An introduction to SHACL

And its possible uses in the **@Web** platform

Leandro Lovisolo
leandro.lovisolo@supagro.inra.fr

INRA SupAgro and INRIA GraphiK
Montpellier, France

October 22, 2015

Outline of the presentation

- Introduction to SHACL
- Supported constraints (with examples)
- Operations supported by a SHACL engine
- An integrity constraint implemented as a SHACL shape
- Comparision between SHACL, Shape Expressions and raw SPARQL

Introduction to SHACL

Introduction to SHACL

- SHACL is a language for describing constraints in RDF graphs.

Introduction to SHACL

- SHACL is a language for describing constraints in RDF graphs.
- Constraints are grouped into *shapes* that apply to nodes in a *data graph*.

Introduction to SHACL

- SHACL is a language for describing constraints in RDF graphs.
- Constraints are grouped into *shapes* that apply to nodes in a *data graph*.
- Shapes are described in RDF and stored in a *shapes graph*.

Introduction to SHACL

- SHACL is a language for describing constraints in RDF graphs.
- Constraints are grouped into *shapes* that apply to nodes in a *data graph*.
- Shapes are described in RDF and stored in a *shapes graph*.
- The simplest interface to a SHACL processor has two inputs:

Introduction to SHACL

- SHACL is a language for describing constraints in RDF graphs.
- Constraints are grouped into *shapes* that apply to nodes in a *data graph*.
- Shapes are described in RDF and stored in a *shapes graph*.
- The simplest interface to a SHACL processor has two inputs:
  - A data graph containing the data to be validated

Introduction to SHACL

- SHACL is a language for describing constraints in RDF graphs.
- Constraints are grouped into *shapes* that apply to nodes in a *data graph*.
- Shapes are described in RDF and stored in a *shapes graph*.
- The simplest interface to a SHACL processor has two inputs:
  - A data graph containing the data to be validated
  - A shapes graph containing shape definitions and other information that can be used e.g. to determine which nodes in the data graph should be evaluated against which shapes

A simple shapes graph

```
ex:IssueShape
  a sh:Shape ;
  sh:scopeClass ex:Issue;
  sh:property [
    sh:predicate ex:state ;
    sh:allowedValues (ex:unassigned
                      ex:assigned) ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:predicate ex:reportedBy ;
    sh:valueShape ex:UserShape ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .
```

```
ex:UserShape
  a sh:Shape ;
  sh:property [
    sh:predicate foaf:name ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:predicate foaf:mbox ;
    sh:nodeKind sh:IRI ;
    sh:minCount 1 ;
  ] .
```

An example of a corresponding valid data graph

```
inst:Issue1
  a ex:Issue ;
  ex:state ex:unassigned ;
  ex:reportedBy inst:User2 .

inst:User2
  a foaf:Person ;
  foaf:name "Bob Smith" ;
  foaf:mbox <mailto:bob@example.org> ;
  foaf:mbox <mailto:rs@example.org> .
```

```
inst:Issue3
  a ex:Issue ;
  ex:state ex:unsinged ;
  ex:reportedBy inst:User4 .

inst:User4
  a foaf:Person ;
  foaf:name "Bob Smith",
            "Robert Smith" ;
  foaf:mbox <mailto:bob@example.org>
  foaf:mbox <mailto:rs@example.org> .
```

Shapes

Validation process (I)

- Shapes are instances of the class `sh:Shape`.

Shapes

Validation process (I)

- Shapes are instances of the class `sh:Shape`.
- A *shape* is a group of constraints that can be validated against nodes.

Shapes

Validation process (I)

- Shapes are instances of the class `sh:Shape`.
- A *shape* is a group of constraints that can be validated against nodes.
- If a node is validated against a constraint then it's called the *focus node*.

Shapes

Validation process (I)

- Shapes are instances of the class `sh:Shape`.
- A *shape* is a group of constraints that can be validated against nodes.
- If a node is validated against a constraint then it's called the *focus node*.
- Shapes may have *scopes* that instruct a SHACL processor on how to select the focus nodes (e.g. class-based scopes, individual scopes, etc.)

