RDF2Graph Documentation

Table of Content

Installation....4
requirements....4
setup....4
General usage....4
Parameters RDF2Graph....5
Cytoscape Exporter....6
OWL Exporter....6
SHEX Exporter....6
Additional notes for the exporters....6
Cytoscape Exporter: Visualization off the RDF structure in Cytoscape....6
The OWL exporter....8
The SHEX exporter....9
Overview internally used ontology....9
Class:....10
Class property:....10
Type Link:....10
Property:....11
Type:....11
Error:....11
Detailed description structure recovery process....12
Predicates collection (step 1)....14
Predicate statistics collection (step 2)....14
Predicate source reference error checking and reporting (step3)....14
Class collection (step4)....14
Class detail collection (step5)....14
Class statistics collection (step 5.1)....14
Check if class is not also used property (step 5.2)....14
Class property collection (step 5.3)....14

'type link' collection (step 5.3.1).....*15*

'type link' statistics collection (step 5.3.1.1).....*15*

'type link' forward multiplicity collection (step 5.3.1.2).....*15*

'type link' reverse multiplicity collection (step 5.3.1.3).....*16*

rdfs:SubClassOf collection (step 6).....**16**

Simplification (step 7).....**16**

Mark instantiated classes (step 7.1).....**17**

Load subClassOf tree into memory (step 7.2).....**17**

Count and store subClassOf instance counts (step 7.3).....**18**

Simplification per predicate (step 7.4).....**18**

Load predicate information (step 7.4.1).....*18*

Project down and merge destination types (step 7.4.2).....*18*

Keep branching only (step 7.4.3).....*18*

Remove root containing targets (step 7.4.4).....*19*

Store results (step 7.4.5).....*19*

Clean pre-simplification data (step 7.5).....**19**

Clean OWL classes (optional).....**19**

An optional step can be performed to remove any information about the OWL ontology related classes. The owl:Class class is ignored.....**19**

Known issues.....19

Installation

requirements

RDF2Graph is created and tested on Ubuntu. RDF2Graph needs Java 1.8, Jena, Cytoscape 3.2, Maven2, GIT, NodeJS and html2text.

```
#install java 8
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
sudo update-java-alternatives -s java-8-oracle
#maven2,git, nodejs and html2text
sudo apt-get install maven2 git nodejs html2text npm
```

To install jena, please download at Jena download. Extract to directory of choose.
To install Cytoscape, please download at Download Cytoscape.
Make download executable with
`chmod +x Cytoscape_3_2_0_unix.sh`
Execute installer and install to directory of choose.

Then add the following lines to `~/.bash.rc` then restart command console.

```
export JENAROOT=<your jena directory>
PATH=$JENAROOT/bin:$PATH
PATH=<your cytoscape path>:$PATH
```

setup

To install please execute the following commands:

```
git clone https://github.com/jessevdam/RDF2Graph
cd RDF2Graph
./setup.sh
```

General usage

The main RDF2Graph Java jar can be used to execute the recovery, which extracts the structure from an RDF resource and stores the results in a local database (Jena TDB). The simplification process can be enabled with the `--executeSimplify` option.
After completion of the recovery the results can exported to either to a network based view(Cytoscape), OWL or Shex. This can be done by using one of the exporter tools. The exporters can be found in the following directories: owlExporter, shexExporter and cytoscapeExporter. Each directory contains an executable export.sh bash script.

The queries executed by recovery process are computational demanding, so please do not execute the recovery directly on any public endpoint. Instead of executing it on a public endpoint set up your own SPARQL endpoint and load the data of the public endpoint. Please note that some versions of Virtuoso contain a bug, which make Virtuoso not return all results as it is should return. This problem could make RDF2Graph to either stop or generate incomplete results.

NOTE: The queries exectuted by RDF2Graph, contain limitation to limit the execution time, however the results might become incomplete.

NOTE!!: If the recovery fails or you want it to rerun it with different settings, please remove the output folder.
NOTE: If you get weird Java exceptions, please first try to remove the output folder.

Parameters RDF2Graph

RDF2Graph can be executed with:

```
java -jar RDF2Graph.jar <output directory> (<user>:<pass>@)?<URL of the SPARQL endpoint>([<uri of graph>])? --multiThread <number of threads> --all --eachThreadOnePort --collectPredicateStatistics --collectClassStatistics --collectShapePropertyStatistics --checkPredicateSourceClass --collectForwardMultiplicity --collectReverseMultiplicity --executeSimplify --treatSubClassOfAsIntanceOf --keepOWLClasses
```

Options

<output directory>: The directory in which the Jena TDB database will be stored, which contains the results of the structure recovery.

(<user>:<pass>)?@<uri of sparql endpoint>([<uri of graph>])?: Sparql endpoint on which the recovery has to be executed.

- <user>:<pass>@... User and password can be optionally given
- <URL fo the SPARQL endpoint> The full URL of the Sparql endpoint, for example http://127.0.0.1:8080/main/sparql
- ...[<uri of graph>] Optional parameter to specify the default graph.

--multiThread <number of threads>: Specify the number of concurrent Threads/requests that will be executed on the Sparql endpoint.

--eachThreadOnePort: In case your Sparql endpoint does not support multi threading, use this option to tell RDF2Graph to use for each concurrent Thread a different instance. If the main instance runs on port 7000 then the other instances has to be running on port 7001, 7002, …,7000 + <number of threads>.

