Semantic Technology – Part 1

**Objectives of this Introduction:**

* To clarify how RDF graph databases are different from SQL "relational" databases
* To introduce the main syntaxes for creating RDF graphs (with focus on the **Turtle syntax**)

**Summary**

The RDF standard is at the core of what we call **Semantic Technology**. It is a standard that can be used in two ways:

* as Graph Databases (an alternative to SQL databases for data storage)
* as a Data Format (an alternative to JSON/XML for transferring data between apps, APIs or services, to ensure interoperability)

This introduction is relevant for both use cases, as it showcases syntaxes that are necessary to create RDF graphs, regardless of what we plan to do with them (to store/query them in a Graph Database or to pass them between applications/APIs).

**Theoretical clarifications**

RDF is a set of formats that allow us to store machine-readable information in ways that are close to how they are managed by the human mind – as **networks of associations between concepts/identifiers/data**, instead of

* lists of records (as we are used to from SQL databases),
* collections of objects or documents (as we are used to from JSON/XML data services).

As a starting example, we want to store the following information hereby expressed in natural language:

*Susana's child is Robert. Robert has two children, Dan and Patrik. Dan likes Terminator. Patrik lives in Viena. Robert works at U.B.B.*

Storing this rather small quantity of information in tables would require:

* A normalization effort, to avoid the presence of NULL values (not all individuals share the same attributes/properties),
* Complex JOINs even in queries where the answer is rather simple and easily identifiable,
* Frequent schema migration/renormalization if we obtain new information about the same individuals, which may involve new relationships not present in this initial dataset.

A graph representation of this information can be obtained by using RDF's Turtle syntax, which allows us to also express it as a set of simple **statements**, somewhat close to the original form:

@prefix : <http://buchmann.ro#>.

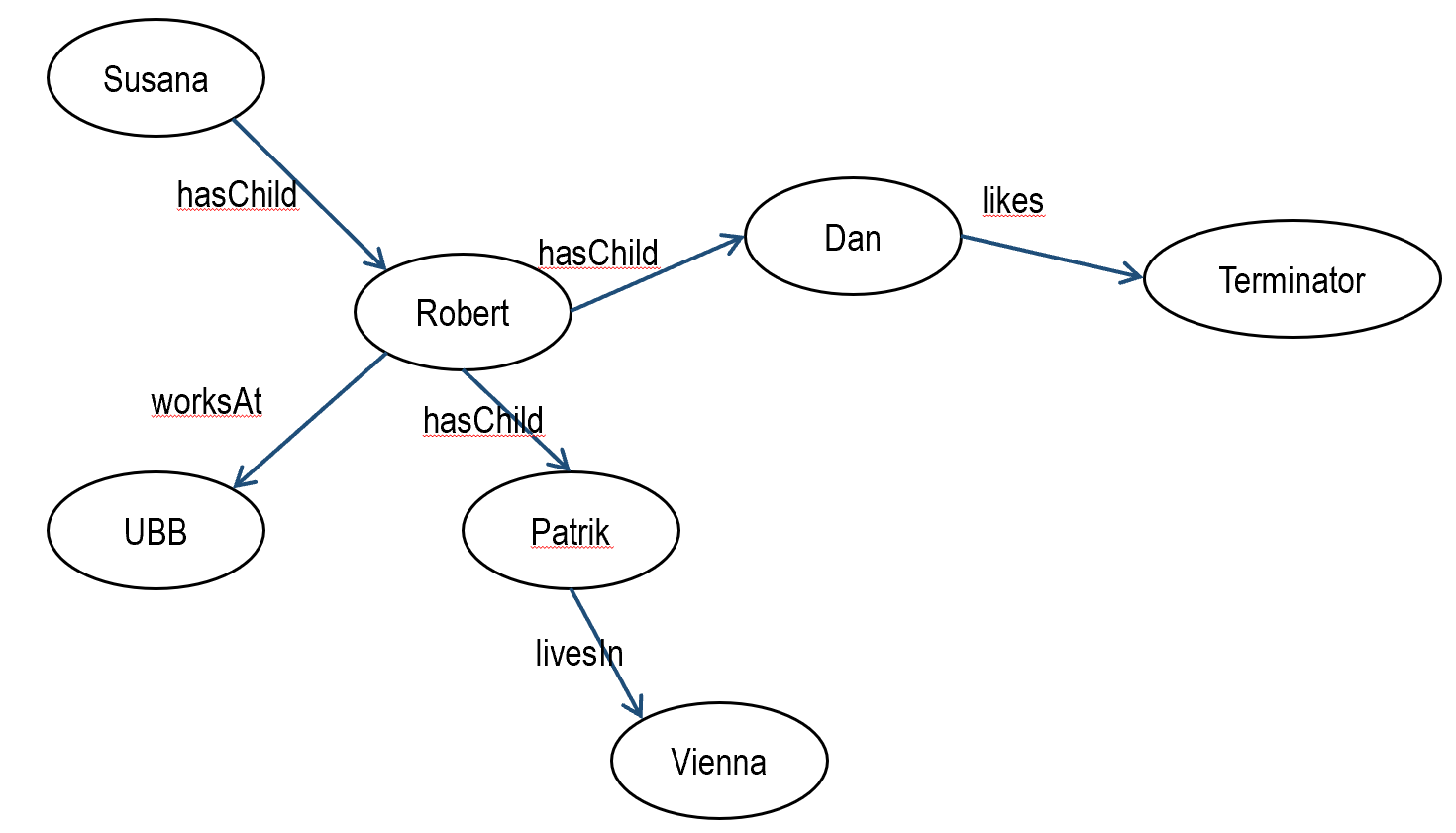
:Susana :hasChild :Robert.

:Robert :hasChild :Dan, :Patrik; :worksAt :UBB.

:Patrik :livesIn :Viena.

:Dan :likes :Terminator.

These statements will form a **directed graph** (edges always start from the subject of a statement; it's allowed to have multiple edges between the same things, even going in both directions):



If we store the Turtle text on an RDF graph database server we will be able to run SPARQL queries such as:

*What is the relationship between Susana and Robert?*

**(i.e., a client app can discover the relationship without knowing it in advance!)**

SELECT ?rel

WHERE {:Susana **?rel** :Robert}

=> hasChild

*What do Susana's grandchildren like?*

**(i.e., a query that navigates a path in the graph)**

SELECT ?x

WHERE

{:Susana **:hasChild/:hasChild/:likes** ?x}

=> Terminator

*Give me everything you know about Robert*

**(i.e., a client app can run such exploratory queries when it only knows the identifier of the individual, but not the relationships involving that individual)**

DESCRIBE :Robert

=> will return the subgraph of all nodes directly connected to Robert (it will reveal his relationships, thus informing future queries about what relations can be found in the database)

