RDF Constraint Checking

Peter M. Fischer, Georg Lausen, Alexander Schätzle
Univ. of Freiburg, Faculty of Engineering, 79110 Freiburg, Germany
{peter.fischer,lausen,schaetzle}@informatik.uni-freiburg.de

Michael Schmidt
metaphacts GmbH, Industriestraße 39c, 69190 Walldorf, Germany
ms@metaphacts.com

ABSTRACT

Linked Open Data (LOD) sources on the Web are increasingly becoming a mainstream method to publish and consume data. For real-life applications, mechanisms to describe the structure of the data and to provide guarantees are needed, as recently emphasized by the W3C in its Data Shape Working Group. Using such mechanisms, data providers will be able to validate their data, assuring that it is structured in a way expected by data consumers. In turn, data consumers can design and optimize their applications to match the data format to be processed.

In this paper, we present several crucial aspects of RDD, our language for expressing RDF constraints. We introduce the formal semantics and describe how RDD constraints can be translated into SPARQL for constraint checking. Based on our fully working validator, we evaluate the feasibility and efficiency of this checking process using two popular, state-of-the-art RDF triple stores. The results indicate that even a naive implementation of RDD based on SPARQL 1.0 will incur only a moderate overhead on the RDF loading process, yet some constraint types contribute an outsize share and scale poorly. Incorporating several preliminary optimizations, some of them based on SPARQL 1.1, we provide insights on how to overcome these limitations.

1. INTRODUCTION

Linked Open Data (LOD) sources on the Web using RDF are increasingly becoming popular. As a consequence mechanisms are needed that can be used to validate RDF datasets. Using such means data providers can validate their data to assure that they are providing information structured in a way as expected by data consumers, and other way round, data consumers can validate their interfaces against the data to be processed. As RDF is a graph based data model, validation not only should refer to single triples, but also graph patterns must be considered. The RDF Validation Workshop [16] states a gap between the current standards offering and the industry needs for validation of RDF data. More recently, in continuation of the workshop, a W3C working group is in the process of being established [4]. As major issues, this working group will address the definition and publication of topology and value constraints of RDF graphs, validation of such constraints and optimization of SPARQL queries based on it. To tackle these issues, in continuation of our previous work [9], we have developed a constraint language RDD (RDF Data Descriptions) [13, 14] that captures a broad range of constraints including keys, cardinalities, subclass, path and subproperty restrictions, making it easy to implement RDD checkers and clearing the way for semantic query optimization.

The intention of an RDD is similar to SPIN [5], IBM's Resource Shapes [11], and Stardog ICV [6], where among these systems Stardog ICV seems to be the one an RDD is mostly related to. In Stardog ICV [6], constraints are stated using OWL and considered relative to a certain inference machinery whose type may range from no inferencing, RDFS- to OWL-inferencing. In contrast, RDD is a language using a compact special-purpose syntax designed for only expressing constraints independent of a specific inference machinery. This makes RDD in particular applicable for RDF under ground semantics, which is a common scenario in the Linked Data context.

Just recently two other interesting validation methods have been proposed. Shape Expressions [10] semantically act as a type inference system that can derive types for given nodes in an RDF graph. Its functionality resembles schema languages for XML, in particular RelaxNG. A test-driven approach for validation is suggested in [8]. Test cases may be manually derived or automatically from existing RDFS/OWL specifications. These approaches are similar to RDDs in the sense that the final validation can be performed using SPARQL query expressions. However, while Shape expressions are based on regular expressions, RDDs incorporate relational constraints to RDF and therefore support the mapping of relational databases to RDF using R2RML [2], for example. While [8] is based on templates which are instantiated and afterwards executed to determine the degree to which corresponding constraints are fulfilled, RDD constraints are checked for fulfillment and in case they are violated, for efficiency reasons, only a small number of counterexamples is listed. As a distinctive feature, different to both discussed approaches, RDDs are based on a human readable language in a similar vein to relational databases. Finally, the topic of our current paper is measuring the cost of various constraint patterns in particular for data sets of varying size to get more information about scalability and starting points for optimization. Neither [10] nor [8] elaborate on these aspects. However, we emphasize that the constraint types considered in these works are similiar to a large degree to the ones imposed by RDDs, such that many of the results on efficient constraint checking via SPARQL proposed in this work carry over to these approaches as well.

In the current paper we elaborate on checking constraints described using RDDs. We first describe an RDD constraint checker which maps a given RDD into a set of SPARQL 1.0 queries. The main contribution of the current paper is a comprehensive experimental evaluation of the checking process. We analyze the overhead induced by the various constraint types and demonstrate the effectiveness of different kinds of optimization, where some optimizations are based on SPARQL 1.1 language features. We show that constraints proposed in RDD can be validated with moderate cost compared to the initial loading of a respective RDF graph.

The paper is organized as follows. In Section 2 we present RDD, the *RDF Data Description* language [13], which we use to define constraints over RDF graphs. Section 3, to have a formal basis, presents a first-order logic (FOL) semantics of RDDs. Section 4 describes our *RDD Checker* implementation of RDD based on a mapping from FOL to SPARQL and Section 5 presents the findings of our *RDD Checker* evaluation. In Section 6 we then discuss several ideas on how to optimize the RDD to SPARQL mapping and illustrate their potential impact on the efficiency of the checking process in Section 7. Section 8 concludes the paper and gives an outlook on future work.

