RDFSharp

Reference Guide (v2.13)

Introduction 3

History of the project 3

Working with RDF models 4

Creating resources, literals and triples 4

Creating, manipulating and exchanging graphs 5

Working with namespaces, datatypes and vocabularies 9

Working with containers and collections 12

The problem of reification 13

Working with SHACL shapes 14

Creating shapes, targets and constraints 14

Validating graphs and reporting results 20

Working with RDF stores 21

Tracking provenance of RDF data with contexts and quadruples 21

Creating and manipulating stores 22

Backing stores on SQL engines 25

Working with SPARQL queries 26

Mirella Engine (Basics): variables, patterns, pattern groups 26

Mirella Engine (Basics): filters, modifiers 28

Mirella Engine (Advanced): property paths 31

Mirella Engine (Advanced): subqueries, aggregators, values 32

Working with federations: seamlessly integrated RDF data 35

Creating and executing Select queries 36

Creating and executing Ask queries 37

Creating and executing Construct queries 38

Creating and executing Describe queries 39

Connecting to SPARQL endpoints 40

Mirella Engine: tips & tricks 41

INTRODUCTION

HISTORY OF THE PROJECT

RDFSharp is a **lightweight** C# framework which can be used to realize applications, services and websites capable of modeling, storing and querying **RDF/SPARQL** data. It can also be used for RDF data validation through modeling of **SHACL** shapes.

The project was launched on Microsoft Codeplex on June 2012 with the goal of providing the .NET community with a **friendly** entry point for easily start working with RDF and Semantic Web concepts.

The project is hosted on *GitHub* (https://github.com/mdesalvo/RDFSharp)

Binaries are also available on *NuGet* (http://www.nuget.org/packages?q=rdfsharp)

The project delivers a **netstandard2.0** library containing the following namespaces:

Namespace	**Description**
RDFSharp.**Model**	Creation, management and exchange of **RDF graphs** Creation and validation of **SHACL shapes**
RDFSharp.**Store**	Creation and management of **RDF stores** and **federations**
RDFSharp.**Query**	Creation and execution of **SPARQL queries**
RDFSharp.**Semantics**	Creation and validation of **OWL-DL/SKOS ontologies**

The project also controls an extension which targets different RDF storage solutions:

RDFSharp.Extensions	Extensions for RDF data storage and querying (**SQLServer**,**SQLite**,**Firebird**,**MySQL**,**PostgreSQL**,**Oracle**) (https://github.com/mdesalvo/RDFSharp.Extensions)

WORKING WITH RDF MODELS

CREATING RESOURCES, LITERALS AND TRIPLES

A resource is an URI-named concept, modeled as instance of **RDFResource**:

```
// CREATE RESOURCE
var donaldduck = new RDFResource("http://www.waltdisney.com/donald_duck");

// CREATE BLANK RESOURCE
var disney_group = new RDFResource();
```

The ***IsBlank*** property is a flag informing about nature of the resource. A blank resource carries an automatically generated Guid-based Uri starting with *bnode* prefix.

A literal is a textual piece of information, modeled as instance of **RDFPlainLiteral** or **RDFTypedLiteral**, depending on the semantic of represented data:

```
// CREATE PLAIN LITERAL
// "Donald Duck"
var donaldduck_name = new RDFPlainLiteral("Donald Duck");

// CREATE PLAIN LITERAL WITH LANGUAGE TAG
// "Donald Duck"@en-US
var donaldduck_name_enus = new RDFPlainLiteral("Donald Duck", "en-US");

// CREATE TYPED LITERAL
// "85"^^xsd:integer
var mickeymouse_age = new RDFTypedLiteral("85", RDFModelEnums.RDFDatatypes.XSD_INTEGER);
```

A triple is an elementary assertion about a resource, modeled as instance of **RDFTriple**:

```
// CREATE TRIPLES
// "Mickey Mouse is 85 years old"
RDFTriple mickeymouse_is85yr
    = new RDFTriple(
          new RDFResource("http://www.waltdisney.com/mickey_mouse"),
          new RDFResource("http://xmlns.com/foaf/0.1/age"),
          new RDFTypedLiteral("85", RDFModelEnums.RDFDatatypes.XSD_INTEGER));

// "Donald Duck has english (US) name "Donald Duck""
RDFTriple donaldduck_name_enus
    = new RDFTriple(
          new RDFResource("http://www.waltdisney.com/donald_duck"),
          new RDFResource("http://xmlns.com/foaf/0.1/name"),
          new RDFPlainLiteral("Donald Duck", "en-US"));
```

It is made up of a *Subject* part (the resource being described), a *Predicate* part (the resource being the verb of description) and an *Object* part (the resource, or the literal, being the knowledge described about the subject). Depending on the nature of the object, a triple assumes a **SPO** or **SPL** flavor represented by the *TripleFlavor* property.

Important: when creating a triple, validation against presence of a blank resource in the predicate position occurs, throwing a **RDFModelException** in case of detection.

CREATING, MANIPULATING AND EXCHANGING GRAPHS

A graph is an URI-named collection of triples, modeled as instance of **RDFGraph**:

```
// CREATE EMPTY GRAPH
var waltdisney = new RDFGraph();

// CREATE GRAPH FROM A LIST OF TRIPLES
var triples = new List<RDFTriple>({mickeymouse_is85yr, donaldduck_name_enus});
var waltdisney_filled = new RDFGraph(triples);
```

Its *Context* property is a non-blank URI for declaring the Web provenance of the graph data. It is initialized to the value of the *DefaultNamespace* static property of the **RDFNamespaceRegister** class, but it can be modified through the ***SetContext*** method:

```
// SET CONTEXT OF A GRAPH
waltdisney.SetContext(new Uri("http://waltdisney.com/"));
```

A graph can be easily transformed into an ADO.NET datatable with *Subject-Predicate-Object* columns through the ***ToDataTable*** method, but can also be obtained back through the static ***RDFGraph.FromDataTable*** method:

```
// GET A DATATABLE FROM A GRAPH
var waltdisney_table = waltdisney.ToDataTable();

// GET A GRAPH FROM A DATATABLE
var waltdisney_newgraph = RDFGraph.FromDataTable(waltdisney_table);
```

RDFGraph exposes a *TriplesEnumerator* and is compatible with *IEnumerable<RDFTriple>*:

```
// ITERATE TRIPLES OF A GRAPH WITH FOREACH
foreach (var t in waltdisney) {
    Console.Writeline("Triple:      " + t);
    Console.Writeline("  Subject:   " + t.Subject);
    Console.Writeline("  Predicate: " + t.Predicate);
    Console.Writeline("  Object:    " + t.Object);
}

// ITERATE TRIPLES OF A GRAPH WITH ENUMERATOR
var triplesEnum = waltdisney.TriplesEnumerator;
while (triplesEnum.MoveNext()) {
    Console.Writeline("Triple:      " + triplesEnum.Current);
    Console.Writeline("  Subject:   " + triplesEnum.Current.Subject);
    Console.Writeline("  Predicate: " + triplesEnum.Current.Predicate);
    Console.Writeline("  Object:    " + triplesEnum.Current.Object);
}
```

The *TriplesCount* property represents the number of triples contained in the graph:

```
// GET COUNT OF TRIPLES CONTAINED IN A GRAPH
var triplesCount = waltdisney.TriplesCount;
```

There are several ways to manage the collection of triples of a graph:

INSERT	***AddTriple*** Adds the given triple.	`waltdisney.AddTriple(mickeymouse_is85yr);`
DELETE	***RemoveTriple*** Removes the given triple.	`waltdisney.RemoveTriple(donaldduck_name_enus);`
	RemoveTriplesBySubject Removes the triples having the given resource as "Subject".	`waltdisney.RemoveTriplesBySubject(donaldduck);`
	RemoveTriplesByPredicate Removes the triples having the given resource as "Predicate".	`waltdisney.RemoveTriplesByPredicate(new RDFResource("http://xmlns.com/foaf/0.1/age"));`
	RemoveTriplesByObject Removes the triples having the given resource as "Object".	`waltdisney.RemoveTriplesByObject(goofygoof);`
	RemoveTriplesByLiteral Removes the triples having the given literal as "Object".	`waltdisney.RemoveTriplesByLiteral( mickeymouse_age);`
	ClearTriples Empties the graph.	`waltdisney.ClearTriples();`
SEARCH	***ContainsTriple*** Checks if the given triple exists in the graph.	`Boolean checkTriple = waltdisney.ContainsTriple(mickeymouse_is85yr);`
	SelectTriplesBySubject Returns a graph containing the triples having the given resource as "Subject".	`RDFGraph triples_by_subject = waltdisney.SelectTriplesBySubject(donaldduck);`

	SelectTriplesByPredicate Returns a graph containing the triples having the given resource as "Predicate".	`RDFGraph triples_by_predicate = waltdisney.SelectTriplesByPredicate(new RDFResource("http://xmlns.com/foaf/0.1/name");`
	SelectTriplesByObject Returns a graph containing the triples having the given resource as "Object".	`RDFGraph triples_by_object = waltdisney.SelectTriplesByObject(goofygoof);`
	SelectTriplesByLiteral Returns a graph containing the triples having the given literal as "Object".	`RDFGraph triples_by_literal = waltdisney.SelectTriplesByLiteral( mickeymouse_age);`
SET	***IntersectWith*** Returns a graph containing the triples found both in this graph and in the given one.	`RDFGraph intersection_triples = waltdisney.IntersectWith(waltdisney_filled);`
	UnionWith Returns a graph containing the triples of this graph and the given one.	`RDFGraph union_triples = waltdisney.UnionWith(waltdisney_filled);`
	DifferenceWith Returns a graph containing the triples of this graph which are not found in the given one.	`RDFGraph difference_triples = waltdisney.DifferenceWith(waltdisney_filled);`

Graph operations are **fluent**, so they can be composed to achieve more complex tasks:

```
// MULTIPLE SELECTIONS
var multiple_selections_graph =
waltdisney.SelectTriplesBySubject(new RDFResource("http://www.waltdisney.com/donald_duck"))
          .SelectTriplesByPredicate(new RDFResource("http://xmlns.com/foaf/0.1/name"));

// SET OPERATIONS
var set_operations_graph = waltdisney.IntersectWith(waltdisney_filled).UnionWith(another_graph);
```

