Semantic Technology:

Write operations +

Machine Reasoning (axiom-based)

**Objective:**

* To learn **Insert/Delete/Update** operations in SPARQL
* To learn **axioms** (how to build a reasoning schema) – the alternative approach to query-based Machine Reasoning

**Prerequisite:**

* You should be familiar with the queries that were discussed in the previous tutorials (SELECT, DESCRIBE, CONSTRUCT/INSERT WHERE)
* For comparability, you should be familiar with the previously presented Machine Reasoning approach (query-based)
* We use the same graph database as in the previous tutorial, but it has to include the latest changes (after the Machine Reasoning rules executed in the previous tutorial). We replicate the dataset here for your convenience

@prefix : <http://buchmann.ro#> .

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

:mymoviegraph

{

#directors

:JamesCameron   a :Director, :WellConnectedArtist;          #types

                :workedWith :SamWorth, :Arnold, :LindaH;    #relationships

                :directed :Avatar, :Terminator;

                :hasName "James Cameron" .                  #data

:JohnMcT        a :Director, :AmericanDirector;

                :workedWith :Arnold, :ShBlack;

                :directed :Predator, :DieHard;

                :hasName "John McTiernan";

                :birthInfo  [:birthDate "1951-01-08"^^xsd:date;

                            :birthPlace :SUA].

:McG            a :Director;

                :workedWith :SamWorth;

                :directed \_:somemovie;

                :hasName "Joseph McGinty";

                :hasNickname "McG" .

:TimBurton      a :Director, :AmericanDirector;

                :workedWith :JackNicholson;

                :birthInfo [:birthPlace :SUA];

                :birthPlace :SUA;       #we did not remove yet the direct relation!

                :directed :Batman;

                :hasName "Tim Burton" .

#actors (and an actor-director)

:SamWorth       a :Actor;

                :playedIn :Avatar, \_:somemovie, :Rogue;

                :playedTheRole [:asCharacter :JakeSully;

                                :inMovie :Avatar];

                :hasName "Sam Worthington" .

:Arnold         a :Actor, :WellConnectedArtist;

                :workedWith :LindaH, :ShBlack;

                :playedIn :Terminator, :Predator;

                :wasGovernorOf :California;

                :hasName "Arnold Schwarzenegger";

                :birthInfo [:birthDate "1947-07-10"^^xsd:date;

                            :birthPlace :Austria].

:LindaH         a :Actor;

                :workedWith :Arnold;

                :playedIn :Terminator;

                :playedTheRole [:asCharacter :SarahConnor;

                                :inMovie :Terminator];

                :hasName "Linda Hamilton" .

:ShBlack        a :Director, :Actor;

                :workedWith :Arnold;

                :directed :KKBB;

                :playedIn :Predator;

                :hasName "Shane Black" .

:JackNicholson  a :Actor;

                :hasName "Jack Nicholson";

                :playedIn :Batman .

#characters (untyped!)

:JakeSully      :hasName "Jake Sully" .

:SarahConnor    :hasName "Sarah Connor" .

#movies

:Avatar         a :Movie;

                :hasTitle "Avatar";

                :hasBudget 237000000 .

:Terminator     a :Movie;

                :hasTitle "Terminator";

                :hasBudget 6400000 .

:Predator       a :Movie;

                :hasTitle "Predator";

                :hasBudget 18000000 .

:DieHard        a :Movie;

                :hasTitle "Die Hard" .

:Rogue          a :Movie;

                :hasTitle "Rogue" .

:KKBB           a :Movie;

                :hasTitle "Kiss Kiss Bang Bang";

                :hasBudget 15000000 .

:Batman         a :Movie;

                :hasTitle "Batman" .

\_:somemovie     a :Movie;

:hasBudget 200000000 .

#countries (untyped)

:SUA            :hasName "United States of America"@en, "Etats Unis d'Amerique"@fr .

:California     :hasName "California"@en, "Californie"@fr .

:Austria        :hasName "Austria"@en, "Autriche"@fr .

}

## Insert stuff, delete stuff, update stuff

**Insertions (and a few introductory axioms)**

We saw how the operation INSERT ... WHERE ... executes a deductive reasoning rule (i.e. to generate new information from existing information – shortcut relationships, new types, new nodes).

There is also a distinct operation INSERT DATA which acts as traditional insertion – i.e. it adds new information regardless of what already exists in the graph.

Although you can insert any set of statements, the following examples will insert **axioms** - a special kind of statements built with the help of standard terminologies. The terms to be exemplified in the following are:

* **rdfs:subPropertyOf** to indicate that several properties have related meanings
* **rdfs:subClassOf** to unify related entity types in a supertype (superclass)
* **owl:SymmetricProperty** to indicate that some properties are bidirectional
* **rdfs:label** to attach the string label for an identifier (URI)

To understand the benefits of such standard terms, try the following examples:

*A. Specialized properties of the same property*. We want to retrieve all string labels – in the current version of the graph an OR must be applied because we have multiple labelling properties (hasName for people and countries, hasTitle for movies, hasNickname for some people):

SELECT ?ID ?labelValue

{

?ID :hasName|:hasNickname|:hasTitle ?labelValue

}

In order to declare that these three properties have related meaning – more precisely, all of them are specific versions of labelling - we will insert the following axioms:

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

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

INSERT DATA

{

:hasName rdfs:subPropertyOf rdfs:label.

:hasNickname rdfs:subPropertyOf rdfs:label.

:hasTitle rdfs:subPropertyOf rdfs:label.

}

Now, whenever we want to obtain all labels, we don't need to know what labelling properties were used, we only need to know that all of them are specific version of the standard term rdfs:label:

SELECT ?ID ?labelValue

{

?ID ?prop ?labelValue.

**?prop rdfs:subPropertyOf rdfs:label**

}

*In GraphDB there is an additional benefit – if we activate the inference ruleset feature, we will be able to find all labels with an even simpler query:*

*SELECT ?ID ?labelValue*

*{*

*?ID* ***rdfs:label*** *?labelValue*

*}*

*This is possible because the inference feature automatically generates the rdfs:label property everywhere where more specific versions are found, based on the above axioms! This is another type of Machine Reasoning which, instead of taking the form of INSERT queries, is based on axioms – however, it only works on servers that have inference support (in GraphDB this is* ***an optional feature that can be activated when the database is created*** *and will be covered in a future section!).*

*B. Unifying entity types*. Let's assume we want to retrieve all persons – they are either actors, or directors, so we need something similar to a logical OR to collect all possibilities. This can be expressed as follows:

SELECT DISTINCT ?personID

{

?personID a ?entityType

FILTER(?entityType IN (:Actor,:Director))

}

Now add axioms to unify all kinds of persons under a single supertype:

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

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

INSERT DATA

{

:Actor rdfs:subClassOf :Person.

:Director rdfs:subClassOf :Person.

}

We can now obtain easily all persons without knowing all kinds of persons mentioned in the graph:

SELECT ?personID

{

?personID **a/rdfs:subClassOf** :Person.