Shapes

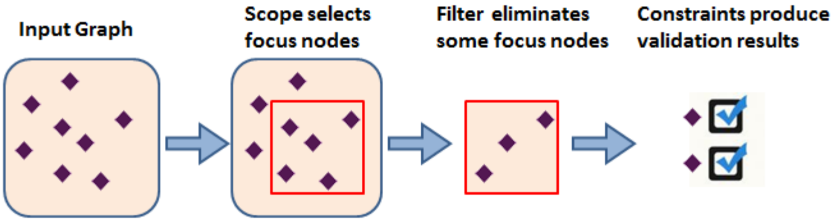
Validation process (I)

- Shapes are instances of the class `sh:Shape`.
- A *shape* is a group of constraints that can be validated against nodes.
- If a node is validated against a constraint then it's called the *focus node*.
- Shapes may have *scopes* that instruct a SHACL processor on how to select the focus nodes (e.g. class-based scopes, individual scopes, etc.)
- Shapes may also have *filter shapes* that narrow down the scope (e.g. instances of the class that have a certain number of values for a given property.)

Shapes

Validation process (II)

The following picture illustrates the validation process.



Shape scopes

Individual scope

Individual nodes can point to the shapes they're supposed to be validated against using the property `sh:nodeShape`, e.g.:

```
ex:ExampleShape
  a sh:Shape ;
  sh:constraint [
    ...
  ] .

ex:ExampleInstance
  sh:nodeShape ex:ExampleShape .
```

Shape scopes

Class-based scope

The property `sh:scopeClass` can be used to link a `sh:Shape` with an `rdfs:Class` and all its subclasses (by following `rdfs:subClassOf`.)

```
ex:ExampleClass
  a rdfs:Class .

ex:ExampleShape
  a sh:Shape ;
  sh:scopeClass ex:ExampleClass ;
  sh:constraint [
    ...
  ] .

ex:ExampleInstance
  rdf:type ex:ExampleClass .
```

Shape scopes

General scopes

It's possible to define many kinds of custom scopes. The example below selects nodes that have at least one value for property `ex:myProperty`.

```
ex:PropertyScopeExampleShape
  a sh:Shape ;
  sh:scope [
    a sh:PropertyScope ;
    sh:predicate ex:myProperty ;
  ] ;
  sh:constraint [
    ...
  ] .
```

Shape scopes

Filter shapes

Filter shapes are used to apply a shape just to a subset of a scope, e.g.:

```
ex:FilteredExampleShape
  a sh:Shape ;
  sh:scopeClass ex:ExampleClass ;
  sh:filterShape [
    sh:property [
      sh:predicate ex:requiredProperty ;
      sh:hasValue ex:requiredValue ;
    ]
  ] ;
  sh:property [
    sh:predicate ex:someProperty ;
    sh:minCount 1 ;
  ] .

ex:FilteredShapeValidExampleInstance
  rdf:type ex:ExampleClass
  ex:someProperty ex:someValue ;
  ex:requiredProperty ex:requiredValue .
```

Constraints

SHACL constraints can be grouped into the following categories:

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.
- Inverse property constraints (`sh:inverseProperty`)

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.
- Inverse property constraints (`sh:inverseProperty`)
- Property pair constraints

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.
- Inverse property constraints (`sh:inverseProperty`)
- Property pair constraints
  - `sh:EqualConstraint`, `sh:NotEqualConstraint`
  - `sh:LessThanConstraint`, `sh:LessThanOrEqualConstraint`
  - etc.

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.
- Inverse property constraints (`sh:inverseProperty`)
- Property pair constraints
  - `sh:EqualConstraint`, `sh:NotEqualConstraint`
  - `sh:LessThanConstraint`, `sh:LessThanOrEqualConstraint`
  - etc.
- Logical constraints

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.
- Inverse property constraints (`sh:inverseProperty`)
- Property pair constraints
  - `sh:EqualConstraint`, `sh:NotEqualConstraint`
  - `sh:LessThanConstraint`, `sh:LessThanOrEqualConstraint`
  - etc.
- Logical constraints
  - `sh:NotConstraint`
  - `sh:AndConstraint`, `sh:OrConstraint`
  - etc.