The content of a graph can be read from, or saved to, an RDF **file** or **stream**. RDFSharp supports **4** standard RDF graph data serialization formats:

Format	Read	Write	

N-TRIPLES	YES	YES	`var ntriplesFormat = RDFModelEnums.RDFFormats.NTriples;`
			
			`// READ N-TRIPLES FILE`
			`var graph = RDFGraph.FromFile(ntriplesFormat, "C:\\file.nt");`
			
			`// READ N-TRIPLES STREAM`
			`var graph = RDFGraph.FromStream(ntriplesFormat, inStream);`
			
			`// WRITE N-TRIPLES FILE`
			`graph.ToFile(ntriplesFormat, "C:\\newfile.nt");`
			
			`// WRITE N-TRIPLES STREAM`
			`graph.ToStream(ntriplesFormat, outStream);`

TRIX	YES	YES	`var trixFormat = RDFModelEnums.RDFFormats.TriX;`
			
			`// READ TRIX FILE`
			`var graph = RDFGraph.FromFile(trixFormat, "C:\\file.trix");`
			
			`// READ TRIX STREAM`
			`var graph = RDFGraph.FromStream(trixFormat, inStream);`
			
			`// WRITE TRIX FILE`
			`graph.ToFile(trixFormat, "C:\\newfile.trix");`
			
			`// WRITE TRIX STREAM`
			`graph.ToStream(trixFormat, outStream);`

TURTLE	YES	YES	`var turtleFormat = RDFModelEnums.RDFFormats.Turtle;`
			
			`// READ TURTLE FILE`
			`var graph = RDFGraph.FromFile(turtleFormat, "C:\\file.ttl");`
			
			`// READ TURTLE STREAM`
			`var graph = RDFGraph.FromStream(turtleFormat, inStream);`
			
			`// WRITE TURTLE FILE`
			`graph.ToFile(turtleFormat, "C:\\newfile.ttl");`
			
			`// WRITE TURTLE STREAM`
			`graph.ToStream(turtleFormat, outStream);`

RDF/XML	YES *	YES	`var xmlFormat = RDFModelEnums.RDFFormats.RdfXml;`
			
			`// READ RDF/XML FILE`
			`var graph = RDFGraph.FromFile(xmlFormat, "C:\\file.rdf");`
			
			`// READ RDF/XML STREAM`
			`var graph = RDFGraph.FromStream(xmlFormat, inStream);`
			
			`// WRITE RDF/XML FILE`
			`graph.ToFile(xmlFormat, "C:\\newfile.rdf");`
			
			`// WRITE RDF/XML STREAM`
			`graph.ToStream(xmlFormat, outStream);`

***** RDF/XML reader does not support nested resource descriptions. In order to not loose data, please ensure your input RDF/XML file has **simple descriptions (1-level depth)** as officially recommended by W3C (https://www.w3.org/wiki/SimpleRdfXml)

```
<rdf:Description rdf:about="http://www.waltdisney.com/donald_duck">
    <foaf:name xml:lang="en-US">Donald Duck</foaf:name>
    <rdf:type rdf:about="http://www.waltdisney.com/character" />
</rdf:Description>
```

WORKING WITH NAMESPACES, DATATYPES AND VOCABULARIES

A namespace is an URI associated to a mnemonic prefix, modeled as instance of **RDFNamespace**. It can be used to speed up the creation of resources and for shortcutting URIs when serializing RDF data into a file format like RDF/XML or Turtle:

```
// CREATE NAMESPACE
var waltdisney_ns = new RDFNamespace("wd", "http://www.waltdisney.com/");

// USE NAMESPACE IN RESOURCE CREATION
var duckburg = new RDFResource(waltdisney_ns + "duckburg");
var mouseton = new RDFResource(waltdisney_ns + "mouseton");
```

RDFSharp has built-in support for **12** of the most common LinkedData namespaces:

DC (and extensions)	http://purl.org/dc/elements/1.1/
EARL	http://www.w3.org/ns/earl#
FOAF	http://xmlns.com/foaf/0.1/
GEO	http://www.w3.org/2003/01/geo/wgs84_pos#
OWL	http://www.w3.org/2002/07/owl#
RDF	http://www.w3.org/1999/02/22-rdf-syntax-ns#
RDFS	http://www.w3.org/2000/01/rdf-schema#
SHACL	http://www.w3.org/ns/shacl#
SIOC	http://rdfs.org/sioc/ns#
SKOS (and extensions)	http://www.w3.org/2004/02/skos/core#
XML	http://www.w3.org/XML/1998/namespace#
XSD	http://www.w3.org/2001/XMLSchema#

Namespaces are collected in a class named **RDFNamespaceRegister**, which represents a singleton in-memory register for storing both built-in and custom namespaces.

There are several ways to manage the collection of namespaces of this register:

INSERT	***AddNamespace*** Adds the namespace to the register.	`RDFNamespaceRegister.AddNamespace(waltdisney_ns);`
DELETE	***RemoveByUri*** Removes the namespace having the given Uri.	`RDFNamespaceRegister.RemoveByUri("http://mysite.org");`
	RemoveByPrefix Removes the namespace having the given prefix.	`RDFNamespaceRegister.RemoveByPrefix("mysite");`

SEARCH	***GetByUri***
	Retrieves a namespace by seeking presence of its Uri (null if not found). Supports **prefix.cc**
		`var ns = RDFNamespaceRegister.GetByUri("http://dbpedia.org", false); //local search`
		`var ns = RDFNamespaceRegister.GetByUri("http://dbpedia.org", true); //search prefix.cc service if no result`

	GetByPrefix
	Retrieves a namespace by seeking presence of its prefix (null if not found). Supports **prefix.cc**
		`var ns = RDFNamespaceRegister.GetByPrefix("dbpedia", false); //local search`
		`var ns = RDFNamespaceRegister.GetByPrefix("dbpedia", true); //search prefix.cc service if no result`

The namespace register has a static property called *DefaultNamespace* which represents the namespace automatically adopted for graphs not specifying their *Context* property:

```
// GET DEFAULT NAMESPACE
var nSpace = RDFNamespaceRegister.DefaultNamespace;

// SET DEFAULT NAMESPACE
RDFNamespaceRegister.SetDefaultNamespace(waltdisney_ns); //new graphs will default to this context
```

The predefined value of this property is: **https://rdfsharp.codeplex.com/**

RDFNamespaceRegister is compatible with *IEnumerable<RDFNamespace>*:

```
// ITERATE NAMESPACES OF REGISTER WITH FOREACH
foreach (var ns in RDFNamespaceRegister.Instance) {
    Console.Writeline("Prefix:    " + ns.Prefix);
    Console.Writeline("Namespace: " + ns.Namespace);
}

// ITERATE NAMESPACES OF REGISTER WITH ENUMERATOR
var nspacesEnum = RDFNamespaceRegister.NamespacesEnumerator;
while (nspacesEnum.MoveNext()) {
    Console.Writeline("Prefix:    " + nspacesEnum.Current.Prefix);
    Console.Writeline("Namespace: " + nspacesEnum.Current.Namespace);
}
```

Each of the predefined namespaces has a vocabulary in the **RDFVocabulary** class:

```
// CREATE TRIPLES WITH VOCABULARY FACILITIES
// "Goofy Goof is 82 years old"
RDFTriple goofygoof_is82yr
      = new RDFTriple(
            new RDFResource(new Uri("http://www.waltdisney.com/goofy_goof")),
            RDFVocabulary.FOAF.AGE,
            new RDFPlainLiteral("82")
        );

// "Donald Duck knows Goofy Goof"
RDFTriple donaldduck_knows_goofygoof
      = new RDFTriple(
            new RDFResource(new Uri("http://www.waltdisney.com/donald_duck")),
            RDFVocabulary.FOAF.KNOWS,
            new RDFResource(new Uri("http://www.waltdisney.com/goofy_goof"))
        );
```

Datatypes are decorators which can be applied to the value of literals in order to give them a strongly typed nature. RDFSharp supports **41** predefined XML Schema datatypes, exposed through the **RDFModelEnums.RDFDatatypes** enumeration:

```
// CREATE TYPED LITERALS
var myAge = new RDFTypedLiteral("34", RDFModelEnums.RDFDatatypes.XSD_INT);
var myDate = new RDFTypedLiteral("2017-01-07", RDFModelEnums.RDFDatatypes.XSD_DATE);
var myDateTime = new RDFTypedLiteral("2017-01-07T23:11:05", RDFModelEnums.RDFDatatypes.XSD_DATETIME);
var myXml = new RDFTypedLiteral("<book>title</book>", RDFModelEnums.RDFDatatypes.RDF_XMLLITERAL);
var myLiteral = new RDFTypedLiteral("generic literal", RDFModelEnums.RDFDatatypes.RDFS_LITERAL);
```

Important: when creating a typed literal, validation against the specified datatype occurs, throwing a **RDFModelException** in case of inconsistency detection. This way, typed literals created with RDFSharp are always semantically consistent.

