Data Quality

Quality assurance for enterprise data unification

Taught by:

Jaroslav Pullmann
Solutions Architect

Learning Objectives

- Understand the concept of data quality and why it matters
- Asses quality requirements for various kinds of data
- Apply appropriate means to assess quality of data
- Evaluate, communicate, and act upon data quality reports
- Operate Stardog to ensure quality of integrated data

Data Quality Overview

Data Quality Concept

- Quality as fitness for use (J. M. Juran)
  - Contextual, relative nature of data quality
  - Depending on consumers' requirements (specification)
- Different degrees of relevance ranging from compliance to critical reliability
  - Cost trade-off (maintain a quality level vs. accept fees and failures)
- General quality assurance principles
  - Test and intervene early, (close to the source), prevent errors to propagate
  - Identify any related validation targets (data, schema etc.)
  - Formalize quality constraints (machine processable), automate their validation
- Multi-dimensional
  - Intrinsic quality dimensions related to data, extrinsic related to digital infrastructure

Some Dimensions of Data Quality

Accuracy	Is the information correct?
Credibility	Are there multiple versions of the same information?
Completeness	Is information missing?
Relevance	Is the information helping answer important questions?
Timeliness	Is the information up-to-date?
Validity	Does the information conform to the definition?

- Compare to ISO/IEC 25012 Quality Dimensions

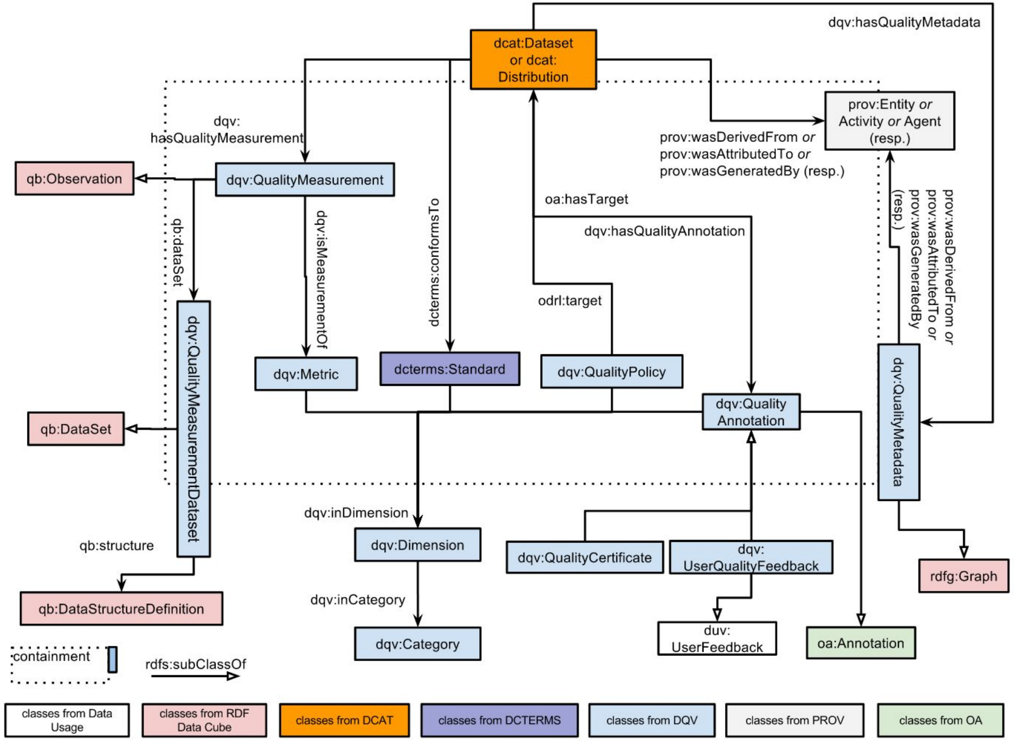
Data Quality Vocabulary (1/2)

- Data Quality Vocabulary (DQV)
  - Model for a structured and actionable expression of data quality
- Quality *Dimension* (`dqv:Dimension`)
  - Addresses a user-oriented *qualitative* characteristic of a dataset, for example:
    - Is it "complete", "valid", "accurate", "up to date", (technically) "available", etc.
  - Related dimensions are grouped to higher-level categories (`dqv:Category`)
  - A dimension is measured via one or more quantifiable metrics
- Quality *Metric* (`dqv:Metric`)
  - Strategy implementing a *measurable* assessment of a data quality dimension
  - Observes a concrete indicator, e.g., spatial resolution (accuracy)
  - Value range is often numeric (percentage) or boolean

Data Quality Vocabulary (2/2)

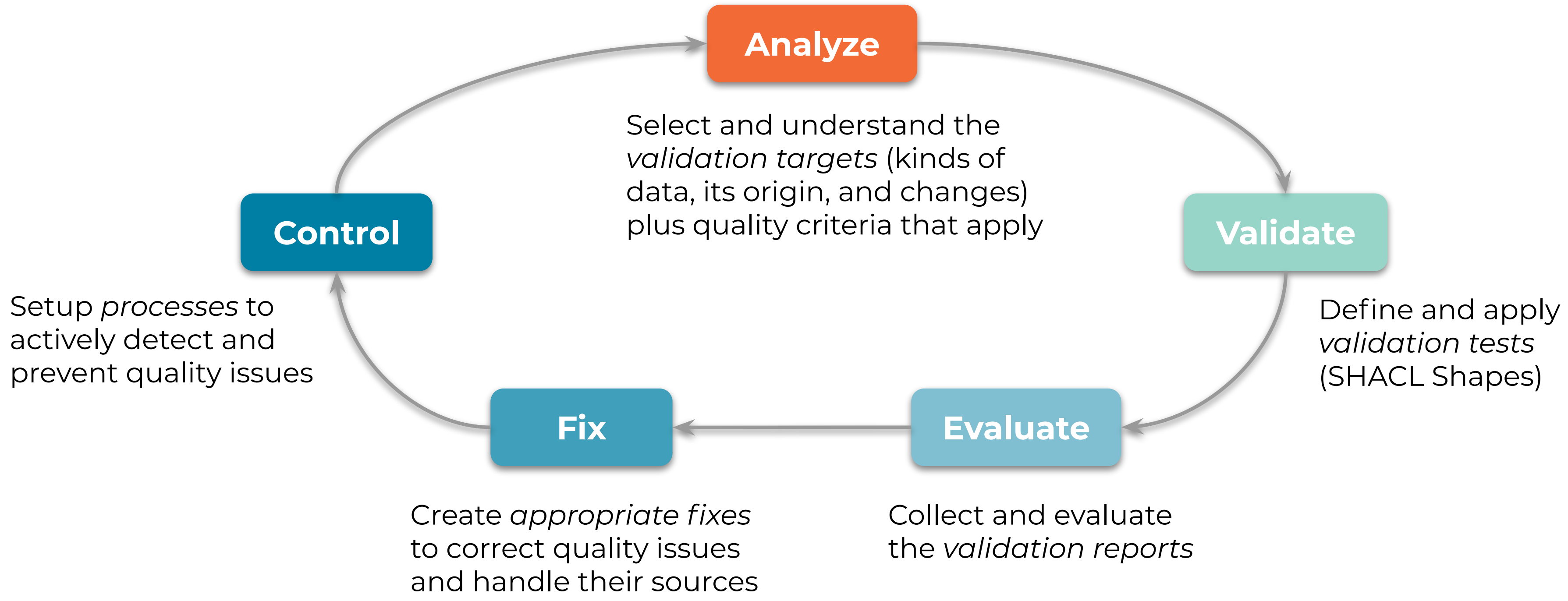
- Measurement (`dqv:QualityMeasurement`)
  - Result of evaluating a given dataset against a specific quality metric
- Quality annotation (`dqv:QualityAnnotation`)
  - Accompanying quality statements, such as ratings, certificates, or feedback
- Metadata (`dqv:QualityMetadata`)
  - Group of quality certificates, policies, measurements, and annotations
  - Further context information about the evaluation (agent or service, time, etc.)

