Aidan Hogan and Macelo Arenas and Alejandro Mallea and Axel Polleres

Everything you always wanted to know about blank nodes (but were afraid to ask)

Everything You Always Wanted to Know About Blank Nodes [*]

Aidan Hogan [a], Marcelo Arenas [b], Alejandro Mallea [b], Axel Polleres [c],

[a] *Department of Computer Science, Universidad de Chile*
[b] *Department of Computer Science, Pontificia Universidad Católica de Chile*
[c] *Vienna University of Economics and Business (WU), Welthandelsplatz 1, 1020 Vienna, Austria*

Abstract

In this paper we thoroughly cover the issue of blank nodes, which have been defined in RDF as 'existential variables'. We first introduce the theoretical precedent for existential blank nodes from first order logic and incomplete information in database theory. We then cover the different (and sometimes incompatible) treatment of blank nodes across the W3C stack of RDF-related standards. We present an empirical survey of the blank nodes present in a large sample of RDF data published on the Web (the BTC–2012 dataset), where we find that 25.7% of unique RDF terms are blank nodes, that 44.9% of documents and 66.2% of domains featured use of at least one blank node, and that aside from one Linked Data domain whose RDF data contains many "blank node cycles", the vast majority of blank nodes form tree structures that are efficient to compute simple entailment over. With respect to the RDF-merge of the full data, we show that 6.1% of blank-nodes are redundant under simple entailment. The vast majority of non-lean cases are isomorphisms resulting from multiple blank nodes with no discriminating information being given within an RDF document or documents being duplicated in multiple Web locations. Although simple entailment is NP-complete and leanness-checking is coNP-complete, in computing this latter result, we demonstrate that in practice, real-world RDF graphs are sufficiently "rich" in ground information for problematic cases to be avoided by non-naive algorithms.

Key words: blank nodes, rdf, simple entailment, leanness, skolemisation, semantic web, linked data

1 Introduction

Although the adoption of RDF [45] has broadened on the Web in recent years [11], one of its core features—blank nodes—has been sometimes misunderstood, sometimes misinterpreted, and sometimes ignored by implementers, other standards, and the broader Semantic Web community. This lack of consistency between the standard and its actual use calls for investigation: are the semantics and the current definition of blank nodes appropriate for the needs of the Web community?

The standard semantics for blank nodes interprets them as existential variables [35], denoting the existence of some unnamed resource. These semantics make even *"simple" entailment checking*—entailment without further well-defined vocabularies—intractable [35]. RDF and RDFS entailment are based on simple entailment, and are thus also intractable due to blank nodes [35].

However, in the documentation for the RDF standard (*e.g.*, RDF/XML [9], RDF Primer [48]), the existentiality of blank nodes is not directly treated; ambiguous phrasing such as "blank node identifiers" is used, and examples for blank nodes focus on rep-

[*] But Were Afraid to Ask

Email addresses: ahogan@dcc.uchile.cl (Aidan Hogan), marenas@ing.puc.cl (Marcelo Arenas), aemallea@ing.puc.cl (Alejandro Mallea), axel.polleres@wu.ac.at (Axel Polleres).

resenting resources that do not have a natural URI. Furthermore, the standards built upon RDF sometimes have different treatment and requirements for blank nodes. As we will see, standards and tools are often, to varying degrees, ambivalent to the existential semantics of blank nodes, where, *e.g.*, the standard query language SPARQL can return different results for two graphs considered equivalent by the RDF semantics [4] and takes seemingly contradictory positions on whether or not (named) graphs can share blank nodes.

Being part of the RDF specification, blank nodes are now a core aspect of Semantic Web technology: they are featured in several W3C standards, a wide range of tools, and hundreds of datasets across the Web, but not always with the same meaning (or at least, with the same intent). Dealing with the issue of blank nodes is thus not only important and timely, but also inherently complex and potentially costly: before weighing up alternatives for blank-nodes, their interpretation and adoption—across legacy standards, tool, and published data—must be considered.

Given the complexity of the situation currently surrounding blank nodes—which spans several standards, several perspectives, a plethora of tools and hundreds of RDF publishers—our goal in this paper is to bring clarity to this thorny issue by covering all aspects in one article. We provide a comprehensive overview of the current situation surrounding blank nodes, from theoretical background to standardisation. We also provide novel techniques and results for analysing the blank nodes published in real-world Linked Data. Our outline is as follows:

- **§2** We provide some formal preliminaries relating to RDF and blank nodes.
- **§3** We discuss blank nodes from a theoretical perspective, relating them to background concepts from logic and database theory and, in so doing, we recapitulate some core worst-case complexity results. We further discuss Skolemisation in the context of RDF.
- **§4** We look at how tasks such as simple entailment and leanness checking can be performed in practice, where we discuss why worst-case complexity results are rarely encountered and remark on how SPARQL (the standard RDF query language) can be used for such tasks.
- **§5** We then survey how blank-nodes are treated in the Semantic Web standards, how they are interpreted, what features rely on them, and remark on trends of adoption.
- **§6** We look at the role of blank nodes in publishing, their prevalence of use in real-world Linked Data, and what blank node morphologies exist in the wild.
- **§7** We give a detailed analysis of the prevalence of blank nodes that are made redundant by simple entailment, designing methods to efficiently identify non-leanness in RDF graphs and discussing the results for a large sample of real-world Linked Data.
- **§8** Finally, in light of the needs of the various stakeholders already introduced, we discuss some alternatives for handling blank nodes.

This paper extends a previously published conference paper [47]. Herein, we provide extended discussion throughout, we update our empirical analysis for a more recent dataset, and we provide detailed techniques and results relating to classifying blank nodes as lean or non-lean.

Running Example. Throughout this paper, we use the RDF graph given in Figure 1 to illustrate our discussion. This graph states that the tennis player `:Federer` won the `:FrenchOpen` in 2009; it also states that he won `:Wimbledon` where one such win was in 2003. [1]

2 Preliminaries

We begin by introducing the abstract representation [31, 55] of the formal RDF model [35, 45] used in our theoretical discussion. We also introduce the semantics of RDF graphs containing blank nodes.

2.1 The RDF data model

We assume the existence of pairwise disjoint infinite sets **U** (URIs), **L** (literals) and **B** (blank nodes), where we write **UB** for the union of **U** and **B**, and similarly for other combinations. [2] An RDF triple is

[1] Throughout, we will omit the prefix declarations in SPARQL queries for brevity. The default prefix is used for example URIs. Other standard prefixes can be checked at the service http://prefix.cc. All URLs mentioned in this paper were retrieved at the time of writing: 2013/11/10.

[2] We generally stick to the term "URI" instead of "IRI" (supported by RDF 1.1) to follow conventions familiar from the literature. For the purposes of this paper, URIs and IRIs can be considered interchangeable.

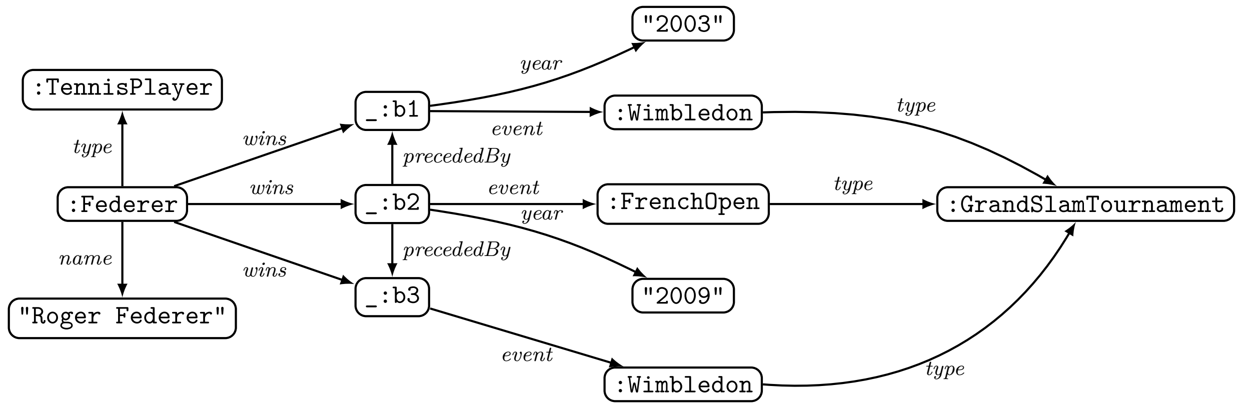


Fig. 1. An RDF graph for our running example. In this graph, URIs are preceded by '`:`', blank nodes by '`_:`' and literals are enclosed in quotation marks (we duplicate the term `:Wimbledon` for clarity).

a tuple $(s, p, o) \in \mathbf{UB} \times \mathbf{U} \times \mathbf{UBL}$ where s is called the *subject*, p the *predicate* and o the *object*.

Definition 2.1 *An* RDF graph *(or simply "graph", where unambiguous) is a finite set of RDF triples.*

The set of *terms* of a graph G, denoted by terms(G), is the set of elements of **UBL** that occur in the triples of G. A *vocabulary* is a subset of **UL**. The vocabulary of G, denoted by voc(G), is defined as terms$(G) \cap \mathbf{UL}$. Given a vocabulary V and a graph G, we say that G is a graph *over* V whenever voc$(G) \subseteq V$. A graph G is *ground* if it does not contain blank nodes (*i.e.*, terms$(G) \cap \mathbf{B} = \emptyset$).

A *map* is a partial function $\mu : \mathbf{UBL} \to \mathbf{UBL}$ whose domain is denoted by dom(μ) and is the identity on URIs and literals, *i.e.*, $\mu(u) = u$ for all $u \in \text{dom}(\mu) \cap \mathbf{UL}$; blank nodes can be mapped to any term. Given a graph G, we define $\mu(G)$ as the set of all $(\mu(s), \mu(p), \mu(o))$ such that $(s, p, o) \in G$. We overload the meaning of map and we say that a *map* μ *is from* G_1 *to* G_2 (or *is a homomorphism from* G_1 *to* G_2), denoted by $\mu : G_1 \to G_2$, if dom$(\mu) =$ terms(G_1) and $\mu(G_1) \subseteq G_2$. Moreover, we use notation $G_1 \to G_2$ if such a map μ exists. Finally, we say that two graphs G_1 and G_2 are *isomorphic*, denoted by $G_1 \cong G_2$, if there exists a map $\mu :$ terms$(G_1) \to$ terms(G_2) such that μ maps blank nodes to blank nodes on a one-to-one basis and such that for every triple (s, p, o), it holds that $(s, p, o) \in G_1$ if and only if $(\mu(s), \mu(p), \mu(o)) \in G_2$. A graph is trivially isomorphic with itself per the identity mapping.

A map μ is *consistent* with G if $\mu(G)$ is an RDF graph, *i.e.*, if s is the subject of a triple in G, then $\mu(s) \in \mathbf{UB}$, and, trivially, if p is the predicate of a triple in G, then $\mu(p) \in \mathbf{U}$, *etc.*[3] If μ is consistent with G, we say that $\mu(G)$ is an *instance* of G. An instance $\mu(G)$ of G is *proper* if $\mu(G)$ has fewer blank nodes than G. This occurs when μ maps blank nodes to URIs or literals and/or maps two or more blank nodes to the same blank node.

We now define two important operations on graphs. The *union* of G_1 and G_2, denoted $G_1 \cup G_2$, is the set theoretical union of their sets of triples. A *merge* of G_1 and G_2, denoted[4] $G_1 + G_2$, is the union $G_1' \cup G_2'$, where G_1' and G_2' are isomorphic copies of G_1 and G_2, respectively, and where the sets of blank nodes in G_1' and G_2' are disjoint from each other. All possible merges of G_1 and G_2 are pair-wise isomorphic such that $G_1 + G_2$ is unique up to isomorphism, and we thus call it *the* merge of G_1 and G_2.

2.2 Semantics of RDF graphs

Applying a standard logical treatment, notions of interpretation and entailment for RDF graphs were defined by Hayes [35]; we now introduce these concepts, omitting *datatype interpretations* for brevity as they are not directly concerned with our discussion of blank nodes. Furthermore, for now, we do not consider the use of vocabularies with predefined semantics (like RDF(S) or OWL); we discuss blank nodes in the context of these standards in Section 5. Graphs that do not use such predefined vocabularies are called *simple*; we now define their semantics.

[3] μ can only be *in*consistent with G if it maps a subject of a triple in G from a blank node to a literal.

[4] Sometimes denoted $G_1 \uplus G_2$.

Definition 2.2 *A* simple interpretation $\mathcal{I}$ *over a vocabulary V is a tuple $\mathcal{I} = (Res, Prop, Ext, Int)$ such that:*

(i) *Res is a non-empty set of* resources*, called the* domain *or* universe *of $\mathcal{I}$;*
(ii) *Prop is a set of properties (not necessarily disjoint from or a subset of Res);*
(iii) *$Ext : Prop \to 2^{Res \times Res}$ is a mapping that assigns an* extension *to each property; and*
(iv) *$Int : V \to Res \cup Prop$ is the* interpretation mapping *that assigns a resource or a property to each element of V such that Int is the identity for literals.*

The semantics of RDF graphs is based on the notion of *simple entailment* whereby a simple interpretation can serve as a *model* of a graph. Given a vocabulary V and a simple interpretation $\mathcal{I} = (Res, Prop, Ext, Int)$ over V, a ground triple (s, p, o) over V is *true* under $\mathcal{I}$ if:

– $\mathcal{I}$ *interprets* p as a property (that is, $Int(p) \in Prop$), and thus $\mathcal{I}$ assigns an *extension* (a set of pairs of resources) to the interpretation of the name p, *and*
– the interpretation of the pair (s, o) belongs to the extension of the interpretation of p, that is, $(Int(s), Int(o)) \in Ext(Int(p))$.

These definitions do not yet consider the interpretation of blank nodes; for this, we need to define a version of the simple interpretation mapping that includes the set of blank nodes as part of its domain.