WORKING WITH CONTAINERS AND COLLECTIONS

RDF specification provides 2 ways for modeling a group of homogeneous items into a single entity, demanding the choice to a purely semantic distinction:

<table>
<tr><td><u>Container</u></td><td>
It is modeled as instance of RDFContainer;

The given list of items may be incomplete: a container is semantically opened to the possibility of having further elements;

The ContainerType property indicates the nature of the container:

▪ Alt: unordered semantic, duplicates not allowed;
▪ Bag: unordered semantic, duplicates allowed;
▪ Seq: ordered semantic, duplicates allowed;

The ItemType property indicates the nature of the container items;

<pre>
// CREATE CONTAINER AND ADD ITEMS
RDFContainer beatles = new RDFContainer(RDFModelEnums.RDFContainerTypes.Bag,
                                        RDFModelEnums.RDFItemTypes.Resource);
beatles.AddItem(new RDFResource("http://beatles.com/ringo_starr"));
beatles.AddItem(new RDFResource("http://beatles.com/john_lennon"));
beatles.AddItem(new RDFResource("http://beatles.com/paul_mc_cartney"));
beatles.AddItem(new RDFResource("http://beatles.com/george_harrison"));
</pre>
</td></tr>
<tr><td><u>Collection</u></td><td>
It is modeled as instance of RDFCollection;

The given list of items may not be incomplete: a collection is semantically closed to the possibility of having further elements;

The ItemType property indicates the nature of the collection items;

When a collection is empty, it assumes the semantic of rdf:Nil

<pre>
// CREATE COLLECTION AND ADD ITEMS
RDFCollection beatles = new RDFCollection(RDFModelEnums.RDFItemTypes.Resource);
beatles.AddItem(new RDFResource("http://beatles.com/ringo_starr"));
beatles.AddItem(new RDFResource("http://beatles.com/john_lennon"));
beatles.AddItem(new RDFResource("http://beatles.com/paul_mc_cartney"));
beatles.AddItem(new RDFResource("http://beatles.com/george_harrison"));
</pre>
</td></tr>
</table>

There are several ways to manage the collection of items of a container or collection:

<u>INSERT</u>	***AddItem***	`beatles.AddItem(ringo_starr);`
<u>DELETE</u>	***RemoveItem***	`beatles.RemoveItem(john_lennon);`
	ClearItems	`beatles.ClearItems();`

Preferred way of modeling containers and collections is to directly add them to a graph:

```
// ADD CONTAINER/COLLECTION TO GRAPH
waltdisney.AddContainer(beatles_cont);
waltdisney.AddCollection(beatles_coll);
```

THE PROBLEM OF REIFICATION

Reification, in the RDF model, is the assertion of knowledge about existing knowledge and is mainly used for specifying metadata (e.g.: provenance, author) of a triple.

It is implemented in a way that makes use of a *ReificationSubject* blank resource acting as representative of the triple; then, 4 standard statements are created with that subject to explode the triple into its elementary components; finally, the asserted knowledge can be linked to the reified triple by using its representative as object.

"*Walt Disney says that* ***Donald Duck represents its funniest character***"		
Subject	**Predicate**	**Object**
bnode:ee56…	rdf:type	rdf:Statement
bnode:ee56…	rdf:subject	**wd:donald_duck**
bnode:ee56…	rdf:predicate	**ex:verb_represent**
bnode:ee56…	rdf:object	**ex:funniest_character**
wd:waltdisney	ex:verb_say	bnode:ee56…

With RDFSharp it is possible to reify triples, quadruples, containers and collections:

```
// REIFY TRIPLE AND MERGE IT INTO A GRAPH
RDFGraph reifGraph = goofygoof_is82yr.ReifyTriple();
waltdisney = waltdisney.UnionWith(reifGraph);

// ASSERT SOMETHING ABOUT REIFIED TRIPLE
waltdisney.AddTriple(new RDFTriple(
    new RDFResource("http://www.wikipedia.com/"),
    new RDFResource("http://example.org/verb_state"),
    goofygoof_is82yr.ReificationSubject
));

// REIFY CONTAINER
existingGraph.AddContainer(beatles_cont);
existingGraph.AddTriple(new RDFTriple(
    new RDFResource("http://www.thebeatles.com/"),
    RDFVocabulary.FOAF.GROUP,
    beatles_cont.ReificationSubject
));
// REIFY COLLECTION
existingGraph.AddCollection(beatles_coll);
existingGraph.AddTriple(new RDFTriple(
    new RDFResource("http://www.thebeatles.com/"),
    RDFVocabulary.FOAF.GROUP,
    beatles_coll.ReificationSubject
));
```

Triple reifications can be removed from a graph through the ***UnreifyTriples*** method, which deletes the 4 standard statements and compacts them into the original triple.

WORKING WITH SHACL SHAPES

CREATING SHAPES, TARGETS AND CONSTRAINTS

RDFSharp provides a **SHACL** engine which allows fluent creation and processing of shape graphs against RDF models, producing reports explaining violation results.

A shapes graph is modeled as instance of **RDFShapesGraph**:

```
// CREATE SHAPES GRAPH
var shapesGraph = new RDFShapesGraph();

// CREATE NAMED SHAPES GRAPH
var namedShapesGraph = new RDFShapesGraph(new RDFResource("ex:shapesGraph"));
```

A shapes graph can be easily transformed into an RDF graph through the ***ToRDFGraph*** method, but can also be obtained back through the static ***RDFShapesGraph.FromRDFGraph*** method:

```
// GET AN RDF GRAPH FROM A SHAPES GRAPH
var graph = shapesGraph.ToRDFGraph();

// GET A SHAPES GRAPH FROM AN RDF GRAPH
var shapesGraph = RDFShapesGraph.FromRDFGraph(graph);
```

A shape describes the *constraints* to be applied on a set of *target nodes* of the input data graph. Targets are modeled as concrete implementations of **RDFTarget** class.

RDFSharp supports **4** built-in SHACL targets:

RDFTargetClass	Targets the set of resources which are instances of the given class	`var dogs = new RDFTargetClass(new RDFResource("ex:dog"));`
RDFTargetNode	Targets the set of given resources	`var fido = new RDFTargetNode(new RDFResource("ex:fido"));`
RDFTargetSubjectsOf	Targets the set of resources which are subject of the given property	`var dogOwners = new RDFTargetSubjectsOf(new RDFResource("ex:ownsDog"));`
RDFTargetObjectsOf	Targets the set of resources which are object of the given property	`var dogs = new RDFTargetObjectsOf(new RDFResource("ex:ownsDog"));`

Shapes are modeled as concrete implementations of **RDFShape** class.

RDFSharp supports **2** built-in SHACL shapes:

RDFNodeShape	Works on the **focus nodes**, which are expressed by the targets	`var nodeShape = new RDFNodeShape(new RDFResource("ex:nodeShape"));`
RDFPropertyShape	Works on the **value nodes**, which are the values assumed by the focus nodes on a **path property**	`var propertyShape = new RDFPropertyShape(new RDFResource("ex:propertyShape"), new RDFResource("ex:property"));`

Targets can be added to a shape through the ***AddTarget*** method:

```
// ADD TARGETS TO SHAPE
nodeShape.AddTarget(dogs);
propertyShape.AddTarget(fido);
```

Shapes can be added to a shapes graph through the ***AddShape*** method:

```
// ADD SHAPES TO GRAPH
shapesGraph.AddShape(nodeShape);
shapesGraph.AddShape(propertyShape);
```

Constraints are modeled as concrete implementations of **RDFConstraint** class.

RDFSharp supports **28** built-in SHACL constraints:

RDFAndConstraint	Requires the value nodes to conform to all the given shapes	`var constraint = new RDFAndConstraint(); constraint.AddShape(new RDFResource("ex:shape1")); constraint.AddShape(new RDFResource("ex:shape2"));`
RDFClassConstraint	Requires the value nodes to be instances of the given class	`var constraint = new RDFClassConstraint(new RDFResource("ex:dog"));`

RDFClosedConstraint	Requires the value nodes to use only paths from property constraints; it also permits use of the set of ignored properties	`var constrain = new RDFClosedConstraint(true); constrain.AddIgnoredProperty (RDFVocabulary.FOAF.KNOWS);`
RDFDatatypeConstraint	Requires the value nodes to be literals of the given datatype	`var constraint = new RDFDatatypeConstraint( RDFModelEnums.RDFDatatypes.XSD_INTEGER);`
RDFDisjointConstraint	Requires the value nodes to not be found on the given property	`var constraint = new RDFDisjointConstraint(new RDFResource("ex:hasDog"));`
RDFEqualsConstraint	Requires the value nodes to be found on the given property	`var constraint = new RDFEqualsConstraint(new RDFResource("ex:hasName"));`
RDFHasValueConstraint	Requires the value nodes to contain the given RDF term	`var constraint = new RDFHasValueConstraint(new RDFResource("ex:Alice"));`
RDFInConstraint	Requires the value nodes to belong to the given RDF terms	`var constraint = new RDFInConstraint(RDFModelEnums.RDFItemTypes.Literal); inConstraint.AddValue(new RDFPlainLiteral("fido"));`
RDFLanguageInConstraint	Requires the value nodes to be plain literals with language tag belonging to the given language tags	`var constraint = new RDFLanguageInConstraint(new List<string>(){"en","it"})`

RDFLessThanConstraint	Requires the value nodes to assume values comparable (less) to the values of the given property	`var constraint = new RDFLessThanConstraint(new RDFResource("ex:name"));`
RDFLessThanOrEqualsConstraint	Requires the value nodes to assume values comparable (less-or-equal) to the values of the given property	`var constraint = new RDFLessThanOrEqualsConstraint (new RDFResource("ex:name"));`
RDFMaxCountConstraint	Requires the value nodes to be assumed at most the given number of times	`var constraint = new RDFMaxCountConstraint(8);`
RDFMaxExclusiveConstraint	Requires the value nodes to assume lower value than the given value	`var constraint = new RDFMaxExclusiveConstraint(new RDFTypedLiteral("5", RDFModelEnums.RDFDatatypes.XSD_INTEGER));`
RDFMaxInclusiveConstraint	Requires the value nodes to assume lower-or-equal value than the given value	`var constraint = new RDFMaxInclusiveConstraint(new RDFTypedLiteral("5", RDFModelEnums.RDFDatatypes.XSD_INTEGER));`
RDFMaxLengthConstraint	Requires the value nodes to have a string representation at most of the given length	`var constraint = new RDFMaxLengthConstraint(64);`

RDFMinCountConstraint	Requires the value nodes to be assumed at least the given number of times	`var constraint = new RDFMinCountConstraint(8);`
RDFMinExclusiveConstraint	Requires the value nodes to assume greater value than the given value	`var constraint = new RDFMinExclusiveConstraint(new RDFTypedLiteral("5", RDFModelEnums.RDFDatatypes.XSD_INTEGER));`
RDFMinInclusiveConstraint	Requires the value nodes to assume greater-or-equal value than the given value	`var constraint = new RDFMinInclusiveConstraint(new RDFTypedLiteral("5", RDFModelEnums.RDFDatatypes.XSD_INTEGER));`
RDFMinLengthConstraint	Requires the value nodes to have a string representation at least of the given length	`var constraint = new RDFMinLengthConstraint(32);`
RDFNodeConstraint	Requires the value nodes to conform to the given node shape	`var constraint = new RDFNodeConstraint(new RDFResource("x:nodeShape"));`
RDFNodeKindConstraint	Requires the value nodes to be of the given kind	`var constraint = new RDFNodeKindConstraint( RDFValidationEnums.RDFNodeKinds.IRI);`
RDFNotConstraint	Requires the value nodes to not conform to the given shape	`var constraint = new RDFNotConstraint(new RDFResource("ex:shape"));`