Data Quality Vocabulary / Outline



Source: https://www.w3.org/TR/vocab-dqv/DataQuality0.2.9.svg

Quality Assurance Cycle



Analyze

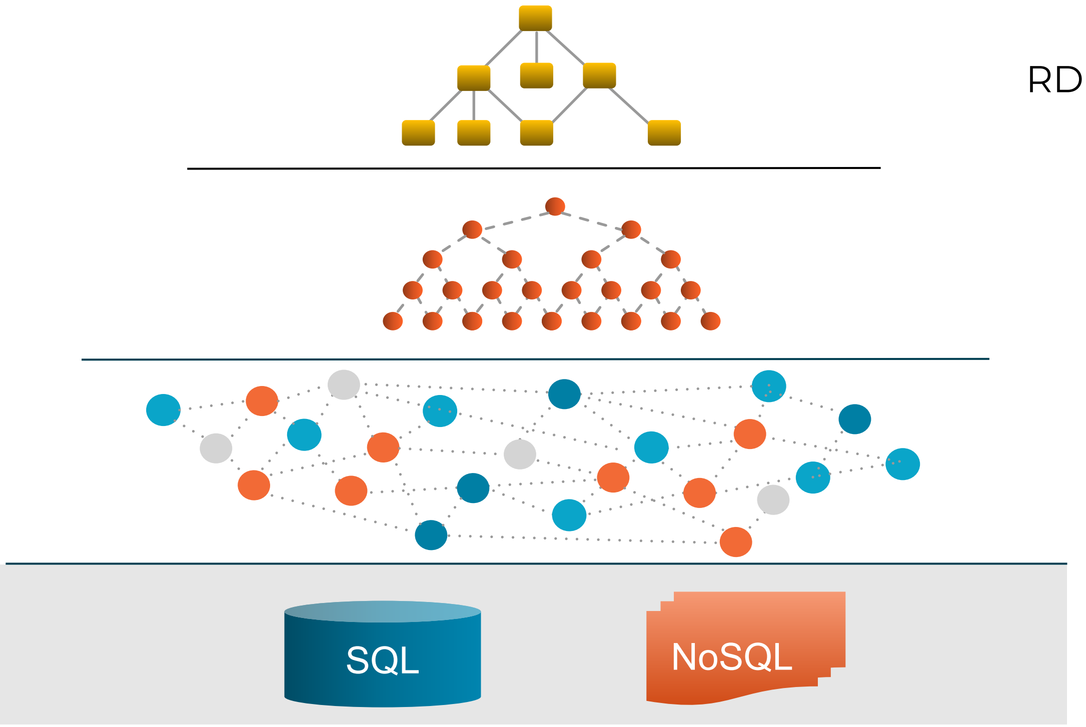
Validation Targets

Graph data (schema) — RDFS/OWL/SWRL

Graph data (concepts) — SKOS

Graph data (instances) — RDF

Source data (non-graph)



Structured Data (SQL)

- Relational data
  - Structured in tables (entity types) of rows (instances) and columns (attributes)
  - Maintained in database systems (RDBMS) and files (Excel, delimiter-separated files)
- Schema defined on creation via Data Definition Language (DDL) part of SQL
- Syntax conformance and validity constraints enforced by RDBMS, for example:
  - Presence (`NOT NULL`)
  - Uniquiness (`UNIQUE`)
  - Referential integrity (`PRIMARY/FOREIGN KEY`)
  - Data-type conformance (`DATE`)
  - Value range (`CHECK`)

Structured Data / Example

SQL / Data

```
+----+------------------+--------------+---------+
| id | name             | release_date | artist |
+----+------------------+--------------+---------+
| 1  | please please me | 1963-03-22   |       5 |
| 2  | mccartney        | 1970-04-17   |       2 |
| 3  | imagine          | 1971-10-11   |       1 |
| 4  | rubber soul      | 1965-12-03   |       5 |
| 5  | let it be        | 1970-05-08   |       5 |
+----+------------------+--------------+---------+
```

SQL / DDL

```
CREATE TABLE Album (
    id              INT,
    name            VARCHAR(30)
                    NOT NULL,
    release_date    DATE,
    artist          INT,
    PRIMARY KEY     (id),
    FOREIGN KEY (artist)
        REFERENCES Artist (id)
);
```

Semi-Structured Data (NoSQL)

- Non-relational, hierarchical data (documents)
  - Structured as hierarchies of non-overlapping blocks (trees)
  - Individual files or managed by document-oriented databases
- Extensible Markup Language (XML)
  - Elements, attributes, processing instructions, etc.
- JavaScript Object Notation (JSON)
  - Objects, arrays and literal values (string, number, boolean)
- Standard syntax definitions (a) and schema languages (b) exist to ensure the data is:

  (a) *Well-formed*, i.e., it uses the correct syntax and could be at least read

  (b) *Valid*: it additionally complies with a (custom) schema and could be reliably processed
- Document schema is optional but capable of expressing complex validation constraints

Semi-Structured Data / Example

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<MusicDB
  xmlns="http://stardog.com/tutorial/">

<Album id="Let_It_Be">
  <name>Let It Be</name>
  <date>1970-05-08</date>
  <artist ref="The_Beatles"/>
  <track ref="Across_the_Universe"/>
  <track ref="Dig_It_(Beatles_song)"/>
  <!-- ... -->
  <producer ref="Phil_Spector"/>
</Album>

</MusicDB>
```

XML Schema

```
<xs:element name="MusicDB">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="tutorial:Album"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Album">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="tutorial:name"/>
        <xs:element ref="tutorial:date"/>
        <xs:element ref="tutorial:artist"/>
        <xs:element minOccurs="1"
             maxOccurs="unbounded" ref="tutorial:track"/>
        <xs:element ref="tutorial:producer"/>
      </xs:sequence>
      <xs:attribute name="id" use="required"
             type="xs:string" />
    </xs:complexType>
  </xs:element>
```

Graph Data / Instances

- Vast majority of graph data
- Arbitrary statements (assertions) on instances of ontology classes (RDF statements)
- Unlike XML or JSON various syntaxes exists for RDF, constraints to operate on triple model
- Instance constraints likely reflect domain-specific assumptions about valid resources
- Example constraints for the Music DB domain:
    - *“An album must contain at least one track”*
    - *“The release date property of an album should be of type `xsd:date`”*
    - *“A track should have a non-zero length”*

Graph Data / Example

Instance data (Turtle)

```
:Let_It_Be
    :artist :The_Beatles ;
    :date "1970-05-08"^^xsd:date ;
    :name "Let It Be" ;
    :producer :Phil_Spector ;
    :track :Across_the_Universe , :Get_Back ...
```

RDF Schema (Turtle)

```
:Album a rdfs:Class ;
   rdfs:label "Album" .

:date a rdf:Property ;
   rdfs:label "date" ;
   rdfs:comment "The release date of an album." ;
   rdfs:domain :Album ;
   rdfs:range xsd:date .

:artist a rdf:Property ;
   rdfs:label "artist" ;
   rdfs:comment "The artist that performed this album." ;
   rdfs:domain :Album ;
   rdfs:range :Artist .

:track a rdf:Property ;
   rdfs:label "track" ;
   rdfs:comment "A song included in an album." ;
   rdfs:domain :Album ;
   rdfs:range :Song .
```

Graph Data / Concepts

- Simple Knowledge Organization System (SKOS)
  - Semi-formal knowledge representation (thesauruses, classification schemes, tags)
  - Hierarchical (`skos:broader`) and associative (`skos:related`) networks of concepts
  - Linguistic annotation is relevant (`skos:prefLabel`) and exhaustive (`skos:altLabel`)
- Example constraints on taxonomy concepts:
  - "*Each concept should refer to the defining scheme*"
  - "*Each concept must have a main title in English*"
  - "*There should be exactly one main title per language tag*"
  - "*Concepts must not recursively link to itself*" *(prevent cyclical hierarchies)*

Concepts / Example

Concept usage

```
:Let_It_Be :track :Across_the_Universe .