--all: Same as --collectPredicateStatistics --collectClassStatistics --collectShapePropertyStatistics --checkPredicateSourceClass --collectForwardMultiplicity --collectReverseMultiplicity

--collectPredicateStatistics: Tell RDF2Graph to collect the predicate statistics.

--checkPredicateSourceClass: Tell RDF2Graph to check if all instances using a specific predicate have a type defined.

--collectClassStatistics: Tell RDF2Graph to collect the class statistics.

--collectShapePropertyStatistics: Tell RDF2Graph to collect the property statistics per class type.

--collectForwardMultiplicity: Tell RDF2Graph to collect the forward multiplicity of each type link.

--collectReverseMultiplicity: Tell RDF2Graph to collect the reverse multiplicity of each type link.

--treatSubClassOfAsIntanceOf: Tell RDF2Graph to consider the rdfs:subClassOf predicate to be also treated as a rdf:type predicate.

--executeSimplify: Perform the simplification step. This options can also be

```
applied on a already created RDF2Graph database, in which only the
simplification step will be executed.

--keepOWLClasses: Any recovered classes other then OWL:Class related to the OWL
ontology itself will be removed from the results.
```

Cytoscape Exporter

The cytoscape exporter can be executed with:

```
cytoscapeExporter/export.sh <input directory> <keep> <output>

<input directory>: The directory in which the Jena TDB database is stored, which
will contain the results of the structure recovery.

<keep>: options to specify whether sub classes for which no instances are found
should be included in the overview. Either specify 'true' or 'false'.

<output>: Specify output mode, which can be one of the following options
   •  view, opens Cytoscape and directly shows the network.
   •  <filename>.cys, creates cytoscape session file and stores it in the
      specified filename.
   •  <filename>.xgmml, creates one xgmml file(<filename>.xgmml) that contains
      the network view and one xgmml file(<filename>_error.xgmml) that contains
      the error report.
```

OWL Exporter

The OWL Exporter can be executed with:

```
owlExporter/export.sh echo <input directory> <include reverse cardinality>
<output>

<input directory>: The directory in which the Jena TDB database is stored, which
contains the results of the structure recovery.

<include reverse cardinality>: Tell the owl exporter to include the reverse
cardinalities in the owl file. Either specify 'true' or 'false'.

<output>: Specify the output owl file. The used output format is N3. Please use
Protoge to convert to other formats.
```

SHEX Exporter

The SHEX exporter can be executed with:

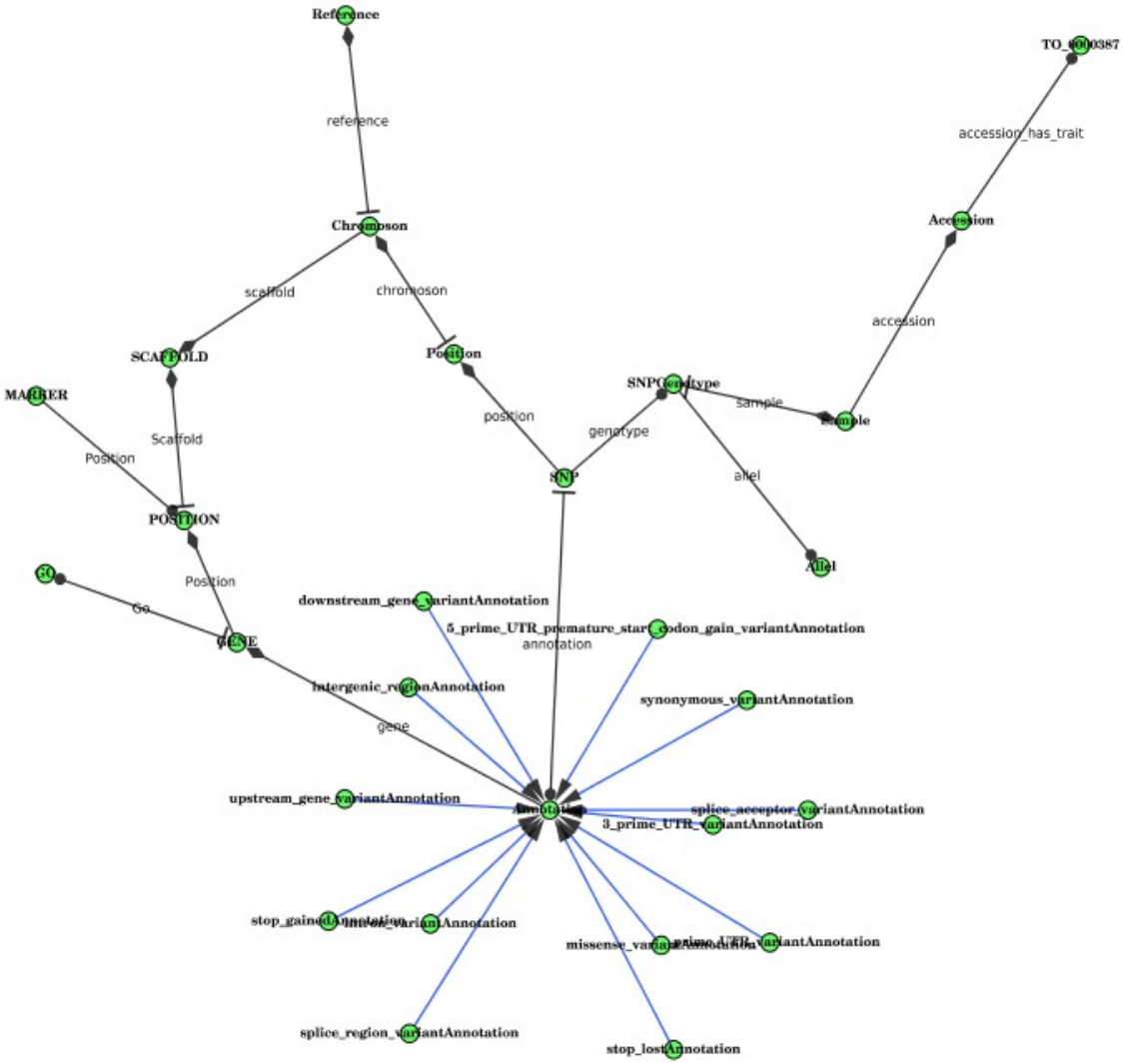
```
shexExporter/export.sh echo <input directory> <output>

<input directory>: The directory in which the Jena TDB database is stored, which
contains the results of the structure recovery.

<output>: Specify the output SHEX file.
```