RDFOrConstraint	Requires the value nodes to conform to at least one of the given shapes	`var constraint = new RDFOrConstraint(); constraint.AddShape(new RDFResource("ex:shape1")); constraint.AddShape(new RDFResource("ex:shape2"));`
RDFPatternConstraint	Requires the value nodes to satisfy the given regex	`var constraint = new RDFPatternConstraint(new Regex("^\d+$"));`
RDFPropertyConstraint	Requires the value nodes to conform to the given property shape	`var constraint = new RDFPropertyConstraint(new RDFResource("x:propShape"));`
RDFQualifiedValueShape	Requires the value minimum/maximum number of working nodes to conform to the given shape	`var constraint = new RDFQualifiedValueShape(new RDFResource("ex:shape"), 1, 3);`
RDFUniqueLangConstraint	Requires the value nodes to be plain literals not assuming duplicate languages	`var constraint = new RDFUniqueLangConstraint(true);`
RDFXoneConstraint	Requires the value nodes to conform to exactly one of the given shapes	`var constraint = new RDFXoneConstraint(); constraint.AddShape(new RDFResource("ex:shape1")); constraint.AddShape(new RDFResource("ex:shape2"));`

Constraints can be added to a shape through the ***AddConstraint*** method:

```
// ADD CONSTRAINTS TO SHAPE
nodeShape.AddConstraint(new RDFMaxLengthConstraint(24));
propertyShape. AddConstraint(new RDFNodeKindConstraint(RDFValidationEnums.RDFNodeKinds.Literal));
```

VALIDATING GRAPHS AND REPORTING RESULTS

A shapes graph can be applied on a data graph through the ***Validate*** extension method, which compiles an **RDFValidationReport** containing the results collected from the evaluation of the constraints declared in the shapes:

```
// EVALUATE SHAPES GRAPH
RDFValidationReport vReport = shapesGraph.Validate(dataGraph);
```

The ***Conforms*** property indicates that the data graph is compliant with the shapes graph, meaning that no results have been produced.

The ***ResultsCount*** property indicates the total number of results which have been produced by the validation.

A validation report can be easily transformed into an RDF graph through the ***ToRDFGraph*** method:

```
// GET AN RDF GRAPH FROM A VALIDATION REPORT
RDFGraph validationReportGraph = vReport.ToRDFGraph();
```

Validation results are modeled as instances of **RDFValidationResult**:

```
// INSPECT VALIDATION RESULTS
foreach (RDFValidationResult vResult in vReport) {
  Console.Writeline("Severity: " + vResult.Severity);
  Console.Writeline("SourceShape: " + vResult.SourceShape);
  Console.Writeline("SourceConstraintComponent: " + vResult.SourceConstraintComponent);
  Console.Writeline("FocusNode: " + vResult.FocusNode);
  Console.Writeline("ResultPath: " + vResult.ResultPath);
  Console.Writeline("ResultValue: " + vResult.ResultValue);
}
```

A validation result indicates that an evidence with the specified level of *Severity* was generated for the shape named *SourceShape* declared in the shapes graph, with origin in the constraint named *SourceConstraintComponent*. Within this constraint, the target node named *FocusNode* has violated the business logics by assuming the *ResultValue* value in correspondence of the *ResultPath* property.

WORKING WITH RDF STORES

TRACKING PROVENANCE OF RDF DATA WITH CONTEXTS AND QUADRUPLES

When working with RDF data, the information is usually made up of graphs dislocated in the worldwide "Linked Data" network. Such a huge amount of available data requires a structured way to track provenance of statements, because the same triple may probably exist into multiple graphs.

In the depicted scenario, the canonical solution is to extend the traditional *Subject-Predicate-Object* triple structure with an additional non-blank URI-based dimension called *Context*, modeled as instance of **RDFContext**:

```
// CREATE CONTEXT FROM STRING
var wdisney_ctx = new RDFContext("http://www.waltdisney.com/");

// CREATE CONTEXT FROM URI
var wdisney_ctx_uri = new RDFContext(new Uri("http://www.waltdisney.com/"));

// CREATE DEFAULT CONTEXT (DEFAULT NAMESPACE)
var wdisney_ctx_default = new RDFContext();
```

This way a triple enriched with context becomes a quadruple, assuming *Context-Subject-Predicate-Object* structure modeled as instance of **RDFQuadruple**:

```
// CREATE QUADRUPLES
// "From Wikipedia.com: Mickey Mouse is 85 years old"
RDFQuadruple wk_mickeymouse_is85yr
      = new RDFQuadruple(
            new RDFContext("http://www.wikipedia.com/"),
            new RDFResource("http://www.waltdisney.com/mickey_mouse"),
            RDFVocabulary.FOAF.AGE,
            new RDFTypedLiteral("85", RDFModelEnums.RDFDatatypes.XSD_INTEGER)
);

// "From WaltDisney.com: Mickey Mouse is 85 years old"
RDFQuadruple wd_mickeymouse_is85yr
      = new RDFQuadruple(
            new RDFContext("http://www. waltdisney.com/"),
            new RDFResource("http://www.waltdisney.com/mickey_mouse"),
            RDFVocabulary.FOAF.AGE,
            new RDFTypedLiteral("85", RDFModelEnums.RDFDatatypes.XSD_INTEGER)
);

// "From Wikipedia.com: Donald Duck has english name "Donald Duck""
RDFQuadruple wk_donald_duck_name_enus
      = new RDFQuadruple(
            new RDFContext("http://www.wikipedia.com/"),
            new RDFResource("http://www.waltdisney.com/donald_duck"),
            RDFVocabulary.FOAF.NAME,
            new RDFPlainLiteral("Donald Duck", "en")
);
```

Important: when creating a quadruple, validation against presence of a blank resource in the predicate position occurs, throwing a **RDFStoreException** in case of detection.

CREATING AND MANIPULATING STORES

Thinking of a store as a natural extension of a graph obtained by introducing the context-awareness of triples, we can easily define a store as a collection of quadruples, modeled as concrete implementation of the abstract foundational **RDFStore** class.

There are several ways to manage the collection of quadruples of a store:

INSERT	***AddQuadruple*** Adds the given quadruple.	`wdStore.AddQuadruple(` `wk_mickeymouse_is85yr);`
	MergeGraph Adds the triples of the given graph, which are transformed into quadruples having the context of the graph.	`wdStore.MergeGraph(waltdisney);`
DELETE	***RemoveQuadruple*** Removes the given quadruple.	`wdStore.RemoveQuadruple(` `wk_mickeymouse_is85yr);`
	RemoveQuadruplesByContext Removes the quadruples having the given context as "Context".	`wdStore.RemoveQuadruplesByContext(` `wdisney_ctx);`
	RemoveQuadruplesBySubject Removes the quadruples having the given resource as "Subject".	`wdStore.RemoveQuadruplesBySubject(` `donaldduck);`
	RemoveQuadruplesByPredicate Removes the quadruples having the given resource as "Predicate".	`wdStore.RemoveQuadruplesByPredicate` `(RDFVocabulary.FOAF.NAME);`
	RemoveQuadruplesByObject Removes the quadruples having the given resource as "Object".	`wdStore.RemoveQuadruplesByObject(` `goofygoof);`
	RemoveQuadruplesByLiteral Removes the quadruples having the given literal as "Object".	`wdStore.RemoveQuadruplesByLiteral(` `mickeymouse_age);`

	ClearQuadruples Empties the store.	`wdStore.ClearQuadruples();`
SEARCH	***ContainsQuadruple*** Checks if the given quadruple exists in the store, returning True/False.	`Boolean checkQuadruple = wdStore.ContainsQuadruple( wk_mickeymouse_is85yr);`
	SelectAllQuadruples Returns a store containing all the quadruples.	`RDFStore all_quadruples = wdStore.SelectAllQuadruples();`
	SelectQuadruplesByContext Returns a store containing the quadruples having the given context as "Context".	`RDFStore quadruples_by_context = wdStore.SelectQuadruplesByContext( wdisney_ctx);`
	SelectQuadruplesBySubject Returns a store containing the quadruples having the given resource as "Subject".	`RDFStore quadruples_by_subject = wdStore.SelectQuadruplesBySubject( donaldduck);`
	SelectQuadruplesByPredicate Returns a store containing the quadruples having the given resource as "Predicate".	`RDFStore quadruples_by_predicate = wdStore.SelectQuadruplesByPredicate (RDFVocabulary.FOAF.NAME);`
	SelectQuadruplesByObject Returns a store containing the quadruples having the given resource as "Object".	`RDFStore quadruples_by_object = wdStore.SelectQuadruplesByObject( goofygoof);`
	SelectQuadruplesByLiteral Returns a store containing the quadruples having the given literal as "Object".	`RDFStore quadruples_by_literal = wdStore.SelectQuadruplesByLiteral( mickeymouse_age);`

RDFSharp provides an in-memory RDF store engine implementation which behaves likes a graph, modeled as instance of **RDFMemoryStore**:

```
// CREATE EMPTY MEMORY STORE
var wdStore = new RDFMemoryStore();

// CREATE MEMORY STORE FROM A LIST OF QUADRUPLES
var quadruples = new List<RDFQuadruple>({wk_mickeymouse_is85yr, wk_mickeymouse_is85yr);
var wdStoreFilled = new RDFMemoryStore(quadruples);
```

A memory store can be easily transformed into an ADO.NET datatable with *Context-Subject-Predicate-Object* columns through the ***ToDataTable*** method, but can also be obtained back through the static ***RDFMemoryStore.FromDataTable*** method:

```
// GET A DATATABLE FROM A MEMORY STORE (any kind of store can be exported to datatable)
var wdStore_table = wdStore.ToDataTable();

// GET A MEMORY STORE FROM A DATATABLE
var wdStore_new = RDFMemoryStore.FromDataTable(wdStore_table);
```

