys (property names) and $\mathcal{V}$: property values.

A record is a partial function $o: \mathcal{K} \to \mathcal{V}$ mapping keys to values.
($\mathcal{R}$ for the set of all records)

Property Graph: $G = (N, E, \rho, \lambda, \pi)$ where:

It is properly a multigraph!

- N is a finite set of nodes (identified by an ID);
- E is a finite set of edges (identified by an ID) such that $N \cap E = \emptyset$;
- $\rho: E \to (N \times N)$ maps edges to pairs of nodes
- $\lambda: (N \cup E) \to 2^{\mathcal{L}}$ labelling function maps nodes and edges to finite sets of labels (including the empty set)
- $\pi: (N \cup E) \to \mathcal{R}$ property mapping is a function mapping nodes and edges to records.

A Standard for Property Graphs?

```
CREATE GRAPH TYPE   fraudGraphType STRICT {
  (personType: Person {name STRING}),
  (customerType: personType & Customer {id INT32}),
  (creditCardType: CreditCard {num STRING}),
  (transactionType: Transaction {num STRING}),
  (accountType: Account {id INT32}),
  (:customerType)
    -[ownsType: owns]->
  (:accountType),
  (:customerType)
    -[usesType: uses]->
  (:creditCardType),
  (:transactionType)
    -[chargesType: charges {amount DOUBLE}]->
  (:creditCardType),
  (:transactionType)
    -[activityType: deposits|withdraws]->
  (:accountType)
}
```

Fig. 2. PG-Schema of a fraud graph schema.

PG-Schema: Schemas for Property Graphs

RENZO ANGLES, Faculty of Engineering, Universidad de Talca, Chile
IFATI, Lyon 1 University & Liris CNRS, France
MBRAVA, ENSIIE & SAMOVAR - Institut Polytechnique de Paris, France
TCHER, Eindhoven University of Technology, Netherlands
EEN, LDBC, UK
, Birkbeck, University of London, UK
USA
N, University of Edinburgh, UK and RelationalAI & ENS, PSL University, France
AULT, LIGM, Université Gustave Eiffel, CNRS, France
S, University of Bayreuth, Germany
, University of Warsaw, Poland
TIKOW, Neo4j, Germany
OVIĆ, Free University of Bozen-Bolzano, Italy
MIDT, Amazon Web Services, USA
A, data.world, USA
ORKO, RelationalAI, USA and Univ. Lille, CNRS, UMR 9189 CRIStAL, France
ASZUK, University of Bialystok, Poland
T, Neo4j, Germany
OČ, University of Zagreb, Croatia and PUC Chile, Chile
gerGraph, USA
VIĆ, Integral Data Solutions, UK

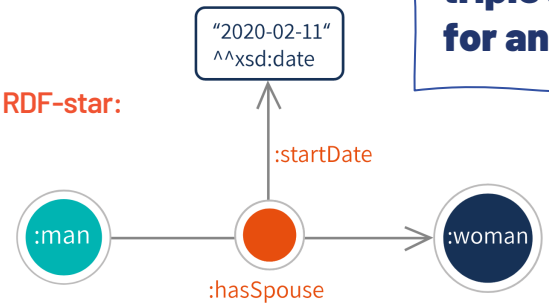
e reached a high level of maturity, witnessed by multiple robust

Edge Property in RDF: RDF-Star

How can we represent edge property in RDF?

Use an entire triple as subject for another triple

RDF-star:



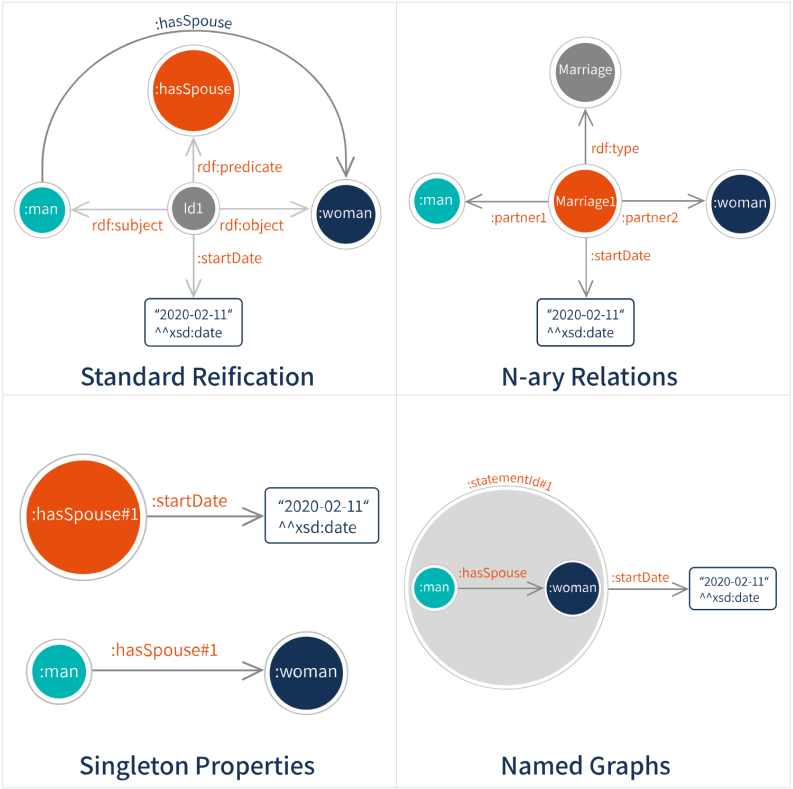
(S, P, O)

:man :**hasSpouse** :woman .

<<:man :hasSpouse :woman>> **:startDate** "2020-02-11"^^xsd:date .

(S, P, O)

Classical RDF:
Only nodes can be subjects of triples



https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-star/
https://w3c.github.io/rdf-star/cg-spec/editors_draft.html
https://www.bobdc.com/blog/rdf-and-sparql/

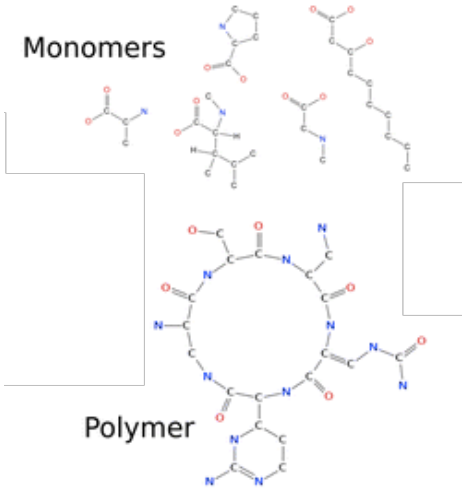
Types of Graph Databases

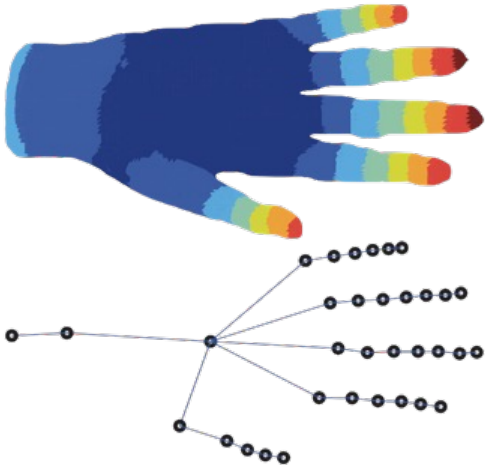
Single large graphs

- The web
- Social network
- Knowledge Graph

Distinct Graphs (a.k.a. database of graphs)

- Protein-Protein interactions
- Products co-purchase
- 3D Objects





Graph-Databases / Databases of Graphs

There is often confusion in how this terminology is used. It depends on the context, pay attention!

Databases vs. DBMS

A collection of related pieces of data vs. Database Management Systems (software)

When using the graph mode

Define:

- Nodes and Edge
- Directed / Undirected
- Labelled / Unlabelled
- Properties / Literal values
- Multiplicity of Edges
- Graph Database / Database of Graphs

Ask yourself: in which way the structure of the graph encodes information?

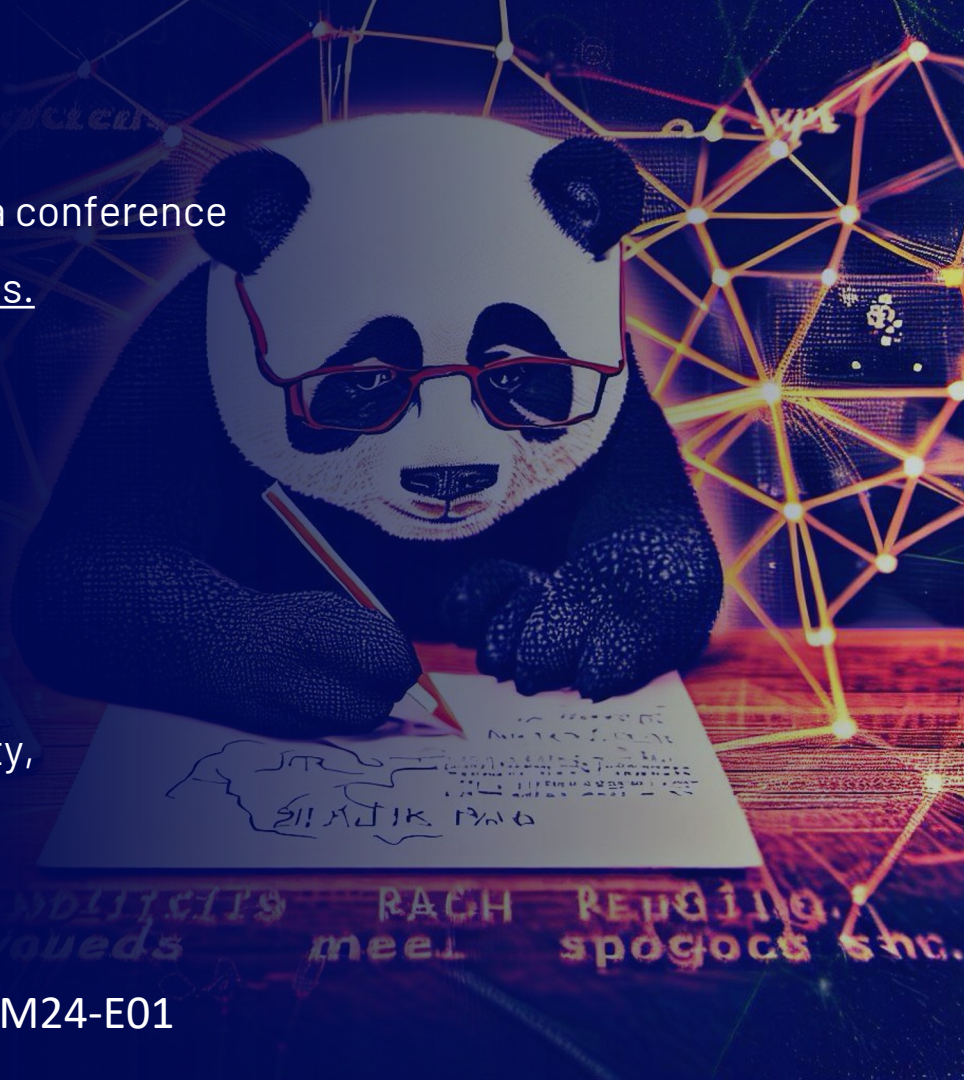
Example: Connectivity? Reachability? Paths? Taxonomy?

Let's model this as a graph (1)

You are helping organize a networking event at a conference and want to model the data about its participants.

You have the citation network of all the people at the conference.
There are **papers**, **authors**, and **universities**.

You know which author **works in** which university, which author **wrote** which paper, and which paper **cited** which paper.



https://bit.ly/ACM24-E01

Let's model this as a graph (2)

You are tracking users on a complex website.

They **visit many pages** of the website to complete their work, when done they close the website.

For each user you know on which **page they start**, what **action** they take, on which **page they end**.

Then from that page they can take another action and **go to another page.** They can never go to random pages.