Definition 2.3 *Let G be an RDF graph and $\mathcal{I} = (Res, Prop, Ext, Int)$ be a simple interpretation. Let $A : \mathbf{B} \to Res$ be a function from blank nodes to resources and let Int_A denote an amended version of Int that includes $\mathbf{B}$ as part of its domain such that $Int_A(x) = A(x)$ for $x \in \mathbf{B}$ and $Int_A(x) = Int(x)$ for $x \in \mathbf{UL}$. We say that $\mathcal{I}$ is a model of G if $\mathcal{I}$ is an interpretation over* voc(G) *and there exists a mapping A such that for each $(s, p, o) \in G$, it holds that $Int(p) \in Prop$ and $(Int_A(s), Int_A(o)) \in Ext(Int(p))$.*

Given RDF graphs G and H, we say that G *simply-entails* H, denoted by $G \models H$, if every model of G is also a model of H. Intuitively speaking, G simply-entails H if the information contained in H is a subset of the information contained in G: if every model of G turns out to be a model of H as well, then H does not provide more information than G. This can be seen from a formal point of view with the following result.

Theorem 2.4 (**[19, 31]**) *Given two RDF graphs G and H, the simple entailment $G \models H$ holds if and only if there is a map $\mu : H \to G$.* $\square$

An immediate consequence of this theorem is that deciding simple entailment is NP-complete. [5] It is also known that the intractability of deciding whether an RDF graph G simply-entails a graph H depends *only* on the structure of the subgraph of H induced by its blank nodes [58].

Along with the notion of entailment comes the notion of *leanness.* Recall that a subgraph is a subset of a graph and a proper subgraph is a subgraph with fewer triples.

Definition 2.5 *An RDF graph G is* lean *if there is no map μ such that $\mu(G)$ is a proper subgraph of G; otherwise, the graph is* non-lean*.*

In other words, a graph is non-lean if it contains "redundant" triples [6]; that is, it is not simple-entailed by a smaller graph. Alongside the notion of graphs being lean or non-lean, we may also intuitively refer to blank nodes as being lean or non-lean.

Definition 2.6 *We call a blank node b in* terms(G) non-lean *with respect to G if there exists a map μ such that $\mu(G)$ is a proper subgraph of G and there exists a term x in* terms(G) *such that $x \neq b$ and $\mu(b) = x$. In this case, we call x a* witness *for non-lean b with respect to G. Otherwise if b has no such witness, we call it* lean *with respect to G.*

Non-lean blank nodes are the cause of redundant triples in non-lean graphs. A graph is non-lean if and only if it contains one or more non-lean blank nodes.

Example 2.7 *The graph G in Figure 1 is* non-lean *as the triple* (`_:b3`, *event*, `:Wimbledon`) *is made redundant by* (`_:b1`, *event*, `:Wimbledon`)*: if μ is a map that replaces* `_:b3` *by* `_:b1` *and is the identity elsewhere, then $\mu(G)$ is a proper subgraph of G. We say that the blank node* `_:b3` *is* non-lean *and that its witness is* `_:b1`*.*

The notion of leanness for RDF graphs corresponds to the notion of the core of a graph introduced by Hell and Nešetřil [38], and studied

[5] Simple entailment can be directly stated in terms of graph homomorphism as first observed by Carroll [35, §7.1]. Later, ter Horst also demonstrated the NP-complete result by reduction from the clique problem [67].
[6] Redundant, at least, in the sense of simple entailment per the official semantics.

in the context of data exchange [24] and Web databases [31]. In fact, from these results [38, 24], it is possible to conclude that the complexity of the problem of verifying whether or not an RDF graph G is lean, is coNP-complete, as demonstrated previously by Gutierrez *et al.* [31].

3 Theoretic background

The idea of existential blank nodes is not entirely new and has direct analogues in other fields. To give a broader theoretical context, in this section, we relate the standard semantics of blank nodes to existentials in first-order logic (§3.1), to null values in database theory (§3.2) and we also look at the formal background to Skolemisation, which has been proposed as a formal method for treating existential blank nodes as fresh constants (§3.3).

3.1 Existential variables in first-order logic

As was mentioned in Section 2, the feature of existentiality of blank nodes is given by the extension function A for an interpretation mapping *Int*. In this section, we briefly show that this way of interpreting blank nodes can be precisely characterised in terms of existential variables in first-order logic.

Let G be a RDF graph. Let $\mathbf{V}$ be an infinite set of variables disjoint with $\mathbf{U}$, $\mathbf{L}$ and $\mathbf{B}$, and assume that $\rho : \mathbf{UBL} \to \mathbf{UVL}$ is a one-to-one function that is the identity on $\mathbf{UL}$. For every triple $t = (s, p, o)$ in G, define $\rho(t)$ to be the fact $\mathit{triple}(\rho(s), \rho(p), \rho(o))$, for *triple* a ternary predicate. We then define $\mathit{Th}(G)$ as a first-order sentence of the following form:

$$\mathit{Th}(G) = \exists x_1 \cdots \exists x_n \left(\bigwedge_{t \in G} \rho(t) \right),$$

where $x_1, \ldots, x_n$ are the variables from $\mathbf{V}$ mentioned in $\bigwedge_{t\in G} \rho(t)$. Then we have the following equivalence between the notion of (simple) implication for RDF graphs and the notion of logical consequence for first-order logic.

Theorem 3.1 (implicit in [22]) [7] *Given two RDF graphs G and H, the simple entailment $G \models H$ holds if and only if $\mathit{Th}(G) \models \mathit{Th}(H)$.* □

[7] The translation to a first-order setting in [22] uses F-Logic [44] instead of classical first-order logic, which slightly differs from our encoding here but may be considered as syntactic sugar.

Interestingly, the above theorem tells us that the (simple) implication problem for RDF graphs can be reduced to the implication problem for existential first-order formulas without negation and disjunction. Given that the latter problem can be solved by checking whether there exists a homomorphism from the consequent to the premise of the implication [19], one obtains Theorem 2.4 as a corollary of Theorem 3.1.

3.2 Incomplete information in database theory

We now show that the work on incomplete information for relational databases can also be used to characterise simple entailment with blank nodes. [8]

In the relational world, null values are used to represent incomplete information [40, 28, 1]. More precisely, assume as given a set $\mathbf{D}$ of constants and a set $\mathbf{N}$ of null values, where $\mathbf{D}$ and $\mathbf{N}$ are disjoint, and assume as given a relational schema $\mathbf{R} = \{R_1, \ldots, R_n\}$, where each R_i is a relation name of arity k_i $(1 \leq i \leq n)$. Then an instance I of $\mathbf{R}$ (with complete information) assigns to each relation symbol R_i a finite k_i-ary relation $R_i^I \subseteq \mathbf{D}^{k_i}$; that is, a k_i-ary relation including only constants. On the other hand, a naive instance I of $\mathbf{R}$ (with incomplete information) assigns to each relation symbol R_i $(1 \leq i \leq n)$ a finite k_i-ary relation $R_i^I \subseteq (\mathbf{D} \cup \mathbf{N})^{k_i}$; that is, a k_i-ary relation including constants and null values.

Example 3.2 *The following is a naive instance over a schema consisting of a ternary relation R:*

R		
a	b	n
c	d	n

This naive instance contains two tuples where a, b, c and d are constants ($a, b, c, d \in \mathbf{D}$), and n is a null value ($n \in \mathbf{N}$). □

The use of the same null value n in the two tuples in Example 3.2 indicates that the value of the third column for these tuples is the same, although it is not known. More precisely, the semantics of naive instances is given in terms of an interpretation function that is defined as follows. Given a naive instance I of a relational schema $\mathbf{R}$, define $\mathrm{nulls}(I)$ as the set of nulls mentioned in I. Moreover, given a null substitution $\nu : \mathrm{nulls}(I) \to \mathbf{D}$, for every $R \in \mathbf{R}$ define

[8] We discuss more closely the treatment of blank nodes in the related RDB2RDF standard later in Section 5.6.

$\nu(R^I) = \{\nu(t) \mid t \in R^I\}$, where $\nu(t)$ is obtained by replacing every null n in t by its image $\nu(n)$. Then for every naive instance I, define the set of *representatives* of I, denoted by $\text{rep}(I)$, as [40]:

$$\{J \mid J \text{ is an instance of } \mathbf{R} \text{ and there exists } \nu : \text{nulls}(I) \to \mathbf{D} \text{ such that for every } R \in \mathbf{R}, \text{ it holds that } \nu(R^I) \subseteq R^J\}.$$

That is, a representative of a naive instance I is obtained by replacing the null values of I by constants, and possibly by also adding some extra tuples mentioning only constants.

Example 3.3 *If e is a constant value, then the following are three representatives of the naive instance mentioned in Example 3.2:*

R		
a	b	a
c	d	a

R		
a	b	e
c	d	e

R		
a	b	e
c	d	e
b	c	d

Each such representative replaces the null term n in the naive instance of Example 3.2 with a constant, and in the third case, adds an extra tuple with only constants.

A naive instance I is then said to be contained in a naive instance J if $\text{rep}(I) \subseteq \text{rep}(J)$ [2].

Example 3.4 *Assume that J is the following naive instance:*

R		
a	b	n_1
c	d	n_2

In this naive instance, $a, b, c, d \in \mathbf{D}$ and $n_1, n_2 \in \mathbf{N}$. Thus, we have that the naive instance I in Example 3.2 is contained in J, while J is not contained in I.

Not surprisingly, the notion of containment for naive instances can be used to characterise the notion of simple entailment for RDF graphs. More precisely, assume that $\mathbf{D} = \mathbf{U} \cup \mathbf{L}$ and $\mathbf{N} = \mathbf{B}$, and for every RDF graph G, define $I(G)$ as a naive instance over the relational schema $\{\textit{triple}(\cdot,\cdot,\cdot)\}$ such that $\textit{triple}^{I(G)} = G$ (that is, each triple (s,p,o) in G is stored in $I(G)$ as the fact $\textit{triple}(s,p,o)$). Then we have the following equivalence between the notions defined in this section:

Theorem 3.5 *Given two RDF graphs G and H, the simple entailment $G \models H$ holds if and only if $I(H)$ is contained in $I(G)$.* $\square$

Thus we see the relationship between simple entailment for RDF graphs and the use of nulls for relational databases with incomplete information.

3.3 Skolemisation

In first-order logic, Skolemisation [9] is a way of removing existential quantifiers from a formula in prenex normal form (a chain of quantifiers followed by a quantifier-free formula). The process was originally defined and used to generalise a theorem by Jacques Herbrand about models of universal theories [16].

The central idea of Skolemisation is to replace existentially quantified variables by "fresh" constants that are not used in the original formula. For example, $\exists x \forall y\, R(x,y)$ can be replaced by the formula $\forall y\, R(c,y)$, where c is a fresh constant, as this new formula also represents the fact that there exists a value for the variable x (in fact, $x = c$) such that $R(x,y)$ holds for every possible value of variable y.

Similarly, if we let f denote a fresh unary function symbol that is not used in the original formula, then $\forall x \exists y\, (P(x) \to Q(y))$ can be replaced by $\forall x (P(x) \to Q(f(x)))$ since we know that for every possible value of variable x, there exists a value of variable y that depends on x and such that $P(x) \to Q(y)$ holds.

When the original formula does not have universal quantifiers, only constants (or 0-ary functions) are introduced in the Skolemisation process. In our study of simple RDF graphs only existential quantifiers are needed (see Definition 2.3), so we will talk about Skolem constants only. However, if Skolemisation were to be used to study satisfiability of logical formulae of more expressive languages (*e.g.*, OWL), Skolem functions would be needed. [10]

The most important property of Skolemisation in first-order logic is that it preserves satisfiability of the formula being Skolemised. In other words, if ψ is a Skolemisation of a formula φ, then φ and ψ are equisatisfiable, meaning that φ is satisfiable (in the original vocabulary) if *and only if* ψ is satisfiable (in the extended vocabulary, with the new Skolem functions and constants). However, all simple RDF graphs are trivially satisfiable thanks to Herbrand interpretations [35], in which URIs and literals are

[9] Named after Norwegian logician Thoralf Skolem.
[10] Skolem functions are also required for the use of blank nodes in SPARQL CONSTRUCT queries [59]. We discuss the role of blank nodes in SPARQL later in Section 5.4.

interpreted as their corresponding (unique) syntactic forms instead of "real world" resources; thus, equisatisfiability is trivial at the level of simple entailment. However, when the satisfiability of logical formulae of more expressive languages are considered (*e.g.*, considering well-defined vocabulary layered on top of simple entailment, such as RDFS or OWL), the equisatisfiability of a formula and its Skolemised form becomes a non-trivial property.

4 Simple Entailment Checks in Practice

Thus far we have looked at the theoretical perspective of simple entailment in relation to problems in other areas, focusing on the worst case complexity of checking simple entailment and leanness. However, worst-case scenarios for simple entailment checking rarely occur in practice. In this section, we look at more practical aspects of the problem of simple entailment, starting with a tighter bound for simple entailment checking in common cases (§4.1), and discussing how checking simple entailment and leanness can be supported through basic graph pattern matching in SPARQL, for which efficient off-the-shelf implementations can be used (§4.2).

4.1 Tighter bound for entailment in practice

As previously discussed, the fact that simple entailment in the presence of existentials is NP-complete follows as a corollary of various NP-complete problems in other related fields [35, 6, 67]. However, Pichler *et al.* [58] examine a tighter bound for common cases, noting that for RDF graphs with certain blank node morphologies, simple entailment checks become tractable.

Towards defining these tractable cases, let G be an RDF graph, and consider the *blank graph* $\mathrm{blank}(G) = (V, E)$ where the set of vertices V is $\mathbf{B} \cap \mathrm{terms}(G)$ and the set of edges E is given as:

$$\{(b, c) \mid b \in V, c \in V, b \neq c \text{ and there exists } P \in \mathrm{terms}(G) \text{ such that } (b, P, c) \in G \text{ or } (c, P, b) \in G\}.$$

In other words, $\mathrm{blank}(G)$ gives an undirected graph connecting blank nodes appearing in the same triple in G (loops are of no consequence). Let G and H denote two RDF graphs with m and n triples respectively. Pichler *et al.* [58] demonstrated that performing the simple entailment check $G \models H$ has the upper bound $O(n^2+mn^{2k})$, where $k = \mathrm{tw}(\mathrm{blank}(H))+1$ for $\mathrm{tw}(\mathrm{blank}(H))$ the *treewidth* of $\mathrm{blank}(H)$ [58]. We will survey the treewidth of such blank graphs in published data in Section 6.2, which provides an upper-bound estimate for the expense of RDF entailment checks in practice.