2. RDF DATA DESCRIPTION (RDD)

The RDF data description language (RDD)[13, 14] allows to express the following kinds of constraints:

- A RangeTypeConstraint indicates that the property *prop* points to either a URI, BlankNode, Resource, or a (possibly typed) Literal.
- A Min/MaxConstraint indicates that the property *prop* occurs at least or at most a number of times, respectively.
- A Domain/RangeConstraint indicates a guaranteed *domain* or *range* for subject and objects associated with property *prop*, respectively.
- A PathConstraint indicates that the value of property *prop* can as well be reached by following a given *path* of properties.
- A SubPropertyConstraint indicates that for every triple using property *subProp*, there is also an identical triple using property *prop*.
- A Partiality/TotalityConstraint expresses that property *prop* occurs at most or exactly one time, respectively.

All these constraints may occur in *unqualified* form, i.e. hold for a property independently of its context, or in *qualified* form, i.e. hold for a property only when the property is used in combination with a subject of a given (fixed) class. While the above mentioned constraints all refer to properties, the following class-specific constraints exist:

- A SingletonConstraint indicates that a class has exactly one instance.
- A KeyConstraint indicates the properties uniquely identifying the entities of a class in all possible class instances.
- A SubclassConstraint allows for the inheritance of constraints along class hierarchies.

```
PREFIX ex: <http://www.example.com#>
...
CWA CLASSES {
 OWA CLASS foaf:Person SUBCLASS ex:Student {
   KEY rdfs:label : LITERAL
   MAX(2) foaf:mbox : LITERAL
   TOTAL foaf:age : LITERAL(xsd:integer)
   RANGE(foaf:Person) foaf:knows : IRI }

 OWA CLASS ex:Student {
   KEY ex:matricNr : LITERAL(xsd:integer)
   MIN(1), RANGE(ex:Course) ex:course : RESOURCE
   PATH(ex:course/ex:givenBy),
     RANGE(foaf:Person) ex:taughtBy : IRI }

 OWA CLASS ex:Course { ... }
}

OWA PROPERTIES {
  PARTIAL foaf:nick : LITERAL
  foaf:knows SUBPROPERTY ex:taughtBy
}
```

Figure 1: Example RDF Data Description

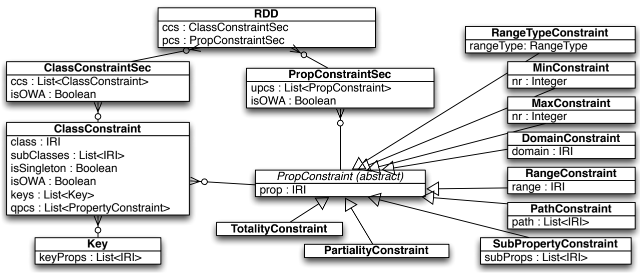


Figure 2: Structural Overview of the RDD Language

Figure 1 provides an example RDD demonstrating the usage of the various constraint types. At top-level, an RDDs consist of two main sections:

(i) A class constraint section (keyword CLASSES) defining *qualified property constraints* and *class-specific* constraints.

(ii) A global property constraint section (keyword PROPERTIES) defining *unqualified property constraints.*

The class constraint section contains a list of `CLASS` definitions, where each may contain a set of (qualified) property constraints. The RDD in Fig. 1 defines a class `foaf:Person`, where property `rdfs:label` as `KEY` uniquely identifies a person, which has at most two mailboxes (`foaf:mbox`) associated, exactly one age (`foaf:age`) and `foaf:knows` always points to objects of type `foaf:Person`. It also defines a class `ex:Student` as a subclass of `foaf:Person`. With RDDs focusing on instance-level constraints, this is not a subclass relation in the sense of RDFS, but guarantees that every instance of class `ex:Student` satisfies the same constraints as defined for `foaf:Person`. Additionally, every student must be uniquely identified by a property `ex:matricNr` as `KEY` and is enrolled in at least one course (`ex:course` always pointing to objects of type `ex:Course`). Moreover, for every property `ex:taughtBy` there is a path along the properties (edges) `ex:course` followed by `ex:givenBy` pointing to the same entity of type `foaf:Person`. The OWA specifications coming with the class definitions for `foaf:Person`, `ex:Student`, and `ex:Course` say that these classes are interpreted under open world assumption, i.e. an instance may carry properties other than those listed in the body. Differing in its semantics, the CWA constraint associated with the top-level CLASSES section implies that there are no classes other than those specified in its body (i.e., `foaf:Person`, `ex:Student`, and `ex:Course`); keyword OWA associated with the PROPERTIES section indicates that no such constraint is imposed at property level. This example illustrates that RDDs allow to specify a mix of open and close world semantics at different levels. Finally, the property constraint section defines that every person or student may have one nickname (`foaf:nick`) and if it has a `ex:taughtBy` property pointing to x, it also has property `foaf:knows` pointing to x.

Figure 2 visualizes the syntactical structure and concepts of RDD in a UML-style notation. Boxes denote concepts, arrowed lines sub-concepts relationships and the remaining line type a uses-relationship. A more detailed description of the RDD syntax can be found in [14].

3. RDD SEMANTICS

We first like to introduce the basic RDF notation [3]. Let U be a set of URI references, B a set of blank nodes and L a set of literals. A triple $t := (s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an *RDF triple*; s is called subject, p property and o object. A finite set of triples is called an *RDF graph*.

Let an RDF graph G and an RDD r be given. Following [14] we shall now demonstrate how a corresponding set *cs* of FOL constraints can be derived. We stick to the following two notational conventions. Variables are distinguished from other terms by using \$ as a prefix. Moreover, formulas of the kind $\forall \$x_1, \ldots, \$x_n \phi$ are abbreviated to ϕ thereby assuming that all free variables in ϕ are globally $\forall$-quantified. We shall use four unary relations IRI, $BNode$, $Resource$, and $Literal$ containing all IRIs, blank nodes, resources (i.e. IRIs and blank nodes), and literals that appear in any position of any triple in G, respectively. An RDF graph G is modeled as a ternary relation $G(s, p, o)$ representing the triples of the respective graph in the obvious way. To improve readability, we define the following two shortcuts:

$$\begin{aligned}&allDist(\$x_1, \ldots, \$x_n) := \textstyle\bigwedge_{1 \le i < j \le n} \$x_i \neq \$x_j, \text{ and}\\ &someEq(\$x_1, \ldots, \$x_n) := \textstyle\bigvee_{1 \le i < j \le n} \$x_i = \$x_j,\end{aligned}$$

enforcing that the n variables passed as parameters are all pairwise distinct (*allDist*) or some of them are equal (*someEq*). In the following, we present the constraint types that are imposed through the constructs in RDDs.

