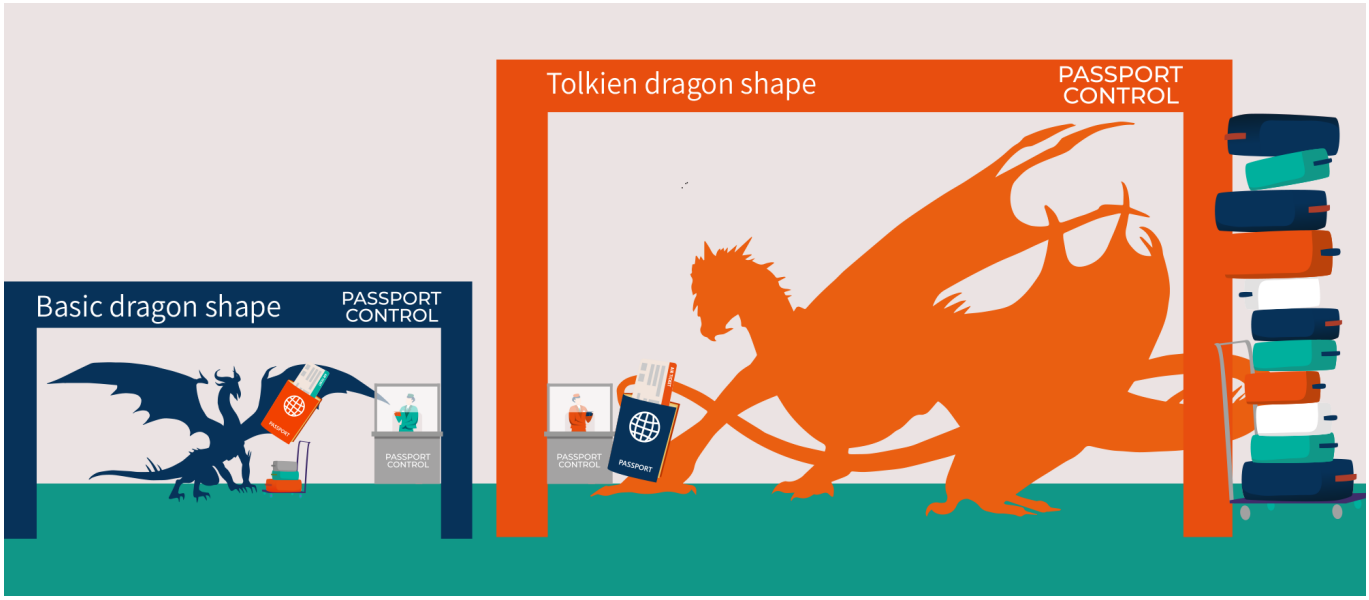



As far as standards go, SHACL is young. Work on it began in 2015 and achieved W3C Recommendation status in mid-2017. This means that implementations haven't solidified yet to the point where you can expect them to behave identically. This can be both a blessing and a curse.

In this third and final post of the series, we will review the advantages of Ontotext GraphDB's support for SHACL.

## Incremental validation — one small step…

This loops back to a topic we touched upon in the second part of our series. All SHACL tools can do bulk validation. However, not many can also do incremental validation. Bulk validation is great when data comes together nicely as one big package. Let's bring out the titular dragon.

```
ex:Smaug a ex:WingedDragon;
             foaf:gender "male" .
```

