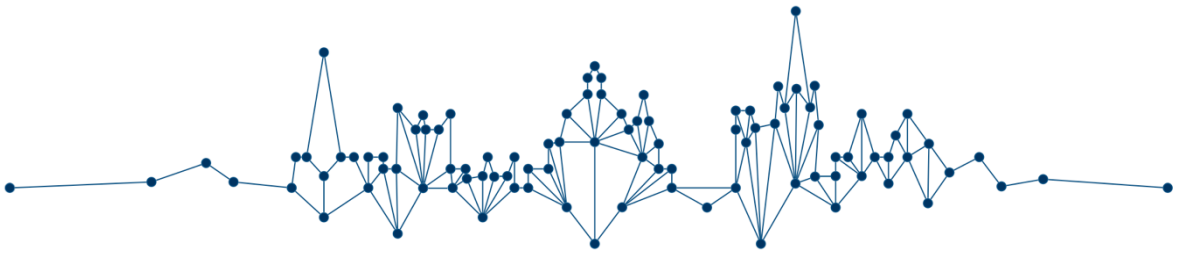
Getting started part 2: Using Rules in RDFox

An in-memory reasoning graph database built for speed and scale



OXFORD SEMANTIC TECHNOLOGIES

Latest revision 5 May 2020

For RDFox 3.0.0

The following guide walks you through some of the common uses of RDFox rules in applications.

The Transitive Relation Pattern

Intent

To define a relation in a graph as transitive.

Description

There are many relations which are naturally transitive. A graph mentioning such a relation, however, may be incomplete and not explicitly contain all of the relevant connections. For instance, a graph may contain the following triples:

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:oxford       :locatedIn :oxfordshire .
:oxfordshire  :locatedIn :england .
:england      :locatedIn :uk .
```

The relation `locatedIn` is intuitively transitive; for instance, Oxford is located in England because it is located in Oxfordshire and Oxfordshire is in turn located in England. However, the triple

```
:oxford  :locatedIn :england .
```

is missing from the graph. We can use RDFox to automatically add the missing triples by importing a single rule, which defines the relationship as transitive.

```
[?x, :locatedIn, ?z] :- [?x, :locatedIn, ?y], [?y, :locatedIn, ?z] .
```

Indeed, the rule says that if an object `x` in the graph is connected by `:locatedIn` to an object `y`, and `y` is in turn connected by `:locatedIn` to an object `z`, then `x` is also connected by `:locatedIn` to `z`.

The following SPARQL query

```
select ?x ?y where {?x :locatedIn ?y}
```

can then be issued to obtain the expected results.

```
:oxford      :uk .
:oxford      :england .
:oxfordshire :uk .
:england     :uk .
:oxfordshire :england .
```



```
:oxford :oxfordshire .
```

Observations

Standard ontology languages such as OWL 2 also provide means for defining a relation as transitive. In particular, the OWL 2 RL axiom

```
TransitiveObjectProperty(:locatedIn )
```

Defines the `:locatedIn` relation as transitive. Indeed, when loading an OWL 2 ontology in RDFox, such axiom would be automatically converted into the rule

```
[?x, :locatedIn, ?z] :- [?x, :locatedIn, ?y], [?y, :locatedIn, ?z] .
```


The Transitive Closure Pattern

Intent

To compute the transitive closure of a non-transitive relation in a graph.

Description

In many other situations, we may have a relation R that is not transitive, but we are interested in defining a different relation that "transitively closes" R.

Consider a social network where users follow other users. The graph may be represented by the triples next.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:alice :follows :bob .
:bob   :follows :charlie .
:diana :follows :alice .
```

Something we often see in social networks is a situation where the system uses the existing connections to suggest new ones. For example, since Alice follows Bob and Bob follows Charlie, the system may suggest that Alice follow Charlie as well. Likewise, the system may suggest that Diana follow Bob; but then, if Diana follows Bob, she should also follow Charlie. We would like to construct an enhanced social network that contains the actual `follows` relations plus all the suggested additional links. The links in such enhanced social network represent the transitive closure of the original `follows` relation, which relates any pair of people who are connected by a path in the network.

The transitive closure of the `follows` relation can be computed using RDFox by defining the following two rules:

```
[?x, :followsClosure, ?y] :- [?x, :follows, ?y] .

[?x, :followsClosure, ?z] :-
   [?x, :follows, ?y], [?y, :followsClosure, ?z] .
```

The first rule "copies" the contents of the direct `follows` relation to the new relation. The second rule implements the closure by saying that if a person p1 directly follows p2 and p2 (directly or indirectly) follows person p3, then p1 (indirectly) follows p3.

If we now issue the SPARQL query

```
select ?x ?y where {?x :followsClosure ?y}
```

We obtain the expected results


```
:diana :charlie .
:alice :charlie .
:diana :bob .
:alice :bob .
:bob :charlie .
:diana :alice .
```

Finally, we may also be interested in computing the suggested links that were not already part of the original `follows` relation. This can be achieved, for instance, by issuing the SPARQL query

```
select ?x ?y where {
    ?x :followsClosure ?y .
    filter not exists {?x :follows ?y}
}
```

The results are the expected ones.

```
:diana :charlie .
:alice :charlie .
:diana :bob .
```

Observations

This pattern cannot be represented in OWL 2. Although the standard ontology language can define a relation as transitive, it cannot define the transitive closure of an arbitrary (non-transitive) relation. In OWL 2 we would only be able to define the `follows` relation as transitive, which would be wrong from a modelling perspective, and wouldn't allow us to distinguish between direct and indirect (I.e., suggested) connections.