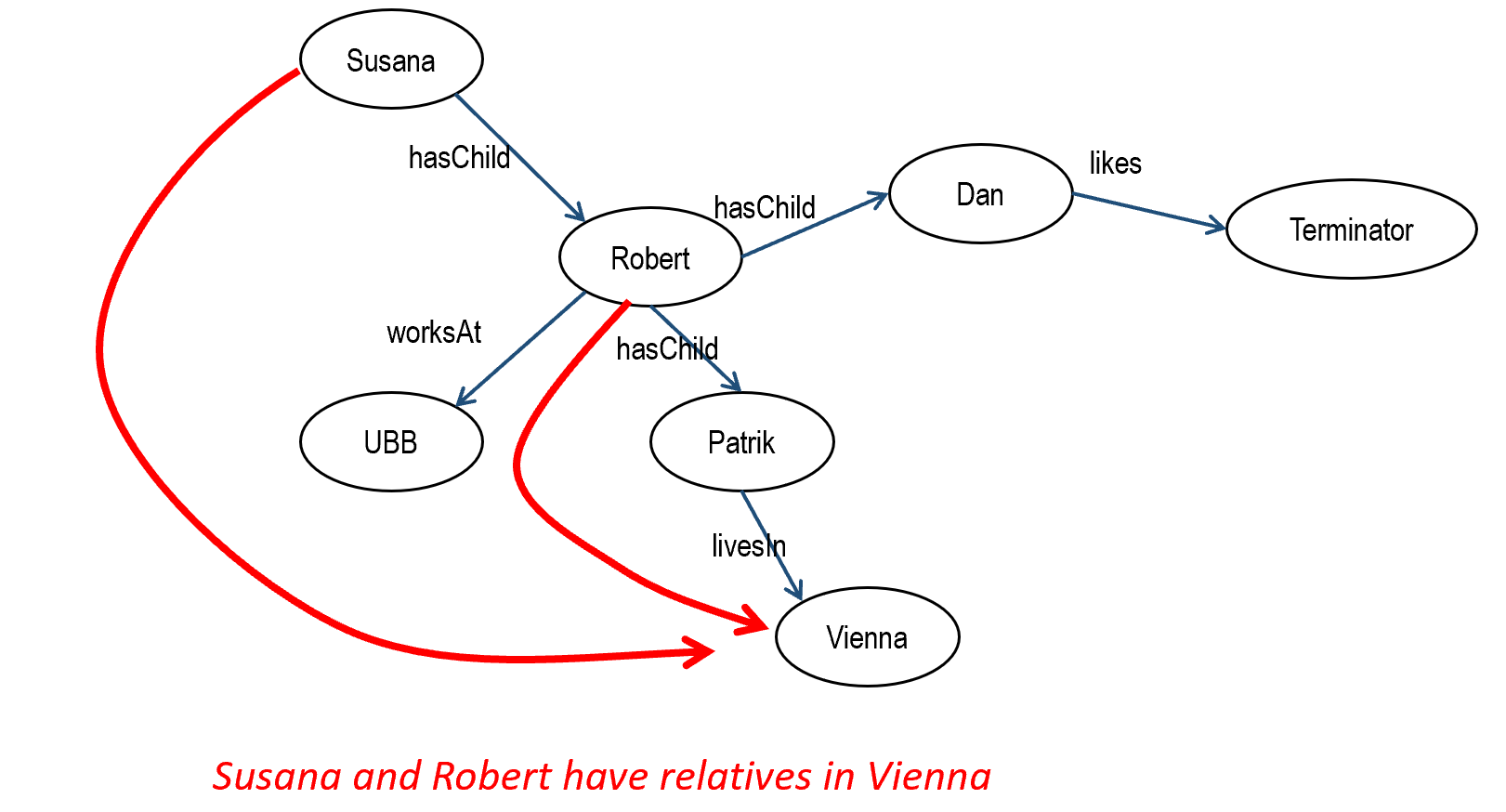
*For every X whose descendants live in Y, generate the shortcut relationships X has relatives in Y*

**(this is an example of Machine Reasoning, the ability to generate new information via deductive rules)**

INSERT {?x :hasRelativesIn ?y}

WHERE {?x :hasChild+/:livesIn ?y}

=> will add to the graph the following statements:  
:Susana :hasRelativesIn :Viena.   
:Robert :hasRelativesIn :Viena.[[1]](#footnote-1)



In order to run such queries, an RDF graph database server is necessary. However, this introductory part of the tutorial will only focus on the Turtle syntax that allows us to write graphs in any text editor.

Before starting the practical part of the tutorial, we summarize below the key characteristics of this type of database – try to correlate these characteristics with the examples provided below:

An RDF graph database may be seen as:

* A "NoSQL database":
  + Data is not stored in tables, but in **network structures** (nodes connected by edges/relationships) therefore SQL cannot be used to retrieve data;
  + The query language for RDF graphs is SPARQL, whose key principle is **to match or navigate graph patterns** (from what we know to what we want to retrieve, following some network of relationships);
* A "knowledge base" (more recently advertised as "knowledge graph"), since it can be used to store knowledge, not only data. A trivial differentiation is the following:
  + **Data** are values of various types (integer, string, boolean etc.), examples: "John" (string), 10 (integer)
  + **Knowledge** comprises two kinds of constructs:
    - **statements** ("facts", expressing complex networks/chains of relationships between entities - possibly involving data, somewhat close to how we express them in natural language), e.g.: John obtained a 10 in the Web Semantic exam at FSEGA during 2019
    - **reasoning rules** ("deductive rules", capable of generating automatically new information from existing facts), e.g.:

IF “X and Y obtained a grade in the same course, same year and same faculty”,

GENERATE (the new statement) "X was a colleague of Y"

* "Linked Data", "Smart Data", ”Structured Data (Markup)” - marketing buzzwords used to promote semantic technology for non-technical people:
  + ”Linked Data” = suggests **the ability to connect graphs that are stored on different servers** (even different organizations), thus allowing queries to collect data from multiple graph servers without having to set up a custom API (a standard data retrieval API comes with any RDF server);
  + ”Smart Data” = suggests **an opposition to “(Dumb) Big Data"** where complex analytics must make sense of large quantities of messy data; therefore "Smart Data" is a about high quality data already prepared to be consumed by smart systems; a typical scenario is to complement the two approaches – processing Big Data in order to produce Smart Data/Knowledge Graphs;
  + ”Structured Data” = when the focus is on **the possibility to embed graphs in HTML code**[[2]](#footnote-2) (for SEO purposes, allowing search engines or Web scrapers to easily retrieve graph data from Web front-ends).
* A "semantic database", suggesting that data is connected to its context/meaning in a way that allows that context/meaning to also be queried.

*We define* ***context*** *as the network of all the* ***relevant relationships, direct or indirect*** *(i.e. relationship chains) around an entity or data point. A key word here is "relevant " – a chain of relationships will lose relevance (or intensity) as it becomes longer. For example*

* *it is irrelevant to know if a person has 6 degrees of social separation from the President of U.S.A. because most people on the planet have a similar distance;*
* *it is more relevant to know that a person knows someone who knows the President (only one degree of separation, a common point of social contact).*

*Context gives* ***meaning*** *to a piece of data – however "meaning" is a bit more: it also includes the* ***rules*** *to which those relationships obey (e.g. what are the things that are supposed to be connected by a relationship; which relationship is mutual, which is not).*

*Graph databases are designed to facilitate the querying of both the context (e.g. how are two things related?) and meaning (e.g. to what rules do those relationships conform?) This is more difficult to achieve with traditional databases, where it is assumed:*

*(a) that relationships are known and fixed (applications use them to perform JOIN queries, they never ask what relationships exist between two given things)*

*(b) that context/meaning is (at least partially) hardcoded in the applications*

*(c) that the purpose of the database schema is to reject any data properties that were not foreseen at design time, therefore context is assumed to be known (and therefore hardcoded in apps)*

With this we suggest the possibility to query not only data (e.g. "select the names of products having price>100 for customer X", "select the names of employees in department X"), but also:

* + **Relationships not previously known** ("what relationship exists between product X and user Y?", "is there a suspicious chain of personal relationships between person X and person Y?")
  + **The nature of those relationships** ("what family relationships exist between the persons mentioned in my graph?", "is the relationship between person X and person Y a mutual one?")
  + **Patterns of combination between relationships and entity types** ("what family relationships exist between the managers and the employees in my company?")

With respect to the term "relational databases", although most often used for SQL databases, it is important to notice that both graphs and SQL databases are ”relational” in the sense that *both work with relationships between entities*. However, the theoretical foundations are different – graph theory vs. relational algebra – and the focus is also different:

* Graph databases are *relationship-centric*: **the ”primary need” is to manage relationships that are complex** (often chained)**, unpredictable** (not necessarily known in advance)**, between heterogeneous entities** (i.e., not sharing a common schema) ... and later we'll see what data can be collected for the entities involved in those relationships;
* SQL databases are *data-centric*: the **”primary need” is to manage lists of homogeneous records** (that fit a shared schema) ... and after that we see what relationships are needed to keep such "lists of records" together, in a coherent data model (which will change rarely).

**Some recommended tools**

RDF graphs can be written in several syntaxes, but the most human-friendly is **Turtle**. If other syntaxes are needed, we can use syntactic convertors. In the following examples we focus on Turtle, but other syntaxes will also be exemplified:

* TriG (only minor difference from Turtle)
* N-triples (we rarely write it manually, but we must be able to recognize/read it)

Tools that can be used to learn RDF syntaxes (not only Turtle):

* <http://www.easyrdf.org/converter>
* http://issemantic.net/

These are syntactic converters – we can write a graph in Turtle, then we can convert it to also see other syntaxes (if no errors are made). **The second converter also provides a visualizer.**

* <http://ttl.summerofcode.be/>

Simple validator for the Turtle syntax, but **does not provide syntactic conversion features**.