}

*Just like in the previous case, this mechanism is more powerful if the* ***inference ruleset*** *feature is activated. We would be able to get all persons directly, because the Person type would be automatically attached to all instances:*

*SELECT ?personID*

*{*

*?personID a :Person.*

*}*

*C. Declaring bidirectional relationships.* This is useful when a relationships is stored in only one direction, but it should be considered bidirectional (mutual). Let's try to obtain everyone with whom Arnold has a mutual relationship, by looking at those relationships for which a reversed one is also stored:

SELECT ?node

{

:Arnold ?relation ?node

FILTER EXISTS {?node ?relation :Arnold}

}

This will find those individuals for whom :workedWith was generated in both directions (i.e. the actors), but will miss the directors, for which the :workedWith was only generated from director to actor.

We have better control on this situation if we declare exactly which relationships should be considered mutual:

INSERT DATA

{

:workedWith a owl:SymmetricProperty.

}

Now we don't care in how many directions a relationship is stored – if it is declared as bidirectional, we know we have to look in both directions (we'll find collaborators even if they have a single link to Arnold):

SELECT ?relation ?node

{

**?relation a owl:SymmetricProperty**

{:Arnold ?relation ?node.}

UNION

{?node ?relation :Arnold.}

}

*Again, the activation of the* ***inference ruleset*** *feature would automatically generate in both directions any relationship declared as symmetric, wherever it is used. This would make the social relationships uniform (both directions everywhere), thus simplifying the above query – all collaborators would be found by looking in a single direction:*

*SELECT ?node*

*{*

*:Arnold :workedWith ?node.*

*}*

*To conclude about bidirectional relationships: they should only be stored in one direction and declared as having symmetric (mutual) meaning:*

* *if the inference feature is available on the server, it will take care about generating those edges in both directions (and queries will only have to navigate them in one direction);*
* *if the inference feature is not available, our queries should check if the symmetric nature is declared and, if yes, should navigate them in both directions.*

*This returns us to the question: how would you modify the following reasoning rule, to make sure that actor-to-actor collaborations are only generated in one direction?*

*INSERT {?x :workedWith ?y }*

*WHERE*

*{*

*?x :playedIn ?film.*

*?y :playedIn ?film.*

*FILTER (?x!=?y)*

*}*

**Deletions**

When exercising deletions, make sure that you keep a copy of the full database to be quickly reloaded in case you mistype something – there's a risk of accidental deletions!

Just like INSERT, DELETE has two versions:

* DELETE DATA ... where we say exactly which statements should be removed
* DELETE... WHERE... where we look for a pattern and then we delete something that is connected to that pattern (i.e. it can be considered a form of reasoning, when information is removed based on logical deduction).

Just like CONSTRUCT, the DELETE WHERE variant has again two versions:

* DELETE {....} WHERE {....} - the stuff we search can be different from the stuff we delete
* DELETE WHERE {...} – the stuff we delete is exactly the stuff we search for, so there's not reason to replicate it in two patterns

We start by deleting all labels that are written in French:

DELETE {**?x :hasName ?name**}

WHERE

{

**?x :hasName ?name.**

FILTER(lang(?name)="fr")

}

One important difference from INSERT WHERE is that, here, the statements to be deleted must be a subset of the pattern to be searched. More precisely, the following will not work because the WHERE clause is not a pattern that fully covers the deleted pattern:

DELETE {**?x :hasName ?name**}

WHERE

{

FILTER(lang(?name)="fr")

}

Check the success of deletion by requesting all names (now countries will only have English names).

Next, we want to delete those birthplace relations that link a person directly to its place of birth. Remember that in most cases there is also an anonymous node between the person and the birth place – but in the case of Tim Burton there was also a direct connection.

We take advantage of the previous axioms to detect all persons:

DELETE {**?x :birthPlace ?place**}

WHERE

{

**?x** a/rdfs:subClassOf :Person; **:birthPlace ?place**

}

*Notice again the bolded part, which makes sure that the deleted pattern is a subset of the searched pattern. It will not work like this:*

*DELETE {?x :birthPlace ?place}*

*WHERE*

*{*

*?x a/rdfs:subClassOf :Person.*

*}*

Next, we delete all statements that involve nicknames – we have a nickname declared for someone, and we have :hasNickname declared as a labelling property. We want to get rid of both – we search and delete exactly what we find, so there's no reason to use two patterns:

DELETE WHERE

{

?x :hasNickname ?y.

:hasNickname ?p ?o.

}

*Reminder: CONSTRUCT also has such a simplified (single pattern) version, when we want to extract exactly the subgraph that we searched for, without generating new information.*

Finally, the simplest form of deletion is when we explicitly write the statements to be removed. The next example will delete the fact that Arnold was the governor of California:

DELETE DATA

{

:Arnold :wasGovernorOf :California

}

**Updates**

There is no UPDATE command in the current version of SPARQL. Updates will have to combine a DELETE with an INSERT.

The first example will be quite complex – we will substitute an anonymous node with an ID, plus a string label attached to that ID. For this, we target the node \_:somemovie and replace it with the ID :TerminatorSalvation plus the label "Terminator Salvation". The main challenge with such a substitution is that

* we need to preserve the existing connections of the replaced node
* we also need to use those connections to properly select the node, because we cannot select anonymous nodes by their name[[1]](#footnote-1)! so, we don't replace the \_:somemovie node, we replace THAT node that is connected to McG and SamWorth (their collaboration is the most precise information we have about this node)

To avoid mistakes, it is useful to think about a replacement in the following steps:

**Step1. First we test a CONSTRUCT that builds the new connections**. Because CONSTRUCT does not save the generated information, we can play around and correct mistakes in time.

CONSTRUCT

{

:McG :directed **?newNode**.

:SamWorth :playedIn **?newNode**.

**?newNode** :hasTitle **?title**; :hasBudget ?budget.

}

WHERE

{

:McG :directed ?oldNode.

:SamWorth :playedIn ?oldNode.

?oldNode :hasBudget ?budget.

**BIND(:TerminatorSalvation AS ?newNode)**

**BIND("Terminator Salvation" AS ?title)**

}

Notice the BIND clauses that create the new URI and its title. In an application's code the BIND clauses would be injected in the query with data collected from a front-end. Execute this CONSTRUCT to see exactly what will be inserted.

**Step2. If the CONSTRUCT runs well, we turn it into an INSERT**.

**Step3. We add at the very beginning the deleted pattern. Remember that it must be a subset of the searched pattern, or exactly the same:**

DELETE {

:McG :directed ?oldNode.

:SamWorth :playedIn ?oldNode.

?oldNode :hasBudget ?budget.

}

**The final form of the query will be**:

DELETE

{

:McG :directed ?oldNode.

:SamWorth :playedIn ?oldNode.

?oldNode :hasBudget ?budget.

}

INSERT

{

:McG :directed **?newNode**.

:SamWorth :playedIn **?newNode**.

**?newNode** :hasTitle **?title**; :hasBudget ?budget.

}

WHERE

{

:McG :directed ?oldNode.

:SamWorth :playedIn ?oldNode.

?oldNode :hasBudget ?budget.

**BIND(:TerminatorSalvation AS ?newNode)**

**BIND("Terminator Salvation" AS ?title)**

}

Even if the pattern order is a bit strange, the real execution order is the following:

* first, WHERE will be executed to search,
* then, INSERT to create and connect the new stuff to the searched pattern,
* finally, DELETE to eliminate the old stuff.

We can test the success of the update with a DESCRIBE that gives us everything that is known about the nodes connected to both McG and SamWorth.

DESCRIBE ?node

WHERE

{

:McG :directed ?node.

:SamWorth :playedIn ?node.