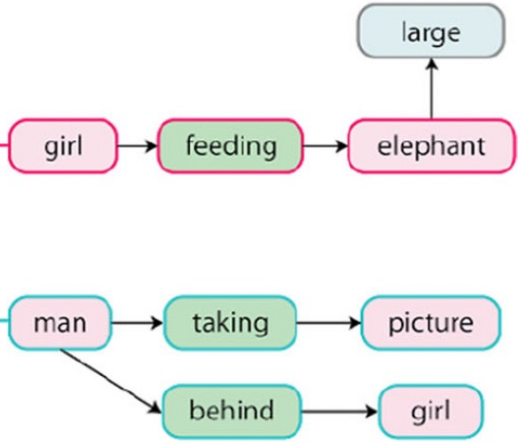
They **can also go back** to the previous page and do an additional actions and thus may end up in a different page from there.

https://bit.ly/ACM24-E02

Scene Graphs: What model is this? Can we do better?

https://bit.ly/ACM24-E03

Flip flops on the ground
Elephant that could carry people
Leaves on the ground
Huts on a hillside
A bag
A bush next to a river.
a woman wearing a brown shirt
Girl feeding large elephant
Woman wearing a purple dress
Tree near the water
a man wearing a hat
A handle of bananas.
a man taking a picture behind girl
Glasses on the hair.
blue flip flop sandals
small houses on the hillside
the nearby river
Elephant with carrier on it's back
Two people near elephants
A blue shirt



Krishna, R., Zhu, Y., Groth, O. et al. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. Int J Comput Vis 123, 32–73 (2017). https://doi.org/10.1007/s11263-016-0981-7

Knowledge Graph or not?

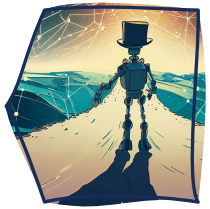
When is it a KG, an Ontology, a Heterogeneous Graph, or a Knowledge Base?

- a **KG is always a heterogenous graph**, but a heterogeneous graph is not always a KG
- a **KG assigns at least an identifier** to each relevant concept/entity, and then:
  - **attaches data** to each/most identifiers
  - **provides definition of types and relationships** that are modelled as graph objects and not as meta-data annotations
- One can model **a KG both as RDF graph or a PG**, yet, PG do not have a standard way to model ontologies and ontological rules
- An **Ontology provides formal naming and definitions** of the categories, properties, and relations between the concepts in a KG, this **allows reasoning and inference**
- While a KG uses binary relations (edges between two nodes), **a Knowledge Base can use n-ary** relations , e.g., president(Obama, USA, 2009,2017)

Outline

1. **Graphs are Everywhere**
   - The Web-Link structure
   - The Query-Log graph
   - The Social network
   - The Knowledge graph

2. **The Graph Model**
   - Undirected/Directed graphs
   - Labelled/Unlabeled graphs
   - N-partite graphs
   - Heterogeneous Information Networks
   - RDF graphs
   - Property Graph
   - Graph Database vs. Database of Graphs
   - Knowledge Graphs



3. **Representing Graphs**
   - Adjacency matrix
   - Adjacency List
   - Triples & Storage for Triplestore
   - Property graph storage models

4. **Graph Navigation**
   - Breadth-First Search / Depth-First Search
   - Connected Components
   - Paths & Shortest path
   - CYPHER
   - SPARQL
   - Gremlin

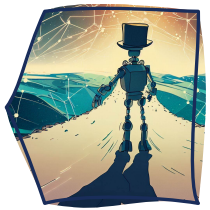
Outline

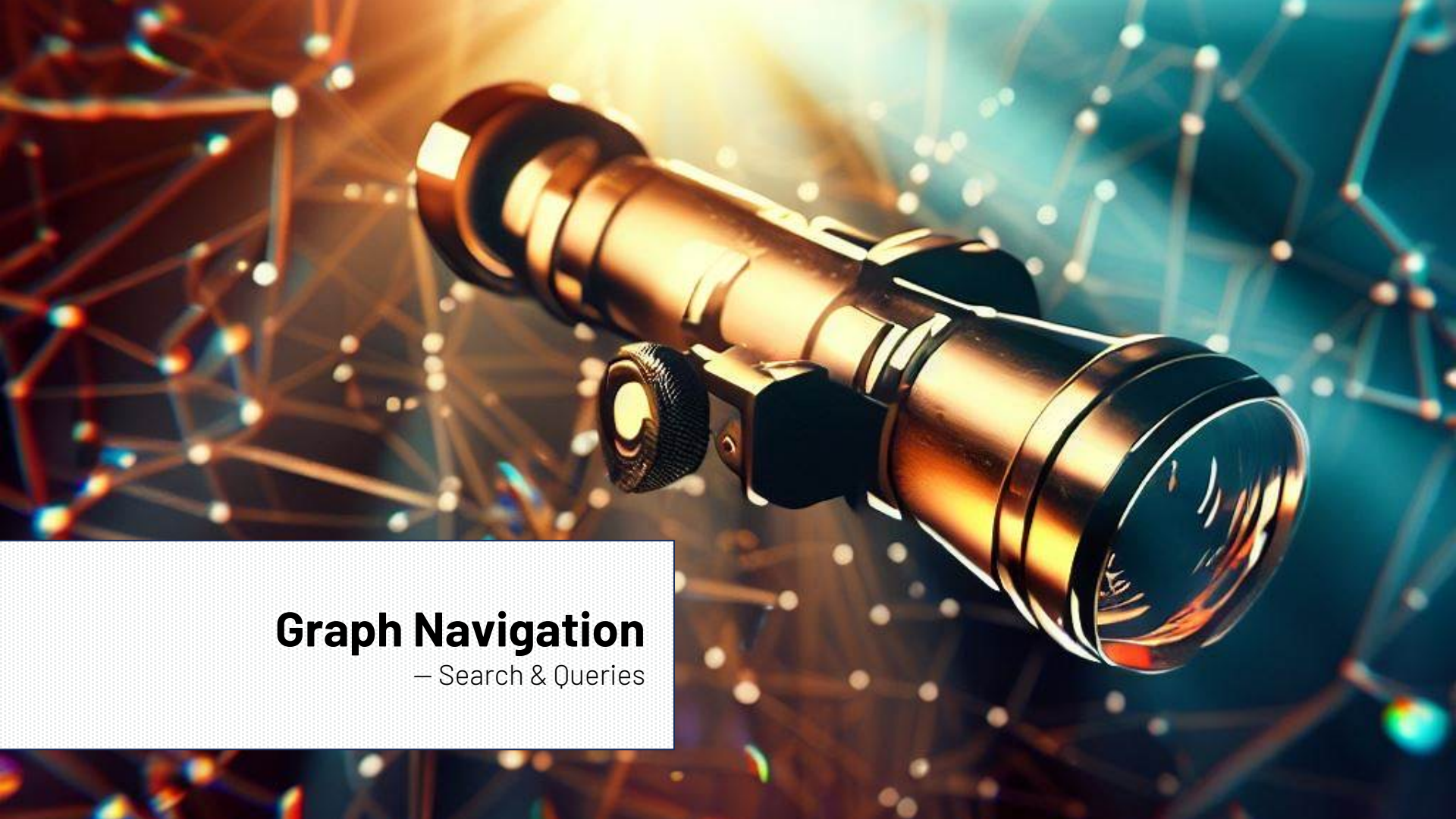
1. **Graphs are Everywhere**
   - The Web-Link structure
   - The Query-Log graph
   - The Social network
   - The Knowledge graph

2. **The Graph Model**
   - Undirected/Directed graphs
   - Labelled/Unlabelled graphs
   - N-partite graphs
   - Heterogeneous Information Networks
   - RDF graphs
   - Property Graph
   - Graph Database vs. Database of Graphs
   - Knowledge Graphs

3. **Representing Graphs**
   - Adjacency matrix
   - Adjacency List
   - Triples & Storage for Triplestore
   - Property graph storage models

4. **Graph Navigation**
   - Breadth-First Search / Depth-First Search
   - Connected Components
   - Paths & Shortest path
   - CYPHER
   - SPARQL
   - Gremlin





Graph Navigation

— Search & Queries

Graph Navigation

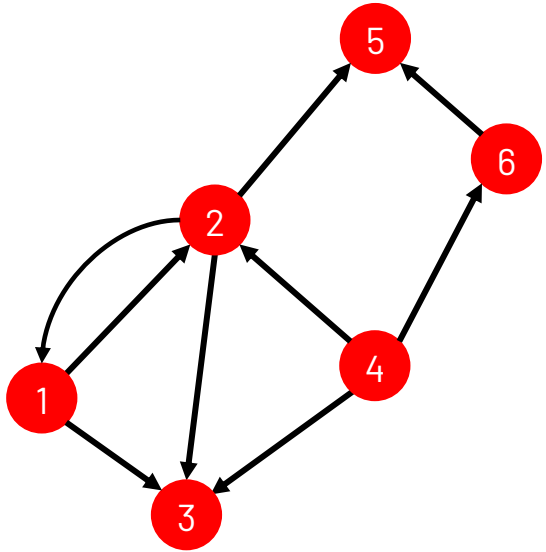
What operations we do on a graph? Given a node obtain:

- **Neighbors:** obtain the list of all nodes connected to it
- **Degree:** number of nodes connected (when undirected)
  - When directed: In-degree / out-degree
- **Graph Traversal**
  - Start from a node, obtain the list of all reachable nodes

$$\text{Neighbors}(1)\text{: } \{2, 3\}$$

$$\text{InDegree}(2) = 2 \quad \text{OutDegree}(2) = 3$$

$$\text{Reachable}(4) = \{2, 3, 1, 6, 5\} \quad \text{Reachable}(1) = \{2, 3, 5\}$$



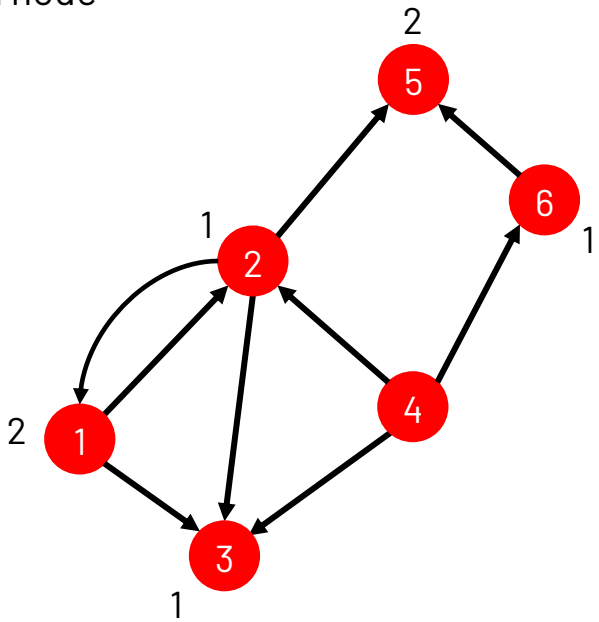
Graph Traversal: BFS vs. DFS

- **Graph Traversal:** start from a node, obtain the list of all reachable nodes, in which order?

- **Breadth-First Search (BFS):** visit first all the neighbors of a node before visiting the other

$$\mathrm{BFS}(4) = [2, 3, 6, 1, 5]$$

```
Q = queue()
Q.enqueue(startNode)
mark startNode as visited
while Q is not empty do
    v := Q.dequeue()
    // do something with v here //
    for all w in neighbors(v) do
       if w is not visited then
            mark w as visited
         Q.enqueue(w)
```



Graph Traversal: BFS vs. DFS

Graph Traversal from a single node is used to find all reachable nodes

- **Graph Traversal:** start from a node, obtain the list of all reachable nodes, in which order?