Additional notes for the exporters

Cytoscape Exporter: Visualization off the RDF structure in Cytoscape

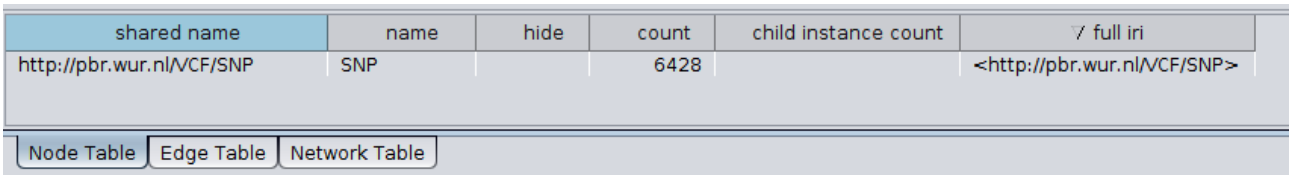
The following image give an example view of the network representation.

- Each node in the network represents one Class.
- Each connective black line represents an object project between two classes.
  - A diamand is presented if the forward multiplicity is 1..1 or 0..1
  - A round blob is presented if the forward multiplicity is 0..N or 1..N
  - A T-bar is presented if the reverse multiplicity is 1..N or x..N
- Each connective blue line represents an rdfs:subClassOf relationship between two classes.



When selecting a class node, the following details will be shown

- name, short name of the node as depicted in the network
- count, number of instances of the class
- child instance count, number of instances including the number of instances of all the child classes
- full IRI, the full IRI of the class
- other, please use the hide function to hide these columns



shared name	name	hide	count	child instance count	/ full iri
http://pbr.wur.nl/VCF/SNP	SNP		6428		<http://pbr.wur.nl/VCF/SNP>

Node Table | Edge Table | Network Table

Figure 1: Node information: table displaying the node information

To view all the properties of a certain class, you can use the 'select adjacent edges' (alt+e) function. This will show all edges including those linking to the hidden data type. All links to a data type such as for example xsd:integer and xsd:string are hidden. Note that also inward edges are also selected so please sort the 'source' column, this will make is easy to see which edges are the outgoing edges.

When using the 'select adjacent edges' or selecting an edge the following information will be shown.

- source, the name of the source class
- predicate name, the name of the predicate of the 'type link'
- destination type, the name of the target class or data type
- forward multiplicity, the forward multiplicity of the 'type link'
  - 1..1, each source instance as exactly one reference to the target (rs:Exactly-one)
  - 1..N, each source instance has at least one or many reference to the target (rs:One-or-may)
  - 0..1, only some source instances have one but no more then one reference to the target (rs:Zero-or-one)
  - 0..N, some source instances have one or even more then one reference to the target (rs:Zero-or-many)
- reverse multiplicity
  - 1..1, each target instance as exactly one reverse reference to the source (rs:Exactly-one)
  - 1..N, each target instance has at least one or many reverse reference to the source (rs:One-or-may)
  - 0..1, only some target instances have one but no more then one reverse reference to the source (rs:Zero-or-one)
  - 0..N, some target instances have one or even more then one reverse reference to the source (rs:Zero-or-many)
  - X..1, target instances have no more then one reverse reference to the source(RDF2Graph:x_1)
  - X..N, target instances can have more then one reverse references to the source(RDF2Graph:x_n)
- is_simple, true is target is a data type and false if target is a instance of a class
- full predicate, the full predicate of the 'type link'
- other, please use the hide function to hide these columns

sh...	sh...	name	inte...	source	predicate na...	destination t...	forwar...	revers...	refere...	is_sim...	full predicate
htt...	htt...	http://...	http:...	SNP	annotation	Annotation	0..N	1..N	14376	false	<http://pbr.wur.nl/VCF/annotation>
htt...	htt...	http://...	http:...	SNP	position	Position	1..1	1..1	6428	false	<http://pbr.wur.nl/VCF/position>
htt...	htt...	http://...	http:...	SNP	genotype	SNPGenotype	1..N	1..1	19284	false	<http://pbr.wur.nl/VCF/genotype>
htt...	htt...	http://...	http:...	SNP	allelySNP	string	1..N	X..N	7626	true	<http://pbr.wur.nl/VCF/allelySNP>
htt...	htt...	http://...	http:...	SNP	refSNP	string	1..1	X..N	6428	true	<http://pbr.wur.nl/VCF/refSNP>

Figure 2: 'type link' table: Table displaying the 'type link' information.