}

This operation is not specific to anonymous nodes – it can be performed on any node that must be replaced while preserving all its connections (and perhaps adding some new ones). The reason why it looks rather complex is that we also ensure what SQL DBs call "cascading" – i.e., we don't only replace an identifier, we also propagate this replacement throughout the graph!

*If we replace an ID that was also used in other graph databases, their admins should be informed or some announcement should be posted – remember that URI nodes are global identifiers, valid across the entire Web in order to facilitate the linking of different graphs hosted on different servers (i.e. changing a URI is similar to breaking a URL link).*

*For this reason, the shape of a URI should be carefully planned and should be stable. Let's assume that we have a URI for a thing and we find out that the thing already received a URI in another organization/server. Instead of replacing a URI, the advised technique is to declare equivalence that can be easily navigated by queries. This is done in a standard way with the help of the* ***owl:sameAs*** *relation:*

:URI1 owl:sameAs :URI2.

*This cannot be done, however, with anonymous nodes – in that case replacement is preferred.*

Not all updates are this complex, because cascading is not always needed (or, it can be simpler). Other examples are provided in the following:

* **update an attribute value** – replace the budget of the movie KKBB with 20000000:

DELETE

{:KKBB :hasBudget ?currentBudget}

INSERT

{:KKBB :hasBudget 20000000}

WHERE

{:KKBB :hasBudget ?currentBudget}

* **replace a property** wherever it was used[[2]](#footnote-2):

DELETE

{?x :hasBudget ?y}

INSERT

{?x :budget ?y}

WHERE

{?x :hasBudget ?y}

* **replace a URI without cascading** – the next example replaces the birth place USA with Albany but only for John McT (Tim Burton's birth place will remain USA); all birth places are attached through the intermediate birthInfo relationship, so that's where we apply the replacement:

DELETE

{?anonym :birthPlace ?place}

INSERT

{?anonym :birthPlace :Albany}

WHERE

{:JohnMcT :birthInfo ?anonym.?anonym :birthPlace ?place}

**The WHERE pattern in inserts/deletes/updates**

Although the official documentation does not specify this insight, the syntax specification implies a key information often missed by users: in all queries having two accolade patterns {...} the pattern of WHERE can benefit from any of the clauses exemplified for SELECT – i.e. OPTIONAL, UNION, MINUS, FILTER, SELECT subqueries, SERVICE.

We demonstrate this in a DELETE WHERE operation: we remove the title and budget for the movie having maximum budget – so a subquery is necessary to first find out which is the maximum budget:

DELETE

{?film :hasTitle ?title; :hasBudget **?maxbudget**}

WHERE

{

?film :hasTitle ?title; :hasBudget **?maxbudget**

{

SELECT (MAX(?budget) AS **?maxbudget**)

WHERE {?film :hasBudget ?budget}

}

}

With a SELECT we can see that for Avatar the title and budget were removed:

SELECT ?film ?title ?budget

WHERE

{

?film a :Movie

OPTIONAL {?film :hasTitle ?title}

OPTIONAL {?film :hasBudget ?budget}

}

What if we want to remove titles and budgets for all movies? We might be tempted to execute:

DELETE WHERE

{

?film :hasTitle ?title; :hasBudget ?budget; a :Movie

}

Repeat the previous SELECT and you'll see that it only removed them for movies having **both** title and budget. Because either title or budget is missing in some cases, we should also use OPTIONAL in DELETE:

DELETE

{?film :hasTitle ?title; :hasBudget ?budget}

WHERE

{

?film a :Movie.

OPTIONAL {?film :hasTitle ?title}

OPTIONAL {?film :hasBudget ?budget}

}

Repeat the previous SELECT to see that now all titles/budgets disappeared.

It is often the case that a deletion or reasoning query must first perform some calculations so the WHERE clause is flexible to allow for anything that can also be done in a SELECT.

We will not continue with other examples – but remember that all clauses exemplified for SELECT are applicable to INSERT/CONSTRUCT/DELETE only if we use these with two patterns! They don't work for the simplified single-pattern version:

DELETE WHERE {....} (here, the stuff to be deleted is the same as the stuff to be searched)

CONSTRUCT WHERE {...} (here, the stuff to be searched is extracted as a subgraph)

Why? Because the single pattern acts as both **the searched pattern** and **the deleted/generated pattern**, and those SELECT clauses we mentioned are selection modifiers, therefore valid only in the searched pattern.

Beginners make a lot of errors that hit this wall!

## Back to Machine Reasoning

We previously established that two reasoning approaches are available for RDF graphs:

1. **Query-based reasoning** is supported by RDF servers by default, through the query language:
   * CONSTRUCT ... WHERE for "reasoning on-the-fly" (results are only useful to the client who executed the query)
   * INSERT ... WHERE ... for "reasoning with writing" (results are saved permanently and available to all future queries)
2. **Axiom-based reasoning** is not supported by default, but some vendors provide optional support ("inference engine", "inference ruleset") that can be activated in GraphDB when the database is created.
   * with this type of reasoning, we build **axioms** – statements written with standard terms that are "understood" by the inference engine; those terms are provided by standards such as RDFS and OWL

There are some major differences between the two types of reasoning:

* **Axioms are stored in the database**, they become part of the graph; this means that the inference engine will track all updates and will reapply the reasoning everytime something is removed or added in the database. In comparison, query-based reasoning only generates stuff based on the information available when the query is executed – *later updates and deletes will require reexecution of reasoning queries or even removal of information that was generated in the past*;
* Since they are stored in the database, **axioms themselves can be queried**, used by query filters etc (they are still RDF statements, only written with specific terms);
* By creating axioms for every property and every entity type used in the graph, **we actually build a graph schema** (also called "ontology", "vocabulary" if it's shared by a community agreeing to use the same terms in their graphs);
* Finally, the **activation of the inference engine comes with a performance cost** (that's why it's an optional feature). It's the responsibility of the database Admin to decide if benefits exceed the performance loss.

To benefit from this feature, we activate it when we create a repository[[3]](#footnote-3). In the Create repository screen, set the following options:

* *RepositoryID*: axiomsExample
* *Repository title*: axiomsExample
* *Rule-set*: OWL2-RL
* *Base URL*: remove the suggested URI to be able to work with your own prefixes/namespaces
* Check the box *Check for inconsistencies* to enable contradiction detection (will reject contradicting statements, however the contradictions must still be defined as axioms)

The key option here is *Rule-set*. In the past we set it to *No inference*, when we used only query-based reasoning. With axioms, we have to choose from several inference engines/rulesets – the weakest is RDFS (it only recognizes RDFS axioms), the most powerful is OWL2-RL[[4]](#footnote-4), however it comes with the biggest cost on performance. We choose anyway the most powerful one, to be able to show a variety of axioms – it also covers everything that RDFS can do.

We activated the inference ruleset/engine when the database was created. **It's also possible to activate it later, or to switch the ruleset**, but there's a risk of messing up the data – however there's no risk if we first had a no-inference database and later we decide to activate an inference ruleset. The activation can be done with two **configuration queries** (GraphDB configurations are also stored as RDF and can be manipulated by such queries!):

PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA { [ sys:addRuleset "owl2-rl"] }

(this will activate the OWL2-RL ruleset on the current, already populated database)

PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA { [ sys:reinfer [ ] ] }

(this will execute all axioms - in case they've been added before activating the ruleset)

Notice that certain terms, under the sys: namespace, are reserved by GraphDB to trigger certain reconfigurations which are mimicked as SPARQL insertions (i.e. they access an invisible graph of configurations). Additional options for manipulating the ruleset after the database was created can be consulted at

https://graphdb.ontotext.com/documentation/standard/reasoning.html#how-to-s

Now connect to the database and go to the SPARQL screen for queries. Make sure that on the right side of the query box the option *Include inferred data in results* is on (otherwise, queries will ignore the statements generated by reasoning).