* <https://github.com/dotnetrdf/dotNetRDF.Toolkit/releases>

This is a toolkit called dotNetRDF[[3]](#footnote-3) which contains among other tools the rdfEditor, providing syntax coloring and validation in multiple syntaxes. **It does not provide syntactic conversion in the editor itself.**

Those who prefer to write their code in Visual Studio Code can also install support for RDF syntax coloring[[4]](#footnote-4).

* <http://www.ldf.fi/service/rdf-grapher>

Useful to generate a graph visualization of reasonable size. **It can also indicate some syntax errors, but the main goal is visualization**.

*For the first steps, we recommend the EasyRDF convertor as it can show us how other syntaxes look. Then, as we will need to write more complex graphs, a combination of syntax coloring and validation will be necessary to correct syntactic mistakes – rdfEditor can help a lot, although VS Code with syntax coloring may be more popular.*

## Writing graphs in Turtle

An **RDF graph** contains multiple **statements**. A statement can be written in Turtle as follows:

@prefix : <http://buchmann.ro#>.

:Mary :daughterOf :John.

The statement (the second line) contains 3 **terms**: a subject, a property (predicate) and an object. There are also some ways of grouping multiple statements about the same thing. For example, we want to add that Mary is the daughter of John and Anna, and that she lives in Cluj. We can write this as distinct statements:

@prefix : <http://buchmann.ro#>.

:Mary :daughterOf :John.

:Mary :daughterOf :Anna.

:Mary :livesIn :Cluj.

Or, we can avoid repetition by grouping everything in a more complex phrase (a **description**):

@prefix : <http://buchmann.ro#>.

:Mary :daughterOf :John, :Anna; :livesIn :Cluj.

Notice the syntactic rules and how we use the delimiters:

* The character "." ends a statement or a phrase
* The character "," separates multiple objects for the same property and the same subject
* The character ";" separates multiple properties of the same subject
* Spaces must separate the terms of a statement/phrase, which means that the space should never occur inside a term (daughterOf is written as one "word")
* We'll see later why all these terms start with the character ":"

The comma can NOT be used to group multiple subjects in the same statement!**When the subject changes, we are forced to start a new description (phrase)***.* So if we want to say that Mary and Anna are the daughters of John, we are not allowed to write it as:

:Maria, :Ana :daughterOf :John.

We can write it as two different statements, or we can reverse the direction of the relationship:

:John :hasDaughters :Maria, :Ana.

When we reverse the direction of a relation we usually employ a different term to reflect this reversal – e.g. :hasDaughters instead of :daughterOf. In such cases we should also declare that the new term has exactly the same meaning, only different direction. This kind of declarations belong to the **graph schema** (where the meaning/usage of each relationship is specified) and is done in a standard way**[[5]](#footnote-5)**:

:hasDaughters owl:inverseOf :daughterOf

This ability to correlate the meanings of terms is the reason why we also call this "semantic technology" (it deals with meaning, not only with data).

You may have noticed that all the terms in this example start with a **prefix** (here, the character ":") which must be declared at the very beginning:

@prefix : <http://buchmann.ro#>.

This is a namespace declaration that should be understood as *all terms starting with : were created by the owner of the domain address* [*http://buchmann.ro*](http://buchmann.ro). In other words, this is how we indicate the **provenance of terms** and it is important for two reasons:

* It indicates whom should be asked about the meaning of each term (ideally, on that address some meaningful descriptions should be provided);
* It allows us to combine terms of diverse provenance in the same graph, even in the same statement.

The prefix declaration is necessary only once at the beginning of a Turtle file; later we will see that **it is also necessary in queries**!

## Another syntax (N-triples) and the notion of URI

Another syntax that we will encounter is N-triples. Our first statement can be rewritten in N-triples by concatenating the provenance address (namespace) to each term:

<http://buchmann.ro#Mary> <http://buchmann.ro#daughterOf> <http://buchmann.ro#John> .

One disadvantage is that in N-triples there's no way of grouping and shortening statements – each statement must be written on a different line. If Turtle allows us to write complex phrases...

@prefix : <http://buchmann.ro#>.

:Mary :daughterOf :John, :Anna; :livesIn :Cluj.

...in N-triples we have to break them down on multiple lines:

<http://buchmann.ro#Mary> <http://buchmann.ro#daughterOf> <http://buchmann.ro#John> .

<http://buchmann.ro#Mary> <http://buchmann.ro#daughterOf> <http://buchmann.ro#Anna> .

<http://buchmann.ro#Mary> <http://buchmann.ro#livesIn> <http://buchmann.ro#Cluj> .

Turtle is of course preferred, but if we receive graphs as N-triples from an API we should be able to recognize what we get.

In N-triples it is more obvious how each term belongs to a domain address/namespace. Such a full term is called **URI (resource identifier)[[6]](#footnote-6)**. Each URI will allow us to see the provenance of a term, and even to consult that address for perhaps additional explanations on the meaning of that term. There are three fundamental principles applicable to URIs:

* **The Principle of Non-ambiguity** (mandatory)**.** A URI is a **global ID**, i.e. it can be used across the entire Web! This means that the same URI **should NOT be reused** with multiple meanings (should not identify more than one thing). For example http://buchmann.ro#Mary cannot identify two different persons, not even in different databases/servers! If we have two Marys under the same domain address, we must force the distinction between their URIs:
  + Either we force it in the local ID part:

http://buchmann.ro#Mary1

http://buchmann.ro#Mary2

* + Or we force it in the provenance address, by extending it conveniently (which implies different prefixes!)

http://buchmann.ro/departmentA#Mary

http://buchmann.ro/departmentB#Mary

* **The Principle of Non-unique identity** (mandatory). Although a URI cannot be reused for different things, it is allowed for the same thing to have multiple URIs – e.g. the same person could be identified as <http://buchmann.ro#Mary> in one graph and [http://mybusiness.ro#Maria](http://mybusiness.ro#PopMaria) in another. The two identifiers will be considered synonyms – i.e. if their equivalence is declared in a standard way, stating that they have the same meaning (we'll see this in the next section);
* **The Principle of Dereferencing** (optional[[7]](#footnote-7)).Notice that all these global identifiers start with "http://..." What if someone thinks they are URL addresses and tries to type one in a browser (or sends an HTTP request)? Ideally something useful should be returned, however there's no imposed standard on what "useful" means:
  + it may be a Web page about that identifier, it may be a dictionary of terms ("terminology") that explains the meaning of the term etc.; for example, DBPedia is a public RDF database where the URI of Donald Trump is <http://dbpedia.org/resource/Donald_Trump> - if we type it in a browser, we get a Web page with all the details available there about Donald Trump;
  + another option: every time the server receives a request on that URI, it should return all URIs directly connected to it (in the graph hosted by that server); this can be programmed as a subgraph extraction query (DESCRIBE, if you remember the introductory example), thus allowing a client to "rummage around", i.e. to explore the graph by sending HTTP requests for hopping from one node to the next

*Applications are responsible to implement these principles. For example the first principle means that*  ***whenever a new URI is inserted in the graph the application should check if it was not already used for something else****. The concatenation of the domain address to every ID is a good way to manage this risk, but additional measures may be required:*

* *Sometimes the graph is generated from existing sources (Excel, SQL tables, JSON); we have to ensure that proper URIs will be generated – e.g. by concatenating the domain address to primary key values, plus perhaps even the table name (if key values are repeated in multiple tables, such as in the case of Autonumber fields);*
* *If the URI is derived from what the user types in some front-end form field, the application should:*
  + *warn the front-end user if the URI already exists*
  + *or, concatenate to the URI a component that is guaranteed to be unique (e.g. UUID codes)*

We hope you noticed the similarity between the notion of **URI** and a version of it that is more popular in common talk: **hashtags**. They are nothing more than URIs connecting together all people, posts, websites that are related to (mention) a certain concept identifier. This allows apps to collect data about all things connected to the same hashtag.

We can even go back with this analogy to the principle of **Dereferencing** explained earlier – when you click a hashtag concept (i.e. you send an HTTP request to it), you will get the "subgraph" of everything that is connected to that concept within a certain system (Facebook, Twitter etc.). In conclusion, graph data structures are all around us, even if you're not aware of them!