- **Breadth-First Search (BFS):** visit first all the neighbors of a node before visiting the other

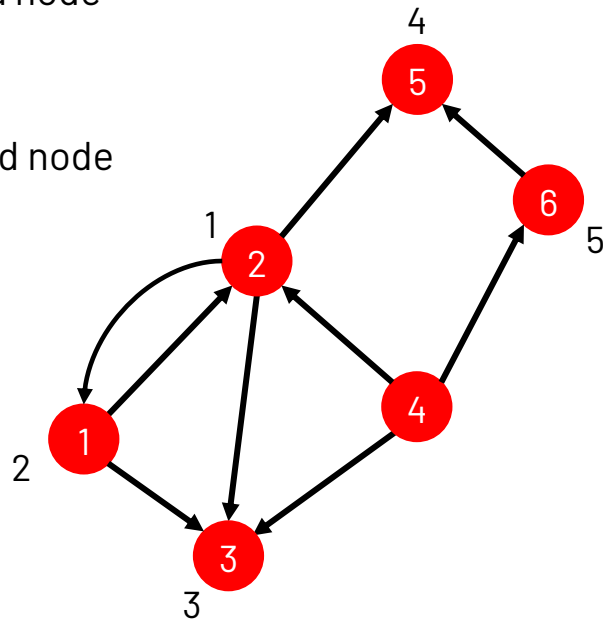
  $\mathrm{BFS}(4) = [2, 3, 6, 1, 5]$

- **Depth-First Search (DFS):** visit a neighbor of the last visited node

  $\mathrm{DFS}(4) = [2, 1, 3, 5, 6]$

```
Q = queue()
Q.enqueue(startNode)
mark startNode as visited
while Q is not empty do
    v := Q.dequeue()
    // do something with v here //
    for all w in neighbors(v) do
       if w is not visited then
            mark w as visited
         Q.enqueue(w)
```

```
Q = stack()
Q.push(startNode)
mark startNode as visited
while Q is not empty do
    v := Q.pop()
    // do something with v here //
    for all w in neighbors(v) do
       if w is not visited then
            mark w as visited
         Q.push(w)
```



Reachability: Connected components

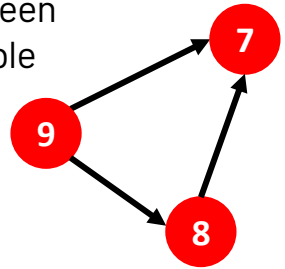
USE BFS :
1) Start from a node;
2) Obtain all reachable nodes and mark them;
3) Increment CC counter
4)Take next node not already marked, and start again

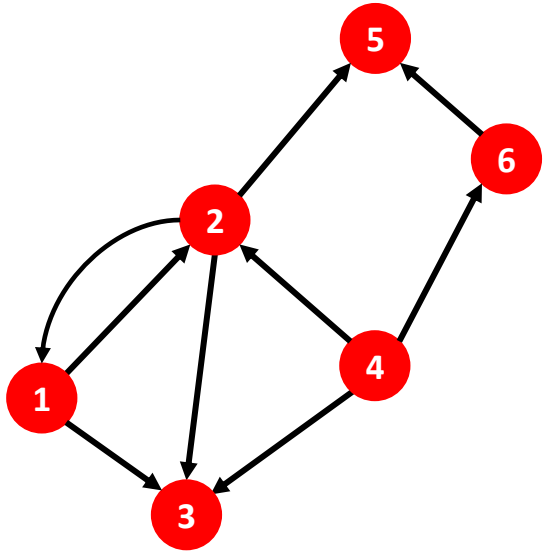
- **A connected component** is a portion of the graph where each node can reach all other nodes: pairwise reachable.

In a directed graph we can have connected components, but if we follow directions, then it may happen that we cannot reach all nodes.

- **A strongly connected component** is a portion of a directed graph where there is a directed path between any two nodes. All nodes are pairwise reachable when following directions.

- **A weakly connected component** is a portion of a directed graph where there is an **undirected** path between any two nodes. All nodes are pairwise reachable when **ignoring** directions.



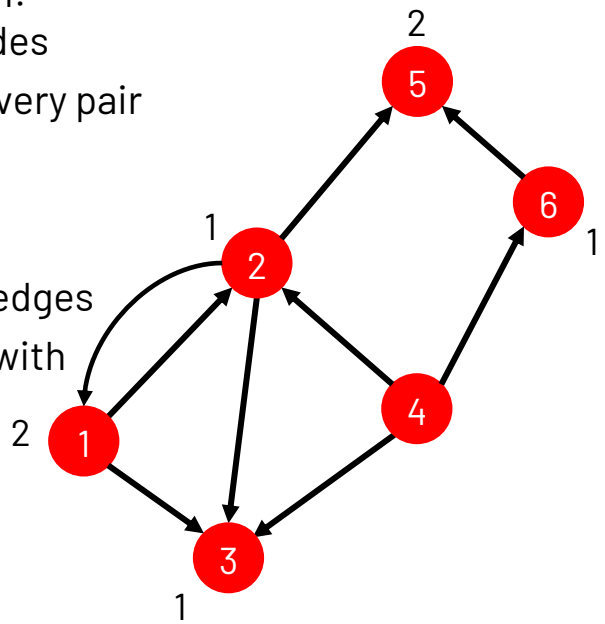


Graph Traversal: Shortest Path

- **Find the "quickest" way to reach nodes** *(Dijkstra's algorithm)*:
  - **Single source:** Given 1 source node find the "quickest" way to reach all other nodes
  - **Single Source-Destination** (pair of nodes) shortest path: find the "quickest" way – if exists – between the two nodes
  - **All-pairs shortest path:** find shortest paths between every pair of vertices in the entire graph
- **Definition of "quickest":**
  - All **edges cost the same** → find the smaller number of edges
  - Edges have **different cost** → weighted path, find path with minimum sum of edge weights

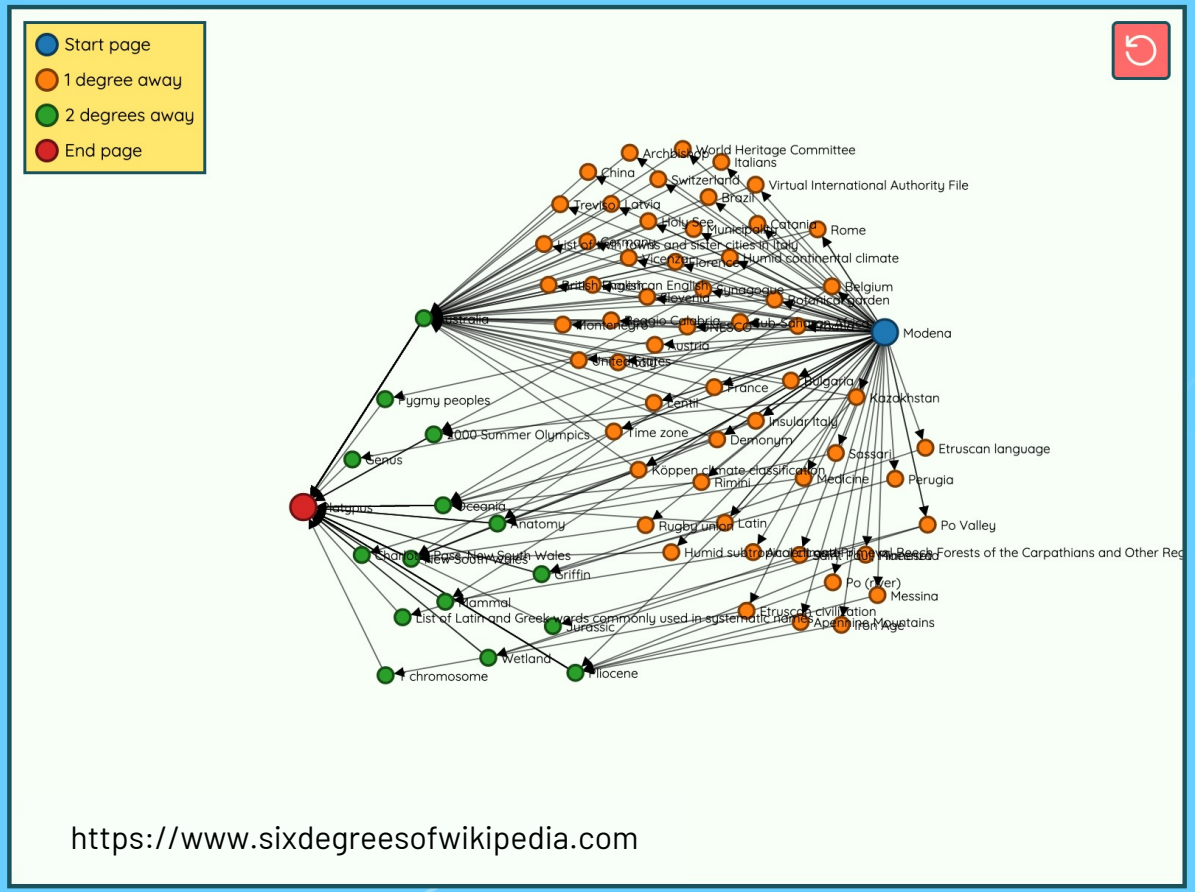
BFS & SHORTEST PATH

If all edges cost equal, BFS can compute the shortest path from one node to all other nodes



https://en.wikipedia.org/wiki/Shortest_path_problem#Single-source_shortest_paths

Found **76 paths** with **3 degrees** of separation from **Modena** to **Platypus** in **1.81 seconds!**



Graph Queries: PGs and Triples

Different Data Models have different Query Paradigms

- **Property Graphs (PGs): everything is an "object" that can contain data**
  - Queries can retrieve: (a) nodes, (b) edges, (c) paths
  - Query language: CYPHER or GQL (gqlstandards.org) or GREMLIN

- **RDF (KGs): everything is a "triple" (a statement)**
  - Queries can only retrieve triples (matching paths / patterns)
  - Query language: SPARQL

OUTPUT:
A Graph query (PG or RDF) does not always (almost never) return a graph, usually they return tuples of variable assignments

Graph Queries: When Inserting Data

How to add information to a PG graph

- Create **new node with a label** (with properties P)
- Add **edge from v_1 to v_2 with a label** (plus some properties P)
- Add **property { Name : Value }** to node v or to edge e

This means we can:

- Find nodes/edges with property P { Name : Value }
- Find edges with a specific label

TYPICAL OF PROPERTY GRAPHS (PGs)

Nodes can be created in isolation
and then annotated & enriched later

How to add information to a RDF graph:

- insert a (s,p,o) triple

Graph Queries: Focus on Edges (PGs)

Direct Neighbors

- Find nodes **directly connected** (all connected edges)
- Find only certain **edges based on label** (all my friends)

Degree based Search

- **Find important nodes: count connected edges**

  then filter to get High Node Degree

- **Find spider traps: Nodes with only Inbound connection**

  count and remove nodes with count > 0 (alternative: use NOT EXISTS)