Constraints

SHACL constraints can be grouped into the following categories:

- Property constraints (`sh:property`)
  - `sh:hasValue`
  - `sh:allowedValues`
  - `sh:valueClass`
  - `sh:valueShape`
  - `sh:minCount`, `sh:maxCount`
  - etc.
- Inverse property constraints (`sh:inverseProperty`)
- Property pair constraints
  - `sh:EqualConstraint`, `sh:NotEqualConstraint`
  - `sh:LessThanConstraint`, `sh:LessThanOrEqualConstraint`
  - etc.
- Logical constraints
  - `sh:NotConstraint`
  - `sh:AndConstraint`, `sh:OrConstraint`
  - etc.
- Native constraints (e.g. SPARQL-based)

Constraints

Property constraints by example (I)

Some examples of property constraints:

```
ex:AllowedValuesExampleShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:someProperty ;
    sh:allowedValues ( ex:Value1
                       ex:Value2
                       ex:Value3 ) ;
  ] .

ex:AllowedValuesExampleValidRes
  ex:someProperty ex:Value2 .
```

```
ex:HasValueExampleShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:property ;
    sh:hasValue ex:Green ;
  ] .

ex:HasValueExampleValidResource
  ex:property ex:Green .
```

Constraints

Property constraints by example (II)

```
ex:ValueClassExampleShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:someProperty ;
    sh:valueClass ex:ClassA ;
  ] .

ex:InstanceOfClassA
  a ex:ClassA .

ex:ValueClassExampleValidResource
  ex:someProperty ex:InstanceOfClassA .
```

```
ex:CountExampleShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:someProperty ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .

ex:CountExampleValidResource
  ex:someProperty ex:OneValue .
```

Constraints

Property constraints by example (III)

```
ex:ValueShapeExampleShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:someProperty ;
    sh:valueShape [
      a sh:Shape ;
      sh:predicate [
        sh:predicate ex:nestedProperty ;
        sh:minCount 1 ;
      ]
    ]
  ] .

ex:ValueShapeExampleValidResource
  ex:someProperty [
    ex:nestedProperty 42 ;
  ] .
```

Constraints

Property constraints by example (IV)

```
ex:QualifiedValueShapeExShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:parent ;
    sh:minCount 2 ;
    sh:maxCount 2 ;
    sh:qualifiedValueShape [
      a sh:Shape ;
      sh:property [
        sh:predicate ex:gender ;
        sh:hasValue ex:female ;
      ]
    ] ;
    sh:qualifiedMinCount 1 ;
  ] .
```

```
ex:QualifiedValueShapeExValidRes
  ex:parent ex:John ;
  ex:parent ex:Jane .

ex:John
  ex:gender ex:male .

ex:Jane
  ex:gender ex:female .
```

Constraints

Property pair constraints

```
ex:EqualExampleShape
  a sh:Shape ;
  sh:constraint [
    a sh:EqualsConstraint ;
    sh:predicate1 ex:firstName ;
    sh:predicate2 ex:givenName ;
  ] .

ex:ValidInstance1
  ex:firstName "John" ;
  ex:givenName "John" .
```

```
ex:LessThanExampleShape
  a sh:Shape ;
  sh:constraint [
    a sh:LessThanConstraint ;
    sh:predicate1 ex:startDate ;
    sh:predicate2 ex:endDate ;
  ] .
```

Constraints

Logical constraints

```
ex:NotExampleShape
  a sh:Shape ;
  sh:constraint [
    a sh:NotConstraint ;
    sh:shape [
      a sh:Shape ;
      sh:property [
        sh:predicate ex:property ;
        sh:minCount 1 ;
      ] ;
    ]
  ] .

ex:InvalidInstance1
    ex:property "Some value" .
```

```
ex:SuperShape
  a sh:Shape ;
  sh:property [
    sh:predicate ex:property ;
    sh:minCount 1 ;
  ] .

ex:ExampleAndShape
  a sh:Shape ;
  sh:constraint [
    a sh:AndConstraint ;
    sh:shapes (
      ex:SuperShape
      [
        sh:property [
          sh:predicate ex:property ;
          sh:maxCount 1 ;
  ] ] ) ] .

ex:ValidInstance1
  ex:property "One" .

# Invalid: more than one property
ex:InvalidInstance2
  ex:property "One" ;
  ex:property "Two" .
```