Unqualified CWA$_P$.

Whenever CWA PROPERTIES is specified in r, the unqualified property constraint cwa_P is used to restrict the usage of properties to only those which are mentioned in the RDD's property section. Let $p_1, \ldots, p_n$ be the properties mentioned in the unqualified property constraint section. Then cwa_P is defined as follows:

$$cwa_P : \; G(\$s, \$p, \$o) \rightarrow \$p = p_1 \vee \cdots \vee \$p = p_n$$

Unqualified Property Constraints.

The *unqualified range type* restriction enforces the range type of a property, according to one of the keywords **IRI**, **BNODE**, **RES(OURCE)**, **LIT(ERAL)** or some given type R specified in the RDD specification. This gives rise to the following kinds of constraints:

$$\begin{aligned}&range(p, \textbf{IRI}) : \; G(\$s, p, \$o) \rightarrow IRI(\$o)\\ &range(p, \textbf{BNODE}) : \; G(\$s, p, \$o) \rightarrow BNode(\$o)\\ &range(p, \textbf{RES}) : \; G(\$s, p, \$o) \rightarrow Resource(\$o)\\ &range(p, \textbf{LIT}) : \; G(\$s, p, \$o) \rightarrow Literal(\$o)\\ &range(p, R) : \; G(\$s, p, \$o) \rightarrow G(\$o, \textbf{rdf:type}, R)\end{aligned}$$

The remaining unqualified property constraints are *domain, min, max, total, subprop, part* and defined as follows:

$$\begin{aligned}&domain(p, D) : \; G(\$s, p, \$o) \rightarrow G(\$s, \textbf{rdf:type}, D)\\ &min(p, n), n \ge 1 : \; Resource(\$s) \rightarrow \exists \$o_1, \ldots \$o_n\\ &\quad (G(\$s, p, \$o_1) \wedge \cdots \wedge G(\$s, p, \$o_n) \wedge allDist(\$o_1, \ldots, \$o_n))\\ &max(p, n), n \ge 1 : \; G(\$s, p, \$o_1) \wedge \cdots \wedge G(\$s, p, \$o_{n+1})\\ &\quad \rightarrow someEq(\$o_1, \ldots, \$o_{n+1})\\ &total(p) : \; min(p, 1) \wedge max(p, 1)\\ &part(p) : \; max(p, 1)\\ &subprop(p, p_s) : \; G(\$s, p_s, \$o) \rightarrow G(\$s, p, \$o)\\ &path(p, q_1, \ldots, q_n), n \ge 1 : G(\$s, p, \$o) \rightarrow \exists \$o_1, \ldots, \$o_{n-1}\\ &\quad (G(\$s, q_1, \$o_1) \wedge \cdots \wedge G(\$o_{n-1}, q_n, \$o))\end{aligned}$$

Note that *total* and *part* both define functional restriction of a property p; using $total(p)$ the property p must be defined for all subjects, whereas for $part(p)$ there may exist subjects in G where p is not defined.

Qualified CWA$_P$ and Qualified Property Constraints.

The qualified versions of constraints are different from the unqualified only in that their application is restricted to a corresponding class C, i.e. conjugating $G(\$s, \textbf{rdf:type}, C)$ to the prerequisites of the corresponding constraint. For example, a range restriction qualified by class C is of the following form:

$$\begin{aligned}&range(p, C, R) :\\ &\quad G(\$s, \textbf{rdf:type}, C) \wedge G(\$s, p, \$o) \rightarrow G(\$o, \textbf{rdf:type}, R)\end{aligned}$$

Class Constraints.

Finally, as part of the constraint section of a class C, class constraints *key* and *singleton* can be specified[1]:

$$\begin{aligned}&key(C, p_1, \ldots, p_n, R_1, \ldots, R_n) :\\ &\quad range(p_1, C, R_1) \wedge \ldots \wedge range(p_n, C, R_n) \wedge\\ &\quad total(p_1, C) \wedge \ldots \wedge total(p_n, C) \wedge\\ &\quad (\; G(\$s_1, \textbf{rdf:type}, C) \wedge G(\$s_2, \textbf{rdf:type}, C) \;\wedge\end{aligned}$$

[1] *subclass* constraints are handled as described in [14] and need not be considered in the context of the current paper.

$G(\$s_1, p_1, \$o_1) \wedge \cdots \wedge G(\$s_1, p_n, \$o_n) \wedge$
$G(\$s_2, p_1, \$o_1) \wedge \cdots \wedge G(\$s_2, p_n, \$o_n) \rightarrow \$s_1 = \$_2$)
$singleton(C)$:
$\exists \$s(G(\$s, \textbf{rdf:type}, C)) \wedge$
($G(\$s_1, \textbf{rdf:type}, C) \wedge G(\$s_2, \textbf{rdf:type}, C) \rightarrow \$s_1 = \$s_2$)

Moreover, whenever CWA CLASSES is specified in r, the constraint CWA$_C$ can be used to restrict the usage of classes to only those which are mentioned in the RDD. Let $c_1, \ldots, c_n$ be the classes mentioned in the class section. Then cwa_C is the constraint:

$$cwa_C: \ G(\$s, \textbf{rdf:type}, \$c) \rightarrow \$c = c_1 \vee \cdots \vee \$c = c_n$$

Finally, we define the consistency of an RDF graph w.r.t. a given RDD as follows:

Definition *Let G be an RDF graph and let cs be the set of first-order logic constraints defined by a corresponding RDD r. RDF graph G is consistent with respect to cs if and only if for all constraints $c \in cs$ it holds that c is valid in G, i.e. $G \models c$, respectively, $G \models cs$.*

It is well-known that for a fixed set of FOL constraints consistency of a given arbitrary RDF data set can be decided in polynomial time. In particular, in the following sections we will discuss an appropriate mapping into SPARQL.