It is worth noting that OWL 2 does provide syntactic constructs that could simulate transitive closure of a relation. For instance, the following OWL 2 axioms correspond directly to our previous rules defining the transitive closure of the `follows` relation.

```
SubObjectPropertyOf( :follows :followsClosure )
SubObjectPropertyOf( ObjectPropertyChain
                     (:follows :followsClosure) :follows)
```

These axioms, however, violate certain global syntactic restrictions in the standard, which are necessary to ensure decidability of reasoning in OWL 2, and hence would be rejected by OWL 2-compliant reasoners.

The Shortcut Pattern

Intent

To generate direct a connection between objects indirectly connected in a graph.

Description

An important practical use of knowledge graphs is to power Open Query Answering (Open QA) applications, where the user would pose a question in natural language, which is then automatically answered against the graph. Open QA systems often struggle to interpret questions that involve several "hops" in the graph. For instance, consider the graph consisting of the triples given next.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:douglasAdams :bornIn  :uk .
:uk           rdf:type :country.
```

A user may ask the Open QA system for the country of birth of Douglas Adams. To obtain this information, the system would need to construct a query involving two hops in the graph. In particular, the SPARQL query

```
select ?c where {
  :douglasAdams :bornIn  ?c .
  ?c            rdf:type :country .
}
```

would yield the desired result.

```
:uk
```

The results of the open QA system would be greatly enhanced if the desired information had been available in just a single hop. RDFox rules can be used to provide a clean solution in this situation. In particular, we can use rules to define a new `countryOfBirth` relation that provides a "shortcut" for directly accessing the desired information.

```
[?x, :countryOfBirth, ?y] :- [?x, :bornIn, ?y], [?y, rdf:type, :country] .
```

The rule says that, if a person p is born in a place c, and that place is a country, then c is the country of birth of p. As a result, RDFox would derive that the country of birth of Douglas Adams is the UK. The Open QA system would now only need to construct the following simpler query, which involves a single hop in the graph, to obtain the desired information.


```
select ?x ?y where {?x :countryOfBirth ?y}
```

This now simpler query yields the result.

```
:douglasAdams :uk .
```

The Shortcut Pattern naturally combines with the Transitive and Transitive Closure patterns. For instance, consider the following (more complicated) graph.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:douglasAdams   :bornIn    :cambridge .
:cambridge      :locatedIn :cambridgeshire .
:cambridgeshire :locatedIn :england .
:england        :locatedIn :uk .
:uk             rdf:type   :country.
```

In the absence of rules, the Open QA system would need to construct and answer a complex query involving many hops in the graph in order to obtain the country of birth of Douglas Adams. In particular, the SPARQL query

```
select ?c where {
  :douglasAdams :bornIn ?u .
  ?u :locatedIn ?w .
  ?w :locatedIn ?z .
  ?z :locatedIn ?c .
  ?c rdf:type :country .
}
```

will yield the desired result.

```
:uk
```

Such a query, however, would be very difficult for the Open QA system to construct, where a specific difficulty is that the system would need to know in advance the number of hops that are necessary to connect Douglas Adams with his country of birth. This difficulty becomes even more apparent if we wanted to list all people together with their country of birth. For instance, suppose that the graph also contains the following triples.

```
:williamShakespeare :bornIn    :stratford .
:stratford          :locatedIn :england .
```

Then, the straightforward variant of the previous query


```
select ?x ?c where {
  ?x :bornIn    ?u .
  ?u :locatedIn ?w .
  ?w :locatedIn ?z .
  ?z :locatedIn ?c .
  ?c rdf:type   :country .
}
```

will only return as answer

```
:douglasAdams :uk .
```

Indeed, the data does not contain a link between the city of Stratford and the county it belongs to, which prevents William Shakespeare to be listed as part of the answer.

The Shortcut and Transitive Relation patterns can be combined in RDFox to provide a clean solution in this situation. In particular, we can now use the following rules to define the `countryOfBirth` relation.

```
[?x, :countryOfBirth, ?z] :-
    [?x, :bornIn, ?y],
    [?y, :locatedIn, ?z],
    [?z, rdf:type, :country] .

[?x, :locatedIn, ?z] :-
    [?x, :locatedIn, ?y],
    [?y, :locatedIn, ?z] .
```

The first rule creates a shortcut; it says that if a person p was born in a place b and that place b is located in a country c, then c is the country of birth of p. The second rule, which we have seen before, defines the relation `locatedIn` as transitive.

Using the second rule, RDFox would be able to derive that both Cambridge and Stratford are located in the UK and derive the corresponding direct `locatedIn connection`. Then, RDFox would be able to use the first rule to derive that the country of birth of both Douglas Adams and William Shakespeare is the UK. Indeed, by constructing the simple query

```
select ?x ?y where {?x :countryOfBirth ?y}
```

The Open QA system would now be able to provide the desired information.

```
:williamShakespeare :uk .
:douglasAdams       :uk .
```


The Query-View Pattern

Intent

To store the result of a SPARQL query as a new relation (a "view"), and then use such view in the subsequent definition of additional relations.

Description

When querying a knowledge graph, we may be interested in materialising the result of a SPARQL query as a new relation in the graph. This can be the case, for instance, if the query is interesting on its own right, can be used to define new relations, or simplify the formulation of additional queries.

We can use an RDFox rule for this purpose, where the SPARQL query that we want to materialise in the graph is represented in the antecedent of the rule and the answer as a new relation in the consequent.