The OWL exporter

The owl exporter automatically creates the following axioms/items

- Object properties
  - domain definitions
  - range definitions
- Data type properties
  - domain definitions
  - range definitions
- class subClassof restrictions
  - 'all values from'
  - cardinality (min, max or exact)
  - inverse cardinality(min, max or exact)

The parent to child subClassOf definitions are included if present in the original RDF resource, but are not automatically generated by RDF2Graph.
If a predicate links to an external reference a data type property is included which has the 'xsd:anyURI' type.
NOTE: If a property is found to be both a object property and domain property it is exported as such. This is against the OWL specifications and so reasoners might not accept it or crash due to this.
NOTE: In case a base class has for predicate 'x' all values from integer, while a subclass has for predicate 'x' all values from string the reasoner while mark the base class to be equal to none.
NOTE: Links, which are defined as invalid are not exported in the OWL exporter.

The SHEX exporter

Not all elements as recovered do not yet have a complete or correct definition as the SHEX standard is still in development.

Overview internally used ontology

The following sections gives a overview of the ontology used in the internal database of the structure recovery tool. The structure/ontology is based on the concept of class properties and 'type links'. So a class can have multiple class properties and each class property can have multiple 'type links'. Each type link is a unique link defined as an unique tuple: `{subject type, predicate, object (data) type}` and so each type link connect to only one Type. So if class person has a predicate age, which is either an integer or string then the class person has one class property that has 2 'type links'. One that has integer as type and one that has string has type.

Additional files:

- owl ontology: doc\RDF2Graph.owl
- network view: doc\RDF2Graph.cys
- uml diagram: doc\RDF2Graph.dia

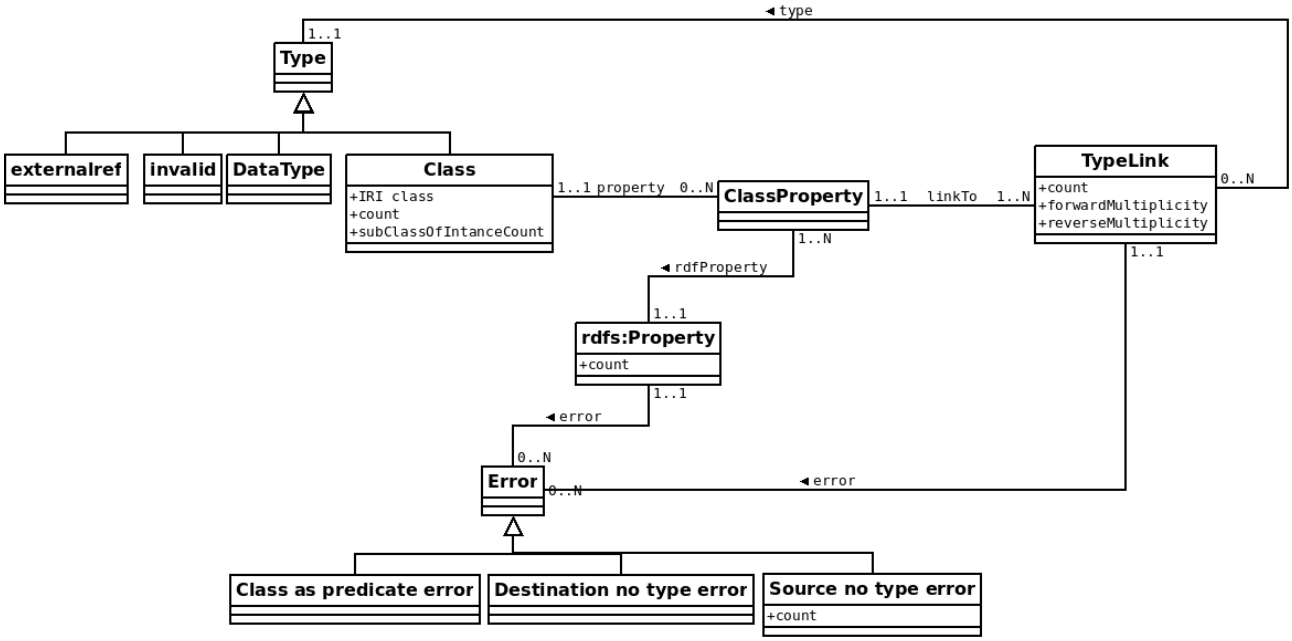


Figure 3: UML diagram: simplified uml diagram of the classes used in the RDF2Graph ontology

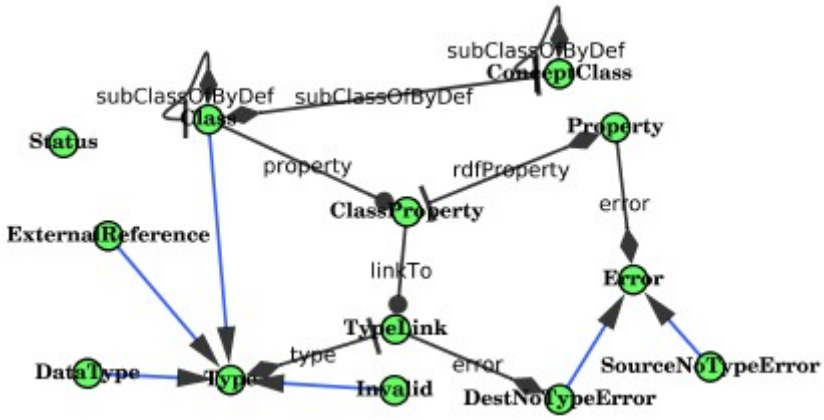


Figure 4: RDF2Graph result from the RDF2Graph internal datastructure

Class:

Properties:

- count: the number of instances of the class.
- subClassOfIntanceCount: the number of instances of all the sub classes of the class
- property: Each class has zero or more class properties.