4. RDD CHECKER

We have implemented the aforementioned decomposition of an RDD into the corresponding set of FOL constraints which can be used for further investigation, e.g. it may serve as input to an FOL reasoner. Furthermore, we have also implemented a mapping from the generated FOL constraints to corresponding SPARQL 1.0 queries which can be used to check the consistency of an RDF dataset w.r.t. a given RDD. As the resulting queries are compliant to the SPARQL 1.0 spec, they can be executed with any SPARQL 1.0 query engine. Our implementation comes with built-in bindings for *Sesame* [1], such that a given RDD can be verified against any RDF dataset out of the box. In addition, the checker can be pointed at arbitrary SPARQL endpoints. The binaries and source code of our *RDD Checker* (implemented in Java) are available for public download[2].

In the following we give an exemplary depiction of how we can use SPARQL to check whether an RDD constraint in FOL holds on a given RDF document. For more details on the connection between SPARQL and FOL, the interested reader may be referred to, e.g., [9, 15].

Consider the totality constraint (`TOTAL foaf:age`) on class `foaf:Person` taken from Figure 1. This constraint assures that every entity in an RDF dataset of type `foaf:Person` needs to have exactly one `foaf:age` property defined. As defined in Section 3, a $total(p)$ constraint is a combination of $min(p, 1)$ and $max(p, 1)$ requiring that every person has at least and at most one age, respectively. This gives rise to the following two (qualified) FOL rules:

$min(\textbf{foaf:age}, \textbf{foaf:Person}, 1)$:
$G(\$s, \textbf{rdf:type}, \textbf{foaf:Person}) \rightarrow \exists \$o_1(G(\$s, \textbf{foaf:age}, \$o_1)$
$max(\textbf{foaf:age}, \textbf{foaf:Person}, 1)$:
$G(\$s, \textbf{rdf:type}, \textbf{foaf:Person}) \wedge$
$G(\$s, \textbf{foaf:age}, \$o_1) \wedge G(\$s, \textbf{foaf:age}, \$o_2) \rightarrow \$o_1 = \$o_2$

[2]`http://dbis.informatik.uni-freiburg.de/forschung/projekte/rdd/`

We do not use a pattern-based translation approach where there is a query pattern for each constraint type but instead define it along the structure of the corresponding FOL rules. The concept of our mapping from an RDD constraint c to SPARQL is to define a query for every FOL rule imposed by c that retrieves those entities from an RDF graph G violating the rule. If no such entities exist for every rule of c, then $G \models c$. The generic idea is to define a *graph pattern* matching the body of the rule, and use a *filter* expression to select only those entities matching the graph pattern that do not fulfill the head of the rule. As a single witness already leads to violation, we can limit the number of results such that a query engine does not have to compute all results, if supported. Our *RDD Checker* implementation uses a customizable default value of three. The corresponding SPARQL queries for the given *min* and *max* rules from above are listed in Figure 3.

MIN(1):
```
SELECT ?s {
  ?s rdf:type foaf:Person
  OPTIONAL { ?s foaf:age ?o1 }
  FILTER (!BOUND(?o1))
} LIMIT 3
```

MAX(1):
```
SELECT ?s {
  ?s rdf:type foaf:Person .
  ?s foaf:age ?o1 . ?s foaf:age ?o2
  FILTER (!(?o1=?o2))
} LIMIT 3
```

Figure 3: SPARQL queries for constraint *total*(foaf:age) on class foaf:Person

The atoms of a rule can be represented by *triple patterns* (i.e. triples with variables) in the SPARQL query, e.g. $G(\$s, \textbf{foaf:age}, \$o_1)$ can be mapped to `?s foaf:age ?o1`. The concatenation of atoms in the body of a rule can then be equivalently represented by a set of `AND(.)` connected triple patterns forming a so-called *basic graph pattern*. The following filter expression then defines the negation of the rule's head. An equality-generating head can be simply represented by a `FILTER` where we negate (denoted by `!`) the conditions of the head (see *max* constraint). In the case of an existentially quantified head we use a combination of `OPTIONAL` and `!BOUND` as SPARQL 1.0 does not have a natural support for negation (see *min* constraint). This way, the `FILTER` only accepts those bindings for variable `?s` where `OPTIONAL` did not find any binding for `?o1`, hence those entities (persons) that do not have an a `foaf:age` property. This construct is equivalent to the explicit `FILTER NOT EXISTS` functionality added in SPARQL 1.1.

This is a rather direct mapping and we can apply this strategy to all RDD constraints to generate queries that adhere to the SPARQL 1.0 spec. Our *RDD Checker* implementation currently uses this mapping such that every available SPARQL 1.0 query engine can be used to check the validity of RDD constraints on an arbitrary RDF dataset.

Though this one-to-one mapping gives us a correct and complete realization of RDD, it is not an optimal solution in terms of efficiency. As the example already illustrates, an RDD constraint can consist of more than one FOL rule in general and hence lead to more than one SPARQL query for verification. As many queries have to iterate over the whole dataset or the same parts of it, this naturally raises the issue of efficiency of the checking process and possible optimizations to reduce the number of iterations over the whole dataset. In the following section, we first present the findings of our *RDD Checker* evaluation based on the mapping to SPARQL as described in this section. In Section 6 we then discuss several ideas on how to optimize the representation of RDD constraints in SPARQL and illustrate their potential impact on the efficiency of the checking process in Section 7.