RDFMemoryStore is compatible with the *IEnumerable<RDFQuadruple>*:

```
// ITERATE QUADRUPLES OF A MEMORY STORE WITH FOREACH
foreach (var q in wdStore) {
    Console.Writeline("Quadruple:   " + q);
    Console.Writeline("  Context:   " + q.Context);
    Console.Writeline("  Subject:   " + q.Subject);
    Console.Writeline("  Predicate: " + q.Predicate);
    Console.Writeline("  Object:    " + q.Object);
}

// ITERATE QUADRUPLES OF A MEMORY STORE WITH ENUMERATOR
var quadruplesEnum = wdStore.QuadruplesEnumerator;
while (quadruplesEnum.MoveNext()) {
    Console.Writeline("Quadruple:   " + quadruplesEnum.Current);
    Console.Writeline("  Context:   " + quadruplesEnum.Current.Context);
    Console.Writeline("  Subject:   " + quadruplesEnum.Current.Subject);
    Console.Writeline("  Predicate: " + quadruplesEnum.Current.Predicate);
    Console.Writeline("  Object:    " + quadruplesEnum.Current.Object);
}
```

The content of a store can be read from, or saved to, a RDF **file** or **stream**. RDFSharp supports **2** standard RDF store data serialization formats:

Format	Read	Write	

N-Quads	YES	YES	`var nquadsFormat = RDFStoreEnums.RDFFormats.NQuads;`
			
			`// READ N-QUADS FILE`
			`var myStore = RDFMemoryStore.FromFile(nquadsFormat,"C:\\file.nq");`
			`// READ N-QUADS STREAM`
			`var myStore = RDFMemoryStore.FromStream(nquadsFormat, inStream);`
			
			`// WRITE N-QUADS FILE`
			`myStore.ToFile(nquadsFormat,"C:\\newfile.nq");`
			`// WRITE N-QUADS STREAM`
			`myStore.ToStream(nquadsFormat, outStream);`

TriX	YES	YES	`var trixFormat = RDFStoreEnums.RDFFormats.TriX;`
			
			`// READ TRIX FILE`
			`var memStore = RDFMemoryStore.FromFile(trixFormat,"C:\\file.trix");`
			`// READ TRIX STREAM`
			`var memStore = RDFMemoryStore.FromStream(trixFormat, inStream);`
			
			`// WRITE TRIX FILE`
			`myStore.ToFile(trixFormat,"C:\\newfile.trix");`
			`// WRITE TRIX STREAM`
			`myStore.ToStream(trixFormat, outStream);`

BACKING STORES ON SQL ENGINES

While there are no restrictions to the number of quadruples which can be contained in a memory store, apart the physical memory limits of the machine, it is reasonable to think about a scalable approach for persisting real-world amounts of RDF data.

RDFSharp provides extensions for **6** RDF store engine implementations:

- **SQL Server** (Microsoft.Data.SqlClient)
- **Firebird** (FirebirdSql.Data.FirebirdClient)
- **MySQL** (MySql.Data)
- **PostgreSQL** (Npgsql)
- **SQLite** (Microsoft.Data.Sqlite)
- **Oracle** (Oracle.ManagedDataAccess)

The SQL schema has been designed for unobtrusive integration into existing databases; a *Quadruples* table is automatically created and maintained with the following structure:

QuadrupleID	PRIMARY KEY, INT64, NOT NULL
TripleFlavor	INT32, NOT NULL
Context	VARCHAR(1000), NOT NULL
ContextID	INT64, NOT NULL
Subject	VARCHAR(1000), NOT NULL
SubjectID	INT64, NOT NULL
Predicate	VARCHAR(1000), NOT NULL
PredicateID	INT64, NOT NULL
Object	VARCHAR(1000), NOT NULL
ObjectID	INT64, NOT NULL

WORKING WITH SPARQL QUERIES

MIRELLA ENGINE (BASICS): VARIABLES, PATTERNS, PATTERN GROUPS

RDFSharp provides a **SPARQL** engine, codenamed **Mirella**, which allows fluent creation and execution of queries over graphs, stores, federations and endpoints. It has been designed on top of the following basic concepts: *variable*, *pattern* and *pattern group*.

A variable is a placeholder for query results, modeled as instance of **RDFVariable**:

```
// CREATE VARIABLE
var x = new RDFVariable("x");  // ?X
var y = new RDFVariable("y");  // ?Y
var n = new RDFVariable("n");  // ?N
var c = new RDFVariable("c");  // ?C
```

A pattern is a triple which can include variables, modeled as instance of **RDFPattern**:

```
// CREATE PATTERNS
var dogOf = new RDFResource(RDFVocabulary.DC.BASE_URI + "dogOf");

var y_dogOf_x = new RDFPattern(y, dogOf, x);       // TRIPLE PATTERN
var c_y_dogOf_x = new RDFPattern(c, y, dogOf, x);  // QUADRUPLE PATTERN
```

It can also represent a quadruple, if the context information is provided. Elements which can be part of a pattern are resources, literals, contexts and variables: these are concrete implementations of the foundational abstract **RDFPatternMember** class.

A pattern group is a collection of patterns, modeled as instance of **RDFPatternGroup**:

```
// CREATE EMPTY PATTERN GROUP
var pg1 = new RDFPatternGroup("PG1");

// CREATE PATTERN GROUP FROM A LIST OF PATTERNS
var patterns = new List<RDFPattern>(){ y_dogOf_x };
var pg2 = new RDFPatternGroup("PG2", patterns);
```

Patterns can be added to a pattern group through the ***AddPattern*** method:

```
// ADD PATTERNS TO PATTERN GROUP
pg1.AddPattern(y_dogOf_x);
pg1.AddPattern(c_y_dogOf_x);
```

Pattern groups can be added to a query through the ***AddPatternGroup*** method:

```
// ADD PATTERN GROUPS TO QUERY
query.AddPatternGroup(pg1);
query.AddPatternGroup(pg2);
```

When a pattern is evaluated, the query engine produces a table having the detected variables as columns: the pattern acts as a scanner which passes over the triples (or quadruples) by checking if they satisfy the given condition, then every time that matching data is found it creates a new row and fills in-place variables. The algorithm is iterated until all the patterns contained in the pattern group have been processed.

The intermediate pattern tables are then merged into a unique pattern group table: default strategy is INNER-JOIN (PRODUCT-JOIN in case of no common columns) but special SPARQL operators may be locally applied to induce a different approach:

Optional	LEFT-JOIN is used with the previous pattern, preserving unbound values.	`y_dogOf_x.Optional();`
UnionWithNext	UNION is used with the next pattern, preserving unbound values.	`y_dogOf_x.UnionWithNext();`

When the pattern group table has been produced, the filters contained in the pattern group are subsequently applied, restricting available solutions.

The algorithm is then replicated on remaining pattern groups, until all corresponding tables have been produced: at this stage of query evaluation, the same considerations done for pattern joins are also suitable for pattern group joins (merge strategy and special local operators) until a unique query results table is produced.

At last, the modifiers contained in the query are applied on the results table, inducing eventual data sorts or cardinality operations. There may be further specific elaborations in order to produce the final query result, depending on the type of modeled query.

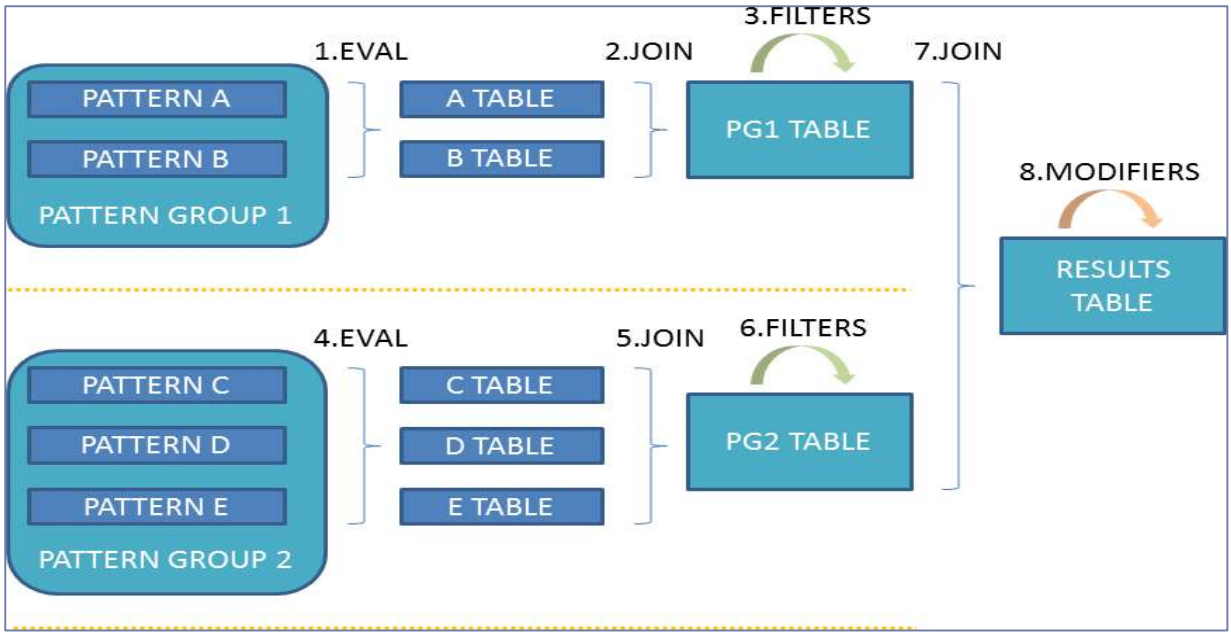


Fig. 1: Algorythm for evaluation of SPARQL queries

MIRELLA ENGINE (BASICS): FILTERS, MODIFIERS

Filters are mechanisms which can be used to keep only rows which satisfy a given condition, modeled as concrete implementations of the abstract **RDFFilter** class.