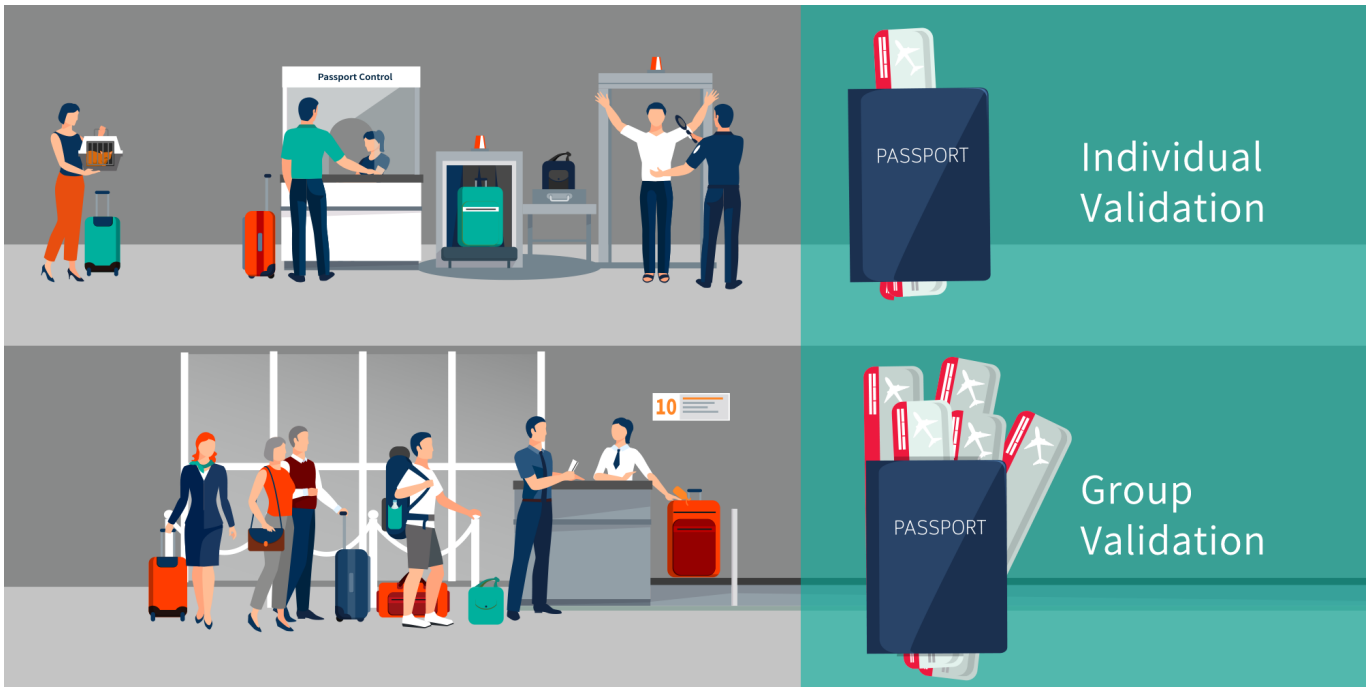
We perform bulk validation. All is well. Now, we add a new triple. `ex:Smaug ex:hasWings false` . So, the winged dragon Smaug doesn't have wings? That doesn't sound right.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

However, the problem is not the incoming triple. The data only becomes problematic when it's joined together with the rest of the dragon data. So, we perform bulk validation again, and an error comes up. However, to do that, we had to perform bulk validation on the whole database. And besides Smaug, we have 1,000 other dragons. That's an issue.

The RDF4J framework, and, by extension, GraphDB, offers a smart approach to this. When an update about Smaug comes up, all potentially relevant data is pulled up from the database into the transaction context. Then, validation happens on the data within that context only. Now the other 1,000 dragons are not a menace to our performance.

This performance-oriented approach to validation needs to be enabled at the repository level. To take advantage of incremental validation, you must use a repository that has SHACL enabled when the repository is initially created to ensure that your data is consistent throughout the repository's lifetime.




Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

# Bulk validation — everything all at once

For a long time, GraphDB had no “out of the box” bulk validation, but it was not impossible. There have always been methods to do it, and more approaches have been added over time. Currently, there are three ways to perform SHACL validation of your data.

## The clear graph approach

The original approach leverages the fact that SHACL shapes are just like any other RDF data. We already mentioned that incremental validation requires that validation is part of the transaction. At the lowest level, to the engine, there’s no difference between “data” and “SHACL shapes” — it’s all triples. This means that when SHACL shapes are initially loaded, as they are “fresh data”, they will trigger validation on the entire database.

The following is a very basic way to do SHACL bulk validation:

1. Enable SHACL to have incremental validation if you want to.
2. When you need to do bulk validation, clear all your graphs that contain SHACL shapes.
3. In a new transaction, reinsert the SHACL shapes. If there are any inconsistencies, the shapes will be rejected and you will receive a violation report.

This is a straightforward approach that is easy to implement and has no significant drawbacks. The disadvantage is that it is not intuitive. Data engineers think about “shapes” and “data” separately and treating shapes as data is an implementation detail. It lets us do bulk validation, but it’s a workaround.

## Low-level validation control from RDF4J

The second approach is to drop down to the RDF4J Java API. Transactions in RDF4J can be controlled at a fine-grained level. This includes a `ShaclSail.TransactionSettings.ValidationApproach` enumerator that contains three options, one of which is bulk.

```
try (RepositoryConnection conn = rep.getConnection()) {
    conn.begin(ShaclSail.TransactionSettings.ValidationApproach.Bulk);
```

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

```
      ....
    conn.commit();
}
```

The other two options for validation are “automatic”, which just does incremental validation and “disabled”, which is self-explanatory.