Graph Queries: Connections & Structures

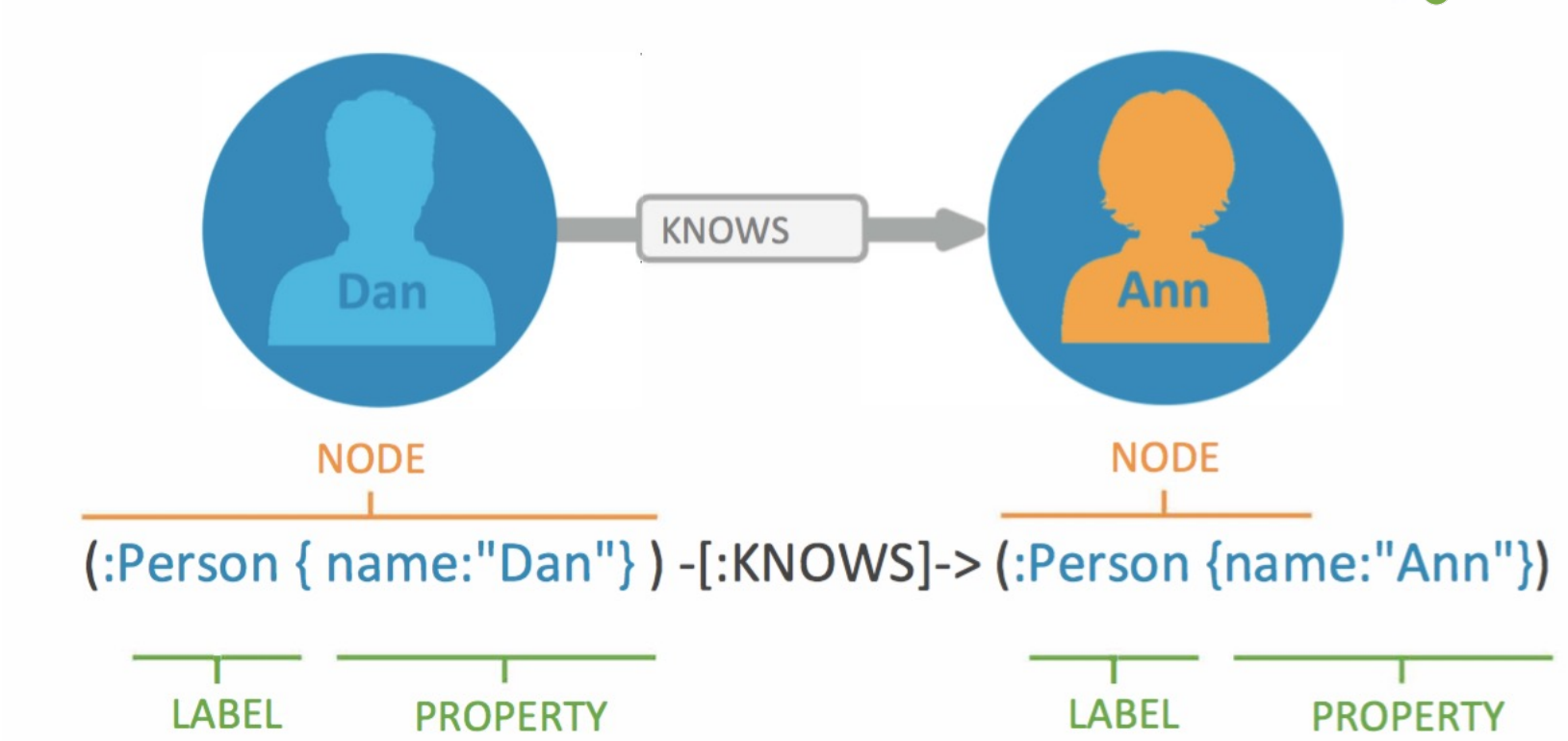
Complex Queries and Shortest Paths

- Find all **nodes reachable in K** or less steps (BFS)
- Find a list **of connections (paths) between two nodes**
- Find **label-constrained paths** (*The Genre of the Book written by My Friend*)
- **More complex structures/patterns** (typical of RDF):
  - e.g., People that have friends in two different countries, and live in a third country

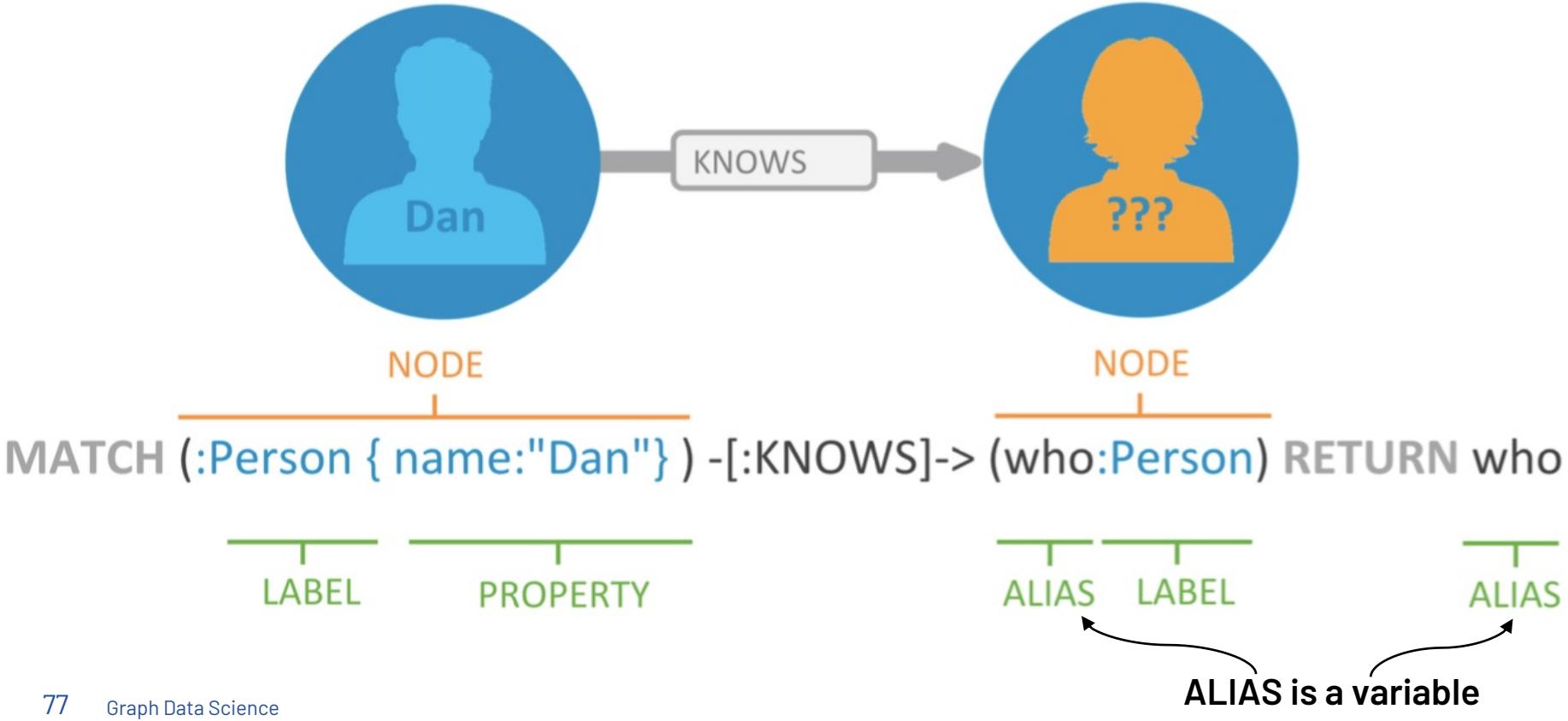
PGs optimize for FAST TRAVERSAL
(Typed) Pointers to Neighbors

GRAPH HOMO/ISOMORPHISM
we will see it later

Graph Example: PG



Graph Query Example



Graph Query Example (II)

```
//Find all the movies Tom Hanks directed and order by latest movie
MATCH (:Person {name:"Tom Hanks"})-[:DIRECTED]->(m:Movie)
RETURN m.title, m.released ORDER BY m.released DESC;
```

```
//Find all of the co-actors Tom Hanks have worked with
MATCH (th:Person{name:"Tom Hanks"})-->(:Movie)<-[:ACTED_IN]-(oth:Person)
WHERE th <> oth
RETURN oth.name;
```

```
MATCH pattern
WHERE predicate
ORDER BY expression
SKIP ... LIMIT ...
RETURN expression AS alias
```

Path pattern variations:

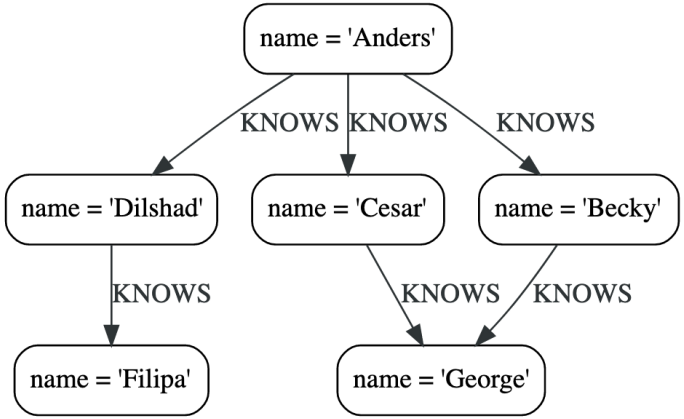
```
(n1)-[r1]->(n2)<-[r2]-(n3)
(n1)-[:KNOWS*]->(n2)
```

https://neo4j.com/developer/download-materials/

Graph Query Example (III)

```
MATCH (node1:Person)-[:KNOWS]->(node2:Person)
      (node1)-[:LIVES_IN]->(node3:City)
      (node2)-[:LIVES_IN]->(node3)
 WHERE node1.age = 30
RETURN node1.name, node2.name, node2.age
```

Graph Query Example (IV) : Paths



In Neo4j, all relationships have a direction. However, you can have the notion of undirected relationships at query time.

```
MATCH (me)-[:KNOWS*1..2]-(other)
WHERE me.name = 'Filipa'
RETURN other.name
```

Results:
"Dilshad"
"Anders"

```
MATCH (me)-[:KNOWS*1..2]->(other)
WHERE me.name = 'Anders'
      AND other.name > 'F''
RETURN other.name
```

Results:
"George"

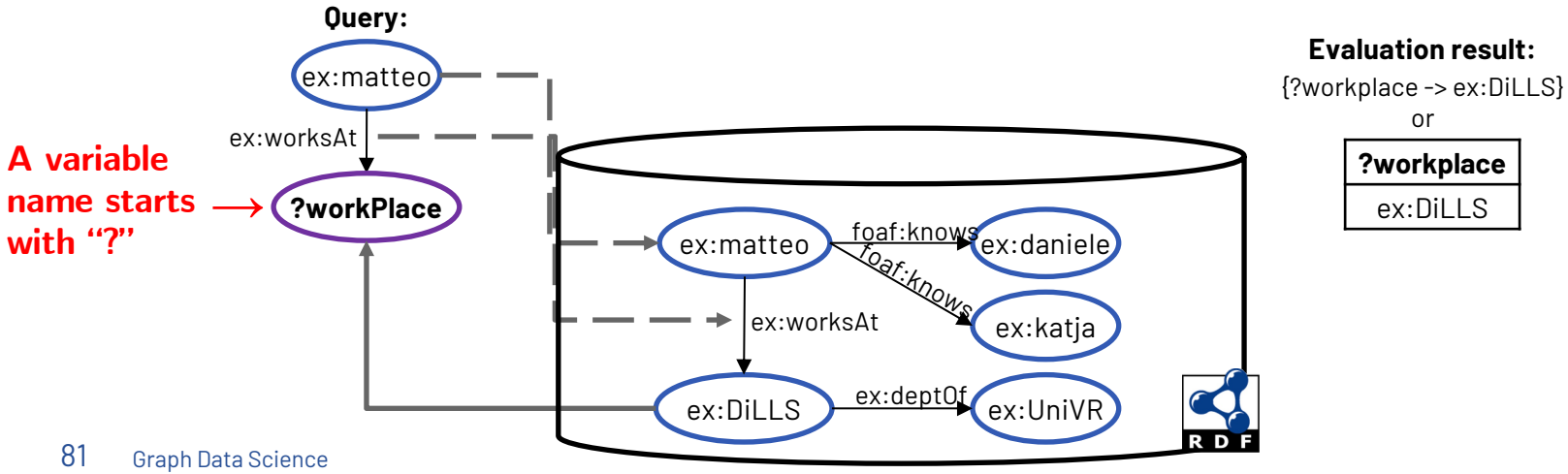
```
MATCH (me)-[:KNOWS*0..2]->(other)
RETURN COUNT(other.name)
```

The SPARQL query language

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

The idea behind SPARQL as a query language:

- Define **patterns** & Patterns have **variables** : a variable can match **ANY NODE or PREDICATE**
  - Everything in RDF is a triple ⟼ so we define **triple patterns**
- Identify portions of the RDF graphs that **match the pattern**
  ⟼ exists a valid **variable assignment**