NOTE: If limitation are included to limit the recovery process the count number can be capped to for example 100000.
NOTE: The subClassOfIntanceCount can be summation of capped instance counts. For example it could be equal to 100000(capped) + 100000(capped) + 24234 + 10 = 224244.

Class property:

Properties:

- linkTo: all 'type links' of the class property.
- rdfProperty: the predicate of the class property

Type Link:

Properties:

- count: the number of triples that belong to this type link.
- forwardMultiplicity: the forward multiplicity of this link, which can be:
  - 1..1, each source instance as exactly one reference to the target (rs:Exactly-one)
  - 1..N, each source instance has at least one or many reference to the target (rs:One-or-may)
  - 0..1, only some source instances have one but no more then one reference to the target (rs:Zero-or-one)
  - 0..N, some source instances have one or even more then one reference to the target (rs:Zero-or-many)
- reverseMultiplicity: the reverse/inverse multiplicity of this link, which can be:
  - 1..1, each target instance as exactly one reverse reference to the source (rs:Exactly-one)
  - 1..N, each target instance has at least one or many reverse reference to the source (rs:One-or-may)
  - 0..1, only some target instances have one but no more then one reverse reference to the source (rs:Zero-or-one)
  - 0..N, some target instances have one or even more then one reverse reference to the source (rs:Zero-or-many)
  - X..1, target instances have no more then one reverse reference to the source(RDF2Graph:x_1)
  - X..N, target instances can have more then one reverse references to the source(RDF2Graph:x_n)
- type: object type of the 'type link'

NOTE: The current implementation has no support for forward or reverseMultiplicity on class properties yet. So for example if a class Person has always an age defined (1..1 forward multiplicity), but if this can be either an integer or string then the forward multiplicity of each 'type link' will become 0..1.

Property:

Properties:

- count: total number of triples that contain this predicated

Type:

A type can be one of the following 4 types

- externalref: An object that have no defined type neither does it have any properties. This means its either a reference to an external database or a faulty link.
- Invalid: An object that have no defined type, but does have some properties. The type definition is missing.
- Simple: An object that is a data type, such as for example integer or string.
- Class: see description above.

Error:

Properties:

- rdfProperty: property to which the errors applies

RDF2Graph also detect and reports some errors, which can be one of the following types.

- Class as predicate error: Reported when a class type is also used as predicate.
- Destination no type error: Reported when a link links to an object that has no type defined. This also presented as an invalid type.

- Source no type error: Reported when the source of a link has no type defined. An additional property is included that report the number of links found with this condition for the specific RDF property.

NOTE: When these errors are found it can be that the structure recovery of the RDF2Graph tool is incomplete.

Detailed description structure recovery process

This section describes in detail the capabilities and functioning of the structure recovery tool. This guide can be use to tune and adapt RDF2Graph to your SPARQL endpoint and needs.

The figure at the following page gives an overview off the steps in the method to recover the structure and extract some useful statistics from a user specified database. Step 1 to 6 collect the information from the targeted RDF database and stores them into the local RDF database, whereas the optional step 7 simplifies the structure such that a nice overview can be generated.

The associated query within the queries/remote for each step 1 to 6 that use an SPARQL query that is executed on the remote endpoint is given. Does that are marked with a * contain some limitations to improve the running time of RDF2Graph. However, these limitation might result in some cases in incomplete results. This section can be used to adopt the queries to your SPARQL endpoint and tune the performance and limitations used. The current queries are optimized for both the Jena and Stardog endpoints. NOTE that there are 2 folders in the queries folder. The remote folder is for a default setting and the remoteSubClassOf folder is for the `--treatSubClassOfAsIntanceOf` setting.