The complexity of checking the simple entailment $G \models H$ thus relies on the treewidth of H. We remark that a graph H may entail non-lean graphs with higher treewidth, but that this does not affect the bound on complexity.

Example 4.1 *Take the graph H:*

```
<x> <p> <x> .
```

This graph simply-entails H' with a blank node cycle as follows:

```
_:x1 <p> _:x2 .
_:x2 <p> _:x3 .
_:x3 <p> _:x1 .
```

The entailment is based on the existence of a map μ that maps all blank nodes in H' back to `<x>`*. In fact, considering such a map μ, H would entail any subset of* $(\mathbf{B} \cup \{\texttt{<x>}\}) \times \{\texttt{<p>}\} \times (\mathbf{B} \cup \{\texttt{<x>}\})$*. Hence H entails RDF graphs with arbitrary blank graphs.*

However, if checking $G \models H$, only the treewidth of H is important, not the graphs that it entails.

4.2 Checking simple entailment and leanness using basic graph pattern evaluation

SPARQL is the standard query language for RDF and can be used to support simple entailment [26].

SPARQL queries are defined over a *SPARQL dataset*, given as $\{G_0, (u_1, G_1), \ldots, (u_n, G_n)\}$ where $u_1, \ldots, u_n$ are distinct URIs and $G_0, \ldots, G_n$ are RDF graphs. Each pair (u_i, G_i) is called a *named graph* and G_0 is called the *default graph*. Thus, SPARQL is not defined directly over RDF, but rather over sets of named RDF graphs. The SPARQL standard then allows for basic graph pattern matching, which primarily involves posing conjunctive queries against named combinations of these graphs, along with other features such as optionals (*i.e.*, left-joins), unions (*i.e.*, disjunctions), filters, solution modifiers, and so forth. SPARQL 1.1 extends this feature-set towards including property paths, aggregates, sub-queries, entailment regimes and much more.

The problem of simple entailment can be trivially stated in terms of evaluating basic graph patterns in SPARQL [26], which allows simple entailment tasks to be performed using widely available, optimised, off-the-shelf SPARQL engines.

To check if the simple entailment $G \models H$ holds using SPARQL:

(i) construct a SPARQL query by using H as a basic graph pattern and embedding it in an `ASK` query;
(ii) construct a SPARQL dataset containing only G as a default graph;
(iii) execute the query against the dataset.

The `true` or `false` result returned from the `ASK` query indicates whether or not $G \models H$.

Example 4.2 *Let G be the graph of Figure 1 and let H be an RDF graph with the following five triples:*[11]

```
_:player a :TennisPlayer ; :wins _:ev1 , _:ev2 .
_:ev1 :year 2003 . _:ev2 :year 2009 .
```

To see if the simple entailment $G \models H$ holds, we can use SPARQL: we can create a default graph using G and evaluate the following query representing H against it:

```
PREFIX : <http://example.org/>
ASK {
  _:player a :TennisPlayer ; :wins _:ev1 , _:ev2 .
  _:ev1 :year 2003 . _:ev2 :year 2009 .
}
```

Blank nodes in SPARQL are treated as non-distinguished variables that will match any term in the data but cannot be projected as a result.

In this case, the answer to the query is `true`, *indicating that $G \models H$.*

SPARQL can also be used to determine if a graph is (non-)lean. Recall that a graph G is lean if and only if it has no proper subgraph $G' \subset G$ such that $G' \models G$ under simple entailment. Let n be the number of triples in G that contain a blank node and let $\{G_1, \ldots, G_n\}$ be the set of graphs constructed by removing one such triple from G. Then G is lean if and only if $G \not\models G_i$ for $1 \leq i \leq n$. Each of these simple (non-)entailment checks can be run using the above procedure with SPARQL.

Alternatively, instead of using n `ASK` queries to see if a graph is non-lean, we can use a single `SELECT` query to look for witnesses for non-lean blank nodes. Let $\mathbf{V}$ represent an infinite set of variables disjoint with $\mathbf{UBL}$ and (as before in Section 3.2) let $\rho : \mathbf{UBL} \to \mathbf{UVL}$ be a one-to-one function that is the identity on $\mathbf{UL}$. Also let $\rho(s,p,o)$ denote $(\rho(s), \rho(p), \rho(o))$ and $\rho(G)$ denote $\{\rho(s,p,o) : (s,p,o) \in G\}$. To check if a graph G is lean using SPARQL basic graph pattern evaluation:

(i) construct a SPARQL query with $\rho(G)$ as a basic graph pattern embedded in a `SELECT *` query;
(ii) construct a SPARQL dataset containing only G as a default graph;
(iii) execute the query against the dataset.

Trivially, if a query variable is bound to a term in G other than the blank node for which it was originally created by ρ, then that term is a witness for the non-leanness of the blank node and thus G is non-lean. Since blank nodes may be arbitrarily relabelled during SPARQL query evaluation, checking that the surrogate variable corresponds with its original blank node may be impossible. However, since a surrogate variable will always bind its original blank node, to check whether or not G is non-lean, one can check the solutions to see if there is any surrogate variable bound to more than one unique term.

Example 4.3 *Take the following RDF graph G; a subset of the RDF graph represented by Figure 1.*

```
:Federer a :TennisPlayer ; :wins _:b1 , _:b2 , _:b3 .
_:b1 :event :Wimbledon ; :year 2003 .
_:b2 :event :FrenchOpen ; :year 2009 .
_:b3 :event :Wimbledon .
_:b2 :precededBy _:b1 , _:b3 .
```

For clarity of example, we map blank nodes to surrogate query variables using the simple syntactic convention $\rho($`_:b`*$) =$* `?b`*. We then wrap $\rho(G)$ into the following SPARQL* `SELECT DISTINCT *` *query.*[12]

```
PREFIX : <http://example.org/>
SELECT DISTINCT * WHERE {
  :Federer a :TennisPlayer ; :wins ?b1 , ?b2 , ?b3 .
  ?b1 :event :Wimbledon ; :year 2003 .
  ?b2 :event :FrenchOpen ; :year 2009 .
  ?b3 :event :Wimbledon .
  ?b2 :precededBy ?b1 , ?b3 .
}
```

[11] We assume reader familiarity with Turtle and SPARQL syntax.

[12] The `DISTINCT` keyword is optional, but helps for clarity.

If we apply the above query against G, the query solutions returned would be:

?b1	?b2	?b3
_:b1	_:b2	_:b3
_:b1	_:b2	_:b1

Intuitively, we can see that `_:b1` *is found to be a witness for* `_:b3`. *In reality, the blank node terms in the solutions may be relabelled during query evaluation. In any case, since* `?b3` *contains two terms in the query solutions, we can conclude that G is non-lean. Furthermore, we can conclude that the blank-node used to generate the surrogate variable* `?b3` *is "redundant": letting G' be the graph G with all triples containing* `_:b3` *removed, we can see that $G' \models G$.*

As per simple entailment, checking if a graph G is (non-)lean can be performed using off-the-shelf SPARQL engines for which optimised implementations exist. Additionally, if the function ρ and its inverse can be "preserved" in an *ad hoc* manner when evaluating the query (*e.g.*, using syntactic convention), then a SPARQL engine can also identify non-lean blank nodes and their witnesses, making the process of leaning a graph straightforward to implement using SPARQL.

5 Blank nodes in the standards

Having looked at the theoretical and practical aspects of the semantics of blank nodes in the RDF Semantics, in this section, we look at the current treatment of blank nodes in standards related to RDF. We first look at the use of blank nodes in the syntaxes recommended for serialising RDF, *viz.* RDF/XML [9], N-Triples [8], Turtle [7], RDFa [39] and JSON-LD [65]. We also provide detailed discussion of the role of blank nodes within standards relating to RDF, *viz.* RDFS [15], OWL (2) [64], SPARQL (1.1) [60, 33], RIF [12] and RDB2RDF [21, 3]. Developements relating to the current RDF 1.1 Working Drafts [20, 36] will be discussed later in Section 8.

5.1 RDF Syntaxes

We first give general discussion on the role of blank nodes in RDF syntaxes, and in particular, how they are serialised.

Support for blank nodes. In all RDF syntaxes, blank nodes can be explicitly labelled such that they can be referred to at any point within the document. In fact, in the N-Triples syntax where all RDF terms must be given in full, blank nodes must always be explicitly labelled. Explicit labels allow blank nodes to be referenced outside of nested elements and thus to be used in arbitrary graph-based data even though the underlying syntaxes (*e.g.*, XML) are inherently tree-based. Note that we will study cyclic blank node structures in published data later in Section 6.2.

Features requiring blank nodes. An RDF tool can safely perform a one-to-one relabelling of blank-nodes without affecting the interpretation of the data: the strings used to label blank-nodes do not matter so long as they do not collide with other such labels. Using this feature, blank nodes play a major role for providing shortcuts in RDF/XML, Turtle and RDFa, where triple positions (that are not important to name with a URI or literal) can be left implicit using syntactic sugar. In such cases, the parser will automatically assign consistent blank-node labels for these implicit positions when extracting triples. Using a similar principle, blank nodes are also used in shortcuts for n-ary predicates and RDF lists (a.k.a. containers) in Turtle, RDF/XML and (potentially) JSON-LD [7, 9, 65] as well as containers and reification in RDF/XML [9].

Example 5.1 *Consider representing an ordered list of Tennis Grand-Slams in RDF, where we can use the Turtle shortcut:*

```
:GrandSlam :order (:AustralianOpen :FrenchOpen
                          :Wimbledon :USOpen) .
```

which encodes an (ordered) RDF list. This would be equivalently representable in Turtle's square-bracket syntax (left) as:

```
:GrandSlam :order
  [ rdf:first :AustralianOpen ; rdf:rest
  [ rdf:first :FrenchOpen ; rdf:rest
  [ rdf:first :Wimbledon ; rdf:rest
  [ rdf:first :USOpen ; rdf:rest rdf:nil ]]]] .
```

or in the full triple form as:

```
:GrandSlam :order _:b1 .
 _:b1 rdf:first :AustralianOpen . _:b1 rdf:rest _:b2 .
 _:b2 rdf:first :FrenchOpen . _:b2 rdf:rest _:b3 .
```

```
_:b3 rdf:first :Wimbledon . _:b3 rdf:rest _:b4 .
_:b4 rdf:first :USOpen . _:b4 rdf:rest rdf:nil .
```

The first shortcut notation omits both the auxiliary blank nodes, as well as the standard RDF vocabulary used to represent ordered lists in triple form. The second shortcut notation omits only the auxiliary blank nodes, using nested implicit elements to represent the tree-structured list. Neither of the first two notations would be possible without automatically-generated blank-node labels.

Similar shortcuts using unlabelled blank nodes hold for n-ary predicates, reification and containers in RDF/XML. It is important to note that such shortcuts can only induce "trees" of blank nodes, branching from subject to object; for example:

```
_:b1 :p _:b2 . _:b2 :p _:b1 .
```

cannot be expressed without manually labelling blank nodes, no matter which RDF syntax is under consideration. This is due to the tree-based syntaxes used to serialise RDF, which rely on nested elements (*e.g.*, XML for RDF/XML and RDFa and JSON for JSON-LD).

The JSON-LD [65] specification departs from RDF by allowing blank nodes as predicates. Much like blank nodes in the subject or object position of RDF, blank nodes in the predicate position allow publishers to forego minting a URI for properties in their JSON-LD document (thus narrowing the adoption gap between native JSON and JSON-LD).

Issues with blank nodes. Given a fixed, serialised RDF graph (*i.e.*, a document), labelling of blank nodes can vary across parsers and across time. Checking if two representations originate from the same data thus often requires an isomorphism check, for which in general, no polynomial algorithms are known (*cf. e.g.* [18] in the RDF context; isomorphism checking is, however, polynomial for "blank node trees" [43]). Furthermore, consider a use-case tracking the changes of a document over time; given that parsers can assign arbitrary labels to blank nodes, a simple syntactic change to the document may cause a dramatic change in blank node labels, making precise change detection difficult (other than on a purely syntactic level).

In practice. Parsers typically feature a systematic means of labelling blank nodes based on the explicit blank node labels and the order of appearance of implicit blank nodes.

The popular Jena Framework[13] offers sound and complete methods for checking the isomorphism of two RDF graphs.

In a study of Linked Data dynamics, Käfer *et al.* [41] applied a heuristic algorithm for guessing if two RDF graphs were equal (*i.e.*, that two versions of an RDF graph remain the same): the algorithm sets the comparison of all pairs of blank nodes across documents as equal, where if the number of triples is the same and the set of RDF triples in both documents is equal under this heuristic, the algorithm considers the documents as provisionally isomorphic. If the heuristic returns true, only then is the Jena library used to execute a full isomorphism check. Comparing 29 versions of over eighty thousand RDF documents, the authors found that in all cases, documents were provisionally isomorphic if and only if they were isomorphic.

Tzitzikas *et al.* [69] propose heuristic methods to identify subgraph-isomorphisms involving blank nodes, with the goal of computing a minimal delta between RDF graphs. They define the size of the delta between two RDF graphs as the edits (triple addition/deletions) required to make the graphs isomorphic and search for a blank node bijection between the two graphs that minimises this edit distance. Since subgraph-isomorphism is NP-Complete, the authors propose two tractable approximations. The first is based on the Hungarian method for pair-wise comparison, which produces smaller deltas but at additional cost. The second algorithm computes signatures for blank nodes based on the ground information associated with them, which produces larger deltas but at reduced cost.

In a separate issue—and as previously mentioned—the JSON-LD [65] specification permits use of blank nodes in the predicate position resulting in a form of "generalised RDF". However, the semantics of blank nodes in the predicate position are not defined by the RDF Semantics [35, 36]. Likewise most of the standards and tools built on top of RDF do not support blank-nodes in such positions. The JSON-LD specification states that to map such data to RDF, URIs (or more accurately IRIs) must first be minted for predicate terms.