For instance, consider again the previous example of a social network, where we were interested in suggesting new followers (recall the Transitive Closure usage pattern). Recall that we used a query

```
select ?x ?y where {
    ?x :followsClosure ?y
    filter not exists {?x :follows ?y}
}
```

to obtain the suggested links that were not already part of the original `follows` relation. We may be interested in storing this query as a separate relation in the graph. For this, we could rewrite the query as a rule defining a new `:suggestFollows` relation:

```
[?x, :suggestFollows, ?y] :-
    [?x, :followsClosure, ?y],
    not [?x, :follows, ?y] .
```

The antecedent of the rule represents the "where" clause in the query. The triple pattern

```
?x :followsClosure ?y
```

is represented by the first body atom [?x, :followsClosure, ?y]. In turn, the filter expression in the query is captured by the atom "not [?x, :follows, ?y]", which uses Negation-as-Failure to select the subset of connections that do not correspond to a explicitly stated `follows` connection. Then, the simple query

```
select ?x ?y where {?x :suggestFollows ?y }
```


Will give us the expected answers

```
:diana :charlie .
:alice :charlie .
:diana :bob .
```

Observations

Only a subset of SPARQL 1.1 queries can be transformed into RDFox rules in this way. In particular, all queries involving basic graph patterns, filter expressions, negation (NOT EXISTS, MINUS) and aggregation can be represented. In contrast, this technique wouldn't work for SPARQL queries using OPTIONAL or UNION. Also, we wouldn't be able to rewrite queries with more than two answer variables.

Finally, it is also worth noticing that the OWL 2 language does not provide the feature of Negation-as-Failure and hence SPARQL queries involving negation cannot be translated as OWL Axioms.

The Data Source Mapping Pattern

Intent

To import data from a data source (e.g., relational) and construct a graph based on it.

Description

Data feeding a knowledge graph often stems from different types of external data sources, such as relational databases. We can use RDFox rules to specify how each record in the external data source corresponds to a set of nodes and edges in the graph.

RDFox allows us to load the information in an external data source by means of a two-stage process. The first step is to attach a data source and assign it to a relation. For instance, consider the following data about the employees of ACME corporation in a CSV file named “employee.csv”.

```
emp_id | emp_name | job_name  |  hire_date | salary
-------+----------+-----------+------------+---------+------------+
68319  | KAYLING  | PRESIDENT |            | 200,000
66928  | BLAZE    | MANAGER   | 2017-05-01 | 90,000
67453  | JONES    | ASSISTANT | 2018-05-03 | 35,000
```

The raw CSV format:

```
emp_id,emp_name,job_name,hire_date,salary
68319,KAYLING,PRESIDENT,,200000
66928,BLAZE,MANAGER,2017-05-01,90000
67453,JONES,ASSISTANT,2018-05-03,35000
```

We attach the table to an RDFox relation `Employee` with 5 arguments, one per column in the table. This can be achieved using the following commands.

```
dsource add delimitedFile "EmployeeDS" \
  file "$(dir.root)csv/employee.csv" \
  header true
```

The net result is that the employee.csv is added as an RDFox data source. We called the data source `EmployeeDS`. Here, `file` specifies the path to the file, and `header` indicates whether the file contains a header row.

At this point, we can check whether the data has been attached correctly and whether the RDFox data source is in place by running the command

```
dsource show EmployeeDS
```


to obtain the expected information

```
Data source type name: delimitedFile
Data source name:      EmployeeDS
Parameters:            file   = employee.csv
                       header = true
------------------------------------------------------------
Table name:            employee.csv
Column 1:              emp_id     xsd:integer
Column 2:              emp_name   xsd:string
Column 3:              job_name   xsd:string
Column 4:              hire_date  xsd:string
Column 5:              salary     xsd:integer
------------------------------------------------------------
```

The next step attaches the RDFox data source to an employee relation in RDFox.

```
prefix : <https://oxfordsemantic.tech/RDFox/tutorial/>

dsource attach :employee "EmployeeDS"  \
   "columns"     5                      \
    "1"          "https://oxfordsemantic.tech/RDFox/tutorial/{1}_{2}" \
    "1.datatype" "iri"                  \
    "2"          "{emp_name}"           \
    "2.datatype" "string"               \
    "3"          "{job_name}"           \
    "3.datatype" "string"               \
    "4"          "{hire_date}"          \
    "4.datatype" "string"               \
    "4.if-empty" "absent"               \
    "5"          "{salary}"             \
    "5.datatype" "integer"              \
    "5.if-empty" "absent"
```

The IRI of the new relation will be `:employee`, where ":" is the default prefix defined beforehand as `https://oxfordsemantic.tech/RDFox/tutorial/`. The `:employee` data relation will contain 5 arguments. The first argument provides an identifier for each employee as a composition of the prefix's IRI, the employee ID (first column in the data source) and the employee name (second column). The remaining arguments are obtained from the column of the corresponding name in the data source.

Since not every employee may have a hiring date or a known salary, the conditions "if-empty" indicate that the corresponding argument in the RDFox relation will be left empty.

Once the relation has been created in RDFox, it can be queried and used in the antecedent of rules. As a first step, we can query the RDFox relation using SPARQL to check whether the data has been imported correctly in the relation. The SPARQL query

```
select ?x ?y ?z ?u ?w where {:employee(?x, ?y, ?z, ?u, ?w)}
```


Will return the following answers:

```
fg:68319_KAYLING "KAYLING" "PRESIDENT" UNDEF        200000 .
fg:66928_BLAZE   "BLAZE"   "MANAGER"   "01/05/2017" 90000 .
fg:67453_JONES   "JONES"   "ASSISTANT" "03/05/2018" 35000 .
```

As we can see, the UNDEF entry represents that the value of the hiring date for the first employee is missing.