5. RDD CHECKER EVALUATION

The goal of our evaluation is to determine how validation compares with common database operations and how individual constraints contribute to it. From these findings, we can then derive potential opportunities for optimizations.

Setup.

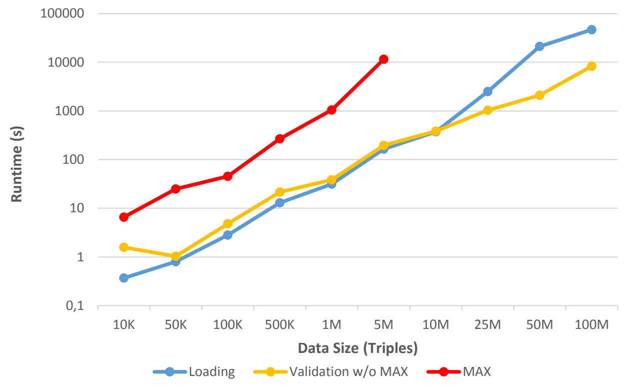
We perform our evaluation on top of Sesame [1] 2.7.12 and Virtuoso Open Source 7.1.0, which are commonly used, highly compliant and feature-complete SPARQL implementations. Sesame supports a range of storage backends, out of which we picked the Native Java Disk Storage without Schema Reasoning, as it supports almost arbitrary data sizes and does not impose any additional cost. The experiments were performed on a system with a single Xeon X5667 with 4 physical cores (8 hyperthreaded) at 3.06 GHz, 32 GB RAM and 12 TB of disk storage (LSI MegaRaid with 4x4TB LGST 7200 rpm SATA Deskstar disks in a RAID 5 configuration), running Ubuntu Linux 12.04 LTS. The Java for Sesame heap size was set to 28 GB, sufficient to keep even the largest dataset we tested in memory. Virtuoso is a native program, so no such tuning was needed.

Data and Constraints.

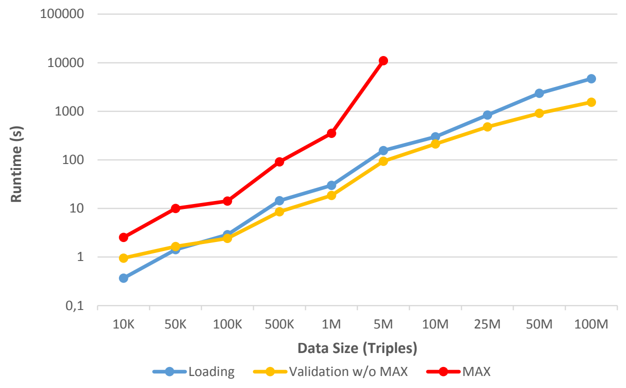
We studied the modeling of constraints and the cost of validating them on the SP2 Benchmark [12]. SP2Bench contains a well-defined and rather structured RDF dataset with documented constraints and data distributions, which models a publication database similar to DBLP. In contrast to most RDF datasets, it includes a generator that can generate a wide of range of scalings while maintaining distributions and correlations. For the scope of this paper, we evaluated datasets ranging from 10K to 100M triples, corresponding to 1.1 MB to 11 GB when stored as N3 files. This range covers a majority of typical, real-life RDF datasets.

Our RDD file takes a class-centric approach (similar to relational or object-oriented modelling), describing 12 classes with mixed OWA/CWA settings, *key*, *partial*, *total* and *range* definitions. In total it contains 215 constraints that were mined from the SP2Bench dataset, and hold over all scalings tested. The RDD is available for download from our project website[3]. All RDDs were translated and directly validated using our RDD checker implementation, yielding 251 SPARQL queries, since the translation of e.g., *total* or *key* constraints needs several SPARQL queries for a single constraint (c.f. Section 4). These queries are set to use the SELECT form of SPARQL with a LIMIT of 3 as to produce a small number of witnesses of the violations. We study lifting this limit in Section 7.

[3]`http://dbis.informatik.uni-freiburg.de/forschung/projekte/rdd/`



(a) Sesame



(b) Virtuoso

Figure 4: Cost of RDD Validation compared to loading

Validation vs. Loading.

In our first experiment, we compare the cost of validating our RDD file with the cost of loading data. We consider such bulk validation a common application in order to publish stable or slowly changing data. Figure 4 shows this comparison over the entire range of data sizes we analyzed for both triple stores. Sesame and Virtuoso show the same behaviour, with just overall higher performance for Virtuoso. The results show that the cost and scaling validation needs to broken down in two, very distinct sets: Without the *max* constraints, validation scales well, even better than loading. At lower scales, validation is actually held back by the effort of invoking 251 individual queries, which in turn mostly need to access the full dataset each, highlighting significant optimization potential. The *max* constraints have much higher cost and scale much worse. Beyond a scale of 5m triples, many individual *max* queries take longer than 90 minutes, which we had set as a timeout (corresponding to the load time of the largest data set). This high cost and bad scaling is caused by the need to express violations of a $max(n)$ constraint by a $n + 1$-way join (see Section 6.1), pointing out a massive inefficiency.

Individual Constraints.

In our second experiment we further investigated the impact of individual constraint classes, giving us insights into possible optimizations and design guidelines. The second most expensive clause is CWA for classes, since it needs to visit all triples belonging to a specific class and check if their predicates belong to these in the class definition. Next, *key* stands out of the remainder because it needs to employ a join on object (to find same key value for different instances) which is not well supported. *min* would suffer from the same issues as *max*, but the RDD file does not contain any *min* constraints with a high threshold.