:Across_the_Universe
   a :Song ;
  :name "Across the Universe"@en ;
  :genre genre:RockMusic ;
.
```

Scheme definition

```
@prefix : <http://stardog.com/tutorial/> .
@prefix genre: <http://stardog.com/tutorial/music_genre/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

:MusicGenre
    a owl:Class ; rdfs:subClassOf skos:Concept ;
    rdfs:label "Music genre"@en ;  rdfs:isDefinedBy : .

genre: a skos:ConceptScheme ;
    rdfs:label "Concept scheme of music genres" ;
    skos:hasTopConcept genre:AbstractGenre .

genre:RockMusic
    a model:MusicGenre   ;
    skos:prefLabel "Rock music"@en ;
    skos:broader genre:PopularMusic ;
    skos:inScheme genre: .

genre:PopularMusic
    a model:MusicGenre   ...
```

Graph Data / Models

- "Terminology" (classes and properties) for instantiation in RDF graphs
- Ranging from simple hierarchies (RDF Schema) to class expressions (OWL)
- Example model-related constraints:
  - *"There should be at most one value type (`rdfs:range`) specified per property"*
  - *"Property names should start with the verbs 'has' or 'is' "*
  - *"Each node (especially blank nodes) should specify a type"*
  - *"At most one single value should be defined for the functional property ID"*

Validate

Overview

- "Unit testing" paradigm
  - Compare to quality assurance in software development
  - Treat data as code, automate testing as part of the continuous development (CD)
- Options in Stardog: database consistency check and constraint validation (SHACL)
- Database (in)consistency testing
  - Tests whether the database is consistent w.r.t. inferences entailed by the schema
- RDF schema languages (RDFS, OWL) are not sufficient for constraint validation
  - They are descriptive, not prescriptive, unlike DDL or document schema languages
  - Do not define constraints, but inference rules to derive (new) facts out of the asserted
  - These *may* disclose inconsistencies (logical errors) contradicting the stated facts

Consistency Test / Invalid Datatype

RDF Schema / Turtle syntax

```
:date a rdf:Property ;
    rdfs:label "date" ;
    rdfs:comment "The release date of an album." ;
    rdfs:domain :Album ;
    rdfs:range xsd:date .
```

RDFS Inference: Objects of the property `:date` are inferred to be of the datatype `xsd:date`. Any invalid values will make the database inconsistent. The value of type `string` in this statement:

```
:MyAlbum :date "2020-01-01" .
```

will conflict with the inferred datatype `date`:

```
$stardog reasoning explain -i music
```

```
VIOLATED :date rdfs:range xsd:date
   ASSERTED :date rdfs:range xsd:date
   ASSERTED :MyAlbum :date "2020-01-01"
```

Consistency Test / Disjoint Types

OWL Ontology / Turtle syntax

```
:date a rdf:Property ;
    rdfs:label "date" ;
    rdfs:comment "The release date of an album." ;
    rdfs:domain :Album ;
    rdfs:range xsd:date .

:artist a rdf:Property ;
   rdfs:label "artist" ;
   rdfs:comment "The artist that performed this album." ;
   rdfs:domain :Album ;
   rdfs:range :Artist .

:producer a rdf:Property ;
   rdfs:label "producer" ;
   rdfs:comment "The producer of this album." ;
   rdfs:domain :Album ;
   rdfs:range :Producer .
# artificial rule preventing Artists be Producers
:Producer owl:disjointWith :Artist .
```

OWL Inference: Objects of the property `:producer` are inferred to be of type `:Producer`.

An artificial rule prohibits a producer to be an artist, i.e., the classes `:Producer` and `:Artist` are disjoint (`owl:disjointWith`). These triples:

```
:MyAlbum :artist :me ; :producer :me .
```

are not consistent w.r.t the model:

```
VIOLATED :Producer owl:disjointWith :Artist
   ASSERTED :Producer owl:disjointWith :Artist
   INFERRED :me a :Producer
      ASSERTED :producer rdfs:range :Producer
      ASSERTED :MyAlbum :producer :me
   INFERRED :me a :Artist
      ASSERTED :artist rdfs:range :Artist
      ASSERTED :MyAlbum :artist :me
```

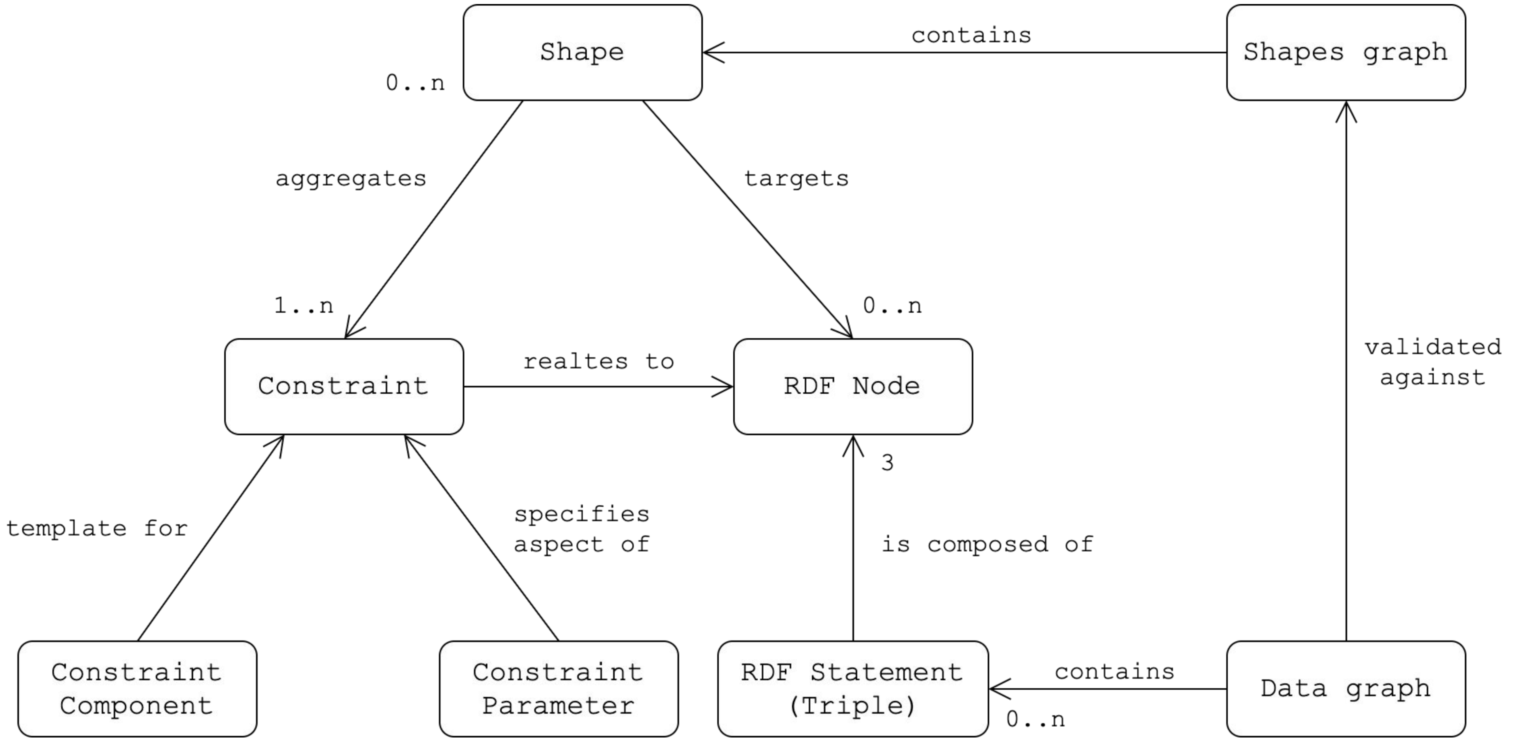
Overview / ICV

- Integrity Constraint Validation (ICV) in Stardog
  - Ensures the ingested data is valid according to a rule set
- On-demand mode: background checks of the database status
- On-commit (guard) mode: apply constraint validation as part of update transactions
  - Transactions will fail upon error preventing DB from becoming invalid
  - Configuration property: `icv.enabled=true`
- SHACL is the standard and recommended means for expressing constraints

SHACL (Shapes Constraint Language)

- Recent W3C specification (2017)
  - RDF vocabulary for describing and validating RDF data
  - Tutorial coverage: SHACL Core and SHACL-SPARQL Extension
- A *Shapes graph* defines constraints on a *data graph*
  - *Shapes graph* consists of a *Shapes* (constraint definitions) applied to "targets"
  - Data graph in Stardog comprises the default graph and all named graphs
- Result of the validation is a *Validation report*
  - The the data graph is valid if all targets conform to related shapes
- Shapes graph and validation report are RDF graphs (easy to query and manipulate)
  - RDF syntax file extensions (`*.ttl, *.rdf`)
  - Prefix `sh`, namespace `http://www.w3.org/ns/shacl#`

SHACL Outline

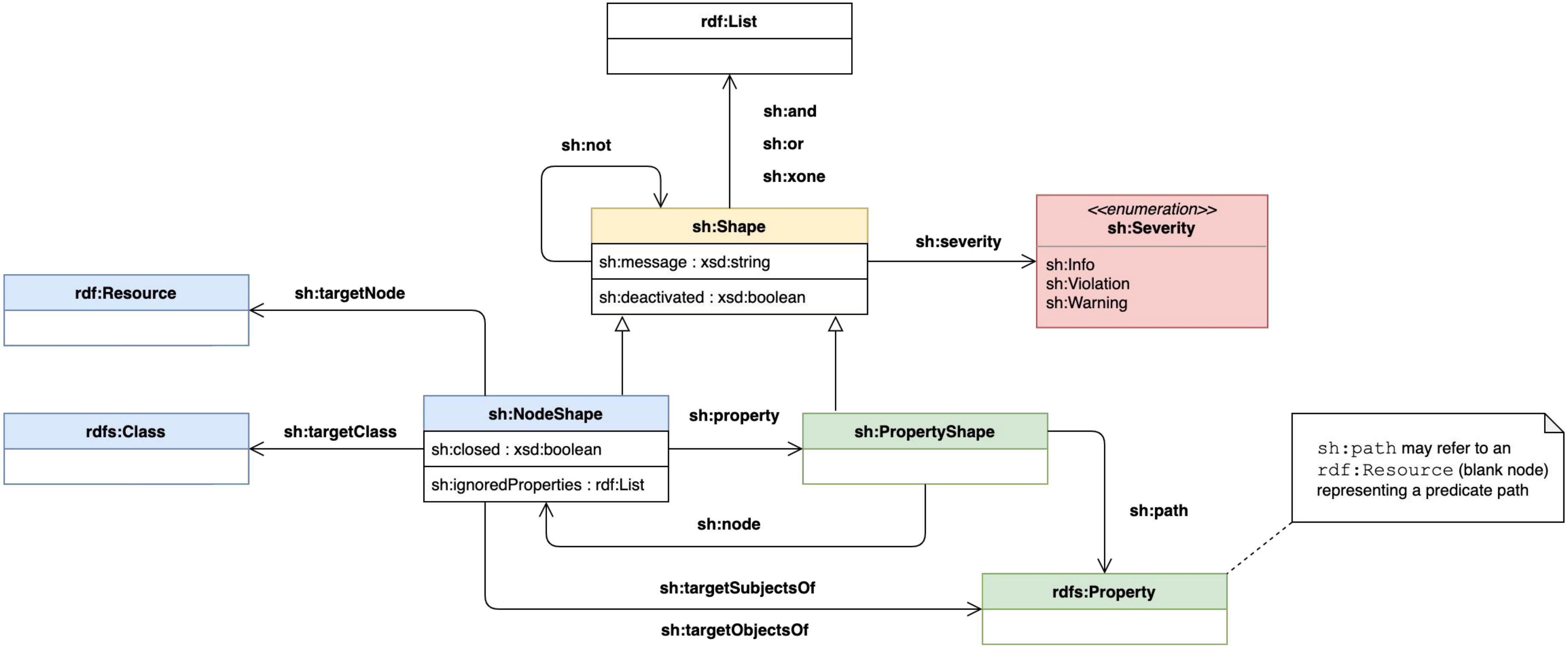


Source: ALIGNED (EU Project deliverable)

SHACL Shapes

- Define a set of constraints a validation target must satisfy
  1) *Node shapes* operate on *target nodes* (set of nodes referred to by a target expression)
  2) *Property shapes* operate on values of *predicates* specified by the `sh:path` property
- Specify inline constraint on the respective target kind (e.g., `sh:nodeKind` or `sh:minCount`)
- Or, delegate the constraint specification to an another shape (`sh:property, sh:node`)
- Relate to other shapes to compose logical expressions (e.g., `sh:and, sh:or`)
- Optionally specify a custom validation message (`sh:message`) and severity (`sh:severity`)
- Severity levels: informative (`sh:Info`), indicative (`sh:Warning`), and critical (`sh:Violation`)

SHACL Shape / Simplified Outline



SHACL Shape / Example

SHACL ICV