RDFSharp provides modeling support for **15** built-in SPARQL filters:

Modeling Class	**Description**	
RDFBooleanAndFilter	Keeps rows which satisfy both the given filters.	`var andFilter = new RDFBooleanAndFilter(leftFilter, rightFilter);`
RDFBooleanNotFilter	Keeps rows which do not satisfy the given filter.	`var notFilter = new RDFBooleanNotFilter(filter);`
RDFBooleanOrFilter	Keeps rows which satisfy at least one of the given filters.	`var orFilter = new RDFBooleanOrFilter(leftFilter, rightFilter);`
RDFBoundFilter	Keeps rows which have a not null value in the given variable binding.	`var boundFilter = new RDFBoundFilter(variable);`
RDFComparisonFilter	Keeps rows which satisfy the given type of comparison between the given RDF terms.	`var compFilter = new RDFComparisonFilter(comparisonFlavor, leftTerm, rightTerm);`
RDFDatatypeFilter	Keeps rows which have a typed literal of the given datatype in the given variable binding.	`var dtypeFilter = new RDFDatatypeFilter(variable, datatype);`
RDFExistsFilter **(not allowed inside Boolean filters)**	Keeps rows which have bindings satisfying the given pattern.	`var existsFilter = new RDFExistsFilter(pattern);`

RDFInFilter	Keeps rows which satisfy the search for the given RDF term.	`var inFilter = new RDFInFilter(patternMember, listOfPatternMembers});`
RDFIsBlankFilter	Keeps rows which have a blank resource in the given variable binding.	`var blankFilter = new RDFIsBlankFilter(variable);`
RDFIsLiteralFilter	Keeps rows which have a literal in the given variable binding.	`var literalFilter = new RDFIsLiteralFilter(variable);`
RDFIsUriFilter	Keeps rows which have a resource in the given variable binding.	`var uriFilter = new RDFIsUriFilter(variable);`
RDFLangMatchesFilter	Keeps rows which have a plain literal with the given language tag in the given variable binding.	`var langmatchesFilter = new RDFLangMatchesFilter(variable, languageTag);`
RDFNotExistsFilter **(not allowed inside Boolean filters)**	Keeps rows which have bindings not satisfying the given pattern.	`var notExistsFilter = new RDFNotExistsFilter(pattern);`
RDFRegexFilter	Keeps rows which have a value satisfying the given regex in the given variable binding.	`var regexFilter = new RDFRegexFilter(variable, regEx);`
RDFSameTermFilter	Keeps rows which have a value equal to given RDF term in the given variable binding.	`var sametermFilter = new RDFSameTermFilter(variable, term);`

Filters can be added (and scoped) to a pattern group through the ***AddFilter*** method:

```
// ADD FILTERS TO PATTERN GROUP
pg1.AddFilter(new RDFSameTermFilter(new RDFVariable("character"), donaldduck));
pg1.AddFilter(new RDFLangMatchesFilter(n, "it-IT"));
```

Modifiers are mechanisms which can be used to alter cardinality and order of results, modeled as concrete implementations of the abstract **RDFModifier** class.

RDFSharp provides modeling support for **5** built-in SPARQL modifiers:

Modeling Class	**Description**	
RDFDistinctModifier	Eliminates duplicate rows. Only one distinct modifier is accepted in a query.	`var distinctModifier = new RDFDistinctModifier();`
RDFOrderByModifier	Sorts rows on the given column. Multiple orderby modifiers can be accepted in a query.	`var orderbyModifier = new RDFOrderByModifier(variable, orderType);`
RDFGroupByModifier	Partitions rows on the given variables and then aggregates results on the given functions. Only one groupby modifier is accepted in a query.	`var groupbyModifier = new RDFGroupByModifier(new List<RDFVariable>() { department });`
RDFLimitModifier	Keeps the given number of rows. If 0 is specified, no rows are kept. Only one limit modifier is accepted in a query.	`var limitModifier = new RDFLimitModifier(limit);`
RDFOffsetModifier	Keeps rows starting from the given position. If it is greater than number of available rows, no rows are kept. Only one offset modifier is accepted in a query.	`var offsetModifier = new RDFOffsetModifier(offset);`

Modifiers can be added to a query through the ***AddModifier*** method:

```
// ADD MODIFIERS TO QUERY
query.AddModifier(new RDFOrderByModifier(n, RDFQueryEnums.RDFOrderByFlavors.ASC));
query.AddModifier(new RDFDistinctModifier());
query.AddModifier(new RDFGroupByModifier(new List<RDFVariable>(){x});
query.AddModifier(new RDFLimitModifier(100));
query.AddModifier(new RDFOffsetModifier(25));
```

MIRELLA ENGINE (ADVANCED): PROPERTY PATHS

SPARQL property path is a syntactic shortcut used for declaring a chain of properties connecting two RDF terms in a graph. It is modeled as instance of **RDFPropertyPath**:

```
// INITIALIZE PROPERTY PATH (VARIABLE TERMS)
var variablePropPath = new RDFPropertyPath(new RDFVariable("START"), new RDFVariable("END"));

// INITIALIZE PROPERTY PATH (MIXED TERMS)
var mixedPropPath = new RDFPropertyPath(new RDFResource("http://res.org/"), new
RDFVariable("END"));
```

Property paths can be added to a pattern group through the ***AddPropertyPath*** method:

```
// ADD PROPERTY PATH TO PATTERN GROUP
pg1.AddPropertyPath(variablePropPath);
```

Once initialized, a property path needs at least one intermediate step to be added, in order to be eligible for evaluability: this is required by the query engine, which translates the property path into an equivalent chain of patterns; if a ground semantic is detected (no variables found in the pattern chain) the property path will be not evaluated at all!

Steps are modeled as instances of **RDFPropertyPathStep** and have **2** semantic flavors:

Sequence	Given step has **sequential** evaluation (**AND**)	`//ADD SEQUENCE STEPS TO PROPERTY PATH`
		`variablePropPath.AddSequenceStep(new RDFPropertyPathStep(new RDFResource("rdf:P1")));`
		
		`variablePropPath.AddSequenceStep(new RDFPropertyPathStep(new RDFResource("rdf:P2")));`

Alternative	Given list of steps has **parallel** evaluation (**OR**)	`//ADD ALTERNATIVE STEPS TO PROPERTY PATH`
		`var altSteps = new List<RDFPropertyPathStep>();`
		
		`altSteps.Add(new RDFPropertyPathStep(new RDFResource("rdf:P3")));`
		
		`altSteps.Add(new RDFPropertyPathStep(new RDFResource("rdf:P7")));`
		
		`variablePropPath.AddAlternativeSteps(altSteps);`

It is also possible to specify that a particular step should be evaluated with **inverse** logic, inducing a temporary local swap of its evaluation direction:

```
// ADD INVERSE SEQUENCE STEP TO PROPERTY PATH:  ?START ^rdf:INVP ?END
variablePropPath.AddSequenceStep(new RDFPropertyPathStep(new RDFResource("rdf:INVP")).Inverse());

//ADD ALTERNATIVE STEPS (ONE INVERSE) TO PROPERTY PATH: ?START (rdf:P3|^rdf:INVP3) ?END
var altSteps = new List<RDFPropertyPathStep>();
altSteps.Add(new RDFPropertyPathStep(new RDFResource("rdf:P3")));
altSteps.Add(new RDFPropertyPathStep(new RDFResource("rdf:INVP3")).Inverse());
variablePropPath.AddAlternativeSteps(altSteps);
```

MIRELLA ENGINE (ADVANCED): SUBQUERIES, AGGREGATORS, VALUES

Within a SPARQL query it is possible to embed full-featured subqueries in a way that makes it possible to achieve better readability and more powerful capabilities.

Subqueries can be added to queries through the ***AddSubQuery*** method:

```
// ADD SUBQUERY TO QUERY
myQuery.AddSubQuery(mySubQuery);
```

Only SPARQL SELECT queries can be added as subqueries: in fact, their variable bindings are projected out to the mother query once available, exactly like pattern groups.

SPARQL subqueries are first-class citizens of RDFSharp, with no limits to the potential nesting level, and support all the available semantics for SPARQL query join operations: in fact, it is possible to specify subqueries as ***Optional*** or ***UnionWithNext***.

Aggregators are functions which can be applied over a group-by partitioned set of SPARQL results, modeled as implementations of foundational **RDFAggregator** class.

RDFSharp provides modeling support for **8** built-in SPARQL aggregators:

Modeling Class	**Description**
RDFPartitionAggregator	Core of the partitioning algorithm, it is automatically associated to the GroupBy modifier in order to compute partitioning keys for other aggregators.
RDFAvgAggregator	Computes the average of its aggregator variable. Effective only on **numeric typed literals**, otherwise returns unbound result.
RDFSumAggregator	Computes the sum of its aggregator variable. Effective only on **numeric typed literals**, otherwise returns unbound result.
RDFCountAggregator	Computes the count of its aggregator variable, discarding unbound occurrences.
RDFGroupConcatAggregator	Computes the string concatenation of its aggregator variable, using the given separator.

RDFSampleAggregator	Computes the sampling of its aggregator variable, returning one of its occurrences.
RDFMinAggregator	Computes the minimum of its aggregator variable. Effective only on **numeric typed literals** when specified flavor is **numeric**, otherwise always effective when specified flavor is **string**.
RDFMaxAggregator	Computes the maximum of its aggregator variable. Effective only on **numeric typed literals** when specified flavor is **numeric**, otherwise always effective when specified flavor is **string**.

Aggregators can be added to group-by modifiers through the ***AddAggregator*** method:

```
// ADD AGGREGATORS TO GROUPBY MODIFIER
gm.AddAggregator(new RDFAvgAggregator(new RDFVariable("age"),new RDFVariable("avg_age")));
gm.AddAggregator(new RDFCountAggregator(new RDFVariable("dept"),new RDFVariable("count_dept")));
gm.AddAggregator(new RDFGroupConcatAggregator(new RDFVariable("name"),new RDFVariable("gc_name"),"-"));
gm.AddAggregator(new RDFSampleAggregator(new RDFVariable("name"),new RDFVariable("sample_name")));
gm.AddAggregator(new RDFSumAggregator(new RDFVariable("salary"),new RDFVariable("sum_salary")));

gm.AddAggregator(new RDFMinAggregator(new RDFVariable("age"),new RDFVariable("min_age"),
RDFQueryEnums.RDFMinMaxAggregatorFlavors.Numeric)); //?age is expected to have numeric typedliterals
gm.AddAggregator(new RDFMinAggregator(new RDFVariable("city"),new RDFVariable("min_city"),
RDFQueryEnums.RDFMinMaxAggregatorFlavors.String));

gm.AddAggregator(new RDFMaxAggregator(new RDFVariable("salary"),new RDFVariable("max_salary"),
RDFQueryEnums.RDFMinMaxAggregatorFlavors.Numeric)); //?salary is expected to have numeric typedliterals
gm.AddAggregator(new RDFMaxAggregator(new RDFVariable("city"),new RDFVariable("min_city"),
RDFQueryEnums.RDFMinMaxAggregatorFlavors.String));
```

It is possible to filter a group-by partitioned set of SPARQL results by applying the ***SetHavingClause*** operator on desired aggregators:

```
// ADD AGGREGATORS TO GROUPBY MODIFIER
RDFModelEnums.RDFDatatypes xsdDbl = RDFModelEnums.RDFDatatypes.XSD_DOUBLE;
RDFModelEnums.RDFDatatypes xsdInt = RDFModelEnums.RDFDatatypes.XSD_INT;
gm.AddAggregator(new RDFAvgAggregator(new RDFVariable("age"),new RDFVariable("avg_age"))
  .SetHavingClause(RDFQueryEnums.RDFComparisonFlavors.GreaterThan,new RDFTypedLiteral("25.5",xsdDbl))
);
gm.AddAggregator(new RDFCountAggregator(new RDFVariable("dept"),new RDFVariable("count_dept"))
  .SetHavingClause(RDFQueryEnums.RDFComparisonFlavors.EqualTo,new RDFTypedLiteral("4",xsdInt)
);
```

Certain modeling situations require complex filter chains in order to get the desired query results. To address these scenarios, SPARQL offers the VALUES syntactic sugar, which allows to inject a tabular binding of variables inside a query.