## Prefixes allow us to combine terms from different sources

We've mentioned that different URIs can identify the same thing – this commonly happens because different organizations (with different domain addresses) would assign different IDs to the same thing without knowing about each other.

To indicate the equivalence of such "synonym" URIs, we can include **declarations of equivalence** (”sameness”), which are statements using the standard property **owl:sameAs**:

@prefix p1: <http://buchmann.ro#> .

@prefix p2: <http://mybusiness.ro#> .

@prefix owl: <http://www.w3.org/2002/07/owl#> .

p1:Mary owl:sameAs p2:Maria .

Or, in N-triples:

<http://buchmann.ro#Mary> <http://www.w3.org/2002/07/owl#sameAs> <http://mybusiness.ro#Maria> .

As the prefix suggests, the term **owl:sameAs** comes from the OWL standard terminology.

Such a declaration of equivalence will establish a machine-readable link between different identities provided by different servers for the same thing – such links can be navigated by queries trying to collect data about the same thing from multiple sources. This practice established the term **Linked Data**, pointing to the possibility of achieving system integration at data level, rather than the traditional API level.

The above example suggests that we can combine terms of different provenances even in the same statement. *Anyone who owns a domain address (a) may invent their own terms, (b) may adopt terms created by others, or (c) may combine them freely. Anyone can define their own terminology and publish it to encourage adoption –* ***such a collection of reusable terms is also called "ontology", "vocabulary" or, by analogy with traditional databases, "schema"****.*

This means that equivalence declarations are not the only statements where terms of different provenance can be combined - we can do this in ANY statements, for example:

<http://facebook.com/Maria> <http://schema.org/parent> <http://buchmann.ro#John> .

This statement is very similar to the one used earlier – it has the same meaning, but each term has a different provenance:

* The subject is an ID assigned by Facebook for Maria; also notice here that character / may also be used to separate the domain address from the local ID (both / and # are acceptable as separators, but # should only be used once in a URI)
* The property *parent* is taken from a well-known terminology called **Schema.org** (established by Google, Microsoft, Yahoo for graphs that should be "understood" by search engines)
* The object is again an ID, this time assigned by the owner of the domain address buchmann.ro.

If we write this in Turtle, the different provenances require us to define different prefixes (only a default prefix can be ”:”, the rest should have some additional, suggestive characters - fb:, schema: etc.)

@prefix fb: <http://facebook.com/>.

@prefix : <http://buchmann.ro#>.

@prefix schema: <http://schema.org/>.

fb:Maria schema:parent :John .

We've already mentioned that all the terms used in the above examples are URIs ("global identifiers")

* Typically, if a URI is used at the beginning or at the end of a statement, it is an ID (of a person, in this example), similar to a global primary key;
* URIs in the middle of a statement typically express the relationship/property expressed by that statement (oftentimes such properties are adopted from public terminologies such as Schema.org[[8]](#footnote-8)).

However, not all the terms used in RDF graphs are URIs! Other categories of terms that can be found in graphs are:

* **Literal values** (data of various types)
* **Anonymous nodes** (blank nodes)
* **URL addresses** (e-mail accounts, Web pages, REST API routes, file paths, any other way of addressing a digital resource by a URL)

In the next sections we'll see how these categories of nodes fit in a RDF graph.

## Literal values (the "data")

Any database should contain data, right? **Data in the traditional sense**  - strings, integers, booleans etc.). In RDF graphs, all those are called **literal values**. Their key characteristics are:

* they can have a datatype (the same datatypes defined by the XML Schema standard[[9]](#footnote-9))
* they don't have a provenance/prefix
* they can only be present at the end of a statement (it's not allowed to start a statement with a number, string etc.)

Here is a simple example in Turtle, expressing the age attribute with an integer value[[10]](#footnote-10):

@prefix : <http://buchmann.ro#>.

:Anna :hasAge 20 .

Therefore two kinds of properties can be used in an RDF graph:

* those that connect two URIs, thus expressing a **relationship** between two things
* those that connect a URI to a data value, thus expressing an **attribute** having that value

In the N-triples syntax, data can be annotated with datatype identifiers (taken from the XML Schema standard):

<http://buchmann.ro#Anna> <http://buchmann.ro#hasAge> "20"^^<http://www.w3.org/2001/XMLSchema#integer> .

Actually, literal values is what we typically store in a traditional database! An SQL table can be converted to a graph according to the following pattern:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID (primary key)** | Name | Age | WorksFor (foreign key) |
| **:PopAna** | Pop Ana | 20 | :CompanyA |
| **:PopAndrei** | Pop Andrei | 30 | :CompanyA |
| **:PopescuVasile** | Popescu Vasile | 50 | :CompanyB |

@prefix : <http://buchmann.ro#> .

:PopAna :age 20; :name "Pop Ana"; :worksFor :CompanyA; a :Person.

:PopAndrei :age 30; :name "Pop Andrei"; :worksFor :CompanyA; a :Person.

:PopescuVasile :age 50; :name "Popescu Vasile"; :worksFor :CompanyB; a :Person.

Notice the conversion rules:

* **primary key values** and **foreign key** values become URIs (the primary key will provide the subject for each statement)
* **all the other data points** become literal values of various types
* the **field names become properties** (except for the primary key, although some automatic converters will redundantly generate that property as well)
* the **property a is a standard form for ”is a”** (it states the type of a thing); in SQL tables, the type of things is given by the table name, but here we have no tables, so this explicit type declaration should be included (*sometimes, instead of ”a” you may find the equivalent form ”rdf:type”, suggesting that this is a standard RDF term).*

This example of converting a table to a graph reveals another important distinction – between identifiers (URIs) and labels (strings). The **URI** is a ”primary key” (whose scope extends to the entire Web) whereas the **name/label** is a string value to be displayed in a front-end.

Reminder: *there can not be two things with the same URI, in the entire Web, not only in a database; however, there can be multiple things with the same name/label, even in the same database.*

* **URIs** should never be displayed in front-end, their goal is to **identify** things and to **connect** information (and, optionally, to be used as URL addresses providing an entry point to a graph-based Information System – see the "Dereferencing" approach discussed earlier);
* **Names/labels** are what we're supposed to display in afront-end, they have to be understandable by a human reader;
* So make sure that every URI has a label assigned to it, to support both functions! The association between a URI and its (one or more) string labels can be done with the standard term **rdfs:label**:

@prefix : <http://buchmann.ro#> .

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:PopAna :age 20; **rdfs:label** "Pop Ana"; :worksFor :CompanyA; a :Person.

:UBB **rdfs:label** "University Babeș-Bolyai"; :nrEmployees 3000 .

:Dan **rdfs:label** "Popescu Dan"; :salary 5000.

We can also use our own improvised properties for this purpose – if we want to differentiate between different kinds of labels: full names for persons, titles for books, labels for products etc.

:PopAna :fullname "Pop Ana".

:Terminator :title "The Terminator".

:P1 :label "Fridge".

In such a case, it is recommended to support client apps to easily collect any kind of label. This can be done by declaring that all these labelling properties are specialized variants of the standard term **rdfs:label**.

:fullname **rdfs:subPropertyOf** rdfs:label.

:title **rdfs:subPropertyOf** rdfs:label.

:label **rdfs:subPropertyOf** rdfs:label.

Client apps are thus given the possibility to either query all kinds of labels, or only a specific variant.

Notice the use of **rdfs:subPropertyOf** to define more specific variants of a property. We can also use it to say that "having a relative" covers a multitude of more specific relationships – sibling, parent, uncle etc. Earlier we saw the use of **owl:inverseOf** to say that two properties have the same meaning but opposite direction. Such declarations (**axioms**) normally belong to the graph schema, but nothing stops us to write them together with the statements about individuals.

Both are examples of mechanisms that will help machines to detect that certain properties have related meanings (reminder: *semantic technology deals with meaning, not only with data*).

## Datatypes and language codes

Any database will need **datatypes** for its data values. In SQL databases these are specified by the schema and a schema is mandatory before any data values can be stored. However:

* In RDF graphs, a prescribed schema is not mandatory, so **datatypes can be attached to each individual data value**
* Moreover, **strings can be annotated with language codes** (if we want to store labels in multiple languages for the same thing).