```
# An album must contain at least one track

prefix sh: <http://www.w3.org/ns/shacl#>
prefix : <http://stardog.com/tutorial/>

:AlbumTrackShape
      a sh:NodeShape ;
      sh:targetClass :Album ;
      sh:property [
            sh:message "An album should contain at least one track" ;
            sh:path :track;
            sh:minCount 1 ;
      ] ;
.
```

Node Shapes

- Node shapes operate upon RDF terms (subjects and objects of triples)
- Typically define validation target(s) via enumeration, node type, or connecting predicate:
    - `sh:targetNode :Please_Please_Me, :Rubber_Soul` (Listing of targeted resources)
        - Nodes missing in the data graph will *not* be reported
    - `sh:targetClass :Album` (Class of nodes this shape applies to *transitively*)
        - Applies to given class *and* any of its subclasses (`rdf:type/rdfs:subClassOf*`)
    - `sh:targetSubjectsOf :writer` (Subjects of triples with given property as predicate)
    - `sh:targetObjectsOf :track` (Objects of triples with given property as predicate)
    - Shape with types `sh:NodeShape` and `rdfs:Class` define an implicit class target of itself
- Individual nodes targeted by the above expressions become a “focus node” of validation
- Next to defining target optionally define constraints on the focus node, e.g., `sh:nodeKind`

Node Shapes / Example

Implicit class target

```
# :Album class defines its own node shape

:Album a rdfs:Class, sh:NodeShape ;
      sh:property [
            sh:path :track;
            sh:minCount 1 ;
      ] .
```

Node constraint

```
# each album should be given a (meaningful) IRI

:AlbumShape a sh:NodeShape ;
      sh:targetClass :Album ;
      sh:nodeKind sh:IRI.
```

Property Shapes

- Property shapes constrain values of a path on the focus node specified via predicate `sh:path`
- `sh:path` may refer to a single predicate (`:track`) or a SPARQL property path:

Predicate path	`:track`	`:track`
Sequence path	`:album/:track`	`( :album :track )`
Alternative path	`:track	:song`	`[ sh:alternativePath ( :track :song ) ]`
Inverse path	`^:track`	`[ sh:inversePath :track ]`
0 - n path	`rdfs:subClassOf*`	`[ sh:zeroOrMorePath rdfs:subClassOf ]`
1 - n path	`rdfs:subClassOf+`	`[ sh:oneOrMorePath rdfs:subClassOf ]`
0 - 1 path	`rdfs:subClassOf?`	`[ sh:zeroOrOnePath rdfs:subClassOf ]`

Property Shapes / Example

Simple predicate