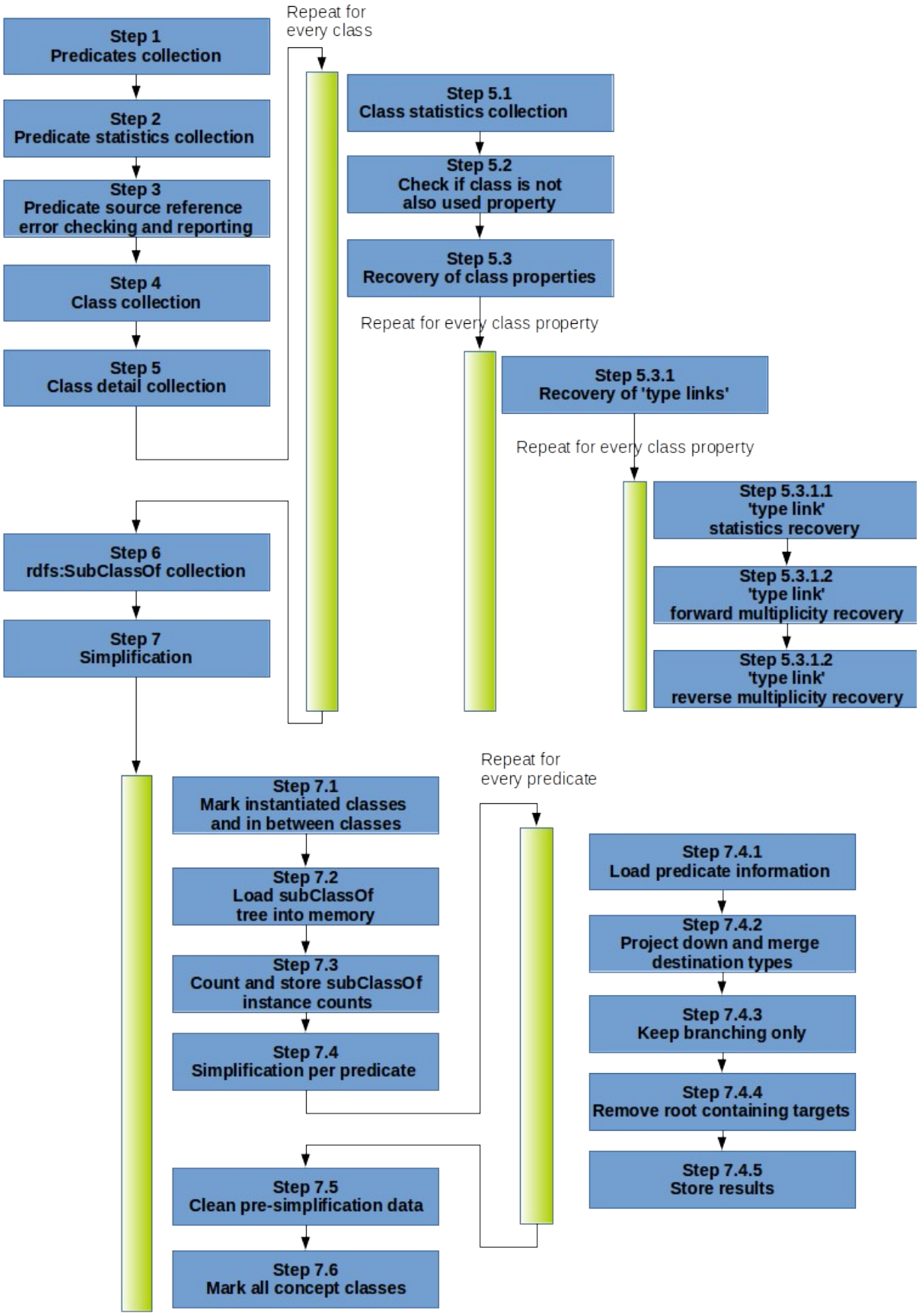


Figure 5: Overview of structrue recovery process RDF2Graph

Predicates collection (step 1)

All unique predicates in the database are recovered. Some predicates related to the RDF namespace are ignored. For each predicate found a property object(RDF2Graph:Class) is generated.

Query: getAllPredicates.txt

Predicate statistics collection (step 2)

For each predicate the number of occurrences of the predicate are counted and the resulting count is stored(RDF2Graph:count) under the property object.

Query: countPredicate.txt

Predicate source reference error checking and reporting (step3)

For each predicate a check is performed, which search for any subject referencing the predicate without having a class type definition. If found, an “source has not type definition”(RDF2Graph:SourceNoTypeError) error is created and linked(RDF2Graph:error) to the predicate object. Furthermore the number of erroneous subjects our counted and the resulting count is stored (RDF2Graph:count) under the error object.

When such an error exists information in the overview will be missing. The tool does not try to 'infer' the missing type and skips any references from these subjects.

Query*: testSourceTypesForPredicate.txt

Class collection (step4)

All classes that have 1 or more instances defined are recovered. Classes without any instances for example from an ontology like the Gene Ontology(GO) are not considered. For each class found a class(RDF2Graph:Class) object is generated.

Query: getAllClasses.txt

Class detail collection (step5)

The following sub step are performed for each class found in the previous step.

Class statistics collection (step 5.1)

The number of instances of the class are counted and the resulting count is stored(RDF2Graph:count) under the class object.

Query: countNumberOfClassInstances.txt

Check if class is not also used property (step 5.2)

If the class is also used as a predicate an class as predicate error object(RDF2Graph:ClassAsPredicateError) is generated and linked to the predicate object. When such an error is detected it could be that the structure recovery becomes incomplete.
Query: checkClassIsNotPredicate.txt

Class property collection (step 5.3)

All unique predicates referenced by the instances of the class are recovered. For each predicate

found a class property object(RDF2Graph:ClassProperty) is generated and linked(RDF2Graph:rdfProperty) to the predicate object. Furthermore the class object is linked(RDF2Graph:property) to the class property object. Then the following step is performed for each class property object.

This query can be either executed per class or executed as once depending on the `--useClassPredFindAtOnce` option.