Both datatypes and language codes are useful as query filters.

Example in Turtle:

@prefix : <http://buchmann.ro#> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:PopAna :age "20"^^**xsd:integer**;

:name "Pop Ana"^^**xsd:string**;

:married "true"^^**xsd:boolean**;

:birthDate "1980-01-01"^^**xsd:date**.

:Cluj :name "Cluj Napoca"**@ro**, "Kolozsvar"**@hu**.

The Turtle syntax has a great advantage – some datatypes are automatically recognized: strings (anything between quotation marks), integers (numbers without quotation marks), booleans (the words *true* and *false* with no prefix and no quotation marks), decimals (numbers with a decimal point). *All other datatypes must be specified explicitly – e.g. calendar dates*. If we apply all these simplifications, only the date must have its type explicit:

@prefix : <http://buchmann.ro#> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:PopAna :age 20; :name "Pop Ana"; :married true; :birthDate "1980-01-01"^^xsd:date.

:Cluj :name "Cluj Napoca"@ro, "Kolozsvar"@hu.

*Notice the prefix xsd: applied to datatypes. This prefix is associated with the* ***XML Schema standard namespace*** *because the RDF standard reuses the XML datatypes! In addition to XML datatypes, RDF only brings a few datatypes of its own:*

* ***rdf:HTML*** *– strings that contain HTML code*
* ***rdf:XMLLiteral*** *– strings that contain XML code*

*This implies that HTML and XML code templates / snippets can be stored directly in a graph database (to be injected in a multilingual front-end in a flexible way).*

## Entity types (classes)

Earlier we saw that the verb "is a" (with its standard forms **a** or **rdf:type**) allows us to state WHAT are the things identified by a URI. We don't *have to* state this, but it helps a lot if we do it – the query possibilities will become much richer (e.g. a query such as "select all movies that..." can only be executed if the graph database knows which are movies and which are not).

Regarding entity types some guidelines should be followed:

1. Any thing mentioned in a graph should have an entity type (in object-oriented terms it is a **class**, while the IDs/individuals are **instances**). Multiple entity types can be assigned to the same individual:

:Ana a :Woman, :Student, :EUCitizen.

2. If some of the types are related, we can "merge" them in supertypes with the help of the standard term **rdfs:subClassOf**:

@prefix : <http://buchmann.ro#> .

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:Woman **rdfs:subClassOf** :Person.

:Student **rdfs:subClassOf** :Person.

:EUCitizen **rdfs:subClassOf** :Person.

This works like a set union - it will allow queries to easily collect all the persons, even if the Person type is not explicitly attached to any individual. Everyone in the following statements will be easily detected as "persons" by queries:

:Ana a :Woman, :Student, :EUCitizen.

:Andrei a :Student.

:Marian a :EUCitizen.

Just like with any other RDF term, entity types can be improvised by us or adopted from known terminologies. For example the Person type is already provided by Schema.org, its prefixed form being **schema:Person**. On the other hand, for Woman, Student, EUCitizen we don't have standard terms.

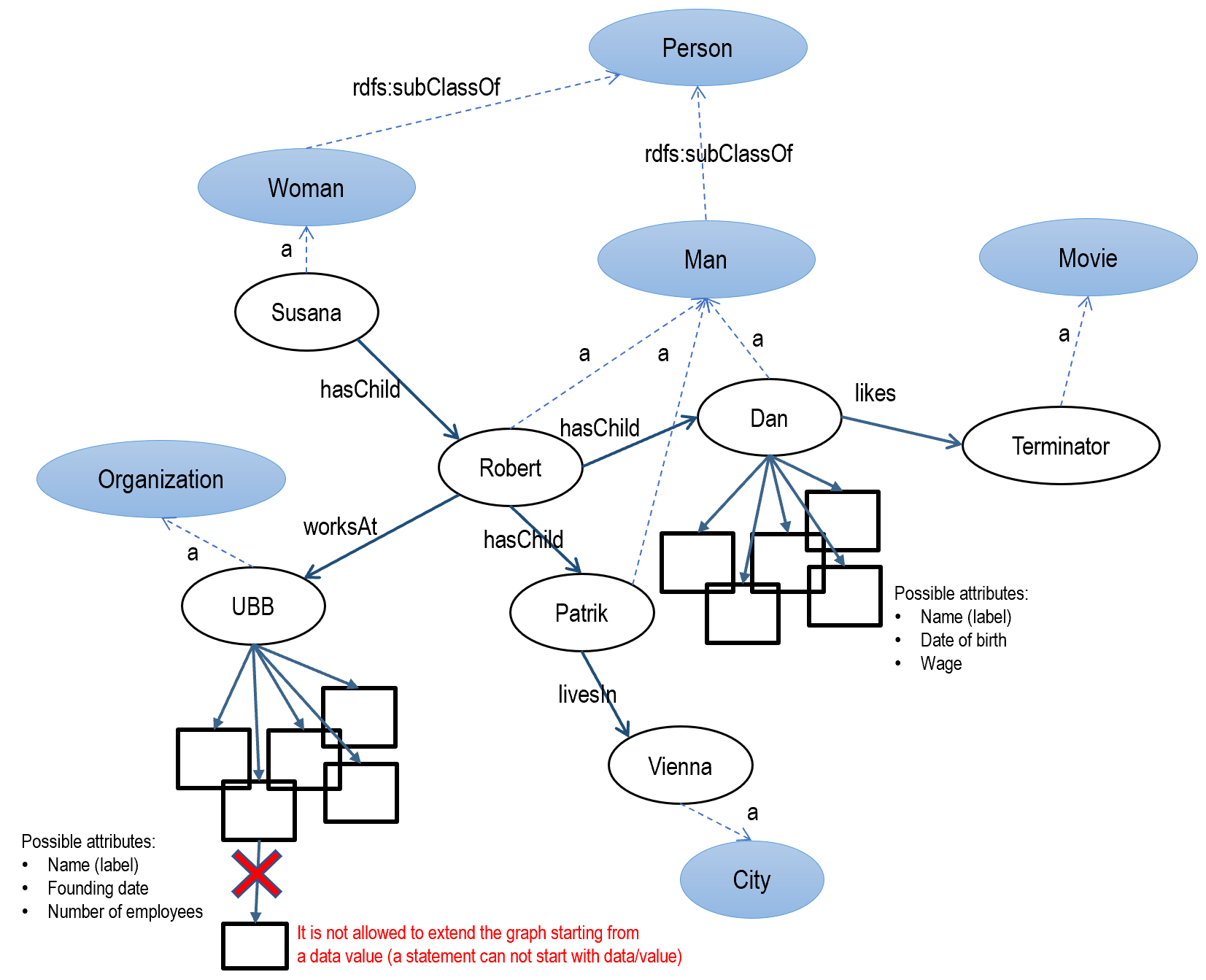
3. Manually assigning entity types to every thing mentioned in a graph can be cumbersome. The good news is that types can also be generated automatically, through **Machine Reasoning** rules that implement the following kind of logic:

IF "X works at Y", GENERATE "X is a Person. Y is a Organization".

*Such queries can be implemented as queries or as axioms, to be presented later.*

To conclude the last few sections, we stress that an RDF graph, to be really useful, should include:

* **Data of various types** ("hanging fruits" attached to every node, to be picked by queries as they navigate along graph paths); data is what a front-end should display, the end-user should never see URIs!
* **Entity types attached to URI nodes**; these will enrich queries with multiple filtering possibilities, based on the meaning of individual nodes.



*For data, we also remind about the constraint highlighted in the figure: a statement can NOT start with data, all data values must be terminal nodes in the graph.*

## Anonymous nodes

Sometimes a graph must store information about things that do not have (yet) an ID; or, we don't care much about their IDs, but we care about their connections to other things. For such cases we use **anonymous nodes** (or **blank nodes**).

**Anonymous nodes** are artificially introduced to obtain certain grouping patterns or complex phrases, to keep things together in useful ways (e.g. to avoid ambiguities for some queries). An anonymous node has a temporary identifier prefixed by an underscore (e.g. \_:x).