Now that we have the RDFox relation correctly in place, the next step would be to turn the data in the relation in the form of a graph. For this we can use the following *mapping rule*, where the RDFox relation forms the antecedent and the generated edges in the graph based on it are described in the consequent of the rule:

```
[?x, rdf:type, :employee],
[?x, :worksfor, :acme],
[?x, :hasName, ?y],
[?x, :hasJob, ?z],
[?x, :hiredOnDate, ?u],
[?x, :salary, ?w] :-
    fg:employee(?x, ?y, ?z, ?u, ?w) .
```

The materialisation of the rule generates a graph from the data in the relation. The new relations in the graph can be used in other rules to define additional concepts and relations. For instance, we can add the rules stating that every employee is a person and every person with a salary higher than £50,000 pays tax at a higher rate.

```
[?x, rdf:type, :person ] :-
    [?x, rdf:type, :employee ] .

[?x, :taxRate, :higher-rate] :-
    [?x, rdf:type, :person],
    [?x, :salary, ?y],
    FILTER(?y > 50000) .
```

Now we can query the graph to obtain, for instance, the list of high-income taxpayers.

```
select ?x where {?x :taxRate :higher-rate }
```

And obtain the expected results.

```
fg:68319_KAYLING .
fg:66928_BLAZE .
```


Observations

Data can be imported from different data sources and merged together in the graph. For instance, if we had a different employee table (e.g., for a different department) in another CSV, we could attach to it a new RDFox data source and exploit a mapping rule akin to the one before to further populate the binary relations in the graph, as well as to create new ones.

Simple Calculations

Intent

To use rules in order to perform computations over the data in the graph (e.g., unit conversions).

Description

We can use RDFox rules to perform computations over the data in a knowledge graph and store the results in a different relation. For instance, consider a graph with the following triples.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:alice :height "165"^^xsd:integer .
:bob   :height "180"^^xsd:integer .
:diana :height "168"^^xsd:integer .
:emma  :height "165"^^xsd:integer .
```

We now would want to compute their height in feet and record it in the graph. For this, we can import the following RDFox rule.

```
[?x, :heightFeet, ?y] :- [?x, :height, ?h], BIND(?h * 0.0328 AS ?y) .
```

The BIND construct evaluates an expression and assigns the value of the expression to a variable.

We can now query the graph for the newly introduced relation to obtain the list of people and their height in both centimetres and the calculated feet.

```
select ?x ?m ?f
where {
    ?x :height ?m .
    ?x :heightFeet ?f .
}
```

And obtain the expected answers

```
:emma  165 5.412 .
:diana 168 5.5104 .
:bob   180 5.904 .
:alice 165 5.412 .
```


Type and Attribute Inheritance Pattern

Intent

To arrange concepts and relations in a hierarchy.

Description

A common use of ontologies is to arrange concepts and relations in a subsumption hierarchy. For instance, we may want to say that dogs and cats are mammals and that mammals are animals. Such subsumption relationships can be easily represented using RDFox rules.

```
[?x, rdf:type, :mammal] :- [?x, rdf:type, :dog] .
[?x, rdf:type, :mammal] :- [?x, rdf:type, :cat] .
[?x, rdf:type, :animal] :- [?x, rdf:type, :mammal] .
```

Suppose that we have a graph with the following triples:

```
:max   rdf:type :dog .
:coco  rdf:type :cat .
:teddy rdf:type :mammal.
```

Then, RDFox will deduce that Max and Coco are both mammals and therefore also animals, and also that Teddy is an animal. In particular, the query

```
select ?x where {?x rdf:type :animal}
```

yields the expected results

```
:max .
:teddy .
:coco .
```

It is also often the case that concepts are “assigned” certain properties. For instance, mammals have children which are also mammals. This is known as a “range restriction” in the ontology jargon, and can be represented using the following RDFox rule

```
[?y, rdf:type, :mammal] :- [?x, rdf:type, :mammal],[?x, :hasChild, ?y] .
```

If we now extend the graph with the following triples.


```
:max  :hasChild :betsy .
:coco :hasChild :minnie .
```

RDFox will derive automatically that both Betsy and Minnie are also mammals (and therefore also animals). Indeed, the query

```
select ?x where {?x rdf:type :mammal}
```

Will yield the expected results.

```
:max .
:betsy .
:minnie .
:teddy .
:coco .
```

In many applications, it is also useful to represent subsumption relations between the edges in a knowledge graph, to specify that one relation is more specific than the other. For instance, we may want to say that the `hasDaughter` relation is more specific than the `hasChild` relation. This can be represented using the following RDFox rule.

```
[?x, :hasChild, ?y] :- [?x, :hasDaughter, ?y] .
```

If we now add the following triple to the graph

```
:betsy :hasDaughter :luna .
```

RDFox can infer that Luna is the child of Betsy and therefore she is also a mammal, and an animal. Indeed, the query

```
select ?x where {?x rdf:type :mammal}
```

Now includes Luna as an answer.

```
:max .
:luna .
:betsy .
:minnie .
:teddy .
:coco .
```


Observations

The ontology language OWL 2 is well-suited for representing this pattern. In particular, the rules in our example are captured by the following axioms expressed in the OWL 2 RL profile.