[13] http://jena.sourceforge.net/

5.2 *RDF Schema (RDFS)*

RDF Schema (RDFS) is a lightweight language for describing RDF vocabularies, which supports features such as class and property hierarchies (*i.e.*, subsumption), the definition of domain- and range-classes associated with property terms, and others besides. The RDFS vocabulary—including, *e.g.*, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf` and `rdfs:subPropertyOf`—is well-defined by means of a (normative) model-theoretic semantics, accompanied by a (non-normative) set of entailment rules to support inferencing [35, 55]. A sample of such rules is shown in Table 1.

Support for blank nodes. RDFS entailment is built on top of simple entailment, and thus supports an existential semantics for blank nodes as described in Section 2.

Features requiring blank nodes. The restrictions placed on which terms can appear in which position of an RDF triple would, without further treatment, make the entailment rules incomplete with respect to RDFS semantics. To (help) overcome this problem, blank nodes are used as "surrogates" to represent literals in the subject position where literals would otherwise be disallowed. The RDF Semantics document [35] proposes using rules lg & gl in Table 1 to implement this bijection between literals and surrogate blank nodes.

Example 5.2 *To see why surrogate blank nodes are necessary, consider the (somewhat unorthodox) RDF graph:*

```
:Federer atp:name "Roger Federer" .
atp:name rdfs:range atp:PlayerName .
```

which should RDFS-entail the triple:

```
"Roger Federer" a atp:PlayerName .
```

However, the latter triple is not a valid RDF triple since a literal appears in the subject position; thus it will not be inferred (the domain of the `?v` *variable in the* rdfs3 *rule would prevent the inference). And so, to achieve the valid inference:*

```
_:RogerFederer a atp:PlayerName .
```

*(*i.e.*, that a member of* `atp:PlayerName` *does exist) requires the use of a surrogate blank node (*viz. `_:RogerFederer`*) through rule* lg.

The inverse rule gl then allows surrogates to "travel" back as literals into the object position, though examples of such behaviour are again not necessarily intuitive.

Example 5.3 *This time take the triples:*

```
:hasType rdfs:range rdfs:Class .
:RogerFederer :hasType "TennisPlayer" .
```

where the first triple is axiomatically true in RDFS; then we should be able to infer the following:

```
:RogerFederer :hasType _:TennisPlayer .
_:TennisPlayer rdf:type rdfs:Class .
_:TennisPlayer rdfs:subClassOf _:TennisPlayer .
_:TennisPlayer rdfs:subClassOf "TennisPlayer" .
```

which ultimately concludes in the final triple that there is a subclass of "TennisPlayer" (in this case, itself). To get this latter inference, we require application of rules lg*,* rdfs3*,* rdfs10 *and then finally* gl*, respectively. The inference would not be possible without a combination of* lg*/*gl*.*

Again, using a literal to represent an RDFS class is highly unorthodox. In summary, the use of surrogate blank-nodes covers certain corner-cases for the completeness of RDFS entailment rules caused by positional restrictions in RDF terms.

Issues with blank nodes. As previously discussed, the existential semantics of blank nodes makes RDFS entailment, which is built upon simple entailment, NP-Complete [35, 31, 67, 55]. Furthermore, simple entailment rules are not *range-restricted* unlike other rules for RDF(S): existential variables that appear in the heads of rules need not appear in the body of rules, with consequences for guarantees of termination. Of course, the lg & gl rules are safe in this respect given that there is a one-to-one mapping between the finite set of literals in the RDF graph and the set of surrogate blank nodes produced. However, in the most naive sense, simple entailment rules can infer an arbitrary set of (highly non-lean) triples with arbitrary blank node labels. In fact, since RDFS entailment axiomatically entails reflexive subclass and subproperty triples, in theory, all RDF graphs, including the empty RDF graph,

Table 1. Selection of RDFS rules. Variables are defined with restrictions as follows: dom(`?a`) = dom(`?b`) = **U**; dom(`?u`) = dom(`?v`) = **UB**; dom(`?x`) = dom(`?y`) = **UBL**; dom(`?l`) = **L**. Rule gl only permits the inverse mapping of lg.

ID	Body		Head
lg	`?u ?a ?l .`	$\Rightarrow$	`?u ?a _:l .`
gl	`?u ?a _:l .`	$\Rightarrow$	`?u ?a "l" .`
rdfs2	`?a rdfs:domain ?x . ?u ?a ?y .`	$\Rightarrow$	`?u rdf:type ?x .`
rdfs3	`?a rdfs:range ?x . ?u ?a ?v .`	$\Rightarrow$	`?v rdf:type ?x .`
rdfs7	`?a rdfs:subPropertyOf ?b . ?u ?a ?y .`	$\Rightarrow$	`?u ?b ?y .`
rdfs9	`?u rdfs:subClassOf ?x . ?v rdf:type ?u .`	$\Rightarrow$	`?v rdf:type ?x .`
rdfs10	`?u rdf:type rdfs:Class .`	$\Rightarrow$	`?u rdfs:subClassOf ?u .`

will RDFS-entail arbitrary blank node graphs (per Example 4.1).

Otherwise, the RDFS rules would only operate over the fixed vocabulary of the RDF graph voc(G) and the built-in RDF(S) vocabulary $\mathcal{R}$ itself. Unfortunately $\mathcal{R}$ contains infinite container-membership properties of the form `rdf:_`n for $n \in \mathbb{N}$. Without simple entailment and these container-membership properties, the complete RDFS-entailments of a graph G would be bounded by $(\text{voc}(G) \cup \mathcal{R})^3$ and could be materialised.

Furthermore, ter Horst [67] showed that the RDFS entailment lemma in the non-normative section of the RDF Semantics is incorrect: blank node surrogates are still not enough for the completeness of the standard RDFS entailment rules, where blank nodes would also need to be allowed in the predicate position. We now give an example.

Example 5.4 *Consider the three triples:*

```
:Federer :wins _:b1 .
:wins rdfs:subPropertyOf _:p .
_:p rdfs:domain :Competitor .
```

Using the standard RDFS entailment rules (including lg *&* gl*), we cannot infer the triple* `:Federer rdf:type :Competitor`*, since the required intermediate triple* `:Federer _:p _:b1` *is not valid in RDF and there is no standard mechanism by which surrogate URIs can be used to represent blank nodes and literals in the predicate position.*

This incompleteness can be remedied by either:

(i) allowing non-valid RDF triples (generalised RDF) in intermediate inferences [67, 29]); *or*

(ii) by the addition of special inference rules to handle such cases; see, for example, [54].

In practice. Many practical RDFS reasoners ignore simple entailment and surrogate blank nodes, instead opting to support a "ground" subset of the RDFS semantics [56, 70, 72].

5.3 Web Ontology Language (OWL)

The Web Ontology Language (OWL) [14] is, in principle, a vocabulary consisting of URIs in the `owl:` namespace that carry additional semantics, thus extending the possibilities of expressing implicit knowledge in RDF beyond RDFS. OWL is thus a more expressive language than RDFS and partly re-uses the RDFS vocabulary.

With the advent of OWL 2, there are now eight standard (sub-)languages in the OWL standard [29]: OWL Lite, OWL DL, OWL Full, OWL 2 EL, OWL 2 QL, OWL 2 RL, OWL 2 DL and OWL 2 Full. Each profile is a syntactic subset of the OWL language. Furthermore, OWL defines two different semantics for its profiles: an RDF-Based Semantics [61] and a Direct Semantics [51]. The RDF-Based Semantics is applicable for arbitrary RDF graphs (a.k.a. OWL 2 Full), but common reasoning tasks are undecidable [29]. The Direct Semantics requires restrictions on RDF data to ensure decidability, with sound and complete algorithms for many reasoning tasks known from ongoing work on Description Logics (DL) theory [5] and other areas.

Support for blank nodes. The OWL Structural Specification [52] permits use of *anonymous individuals* in assertions, which allow for representing objects that are local to a given ontology and whose identity is not given. Individuals that are not anonymous are called *named individuals.* Anonymous indi-

[14] Recently extended to OWL 2.

viduals are analogous to blank nodes in RDF and are represented in the structural specification with the familiar blank node syntax (*e.g.*, `_:anAnonIndiv`). The structural syntax also states that if two ontologies are being imported, any anonymous individuals they share with the same labels must be "standardised apart"; this is directly analogous to the notion of an RDF merge.

The RDF-Based Semantics of OWL is built on top of simple entailment and thus directly considers blank nodes as existentials [61]. Conversely, the Direct Semantics of OWL does not directly treat any notion of simple entailment or leanness with respect to anonymous individuals [51]; instead, the existential semantics of anonymous individuals is somewhat hidden in the definition of a model:

> *"... an interpretation I = [...] is a model of an OWL 2 ontology O [...] if an interpretation J = [...] exists such that $\cdot^J$ coincides with $\cdot^I$ on all named individuals and J satisfies O"*
>
> — **[51, §2.4]**

In this definition, J can vary from I on the interpretation of anonymous individuals. As such, this definition paraphrases the usual semantic definition of existentials in first-order logic, or, respectively, the semantic definition of blank nodes in simple entailment, which is defined in terms of a blank node assignment A *extending* an interpretation I (see Definition 2.3).

Apart from anonymous individuals, concept-level existentials are commonly used in OWL axioms: for example, the implicit assertion that `:Federer` won "something" can be expressed in the DL axiom $\{\,\texttt{Federer}\,\} \sqsubseteq \exists\texttt{wins}.\top$, *i.e.*, on a semantic level above blank nodes and more generally, above the RDF representation of OWL axioms. Such axioms can then entail the existence of novel anonymous individuals that may not otherwise hold under simple entailment.

Features requiring blank nodes. The Direct Semantics of OWL does not operate directly over RDF, but rather operates over axioms that can be mapped to and from RDF triples [57]. A single axiom can be serialised as multiple triples, involving either an n-ary predicate representation, or sometimes an RDF list. Once parsed, these axioms can themselves be mapped to logical formulae for interpretation by a reasoner.

Example 5.5 *The DL concept* $\exists\texttt{wins}.\top$ *can be expressed structurally as the axiom*

```
ObjectSomeValuesFrom(OPE(:wins) CE(owl:Thing))
```

which maps to the following three RDF triples:

```
_:x a owl:Restriction .
_:x owl:someValuesFrom owl:Thing .
_:x owl:onProperty :wins .
```

Such axioms can always be mapped to RDF triples. The mapping can also be executed in the reverse direction: from RDF to structural axioms and logical formulae. However, the mapping from RDF graphs to OWL 2 structural axioms is only possible for a restricted subset of RDF graphs [57].

Blank nodes are also required to represent RDF lists used in the mapping, *e.g.*, of OWL union classes, intersection classes, enumerations, property chains, complex keys, *etc.* An important aspect here is the locality of blank nodes: if the RDF representation from Example 5.5 is valid in a given graph, it is still valid in an Open World since, *e.g.*, an external document cannot add another value for `owl:onProperty` to `_:x`. This protects axioms from interference with other documents and also ensures that the descriptions of axioms are "closed" within the local document. For this reason, the use of blank nodes as auxiliary nodes for representing axioms is *enforced* by the OWL standard when interpreting RDF graphs using the Direct Semantics [57].

Issues with blank nodes. RDF Semantics-based tools encounter similar issues as for RDFS, where simple entailment is NP-Complete and where, *e.g.*, the OWL 2 RL/RDF ruleset requires use of generalised triples [29]. The Direct Semantics places restrictions on the use of certain features for anonymous individuals; the most prominent example is the `owl:hasKey` feature, which can only be used to infer equivalence between named individuals.

As an aside, OWL contains the vocabulary term `owl:differentFrom`, which can be used to state that two terms refer to different elements of the domain. An interesting case could thus arise if two blank nodes are inferred to be `owl:differentFrom` each other but where one makes the other non-lean. If the RDF graph representing such an ontology was leaned, the OWL semantics of the ontology would then change. However, to the best of our knowledge, such a case is effectively impossible. We speculate that due to the Open World Assumption and a lack of a Unique Name Assumption, it is impossible to construct a case that distinguishes two blank nodes as different-from each other while making one non-lean due to the other. The only counter-example we could find relied on a syntactic relaxation of lists in the OWL RDF-Based Semantics [61]:

```
_:x a owl:AllDifferentFrom ;
  rdf:first _:b1 , _:b2 ;
  rdf:rest _:y .
_:y rdf:first _:b1 , _:b2 ;
  rdf:rest rdf:nil .

:Fred :spouse _:b1 , _:b2 .
:Polygamist owl:equivalentClass
  [ owl:minCardinality 2 ; owl:onProperty :spouse ] .
:Monogamist owl:equivalentClass
  [ owl:cardinality 1 ; owl:onProperty :spouse ] .
```

In this case, `_:b1` is rendered non-lean by `_:b2` and vice-versa, and we now also have the distinction `_:b1 owl:differentFrom _:b2`. If we leaned the graph, we would have that `:Fred` was of type `:Monogamist`. If we did not lean the graph, we would have that `:Fred` was of type `:Polygamist`. However, this case additionally constructs inconsistencies due to having that `_:b1 owl:differentFrom _:b1` and `_:b2 owl:differentFrom _:b2` from the ill-formed list (one such inconsistency would still be preserved if the graph were leaned).

In practice. Rule-based reasoners, which typically support some partial axiomatisation of the RDF-Based Semantics such as DLP [30], pD* [67] or OWL 2 RL/RDF [29], often apply Herbrand interpretations over blank nodes effectively turning the problem of simple entailment into set containment. Conversely, ter Horst proposed pD*sv [67], which contains an entailment rule with an existential blank node in the head to support `owl:someValuesFrom`, but we know of no system supporting this rule.

Conversely, reasoners that implement OWL's Direct Semantics – such as FaCT++ [68], HermiT [53], RacerPro [32], Pellet [63], *etc.* – often support existential semantics and anonymous individuals.

5.4 SPARQL Protocol and RDF Query Language

As discussed in Section 4.2, SPARQL [60] is the standard query language for RDF. The extended SPARQL 1.1 specification has recently become a W3C recommendation [33], adding new features such as SPARQL 1.1 property paths, aggregates, sub-queries, entailment regimes and much more.