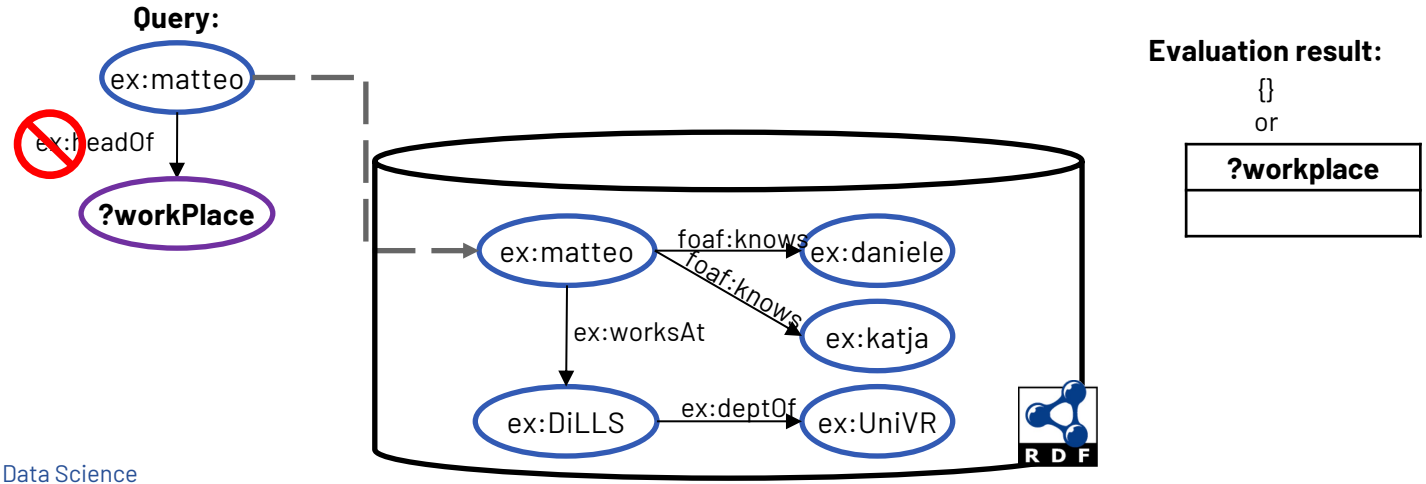
Valid Assignement

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Patterns have **variables** : a variable can match **ANY NODE or PREDICATE**

- Everything in RDF is a triple ⟼ so we define **triple patterns**

match the pattern ⟼ does it exists a valid **assignment?**



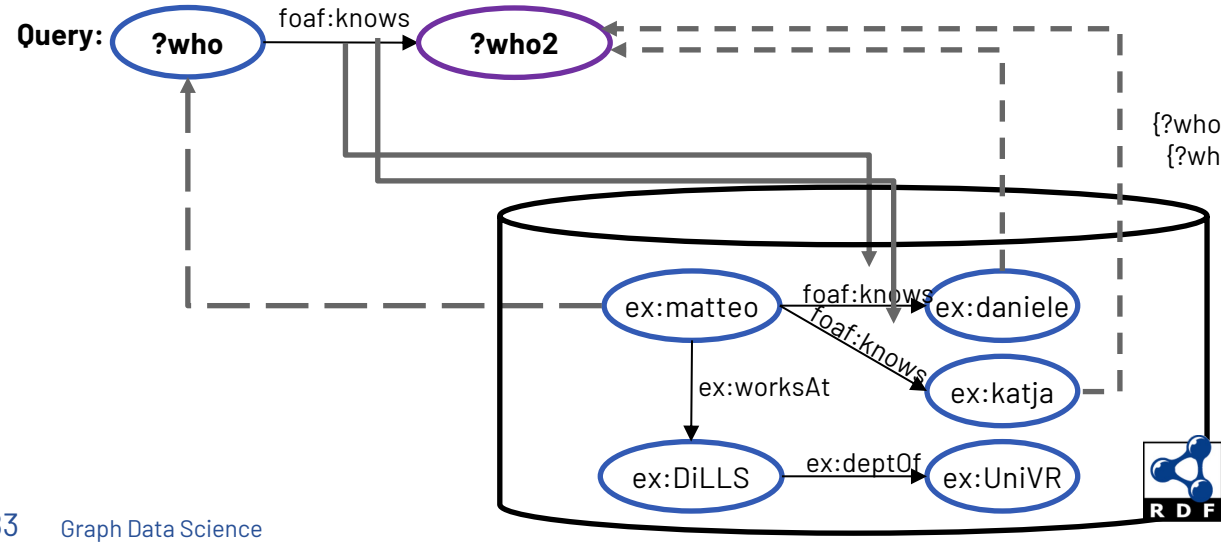
Valid Assignement (2)

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Patterns have **variables** : a variable can match **ANY NODE or PREDICATE**

- Everything in RDF is a triple ⟼ so we define **triple patterns**

match the pattern ⟼ find all possible valid assignments



Evaluation result:

{?who -> ex:matteo, ?who2 -> ex:daniele}
{?who -> ex:matteo, ?who2 -> ex:katja}
or

?who	?who2
ex:matteo	ex:daniele
ex:matteo	ex:katja

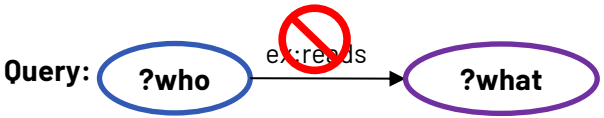
Valid Assignement (3)

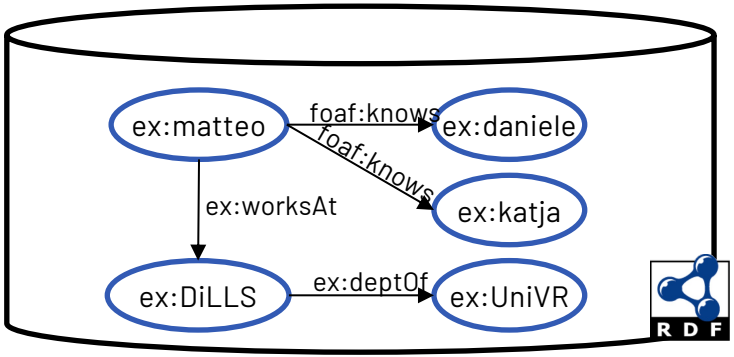
```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Patterns have **variables** : a variable can match **ANY NODE or PREDICATE**

- Everything in RDF is a triple ⟼ so we define **triple patterns**

match the pattern ⟼ find all possible valid assignments?





Evaluation result:

{}

or

?who	?what

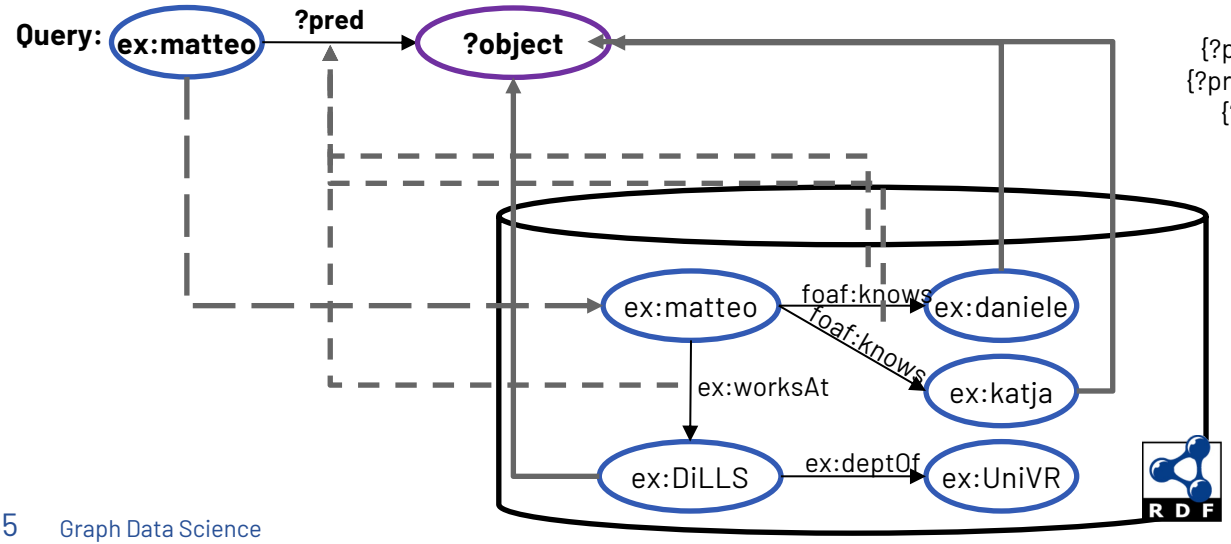
Valid Assignement (4)

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Patterns have **variables** : a variable can match **ANY NODE or PREDICATE**

- Everything in RDF is a triple ⟼ so we define **triple patterns**
  Any position can be a variable

match the pattern ⟼ find all possible valid assignments?



Evaluation result:

{?pred -> ex:worksAt ; ?object -> ex:DiLLS},
{?pred -> foaf:knows ; ?object -> ex:daniele},
{?pred -> foaf:knows ; ?object -> ex:katja}
or

?pred	?object
ex:worksAt	ex:DiLLS
foaf:knows	ex:daniele
foaf:knows	ex:katja

Triple patterns and solution mappings

Triple pattern: an RDF triple where one or more nodes are variables

Variables are denoted by ? (or $) at their beginning

1. ex:matteo ex:worksAt **?workPlace**
2. ex:matteo **?property ?object**
3. **?person** ex:reads **?book**

Evaluating a triple pattern over an RDF graph produces a **multiset (bag) of solution mappings,** examples:

1. {?workplace -> ex:DiLLS} **A solution containing a single mapping**
2. {?property -> ex:worksAt ; ?object -> ex:DiLLS},
   {?property -> foaf:knows ; ?object -> ex:daniele},
   {?property -> foaf:knows ; ?object -> ex:katja}
   A solution containing multiple mappings
3. { } **An empty solution**

Basic graph patterns

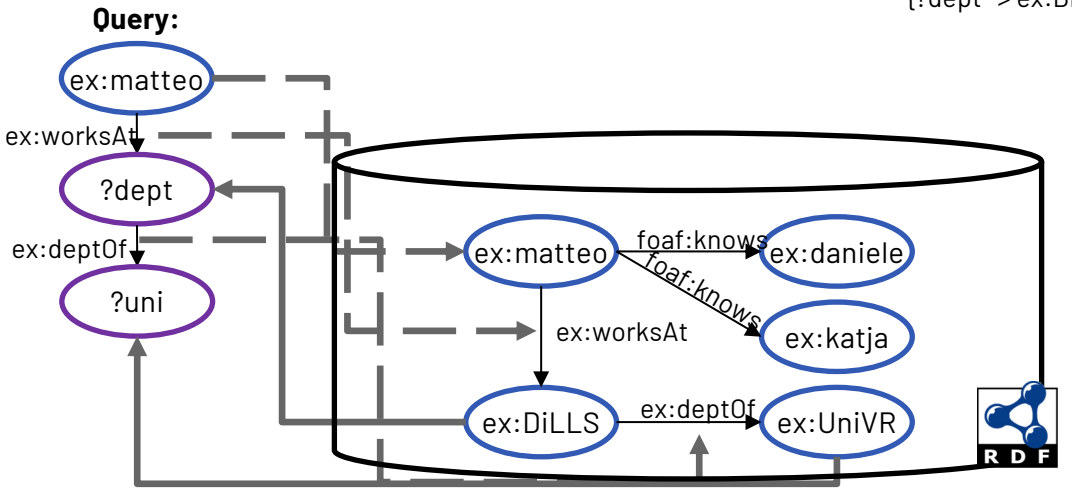
```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Basic graph pattern (BGP): a set of ***one or more triple patterns***

(with optional FILTER clauses)

Evaluation result:

{?dept -> ex:DiLLS ; ?uni -> ex:UniVR}



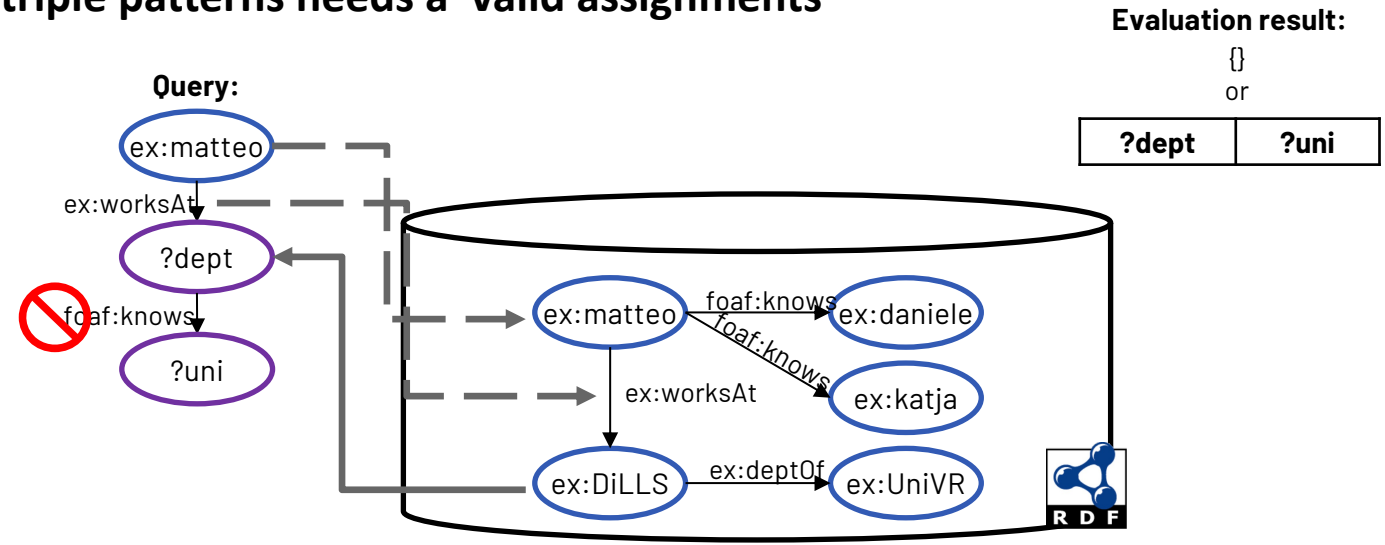
Basic graph patterns (2)

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Basic graph pattern (BGP): a set of ***one or more triple patterns***

(with optional FILTER clauses)

All triple patterns needs a valid assignments



Evaluation result:

{}

or

?dept	?uni

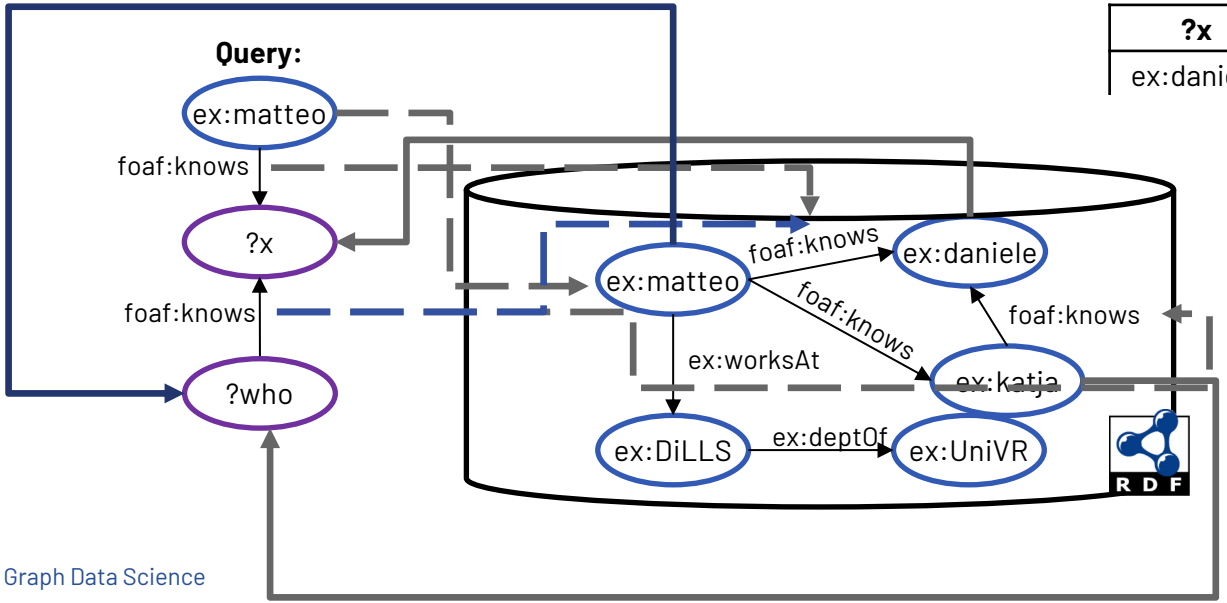
Basic graph patterns (3)

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

Basic graph pattern (BGP): a set of ***one or more triple patterns***

(with optional FILTER clauses)

The same graph object can match multiple times



Evaluation result:

?x	?who
ex:daniele	ex:katja

FILTER

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

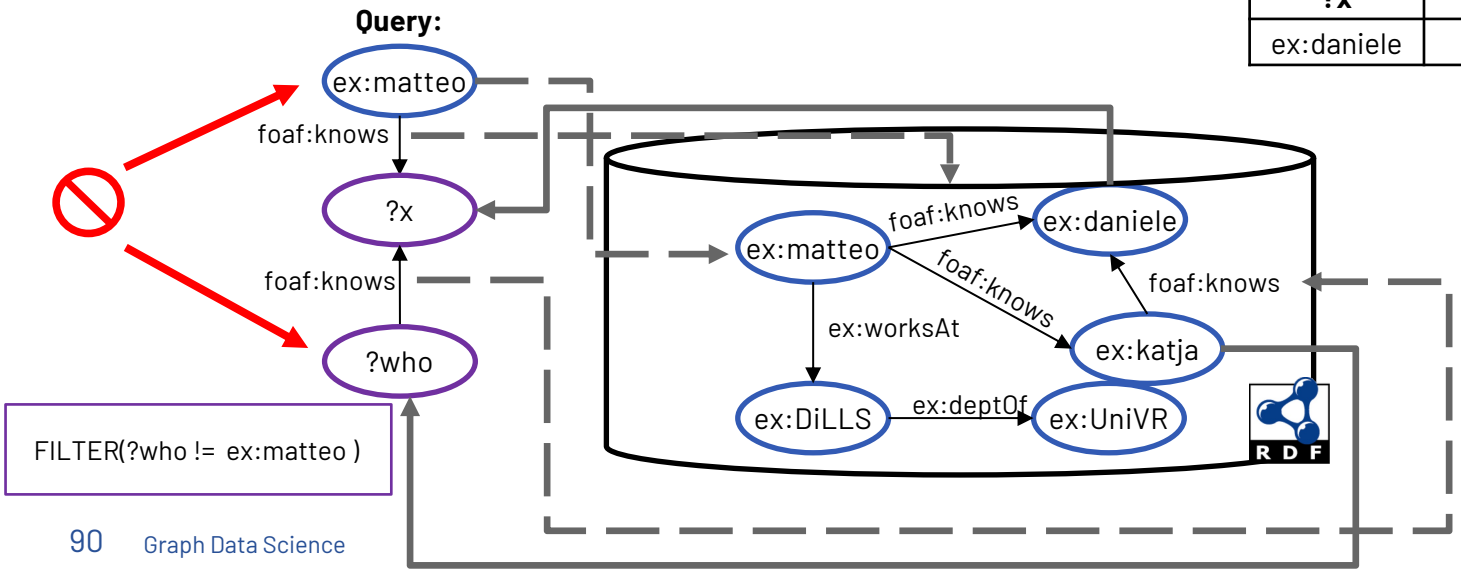
Basic graph pattern (BGP): a set of ***one or more triple patterns***

(with optional FILTER clauses)

The same graph object can match multiple times, we should use FILTER if we want to avoid it

Evaluation result:

?x	?who
ex:daniele	ex:katja

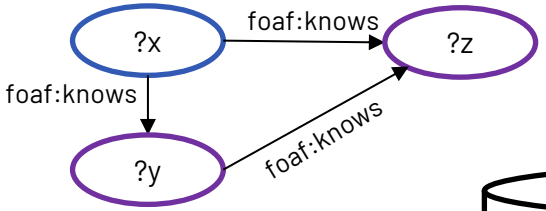


Example 1

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

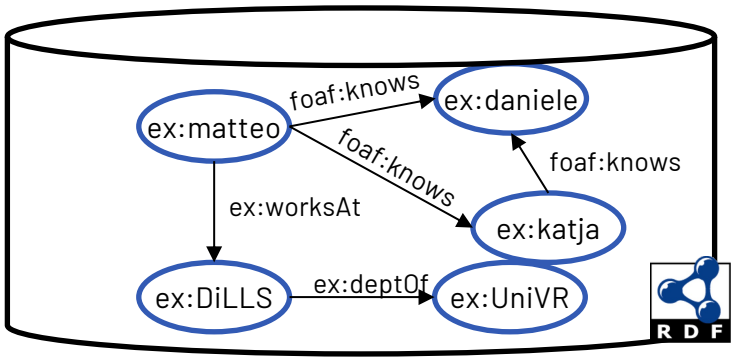
Compute the answer for this BGP
(look closely to the arrows)

Query:



Evaluation result:

?x	?y	?z

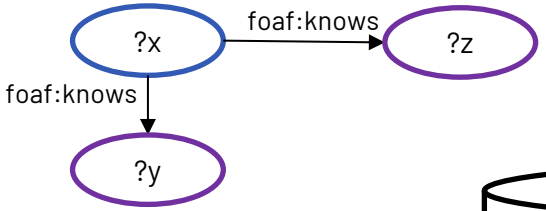


Example 2

```
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/
```

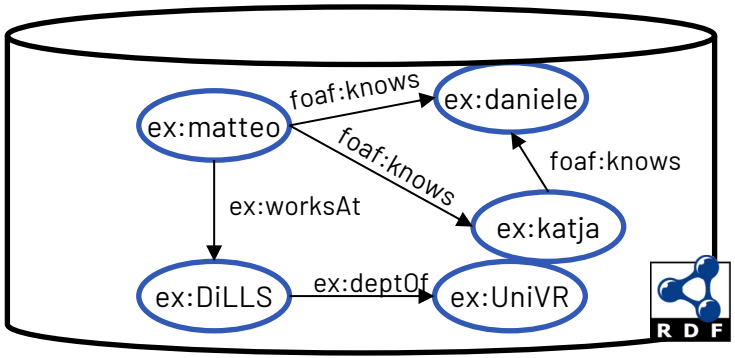
Compute the answer for this BGP
(look closely to the arrows)

Query:



Evaluation result:

?x	?y	?z



Basic Graph Pattern (BGP) Syntax

Basic graph patterns are written in a Turtle-like style:

A set of triple patterns separated by dots (you can read it as "AND")

•ex:matteo ex:worksAt ?dept . ?dept ex:deptOf ?uni .