Query*: getClassShapeProperties.txt (`--useClassPredFindAtOnce == false`)
Query: getClassAllShapeProperties.txt (`--useClassPredFindAtOnce == true`)

'type link' *collection (step 5.3.1)*

All unique value types referenced by the class property are recovered. The following 4 groups of value type(s) exist(s). (1) An subject, which is an instance of any class type. (2) An literal value of any literal data type. (3) A reference to an external resource, which defined as reference to an IRI that has no properties (RDF2Graph:externalref). (4) An subject, which has not class type defined, but has some properties. (RDF2Graph:invalid). Each class type and/or literal data type is considered as an unique reference. For each unique value type a 'type link' object (RDF2Graph:TypeLink) is generated and linked(RDF2Graph:type) the value type.
Furthermore the class property object is linked(RDF2Graph:linkTo) to the 'type link' object. The value type being either (1) a unique class type, (2)a unique literal data type, (3) external reference (RDF2Graph:externalref) or (4) invalid reference(RDF2Graph:invalid). In case it is an invalid reference an “destination has no type definition” error object(RDF2Graph:DestNoTypeError) is created and linked(RDF2Graph:error) to the predicate object. Then the following steps are performed for each 'type link' object.

Query*: getClassPropertyDetails.txt

If a invalid or external reference is found an additional check is performed.

Query: checkClassPropertyExternalRef.txt

'type link' statistics collection (step 5.3.1.1)

The number of occurrences of the 'type link' is counted and the resulting count is stored(RDF2Graph:count) under the 'type link' object.

Query*: getShapePropertyCount_<suffix>.txt
The suffix depends on 'type link' found.

- exref: used when the object is a external reference.
- ref: used when the object is an instance which has a class type definitions.
- simple: used when the object is a data type.

'type link' forward multiplicity collection (step 5.3.1.2)

The minimum and maximum multiplicity of the 'type link' are separately recovered. Then the minimum and maximum multiplicity are encoded into one of the following multiplicity types: each source instance as exactly one reference to the target (rs:Exactly-one), each source instance has at least one or many reference to the target (rs:One-or-may), only some source instances have one but no more then one reference to the target (rs:Zero-or-one) or some source instances have one or even more then one reference to the target (rs:Zero-or-many). The resulting encoded multiplicity is stored(RDF2Graph:forwardMultiplicity) under the 'type link' object.

Query*: getShapePropertyForwardMaxMultiplicity_<suffix>.txt and
getShapePropertyForwardMinMultiplicity_<suffix>.txt

'type link' reverse multiplicity collection (step 5.3.1.3)

The same method is used as for the forward multiplicity in which the source and target are inter changed. However, the minimum multiplicity can only be determined for a unique class type and the maximum multiplicity can only be determined for a unique class type, external reference and the literal data type string. When an undetermined minimum and maximum multiplicity is encountered it is encoded into one of the following multiplicity types: target instances have no more then one reverse reference to the source(RDF2Graph:x_1), target instances can have more then one reverse references to the source(RDF2Graph:x_n) or no reverse multiplicity can determined(RDF2Graph:none).
The resulting encoded multiplicity is stored(RDF2Graph:reverseMultiplicity) under the 'type link' object.

Query*: getShapePropertyReverseMaxMultiplicity_<suffix>.txt and
getShapePropertyReverseMinMultiplicity_ref.txt

rdfs:SubClassOf collection (step 6)

All rdfs:SubClassOf relationships are recovered and stored, which are used for the simplification step.

Query: getAllSubClassOf.txt

Simplification (step 7)

Some RDF databases contain many subclasses, each of these subclasses can have instances that contain the same predicate as the parent class. If no simplification step is performed then for each subclass containing the same predicate a link will generated. If many subclasses exists the network will become an unreadable hairball. To overcome this a simplification step is performed that both find the common source class as well the common target class for a set of 'type link' that share the same predicate. The following figure gives 3 types of example simplifications performed by the simplification step.