We also investigated the effect of scoping on the validation, gradually moving constraints that are shared over almost all classes or can be shared with certain relaxations (e.g., a higher *max* value) to the global properties section. Clearly, these constraints now need to be tested over a larger set of data, but the number of tests will be smaller (one test per property, not one test per property and class) and the test queries themselves will be simpler (avoiding a join on the class type). In the first step, we moved *partial* and *range* constraints, if possible, from the classes section to the global properties section as they are non-conflicting. This change reduced the number of constraints by around 50 percent and yielded a runtime saving of around 25-30 percent. In the second step, we consolidated the *max* constraints of the same property in various classes into a single global property. Since not all classes had the same $max(n)$ value for the same property, we always chose the maximum n, thus weakening the precision of checking. The number of *max* constraints goes down from 10 to 2, but we mostly eliminate those with a small threshold. As a result, the savings are rather limited, yielding only 2 to 5 percent. The corresponding RDDs are also available for download on the project website.

6. OPTIMIZATIONS

It is fairly obvious that the one-to-one mapping of FOL rules representing RDD constraints to SPARQL queries, as described in Section 4, leaves a lot of leeway for optimizations in various directions. Since these optimization rely on deeper understanding of the constraint semantics, SPARQL optimizers cannot detect them. First, there is a potential for *intra-query* optimization, i.e. the individual SPARQL query. Second, *intra-constraint* optimization can reduce the number of SPARQL queries required for checking an individual constraint. And third, *inter-constraint* optimization may give the chance to check several constraints at once in a single query (e.g. several *max* constraints for different properties), also reducing the total number of queries.

In the following we give some insights on intra-query and intra-constraint optimization that we have identified in our *RDD Checker* evaluation (see Section 5). A study of inter-constraint optimization is left for future work.

6.1 Intra-Query Optimization

If we look at the meaning of RDD constraints, many of them are intended to restrict the number of occurrences of specific properties in one way or another, e.g. *min*, *max*, *part*, *total*. This kind of restriction naturally leads to grouping and counting the elements of a group, which is not provided by SPARQL 1.0 directly. Instead, it can be realized by a series of joins as described in the following.

Consider the maximum constraint (`MAX(2) foaf:mbox`) on class `foaf:Person` taken from Figure 1, assuring that every person has at most two mailboxes. This gives rise to the following (qualified) FOL rule:

$$max(\textbf{foaf:mbox}, \textbf{foaf:Person}, 2):$$
$$G(\$s, \textbf{rdf:type}, \textbf{foaf:Person}) \wedge G(\$s, \textbf{foaf:mbox}, \$o_1) \wedge$$
$$G(\$s, \textbf{foaf:mbox}, \$o_2) \wedge G(\$s, \textbf{foaf:mbox}, \$o_3)$$
$$\rightarrow \$o_1 = \$o_2 \vee \$o_1 = \$o_3 \vee \$o_2 = \$o_3$$

If we look at the corresponding mapping to SPARQL 1.0 in Figure 5, following the strategy described in Section 4, we can see that it results in a graph pattern consisting of four triple patterns. To retrieve the result of this pattern, a query engine typically has to compute three joins between subsets of the data. In the following filter expression we then have to check whether any two variables are bound to the same entity. If not, there exists a person with at least three mailboxes violating the constraint. It is obvious that the complexity of the query increases with the specified maximum. In general, a $max(p, C, n)$ query following this strategy contains $n + 1$ joins and $\binom{n+1}{2}$ filter conditions.

SPARQL 1.0:
```
SELECT ?s {
  ?s rdf:type foaf:Person .
  ?s foaf:mbox ?o1 . ?s foaf:mbox ?o2 . ?s foaf:mbox ?o3
  FILTER (!(?o1=?o2 || ?o1=?o3 || ?o2=?o3))
} LIMIT 3
```

SPARQL 1.1:
```
SELECT ?s {
  ?s rdf:type foaf:Person . ?s foaf:mbox ?o1
} GROUP BY ?s HAVING (COUNT(?o1) > 2)
LIMIT 3
```

Figure 5: Intra-query optimization for constraint max(foaf:mbox, 2) on class foaf:Person

One of the functionalities introduced in SPARQL 1.1 is the support for groupings and aggregations similar to those in SQL. Using these features we can simplify the corresponding SPARQL query as also illustrated in Figure 5, reducing the number of joins to only a single one. In fact, the query structure in SPARQL 1.1 is independent from the specified maximum as it only affects the counting threshold. Assuming that query engines (which are typically based on simple algebraic rewritings) are not capable of performing such complex optimizations, one may expect the query in SPARQL 1.1 to be much more efficient than in SPARQL 1.0 for larger maximum values. This assumption could be clearly confirmed in our experiments (see Section 7). A very similar optimization is also possible for *min* constraints.

6.2 Intra-Constraint Optimization

Following the mapping from FOL rules to SPARQL queries described in Section 4, some RDD constraints generate more than one SPARQL query for verification. This raises the issue if we can combine some of these to reduce the total number of queries. Here, we focus on the combination of queries for an individual constraint, whereas generally it might also be possible to combine queries of different constraints.

As an example, consider again the totality constraint illustrated in Section 4, $total(\textbf{foaf:age}, \textbf{foaf:Person})$, and the corresponding queries listed in Figure 3. Limited to the functionality of SPARQL 1.0 we cannot usefully combine both queries as they are structurally different from each other. The only way would be to use a `UNION` of both patterns but it is very likely that query engines will not be able to combine them and just compute both patterns the same way as for different queries which would not reduce the computation cost at all.