There are several common use cases for anonymous nodes:

1. As an intermediary placeholder for something that does not have yet an ID,
2. As a way of composing multiple binary relationship into a more complex relationship (with multiple participants),
3. As a way of attaching data values to property instances (not only to nodes),
4. As a data structure (similar to a JSON object) that groups several "fields",
5. As a way of expressing ordered structures (lists)

*Use Case 1: Placeholder nodes*

We want to express the fact ”Anna knows someone who knows Peter”:

@prefix : <http://buchmann.ro#>.

:Anna :knows \_:x .

\_:x :knows :Peter .

We don't even need to create the local identifier \_:x, we can write the same as:

@prefix : <http://buchmann.ro#>.

:Anna :knows [:knows :Peter] .

Notice the square brackets that contain everything we know about the "unknown" thing.

Several anonymous nodes can be chained, as in ”Anna knows someone who lives somewhere in Romania”.

@prefix : <http://buchmann.ro#>.

:Anna :knows [:livesIn [:locatedIn :Romania]] .

Anonymous nodes can also be subjects in RDF statements, as in "Someone who lives in Cluj knows Anna":

@prefix : <http://buchmann.ro#>.

[ ] :knows :Anna; :livesIn :Cluj.

...or, if we include everything we know about the node between its brackets:

@prefix : <http://buchmann.ro#>.

[:knows :Anna; :livesIn :Cluj ].

*In most cases, the square brackets are preferred instead of the underscore ID. Generally it does not matter how we identify the anonymous nodes, because (a) underscore IDs will be replaced by RDF servers with arbitrary codes and (b) they cannot be used in queries. There is however a situation when we have to provide such an ID: when multiple statements about the same anonymous node are written in different parts of the same file, and cannot be grouped around the same square brackets.*

*Use case 2: Creating a "data structure" (multiple "fields" grouped together)*

This is similar to creating an ad-hoc "object" in programming, by simply assigning values to its "attributes". Those attributes can be collected by queries either individually or together.

We want to express that ”Anna lives at the address Cluj, str. Horea 14, Romania”.

@prefix : <http://buchmann.ro#>.

:Anna :livesAt [:city :Cluj; :street :Horea\_Str; :number 14; :country :Romania].

Notice that the street identifier is :Horea\_Str and not simply :Horea because for the future we expect that some persons might get the identifier :Horea. *Reminder: the same URI cannot identify multiple things in the entire Web! So we need a strategy to prevent using the same identifier with multiple meanings.*

*Use case 3: Non-binary relationships*

Anonymous nodes can express complex relationships (**n-ary relationships**) that involve more than two things. Such complex relationships can be:

* heterogeneous, when different properties are used to express the involvement of the multiple participants;
* homogeneous, when the same property expresses similar involvement for all participants.

An example of a heterogeneous involvement is: ”Linda Hamilton played the character Sarah Connor in the movie Terminator” – it involves a person, a character, and a movie. We can "tie them together" with different relationships connected to the same anonymous node:

:LindaHamilton :actorIn [:movie :Terminator; :role :SarahConnor].

...or, using the anonymous node as subject:

[:actor :LindaHamilton; :movie :Terminator; :role :SarahConnor].

An example of homogeneous involvement is: "Paul, Ana and Andrei are colleagues in the same student group at FSEGA". Instead of connecting them pairwise, it's more efficient to connect all of them to an anonymous node representing the student group (it will be easier in the future to add new students to the group!):

[a :StudentGroup; rdfs:member :Paul, :Ana, :Andrei] :enrolledAt :FSEGA.

Notice the standard term **rdfs:member** for decomposing a complex thing in its parts.

*Use case 4: Data values attached to property instances*

We saw how data are typically the "hanging fruits" attached to any URI (or anonymous!) node. Sometimes we need to attach data values to different instances of the same property, as in: "The distance Cluj-București is 450km, the distance Cluj-Bistrița is 110 km". We can write this as follows:

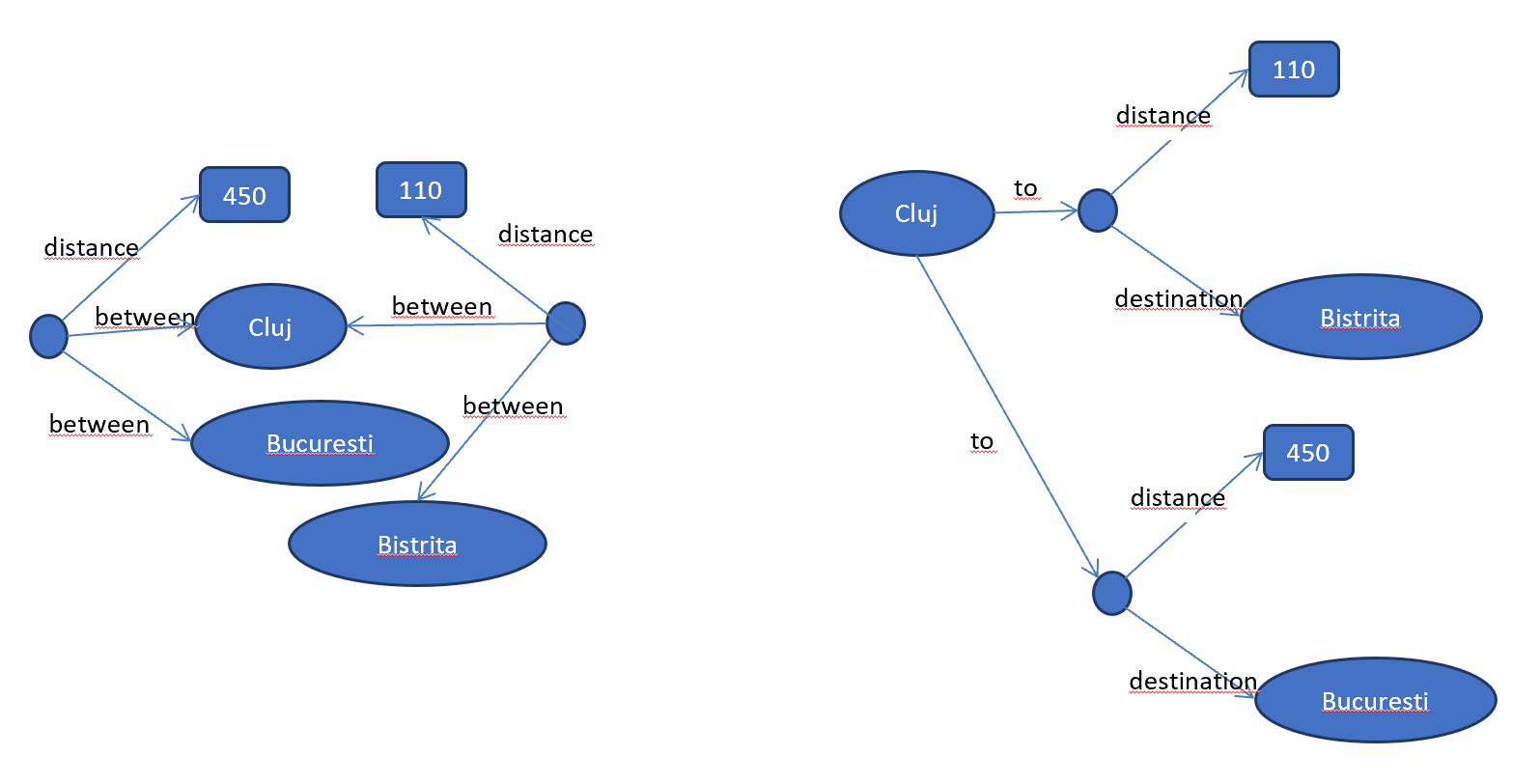
:Cluj :to [:destination :Bucuresti, :distance 450], [:destination :Bistrita, :distance 110].

...or, with the anonymous node as subject (we lose a sense of direction, as both cities are attached in the same way to the anonymous node):

[:between :Cluj, :Bucuresti; :distance 450].

[:between :Cluj, :Bistrita; :distance 110].