```
SubClassOf( :cat :mammal )
SubClassOf( :dog :mammal )
SubClassOf( :mammal :animal )
SubClassOf( :mammal ObjectAllValuesFrom ( :hasChild :mammal )
SubObjectPropertyOf ( :hasDaughter :hasChild )
```

RDFox provides full support for OWL 2 RL, and any OWL 2 RL ontology such as the previous one can be loaded and transformed into rules.

The Cycle Detection Pattern

Intent

To detect whether a relationship in a graph has a cycle.

Description

A common task in knowledge graphs is to identify cyclic relationships. For instance, partonomy relations are typically acyclic (e.g., if an engine is part of a car, we wouldn't expect the car to be part of the engine as well!). In these cases, cycle detection may be needed to detect errors in the graph and thus provide data validation.

A simple case of this pattern is when the relation we are checking for cyclicity is naturally transitive. Such is the case, for instance of the partOf relation. Consider the following graph:

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .
:a :partOf :b .
:b :partOf :c .
:c :partOf :a .
```

The graph clearly has a cycle since there is a cyclic path :a -> :b -> :c -> :a. via the partOf relation. The relationship is naturally transitive and hence we can use the corresponding pattern to define it as such.

```
[?x, :partOf, ?z] :- [?x, :partOf, ?y], [?y, :partOf, ?z] .
```

The following SPARQL query now gives us which elements are part of others (directly or indirectly)

```
select ?x ?y where {?x :partOf ?y}
```

Which gives us the following results

```
:a :a .
:c :c .
:b :b .
:a :c .
:b :a .
:c :b .
:c :a .
:b :c .
:a :b .
```


Cyclicity manifests itself by the presence of self-loops (e.g., :a is derived to be a part of itself). Hence, it is possible to detect that the part of relation is transitive by issuing the following SPARQL query.

```
Ask {?x :partOf ?x}
```

Where the result comes true since the partonomy relation does have a self-loop.

Alternatively, we could have defined the following additional rule.

```
[:partOf, rdf:type, :cyclicRelation] :- [?x, :partOf, ?x] .
```

Which tells us that if any object is determined to be a part of itself, then the partonomy relation is cyclic.

We can now issue the following SPARQL query, which retrieves the list of cyclic relations in the graph.

```
select ?x where {?x rdf:type :cyclicRelation}
```

To obtain the expected result.

```
:partOf .
```


The Aggregation Pattern

Intent

To compute aggregated values (e.g., sums, counts, averages, etc) over the graph and store the results in a new relation.

Description

Consider a social network graph consisting of the following triples.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:alice   :follows :bob .
:bob     :follows :charlie .
:diana   :follows :alice .
:charlie :follows :alice.
:emma    :follows :bob .
:alice   rdf:type :person .
:bob     rdf:type :person .
:charlie rdf:type :person .
:diana   rdf:type :person .
:emma    rdf:type :person .
```

The graph contains also information about people's hobbies, as represented by the following triples.

```
:alice    :likes   :tennis .
:bob      :likes   :music .
:diana    :likes   :swimming .
:charlie  :likes   :football .
:emma     :likes   :reading .
:tennis   rdf:type :sport .
:swimming rdf:type :sport.
:football rdf:type :sport.
```

We would like, for instance, to count for each person the number of followers who enjoy practicing a sport.

RDFox provides aggregation constructs which enable these kinds of computations.

```
[?y, :sportyFollowerCnt, ?cnt] :-
    [?y, rdf:type, :person],
    AGGREGATE( [?x, :follows, ?y],
        [?x, :likes, ?w],
        [?w, rdf:type, :sport]
        ON ?y BIND COUNT(DISTINCT ?x) AS ?cnt) .
```


The query part of the rule says that if p1 Is a person, then count all distinct people who follow p1 and who like some sport, store the result in a count, and propagate it to the consequent of the rule.

By issuing the following SPARQL query

```
select ?x ?cnt where {?x :sportyFollowerCnt ?cnt}
```

We obtain that Bob has one sporty follower (Alice), whereas Alice has 2 sporty followers (Diana and Charlie).

```
:bob   1 .
:alice 2 .
```

This pattern can be combined with the Transitive Closure pattern. For instance, we may be interested in counting the number of (direct or indirect) followers who are sporty. For this, we can use RDFox rules to compute the transitive closure of the `follows` relation (c.f. Transitive Closure pattern)

```
[?x, :followsClosure, ?y] :-
    [?x, :follows, ?y] .

[?x, :followsClosure, ?z] :-
    [?x, :follows, ?y],
    [?y, :followsClosure, ?z] .
```

And use the following rule, which is analogous to the one we used for the “follows” relation in order to compute the desired count.

```
[?y, :sportyFollowerClosureCnt, ?cnt] :-
    [?y, rdf:type, :person],
    AGGREGATE(
        [?x, :followsClosure, ?y],
        [?x, :likes, ?w],
        [?w, rdf:type, :sport]
        ON ?y BIND COUNT(DISTINCT ?x) AS ?cnt
    ) .
```

The following SPARQL query


```
select ?x ?cnt where {?x :sportyFollowerClosureCnt ?cnt}
```

Then provides the following results.

```
:charlie 3 .
:bob     3 .
:alice   3 .
```

We observe that the count for Charlie doesn't seem quite right. Charlie is followed directly only by Bob (who is not sporty); however, Bob is followed by Alice (a sporty person) and Alice is followed by Diana (another Sporty person). Naturally, we would have obtained a count of 2; however, Charlie also follows Alice and hence he transitively follows himself, thus the count of 3!. If we wanted to prevent this situation, we can modify the second rule implementing transitive closure to eliminate self-loops as follows:

```
[?x, :followsClosure, ?z] :-
    [?x, :follows, ?y],
    [?y, :followsClosure, ?z],
    FILTER(?x != ?z) .
```

Now, our query before yields the expected results