Constraints

Native constraints

```
ex:LanguageExampleShape
  a sh:Shape ;
  sh:scopeClass ex:Country ;
  sh:constraint [
    sh:message "Values must be literals with German language tag." ;
    sh:sparql """
      SELECT $this ($this AS ?subject)
                   (ex:germanLabel AS ?predicate)
                   (?value AS ?object)
      WHERE {
        $this ex:germanLabel ?value .
        FILTER (!isLiteral(?value) || !langMatches(lang(?value), "de"))
      }
      """ ;
  ] .

ex:ValidCountry
  a ex:Country ;
  ex:germanLabel "Spanien"@de .

ex:InvalidCountry
  a ex:Country ;
  ex:germanLabel "Spain"@en .
```

Validation results

The output of a SHACL constraint validation process is a set of *validation results*, represented as RDF triples. An example is show below.

```
ex:ExampleConstraintViolation
  a sh:ValidationResult ;
  sh:severity sh:Violation ;
  sh:focusNode ex:MyCurrentNode ;
  sh:subject ex:MyCurrentNode ;
  sh:predicate ex:someProperty ;
  sh:object ex:someInvalidValue ;
  sh:message "Incorrect value: expected something else here." .
```

Operations supported by a SHACL engine

Operations supported by a SHACL engine

- **Validate graph**: Validate a whole data graph against all shapes associated with its resources, based on the available scope definitions.

Operations supported by a SHACL engine

- **Validate graph**: Validate a whole data graph against all shapes associated with its resources, based on the available scope definitions.
- **Validate shape**: Validate all nodes that are in the scope of a given shape against the constraints of that shape.

Operations supported by a SHACL engine

- **Validate graph**: Validate a whole data graph against all shapes associated with its resources, based on the available scope definitions.
- **Validate shape**: Validate all nodes that are in the scope of a given shape against the constraints of that shape.
- **Validate node against shape**: Validate a given node against the constraints of a given shape.

Operations supported by a SHACL engine

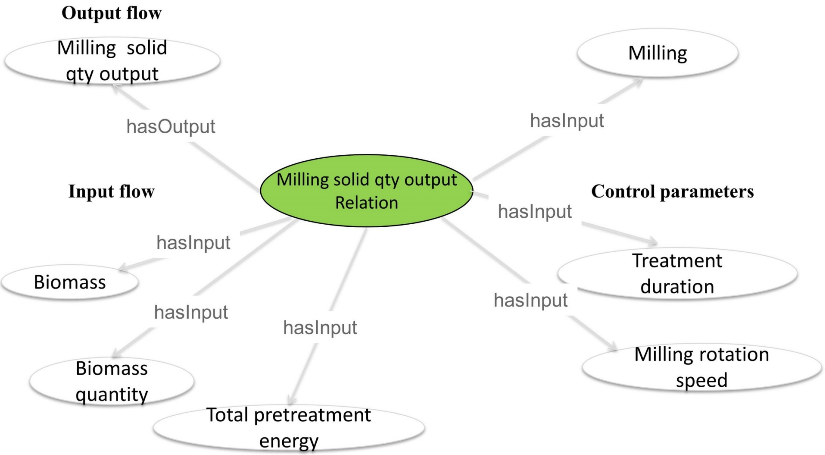
- **Validate graph**: Validate a whole data graph against all shapes associated with its resources, based on the available scope definitions.
- **Validate shape**: Validate all nodes that are in the scope of a given shape against the constraints of that shape.
- **Validate node against shape**: Validate a given node against the constraints of a given shape.
- **Validate node against constraint**: Validate a given node against a given constraint from a given shape.

Operations supported by a SHACL engine

- **Validate graph**: Validate a whole data graph against all shapes associated with its resources, based on the available scope definitions.
- **Validate shape**: Validate all nodes that are in the scope of a given shape against the constraints of that shape.
- **Validate node against shape**: Validate a given node against the constraints of a given shape.
- **Validate node against constraint**: Validate a given node against a given constraint from a given shape.
- **Validate node**: Validate a given node against the constraints of all shapes that it is in the scope of.