```
:SongLengthShape
      a sh:NodeShape ;
      sh:targetClass  :Song ;
      sh:property [
            sh:path  :length ;
            sh:datatype xsd:integer ;
            sh:minExclusive 0 ;
      ] ;
.
```

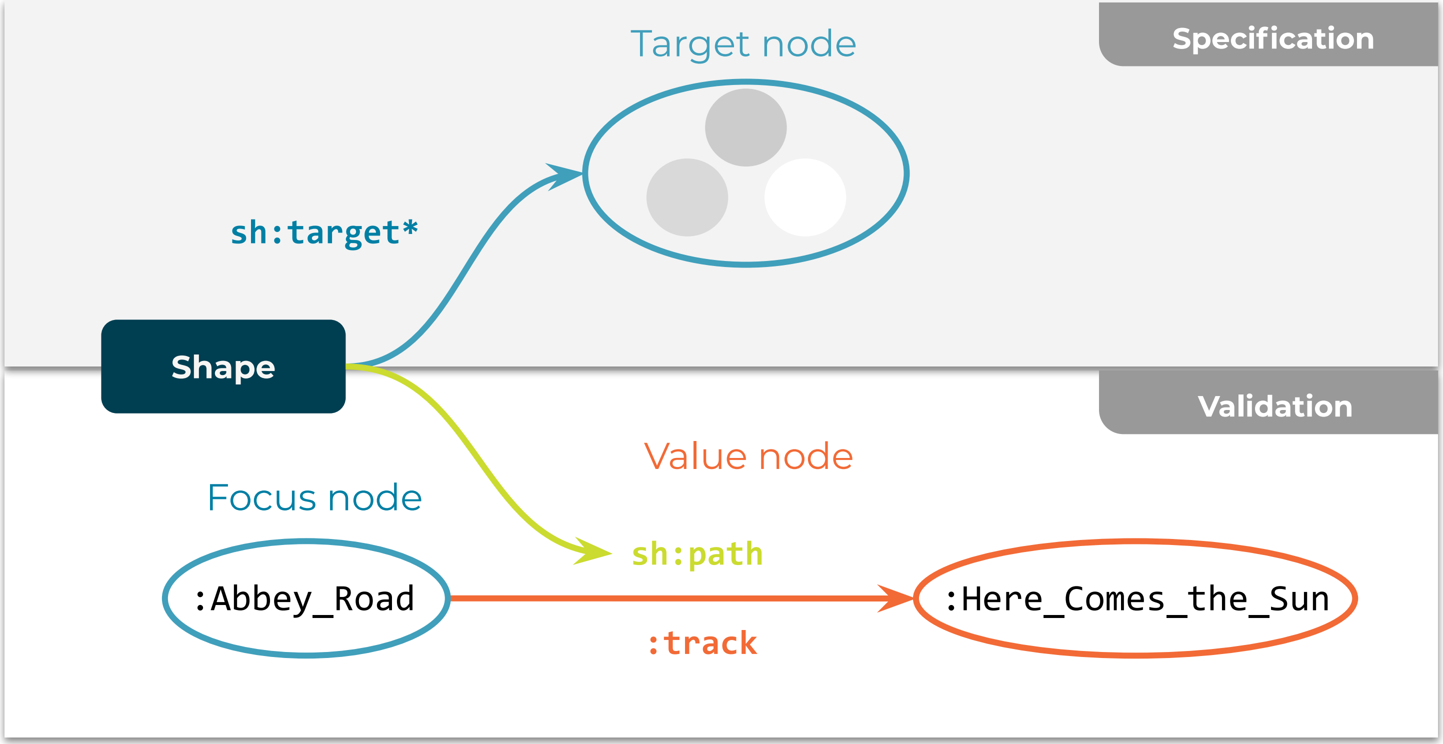
Property path

```
:SongLengthShapePath
      a sh:NodeShape ;
      sh:targetClass  :Album ;
      sh:property [
            sh:path  (:track  :length)  ;
            sh:datatype xsd:integer ;
            sh:minExclusive 0 ;
      ] ;
.
```

Node Types (1/2)

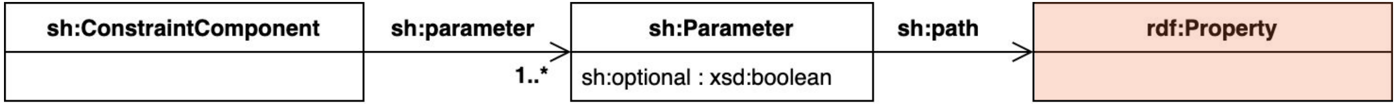
- Target node: Any node that *satisfies the target condition* of a shape.
  - Each target node becomes a focus node during the validation process.
- Focus node: A node that is *being validated* against a shape
  - Target nodes *plus* nodes selected implicitly by shape-based constraints (via `sh:node`)
- Value nodes: A node that is *used for validation*
  - For node shapes the value node is the same as the focus node
  - For property shapes any node reachable from the focus node via expression defined by the `sh:path` predicate
- Focus and value nodes is a terminology used to describe the validation process
- Target nodes denote the initial set of nodes specified for validation

Node Types (2/2)

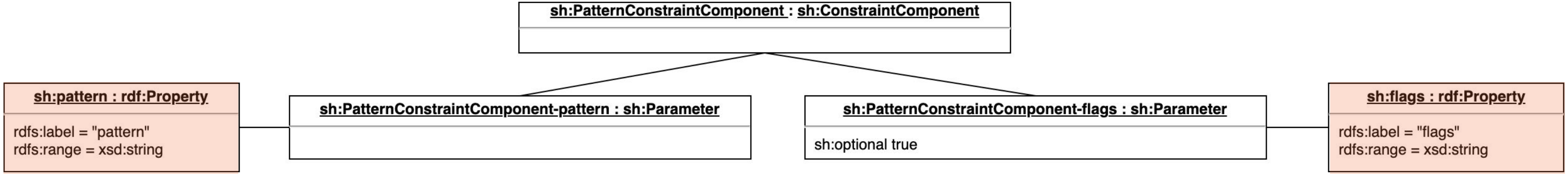


Constraints

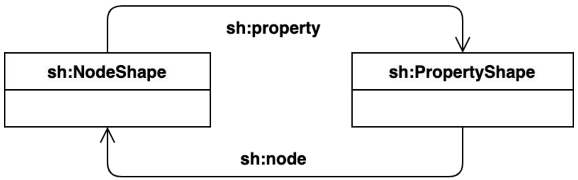
- SHACL constraints express *general purpose tests* to be applied on value nodes
- Internally represented by constraint components (instances of `sh:ConstraintComponent`)
- Components define at least one mandatory and arbitrary optional parameters (`sh:Parameter`)
- Parameters are on defined on SHACL shapes via corresponding RDF properties



- Object diagram of the [`sh:PatternConstraintComponent`]



Shape-Based Constraints



Complex shapes

```
# Songs linked in an album should refer back
:AlbumTrackShape
      a sh:NodeShape ;
      sh:targetClass :Album ;
      sh:property [
            sh:path :track ;
            sh:minCount 1 ;
            sh:node :TrackAlbumShape ;
      ] .
:TrackAlbumShape
      a sh:NodeShape ;
      sh:property [
            sh:path :trackOf ;
            sh:class :Album ;
            sh:minCount 1 ;
      ] .
```

focus nodes

value nodes

- [Shape-based constraints]() specify *complex conditions* combining node and property shapes
- All value nodes must comply with the shape linked via `sh:node` or `sh:property` predicates

Qualified Shape Constraints

sh:PropertyShape
sh:qualifiedValueShape : sh:Shape
sh:qualifiedMinCount : xsd:integer
sh:qualifiedMaxCount : xsd:integer
sh:qualifiedValueShapesDisjoint : xsd:boolean

Qualified constraints

```
# Example from the SHACL specification, hand to have 4 fingers and 1 thumb
ex:HandShape
        a sh:NodeShape ;
        sh:targetClass ex:Hand ;
        sh:property [
                sh:path ex:digit ;
                sh:maxCount 5 ;
        ] ;
        sh:property [
                sh:path ex:digit ;
                sh:qualifiedValueShape [ sh:class ex:Thumb ] ;
                sh:qualifiedValueShapesDisjoint true ;
                sh:qualifiedMinCount 1 ;
                sh:qualifiedMaxCount 1 ;
        ] ;
        sh:property [
                sh:path ex:digit ;
                sh:qualifiedValueShape [ sh:class ex:Finger ] ;
                sh:qualifiedValueShapesDisjoint true ;
                sh:qualifiedMinCount 4 ;
                sh:qualifiedMaxCount 4 ;
        ] .
```

node shape

- Only a subset of value nodes is required to comply with a sh:qualifiedValueShape
- Mandatory cardinality bounds: lower (`sh:qualifiedMinCount`), upper (`sh:qualifiedMaxCount`)
- Values must not conform to any sibling shape when `sh:qualifiedValueShapesDisjoint` is `true`

Closed Shapes

sh:NodeShape
sh:closed : xsd:boolean
sh:ignoredProperties : rdf:List

Closed shape

```
# Any property not covered by a shape is invalid
:ClosedSongShape
      a sh:NodeShape ;
      sh:targetClass :Song ;
      sh:closed true ;
      sh:ignoredProperties (rdf:type) ;
      sh:property
       :NameShape ,
       :WriterShape ,
       :LengthShape ,
       :DescriptionShape ;
.
```

- SHACL shapes are not required to cover all existing properties of the target nodes
- Closed node shapes (`sh:closed = true`) will report any remaining, unsupervised properties
- Supply an optional list of (generic) properties, that are not covered by a shape, to be ignored

Cardinality Constraints

sh:PropertyShape
sh:minCount : xsd:integer
sh:maxCount : xsd:integer

Property count