```
:charlie 2 .
:bob     3 .
:alice   2 .
```

Observations

There are certain limitations that apply when using the aggregation pattern. In particular, using aggregation inside a recursive rule leads to well-known semantic problems and it is precluded in RDFox. Note that the definition of transitive closure of a relation *is* recursive; however, the aggregation that we computed in our example lies outside the recursion (intuitively, one can fully compute the transitive closure first and then aggregate).

It is also worth noticing that OWL 2 does not provide aggregation features.

The Mandatory Attribute Pattern

Intent:

To check for incompleteness in the data and raise a warning wherever expected data is missing. In this particular pattern, we are concerned with defining a relation in the graph as "mandatory", in the sense that all objects of a particular type in the graph must have a value for that relation.

Description

In knowledge graphs, data is typically incomplete.

For instance, suppose that the data in a knowledge graph has been obtained from a variety of sources. The graph has different types of information about people, such as their name, job title and so on. We notice that some people in the graph have a date of birth, whereas others do not. Because of the nature of our application, we would like to have the date of birth of each person represented in the graph and would like to find out which people are missing this information. That is, we would like to make the presence of a date of birth value mandatory for every person in the graph. In relational databases this is typically solved by declaring an integrity constraint.

Consider the following graph.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .
:alice :dob      "11/01/1987"^^xsd:string .
:alice rdf:type  :person .
:bob   :dob      "23/07/1980"^^xsd:string .
:bob   rdf:type  :person .
:diana :height   "168"^^xsd:integer .
:diana rdf:type  :person .
:emma  :dob      "10/02/1965"^^xsd:string .
:emma  rdf:type  :person.
:max   rdf:type  :dog .
```

We can use the following rule to record absence of a date of birth for people.

```
[?x, rdf:type, owl:Nothing] :-
    [?x, rdf:type, :person],
    not exists ?y in ([?x, :dob, ?y]) .
```

The query says that if a person p lacks a date of birth d, then p violates a constraint. The constraint violation is recorded by making person p an instance of the special owl:Nothing unary relation, which is also present in the OWL 2 standard.

The following SPARQL query then correctly reports that Diana violates the constraint (whereas Max doesn't because he is a dog).

```
select ?x where {?x rdf:type owl:Nothing}
```

This pattern combines well with the inheritance pattern. For instance, suppose that we add the following triple:

```
:charlie rdf:type :student .
```

And the following rule stating that every student is a person

```
[?x, rdf:type, :person] :- [?x, rdf:type, :student] .
```

Then, the previous query will give as results

```
:charlie :dob .
:diana   :dob .
```

Indeed, since Charlie is a student, he is also a person; furthermore, Charlie lacks date of birth information.

Observations

The meaning of the special class `owl:Nothing` is different in RDFox and the OWL 2 standard. If one can derive from an OWL 2 ontology that that an object is an instance of owl:Nothing, then the ontology is inconsistent and querying the ontology becomes logically meaningless. Thus, the OWL 2 standard would require users to modify the data and/or ontology to fix the inconsistency prior to attempting to issue queries. Furthermore, it is worth noting that in OWL 2 it is not possible to write statements that check for "absence of information"; this is due to the monotonicity properties of OWL 2 as a fragment of first-order logic.

In contrast, in RDFox, deriving an instance of owl:Nothing does not lead to a logical inconsistency and the answers to queries remain perfectly meaningful. In the pattern we have described, querying for owl:Nothing simply provides users with the list of all nodes in the graph for which mandatory information is missing. As a result, the user is "warned" rather than prevented from carrying out a task such as issuing a query. For instance, if we were to ask a query to RDFox such as the following

```
select ?x where {?x rdf:type :person}
```


We would still obtain the expected results (see below) despite the fact that there are constraint violations in the data.

```
:alice .
:charlie .
:emma .
:diana .
:bob .
```

This behaviour is also different from relational databases, where the system would typically reject updates that lead to a constraint violation. As already mentioned, RDFox continues to operate normally and would accept any updates although constraints are being violated. Of course, users are encouraged to query the system in order to detect and maybe also fix such violations.

The Data Restructuring and Reification Pattern

Intent

To transform the structure of the data in a knowledge graph (e.g., by adding properties to a relationship).

Description

Consider the following knowledge graph representing employees and their employer.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:alice   :worksFor :oxfordUniversity .
:bob     :worksFor :acme .
:charlie :worksFor :oxfordUniversity .
:charlie :worksFor :acme .
```

Suppose that we now want to expand the graph by adding further information about the employment, such as the salary and the start date. This information is relative to each specific employment of an employee; for instance, Charlie will have a different salary and start date for his employment with Oxford University and his employment with Acme.

We can use RDFox rules to automatically restructure the data in the graph to account for the new information.

```
[?z, rdf:type, :Employment],
[?z, :employee, ?x],
[?z, :employer, ?y] :-
    [?x, :worksFor, ?y],
    BIND(SKOLEM("Employment",?x,?y) AS ?z) .
```

For each edge connecting a person x with their employer y, the rule creates a new employment instance z as an RDF blank node, and relates it to employee x and employer y. The name of generated instance starts with the character '_' indicating that it is a blank node, followed by the string "Employment" and finally the RDFox internal integer dictionary IDs for the corresponding employee x and employer y.

The query

```
select ?x ?y ?z where {?x ?y ?z . ?x rdf:type :Employment}
```


gives us the new triples generated by the application of the previous rule

```
_:Employment_116_200 :employer :acme .
_:Employment_116_200 :employee :bob .
_:Employment_116_200 rdf:type :Employment .
_:Employment_113_199 :employer :oxfordUniversity .
_:Employment_113_199 :employee :alice .
_:Employment_113_199 rdf:type :Employment .
_:Employment_156_199 :employer :oxfordUniversity .
_:Employment_156_199 :employee :charlie .
_:Employment_156_199 rdf:type :Employment .
_:Employment_156_200 :employer :acme .
_:Employment_156_200 :employee :charlie .
_:Employment_156_200 rdf:type :Employment .
```

It is important to notice that the generated SKOLEM IDs such as "_:Employment_116_200" cannot be considered stable across runs (or RDFox versions) since they are generated based in the dictionary IDs of the arguments. Further data relative to an employment, such as associated salary and start date, should therefore not be inserted directly as triples, but rather rules such as the following ones:

```
[?z, :salary, "60000"^^xsd:integer] :-
    BIND(SKOLEM("Employment", :alice, :oxfordUniversity) AS ?z) .