But, as already illustrated in Section 6.1, with SPARQL 1.1 we can write *min* and *max* queries using groupings and aggregations. In this way, both queries can be easily combined as they exhibit a very similar structure. Figure 6 shows the combined query in SPARQL 1.1 that retrieves all persons (i.e. entities of type `foaf:Person`) where the number of occurrences of property `foaf:age` is not equal to one. If such a person exists, the *total* constraint is violated.

Combine MIN(1) and MAX(1) using SPARQL 1.1:

```
SELECT ?s {
  ?s rdf:type foaf:Person
  OPTIONAL { ?s foaf:age ?o1 }
} GROUP BY ?s HAVING (COUNT(?o1) != 1)
LIMIT 3
```

Figure 6: Intra-constraint optimization for constraint *total*(foaf:age) on class foaf:Person

The structural difference to the *max* query as illustrated in Section 6.1 is that we have to use an `OPTIONAL` clause for the triple pattern matching the constraint property (here `foaf:age`). This is required because the query must also retrieve those persons that have no specified age property as this is a violation of the *min*(1) requirement and thus also a violation of the *total* constraint. If we would use a simple basic graph pattern (as we can do it for a *max* constraint) these persons would not be part of the query result.

In the following experiments (Section 7) we demonstrate that an individual rewriting of both queries, *min*(1) and *max*(1), to use SPARQL 1.1 does not give a performance benefit compared to the representation in SPARQL 1.0 (the performance is actually even worse). However, the combination of both queries into a single one leads to a performance improvement between 30 and 50 percent.

7. OPTIMIZATION EXPERIMENTS

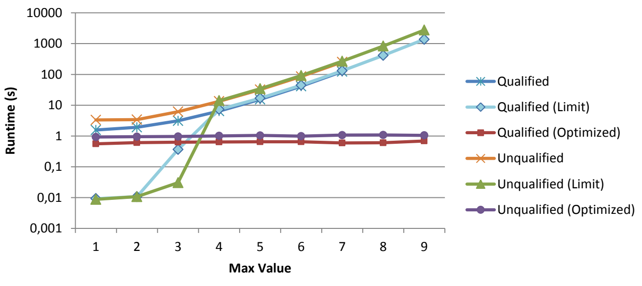
We have done some experiments on the impact of the aforementioned optimizations on the efficiency of the checking process, especially for *max* constraints as they have proven to be by far the most expensive constraint type in our evaluation (cf. Section 5).

Intra-Query Optimizations.

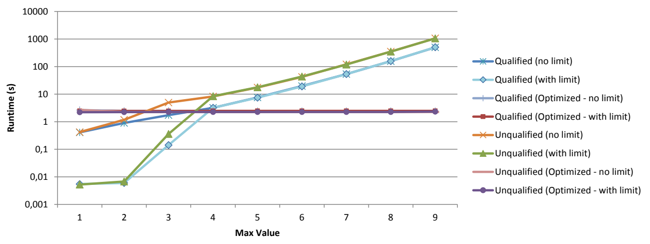
We begin our study with optimizations on individual queries. Our main focus is on *max*, given its cost and the potential benfits it can draw from SPARQL 1.1 features. Furthermore, *max* provides insights into *min*, *total*, *partial* and *key* constraints, which can be expressed at least partially as variants of *max*. For this purpose, we tested the optimized queries explained in Section 6.1.

We compare the join-based approach of *max* against the group-based approach (possible only in SPARQL 1.1), both in qualified (i.e, within a class definition) and unqualified (i.e, as a global property) variants. Furthermore, we use variants without the LIMIT clause to determine the cost of generating witnesses for all violations. Figure 7 show the results of this comparison when varying the *max*(n) threshold between 1 and 9, using a dataset with 1M triples. The overall results are the same for Sesame and Virtuoso, but there are subtle differences.

Without a LIMIT clause, the join based approach shows exponential growth, roughly tripling the cost when increas-



(a) Sesame



(b) Virtuoso

Figure 7: Cost of Max Constraints for 1M triples

ing the threshold by one. In contrast, the group based approach (Optimized) has a constant cost, since the the cost of creating groups does not depend on the threshold value, which in turn can be checked in constant time. In all these cases, the unqualified variant takes about twice as much time than the qualified variant, given the larger set of candidates to consider. For Sesame, the group-based variant always outperforms the join-based variant without limits, while for Virtuoso this is only true for threshold values greater than 4, as joins are faster, but grouping slower.

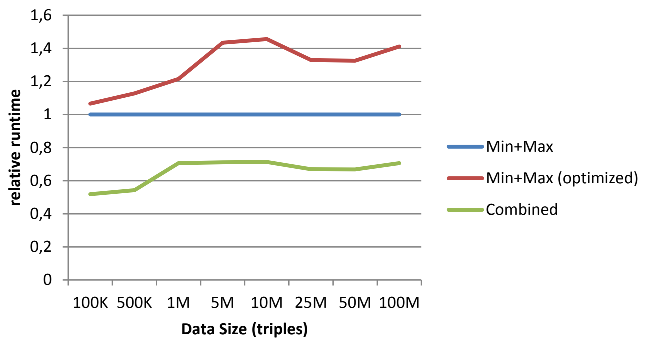
When we consider LIMIT clauses, we gain a number of insights on the intricacies of optimizing queries for validation: As long as the constraint is violated, the join-based approach now clearly outperforms the group-based approach, which is not at all affected by the LIMIT clause. Grouping always has to be performed over the whole dataset, while the optimizers of both systems can perform an early stop on joins, similar to the optimizations possible in SQL [7]. When there are no violations and thus no results, the entire dataset needs to be considered, explaining why at larger thresholds the benefit of LIMIT disappears. We performed this analysis on different dataset sizes, yielding the same overall results.