```
# There should be at least one song per album

:AlbumTrackShape
      a sh:NodeShape ;
      sh:targetClass :Album ;
      sh:property [
            sh:path :track;
            sh:minCount 1 ;
      ] ;
.
```

- Cardinality constraints refer to the expected range of *property* occurrences on a focus node
- `sh:minCount` and `sh:maxCount` are inclusive
- Consider qualified value shapes to restrict (qualify) the value of the counted property

Value Type Constraints / Property Shapes

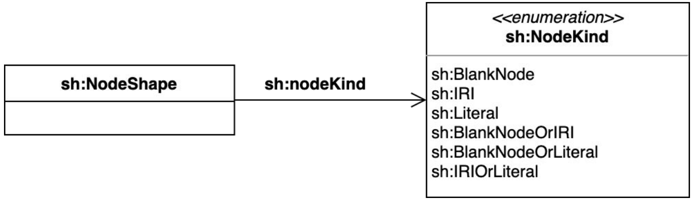
sh:Shape
sh:class : rdfs:Class
sh:datatype : rdfs:Datatype

Property range

```
# The release date of an album must be a
# valid xsd:date (for date-based queries)

:AlbumDateTypeShape
      a sh:NodeShape ;
      sh:targetClass :Album;
      sh:property [
            sh:path :date ;
            sh:datatype xsd:date ;
      ] ;
.
```

- [Value type constraints]() restrict the *type of value nodes* to a data-type or class
- Constraint applicable to property and node shapes (e.g. as part of qualified constraints)
- Class constraint matches given class *and* its superclasses (`rdf:type/rdfs:subClassOf* $class`)
- Multiple values for `sh:class` are interpreted as a conjunction, use `sh:or` for alternatives

Value Type Constraints / Node Shapes



Node kind

```
# An album should use IRIs to refer to
# the producer instead of a literal name

:AlbumProducerKindShape
      a sh:NodeShape ;
      sh:targetClass :Album ;
      sh:property [
            sh:path :producer ;
            sh:node [
                  sh:nodeKind sh:IRI;
            ]
      ] ;
.
```

- [Node kind](#) constraints enforce the generic (RDF) type of value nodes targeted by a node shape
- At most one `sh:nodeKind` constraint may be defined per shape
- The value range is a set of (combined) [RDF term](#) identifiers (IRIs, literals and blank nodes)

Property Pair Constraints (1/2)

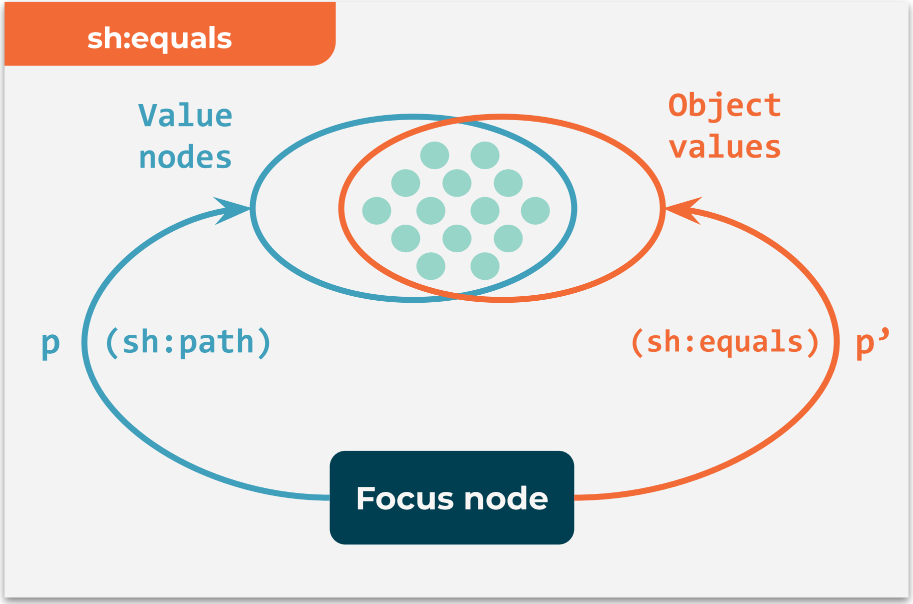
sh:PropertyShape
sh:equals : rdf:Property
sh:disjoint : rdf:Property
sh:lessThan : rdf:Property
sh:lessThanOrEquals : rdf:Property

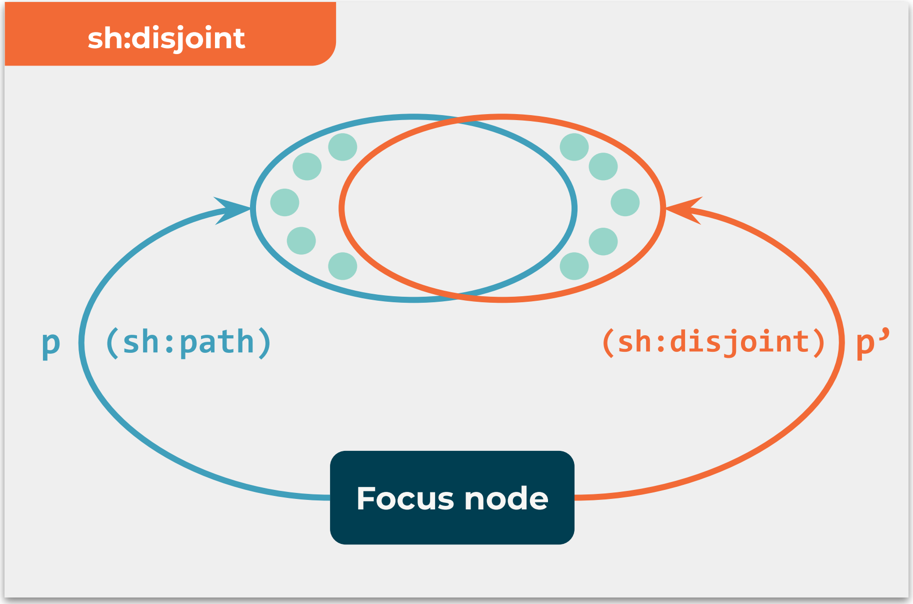
Less than

```
# The end date should follow the start of a tour
:TourDatesShape
      a sh:NodeShape ;
      sh:targetClass :Tour;
      sh:property [
            sh:path :start ;
            sh:datatype xsd:date ;
            sh:lessThan :end ;
      ] ;
      sh:property [
            sh:path :end ;
            sh:datatype xsd:date ;
      ].
```

- Property pair constraints relate *all value nodes* for given path to *all values* of a sibling property
- Depending on the constraint values in both sets are required to either:
  - Overlap (`sh:equals`), be disjoint (`sh:disjoint`) or be less (or equal) w.r.t to the other set

Property Pair Constraints (2/2) / Outline





Value Constraints

sh:PropertyShape
sh:hasValue : RDF term
sh:in : rdf:List

Same value

```
# A rock album should indicate appropriate genre

:RockAlbumShape
      a sh:NodeShape ;
      sh:targetClass :RockAlbum;
      sh:property [
            sh:path :genre ;
            sh:hasValue genre:RockMusic ;
      ] .
```

- Value constraints require the value node to be *equal to* a given value or *included in* a value set
- *At least one* value node must equal to the `sh:hasValue` property
- Alternatively, *each value node* must be a member of the `sh:in` list

Value Range Constraints

sh:PropertyShape
sh:minInclusive : rdfs:Literal
sh:maxInclusive : rdfs:Literal
sh:minExclusive : rdfs:Literal
sh:maxExclusive : rdfs:Literal

Value range

```
# A track should indicate a plausible length
# (i.e., a non-zero duration in seconds)

:SongLengthShape
      a sh:NodeShape ;
      sh:targetClass :Song ;
      sh:property [
            sh:path :length ;
            sh:datatype xsd:integer ;
            sh:minExclusive 0 ;
      ] ;
.
```

- Value range constraints specify the lower and upper *bounds of comparable literal values*
- Supported are, among others, the numeric and date XSD data-types (`xsd:date, xsd:integer`)
- Failures to compare incompatible data-types (`xsd:string` vs. `xsd:date`) result in validation error

String-Based Constraints

sh:PropertyShape
sh:minLength : xsd:integer
sh:maxLength : xsd:integer
sh:pattern : xsd:string
sh:flags : xsd:string
sh:languageIn : rdf:List
sh:uniqueLang : xsd:boolean

Unique language