An integrity constraint

Milling solid quantity output relation



An integrity constraint

Guideline

"The output quantity of a step is equal to the sum of the quantity of water used and the quantity of biomass present in the step."

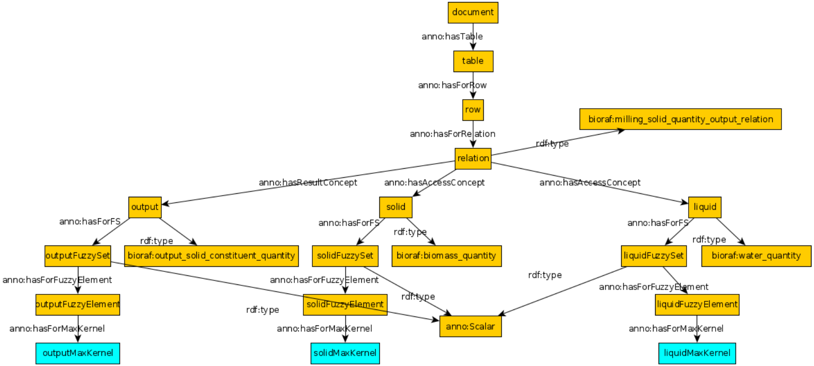
An integrity constraint

SPARQL query

```
SELECT ?docid ?doctitle ?tableid ?tabletitle ?rownum  ?solid_qty ?liquid_qty ?output_qty
WHERE {
?doc anno:hasForID ?docid ;
     dc:title ?doctitle ;
     anno:hasTable ?table .

?table anno:hasForID ?tableid ;
       dc:title ?tabletitle ;
       anno:hasForRow ?row .

?row anno:hasForRowNumber ?rownum ;
     anno:hasForRelation [a bioraf:milling_solid_quantity_output_relation ;
                          core:hasAccessConcept ?solid ;
                          core:hasAccessConcept ?liquid ;
                          core:hasResultConcept ?output] .

?solid a bioraf:biomass_quantity ;
       anno:hasForFS [a anno:Scalar ;
                      anno:hasForFuzzyElement /
                      anno:hasForMaxKernel ?solid_qty] .

?liquid a bioraf:water_quantity ;
        anno:hasForFS [a anno:Scalar ;
                       anno:hasForFuzzyElement /
                       anno:hasForMaxKernel ?liquid_qty] .

?output a bioraf:output_solid_constituent_quantity ;
        anno:hasForFS [a anno:Scalar ;
                       anno:hasForFuzzyElement /
                       anno:hasForMaxKernel ?output_qty] .

FILTER (xsd:float(?output_qty) != xsd:float(?solid_qty) + xsd:float(?liquid_qty))
}
```

An integrity constraint

Graph view of the SPARQL query



An integrity constraint

Shape Expression

```
<DocumentShape> { rdf:type anno:Document, anno:hasTable @<TableShape> }
<TableShape> { anno:hasForRow @<RowShape> }
<RowShape> { anno:hasForRelation @<MillingSolidQuantityOutputRelationShape> }

<MillingSolidQuantityOutputRelationShape> {
  rdf:type bioraf:milling_solid_quantity_output_relation,
  core:hasAccessConcept @<SolidAccessConceptShape>,
  core:hasAccessConcept @<LiquidAccessConceptShape>,
  core:hasResultConcept @<OutputResultConceptShape>
}

<SolidAccessConceptShape> {
  rdf:type bioraf:biomass_quantity,
  anno:hasForFS @<FuzzySetShape>
}

<LiquidAccessConceptShape> {
  rdf:type bioraf:water_quantity,
  anno:hasForFS @<FuzzySetShape>
}

<OutputAccessConceptShape> {
  rdf:type bioraf:output_solid_constituent_quantity,
  anno:hasForFS @<FuzzySetShape>
}

<FuzzySetShape> {
  rdf:type anno:Scalar,
  anno:hasForFuzzyElement @<FuzzyElementShape>
}

<FuzzyElementShape> { anno:hasForMaxKernel xsd:string }
```