There are three caveats:

- You have to use the RDF4J Java API. Transaction settings are not exposed to the HTTP API.
- Your repository still needs to have SHACL enabled, configurable only at startup.
- You need to make an empty transaction. Opening and closing a connection should suffice.

The first point is usually the deal breaker. Most data engineers are not Java developers. This approach is not well-suited to rapid prototyping.

## REST API for validation

Our latest approach addresses the issues of the past. It is not a workaround and requires no preconfiguring or low-level access to the API. Starting with GraphDB 10.3 Ontotext provides a straightforward REST API that allows you to validate your data regardless of initial repository configurations. There are two approaches you can take using the HTTP interface:

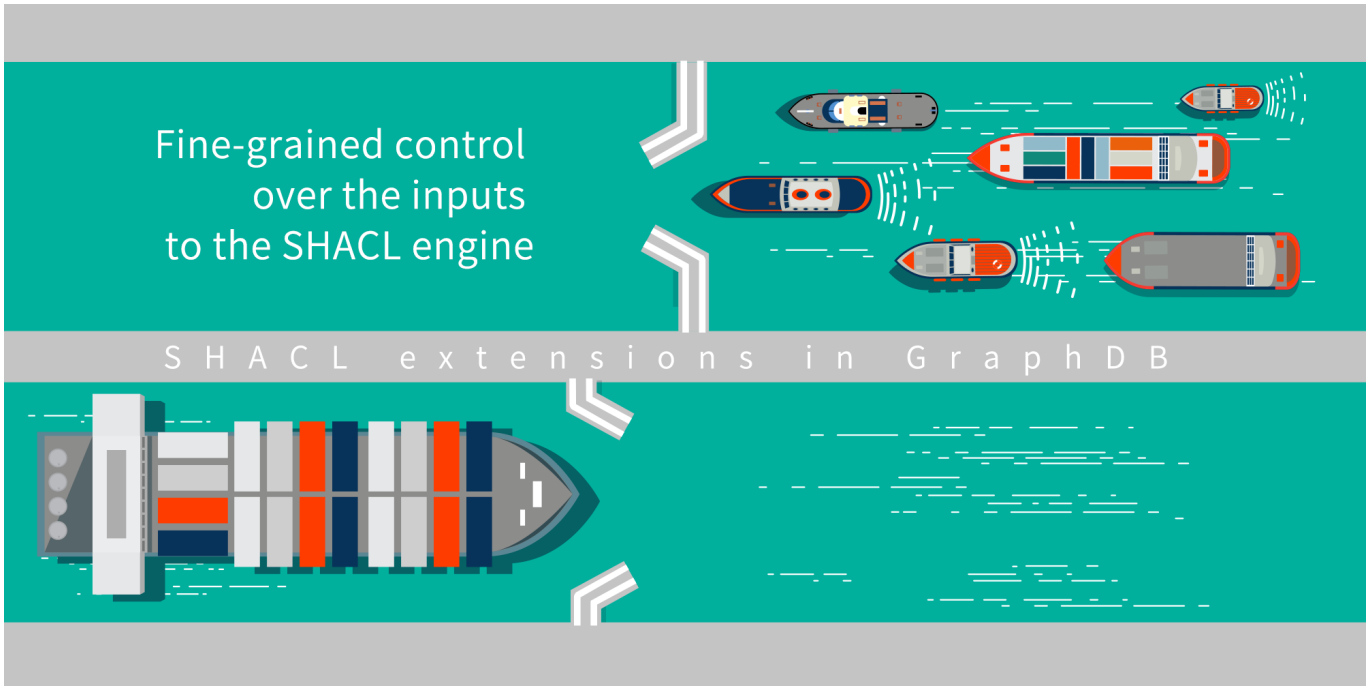
- Validate data from the repository, identified by `repositoryID` with the shapes stored in a repository, identified by `shapesRepositoryID`. `POST: /rest/repositories/{repositoryID}/validate/repository/{shapesRepositoryID}`
- Validate data from the repository, identified by `repositoryID`, with the shapes you send in the request. `POST: /rest/repositories/{repositoryID}/validate/text`