Run the generic query:

SELECT \* WHERE {?x ?y ?z}

Although the database is empty, you should see more than 1000 statements! These are some **default axioms** applicable to every statement and every URI to be inserted. You can notice that the default axioms are built with terms from the RDFS and OWL terminologies.

Upload the following statement (use the same Import-RDF screen as before):

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

:JamesCameron :directed :Terminator.

Run the generic query again. You will notice that the total number of statements increased (probably by 10-12 statements, depending on the version), **although you only uploaded a single statement**! The rest was generated by the inference engine – you should understand now where the performance cost comes from (this will happen for every inserted/imported statement)!

You can see the newly generated stuff if, for example, you query for a description of :directed

PREFIX : <http://buchmann.ro#>

DESCRIBE :directed

The generated statements are some default types that every property gets:

* **rdf:Property** is the default type for all properties (i.e. all terms used in the middle of a statement);
* **rdfs:Resource** is the default type for all URIs used in any graph; **owl:Thing** has the same meaning (introduced by the OWL terminology) – all URIs will receive these default types;

in the future others may appear – e.g. when we declare or generate the type of an individual, the type itself will receive a default type called **rdfs:Class** (i.e. this represents "the type of a type"; closely related is **owl:Restriction** which is also a type for types, but only for those types to be generated by some specific axioms (those using the term **owl:onProperty**, acting as property restrictions).

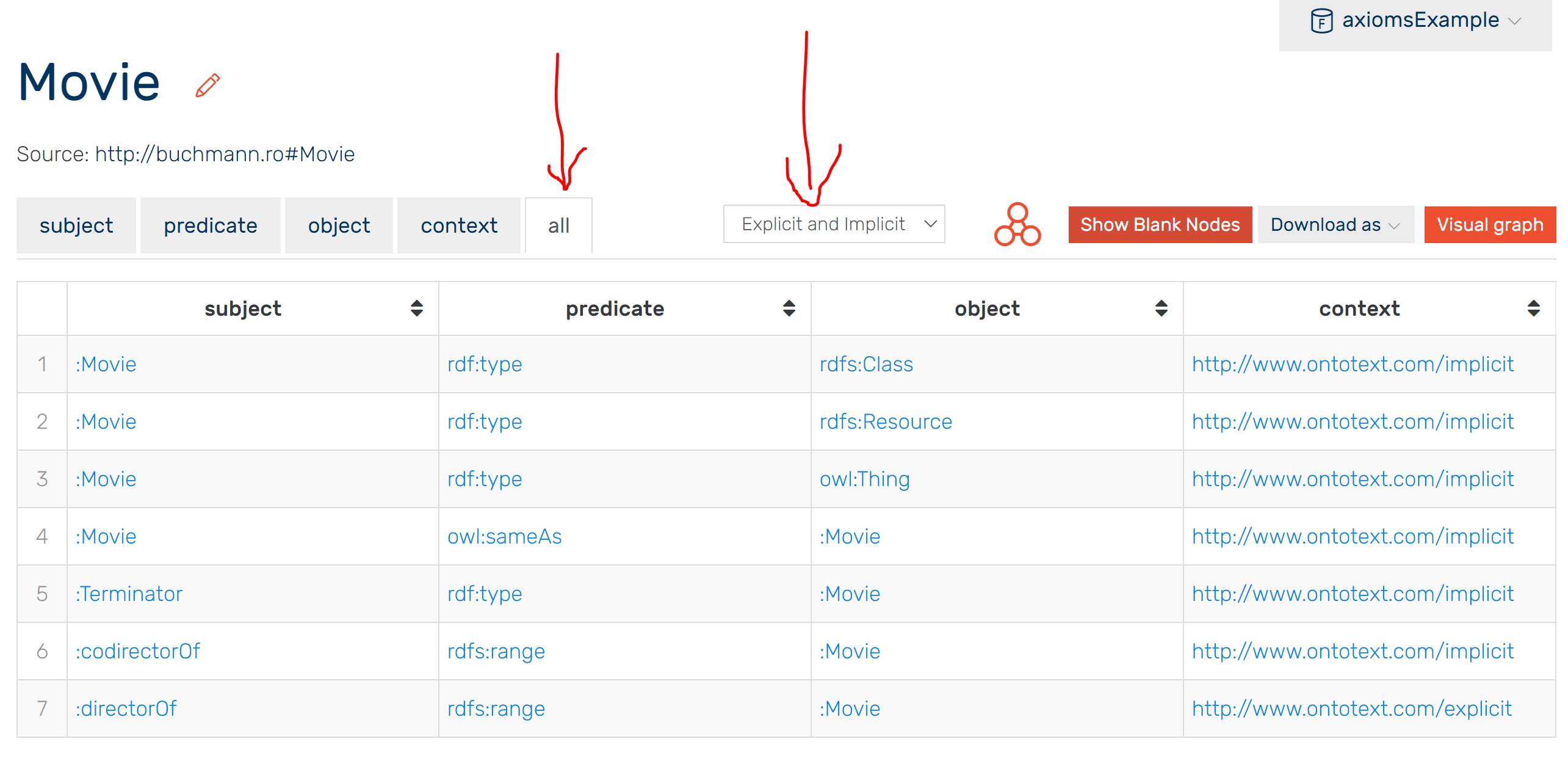
We will generally not need to use such highly abstract terms, but they can be useful in some generalist queries – e.g. to get a list of all IDs mentioned in the database (regardless if they are individuals, types or properties):

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

SELECT \* WHERE {?x a rdfs:Resource}

Before we start writing our own axioms, a final warning: when working with the inference engine activated, we have to pay attention to some options:

* on the right-side of the query box, make sure that the button *Include inferred data* is ON (otherwise the query will avoid the generated parts of the graph) – this should be on by default;
* after query results are displayed, if we want to navigate results by clicking URIs, we have to make sure that all results are visible:
  + there is an *all* Tab to make sure that we get all statements (and not only those where the clicked URI is a subject);
  + there is a drop-down where we should select *Explicit and Implicit* to make sure that we see both the manually added statements (Explicit) and the generated statements (Implicit, actually stored as a separate graph whose ID is visible in the *context* column).



## Reasoning with the RDFS inference ruleset

RDFS is a terminology (i.e. dictionary of terms) that provides only a minimum set of terms to be used in axioms. It is the minimum necessary to create the simplest graph schema, with the minimum performance cost. Although we activated the OWL2-RL inference engine in the previous section, it also covers RDFS (RDFS is a subset of OWL) so we don't have to create different databases to exercise with the two rulesets.

**Domain and Range axioms**

Our database now only has one statement, previously inserted: *James Cameron directed Terminator*. The schema for this statement can be built by adding the following axioms[[5]](#footnote-5):

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

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

:directed **rdfs:domain** :Director; **rdfs:range** :Movie.