```
# Each skos:Concept (genre) should have
# a unique label per language

:ConceptUniquePrefLabel
      a sh:NodeShape ;
      sh:targetClass skos:Concept ;
      sh:property [
            sh:path skos:prefLabel ;
            sh:uniqueLang "true"^^xsd:boolean
      ] ;
.
```

- String-based constraints specify various tests on textual value nodes
- `sh:flags` (`"ism"`) optionally modifies the RegEx pattern in `sh:pattern` (`"^(has|is)[A-Z].*"`)
- `sh:languageIn` requires the language tag of literals to match prescribed values (`"en" "de"`)
- `sh:uniqueLang` when set to `true` prohibits multiple literals per focus node and language

Logical Constraints (1/2)

sh:Shape
sh:and : rdf:List
sh:or : rdf:List
sh:xone : rdf:List
sh:not : sh:Shape

Negation

```
# the classes Album and Song are disjoint,
# i.e., do not have any instances in common

:AlbumShape a sh:NodeShape ;
      sh:targetClass :Album ;
      sh:not [ sh:class :Song ]
.
```

- [Logical constraints](#) relate constraints via the *logical operators* ‘and’, ‘or’, ‘not’, ‘exclusive or’
- Depending on the operator either:
  - none (`sh:not`), all (`sh:and`), at least one (`sh:or`) or exactly one (`sh:xone`) constraints apply
- `sh:not` refers to a single (negated) shape, other logical constraints operate on lists of shapes

Logical constraints (2/2)

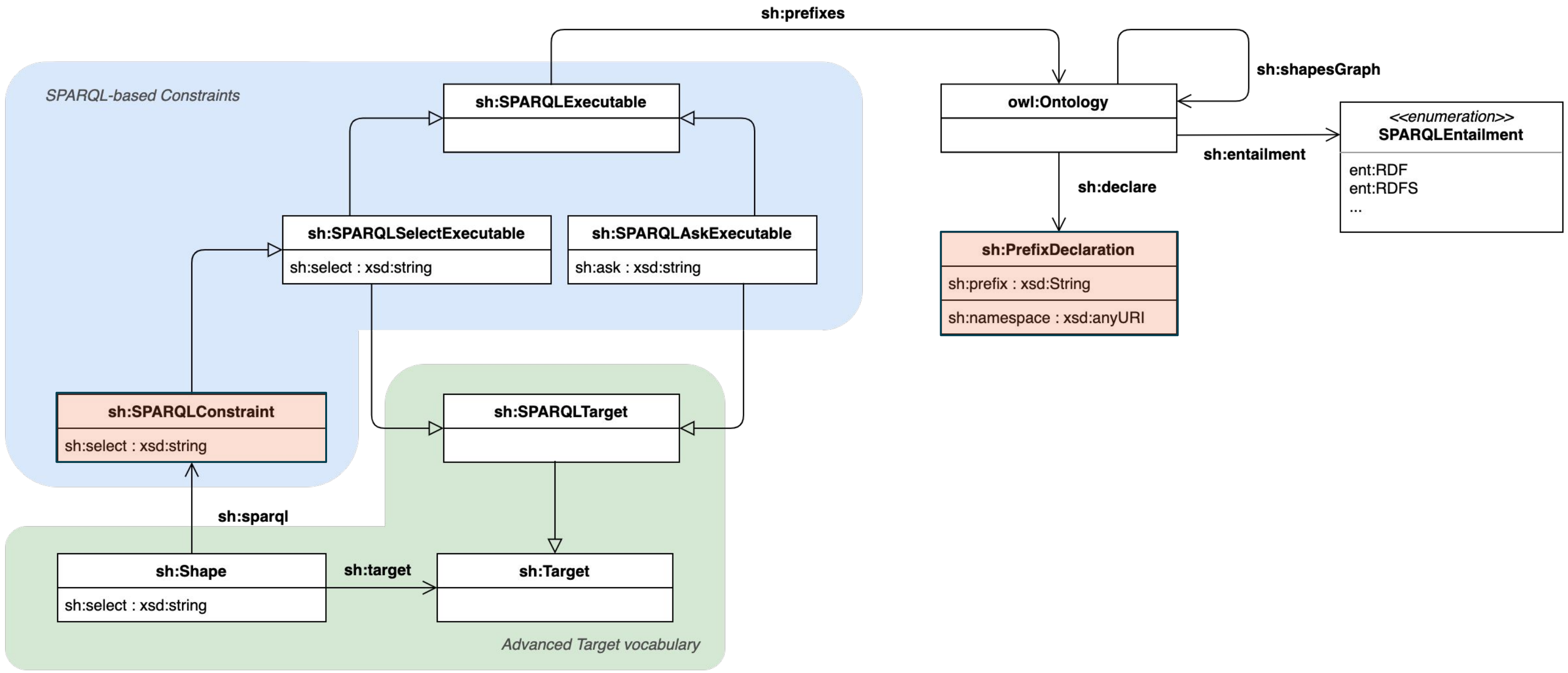
Exclusive OR

```
# An artist record should either contain
# the full name or the first and last name

:SoloArtistNameVariantConstraint
       a sh:NodeShape ;
       sh:targetClass :SoloArtist ;
       sh:xone (
              [
                     sh:property [
                            sh:path ex:fullName ;
                            sh:minCount 1 ;
                     ]
              ]
              [
                     sh:property [
                            sh:path ex:firstName ;
                            sh:minCount 1 ;
                     ] ;
                     sh:property [
                            sh:path ex:lastName ;
                            sh:minCount 1 ;
                     ]
              ]
       ) .
```

Node shape

Property shape

SPARQL Extension / Outline



SPARQL-Based Constraints

- Advanced tests on focus nodes expressed via SPARQL SELECT queries

SPARQL constraint

```
# Warn about albums with a high number of tracks
:AlbumTrackShapeSparql
  a sh:NodeShape ;  # Apply to both, node and property shapes
  sh:targetClass :Album ;
  sh:sparql [
    a sh:SPARQLConstraint ; # optional type statement
    sh:prefixes tut: ; # reference to a prefix declaration
    sh:severity sh:Warning ;
    sh:message "Album with a high number of tracks (25+)" ;
    sh:select """
    # Query should specify dataset (graphs) it operates upon
    SELECT $this (tut:track AS ?path) (COUNT(?track) as ?value)
       WHERE {
         $this tut:track ?track .
       }
    GROUP BY $this ?value
    HAVING (?value > 25)
    """;
] .
```

- Examples: complex graph traversals, filter conditions, or aggregations.
- Variable $this pre-bound to focus node being validated
- Query solutions are used to generate validation results
- Variable ?path mapped to `sh:resultPath` (optional)
- Variable ?value mapped to `sh:value` of the report (optional)

SPARQL-Based Constraints / Prefix Definition

- Namespace prefixes for use within SPARQL queries are declared via `sh:PrefixDeclaration`
- Conventially attached to an `owl:Ontology` instance via the `sh:declare` predicate
- SPARQL constraint refers to the same instance via the `sh:prefixes` predicate

Prefix declaration

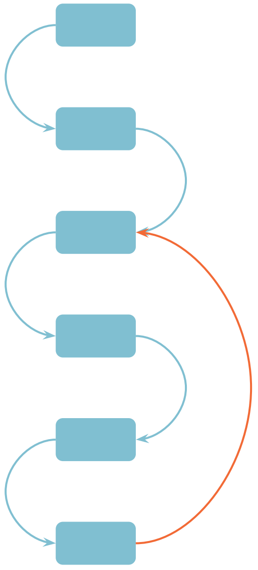
```
@prefix tut: <http://stardog.com/tutorial/> .
@prefix sh:  <http://www.w3.org/ns/shacl#> .

tut:
  a owl:Ontology ;
  sh:declare [
    a sh:PrefixDeclaration ;
    sh:prefix "tut" ; # String
    sh:namespace "http://stardog.com/tutorial/"^^xsd:anyURI ;
  ] .
```

Prefix reference

```
:AlbumTrackShapeSparql
  a sh:NodeShape ;
  sh:targetClass :Album ;
  sh:sparql [
    sh:prefixes tut:
  # ...
  ] .
```

SPARQL-Based Constraints / Example

Concept hierarchy



SPARQL Constraint

```
# Detect concept hierarchy cycles
:ConceptHierarchyCyclesShape
      a sh:NodeShape ;
      sh:targetClass skos:Concept ;
      sh:sparql [
            a sh:SPARQLConstraint ;
            sh:message "Cyclic hierarchical link detected" ;
            sh:select """
             # Inline prefix definition
             prefix skos: <http://www.w3.org/2004/02/skos/core#>
                  select distinct $this where {
                  {
                      $this skos:narrower+ ?narrower .
                      ?narrower skos:narrower+ $this
                  }
                  union
                  {
                      $this skos:broader+ ?broader .
                      ?broader skos:broader+ $this
                  }
            }
                  """;
      ] .
```

Constraint Generation / CLI

- Generate an initial version of SHACL shapes from the schema

```
stardog data model --input owl --output shacl music
```

RDF Schema

```
:Album a rdfs:Class .