Support for blank nodes. With respect to querying over *blank nodes in the dataset*, SPARQL considers blank nodes as constants that are local to the scoping graph they appear in [60]. SPARQL does not rigourously define the notion of a scoping graph, except to state that the same scoping graph is used to generate all results for a query, which leaves open the possibility of blank nodes being shared across different named graphs. SPARQL does however distinguish the scopes of query, results and data, stating that the blank nodes cannot be shared across these scopes. [15]

Example 5.6 *The query:*

```
SELECT DISTINCT ?X
WHERE {
 :Federer :wins ?X .
 ?X :event :Wimbledon .
}
```

issued over the graph depicted in Figure 1 would return $\{\{(\texttt{?X}, \texttt{\_:b1})\}, \{(\texttt{?X}, \texttt{\_:b3})\}\}$ *as distinct solution mappings, here effectively considering blank nodes as constants. Note that the blank node labels are not significant.*

As discussed in Section 4.2, SPARQL engines can also be used to support various tasks over RDF graphs containing existential blank nodes, including simple entailment and leanness checking.

Features requiring blank nodes. SPARQL uses *blank nodes* in the `WHERE` clause of the query to represent *non-distinguishable variables*, *i.e.*, variables that can be arbitrarily bound, but that cannot be returned in a solution mapping. [16] Blank nodes can also be scoped within a query at the level of

[15] This clarification may serve as a corrigendum for our previous paper in which we stated that blank nodes cannot be shared across graphs in SPARQL [47]. This statement is misleading in that although blank nodes cannot be shared across *scoping* graphs, they can be shared across *named* graphs.

[16] In SPARQL, blank nodes are not true existential variables in that they must be bound to a specific term. As such, blank-nodes act analogously to query variables whose substitutions cannot be projected. This will be discussed again later in the context of SPARQL 1.1 Entailment Regimes.

basic graph patterns, which are often (but not always) delimited using braces. Basic graph patterns with blank nodes can always be expressed by replacing blank nodes with fresh query variables that are themselves non-distinguished; however, since SPARQL inherits the same syntax as Turtle, blank nodes do enable shortcuts for querying lists and anonymous nested elements.

A second use for blank-nodes is within `CONSTRUCT` templates, which generate RDF data from solution mappings: a blank node appearing in a query's `CONSTRUCT` clause is replaced by a fresh blank node for each solution mapping in the resulting RDF (similar, in fact, to a Skolem function).

Example 5.7 *This query exemplifies the use of blank-nodes as non-distinguishable variables in the query body and their use in the* `CONSTRUCT` *clause:*

```
CONSTRUCT { _:FedererWins :yearWon ?y ; :event ?e . }
WHERE {
  :Federer :wins _:t .
  _:t :event ?e ; :year ?y .
}
```

requests some *tournaments (*`_:t`*) in which* `:Federer` *won, as well as the year (*`?y`*) and event (*`?e`*) for each. In fact, the term* `_:t` *could be replaced by any arbitrary variable (*e.g.., `?t`*) without affecting the query. The bindings for year and event are then used to generate RDF triples as a result, where based on the graph in Figure 1, the following four triples would be produced:*

```
 _:x :yearWon "2003" ; :event :Wimbledon .
 _:y :yearWon "2009" ; :event :FrenchOpen .
```

whereby a fresh blank node is produced for each result tuple. Again, the blank-node labels are not significant.

Issues with blank nodes. A practical problem posed by blank nodes is that which is often called "round-tripping" whereby a blank node returned in a solution mapping cannot be referenced in a further query. More generally, once a blank node leaves its original scope it can no longer be directly referenced. Consider receiving the result binding (`?X`, `_:b1`) for the query in Example 5.6. One cannot ask a subsequent query for what year "the" tournament labelled `_:b1` took place since the `_:b1` term in the solution mapping no longer has any relation to that in the originating graph: again, the labels need not correspond to the original data. Even if the blank-node label used in the data is known, there is no mechanism to reference that blank node in SPARQL. This issue was discussed by the SPARQL Working Group, but was postponed and left without a solution. [17]

An additional problem is posed by the `COUNT` feature introduced by SPARQL 1.1, which can be used to count result bindings. Keeping aligned with the semantics of SPARQL, `COUNT` enumerates terms in the RDF graph, and not resources in the interpretation. Thus, the `COUNT` feature will consider all URIs as distinct, even though, for example, OWL does not have a Unique Name Assumption: two URIs that may refer to the same real-world "element" of the interpretation will still be counted twice. A similar treatment is applied to blank nodes, which are treated as distinct terms in the graph. However, two graphs that the RDF semantics considers to be equivalent (under simple entailment) may give different results for `COUNT`. For instance, applying `COUNT(?X)` in an analogue of the query in Example 5.6 would answer that `:Federer` won an event at `:Wimbledon` *twice*. Posing the same `COUNT` query over a lean (and thus RDF equivalent [35]) version of Figure 1 would return *once*.

This is, in fact, a specific symptom of an underlying mismatch between the semantics of SPARQL and RDF [4]. Even if $G \models H$ under simple entailment, the results for a SPARQL query over H need not be a subset of G; the `COUNT` feature is an obvious example, but SPARQL also contains features like `NOT EXISTS`, filters, *etc.*, that break this monotonicity.

With respect to the possibility of blank nodes being shared across named graphs, one potential issue occurs with the definition of the `FROM` keyword in SPARQL, which is used to create a new default graph from the content of one or more named graphs such that their combined content can be queried without requiring explicit `GRAPH` clauses in the basic graph pattern. When multiple `FROM` graphs are specified, SPARQL states that the graphs should be *merged* together (as defined in Section 2.1), such that blank nodes in different named graphs are forced to remain distinct in the generated default graph [60, §12.3.2]. This seems contrary to the position that SPARQL allows named graphs to share blank nodes, in which case a *union* would seem preferable. In fact, the SPARQL 1.1 Service Descrip-

[17] http://w3.org/2001/sw/DataAccess/issues#bnodeRef

tion specification [73] takes a different perspective, and provides a vocabulary term for endpoints to state that they are initialised with a default graph that is the union of all named graphs but provides no such term for merging all named graphs.

Example 5.8 *We use an example to demonstrate why the confusion over whether named graphs should be unioned or merged can affect query answering.*

Take two named graphs. The first, named `:g1`*, contains the following triple:*

```
_:b1 :year "2003" .
```

The second, named `:g2`*, contains the following triple:*

```
_:b1 :event :Wimbledon .
```

We can then ask if there was a 2003 Wimbledon event mentioned in the data:

```
ASK { ?s :year "2003" ; :event :Wimbledon . }
```

If the SPARQL dataset is initialised with the union of `:g1` *and* `:g2` *as the default graph, then the answer is* `true`*.*

Consider the same query but where the default dataset is explicitly constructed using `FROM` *clauses:*

```
ASK FROM :g1 FROM :g2
WHERE { ?s :year "2003" ; :event :Wimbledon . }
```

The blank nodes from the two graphs will be kept distinct by the merge and the answer will be `false`*.*

Blank nodes have also caused issues in the definition of SPARQL 1.1 Entailment Regimes [26, 25], which state how various standard entailment regimes (including RDF, RDFS, D, OWL RDF-Based, OWL Direct and RIF Core) can be used to provide "implicit answers" to SPARQL queries. In particular, there has been some debate about how blank nodes should be treated in the context of the OWL Direct Semantics entailment regime [46, 25, 26], mostly due to the limited use of blank nodes as non-distinguished variables such that they must match a specific term in the graph (or its entailments) rather than being satisfied when something is implicitly known to exist.[18] The reason for debate is best illustrated with an example.

[18] See the mail-thread starting at http://lists.w3.org/Archives/Public/public-rdf-dawg/2010OctDec/0318.html.

Example 5.9 *Take the simple query:*

```
SELECT ?winner
WHERE { ?winner :wins _:something . }
```

As per Figure 1, consider a graph containing the triples:

```
:Federer :wins _:b1 , _:b2 , _:b3 .
```

The above query will return `:Federer` *as an answer. However, if instead the RDF graph encoded the DL axiom* $\{\,\texttt{:Federer}\,\} \sqsubseteq \exists\texttt{wins}.\top$*, which can be interpreted under OWL semantics as stating that* `:Federer` *did win something, the answer set will be empty. This is because the term* `_:something` *is expected to match a specific RDF term in the graph (or its entailments), and does not behave as a true existential variable. OWL Direct Semantics would not make existential knowledge explicit using blank nodes, but rather using concepts, where a more complete query could be written as:*

```
SELECT ?winner
WHERE {
 { ?winner :wins _:something . }
  UNION
 { ?winner rdf:type _:hasWin .
   _:hasWin rdf:type owl:Restriction .
   _:hasWin owl:someValuesFrom owl:Thing .
   _:hasWin owl:onProperty :wins . }
}
```

which would cover both representations of existential knowledge. (If the OWL Direct-Semantics Entailment Regime were enabled, only the latter part of the `UNION` *would be necessary: the latter axiom would be entailed from any triples matching the former part.)*

Blank nodes in SPARQL basic graph patterns are *not* considered to be true existential variables as this would change the core meaning of blank nodes in SPARQL, leading to different behaviours across different entailment regimes. When querying for the existence of an implicit element under the OWL Direct Semantics entailment regime, it is thus necessary to query for existential concepts as illustrated in the previous example.

In practice. Implementations generally follow the SPARQL specification in their treatment of blank nodes. However, to support "round-tripping" of blank nodes, SPARQL engines often implement custom syntaxes that allow blank nodes to be referenced outside of their original scoping graph (colloquially known as "Skolemisation" where, per Section 3.3, the existential variable is replaced with a fresh constant). For example, ARQ[19] is a commonly (re)used SPARQL Java library and it supports a non-standard `<_:b1>` style syntax for terms in queries, indicating that the term can only be bound by a blank node labelled "`b1`" in the data. Other engines supporting similar syntax include Garlik and RDFLib. Virtuoso[20] supports the `<nodeID://b1>` syntax with similar purpose, but where blank nodes are only externalised in this syntax and (perhaps unusually) where the built-in SPARQL function `isBlank(<nodeID://b1>)` evaluates as `true`. Another solution proposed to the SPARQL Working Group was to specify a `USING BNODEREF` key-phrase before the `WHERE` clause to indicate that blank nodes in the respective query should be interpreted as constants.[21] However, this was not included for SPARQL 1.1.

5.5 Rule Interchange Format (RIF)

Both RDFS and OWL are associated with various sets of entailment rules that support some subset of the semantics of the respective language. However, neither standard supports the idea of user-defined rules. Instead, the recently standardised Rule Interchange Format (RIF) can be used. RIF aims to offer a common means to interchange rules across the Web, and thus goes beyond RDF in scope. Most relevant for RDF are the RIF Basic Logic Dialect (BLD) [13] and RIF Core [12]. RIF BLD allows for serialising and exchanging domain-specific entailment rules and can be applied for RDF data [13]. RIF Core is a terse syntactic subset of RIF BLD [12].

Example 5.10 *The RDFS-entailment rule* rdfs2 *from Table 1 could be written in RIF's presentation syntax [13] as follows:*

```
Forall ?u ?x ?a ?y (?u [ rdf:type ->  ?x] :-
  And( ?a [rdfs:domain -> ?x ]  ?u [ ?a -> ?y ] ) )
```

RIF's presentation syntax borrows from F-logic [44], encoding RDF triples (`s p o`*) as frames* `s[p->o]` *and using '*`:-`*' for encoding (rule) implication.*

Aside from the RDF(S) entailment rules, any arbitrary Horn rules over RDF, optionally with built-in calls in the rule body, can be expressed in RIF.

Support for blank nodes. RIF does not directly support blank nodes. Quoting from the standard:

> *"RIF does not have a notion corresponding exactly to RDF blank nodes. RIF local symbols, written* `_symbolname`*, have some commonality with blank nodes;"*
>
> —**[23, §2]**

In other words, although RDF graphs with blank nodes cannot be directly expressed in RIF, "local symbols" are supported that are only visible within the scope of a RIF document.

While RIF allows existentially quantified variables in rule bodies, existential quantification in rule heads (and thus in factual statements such as in RDF triples) is not supported. Inspired by support for existentials in Description Logics, there has been some recent work on likewise extending certain guarded fragments of Horn rules with existentials while still preserving decidability of basic reasoning tasks [17]; however, these proposals have not yet made it into the RIF standard.

Example 5.11 *For instance, a "rule" expressing that a member of the class* `:Winner` *has won "something" would need an existential in the rule head.*

```
Forall ?X ( Exists ?Y ( ?X [ :wins ->  ?Y ] :-
                ?X [ rdf:type -> :Winner ] ) )
```

While such rules would not be expressible in RIF BLD (which disallows existentials in rule heads), the same can be modelled in DL (and likewise in OWL) easily: $\texttt{Winner} \sqsubseteq \exists\texttt{wins}.\top$.

While blank nodes and existentials in rule heads are not supported in RIF natively, the Skolemisation of rules with existentials in rule heads could be expressed in RIF BLD, which supports full function symbols. That is, a Skolemised form of the existential rule from Example 5.11 could be expressed as follows:

```
Forall ?X ( ?X[ :wins -> sk(?X) ]  :-
         ?X [ rdf:type -> :Winner ] )
```

[19] http://jena.sourceforge.net/ARQ/
[20] http://virtuoso.openlinksw.com/
[21] http://www.w3.org/2009/sparql/wiki/Feature:BlankNodeRefs

The combination of arbitrary RDF graphs (including blank nodes) and RIF rules is defined in [23], which combines RDF interpretations and interpretations of a RIF ruleset. To check whether an RDF graph G and a RIF ruleset R entails an RDF graph G', it is sufficient to encode a Skolemised version of G as a set of (skolemized) RIF facts $sk(G)$, and to subsequently encode G' as a query (with existentials) over $R \cup sk(G)$ (see [23, §9.1]). As such, simple entailment can also be supported in RIF.

Example 5.12 *As a continuation of Example 4.2, checking whether the graph G from Fig. 1 simple-entails H can also be tested in RIF. First we encode G as a set of RIF facts $sk(G)$ using Skolemization; in this example, we use a RIF local constant* `_x` *to encode a blank node* `_:x` *with the same label.*

```
Document ( Group (
 :Federer [ rdf:type -> :TennisPlayer ]
 :Federer [ :name -> "Roger Federer" ]
 :Federer [ :wins -> _b1 ]
 :Federer [ :wins -> _b2 ]
 :Federer [ :wins -> _b3 ]
 ... ) )
```

The entailment graph H from Example 4.2 would then be translated into the following existential RIF formula, where blank nodes are translated to variables instead of local constants.

```
Exists ?player ?ev1 ?ev2
(And ( ?player [ rdf:type -> :TennisPlayer]
     ?player [ :wins -> ?ev1 ]
     ?player [ :wins -> ?ev2 ]
     ?ev1 [ :year -> 2003 ]
     ?ev2 [ :year -> 2009 ] ) )
```

This "query" can then be issued against $sk(G)$ (in combination with a RIF ruleset R if provided) to see if the entailment holds.