An integrity constraint

SHACL shapes graph (I)

```
anno:MillingSolidOutputQuantityRelationshipShape
  a sh:Shape ;
  sh:scopeClass bioraf:milling_solid_quantity_output_relation ;

  sh:filterShape [
    sh:inverseProperty [
      sh:predicate anno:hasForRelation ;
      sh:valueShape [
        sh:inverseProperty [
          sh:predicate anno:hasForRow ;
          sh:valueClass anno:Table ;
          sh:minCount 1 ;
          sh:maxCount 1 ;
        ] ;
      ] ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
    ]
  ] ;

  sh:property [
    sh:predicate core:hasAccessConcept ;
    sh:qualifiedValueShape [
      sh:property [
        sh:predicate rdf:type ;
        sh:hasValue bioraf:biomass_quantity
      ]
    ] ;
    sh:qualifiedMinCount 1 ;
    sh:qualifiedMaxCount 1 ;
  ] ;

sh:property [
  sh:predicate core:hasAccessConcept ;
  sh:qualifiedValueShape [
    sh:property [
      sh:predicate rdf:type ;
      sh:hasValue bioraf:water_quantity
    ]
  ] ;
  sh:qualifiedMinCount 1 ;
  sh:qualifiedMaxCount 1 ;
] ;

sh:property [
  sh:predicate core:hasResultConcept ;
  sh:valueClass bioraf:output_solid_constituent_quanti
  sh:minCount 1 ;
  sh:maxCount 1 ;
] ;
```

An integrity constraint

SHACL shapes graph (II)

```
    sh:constraint [
      sh:predicate anno:width ;
      sh:sparql """
        SELECT $this ($this AS ?subject)
               (CONCAT("Output quantity must be the sum of the solid and liquid input quantities
                        (solid=", STR(?solid_qty),
                       ", liquid=", STR(?liquid_qty),
                       ", output=", STR(?output_qty), ")") as ?message)
        WHERE {
          $this core:hasAccessConcept ?solid ;
                core:hasAccessConcept ?liquid ;
                core:hasResultConcept ?output .

          ?solid a bioraf:biomass_quantity ;
                 anno:hasForFS [a anno:Scalar ;
                                anno:hasForFuzzyElement /
                                anno:hasForMaxKernel ?solid_qty] .

          ?liquid a bioraf:water_quantity ;
                  anno:hasForFS [a anno:Scalar ;
                                 anno:hasForFuzzyElement /
                                 anno:hasForMaxKernel ?liquid_qty] .

          ?output a bioraf:output_solid_constituent_quantity ;
                  anno:hasForFS [a anno:Scalar ;
                                 anno:hasForFuzzyElement /
                                 anno:hasForMaxKernel ?output_qty] .

          FILTER (xsd:float(?output_qty) !=
                  xsd:float(?solid_qty) + xsd:float(?liquid_qty))
        }
      """ ;
  ] .
```

SHACL, ShEx and raw SPARQL pros and cons (I)

SHACL pros:

- Constraints are represented as RDF triples; no additional storage medium needed.
- Rich core constraints vocabulary.
- Possible to define arbitrary constraints using SPARQL.
- SHACL implementation readily available (Java language).
- Already being used in the industry (TopQuadrant).

SHACL cons:

- Constraints involving properties from different nodes require describing the graph structure within SPARQL queries, rendering the SHACL shapes redundant.

SHACL, ShEx and raw SPARQL pros and cons (II)

ShEx pros:

- Conceptually simple, familiar model inspired in regular languages.
- Extensible by means of semantic actions.

ShEx cons:

- No feature complete implementations available at the moment.
- Semantic actions are not fully specified in the current draft.
- Requires learning a new language.
- Constraints over paths require defining lots of intermediate shapes for internal nodes in the paths.

SHACL, ShEx and raw SPARQL pros and cons (III)

Raw SPARQL pros:

- Well known; technology and tooling readily available.
- Doesn't require introducing new dependencies into the **@Web** stack.
- Complex constraints can be implemented with less code compared to SHACL and ShEx.

Raw SPARQL cons:

- Simple constraints can be more easily and briefly expressed with SHACL and ShEx.
- Harder than SHACL and ShEx, except for the cases where SPARQL code is needed for custom constraints.

Thanks!