It is modeled as instance of **RDFValues** and is built by column with ***AddColumn*** method:

```
//Declare the following SPARQL values:
/*
    VALUES (?a ?b ?c) {
        ("1" "2" "3")
        ("2" "4" "6")
        ("3" "6" UNDEF)
    }
*/
RDFValues myValues = new RDFValues()
                              .AddColumn(new RDFVariable("a"),
                                         new List<RDFPatternMember>()
                                         {
                                             new RDFPlainLiteral("1"),
                                             new RDFPlainLiteral("2"),
                                             new RDFPlainLiteral("3")
                                         })
                              .AddColumn(new RDFVariable("b"),
                                         new List<RDFPatternMember>()
                                         {
                                             new RDFPlainLiteral("2"),
                                             new RDFPlainLiteral("4"),
                                             new RDFPlainLiteral("6")
                                         })
                              .AddColumn(new RDFVariable("c"),
                                         new List<RDFPatternMember>()
                                         {
                                             new RDFPlainLiteral("3"),
                                             new RDFPlainLiteral("6"),
                                             null   //UNDEF
                                         });
```

Values can be added to a pattern group through the ***AddValues*** method:

```
// ADD PROPERTY PATH TO PATTERN GROUP
pg1.AddValues(myValues);
```

WORKING WITH FEDERATIONS: SEAMLESSLY INTEGRATED RDF DATA

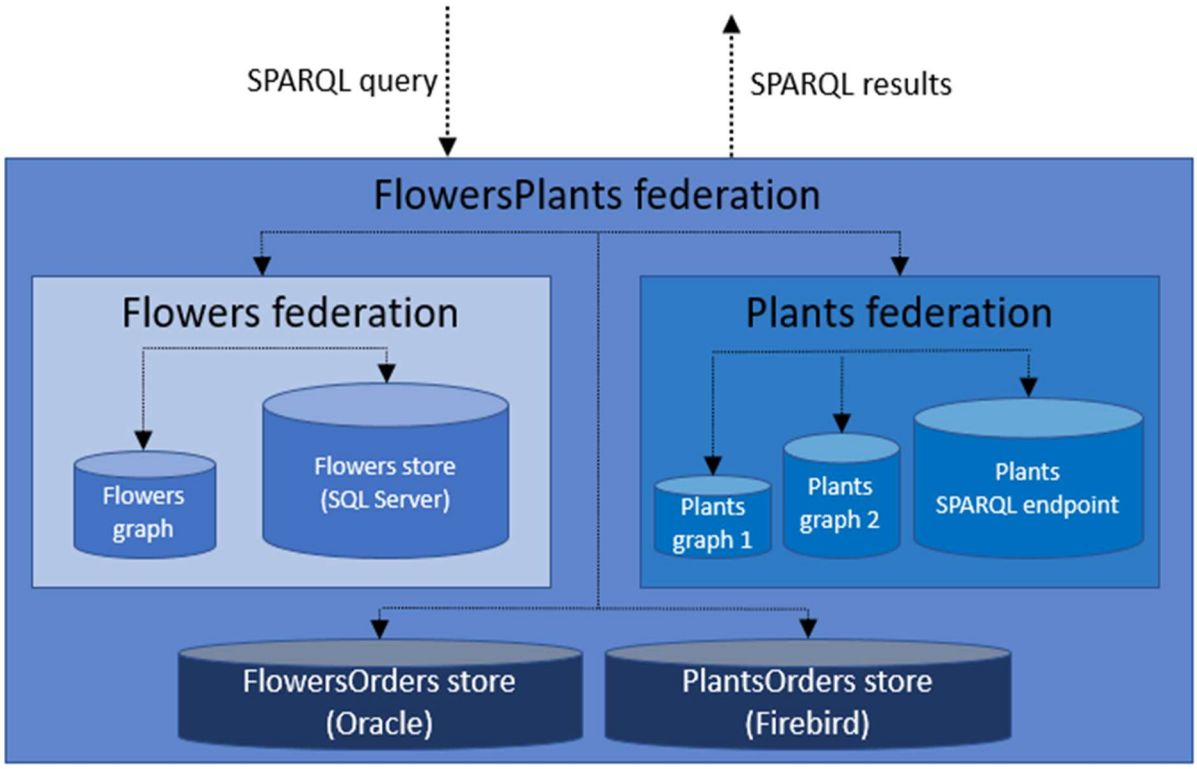
RDFSharp supports the creation of seamlessly integrated RDF data sources through the concept of *federation*, which is modeled as instance of **RDFFederation**:

```
// COMPOSE FEDERATIONS WITH RDF DATA SOURCES
RDFFederation federation1 =
    new RDFFederation()
        .AddGraph(graph1)
        .AddGraph(graph2)
        .AddStore(memStore1)
        .AddSPARQLEndpoint(dbPediaEndpont);

RDFFederation federation2 =
    new RDFFederation()
        .AddGraph(graph3)
        .AddStore(sqlServerStore1)
        .AddSPARQLEndpoint(playgroundEndpnt);

RDFFederation federation3 =
    new RDFFederation()
        .AddFederation(new RDFFederation().AddFederation(federation1))
        .AddStore(memStore2)
        .AddFederation(new RDFFederation().AddFederation(federation2));
```

By composing federations, it is possible to give RDF data sources a rational semantic organization and to realize a **truly integrated** SPARQL query experience:



CREATING AND EXECUTING SELECT QUERIES

SPARQL "Select" query type is modeled as instance of **RDFSelectQuery**:

```
// CREATE SELECT QUERY
RDFSelectQuery selectQuery = new RDFSelectQuery();
```

This is the only SPARQL query type which supports the projection operator on results, which is applied through the ***AddProjectionVariable*** method.

It is possible to apply a "Select" query on graphs, stores or federations, producing results in form of tabular data modeled as instance of **RDFSelectQueryResult**:

```
// APPLY SELECT QUERY TO GRAPH
RDFSelectQueryResult selectQueryResult = selectQuery.ApplyToGraph(graph);

// APPLY SELECT QUERY TO STORE
RDFSelectQueryResult selectQueryResult = selectQuery.ApplyToStore(store);

// APPLY SELECT QUERY TO FEDERATION
RDFSelectQueryResult selectQueryResult = selectQuery.ApplyToFederation(federation);
```

The "*SelectResults*" property is an ADO.NET datatable storing result variable bindings.

RDFSharp supports the W3C standard "**SPARQL Query Results XML Format**", thus it is possible to exchange SPARQL Select/Ask query results with external systems:

```
 // EXPORT SELECT QUERY RESULTS TO SPARQL XML FORMAT (FILE)
selectQueryResult.ToSparqlXmlResult("C:\select_results.srq");

// EXPORT SELECT QUERY RESULTS TO SPARQL XML FORMAT (STREAM)
selectQueryResult.ToSparqlXmlResult(myStream);

// IMPORT SELECT QUERY RESULTS FROM SPARQL XML FORMAT (FILE)
selectQueryResult = RDFSelectQueryResult.FromSparqlXmlResult("C:\select_results.srq");

// IMPORT SELECT QUERY RESULTS FROM SPARQL XML FORMAT (STREAM)
selectQueryResult = RDFSelectQueryResult.FromSparqlXmlResult(myStream);
```

Let's model a simple SPARQL "Select" query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?Y ?X ?N
WHERE  {

  #PG1
  {
    ?Y dc:dogOf ?X .
    OPTIONAL { ?X foaf:name ?N } .
    ?X foaf:knows ?H .
    FILTER ( REGEX(STR(?N), "Mouse", "i") )
  }

}
ORDER BY DESC(?Y)
LIMIT 5
```

```
// Create variables
RDFVariable x = new RDFVariable("x");
RDFVariable y = new RDFVariable("y");
RDFVariable n = new RDFVariable("n");
RDFVariable h = new RDFVariable("h");
// Compose query
RDFSelectQuery query = new RDFSelectQuery()
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("dc"))
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("foaf"))
 .AddPatternGroup(new RDFPatternGroup("PG1")
   .AddPattern(new RDFPattern(y, dogOf, x))
   .AddPattern(new RDFPattern(x, name,  n).Optional())
   .AddPattern(new RDFPattern(x, knows, h))
   .AddFilter(new RDFRegexFilter(n, new Regex(@"Mouse",
RegexOptions.IgnoreCase))))
 .AddModifier(new RDFOrderByModifier(y,
RDFQueryEnums.RDFOrderByFlavors.DESC))
 .AddModifier(new RDFLimitModifier(5))
 .AddProjectionVariable(y)
 .AddProjectionVariable(x)
 .AddProjectionVariable(n);
// Apply query
RDFSelectQueryResult selectResult =
query.ApplyToGraph(waltdisney);
```

CREATING AND EXECUTING ASK QUERIES

SPARQL "Ask" query type is modeled as instance of **RDFAskQuery**:

```
// CREATE ASK QUERY
RDFAskQuery askQuery = new RDFAskQuery();
```

It is possible to apply an "Ask" query on graphs, stores or federations, producing results in form of Boolean data modeled as instance of **RDFAskQueryResult**:

```
// APPLY ASK QUERY TO GRAPH
RDFAskQueryResult askQueryResult = askQuery.ApplyToGraph(graph);

// APPLY ASK QUERY TO STORE
RDFAskQueryResult askQueryResult = askQuery.ApplyToStore(store);

// APPLY ASK QUERY TO FEDERATION
RDFAskQueryResult askQueryResult = askQuery.ApplyToFederation(federation);
```

The "*AskResult*" property is a Boolean storing response.

Let's model a simple SPARQL "Ask" query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

ASK WHERE  {

   #PG1
   {
      ?Y dc:dogOf ?X .
      { ?X foaf:name ?N }
      UNION
      { ?X foaf:knows ?H }
      ?H dc:relation ?X .
      FILTER (?X = <http://waltdisney.com/donald_duck> )
   }

}
```

```
// Create variables
RDFVariable x = new RDFVariable("x");
RDFVariable y = new RDFVariable("y");
RDFVariable n = new RDFVariable("n");
RDFVariable h = new RDFVariable("h");

// Compose query
RDFAskQuery q = new RDFAskQuery()
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("dc"))
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("foaf"))
 .AddPatternGroup(new RDFPatternGroup("PG1")
  .AddPattern(new RDFPattern(y, dogOf, x))
  .AddPattern(new RDFPattern(x, name, n).UnionWithNext())
  .AddPattern(new RDFPattern(x, knows, h))
  .AddPattern(new RDFPattern(h, RDFVocabulary.DC.RELATION, x))
  .AddFilter(new RDFComparisonFilter(RDFQueryEnums.RDFComparisonFlavors.EqualTo, x,
donald_duck)))));

// Apply query
RDFAskQueryResult askResult = q.ApplyToGraph(waltdisney);
```

CREATING AND EXECUTING CONSTRUCT QUERIES

SPARQL "Construct" query type is modeled as instance of **RDFConstructQuery**:

```
// CREATE CONSTRUCT QUERY
RDFConstructQuery constructQuery = new RDFConstructQuery();
```

It is possible to apply a "Construct" query on graphs, stores or federations, producing results in form of tabular data modeled as instance of **RDFConstructQueryResult**:

```
// APPLY CONSTRUCT QUERY TO GRAPH
RDFConstructQueryResult constructQueryResult = constructQuery.ApplyToGraph(graph);

// APPLY CONSTRUCT QUERY TO STORE
RDFConstructQueryResult constructQueryResult = constructQuery.ApplyToStore(store);

// APPLY CONSTRUCT QUERY TO FEDERATION
RDFConstructQueryResult constructQueryResult = constructQuery.ApplyToFederation(federation);
```

The "*ConstructResults*" property is an ADO.NET datatable storing result filled templates.

Let's model a simple SPARQL "Construct" query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT {
 ?Y rdf:type foaf:dog .
}
WHERE{

 #PG1
 {
  ?Y dc:dogOf ?X .
  OPTIONAL { ?X foaf:age ?N } .
  FILTER ( ?N >= "45.0" )
 }

}
LIMIT 10
```

```
// Create variables
RDFVariable x = new RDFVariable("x");
RDFVariable y = new RDFVariable("y");
RDFVariable n = new RDFVariable("n");
RDFVariable h = new RDFVariable("h");

// Compose query
RDFConstructQuery q = new RDFConstructQuery()
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("rdf"))
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("dc"))
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("foaf"))
 .AddPatternGroup(new RDFPatternGroup("PG1")
   .AddPattern(new RDFPattern(y, dogOf, x))
   .AddPattern(new RDFPattern(x, age, n).Optional())
   .AddFilter(new RDFComparisonFilter(
        RDFQueryEnums.RDFComparisonFlavors.GreaterOrEqualThan, n, new RDFPlainLiteral("45.0"))))
 .AddTemplate(new RDFPattern(y, RDFVocabulary.RDF.TYPE, dog))
 .AddModifier(new RDFLimitModifier(10));

// Apply query
RDFConstructQueryResult constructResult = q.ApplyToGraph(waltdisney);
```

CREATING AND EXECUTING DESCRIBE QUERIES

SPARQL "Describe" query type is modeled as instance of **RDFDescribeQuery**:

```
// CREATE DESCRIBE QUERY
RDFDescribeQuery describeQuery = new RDFDescribeQuery();
```

It is possible to apply a "Describe" query on graphs, stores or federations, producing results in form of tabular data modeled as instance of **RDFDescribeQueryResult**:

```
// APPLY DESCRIBE QUERY TO GRAPH
RDFDescribeQueryResult describeQueryResult = describeQuery.ApplyToGraph(graph);

// APPLY DESCRIBE QUERY TO STORE
RDFDescribeQueryResult describeQueryResult = describeQuery.ApplyToStore(store);

// APPLY DESCRIBE QUERY TO FEDERATION
RDFDescribeQueryResult describeQueryResult = describeQuery.ApplyToFederation(federation);
```

The "*DescribeResults*" property is an ADO.NET datatable storing result informations.

Let's model a simple SPARQL "Describe" query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

DESCRIBE ?Y
WHERE {

 #PG1
 {
  ?Y dc:dogOf ?X .
  ?X foaf:name ?N .
  FILTER ( REGEX(STR(?N), "Mickey", "i") ) .
 }

}
LIMIT 20
```

```
// Create variables
RDFVariable x = new RDFVariable("x");
RDFVariable y = new RDFVariable("y");
RDFVariable n = new RDFVariable("n");
RDFVariable h = new RDFVariable("h");

// Compose query
RDFDescribeQuery q = new RDFDescribeQuery()
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("dc"))
 .AddPrefix(RDFNamespaceRegister.GetByPrefix("foaf"))
 .AddPatternGroup(new RDFPatternGroup("PG1")
    .AddPattern(new RDFPattern(y, dogOf, x))
    .AddPattern(new RDFPattern(x, name, n))
    .AddFilter(new RDFRegexFilter(n, new Regex(@"Mickey", RegexOptions.IgnoreCase))))
 .AddDescribeTerm(y) //Only resources and variables are allowed to be describe terms
 .AddModifier(new RDFLimitModifier(20));

// Apply query
RDFDescribeQueryResult describeResult = q.ApplyToGraph(waltdisney);
```

CONNECTING TO SPARQL ENDPOINTS

RDFSharp supports modeling of SPARQL endpoints as instances of **RDFSPARQLEndpoint**:

```
// CREATE SPARQL ENDPOINTS
var dbPediaEndpnt     = new RDFSPARQLEndpoint(new Uri("https://dbpedia.org/sparql"));
var playgroundEndpnt  = new RDFSPARQLEndpoint(new Uri("http://sparql-
playground.sib.swiss/sparql"));
var wikidataEndpnt    = new RDFSPARQLEndpoint(new Uri("https://query.wikidata.org/sparql"));
var linkgeodataEndpnt = new RDFSPARQLEndpoint(new Uri("http://linkedgeodata.org/sparql"));
```

Optional querystring parameters, defined by the SPARQL standard, can be eventually specified through the ***AddDefaultGraphUri*** and ***AddNamedGraphUri*** methods:

```
// SPECIFY DEFAULT-GRAPH-URI PARAMETER TO BE SENT TO THE ENDPOINT
dbPediaEndpnt.AddDefaultGraphUri("http://dbpedia.org");
```

Queries can be sent to the endpoint with the ***ApplyToSPARQLEndpoint*** method:

```
// APPLY SELECT QUERY TO SPARQL ENDPOINT
RDFSelectQueryResult selectQueryResult = selectQuery.ApplyToSPARQLEndpoint(dbPediaEndpnt);

// APPLY ASK QUERY TO SPARQL ENDPOINT
RDFAskQueryResult askQueryResult = askQuery.ApplyToSPARQLEndpoint(dbPediaEndpnt);

// APPLY CONSTRUCT QUERY TO SPARQL ENDPOINT
RDFConstructQueryResult constructQueryResult = constructQuery.ApplyToSPARQLEndpoint(dbPediaEndpnt);

// APPLY DESCRIBE QUERY TO SPARQL ENDPOINT
RDFDescribeQueryResult describeQueryResult = describeQuery.ApplyToSPARQLEndpoint(dbPediaEndpnt);
```

Every type of SPARQL query can be stringified and sent to a valid SPARQL endpoint, obtaining a result which is compatible with W3C standards:

- Tabular query types (ASK, SELECT) instruct the endpoint to provide response in W3C "**SPARQL Query Results XML Format**" standard

- Graph query types (CONSTRUCT, DESCRIBE) instruct the endpoint to provide response in W3C "**Terse RDF Triple Language**" standard

MIRELLA ENGINE: TIPS & TRICKS

Evaluation order of patterns, pattern groups, values and subqueries is **sequential**: you may experience that the same query can run faster if you try to rearrange your patterns by juxtaposing ones with common variables, inducing more inner-joins than cartesian products (be careful to maintain expected semantics of optional and union operators!)

Optional and Union operators impose their null-conservative logics within all the time window required by their semantics, adopting a more expensive left-outer-join or union strategy to merge intermediate data tables

Projection operator of SELECT queries is recommended and can help in slightly increasing query performances: ensure to specify only desired projection variables in your queries; if you do not specify any, the engine will behave like a SELECT *

In order to shorten the string representation of your queries for better readability, you should always add to your queries the most commonly expected prefixes

When grouping and aggregating SPARQL results, remember that many aggregators (**AVG**, **SUM**, **MIN**, **MAX**) are optimized to work with bindings in form of **numeric typed literal**, otherwise weird results due to plain literal lossy abstraction may happen!

When modeling property paths, remember that the query engine translates them into an equivalent chain of patterns: since *ground* patterns (without variables in any place) are not supported, you may experience that your property path does not appear in the query at all. This is because it has been detected as ground and has not been activated!

Use plain literals to model labels, descriptions and textual data on which you expect to perform lexical filters, regexes or comparisons; use **typed literals** when you require higher level of intelligence deriving from **data semantics** (numbers, dates, boolean)

Organize your RDF data sources in federations whenever possible: it leads to a better distribution of computation tasks and gives you a granular control on location and availability of your data (no more need to have a store with millions of quadruples!)

Avoid the temptation of asking for *?S ?P ?O*, or poorly selective, bomb patterns!