Shared variables refer to the same node in the graph (like joins)

Turtle abbreviations (using **;** and **,**) can be used

•ex:matteo ex:worksAt ?dept **;**

• foaf:knows ?person1 **,** ?person2 **.** FILTER (?person1 != ?person2)

evaluation result:

{?dept -> ex:DiLLS ; ?person1 -> ex:daniele ; ?person2 -> ex:katja},
{?dept -> ex:DiLLS ; ?person1 -> ex:katja ; ?person2 -> ex:daniele }

FILTER Syntax

- FILTER denotes selection in relational algebra

```
FILTER (?person1  != ?person2)
```

- FILTER allows to specify common **unary/binary operators**:
  - *Less than, greater than, equalities for integer, decimals and date/time*
  - **Conditions over strings**: *Regular expressions*
  - A list of **functions** for specific situations:
    isURI, isIRI, isBlank, isLiteral, isNumeric

  - Selection for **lang & datatype**

```
FILTER ( isLiteral(?age)
                && datatype(?age) = xsd:integer )
                &&  ?age > 30)
)
```


More at: https://www.w3.org/TR/sparql11-query/#func-rdfTerms

SPARQL query with BGP – syntax

The query structure is similar to SQL:

SELECT [FROM] WHERE

```
PREFIX ex: <http://example.org#>
SELECT ?uni
WHERE {
      ex:matteo ex:worksAt ?dept .
      ?dept ex:deptOf ?uni .
}
```

← Prefixes

← Variable(s) of interest (projection)

← BGP (join)

SPARQL query with BGP – syntax 2

The query structure is similar to SQL:

SELECT [FROM] WHERE

```
PREFIX ex: <http://example.org#>
SELECT ?uni
WHERE {
        ex:matteo ex:worksAt ?dept .
        ?dept ex:deptOf ?uni .
        ex:matteo ex:worksAt ?dept2 .   FILTER( ?dept != ?dept2)
}
```

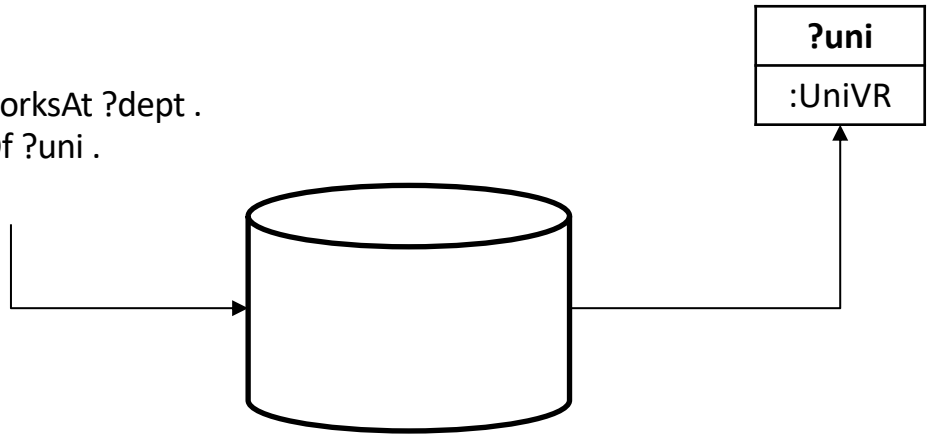

Thanks Daniele Dell'Aglio, AAU

SPARQL query with BGP – results

The result of a SPARQL SELECT query is a collection of bindings

- Related to solutions mappings, but not always in a one-to-one relation
- SELECT results are usually represented as tables

```
PREFIX ex: <http://example.org#>
SELECT ?uni
WHERE {
  ex:matteo ex:worksAt ?dept .
  ?dept ex:deptOf ?uni .
}
```



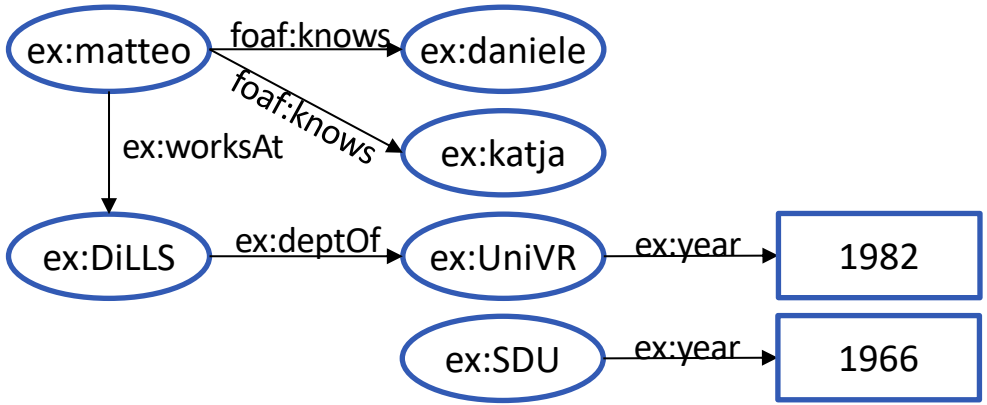
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

Matching literals

- Literals can be included in triple patters (similarly to Turtle)

```
PREFIX ex: <http://example.org#>
SELECT ?uni
WHERE {
  ?uni ex:year 1982 .
}
```

?uni
ex:UniVR

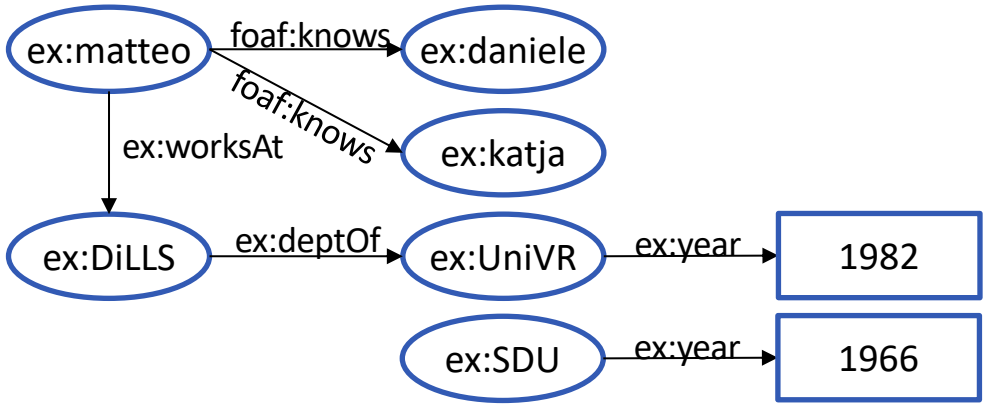


ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

Inequalities over literals

- In several cases we do want to assess inequalities over literals (e.g. less or greater than)
- SPARQL manages these conditions through FILTER

```
PREFIX ex: <http://example.org>
SELECT ?uni
WHERE {
  ?uni ex:year ?year .
  FILTER( ?year < 1990 ) .
}
```



?uni
ex:UniVR
ex:SDU

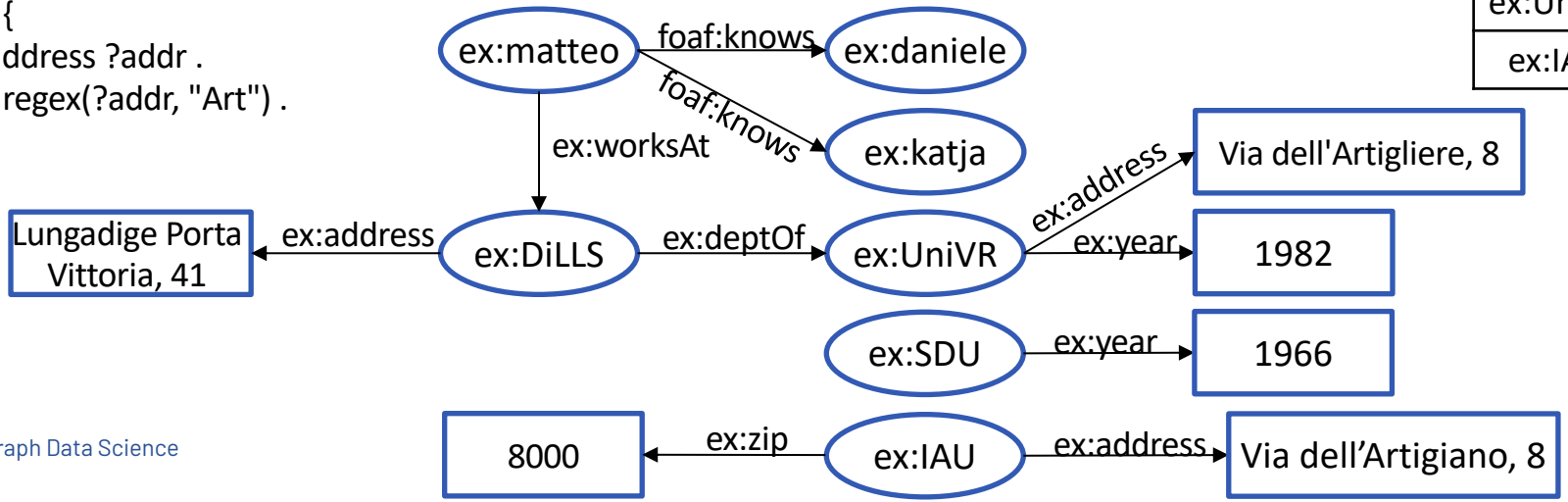
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

Regular expressions

- Regular expressions are used to match strings
  - Managed through the regex function
  - More at: https://www.w3.org/TR/xpath-functions/#regex-syntax

```
PREFIX ex: <http://example.org#>
SELECT ?x
WHERE {
  ?x ex:address ?addr .
  FILTER regex(?addr, "Art") .
}
```

?x
ex:UniVR
ex:IAU



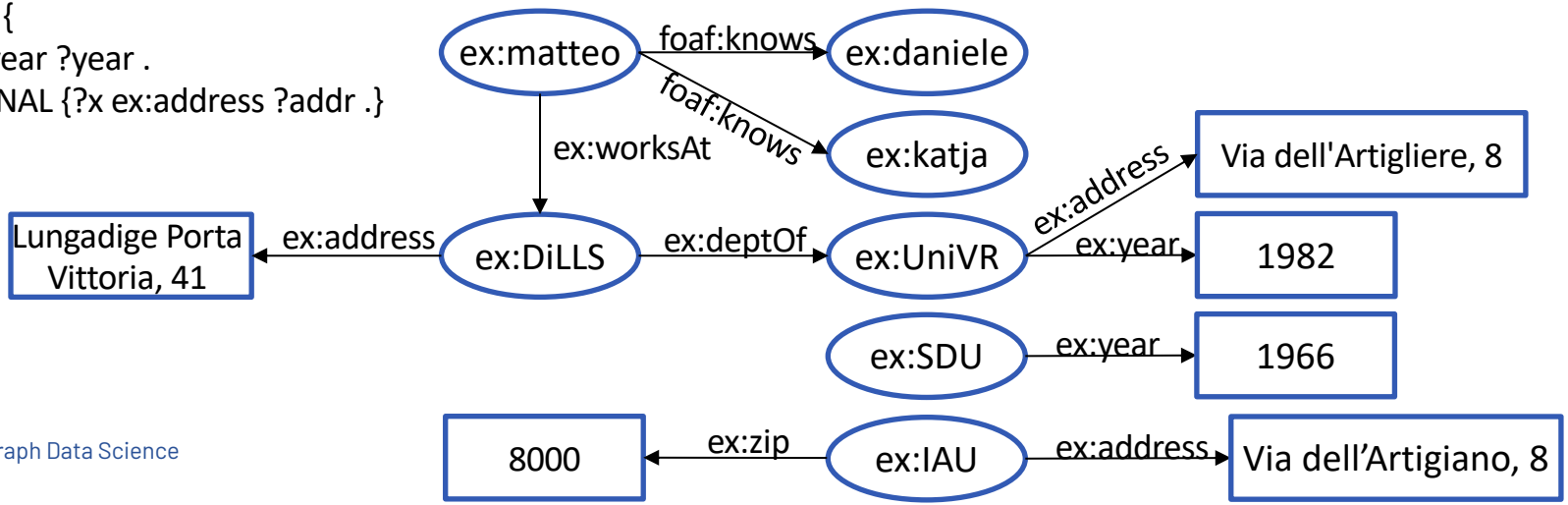
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

The optional graph pattern

- Optional graph patterns contain triple patterns that may be without matches in the RDF graph
- it allows partial match like a LEFT JOIN

?x	?year	?addr
ex:AAU	1974	Fredrik Bajers Vej 7K
ex:SDU	1966	

```
PREFIX ex: <http://example.org#>
SELECT ?x ?year ?addr
WHERE {
  ?x ex:year ?year .
  OPTIONAL {?x ex:address ?addr .}
}
```