Notice the difference between these new statements (which are **axioms**) and the previous upload (which was a **fact**). Axioms are usually statements about a property or an entity type (not an individual) – since properties and types are also URIs, we can make statements about them just like about any ID.

This particular example says that the *:directed* relation connects directors to movies. To make sure that this is true, the reasoning engine will automatically generate the types Director and Movie for all things connected by this relation! Check this by asking what is Terminator:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:Terminator a ?z}

If we want to avoid the default types (resource/thing) we can filter them out based on the fact that they belong to the official W3C domain address:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE

{

:Terminator a ?z

FILTER (!STRSTARTS(STR(?z),"http://www.w3.org"))

}

We can also ask what is James Cameron:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:JamesCameron a ?z}

Notice that this kind of reasoning is the reverse of data validation found in SQL databases:

* With **validation**, individuals are rejected if they don't comply with their type – "if individual X does not have these fields, it cannot be added to the table";
* With **reasoning**, their types are derived from information available about the individuals – "if X has these fields/properties, it becomes a Director";

We are familiar with validation from SQL databases (where the schema has a validation role) but here the **graph schema fulfils multiple roles: documentation, reasoning** (what we just saw),... **and even validation** (if we add *SHACL validation rules* on top of these axioms[[6]](#footnote-6)).

Now upload some additional statements:

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

:TimBurton :directed :Batman.

:StevenSpielberg :directed :ET.

The domain/range axioms remain active (until you delete them) and will keep generating types every time we use the property *:directed*. You can now retrieve a list of all directors (similar for movies):

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x a :Director}

Axioms are created once and will trigger their reasoning for every modification to the database (unlike INSERT/CONSTRUCT which act only on the information available when the query is executed). This means that the axioms will always guarantee a **logically consistent database**.

Domain and range axioms are the "glue" that brings together types and properties in the graph schema.

Because some properties can have data values, it means that the range can also be a data type, e.g.

* For statements such as...

:John :hasAge 42.

* The schema-level axiom would be:

:hasAge rdfs:domain :Person; rdfs:range **xsd:integer**.

In such cases however, reasoning will not be performed on data values – so there's no danger that from saying...

:John :hasAge "fourty".

...the server would generate that "fourty" is an integer number. In short, data type ranges are only used for documentation purpose. Reasoning only works on entity types.

**Subclass axioms**

Upload a new axiom:

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

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

:Director **rdfs:subClassOf** :Artist.

This declares inclusion between two types, translating to: "any director must be considered an artist" (or, in set terms, directors are a subset of artists). Now if we ask what is JamesCameron we get richer answers - not only Director, but also Artist:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:JamesCameron a ?z}

Multiple subClass axioms will build a **hierarchy of types** ("taxonomy"). A taxonomy generates multiple types for an individual by going up the type hierarchy (Director -> Artist -> resource/thing). However it does not work downwards - you can check this by adding a new subtype:

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

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

:AmericanDirector rdfs:subClassOf :Director.

rdfs:subClassOf

(generated)

rdfs:subClassOf

a

(generated)

a

(generated)

rdfs:subClassOf

rdfs:subClassOf

(generated)

a

(explicit)

rdfs:subClassOf

a

Run the following query:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:JamesCameron a ?z}

The results did not change, AmericanDirector is NOT part of the answer (any american director is a director but not vice versa)! Basic logic dictates that an individual only receives types that subsume the one it currently has.

We've mentioned in a previous section that **subclass axioms can be used to unify types** – e.g. MovieDirector and TheaterDirector as specialized categories of Director. This has an important implication on domain/range axioms: if we want to declare multiple domains and ranges on the same relation, we might be tempted to just enumerate them:

:directed rdfs:domain :MovieDirector, :TheaterDirector.

This, however, is a logical fallacy – what we're actually saying is that all subjects of the *:directed* relationship must become simultaneously movie directors and theater directors (the enumeration of multiple domains/ranges defines an intersection!) The correct way is to unify the multiple domains (or ranges) in a supertype and use that supertype as a single domain/range:

:MovieDirector rdfs:subClassOf :Director.

:TheaterDirector rdfs:subClassOf :Director.

:directed rdfs:domain :Director.

This says that the *:directed* relation can be used for any of those director subtypes – so this relation alone is not sufficient to say which kind of director are we talking about. More precise properties or more advanced axioms would be needed to detect the distinction!

**Subproperty axioms**

Upload a new axiom:

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

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

:directed **rdfs:subPropertyOf** :created.

This axiom generates a similar hierarchy, but for properties. It will be interpreted as a logical implication: "If x is director of y, then x created y". We can check that a new relation was generated between the directors and their movies:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x :created ?y}

The superproperty is a more general version of its subproperties, and will be generated between the same pair of instances – remember also the case of multiple labelling properties (hasName, hasTitle, hasNickname) declared as particular versions of **rdfs:label**.

The propagation of properties also works only upwards in the hierarchy (i.e. creating someting does not imply directing something). However some additional stuff is also generated at schema level:

* subproperties inherit the domain and range from their superproperties;
* subproperties receive as domain/range all the superclasses (supertypes) of their current domain/range.

You can already check the second point - :directed now has multiple domains (so its subjects will get multiple types, which is fine in this case since one is a subtype):

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

SELECT \* WHERE {:directed rdfs:domain ?x}

To check the first point, add a subproperty of *:directed* – what relation has similar meaning but is more specific? Let's say *:codirected*:

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

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

:codirected rdfs:subPropertyOf :directed.

You can check that this axiom did not generate new relations:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x :codirected ?y}

However, the new property inherited a domain and a range:

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

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:codirected rdfs:domain ?d; rdfs:range ?r}

**Removing axioms**

Delete the axiom that defined the original domain for directed:

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

PREFIX : <http://buchmann.ro#>

DELETE DATA {:directed rdfs:domain :Director}

Now if you ask what is James Cameron you will notice that all the generated types were removed together with the axiom (only the default types remain):

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:JamesCameron a ?x}

This also happened for all the other directors - actually we have no more directors:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x a :Director}

As we stressed earlier, all generated information was retracted! The database is adjusted every time something changes, to keep it logically consistent. In this case, the axiom was removed together with everything that was reasoned based on it. This is not possible with query-based reasoning, because we lose track of what was generated.

We mentioned earlier that such axioms form a graph schema (ontology, vocabulary). To have a graph schema, we need to:

* declare all entity types used in the graph, and optionally unify them in a common hierarchy with **rdfs:subClassOf**
* declare all properties used in the graph, and where applicable connect them in a common hierarchy with **rdfs:subPropertyOf**
* indicate for each property what are the things it connects with **rdfs:domain** and **rdfs:range** (data properties will have as range a data type – xsd:integer, xsd:string etc.!)
* optionally, some comments for documentation purposes (**rdfs:comment** is the property to attach a string comment to any URI).

If we don't activate the inference engine, these should be added manually. With an inference engine, we just saw that some parts will be generated automatically.

## Reasoning with the OWL inference ruleset

The previous section presented the minimal reasoning patterns and the minimal constructs that can build an RDF graph schema. The properties and types defined in a schema can be further refined by OWL axioms that enable more powerful reasoning.

**More advanced reasoning based on properties**

Upload the following axioms:

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

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

:hasFriend a **owl:SymmetricProperty**.

:hasRelative a **owl:TransitiveProperty**.

:hasHusband **owl:inverseOf** :hasWife.