[?z, :salary, "55000"^^xsd:integer] :-
    BIND(SKOLEM("Employment", :charlie, :oxfordUniversity) AS ?z) .

[?z, :salary, "40000"^^xsd:integer] :-
    BIND(SKOLEM("Employment", :charlie, :acme) AS ?z) .

[?z, :salary, "45000"^^xsd:integer] :-
    BIND(SKOLEM("Employment", :bob, :acme) AS ?z) .
```

Note that each of these rules uses the SKOLEM construct in the antecedent to make sure that they match correctly to the generated triples listed above.

To check that the salary data has been inserted correctly, we can issue the query

```
select ?x (SUM(?y) AS ?income)
where {
    ?e :employee ?x . ?e :salary ?y
}
group by ?x
```

which gives us the total yearly income for each person by summing up the salary of each of their employments, giving the expected results.


```
:alice   60000 .
:charlie 95000 .
:bob     45000 .
```

Observations

The data restructuring and reification pattern has multiple applications. In particular, RDF can only represent directly binary relations and hence the representation of higher arity relations is only possible through reification. Reification is also needed if we want to qualify or annotate edges in a graph (e.g., by adding weights, or dates, or other relevant properties).

The Ordering Pattern

Intent

To find the first and last elements in an order.

Description

Many relations naturally imply some sort of order, and in such cases we are often interested in finding the first and last elements of such orders. For instance, consider the managerial structure of a company.

```
@prefix : <https://oxfordsemantic.tech/RDFox/tutorial/> .

:alice  :manages :bob .
:bob    :manages :jeremy .
:bob    :manages :emma .
:emma   :manages :david .
:jeremy :manages :monica .
```

We would like to recognise which individuals in the company are “top level managers”. We can use a rule to define a “top level manager” as a person who manages someone and is not managed by anyone else.

```
[?x, rdf:type, :topLevelManager] :-
    [?x, :manages, ?y],
    not exists ?z in ([?z, :manages, ?x]) .
```

The query

```
select ?x  where {?x rdf:type :topLevelManager}
```

asking for the list of top level managers gives as :alice as the answer. We can now use a rule to define “junior employees” as those who have a manager but who themselves do not manage anyone else.

```
[?x, rdf:type, :juniorEmployee] :-
    [?y, :manages, ?x],
    not exists ?z in ([?x, :manages, ?z]) .
```

The query


```
select ?x  where {?x rdf:type :juniorEmployee}
```

Gives us :monica and :david as answers.

Observations

Prominent examples of ordered relations where we may be interested in finding the top and bottom elements are partonomies (part-whole relations) and is-a hierarchies. It is also worth noticing that this pattern is not expressible in OWL 2.

The Clique Representative Pattern

Intent

To create a representative for a set of objects that have been identified as equal during data integration.

Description

When integrating data from multiple sources using a knowledge graph, it is usually the case that objects from different sources are identified to be the same.

In this setting, we want to be able to answer complex queries that span across the different sources, and to easily identify the source where the information came from.

For instance, assume that we are integrating sources s1, s2, and s3 containing information about music artists and records. Assume that we have determined (e.g., using entity resolution techniques or exploiting explicit links between the sources) that "John Doe" in s1 is the same as "J. H. Doe" in s2 and "The Blues King" in s3. We can represent these correspondences using a binary relation "ost:same" which we define as reflexive, symmetric, and transitive using RDFox rules as given next.

```
@prefix  ost:<https://oxfordsemantic.tech/> .
@prefix s1:<https://source1> .
@prefix s2:<https://source2> .
@prefix s3:<https://source3> .

s1:john_doe   rdf:type s1:artist .
s2:john_H_doe rdf:type s2:performer .
s3:blues_king rdf:type s3:musician .
s1:john_doe   ost:same s2:john_H_doe .
s1:john_doe   ost:same s3:blues_king .
s2:john_H_doe ost:same s3:blues_king .

[?x, ost:same, ?x] :- [?x, ost:same, ?y] .
[?y, ost:same, ?x] :- [?x, ost:same, ?y] .
[?x, ost:same, ?z] :- [?x, ost:same, ?y], [?y, ost:name, ?z] .
```

In this way, the aforementioned objects form a clique in the integrated graph. Indeed, the query

```
select ?x ?y  where {?x ost:same ?y }
```

returns the answer


```
s3:blues_king   s2:john_H_doe .
s2:john_H_doe   s3:blues_king .
s2:john_H_doe   s2:john_H_doe .
s3:blues_king   s3:blues_king .
s2:john_H_doe   s1:john_doe   .
s3:blues_king   s1:john_doe   .
s1:john_doe     s1:john_doe   .
s1:john_doe     s3:blues_king .
s1:john_doe     s2:john_H_doe .
```

In order to be able to query across artists from different sources, we want to define a unique representative for the elements in the clique. A plausible strategy is to first select the smallest individual according to some pre-defined total order (the order itself is irrelevant, and we can choose for example the order of resource IDs within RDFox). To select the smallest object, we introduce the following rules.

```
[?x, ost:comesbefore, ?y] :-
    [?x, ost:same, ?y], FILTER (?x < ?y)  .

[?y, rdf:type, ost:NotSmallestInClique] :-
    [?x, ost:comesbefore, ?y] .