The graphs for the two variants look quite different, and we have to consider this when trying to query them:



*More recently, a proposed extension to the RDF standard, called* ***RDF-star[[11]](#footnote-11)****, introduced the following way of attaching data to relationships by* ***embedding statements*** *(statements become subjects or objects in other statements):*

*<<:Cluj :connectedTo :Bucuresti>> :distance 450.*

*<<<<:Cluj :connectedTo :Bucuresti>> :distance 450>> :connectionType :road.*

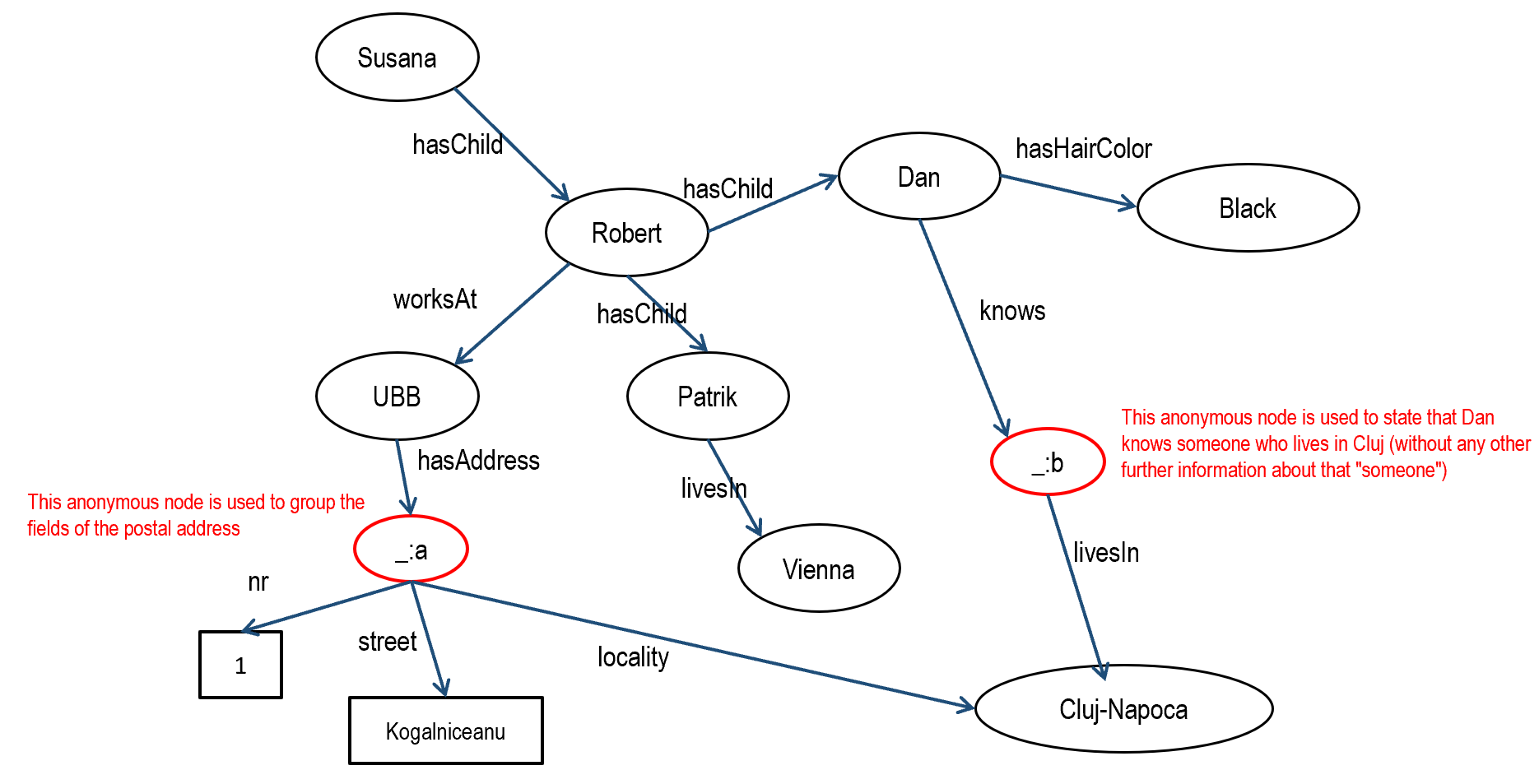
*<<<<:Cluj :connectedTo :Bucuresti>> :distance 320>> :connectionType :air.*

*This is however an experimental feature not yet standardized. It is gradually adopted by most servers, also currently supported in GraphDB. Programming libraries are not yet updated for this, which complicates the parsing of results containing such statements.*

*Use case 5. Expressing ordered lists*

There will be a dedicated subsection for expressing order in RDF statements so we postpone a bit this use case.

To conclude about anonymous nodes, it is important to understand their role in queries. We extend the earlier graph example with a couple of anonymous nodes, illustrating two of the discussed use cases:



We mentioned that underscore IDs are usually irrelevant, because we cannot use them in queries. It's not possible to write queries such as:

SELECT ?x WHERE {\_:b :livesIn ?x} *(where does b live?)*

SELECT ?x WHERE {\_:a :city ?x} *(in which city is address a?)*

However, queries **can pass through** the anonymous nodes:

SELECT ?x WHERE {?x :hasChild/:hasChild/:knows**/**:livesIn :ClujNapoca}

(who has grandchildren who know someone in Cluj?)

SELECT ?field ?value WHERE {:UBB :hasAddress **[**?field ?value**]**}

(retrieve all fields for the post address of UBB)

These examples illustrate how anonymous nodes are useful as placeholders even if their identity is unknown or irrelevant.

## URL addresses

Sometimes a graph contains information about **digital resources that have a fixed Web address**: Web pages, API routes, IoT sensors, email accounts, FTP resources, file paths – anything that has a URL address.

Data about a URL (also called **metadata**) can be stored naturally in a graph:

@prefix : <http://buchmann.ro#> .

<http://buchmann.ro/homepage> :ownedBy :Robert; :authoredBy :Robert; :createdIn :PHP; :keywords "programming","graphs" .

Or, multiple URLs can be attached to a thing (URI):

@prefix : <http://buchmann.ro#> .

:Robert :hasEmail <mailto:robert@buchmann.ro>;

:hasWebsite <http://buchmann.ro/homepage>;

:hasPicture <http://buchmann.ro/pics/myimage.png>;

:hasCV <http://buchmann.ro/homepage#CVsection> .

Notice that URLs are not prefixed. If we want to avoid the repetition of their base address, we can use relative paths between <....>:

@prefix : <http://buchmann.ro#> .

**@base <http://buchmann.ro/> .**

:Robert :hasEmail <mailto:robert@buchmann.ro>;

:hasWebsite **<homepage>**;

:hasPicture **<pics/myimage.png>**;

:hasCV **<homepage#CVsection>** .

*We can have multiple prefixes in the same Turtle file, but only one base address applicable to all relative paths following after it.*

A special URL (actually a relative path) is **<>**, which represents the "current file" (where we write the current graph):

<> a :Graph; :authoredBy :Robert; :version 1.0 .

When we upload this on an RDF server, this special URL will be replaced with the file origin (path) from which it is uploaded.

Although we suggested earlier that a URI may be used in browser as a URL address, it is important to distinguish between URIs and URLs – you should distinguish between writing **statements about a thing** (URI) and **about some digital resource** associated with that thing (URL). In N-triples this distinction is easily seen – e.g. for example, the previous example would be written as:

<http://buchmann.ro#Robert> <http://buchmann.ro#hasEmail> <mailto:robert@buchmann.ro> .

<http://buchmann.ro#Robert> <http://buchmann.ro#hasWebsite> <http://buchmann.ro/homepage> .

<http://buchmann.ro#Robert> <http://buchmann.ro#hasPicture> <http://buchmann.ro/pics/myimage.png> .

<http://buchmann.ro#Robert> <http://buchmann.ro#hasCV> <http://buchmann.ro/homepage#CVsection> .

(the first term is the person, the last terms are its digital resources)

## Expressing order

By default there is no order between the statements of a graph. The following sets of statements are one and the same graph:

Set 1:

@prefix : <http://buchmann.ro#>.

:Romania :hadPresidents :Iliescu, :Basescu, :Iohannis .

Set 2:

@prefix : <http://buchmann.ro#>.

:Romania :hadPresidents :Iohannis, :Iliescu, :Basescu .

Set 3:

@prefix : <http://buchmann.ro#>.

:Romania :hadPresidents :Basescu .

:Romania :hadPresidents :Iliescu .

:Romania :hadPresidents :Iohannis .