Their interpretation is the following:

* hasFriend should be interpreted as bidirectional (it will generate this relation in both directions)
* IF x hasRelative y AND y hasRelative z IT MEANS x hasRelative z (it will generate the same relation as a shortcut over a chain of itself)
* hasHusband is the same relation as hasWife but in reverse direction (the presence of one will generate the inverse of it)

To check the effect of these axioms, upload:

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

:Jim :hasFriend :George.

:Anna :hasHusband :George.

:Andrew :hasRelative :Jim.

:Jim :hasRelative :Jane.

The generated stuff is:

:George :hasFriend :Jim (because of the symmetry of friendship)

:George :hasWife :Anna (because it is inverse to the husband relationship)

:Andrew :hasRelative :Jane (because being a relative is a transitive relationship)

To check, run the following queries:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x :hasFriend ?y}

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x :hasWife ?y}

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x :hasRelative ?y}

Next, we mix in some RDFS axioms to trigger a chain reaction of reasoning:

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

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

:hasWife **rdfs:subPropertyOf** :hasRelative.

:hasHusband **rdfs:subPropertyOf** :hasRelative.

:hasRelative **rdfs:domain** :Person; **rdfs:range** :Person.

Interpretation:

* hasWife and hasHusband are specific variants of hasRelative
* hasRelative connects two Persons

The generated information is:

* hasRelative connections everywhere:

:Anna :hasRelative :George.

:George :hasRelative :Anna.

:George :hasRelative :George. #because of transitivity combined with inverse

:Anna :hasRelative :Anna. #because of transitivity combined with inverse

* If hasRelative connects persons, its subproperties also connects persons:

:hasWife rdfs:domain :Person; rdfs:range :Person.

:hasHusband rdfs:domain :Person; rdfs:range :Person.

* the Person type for everyone:

:Anna a :Person.

:George a :Person.

:Andrew a :Person.

:Jim a :Person.

:Jane a :Person.

Queries to check them:

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

SELECT \* WHERE {?x a :Person}

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

SELECT \* WHERE {?x :hasRelative ?y}

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

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x rdfs:domain :Person; rdfs:range :Person}

Upload the following axiom:

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

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

:hasUncle **owl:propertyChainAxiom** (:hasParent :hasBrother).

Interpretation:

* generate hasUncle as a shortcut over a chain of hasParent and hasBrother ("IF x hasParent y and y hasBrother z, GENERATE x hasUncle z")

Now upload some relationships:

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

:John :hasParent :Richard.

:Richard :hasBrother :Mike.

This will generate:

:John :hasUncle :Mike.

You can find it with the query:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:John ?y ?z}

**More advanced ways to detect types**

In the RDFS section we saw how the type of things can be generated from the relationships where they participate (subjects receive the type indicated by domain, objects receive the type indicated by range). OWL comes with more advanced features to generate entity types – not only on the existence of a relation, but also on the thing to which the relation connects – e.g. a MovieDirector is someone who directed a movie, it's not sufficient to know that he directed something.

Add the axioms:

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

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

:SolarSystemBody **owl:onProperty** :orbits; **owl:hasValue** :Sun.

Interpretation (the deduction works two ways!):

* generate the type SolarSystemBody for everything that orbits the Sun
* generate the statement that something orbits the Sun, if that something is of type SolarSystemBody

Add the statements:

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

:Mars :orbits :Sun.

:Earth a :SolarSystemBody.

Generated information:

:Mars a :SolarSystemBody.

:Earth :orbits :Sun.

Queries for checking:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x :orbits ?y}

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x a :SolarSystemBody}

Now let's adjust this so that the generated type is not determined by the target object, but by the type of the target object. Add the axioms:

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

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

:TheaterDirector **owl:onProperty** :directed; **owl:someValuesFrom** :Play.

Interpretation (unlike the Sun example, here the deduction only works one way – the object type determines the subject type):

* generate the type TheaterDirector for everyone who directed at least a play

Add the statements:

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

:Tempest a :Play. **#this could be generated from some other axiom, or inserted manually**

:EugenGyemant :directed :Tempest.

Generated information:

:EugenGyemant a :TheaterDirector.

Queries for checking:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {?x a :TheaterDirector}

The next example does the same in the reverse direction – the subject type determines the object type. Add the axioms:

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

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

:Vegan **owl:onProperty** :eats; **owl:allValuesFrom** :Vegetable.

Interpretation (also one way deduction):

* generate the type Vegetable for all things that are eaten by someone known to be a Vegan

Add the statements:

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

:John a :Vegan; :eats :cabbage.

Generated information:

:cabbage a :Vegetable.

Queries for checking:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE {:cabbage a ?x }

**More precise ways of unifying and intersecting entity types**

We saw in earlier tutorials how we can unify types with **rdfs:subClassOf** – however that's not exactly a union in mathematical sense because the supertype could be more than the union. For example, we can declare that Man and Woman are subtypes of Person – but this does not exclude having additional persons that are neither men, nor women. To put it short, a supertype includes all its subtypes but is not limited to them. A similar argument applies to intersections (a subtype of two supertypes is not exactly their intersection, it is part of their intersection).

OWL brings a more precise way of defining unions and intersections. Upload the following axioms:

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

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

:Human **owl:unionOf** (:Man :Woman).

:AustrianWriter **owl:intersectionOf** (:Writer :Austrian).

Interpretation:

* Man and Woman are subtypes of Human (which further means that every man and woman will receive the Human supertype)
* AustrianWriter is a subtype of both Writer and Austrian (which means that every Austrian writer will receive the types Writer and Austrian)

Without even waiting for new statements, the following hierarchy will be generated (so axioms can also generate parts of the graph schema!):

:Man rdfs:subClassOf :Human.

:Woman rdfs:subClassOf :Human.

:AustrianWriter rdfs:subClassOf :Writer, :Austrian.

Upload the following statements:

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

:Johann a :AustrianWriter, :Man.

Since we now have a richer class hierarchy, Johann will receive all the other types:

:Johann a :Writer, :Austrian, :Human.

Check this:

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE { :Johann a ?x}

Notice that both the intersection and union specify **ordered lists of types** – these lists can be as long as necessary and can be combined with anonymous nodes to nest multiple set operations. Upload the axioms:

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

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

:ScientificResearcher **owl:unionOf** (:UniversityResearcher :InstituteResearcher [**owl:oneOf** (:Galilei :Newton :Arhimede)]).

:UniversityResearcher **owl:intersectionOf** (:UniversityEmployee :Teacher).

:InstituteResearcher **owl:intersectionOf** (:ResearchInstituteEmployee [**owl:complementOf** :AdministrativeEmployee]).

Interpretation: Scientific Researchers may be of several kinds:

* UniversityResearcher = those University Employees that are also Teachers (university teachers have mandatory research tasks);
* InstituteResearcher = those Research Institute Employees that are not part of the administrative staff (**owl:complementOf** excludes a type, but is used in combination with **owl:intersectionOf** to indicate where to exclude from);
* plus, some enumerated historical individuals Galilei, Newton, Arhimede – **owl:oneOf** is an explicit enumeration of set elements (i.e. individuals that should each receive a type).

Anonymous nodes were used to avoid inventing IDs for some sets (the complement, the enumeration, they could also be used for unions/intersections if we don't plan to reuse or refer those).

The propagation of generated types can be understood by adding the statements:

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

:Robert a :UniversityResearcher.