One small caveat: by default, when using incremental validation, all SHACL shapes must be stored within the SPARQL named graph `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph`. There is no such limitation for

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

the REST API.
Keep in mind how named graphs interact with your validation:

- The SHACL shapes graph will validate the union of all graphs. That is, it validates in bulk. This is the same behavior as adding new SHACL shapes to a standard SHACL-enabled repository.
- Unlike incremental validation, by default, SHACL shapes kept inside the default graph of the shapes repository will validate all data in the data repository, exactly like the SHACL shapes graph.
- Starting in GraphDB 10.3.2, you can use `sh:shapesGraph` to define specific pairings between data graphs and shapes graphs. Those pairings must be declared inside the shapes repository, inside the default graph, or in the SHACL shapes named graph




## Controlling the validation scope with named graphs

There's a pattern to how SHACL validation with GraphDB developed, and it happens to be incremental. Ontotext started with the basic assumption for big data — everyone will want to

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

validate everything, and since data is large, it better be incremental, as performance will be poor otherwise.

However, that's not always the case. Many users need to apply different validations on different parts of the dataset.

## Validating multiple graphs with different shapes

Let's return to our dragon friend. Say that we have two "ways" to talk about a dragon because there are two different taxonomies about them. To reason about dragons as a whole, you decide to store them inside the same repository. So, you have two different graphs, with different data about dragons.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.example.org/vocabulary#> .
@prefix rdfs: <https://www.w3.org/2000/01/rdf-schema#> .

ex:BasicDragonTaxonomy {
ex:Smaug a ex:WingedDragon ;
        rdfs:label "Smaug" ;
                foaf:gender "male" .
}
ex:Tolkien {
        ex:Smaug a ex:WingedDragon ;
                rdfs:label "Smaug", "Smaug the Golden", "Smaug the
Impenetrable", "Smaug the Terrible", "The Dragon Dread"; "Trāgu"
}
```

We have two data sources: a basic dragon taxonomy and another based on Tolkien's Legendarium. In the basic taxonomy, each dragon must have exactly one label, whereas in the Tolkien ontology, each dragon may have multiple labels. However, they both use the type `ex:WingedDragon`. We don't want to define different types for the two Smaugs, as requests are made across both graphs. However, this means that naive validation using `sh:targetClass` wouldn't work.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

The way to handle this is to build two separate shapes to use in the two different dragon validation cases — `ex:TolkienDragonShape` and `ex:BasicDragonShape`. Both can use `ex:WingedDragon` as a `sh:targetClass`. Then, you place them in two different graphs, `ex:BasicDragonsShapesGraphs` and `ex:TolkienShapesGraph`. The next step is to link the data graph to the shapes graph: `ex:TolkienDragonShape sh:shapesGraph ex:TolkienShapesGraph`. You place this mapping inside the default SHACL shapes graph.

This technique can be especially useful in data integration projects where you are combining related, potentially overlapping data from multiple sources. Remember to set up your shapes graph in a repository that has been configured from the beginning to support SHACL, as described in our documentation.

## Validating a union of graphs

Up till now, we have talked about data that is added incrementally. However, all of this data is added to the same graph. We are breaking up the set of triples being added, but not their destinations. When `sh:shapesGraph` was specified, it was not considered that the added data could be broken up between different named graphs.

Coming back to our previous example, Smaug has a gender defined in the basic dragon taxonomy graph, but not in the Tolkien graph. We may want to validate the two graphs together. We can do this with the default SHACL graph, `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph`. However, it validates all graphs. So, if we have graphs for other dragons that don't need instances of `foaf:gender`, we are in trouble.

That is why GraphDB 10.3 introduced a way to validate a union of some specific data graphs:

```
ex:DragonGraphsValidationLink
  a rsx:DataAndShapesGraphLink;
  rsx:shapesGraph ex:JointShapesGraph ;
  rsx:dataGraph ex:BasicDragonTaxonomy, ex:Tolkien;
```

This is an extension specific to RDF4J and GraphDB, hence the `rsx` namespace. It stands for `http://rdf4j.org/shacl-extensions#`. The use of an

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

`rsx:DataAndShapesGraphLink` specifies that when ingestion happens, the SHACL engine will look up the `ex:JointShapesGraph` and treat the union of the two data graphs as if they were joined together. This means that we can store the gender of a certain dragon in one graph, and that dragon's other other data in another.

## Going above and beyond — custom SHACL extensions in GraphDB

GraphDB and the underlying RDF4J framework offer some enhancements to the SHACL standard that can ease the life of data engineers.

Some of these, like the Eclipse RDF4J SHACL extensions, are specific to RDF4J. We already discussed the union of data graphs above. But while it's the most significant customization, it's certainly not the only one.

### RSX targets

In the previous posts in this series, we discussed targeting and SHACL-SPARQL, or the ability to specify validation targets with arbitrary SPARQL queries. SHACL-SPARQL is flexible and gives you a lot of power, but often at a price. It executes a SPARQL query, which can get very complex. To this end, before RDF4J implemented SPARQL targets, there were the RSX targets.

The most generic use case of RSX targets is "custom types". Imagine that, for whatever reason, the instances of the dragons are not grouped by `rdf:type`, but rather, by a different predicate. While we still have `rdf:type`, when we are arguing about dragons, we are usually grouping them using `ex:species ex:Dragon`.

There's no easy way to specify this with standard SHACL shape. Every type of standard targeting relies on a specific predicate — a custom one for target subjects or objects, or on the type predicate for target class. A combination of a custom predicate and a custom value is not an option. This is where RSX comes in:

```
ex:DragonShape a sh:NodeShape;
rsx:targetShape [sh:path ex:species; sh:hasValue ex:Dragon].
```

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

This specifies that ex:DragonShape applies to any resource that has the property-value pair shown in the square brackets specified as the value of `rsx:targetShape`.

### RSX reports

When you perform SHACL validation with GraphDB, you will see `rsx:shapesGraph` and `rsx:dataGraph` values included in your validation report to help you determine where that violation came from. Note that if you are using the `rsx:DataAndShapesGraphLink` construct we mentioned earlier, the data graph will contain all graphs that are specified in the link, as currently there's no logic to differentiate where exactly the data is stored.

### Inclusive lists

A common issue with RDF is inclusivity. We alluded to this earlier, when describing a range constraint whose minimum values were exclusive and inclusive. The `sh:hasValue` property specifies that this subject-predicate pair must have **at least** the value specified by `sh:hasValue`. "Has value" is inclusive. On the other hand, `sh:in` states that the value **must be** from the list provided. "In" is exclusive. We need a way to have an inclusive list.

For example, we can say that a dragon can have the type "winged dragon" or "wingless dragon". We want to include data about Puff the Magic Dragon, which is a winged dragon, but is also classified as `"ex:ImaginaryDragon"`. Since Puff has the type "winged dragon" already, that should be valid. And we can use `dash:hasValueIn (ex:WingedDragon ex:WinglessDragon)` to that end. Here, the prefix `dash:` stands for the following IRI: `http://datashapes.org/dash#`.

## Missed connections — constraints that are not supported by GraphDB

The list of constraints that GraphDB doesn't support has steadily shrunk ever since we introduced SHACL validations. At the moment, all use cases can be accomplished using SHACL-SPARQL. Some of them would have been more concise if using the following (unsupported) constraints:

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

- Pairwise comparisons.
- Wildcard paths. Zero-or-more, one-or-more, zero-or-one path. The RDF4J SHACL engine currently works only on explicit paths for performance reasons.
- sh:xone. In essence, the XOR condition. This can be achieved by chaining AND, NOT, and OR.
- Qualified shapes. Qualified shapes are a bit hard to understand when reading shapes that use them and are both more readable and more concise when written as SHACL-SPARQL.
- sh:closed and related properties. RDF4J works on a strong open-world assumption, as the opposite would be computationally prohibitive and can be achieved with SHACL-SPARQL with no performance difference.
- Non-validating characteristics. While these provide no instructions to a SHACL engine, the use of non-validating characteristics such as `sh:name` and `sh:description` can add metadata to your shapes that make them easier to maintain as they scale up.

If you need any of these features and SHACL-SPARQL doesn't let you do it, or isn't readable enough, let us know. We want our implementation of the SHACL standard to help you get the most out of your knowledge graphs.

## Conclusion

Over the last three posts, we have outlined the issue, given you the tools to address it, and shown you what's the output of applying those tools. Now, you should have a solid foundation for putting this knowledge into practice, with specific caveats for how to handle both incremental and bulk validation and fine-grained control over the inputs to the SHACL engine. The next step is to get out there and challenge your data quality dragons.
Get GraphDB and try your hand with our examples for SHACL validation from the documentation!

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

Ontotext's GraphDB
Give it a try today!
Download Now

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com