This method of performing simple entailment using RIF is analogous to that presented in Example 4.2 for SPARQL.

Features requiring blank nodes. As with the OWL mapping, RIF rules and formulas can be serialised as RDF, where the mapping again makes heavy use of blank nodes [34]. Since the syntax is even more verbose than the encoding of OWL axioms into RDF triples, we refer the reader to, *e.g.*, [34, §11] for a concrete example rather than including one herein.

In practice. To the best of our knowledge, there have been few instances of RIF being adopted in the context of RDF in practice. Among the implementations listed at the RIF Working Group's implementation page [22], SILK [23] is probably the most actively developed tool, but does not report full support of RDF compatibility (as defined in [23]), nor does it mention any issues with blank nodes explicitly. FuXi is the only system in the list that mentions explicit support for RDF – in the form of OWL 2 RL in RIF – but details are not published; from the web-page on FuXi's semantics, it is not clear if any issues with blank nodes were encountered, though at the time of writing, there is a brief mention of blank nodes appearing in the head of a rule. [24] Another (unlisted) academic RIF implementation with RDF support has been reported by Marano *et al.* [49], but is not actively maintained at the time of writing of this paper.

5.6 RDB2RDF

Given an increasing interest in publishing relational data as RDF, the RDB2RDF W3C Working Group was tasked with standardising a language for mapping relational data into RDF. As the result of the activity of this group, two languages were proposed: a direct mapping [3] that translates a relational database into RDF without any input about the transformation process from the user, and a general mapping language [21] where users can specify their own rules for translating a relational database into RDF. In what follows, we show how blank nodes are used in the direct mapping [3].

Support for blank nodes. The input of the direct mapping is a relational database, including the schema of the relations being translated and the set of keys and foreign keys defined over them. The output of this language is an RDF graph that may contain blank nodes.

Features requiring blank nodes. The RDF graph generated in the translation process identifies each tuple in the source relational database by means of a URI. If the tuple contains a primary key, then this URI is based on the value of the primary

[22] http://www.w3.org/2005/rules/wiki/Implementations
[23] http://silk.semwebcentral.org/
[24] https://code.google.com/p/fuxi/wiki/FuXiSemantics

key. If the tuple does not contain such a constraint, then a blank node is used to identify it in the generated RDF graph [3].

Example 5.13 *Assume that the following table* Tweets *stores information about tweets in Twitter [3]:*

Tweets	ID	Text
	1	*I like RDF*
	1	*I like blank nodes*

Each row of the table stores a tweet (Text) *from a person with identifier* ID. *This table does not have a primary key, thus each of its tuples is identified by a blank node when translated by the direct mapping. More precisely, in this case the direct mapping produces the following triples:*

```
_:a rdf:type <Tweets> .
_:a <Tweets#ID> "1" .
_:a <Tweets#Text> "I like RDF" .

_:b rdf:type <Tweets> .
_:b <Tweets#ID> "1" .
_:b <Tweets#Text> "I like blank nodes" .
```

In these triples, URI `<Tweets>` *is generated by concatenating some base URI (for example,* `http://example.org/`*) with the string* `Tweets`*, while URIs* `<Tweets#ID>` *and* `<Tweets#Text>` *are generated by concatenating the base URI with the strings* `Tweets#ID` *and* `Tweets#Text`*.*

Issues with blank nodes. In the mapping process, blank nodes are used as identifiers of tuples without primary keys [3], and as such, two of these blank nodes should not be considered as having the same value. Thus, the existential semantics of blank nodes in RDF is not appropriate for this use.

Example 5.14 *Continuing with Example 5.13, now assume that table* Tweets *contains repeated tuples:*

Tweets	ID	Text
	1	*I like RDF*
	1	*I like blank nodes*
	1	*I like RDF*

In this case, the direct mapping produces the following triples:

```
_:a rdf:type <Tweets> .
_:a <Tweets#ID> "1" .
_:a <Tweets#Text> "I like RDF" .

_:b rdf:type <Tweets> .
_:b <Tweets#ID> "1" .
_:b <Tweets#Text> "I like blank nodes" .

_:c rdf:type <Tweets> .
_:c <Tweets#ID> "1" .
_:c <Tweets#Text> "I like RDF" .
```

The generated RDF graph G is not lean: given the map μ such that $\mu($`_:a`*$) = \mu($*`_:c`*$) =$* `_:a` *and $\mu($*`_:b`*$) =$* `_:b`*, we have that $\mu(G)$ is a proper subgraph of G. However, the blank nodes* `_:a` *and* `_:c` *are generated to represent distinct tuples in the table* Tweets*, and as such, they should not be considered as having the same value.*

In practice. The direct mapping has been implemented in several systems: D2RQ[25], RDF-RDB2RDF[26], SWObjects dm-materialize[27], XSPARQL [10], Ultrawrap [62] and db2triples[28]. In all these systems, blank nodes are used as identifiers when translating a relation without a primary key, so the existential semantics of blank nodes in RDF is not appropriate for the RDF graphs generated by any of these systems.

5.7 Summary of standards

We have looked at blank nodes in the context of all the Web standards directly related to RDF, discussing how blank nodes are supported, which features rely on them, what issues have arisen surrounding them, and how implementations handle them. In the various RDF syntaxes, blank nodes enable various syntactic shortcuts and relax the requirement to assign a global URI to everything. In RDFS and OWL, blank nodes can be interpreted as existential variables, although ground semantics are often applied in practice. In OWL, RIF and RDB2RDF, blank nodes are used as unnamed nodes when mapping structural information to RDF. In SPARQL, blank nodes are interpreted in queries as non-distinguished variables, or as Skolem functions when given in `CONSTRUCT` clauses.

The two primary “semantic mismatches” we identify with respect to blank nodes involve SPARQL

[25] http://d2rq.org/
[26] https://metacpan.org/release/RDF-RDB2RDF
[27] http://swobjects.svn.sourceforge.net
[28] https://github.com/antidot/db2triples

and the direct mapping of RDB2RDF. In SPARQL, two non-isomorphic RDF graphs that simple-entail each other can return different answers: thus, for example, leaning an RDF graph can change the SPARQL answers derived from it for certain queries. In the RDB2RDF direct mapping, identical source tuples in the relational table will yield non-lean blank nodes in the output RDF graph, but each such blank node represents the existence of a tuple in the source and should not be considered "redundant".

6 Blank nodes in publishing

In this section, we survey the use of blank nodes in RDF data published on the Web. The recent growth in RDF Web data is thanks largely to the pragmatic influence of the Linked Data community [11, 37]. Linked Data guidelines are unequivocal on the subject of blank node usage. In the recent book "Linked Data: Evolving the Web into a Global Data Space" [37], Heath and Bizer make their only reference to blank nodes in the section entitled "RDF Features Best Avoided in the Linked Data Context", as follows:

> *"The scope of blank nodes is limited to the document in which they appear,* [...] *reducing the potential for interlinking between different Linked Data sources.* [...] *it becomes much more difficult to merge data from different sources when blank nodes are used,* [...] *all resources in a data set should be named using URI references."*
>
> —**[37, §2.4.1]**

With this (recent) guideline discouraging blank nodes in mind, we now provide an empirical study of blank nodes in published data on the Web.

Our analyses are based on the Billion Triple Challenge 2012 (BTC–2012) corpus, which represents a large sample of RDF published on the Web. Using conventions for a SPARQL dataset, we may denote this corpus as $\{M, (u_1, G_1), \ldots, (u_n, G_n)\}$, where each G_i is an RDF graph referring to an individual Web document, u_i is the URL from which that RDF document was retrieved (with `200 Okay`), and M is a (virtual) default graph composed of the RDF merge of all $G_1, \ldots, G_n$. The sets of blank nodes in each graph $G_1, \ldots, G_n$ are pairwise disjoint.

The dataset is represented on-disk as a list of *quadruples* Q, written in the N-Quads syntax. Quadruples extend RDF triples to add a fourth element containing the graph URI. Thus we can say that $Q = \bigcup_{1 \leq i \leq n} G_i \times u_i$. Letting π denote a projection operator, we can also say $M = \pi_{s,p,o}(Q)$. In the following, we use the dataset notation or the quadruple notation equivalently, as convenient.

The BTC–2012 corpus consists of 1.230 billion unique quadruples extracted from 8.373 million RDF documents, collected through a crawl conducted in May 2012. [29] The corpus consists of data collected from 829 different *pay-level domains*, which are direct subdomains of either top-level domains (such as `dbpedia.org`), or country code second-level domains (such as `bbc.co.uk`). Henceforth, when we mention domain, we thus refer to a PLD (unless otherwise stated).

We begin this section by looking generally at the prevalence of blank nodes in published data (§6.1). We then look at the morphology of blank nodes in such data, looking at how blank-nodes are interconnected and measuring the treewidth of blank-node structures embedded in RDF Web documents to get an idea of how difficult simple entailment and leaning are in practice (§6.2). In Section 7, we continue the discussion by looking at the prevalence of non-lean RDF data in our sample of Web data.

6.1 *Prevalence of blank nodes in Web data*

First, we looked to measure the raw prevalence of blank nodes and their use in real-world RDF data:

(i) Of the 1.230 billion unique quadruples in the BTC–2012 corpus, 274.194 million (22.3%) had a blank node in the subject position and 94.211 million (7.7%) had a blank node in the object position.

(ii) Of the 8.373 million documents comprising the corpus, 3.758 million (44.9%) featured at least one blank node.

(iii) Of the 341.733 million unique RDF terms (URIs, literals and blank nodes) appearing in the data, 88.677 million (25.9%) were blank nodes. [30]

[29] http://km.aifb.kit.edu/projects/btc-2012/. We pre-filtered the data to remove HTTP header information output by the crawler as this is not "native" RDF. Note that no BTC–2013 dataset has been made available: BTC–2012 is the most recent edition.

[30] 146.871 million (43.0%) were literals and 106.185 million (31.1%) were URIs.

Table 2. Top 25 publishers of blank nodes in our corpus

№	Domain	BNodes	%BNodes	LOD?
1	data.gov.uk	54,898,287	27.39	✓
2	freebase.com	14,918,969	31.95	✓
3	livejournal.com	11,757,431	56.97	✗
4	legislation.gov.uk	3,310,772	46.45	✓
5	ontologycentral.com	1,907,525	79.47	✓
6	vu.nl	658,538	37.27	✓
7	neuinfo.org	279,935	42.44	✗
8	opera.com	233,578	6.89	✗
9	geovocab.org	210,263	67.45	✗
10	loc.gov	147,997	10.95	✓
11	bbc.co.uk	94,077	16.41	✓
12	bibsonomy.org	79,543	41.17	✗
13	codehaus.org	48,943	90.52	✗
14	opencalais.com	28,873	41.32	✓
15	vocab.org	14,867	75.33	✗
16	w3.org	11,141	9.47	✓
17	174.129.12.140	8,136	54.07	✗
18	soton.ac.uk	7,970	2.76	✓
19	southampton.ac.uk	4,420	9.40	✗
20	fao.org	4,183	2.59	✓
21	identi.ca	4,098	0.34	✗
22	semanticweb.org	3,932	2.50	✓
23	mondeca.com	3,849	49.74	✗
24	lehigh.edu	3,781	18.72	✗
25	openlylocal.com	3,431	13.09	✓

(iv) Of these 88.677 million unique blank nodes, 88.673 million (99.995%) appeared in the subject position of at least one triple, and 86.596 million (97.7%) appeared in the object position of at least one triple.

Thus, we can already surmise that blank nodes are prevalent in RDF data on the Web. Furthermore, we see that almost all blank nodes appear at least once in both the subject and object position, but occur most prevalently in the former: on average, a given blank node appears as the subject of 3.09 triples, and the object of 1.06 triples. Relatedly, in the various RDF syntaxes, blank nodes can appear multiple times in the subject position without the need for explicit labelling, but can only appear once in the object position without such labelling.

Next we look at the domains publishing blank nodes: of the 829 pay-level domains contributing to the corpus, 549 (66.2%) feature use of at least one blank node in their published data. Table 2 shows the top 25 domains exporting blank nodes in the corpus.[31] The "%BNodes" column indicates the percentage of all unique terms appearing in the domain's corpus which are blank nodes (*i.e.*, $\frac{|\mathrm{voc}(M_d)|}{|\mathrm{terms}(M_d)|}$ represented as a percentage for M_d the RDF merge of all documents from that domain d in the corpus). The "LOD?" column indicates whether the domain is featured in the LOD cloud: we extracted the list of domains mentioned in the CKAN/LOD metadata repository, where of the 829 domains contributing to the BTC–2012 dataset, 78 (9.4%) were mentioned in the LOD repository (see [42] for related discussion comparing coverage of the BTC–2011 and the LOD cloud).

Summarising the use of blank nodes on a domain level, of the 829 domains contributing to our corpus, 280 (33.8%) did not publish any blank nodes. The mean percentage of unique terms that were blank nodes across all domains—*i.e.*, the mean of %BNodes for all domains—was 7.6% (±12.3 pp.), indicating that although a small number of high-volume domains publish many blank nodes (*cf.* Table 2), many other domains publish blank nodes much more infrequently. The analogous mean figures including only those domains appearing in the LOD cloud diagram was (surprisingly) 26.4% (±22.5 pp.) and excluding LOD domains was 5.7% (±8.6 pp.).

6.2 Structure of blank nodes in Web data

As per Section 3.1, the problem of checking the simple entailment $G \models H$ is made difficult by the presence of connected blank nodes in H—*i.e.*, two blank nodes appearing in the same triple as defined by blank(H) in Section 4—forming cycles, and in particular, having high treewidth. To get an overview of the structure of the connected blank nodes in the BTC–2012 corpus, for each document G_i contained within, we extracted blank(G_i) and separated out the non-singleton connected components (henceforth simply called "components") thereof using a Union-Find algorithm [66]: recall that given the locality of blank nodes, they can only be linked within the given document.