:Andrei a :InstituteResearcher.

Check the types received by Robert (he should become UniversityEmployee, Teacher, Scientific Researcher):

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE { :Robert a ?x}

Check the types received by Andrei (he should become ResearchInstituteEmployee, ScientificResearcher plus an anonymous type caused by the complement):

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE { :Andrei a ?x}

For intersection, the reasoning works both ways – not only from intersection to supertypes, but also vice versa – if we declare something to belong to both supertypes, it logically becomes an element of their intersection:

Add:

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

:Max a :UniversityEmployee, :Teacher.

You can check that this generated the intersection type for Max:

:Max a :UniversityResearcher.

How do we use the exclusion (**owl:complementOf**)? That 's not for assigning types, exclusion is a form of negation – therefore useful for contradiction detection. We already stated that Andrei is an Institute Researcher. Now try to also make him an Administrative Employee (the subset excluded from InstituteResearcher):

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

:Andrei a :AdministrativeEmployee.

You should get a contradiction error (if the *Check for Inconsistency* box was activated when the database was created). The next section will get in more details about contradictions, for now we indicate that there are **two strategies for reacting to contradictions**:

* what we did now – i.e. to activate the inconsistency check from the very beginning; this rejects any statements that contradict those already stored;
* the alternative is to not activate it from the very beginning, to allow any statements to be accumulated and only later to execute a global verification – this can be triggered by a system configuration query:

PREFIX sys: <http://www.ontotext.com/owlim/system#>

INSERT DATA {

[sys:consistencyCheckAgainstRuleset "owl2-rl"].

}

This global verification will not remove anything, it will only point to the problem – **with a contradiction it cannot be established automatically which is the wrong information and which is right**, so this technique allows for human checking instead of just rejecting statements.

**More Contradiction detection (still OWL, cannot be done with RDFS axioms)**

Another way to detect contradictions is based on incompatible types. Upload the axiom:

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

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

:Man **owl:disjointWith** :Woman.

Interpretation:

* The types Man and Woman cannot have common instances

For convenience, the distinctions can also be enumerated as follows:

[a **owl:AllDisjointClasses**; owl:members (:Man :Woman)].

This makes it easier to enumerate a long list of incompatible types without pairing them two-by-two.

Now try to upload an individual that has both incompatible types:

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

:Anna a :Man, :Woman.

The upload will be rejected with a contradiction error.

Disjointness also has an interpretation for properties: when two relations should not be allowed between the same two things, i.e. they are mutually exclusive. The next axiom declares that it's not allowed to be the boss of your own relatives:

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

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

:bossOf **owl:propertyDisjointWith** :hasRelative.

For convenience, the distinctions can also be enumerated as follows:

[a **owl:AllDisjointProperties**; owl:members (:bossOf :hasRelative)].

This makes it easier to enumerate a long list of incompatible relations without pairing them two-by-two.

Now try to upload the two incompatible relations between the same individuals:

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

:John :bossOf :Jim; :hasRelative :Jim.

The upload will be rejected with a contradiction error.

Now declare a relationship that cannot be reversed between the same individuals:

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

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

:motherOf a **owl:AsymmetricProperty**.

And try to upload statements that switch the direction of motherhood:

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

:Anna :motherOf :Mary.

:Mary :motherOf :Anna.

The upload will be rejected with a contradiction error.

Finally, one very straightforward way of testing contradiction detection is to **explicitly declare that a particular statement is false** (negated or forbidden). Let's say we want to negate the information *Ana likes John* and *Romania has a population of 1000*. Upload the declarations as follows:

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

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

[**owl:sourceIndividual** :Ana; **owl:assertionProperty** :likes; **owl:targetIndividual** :John].

[**owl:sourceIndividual** :Romania; **owl:assertionProperty** :hasPopulation; **owl:targetValue** 1000].

Notice that the false statement is actually decomposed in its three terms and held together by an anonymous node. When uploaded, the anonymous node automatically receives the type **owl:NegativePropertyAssertion**, which will make it easily found by queries.

Now try to upload these two statements:

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

:Ana :likes :John.

:Romania :hasPopulation 1000 .

The upload will be rejected with a contradiction error.

This reasoning pattern is useful in scenarios where we want **to store negated statements besides true statements** – e.g. in social networks or recommender systems where we want to store both likes and dislikes in a way that makes it clear they are opposite (not only mutually exclusive, which we can declare with **owl:propertyDisjointWith**). Another typical area where information is stored both in asserted and negated forms is the judicial domain, with a typical reasoning scenario being: *if X is currently a Defendant accused of murdering Y and we have proven information that X did not murder Y, assign to X the type Acquitted*:

INSERT {?x a :Acquitted}

WHERE {

?x a :Defendant; :accusedOf [:deed :murder; :victim ?y].

[a **owl:NegativePropertyAssertion**; **owl:sourceIndividual** ?x; **owl:assertionProperty** :murdered; **owl:targetIndividual** ?y].

}

Remember that a FILTER NOT EXISTS clause can also be used to filter or reason based on the absence of a pattern – however the idea of this negation is that **absent information should not imply negated information**. Not knowing something (yet) does not imply it's not true (it could be added later in the database)!

In this context, remember that contradictions are not necessarily rejected by the database. We can allow them entering the database, if we don't activate the inconsistency check box; later, we can apply a global check to detect them with the help of a system configuration query.

**Equivalence generation (and how it affects Contradiction detection)**

Before going into deeper details about contradictions, upload the following axioms:

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

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

:hasEmail a **owl:InverseFunctionalProperty**.

:hasMother a **owl:FunctionalProperty**.

Interpretation:

* a mother is unique, an individual cannot have two mothers (a functional property cannot have multiple values for the same individual);
* an e-mail account is individual, it should not be assigned to two distinct individuals (an inverse functional property cannot have the same value for multiple subjects).

Upload the statements:

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

:Anne :hasEmail <mailto:anne@yahoo.com>.

:Ana77 :hasEmail <mailto:anne@yahoo.com>.

:Peter :hasMother :Mary, :Mimi.

We might be tempted to think that this is a contradiction situation. Even if the interpretation sounds like a constraint, it is not! The statements are not rejected – instead, the two mothers become the same (with multiple IDs); similar for emails – the two persons having the same email account become one and the same. The generated information is:

:Mary owl:sameAs :Mimi.

:Anne owl:sameAs :Ana77.

This is a pattern for **equivalence generation** – remember that owl:sameAs is used to declare that different URIs (usually created in different organizations) identify the same thing. We can see now that equivalence is not only declared, it can also be detected automatically by certain axioms (e.g. people with pseudonyms using the same e-mail account in different situations).

Check the generated equivalencies with a query (the filter removes default equivalence axioms):

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE { ?x owl:sameAs ?y. FILTER (?x!=?y)}

Equivalence is bidirectional, the inference engine will generate it in both directions between a pair of IDs. Moreover, all information about one of the involved IDs will be duplicated for the other one – this can be seen as:

* **a nice benefit for queries** (they will find all information regardless of which of the equivalent IDs is used in the query);
* a large expansion of the database (**duplicated information**).