[?x, rdf:type, ost:SmallestInClique] :-
    [?x, ost:comesbefore, ?y],
    NOT [?x, rdf:type, ost:NotSmallestInClique] .
```

The first rule generates an order amongst the elements of the clique. The second rule says that if ?x comes before ?y then ?y is not the smallest element. The third rule finally identifies the smallest element in the clique. The following query

```
select ?x ?y  where {?x ost:comesbefore ?y}
```

reveals the generated order

```
s2:john_H_doe s3:blues_king .
s1:john_doe   s2:john_H_doe .
s1:john_doe   s3:blues_king .
```

where s1:john_doe is correctly identified as the smallest element by the query.

```
select ?x  where {?x rdf:type ost:SmallestInClique}
```


Now that we have identified an element of the clique, we can create a representative of the clique using a Skolem constant, as given next.

```
[?z, rdf:type, ost:Artist],
[?z, ost:represents, ?x]  :-
    [?x, rdf:type, ost:SmallestInClique],
    BIND(
        SKOLEM("OSTArtist", ?x) AS ?z
    ) .

[?x, ost:represents, ?z] :-
    [?x, ost:represents, ?y],
    [?y, ost:comesbefore, ?z] .
```

The first rule creates the Skolem constant and states that it represents the smallest element. The second rule states that the Skolem constant also represents every other element in the clique.

The query

```
select ?z ?x  where {?z ost:represents ?x}
```

Yields the expected result.

```
_:OSTArtist_2136 s2:john_H_doe .
_:OSTArtist_2136 s3:blues_king .
_:OSTArtist_2136 s1:john_doe .
```

Observations

It is possible to achieve the same results by using an "optimised" version of the pattern that generates fewer triples.

In particular, this optimised representation avoids axiomatising the ost:same property as reflexive and symmetric. Let's reconsider the data.

```
s1:john_doe   rdf:type s1:artist .
s2:john_H_doe rdf:type s2:performer .
s3:blues_king rdf:type s3:musician .
s1:john_doe   ost:same s2:john_H_doe .
s1:john_doe   ost:same s3:blues_king .
s2:john_H_doe ost:same s3:blues_king .
```


We now redefine directly the “comesBefore” relation using the following rules

```
[?x, ost:comesBefore, ?y] :-
    [?x, ost:same, ?y], FILTER(?x > ?y) .

[?x, ost:comesBefore, ?y] :-
    [?y, ost:same, ?x], FILTER(?x > ?y) .

[?x, ost:comesBefore, ?z] :-
    [?x, ost:comesBefore, ?y],
    [?y, ost:comesBefore, ?z],
    FILTER(?x > ?y) .

[?x, ost:comesBefore, ?y] :-
    [?z, ost:comesBefore, ?x],
    [?z, ost:comesBefore, ?y],
    FILTER(?x > ?y) .
```

The query

```
select ?x ?y  where {?x ost:comesBefore ?y}
```

reveals a generated order.

Once we have the order, the pattern works as before.

Named Graphs

Named Graphs are an RDF construct for organizing graph datasets into subgraphs. In addition to the default graph, a graph dataset may contain any number of named graphs, which, as the name suggests, are simply graphs associated with names (i.e. IRIs). This allows for the logical partitioning of the application's knowledge graph into subgraphs each representing a different aspect of the particular domain. So, for example, the data about an organization can contain a subgraph about the organizational structure (e.g. information about employees, their positions, their line managers, etc.), and another subgraph containing payroll information about employees. Structuring graph datasets using named graphs, allows, among other things, for higher modelling flexibility and fine-grained access control over the different parts of the graph data.

Importing Named Graphs from Trig Files

Graph datasets with named graphs are usually serialized in the RDF format TriG, which is an extension of Turtle, with information about graphs. The following is an example of a graph dataset with two graphs containing information about employees and their salaries.

```
# TriG File
# prefixes definition

# graph definition
:employees {
    :Monica a :Employee;
        :position :StoreManager;
        :manages  :John.

    :John a :Employee;
        :position :salesAssistant.
}
:payroll {
    :Monica :hasSalary "32000"^^xsd:integer.
    :John   :hasSalary "25000"^^xsd:integer.
}
```

Importing this file into RDFox will automatically create the two named graphs :employees and :payroll.

This data can be queried using SPARQL by specifying the graph for each triple pattern as shown below.

```
SELECT ?Person WHERE { GRAPH :employees {?Person :position :StoreManager}}
```

Manual Creation of Named Graphs

In RDFox, named graphs are simply a special type of a tuple table, the abstraction RDFox uses for a collection of tuples. Thus, in order to manually create a named graph in RDFox, one simply adds a tuple table of type triples. So, for example, the following command creates a named graph called :business.


```
tupletable add :business type triples
```

Named Graphs in Rules

The data in named graphs can be accessed in rules in much the same way as the data in the default graph. Namely, triples in named graphs can be used in bodies of rules, and triples can also be derived in named graphs as well. The syntax for accessing all subject-predicate-object triples in a graph named `:G` is `:G (?s ?p ?o)`. Rules can derive triples only in already created named graphs using one of the ways described above.

The following rule, for example, identifies as senior managers all employees with a salary greater than $30,000 and managing at least one person, and derives those tuples in the named graph :business.

```
:business(?Emp, a, :SeniorManager) :-
    :employees(?Emp, a, :Employee),
    :employees(?Emp, :manages, ?R),
    :payroll(?Emp, :hasSalary, ?salary),
    filter (?salary >= 30000).
```

As usual, once the rule is evaluated (i.e. imported in RDFox), the information about senior managers becomes available and can be accessed using SPARQL queries. So, for example, we can query for all senior managers using the following SPARQL query.

```
 SELECT ?Manager WHERE { GRAPH :business {?Manager a :SeniorManager } }
```