This means that queries cannot find out who is the first president, who is the next president after Basescu etc. At most, query results can be ordered syntactically (e.g. alphabetic). Sometimes we want to capture a **semantic order**, given by intended meaning, not by alphabet – e.g. the order of presidencies.

There are several ways of expressing a semantic order:

*Solution 1: add an index property to add numbering to the desired order:*

@prefix : <http://buchmann.ro#>.

:Romania :hadPresidents :Iliescu, :Basescu, :Iohannis .

:Iliescu :index 1.

:Basescu :index 2.

:Iohannis :index 3.

*Solution 2: combine an index property with anonymous nodes (avoids ambiguity regarding the property to which the index refers):*

@prefix : <http://buchmann.ro#>.

:Romania :hadPresidents [:president :Iliescu; :index 1], [:president :Basescu; :index 2], [:president :Iohannis; :index 3] .

*Soluția 3: use an ordered list:*

@prefix : <http://buchmann.ro#>.

:Romania :hadPresidents (:Iliescu :Basescu :Iohannis) .

The preferred solution depends on potential queries and ambiguities:

* Solution 1 seems simple but it will be ambiguous if the same presidents are involved in multiple lists (possibly different orders, each requiring a different index value);
* Solution 2 solves that ambiguity, since **each pair of [ ] generates a unique anonymous node** (other lists will have other anonymous nodes); this is good for read-only lists, but if elements of the list are frequently changed, the renumbering becomes tedious;
* Solution 3 is the most elegant for a human reader, and has several advantages: easier to insert new elements (no need to renumber), easier to find elements by position (the first, the last, the relative next). The disadvantage is that the simple parentheses are a syntactic simplification of a much more complicated graph made of multiple nested anonymous nodes - it looks like this:

:Romania :hadPresidents [**rdf:first** :Iliescu;

**rdf:rest** [**rdf:first** :Basescu;

**rdf:rest** [**rdf:first** :Iohannis;

**rdf:rest rdf:nil**]]].

The standard terms **rdf:first** (current list element), **rdf:rest** (the rest of the list, after the current element), **rdf:nil** (the end of the list) form a **recursive list structure** – every step of the recursion indicates the current element and the next one, until the list is closed with rdf:nil.

When creating the list we can use the simplified syntax (enumeration in round parentheses), but when querying the list we have to be aware of its real structure and to manipulate the nested first/rest connections correctly, especially when we INSERT/DELETE elements. Even with such complications, advantages are numerous:

* Queries can easily navigate the list;
* It can be easily combined with other lists or anonymous nodes to obtain more sophisticated structures, e.g. lists of objects mimicking known JSON structures[[12]](#footnote-12):

@prefix : <http://buchmann.ro#>.

:PopIoan a :Client; :hasOrder ( [:id "p1"; :name "televizor"; :price 100; a :Product] [:id "p2"; :name "telefon"; :price 500; a :Product] ).

* INSERT/DELETE have much better performance than renumbering index values.

## Grouping statements in multiple (named) graphs

We can write multiple graphs in the same file, each having its own graph identifier. This will allow our queries to target a particular data subset. There are multiple ways of specifying in a query the targeted graph ID:

SELECT ?x **FROM :graphID** WHERE {....}

SELECT ?x WHERE {**GRAPH :graphID** {....}}

**WITH :graphID** DELETE/INSERT ....

Moreover, graph IDs are also URIs, so they can be used to make statements about a graph as a whole (typically metadata – who created the graph, when etc.)

The grouping of statements in multiple graphs can be done with the help of accolades (the graph ID is indicated when opening the accolade):

@prefix x: <http://site1.com#> .

@prefix o: <http://site2.com#> .

@prefix : <http://buchmann.ro#> .

**x:Graph1**

**{**

x:John a x:Human;

x:isBrotherOf x:George, x:Anna;

x:fullName "John Smith".

x:Anna a x:Human; x:fullName "Anna Smith".

x:George a x:Human; x:fullName "George Smith".

**}**

**o:Graph2**

**{**

o:Mary o:worksAt [a o:Company; o:locatedIn o:Wien]

**}**

**:Graph3**

**{**

x:Graph1 :author :Robert.

:Robert :contact <mailto:robert@buchmann.ro>.

o:Graph2 :author <http://www.imdb.com>.

**}**

Some important remarks:

* When using accolades, the file is not anymore in Turtle format! A different format named TriG is used (a Turtle file cannot contain multiple graphs according to the standard);
* Notice how the third graph contains information ABOUT the other two graphs. This is possible because graph IDs are also URIs, regular terms to be used in statements!

## Standard terms

We've mentioned several times the existence of standard terminologies (**vocabularies**) that provide terms for certain standard declarations. Standard terms have several advantages:

* Our graphs will be understood by others (if we care about that);
* Some libraries/applications rely on those standard terms to provide certain features (e.g. Machine Reasoning).

The most well-known terminologies are:

* **RDFS:** provides terms for some standard declarations. In this tutorial we used:
  + **a** (or **rdf:type**) – to declare WHAT are the things mentioned in a graph (their entity type)
  + **rdfs:label** - to attach string labels (displayable in front-end) to a URI node
  + **rdfs:member** – to decompose a complex thing in components
  + **rdfs:subPropertyOf** – to declare specialized variants of a property (e.g. *parent*, *uncle* as specific variants of being a *relative*)
  + **rdfs:subClassOf** – to declare subtypes of an entity type (*Student* and *Employee* as subtypes of *Person*)
* **OWL:** also provides terms for some standard declarations. In this tutorial we used:
  + **owl:sameAs** – to declare that two URIs identify the same thing
  + **owl:inverseOf** – to say that two properties have the same meaning, but inverse directions (*hasChild* is the inverse for *hasParent*)
* **Schema.org:** provides terms that are recognized by major search engines, for the kinds of things that are most often searched on the Web (persons, places, events, products etc.). In this tutorial we used:
  + **schema:parent** –the relationship between children and their parents (instead of our own term *daughterOf*)
  + **schema:Person** – identifier for the type Person
* **XSD**: provides identifiers for datatypes (xsd:date, xsd:boolean etc.)

All standard terminologies have their own namespaces that can be found at http://prefix.cc. Their standard prefix declarations are:

@prefix schema: <http://schema.org/> .

@prefix owl: <http://www.w3.org/2002/07/owl#> .

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

1. We could do something similar for Dan, since he has a brother in Vienna, but that would require additional rules or axioms – we keep this introductory example minimal, i.e. applicable only to descendants. [↑](#footnote-ref-1)
2. See <https://developers.google.com/search/docs/guides/intro-structured-data> [↑](#footnote-ref-2)
3. It includes .NET libraries for processing RDF graphs [↑](#footnote-ref-3)
4. See https://marketplace.visualstudio.com/items?itemName=stardog-union.stardog-rdf-grammars&ssr=false#overview [↑](#footnote-ref-4)
5. The *owl:inverseOf* term is provided by the OWL standard - which is a collection of terms for specifying in a standard way (i.e. interpretable by machines) how relationships are related to one another (among other things) [↑](#footnote-ref-5)
6. Some authors call them IRI, to emphasize their internationalization [↑](#footnote-ref-6)
7. The domain address owner can decide to ignore it, depending if they care or not about the widespread adoption of those URIs [↑](#footnote-ref-7)
8. There are also some public sources for IDs (of persons, locations etc.). For example DBPedia and Wikidata provide public graphs about all things that have Wikipedia pages – therefore they also provide public IDs for those things (of course, only things famous enough to have a Wikipedia page). [↑](#footnote-ref-8)
9. https://www.w3.org/TR/xmlschema-2/type-hierarchy.gif [↑](#footnote-ref-9)
10. The space between 20 and the final dot is often necessary, to avoid the situation when a parser considers that there's a decimal number without decimals. [↑](#footnote-ref-10)
11. https://graphdb.ontotext.com/documentation/10.0/devhub/rdf-sparql-star.html [↑](#footnote-ref-11)
12. Obviously, it's more intuitive to store in the graph JSON strings directly (as simple string values of some property), thus leaving the responsibility of navigating JSON structures to the client app. We point however, that those structures can also be created from combinations of RDF lists and anonymous nodes, thus making them processable directly by graph queries. [↑](#footnote-ref-12)