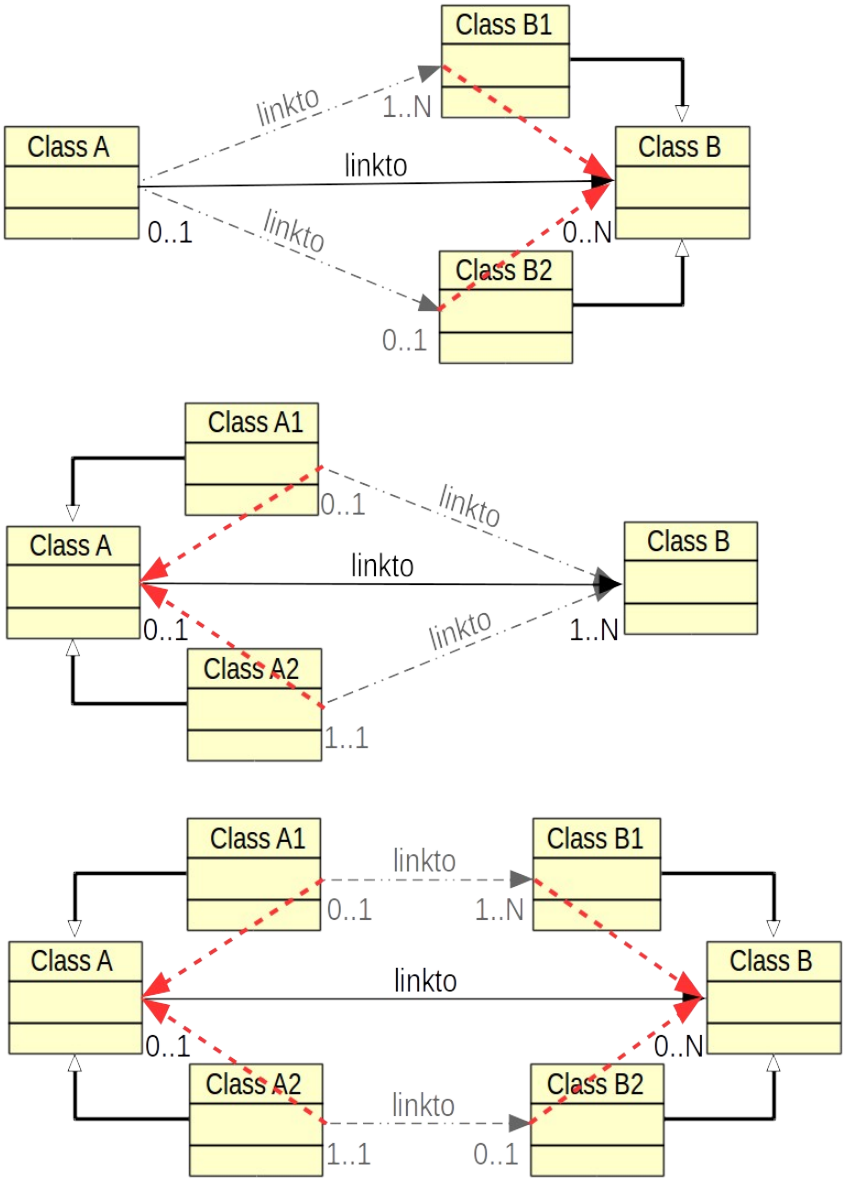


Figure 6: A. The red connection indicate the resulting connection after simplification in which for the source an common ancestor is found. '. B. The red connection indicate the resulting connection after simplification in which for both the target an common ancestor is found. C. The red connection indicate the resulting connection after simplification in which for both the source and the target an common ancestor is found.

This simplification process consists of the following steps.

Mark instantiated classes (step 7.1)

All classes that have either some instances or a child class that have some instances are marked as having been instantiated. A class is marked by making it an instance of RDF2Graph:Class.

Load subClassOf tree into memory (step 7.2)

All marked classes and associated subclassOf links are loaded into a memory based directed graph. This memory loaded class structure can then be used to load additional information and find a the common parent of two classes.

Count and store subClassOf instance counts (step 7.3)

For each class the number of instances of all its subclasses are counted and the resulting count is stored(RDF2Graph:subClassOfIntanceCount) under the class object.

Simplification per predicate (step 7.4)

The simplification process is executed per retrieved predicate. For each predicate following steps are performed.

Load predicate information (step 7.4.1)

The information for all 'type links' object that belong to the currently processed predicate are loaded into the tree. For each 'type link' object a reference object is added to the source class in the tree that points towards the the target class in the tree. The associated forward multiplicity, reverse multiplicity and count are loaded into the reference object.

The algorithm in the following steps will merge these reference objects. Upon merging two reference objects, the counts are added to each other, the forward multiplicities are merged and the reverse multiplicities are merged. Upon merging two multiplicities the new minimum per instance reference count will be equal to lowest per instance reference count of the merged multiplicities and the new maximum per instance reference count will be equal to highest per instance reference count of the merged multiplicities.

Project down and merge destination types (step 7.4.2)

This steps ensures that all source references contained within the child classes get copied into their parents classes after which the references gets merged if the target classes of the references that share a common parent.

This steps is based on a recursion depth first process over the sub class of tree in which the following steps are performed per class object. All references in the child classes are copied and added to the references of parent class. After which all references in the parent class are merged which each other. They are merged if the referenced target classes have a common parent class. The resulting merged reference will point to the first found common parent of the target classes. If the common parent is owl:Thing no merging is performed.

Keep branching only (step 7.4.3)

The previous steps also copies none shared references to the parent classes, this steps ensures that this unwanted behavior is undone and so any reference not shared by multiple child get removed again.

This step is based on a recursion breadth first process over the sub class of tree in which the following step is performed for each class in the tree.

- For each reference(varC) to a type in the currently processed class(varA) the following step is performed.
  - Count the number of direct subclasses(varB) of the currently processed class(varA) that applies to the following condition:
    - The direct subclass(varB) has at least one reference that targets a class that is either equal or a subclass of the class referenced by the currently processed reference(varC).
  - If the number is equal to one then remove the currently processed reference(varC) from the currently processed class(varA).

Remove root containing targets (step 7.4.4)

Step 7.4.2 does not remove the source references if they where merged in one of the parent classes, so this step cleans this merged references.

This steps is based on a recursion depth first process over the sub class of tree in which all the references that target the currently process class are removed if the apply to the following condition.

Any reference for which the target class is already referenced by the currently processed class or parent class of the currently processed class.

Store results (step 7.4.5)

In this step new class property and 'type links' are created for the newly calculated references. This includes the merged reference counts, forward and reverse multiplicities. Each newly created class property objects is marked such that they can be distinguished from the old ones.
At last the in memory sub class of tree get cleaned so that the next predicate can be processed.

Clean pre-simplification data (step 7.5)

All the class property and associated 'type links' inserted before the simplification step are removed and the mark put on the new class property objects are removed.

Mark all concept classes (step 7.6)

All the classes that do have any instances or that to not have an sub classes that have some instances are marked as RDF2Graph:ConceptClass.

Clean OWL classes (optional)

An optional step can be performed to remove any information about the OWL ontology related classes. The owl:Class class is ignored.

Known issues

The simplification process miscalculates the minimum cardinality in case a class has multiple subclasses of which some have a predicate 'x' with a minimum cardinality of 1 while other subclasses to not have the predicate at all. The minimum cardinality should become 0, however 1 is kept.