In terms of the connectedness of blank-nodes within documents, we observed the following:

(i) Of the 3.758 million documents containing at least one blank node, 1.477 million (39.3%)

[31] In the older corpus analysed in the previous version of this paper [47], the hi5.com domain was found to export massive volumes of blank nodes. This domain no longer exports RDF.

contained connected blank-nodes: *i.e.*, contained at least one triple with two unique blank nodes, giving a non-empty blank(G_i).

(ii) Across these documents, we found a total of 3.334 million components, with an average of 2.26 components for a document containing some connected blank nodes.

(iii) Taken together, these components contained 62.938 million unique blank nodes, which implies that 71.0% of all unique blank nodes were connected, and each component contained on average 18.8 unique blank nodes.

Hence, we see that the majority of blank nodes are connected to (appear in the same triple as) at least one other blank node. If connected, a blank node connects to approximately eighteen other blank nodes, on average.

We are now interested in the nature of these connections between blank nodes. In Figure 2 we plot the distribution of the 62.938 million connected blank-nodes for different values of in-degree and out-degree: here, we consider a variation of blank(G_i) which takes *directed* edges from subject blank nodes to object blank nodes in the same triple (and again disregards loops).

(i) The graph shows that blank nodes occasionally have much higher values for out-degree relative to in-degree. Though the highest in-degree observed was 17, the analogous figure for out-degree was 1,320.

(ii) The outliers observable around the 1,000 mark are due to FOAF social-data exporters that implement a limit on the number of connections a user can have.

(iii) Not shown in the (log/log) graph are the number of connected blank-nodes with in-degree or out-degree of zero, which, resp., was 2.241 million (3.5%) and 15.212 million (4.1%).

(iv) We see that the vast majority of blank nodes have an in-degree of 0 or 1 (98.0%) and an out-degree of 0 or 1 (95.6%).

The distribution of in-degree and out-degree suggests again that blank nodes tend to "fan out" from subject to object, and not vice-versa. This again could be attributed to the tree-like layout of popular RDF syntaxes and the role blank-nodes play in them. Conversely, the low number of blank nodes with an in-degree of zero—which are candidates to form the root of a polytree—sets an upper-bound on the percentage of subject-to-object polytrees represented by the 3.334 million components at 67.1%.

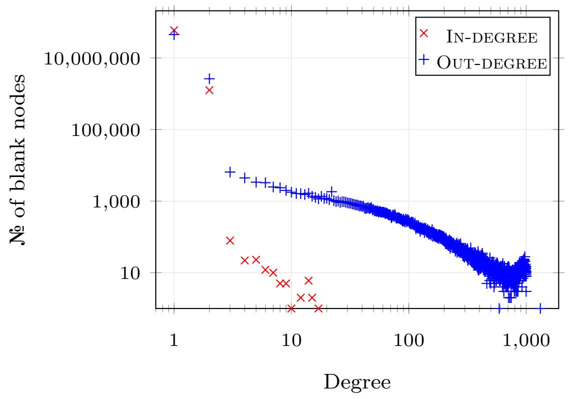


Fig. 2. Distribution of degree of connected blank nodes in directed blank graphs (log/log)

Digging into the structure of these components in more detail, recall from Section 4 that checking the simple entailment $G \models H$ has the upper bound $\mathcal{O}(n^2 + mn^{2k})$, where k is one plus the *treewidth* of the blank node structure blank(H) [58]. All graphs whose treewidth is greater than one are cyclic, and the higher the treewidth, the higher the cyclicity and the harder entailment becomes. As such, and as stated previously, simple entailment checking only becomes difficult when blank nodes form cycles:

> "[...] *in practice, an RDF graph contains rarely blank nodes, and even less blank triples.*[32] *Hence, most of the real RDF graphs are acyclic or have low treewidth such as 2, and the entailment can be tested efficiently* [...]."
>
> —**[58, §4]**

To cross-check this claim, we checked the treewidth of all 3.334 million (undirected) components using the QuickBB algorithm [27], implemented in the LibTW package [71]. The distribution of values is given in Table 3. Notably, 62.3% of the components are acyclical with a treewidth of one, and supporting the above claim, we found that only 19 components have a treewidth of three or more. A total of 17 domains published blank-node components with cycles, 4 of which published components with treewidth greater than two.

The two components with the highest treewidth (of six) were found in one document.[33] To give an impression of the "complexity" of such a graph, Fig-

[32] In the terminology of [58], a *blank triple* is an element of $\mathbf{B} \times \mathbf{U} \times \mathbf{B}$.
[33] http://smiy.sourceforge.net/prv/rdf/spin_-_prv_rules.owl

Table 3. The distribution of treewidths for blank node components in our data ('∼' indicates a negligible percentage)

Treewidth	Components	% Components
1	2,082,921	62.3
2	1,258,774	37.7
3	11	∼
4	4	∼
5	2	∼
6	2	∼

ure 3 draws one such component, where vertexes are blank nodes and edges represent blank triples.[34] A minimal sub-graph with treewidth of 6 is highlighted in bold: removing any edge from this subgraph would reduce its treewidth (this minimal subgraph is not necessarily unique). From this example, we can state an empirical upper bound of $O(n^2 + mn^{2\times(6+1)}) = O(n^2 + mn^{14})$ for simple entailment within this large sample of real-world graphs.

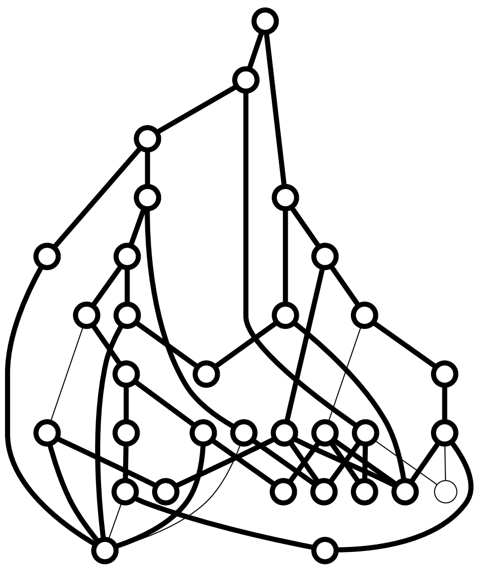


Fig. 3. The blank-node component with the highest treewidth in our data (treewidth of 6, 32 vertices, 47 edges). A minimal sub-graph with treewidth 6 is highlighted in bold.

We conclude that the majority of documents surveyed contain acyclical blank node structures. Furthermore, with a low average in-degree of 1.07, we conclude that blank nodes mostly tend to form directed trees from subject to object. However, unlike observations for previous datasets [47], we see a significant number of blank-node components (37.7%) containing cycles. Of the 1,258,774 with a treewidth of 2, we found that 1,257,229 of these (99.9%) originated from a single domain, `data.gov.uk`, which is in fact the largest producer of blank nodes in our data (*cf.* Table 2). Aside from this one domain, the vast majority of blank nodes form acylical graph structures.

6.3 Survey of publishers

To further understand how blank nodes are used, we made a simple poll asking interested parties what is their intended meaning when they publish triples with blank nodes. We sent the poll to two W3C's public mailing lists, Semantic Web and Linking Open Data[35], and got 88 responses. In order to identify active publishers, we asked participants to indicate which datasets appearing in the LOD cloud (if any) they have contributed to, where 10 publishers claimed contributions to a current LOD dataset.

At the top of the web page, before the questions, we explicitly stated that *"... the poll is trying to determine what you intend when you publish blank nodes. It is not a quiz on RDF Semantics. There is no correct answer"*. We deliberately kept the survey terse, asking two simple questions.

The options and results for both questions are presented in Table 4, broken down by all responses (88) and responses from publishers involved in a LOD dataset (10).

In the first question, we asked participants in which scenarios they would publish a graph containing the following triple: "`:John :telephone _:b .`". We chose the `:telephone` predicate as an abstract example that could be read as having a literal or URI value. Participants were told to select zero or more options which would cover all reason(s) why they might publish such a triple.

In the second question, we asked participants to select zero or more scenarios in which they would publish a graph containing (only) the two triples "`:John :telephone _:b1, _:b2 .`".

[34] In the earlier version of this paper, we found a variety of documents with high-treewidth blank-node structures on the `rdfabout.com` site [47]. The highest treewidth found was seven for a component of 451 blank nodes with 887 edges. However, these documents are no longer available on the Web and are not found in our updated data.

[35] mailto:semantic-web@w3.org and mailto:public-lod@w3.org respectively.

Table 4. Details and results of the survey of two mailing lists (`public-lod@w3.org` and `semantic-web@w3.org`)

Question 1: *When would you publish the triple "*`:John :telephone _:b .`*" alone?*

	Option	Responses	
		All (88)	LOD (10)
1a	*John has a tel. number whose value is unknown.*	46.4%	20.0%
1b	*John has a tel. number but its value is hidden,* e.g., *for privacy.*	23.9%	0.0%
1c	*John has no tel. number.*	0.0%	0.0%
1d	*John may or may not have a tel. number.*	2.3%	0.0%
1e	*John's number should not be externally referenced.*	18.2%	0.0%
1f	*I do not want to mint a URI for the tel. number.*	37.5%	30.0%
1g	*I would not publish such a triple.*	41.0%	70.0%

Question 2: *When would you publish the triples "*`:John :telephone _:b1, _:b2 .`*" alone?*

	Option	Responses	
		All (88)	LOD (10)
2a	*John does not have a tel. number.*	0.0%	0.0%
2b	*John may not have a tel. number.*	0.0%	0.0%
2c	*John has at least one tel. number.*	23.9%	0.0%
2d	*John has two different tel. numbers.*	23.9%	10.0%
2e	*John has at least two different tel. numbers.*	35.2%	40.0%
2f	*I would not publish such triples.*	50.0%	70.0%

The poll had an optional section for comments; a number of criticisms (∼12) were raised about the `:telephone` example used and the restriction of having only one or two triples in the graph. This leaves ambiguity as to whether the participant would publish blank nodes at all (which was the intended effect) or would not publish that specific example (an unintended effect). Thus, we note that answers 1g and 2f might be over-represented. Also, one concern was raised about the "right" semantics of blank nodes in RDF (namely, that John has a telephone number, without saying anything about our knowledge of the number) not being an alternative, but we felt that with respect to the *intent* of the publisher, this was covered by option 1b.

Despite possible limitations of the poll, we can still see that the majority of intent with which blank nodes are used is compatible with the (more general) semantics of blank nodes. Noting that the standard semantics for both graphs is simply "John has at least one telephone number", we see that only non-existent/non-applicable options 1c, 1d, 2a and 2b contradict or are more general than the standard semantics. Of these, only 1d was selected and only by 2.3% of participants.

The intent represented by the other options are compatible with the standard semantics, either being equivalent or being more specific. Taking 1g minus 2f, 9% of all participants would not publish *specifically* non-lean blank nodes. Almost half would publish the triple to represent an unknown value (compatible with existential semantics). Over one third would use blank nodes simply to avoid minting URIs. Of the 10 LOD publishers, 7 would not publish such examples, 2 would publish blank nodes to represent unknown values, *etc.*

In the second question, with respect to the intent to more specifically state that John has (at least) two telephone numbers, even using URIs, this is not possible within RDF(S) due to the Open World Assumption and the lack of a Unique Name Assumption.[36] A publisher can only state that John has at least zero or at least one telephone number(s). However, there is a slight nuance between the URI case and the blank node case. As opposed to the URI case, under the standard RDF semantics, additional blank nodes would be considered redundant and could validly be removed by a leaning operation. As stated before, this could affect, for example, the results of SPARQL queries over the data.[37]

Although blank nodes are a divisive issue, our survey results show that in ∼97.7% of cases, the intent

[36] It would be possible using literals or using OWL semantics.
[37] One could subjectively argue that this is a problem with SPARQL or a problem with blank nodes or a problem with RDF or a problem with the publishers' intent. There is no clear answer.

with which blank nodes are published is not incompatible with their semantics. However, publishers often have more specific intents (such as representing the existence of multiple real-world relationships), which cannot be captured by existential blank nodes or more generally by the semantics of RDF(S) alone.

7 (Non-)lean blank nodes in Web data

Our previous analysis has shown that blank nodes are prevalent in real-world data, that they often form trees and contain low treewidth and that the intent of publishers when using blank nodes can vary. In this section, we explore further the prevalence of lean versus non-lean blank nodes in real-world data (see Definition 2.6). Based on our BTC–2012 corpus, we analyse how often non-lean blank nodes occur in real-world data, where they occur, and in what form. We are interested in non-lean blank nodes across documents as well as within documents: *i.e.*, with respect to the RDF merge of our entire corpus (M) and not just within individual documents (G_i).

In general, leaning requires finding a (non-trivial) homomorphism within or between blank-node components as per the previous definition of a map μ. Homomorphisms are also at the core of evaluating SPARQL BGPs and in Section 4, we discussed how (non-)leanness can be evaluated using SPARQL queries. By viewing leanness-checking from the perspective of surrogate SPARQL queries, one gets a better sense of the challenges faced when trying to classify blank nodes as lean or non-lean in a corpus such as the BTC–2012 dataset, and the vast amount of computation that is involved.