We also evaluated additional triple storage schemes (or indexes), but we did not determine any speedup: Nearly all queries use subject joins and predicate selections, which fits well with the default storage of both systems.

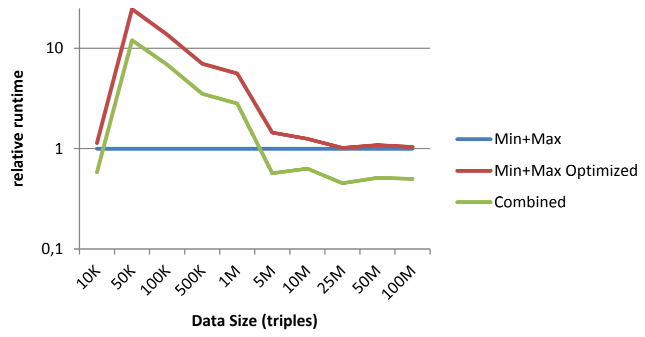
Overall, grouping is a well suited strategy if it is unclear if the constraint holds. Joins with limits work well if the threshold is small and a constraint is expected to not hold.

Intra-Constraint and Inter-Constraint Optimizations.

In our last experiment, we provide a first insight into optimization spanning multiple queries and constraints, enabling us to reduce the number of times the data needs to be accessed. The grouping-based optimization lends itself well for composition, as the same aggregated values can be checked against multiple constraints. We investigate the tradeoffs by following the example in Section 6.2, comparing the runtime



(a) Sesame



(b) Virtuoso

Figure 8: Query Combination Optimization

of the parts of a *total* constraint ($max(1)$ and $min(1)$) in their optimized and non-optimized form against a combined query. As we show in Figure 8, the runtime of the combined eventually falls below the sum of the runtimes of the individual queries. Furthermore, it shows that grouping comes at a cost: For Sesame, the difference is moderate, since the existential check needed for $min(1)$ can be performed faster than a grouping, for Virtuoso the cost of grouping is prohibitive for small scales.

8. CONCLUSION

In this paper, we presented the methodology as well a working system to validate an expressive RDF constraint language using standard SPARQL queries in a "bulk" fashion. Using pure SPARQL is not only conceptually desirable, but also allows the validation of such constraints without having to modify the often loosely coupled and heterogeneous RDF storage systems present in the Linked Open Data environment. The results of our evaluation on state-of-the-art SPARQL databases and a well-established RDF dataset show that such a SPARQL-based validation is feasible with acceptable cost, matching typical loading times. We did, however, identify certain classes of constraints which are expensive to validate using SPARQL 1.0. In turn, we investigated several direction on how to overcome this challenge. Within the scope of individual queries, using a number of SPARQL 1.1 features improves scalability. In the near future, we plan to cross-validate our results on different datasets (e.g., DBPedia, Linked Sensor Data or Bio2RDF) and other RDF constraint languages like RDF Shapes.

Our current work opens up several avenues of further research: Considering that currently several 100s of queries need to be run in order to validate a single RDD and each of these queries has to touch the full dataset, sharing as many validation steps as possible seems to be very promising. On the language side it is currently not clear if the expressiveness of SPARQL 1.1 is sufficient for this purpose, in particular with the flexibility and composability of GROUP BY. On a more conceptual side, we want to understand how far this combination can go and if we can determine a lower limit. Such a limit ties also into an evaluation of the expressive power and cost of some of the competing proposals (such as RDF shapes) in order to identify a "sweet spot" of expressive power and validation cost over these proposals.

Acknowledgements.

The implementation of the RDD prototype was supported by Deutsche Forschungsgemeinschaft grant LA 598/7-1.

9. REFERENCES

[1] OpenRDF Sesame. `http://www.openrdf.org/`.

[2] R2RML: RDB to RDF Mapping Language. `http://www.w3.org/TR/r2rml/`.

[3] Rdf 1.1 semantics. `http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`.

[4] Rdf data shapes working group charter. `http://www.w3.org/2014/data-shapes/charter`.

[5] RDF Specification Overview (W3C). `http://www.w3.org/Submission/2011/SUBM-spin-overview-20110222/`.

[6] Stardog. `http://Stardog.com/`.

[7] Michael J. Carey and Donald Kossmann. Reducing the Braking Distance of an SQL Query Engine. In *Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB*, pages 158–169, 1998.

[8] Dimitris Kontokostas et al. Test-driven Evaluation of Linked Data Quality. In *Proceedings of the 23rd International World Wide Web Conference, WWW*, pages 747–758, 2014.

[9] Georg Lausen, Michael Meier, and Michael Schmidt. SPARQLing Constraints for RDF. In *Proceedings of the 11th International Conference on Extending Database Technology, EDBT*, pages 499–509, 2008.

[10] Eric Prud'hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. Shape Expressions: an RDF Validation and Transformation Language. In *SEM '14: Proceedings of the 10th International Conference on Semantic Systems*, pages 32–40, 2014.

[11] Arthur Ryman, Arnaud Le Hors, and Steve Speicher. OSLC Resource Shape: A Language for Defining Constraints on Linked Data. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, LDOW*, 2013.

[12] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2 Bench: A SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering, ICDE*, pages 222–233, 2009.

[13] Michael Schmidt and Georg Lausen. Pleasantly Consuming Linked Data with RDF Data Descriptions. In *Proceedings of the Fourth International Workshop on Consuming Linked Data, COLD*, 2013.

[14] Michael Schmidt and Georg Lausen. Pleasantly Consuming Linked Data with RDF Data Descriptions. `http://arxiv.org/abs/1307.3419`, 2013.

[15] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory, ICDT*, pages 4–33, 2010.

[16] W3C. RDF Validation Workshop, Practical Assurances for Quality RDF Data. `http://www.w3.org/2012/12/rdf-val/`, 2013.