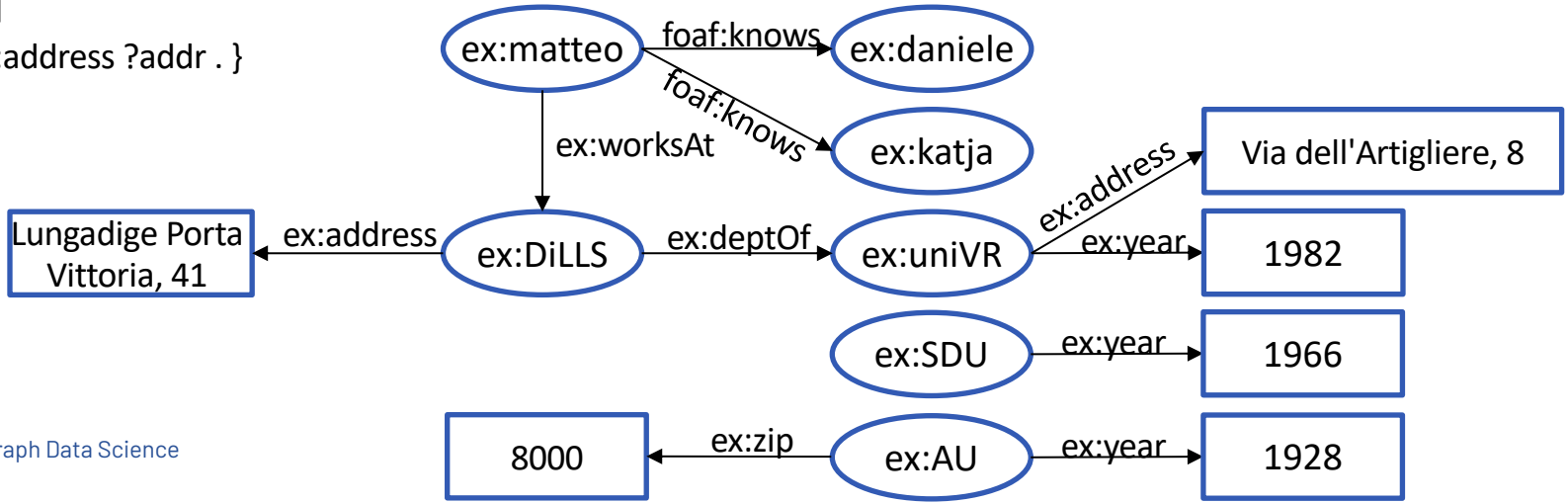


ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

The union graph pattern

- Union graph patterns contain triple patterns that may be without matches in the RDF graph

```
PREFIX ex: <http://example.org#>
SELECT ?x ?addr
WHERE {
  { ?x ex:zip ?addr . }
  UNION
  { ?x ex:address ?addr . }
}
```



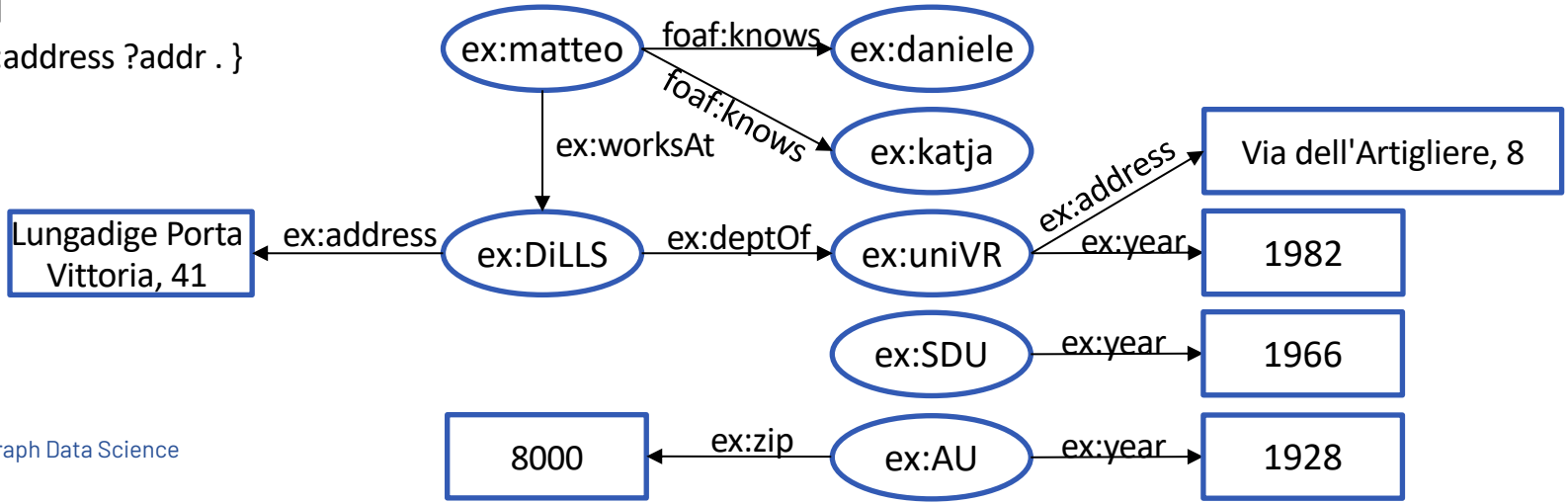
ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

The union graph pattern

- Union graph patterns contain triple patterns that may be without matches in the RDF graph

```
PREFIX ex: <http://example.org#>
SELECT ?x ?addr
WHERE {
  { ?x ex:zip ?addr . }
  UNION
  { ?x ex:address ?addr . }
}
```

?x	?addr
ex:DiLLS	Lungadige Porta Vittoria, 41
ex:UniVR	Via dell'Artigliere, 8
ex:AU	8000



Property path: Sequence and alternative paths

Property path allows to define routes between nodes

- Sequence path /

```
PREFIX ex: <http://example.org#>
SELECT ?uni
WHERE {
 ex:matteo ex:worksAt/ex:deptOf ?uni .
}
```

```
PREFIX ex: <http://example.org#>
SELECT ?uni
WHERE {
  ex:matteo ex:worksAt ?loc .
  ?loc ex:deptOf ?uni .
}
```

- Alternative path |

```
PREFIX ex:
<http://example.org#>
SELECT ?x ?addr
WHERE {
  ?x ex:zip|ex:address ?addr .
}
```

```
PREFIX ex:
<http://example.org#>
SELECT ?x ?addr
WHERE {
 { ?x ex:zip ?addr . }
 UNION
 { ?x ex:address ?addr . }
}
```

ex -> http://example.org#
foaf: -> http://xmlns.com/foaf/0.1/

Property path: Arbitrary length path

- Specify path of variable length
- One or more path +

```
{ ex:matteo foaf:knows+ ?person . }
```

{?person -> ex:daniele}, {?person -> ex:katja},
{?person -> ex:alice}, {?person -> ex:bob}

- Zero or one path ?

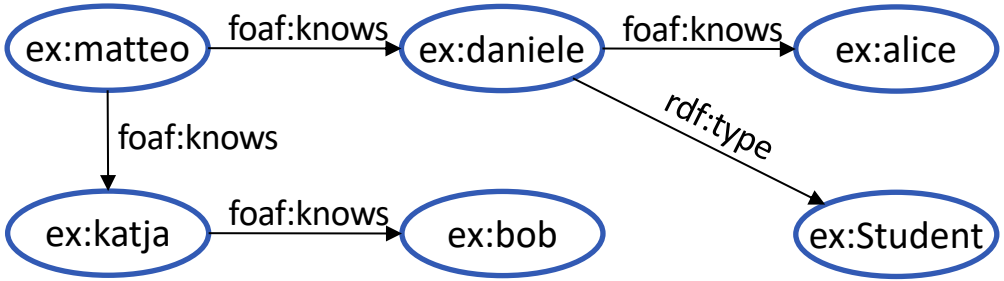
```
{ ex:matteo foaf:knows? ?person . }
```

{?person -> ex:matteo}, {?person -> ex:daniele}
{?person -> ex:katja}

- Zero or more path *

```
{ ex:matteo foaf:knows* ?person . }
```

{?person -> ex:daniele}, {?person -> ex:katja},
{?person -> ex:alice}, {?person -> ex:bob},
{?person -> ex:matteo}



Property path: Other path operators

- Inverse property path ^

```
PREFIX ex: <http://example.org#>
SELECT ?loc
WHERE {
  ex:daniele ex:worksAt ?loc .
}
```

```
PREFIX ex: <http://example.org#>
SELECT ?loc
WHERE {
 ?loc ^ex:worksAt ex:daniele .
}
```

- Negated property set !

```
PREFIX ex: <http://example.org#>
SELECT ?loc
WHERE {
  ?person !ex:worksAt ?loc .
}
```

Exploring with SPARQL Queries (I)

Show all the types being used

```
SELECT DISTINCT ?type
WHERE { ?s a ?type }
```

What Properties connect each type

```
SELECT DISTINCT ?typeS ?property ?typeO
WHERE {
?s a ?typeS .
?s ?property ?o .
?o a ?typeO .
} ORDER BY ?typeS ?property ?typeO
```

Find the Taxonomy

```
SELECT DISTINCT ?type ?super
WHERE {
?s a ?type .
?type rdfs:subClassOf ?super
} ORDER BY ?type ?super
```

https://www.bobdc.com/blog/exploringadataset/

Exploring with SPARQL Queries (II)

Types and super types for some entities

```
SELECT DISTINCT ?type WHERE {
db:Isaac_Newton a / rdfs:subClassOf* ?type .
}
```

Incoming & Outgoing properties

```
SELECT DISTINCT ?propertyO ?typeO ?propertyI ?typeI
WHERE {  {
  db:Isaac_Newton ?propertyO ?o .     ?o a ?typeO .
} UNION  {
    ?i ?propertyI db:Isaac_Newton .   ?i a ?typeI .
} }
```

https://www.bobdc.com/blog/exploringadataset/

Property Graph Query Language: Gremlin

Imperative Graph Traversal

```
g.V().has('name', 'Tom Hanks').out() .values("name");
```

```
g.V().has('name', 'Tom Hanks').out().out().out().values("name");
```

Declarative Graph Traversal

```
g.V().match(
  as("a").has("name", "Tom Hanks"), as("a").out("directed").as("b"),
  as("b").in("acted_in").as("c"), where("a",neq("c"))
).values("name")
```

Try out: https://gremlify.com/

https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html

Outline

1. **Graphs are Everywhere**
   - The Web-Link structure
   - The Query-Log graph
   - The Social network
   - The Knowledge graph
2. **The Graph Model**
   - Undirected/Directed graphs
   - Labelled/Unlabelled graphs
   - N-partite graphs
   - RDF graphs
   - Property Graph
   - Graph Database vs. Database of Graphs
3. **Representing Graphs**
   - Adjacency matrix
   - Adjacency List
   - Triples & Storage for Triplestore
   - Property graph storage models
4. **Graph Navigation**
   - Breadth-First Search / Depth-First Search
   - Connected Components
   - Paths & Shortest path
   - CYPHER
   - SPARQL
   - Gremlin

Further References

Davis Shurbert, An Introduction to Graph Homomorphisms
http://buzzard.ups.edu/courses/2013spring/projects/davis-homomorphism-ups-434-2013.pdf

Andreas Schmidt, Iztok Savnik, Conference on Advances in Databases, Knowledge, and Data Applications, Overview of Regular Path Queries in Graphs
https://www.iaria.org/conferences2015/filesDBKDA15/graphsm_overview_of_regular_path_queries_in_graphs.pdf

Jure Leskovec, CS224W: Machine Learning with Graphs | 2019 |
Lecture 3-Motifs and Structural roles in networks
http://snap.stanford.edu/class/cs224w-2019/slides/03-motifs.pdf

Davide Mottin and Emmanuel Müller, Graph Exploration:
Let me Show what is relevant in your graph, KDD tutorial
https://mott.in/slides/KDD2018-Tutorial-Compressed.pdf

Kolaczyk, E.D., Statistical Analysis of Network Data, Chapter 5: Sampling and Estimation in Network Graphs.
https://link.springer.com/chapter/10.1007/978-0-387-88146-1_5

Jure Leskovec, Christos Faloutsos, Sampling from Large Graphs
https://dl.acm.org/doi/pdf/10.1145/1150402.1150479