:artist a rdf:Property ;
   rdfs:domain :Album ;
   rdfs:range :Artist .

:track a rdf:Property ;
   rdfs:domain :Album ;
   rdfs:range :Song .

:date a rdf:Property ;
   rdfs:domain :Album ;
   rdfs:range xsd:date .
```

Generated shapes

```
:Album a sh:NodeShape , rdfs:Class ;
    sh:property [
        sh:path :artist ;
        sh:class :Artist
    ] , [
        sh:path :track ;
        sh:class :Song
    ] , [
        sh:path :date ;
        sh:datatype xsd:date
    ] .
```

Constraint Operations / CLI

a) Maintain constraints as an external resource (file)

- No need to persist in the database for on-demand validation
- Most recent version supplied via Studio editor or CLI argument

```
stardog icv report music constraints.ttl
```

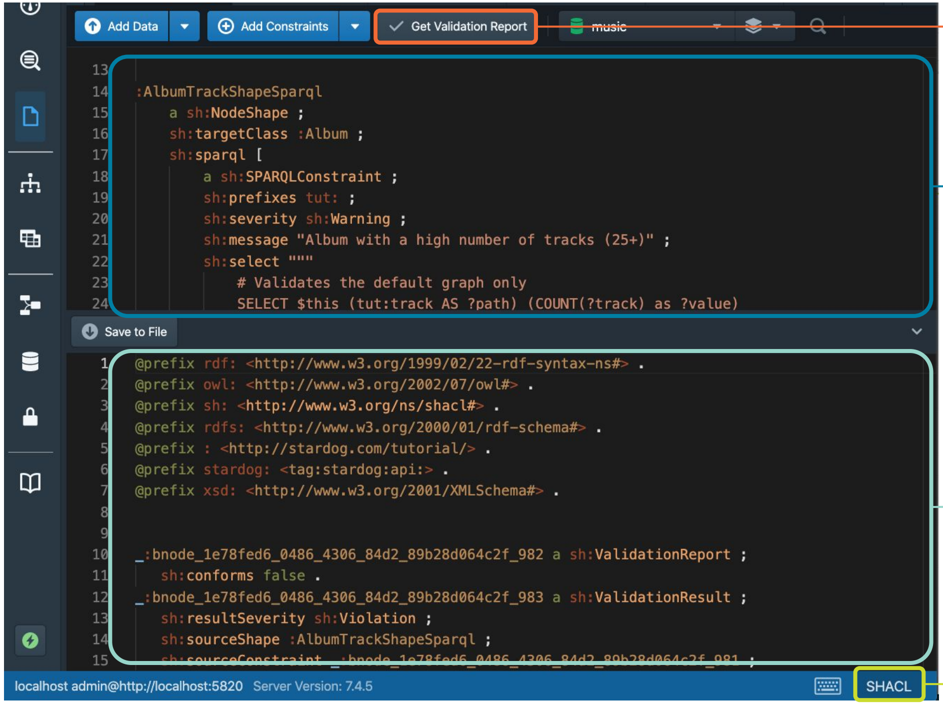
b) Persist the constraints in Stardog (custom named graph) for on-commit validation

- SHACL is RDF so `stardog data add / remove` commands apply

```
stardog data add --named-graph urn:graph:constraints music constraints.ttl
stardog data remove --named-graph urn:graph:constraints music
```

```
stardog icv report music
```

Constraint Usage / Studio



Click *Get Validation Report* to submit content of the editor window as constraints

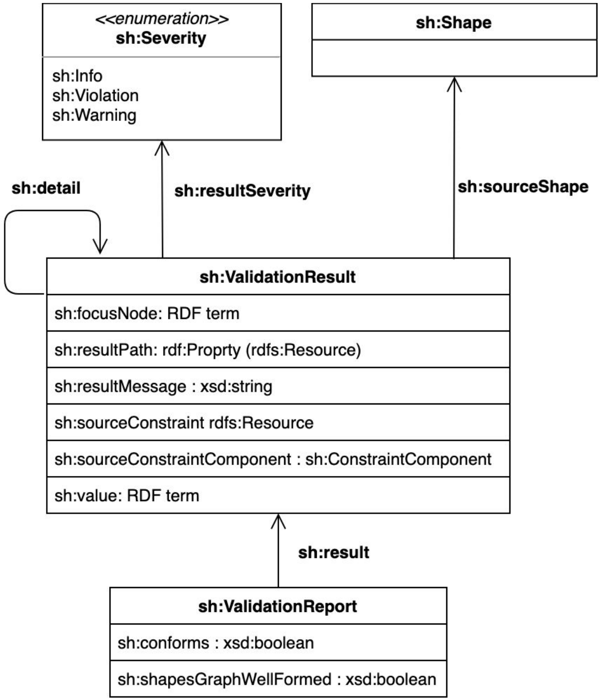
Editor window for developing SHACL shapes

Validation report based on shapes supplied in the editor window (or stored in the DB, when empty)

Select *SHACL* as content type in editor

Evaluate

Validation Report



Conformance report

```
[     a sh:ValidationReport ;
          sh:conforms true ;
] .
```

Issue report

```
_:report a sh:ValidationReport ;
   sh:conforms false .

_:report sh:result _:result1 .

_:result1 a sh:ValidationResult ;
   sh:resultSeverity sh:Violation ;
   sh:sourceShape :ConceptHierarchyCyclesShape ;
   sh:sourceConstraint _:constraint ;
   sh:sourceConstraintComponent sh:SPARQLConstraintComponent
;
   sh:focusNode genre:PopularMusic ;
   sh:value genre:PopularMusic ;
   sh:resultMessage "Cyclic hierarchical link detected" .
```

Report Overview

- `sh:conforms` – `true` if no validation results (regardless of severity) were produced
- `sh:result` - links the report to individual test results (`sh:ValidationResult`)
- `sh:sourceShape` – immediate shape that generate the result (often a blank node)
- `sh:resultSeverity` - (user defined) severity of the result, instance of `sh:Severity` class
- `sh:sourceConstraintComponent` – SHACL constraint component that has been violated
- `sh:focusNode` – node that produced the results i.e., the potentially problematic node
- `sh:value` – identifies what value was reported by the shape
- `sh:resultPath` – identifies how the (incorrect) value is connected to the focus node
- RDF terms linked by above 3 predicates represent the objected triple in data graph:

```
{sh:focusNode} -{sh:resultPath}-> {sh:value}
```

- Consider those for retrieving additional context in report queries

Report Evaluation

- Create, store and query the validation report

```
stardog icv report music > report.ttl
stardog data add --named-graph urn:graph:report music report.ttl
stardog query execute music report_query.rq
```

Report Query

Query stored report

```
select ?shape (count(distinct ?focusNode) as ?count) ?message
where {
    graph <urn:graph:report>
    {
        ?result
            a sh:ValidationResult ;
            sh:resultSeverity sh:Violation ;
            sh:sourceShape ?shape ;
            sh:focusNode ?focusNode ;
            sh:resultMessage ?message ;
    }
}
group by ?shape ?path ?message
order by desc(?count)
```

shape	count	message
:ConceptHierarchyCyclesShape	3	"Cyclic hierarchical link between concepts detected"

Demo

Resources

- Data Quality Vocabulary (DQV), W3C Note
- RDF Data Quality Assessment (presentation)
- Chapter 5, SHACL, in Validating RDF Data (online resource)
- Shapes Constraint Language (SHACL), W3C Recommendation
- SHACL Playground
- Data Validation and SHACL (webinar)
- Improve data quality with SHACL
- ICV Examples

Learning Objectives

Learning Objectives

- Understand the concept of data quality and why it matters
- Asses quality requirements for various kinds of data
- Apply appropriate means to assess quality of data
- Evaluate, communicate and act upon data quality reports
- Operate Stardog to ensure quality of integrated data

Thank you