As a compromise, GraphDB allows us to disable sameAs reasoning on various levels[[7]](#footnote-7). For example, when we create the database we can check the box *Disable owl:sameAs support* – equivalences are thus only stored in one direction and information is not copied between equivalent IDs. This can be a huge performance improvement, with the cost of having to include equivalences explicitly in all queries that need to collect information about both IDs (plus some filters to avoid getting superfluous results):

SELECT DISTINCT ?prop ?val WHERE

{

{:Ana ?prop ?val. FILTER (?prop!=owl:sameAs)}

UNION

{:Ana owl:sameAs ?otherID. ?otherID ?prop ?val. FILTER (?otherID!=:Ana)}

}

In the current exercise the owl:sameAs reasoning was not disabled, so if you add something about :Ana77...

INSERT DATA {:Ana77 :knows :Richard}.

...you will find the information duplicated for Anne:

SELECT ?x WHERE {:Anne :knows ?x}

(and the same for Mary/Mimi)

What about contradictions? **We can still reach a contradiction state, if we negate the equivalence** – we saw earlier the negation pattern, but here we have a dedicated term to negate equivalence:

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

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

:Mimi **owl:differentFrom** :Mary.

:Anne **owl:differentFrom** :Ana77.

GraphDB will reject the upload - **we cannot have two individuals who are both the same and different**!

For convenience, the distinctions can also be enumerated:

[a **owl:AllDifferent**; owl:members (:Mimi :Mary)].

This kind of declaration allows us to easily enumerate a long list of distinct things without pairing them two-by-two.

Usually this kind of obvious contradiction is not something we would upload ourselves – in practice at least one of the two sides (the equivalence or the distinction) would be generated via reasoning, possibly from information retrieved from different contradicting sources.

Several other contradiction patterns work in a similar fashion, by actually generating equivalence (and waiting for it to be denied):

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

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

:Student **owl:hasKey** (:studentID :studiesAt).

This is a more sophisticated version of owl:FunctionalProperty:

* it is applicable only to the mentioned type (Student)
* it allows for "multi-field keys" (the combination of studentID and studiesAt values must be unique, but the same studentID may exist in different universities).

A dataset that can test this is:

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

:Rachel :studentID **100**; :studentAt :**UNIVIE**; a :Student.

:Rahela :studentID **100**; :studentAt :**UNIVIE**; a :Student.

:Gigi :studentID 100; :studentAt :UBB; a :Student.

Generated information:

:Rachel owl:sameAs :Rahela.

Search for the equivalences and you will notice that Gigi was left out – even if the studentID value is the same, her university is different (only the combination must be unique!):

PREFIX : <http://buchmann.ro#>

SELECT \* WHERE { ?x owl:sameAs ?y. FILTER (?x!=?y)}

The earlier discussion applies here also: we only get contradiction if the generated equivalence clashes with a distinction.

A result similar to **owl:FunctionalProperty** may also be achieved with cardinality limited to 1 (more than one values for a property will be considered equivalent). However, with cardinalities we can also obtain contradiction directly – i.e. when limiting the cardinality to 0:

Upload the axioms:

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

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

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

:Human **owl:onProperty** :hasFather;

**owl:maxCardinality** "1"^^xsd:nonNegativeInteger.

:Orphan **owl:onProperty** :hasFather;

**owl:maxCardinality** "0"^^xsd:nonNegativeInteger.

Interpretation:

* a human can have maximum one father (if it has more they are considered the same);
* for an orphan it is forbidden to use the *:hasFather* relation (direct contradiction if it is used).

Upload the statements:

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

:Jack a :Human; :hasFather :Chris, :Christoph.

No contradiction is triggered, but you can check that the following was generated:

:Chris owl:sameAs :Christoph.

You may also check that cardinality reasoning is conditioned by the entity type – i.e., only works if Jack is declared as Human. This is more obvious in the second part with cardinality zero. Upload the statements:

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

:Jack a :Orphan.

The information is rejected because we already declared fathers for Jack, so he cannot be an orphan. It would also work in reverse order – to have in the database the fact that Jack is an orphan, and later to try to declare some father for him.

A more sophisticated version of this is to condition the reasoning not only on the subject type, but also on the object type – this can be achieved by limiting **maxQualifiedCardinality**. Upload the following axioms:

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

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

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

:OrphanOfMother **owl:onProperty** :hasParent;

**owl:maxQualifiedCardinality** "0"^^xsd:nonNegativeInteger;

**owl:onClass** :Woman.

Interpretation:

* a contradiction is triggered if someone of type OrphanOfMother gets a parent of type Woman

Now upload the facts to trigger the contradiction:

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

:Andrew a :OrphanOfMother; :hasParent :Mimi.

:Mimi a :Woman.

Qualified contradiction can also be limited to 1, having the same effect as in Jack's example above – i.e. to generate equivalence between multiple IDs acting as targets for the same relation, but this time they also need to have a specific type.

Finally, it's important to note that equivalence can also be stated:

* between types, with **owl:equivalentClass**
* between properties, with **owl:equivalentProperty**

However, GraphDB's rulesets cannot generate automatically such equivalences, since there is no logical pattern leading to a guarantee that two relations or types have the same meaning. They can be entered manually during a "schema mapping" effort (or "ontology mapping") – when we realize that our types and properties have the same meaning as some existing ones from known sources (e.g. Schema.org) and we want to make this similarity available to queries. With an inference engine active, they will also trigger the expected reasoning:

* any individual will get all types equivalent to the one it already has;
* statements involving equivalent properties will be multiplied for each property version.

To conclude, we should not lose sight about the following: it should not be understood that every time we work with these axioms we must activate an inference ruleset and generate new information:

* **a graph schema** (which is minimally made of RDFS axioms and optionally of OWL axioms) **can also be created for documentation purposes** (to communicate to others what types and properties are used in our graphs, how they're supposed to be connected);
* **inference engines come with a performance cost** (especially in updates) and an increase in the database size – the more powerful the ruleset, the higher this cost will be; we activated OWL2-RL to be able to demonstrate a large variety of axioms, but GraphDB allows us to select less powerful subsets that only support some of these axioms – it's a tough decision to decide "how much reasoning" should be enabled for a particular scenario;
* **axioms can be useful even without an inference engine** – we saw in an earlier section (about INSERT) that queries can benefit from "inactive" axioms to check how certain types or properties are related and filter results based on this;
* **disabling owl:sameAs support from the very beginning has a major impact on performance**, with the cost of having to write more complicated queries.

1. Reminder: underscore IDs cannot be used in queries [↑](#footnote-ref-1)
2. We assumed to know that the property is only used in the middle of sentences. If the graph would contain a schema (axioms), properties would also appear in other positions! [↑](#footnote-ref-2)
3. It's also possible to activate it later [↑](#footnote-ref-3)
4. Details on the different OWL subsets, called "profiles", may be consulted at https://www.w3.org/TR/owl2-profiles/ [↑](#footnote-ref-4)
5. Depending on the version of GraphDB you use, it's possible that standard prefixes (rdfs:, owl:) and already uploaded prefixes do not have to be redeclared in subsequent imports. However, we keep in these examples the prefix declarations for every upload – normally they should accompany any dataset, even if GraphDB spares us the effort. [↑](#footnote-ref-5)
6. SHACL rules form a "validation schema", i.e. they specify what properties are **allowed** for something that already has the types Director or Movie; but for now let's get our head around reasoning rules (which form a "reasoning schema", or "ontology") [↑](#footnote-ref-6)
7. See https://graphdb.ontotext.com/documentation/standard/sameas-optimisation.html [↑](#footnote-ref-7)