Using this SPARQL analogy, we can state that the problem of classifying (non-)lean blank nodes across the entire BTC–2012 corpus is equivalent to running 29.081 million SPARQL queries (for all components, including singletons), 1.258 million of which contain cycles, with an average of 3.04 variables and 10.54 conjunctive patterns per query, with the largest query containing 4,570 variables and 9,155 conjunctive patterns, all to be evaulated over a dataset consisting of 1.230 billion quadruples. For the set of 29.081 million SPARQL queries that would be used to represent these components, the full distribution of variables and patterns per query is depicted in Figure 4, which gives an impression of the scale of the problem faced, where there would be a non-trivial amount of queries with thousands of query patterns and query variables.[38]

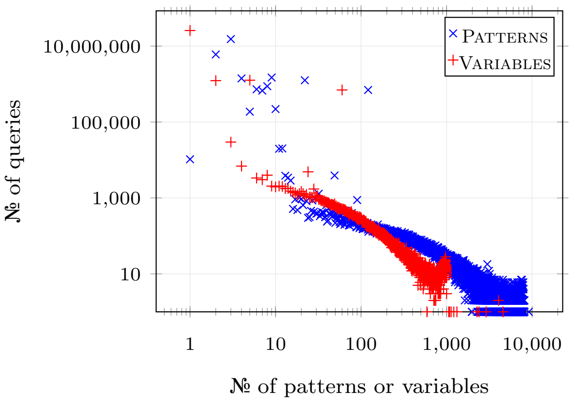


Fig. 4. Distribution of patterns and variables in the SPARQL queries representing blank node components in the data

Given the sheer scale of the problem, we do not use a SPARQL engine to determine which blank nodes are lean or non-lean in the corpus: there is simply no precedent for a SPARQL engine being able to support such a computationally intensive workload of extremely large queries. Instead of trying to build surrogate queries and answer them all individually as fast as possible, we heavily batch-process the data, trying to partially answer many queries at once. We rely heavily on sorts, scans and merge-joins. In the design of our algorithms, we rely on certain characteristics of the data, for example, that it consists of millions of "small" documents within which blank nodes are locally scoped. Our core approach is to reduce, insofar as possible and as early as possible, the problem space by iteratively identifying lean blank-nodes (that cannot "match" anything and thus no longer have to be considered as "variables") and factoring them out of the computation. We thus have a multi-phase operation, which increases in complexity for a smaller volume of data as more and more lean blank nodes are identified.

All processing is run on a single machine with a Quad-code Intel® Xeon® E5606 @2.13GHz, 64GB RAM, and an SATA hard-drive. We do not focus primarily on optimising runtimes, but given the challenging nature of the computation involved, we rather settle for computing the results on the available hardware within a reasonable time-frame and

[38] This distribution is equivalent to the distribution of the number of blank nodes per component (variables) and the number of triples in which those blank nodes appear per component (patterns).

presenting the results. For example, although much of the processing—primarily involving sorts, scans and merge-joins—could be distributed over multiple machines, or even parallelised to take advantage of multiple cores, we currently implement single-threaded programs. Code is developed in Java using the NxParser library [39] for processing and sorting data in the N-Quads format. [40] Our methods are all based on tuple-at-a-time iterators built over on-disk GZipped files of the line-delimited syntaxes similar to N-Quads or N-Triples. Source code is available from http://sw.deri.org/svn/sw/2011/06/bnodes/.

In preparation for the following processing, we dictionary encode the blank nodes of the BTC–2012 dataset, assigning unique integer labels in a manner consistent with an RDF merge of the individual documents. This process helps reduce later memory requirements: integers have less overhead than strings in-memory. To perform the encoding, for each document in the corpus, we build an in-memory dictionary that maps the original blank node label to an integer identifier based on a running global count of all blank nodes encountered. The blank nodes are rewritten and the corresponding statements are output. The encoding took 3.5 h to run.

Before we continue, we also formalise some core concepts that will be used frequently in the following sections.

When we classify a blank node as lean, we consider it thenceforth as a Skolem constant (or simply a "Skolem"). For this, we reserve the set $\mathbf{S}$ for Skolemised blank nodes that are *known* to be lean. Blank nodes that have not been classified as lean (*i.e.*, that have been classified as non-lean or remain unclassified) are contained in the usual set $\mathbf{B}$, where $\mathbf{B}$ and $\mathbf{S}$ are considered disjoint. We denote by $\mathrm{sk} : \mathbf{B} \to \mathbf{S}$ the one-to-one Skolemisation function for blank nodes. We likewise relax the notion of RDF graphs to allow Skolems in positions where blank nodes are allowed.

Our methods frequently rely on the notion of an *edge* or set of edges for an RDF term, which we now formally define.

Definition 7.1 *Let G be an RDF graph and let $x \in \mathrm{terms}(G)$ be an RDF term in that graph. We denote by $\mathrm{out}(G,x) := \{(p,o) \mid (x,p,o) \in G\}$ the* outward labelled edges *(or simply "out-edges") of x in G. We denote by $\mathrm{in}(G,x) := \{(p,s) \mid (s,p,x) \in G\}$ the analogous* inward labelled edges *(or simply "in-edges") of x in G.*

For convenience, we also provide notation that captures both inward and outward edges in the one representation. We define the set of all such edges as $\mathbf{E} := \mathbf{U} \times \mathbf{ULBS} \times \{+,-\}$, where the first element denotes the predicate, the second element denotes the value (subject or object), and the third element is a special symbol used to denote either an outward edge ($+$) or an inward edge ($-$). We denote by $\mathrm{edge}(G,x) := (\mathrm{out}(G,x) \times \{+\}) \cup (\mathrm{in}(G,x) \times \{-\})$ the set of all edges for x in G. For an edge $e \in \mathbf{E}$, $\mathrm{edge}^-(G,e) := \{x \in \mathrm{terms}(G) \mid e \in \mathrm{edge}(G,x)\}$ denotes the set of all terms in G with that edge. By $\mathrm{gedge}(G,x) := \mathrm{edge}(G,x) \cap \mathbf{U} \times \mathbf{ULS} \times \{+,-\}$, we denote the set of all ground edges *for x in G. Finally, we denote by $\mathrm{edges}(G) := \bigcup_{x\in\mathrm{terms}(G)} \mathrm{edge}(G,x)$ the set of all edges for all RDF terms in G and analogously by $\mathrm{gedges}(G)$ the set of all ground edges in G.*

7.1 Skolemising trivially lean blank-nodes

We start with a simple sufficient condition for a blank node b to be lean with respect to a graph G: we can say that b is lean with respect to G if b is associated with a unique ground edge. [41] More specifically, we call such a blank node *trivially lean.*

Proposition 7.2 *A blank node b is* trivially lean *for an RDF graph G if there exists a ground edge $e \in \mathrm{gedge}(G,b)$ such that for all $x \in \mathrm{terms}(G)$, $x \neq b$, it holds that $e \notin \mathrm{gedge}(G,x)$. A trivially lean blank node is also lean (per Definition 2.6).* □

The first phase of our analysis thus identifies blank nodes that are trivially lean. We also remark that there is a notion of recursion implicit in Proposition 7.2: if we Skolemise blank nodes found to be trivially lean, these new Skolems present new ground edges that may trigger the Skolemisation of further trivially lean blank nodes. We will analyse the effect of this recursion later when discussing results.

Example 7.3 *In Figure 1, the blank node* `_:b2` *is trivially lean since it contains two ground edges that do not appear for another term elsewhere in the graph:* (`:event`, `:FrenchOpen`, $+$) *and*

[39] http://sw.deri.org/2006/08/nxparser/
[40] http://sw.deri.org/2008/07/n-quads/

[41] A blank node with a blank edge on a unique predicate would also be lean, but we do not consider this rare case: there are only 57,235 unique predicates in the BTC–2012 dataset vs. 88.677 million unique blank nodes.

```
1: function TRIVIALLYLEAN($Q$)                         ▷ $Q$ a list of quadruples
2:     $Q_{\texttt{SPOG}} \leftarrow$ sort $Q$ by SPOG
3:     $Q_{\texttt{OPSG}} \leftarrow$ sort $Q$ by OPSG
4:     $S \leftarrow$ UNIQUEEDGES($Q_{\texttt{SPOG}}, 0, 1, 2$)
5:     $S \leftarrow S \cup$ UNIQUEEDGES($Q_{\texttt{OPSG}}, 2, 1, 0$)
6:     $Q' \leftarrow$ mark all Skolems $S$ in $Q$
7:     return $Q'$                          ▷ trivially lean blank nodes Skolemised
8: function UNIQUEEDGES($G$,$x$,$p$,$y$)                  ▷ also accepts quads
9:     assumes $G = (t_1, \ldots, t_n)$, unique, grouped-by $x, p$
10:    $S \leftarrow \emptyset$
11:    for $i \leftarrow 1; i \leq n; i$++ do                 ▷ $\pi$ denotes projection
12:        if $\pi_x(t_i) \in \mathbf{ULS} \wedge (i = 1 \vee \pi_{x,p}(t_i) \neq \pi_{x,p}(t_{i-1}))$
   $\wedge (i = n \vee \pi_{x,p}(t_i) \neq \pi_{x,p}(t_{i+1})) \wedge \pi_y(t_i) \in \mathbf{B}$ then
13:            $S \leftarrow S \cup \mathrm{sk}(\{\pi_y(t_i)\})$
14:    return $S$                                  ▷ $S \subset \mathbf{S}$: a set of Skolems
```

Algorithm 1. Find trivially lean blank nodes

(`:year`, `"2009"`, $+$). *Either of these edges would be sufficient to make* `_:b2` *lean. Similarly* `_:b1` *is also trivially lean due to the edge* (`:year`, `"2003"`, $+$) *However,* `_:b3` *is not trivially lean since neither of its two ground edges*—(`:wins`, `:Federer`, $-$) *and* (`:event`, `Wimbledon`, $+$)*—are unique.*

Consider a version of Figure 1 without the edge `:precededBy` *between* `_:b1` *and* `_:b2`*, and consider that we Skolemise* `_:b2` *per the reasons above. Now* `_:b3` *can be (recursively) considered lean per Proposition 7.2 since in this version of the graph,* `_:b3` *contains a unique grounded edge from the Skolem* `_:b2` *through the relation* `:precededBy`*.*

Implementation Algorithm 1 outlines our method for finding trivially lean blank nodes. As per lines 2–3, we can sort the BTC–2012 dataset on-disk into two lexicographical orders: subject–predicate–object–graph (`SPOG`) and object–predicate–subject–graph (`OPSG`). Sorting the data groups all triples with the same `SP`/`OP` edges together. Scanning first the `SPOG` order (line 4), then the `OPSG` order (line 5), we can quickly identify blank-nodes that do not share an `SP` or `OP` edge with any other RDF term in the graph, noting these blank nodes as Skolems (function UNIQUEEDGES, where we overload the function to work for triples or quadruples). We use new Skolems to detect unique edges during an iteration, but only apply each scan once (*i.e.*, we do not reach a fixpoint in this phase although the function TRIVIALLYLEAN in Algorithm 1 could be looped until a fixpoint on S). After a single scan of both orders, we then create one output file that Skolemises blank-nodes by marking them with a reserved syntax (line 6).

Timing Overall, the entire process of sorting and scanning both orders and writing the Skolemised output took approximately 23.4 h. Two external sorts for `SPOC` and `OPSC` ordering took 8.3 h and 9.8 h respectively. Thereafter, scanning each order required about 1.1 h, and writing the Skolemised output required 3.1 h, giving a total of 5.3 h runtime (after sorting).

Results From the total set of 88.677 million unique blank nodes, in the first scan of `SP` edges, we Skolemised 15.148 million blank nodes (17.1% of all blank nodes). In the first scan of `OP` edges, we Skolemised a further 46.696 million blank nodes (52.7% of all blank nodes). Thus, per Proposition 7.2, using only one iteration, we could Skolemise a total of 61.844 million trivially lean blank nodes (69.7% of all blank nodes) due to having a unique ground edge. These novel Skolems are mentioned in 216.214 million statements, representing 17.6% of all data and 78.9% of the statements containing a blank node.

As previously stated, the process is in theory recursive and the algorithm could be applied until a fixpoint. We tested a fixpoint version of the algorithm, which required 26 iterations of both orders and took 53.9 h (excl. sorting, vs. 5.3 h for one pass). Figure 5 plots the number of additional Skolems found in each subsequent iteration, where we found that the number of additional blank nodes identified as lean trailed off dramatically after the first iteration (the 26th iteration found no new lean blank nodes, confirming the fixpoint). Since 99.81% of the blank nodes that were Skolemised were found in the first pass, we deemed it unproductice to run Algorithm 1 until fixpoint. Instead, we only run a single pass and propose an alternative and more efficient method for recursively identifying further lean blank nodes in the next section.

7.2 Propogating Skolems through reachability

We have found that the majority of blank nodes are trivially lean. However, after one iteration of the previous phase, there are still 26.834 million blank nodes (30.3%) left unclassified. Instead of running the previous "global" algorithm until fixpoint, we can take advantage of the fact that unlike URIs and literals, Skolems are local to a given document G_i. This locality means that if a blank node has a unique edge involving a Skolem in its local docu-

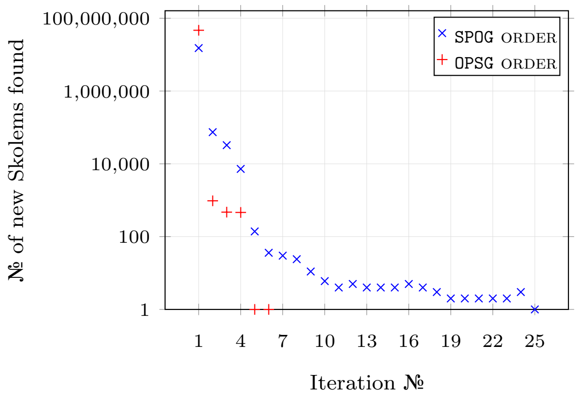


Fig. 5. Number of trivially lean blank nodes found per iteration

ment, then that edge is also globally unique. Thus we can partition the problem of detecting and propagating unique edges with Skolems over individual documents G_i instead of the global merge M, allowing to reach local closures on a per-document basis. This offers huge computational benefits.

Still, we see from the global experiment that the amount of Skolems found in subsequent iterations trails off very quickly, relative to the overall volume of blank nodes. Analogously, we would expect relatively few Skolems to be produced by applying local closures, even if more efficient. However, the locality of Skolems allows us to further broaden our search for lean blank nodes with the following observation. If a blank node has an edge connecting to a Skolem, then it can only be made non-lean by a term in the local document connected by the same predicate in the same direction to the same Skolem.

Definition 7.4 *Let G be an RDF graph and let $E :=$ edges$(G) \cap (\mathbf{U} \times \mathbf{S} \times \{+,-\})$ denote the set of all edges in G associated with a Skolem. We then denote the set of all sets of* Skolem neighbours *for G by* skn$(G) := \{$edge$^-(G, e) \mid e \in E\}$.

Lemma 7.5 *Let G be an RDF graph and let $N \in$ skn(G) be a set of Skolem neighbours in G. Let G' be an arbitrary RDF graph such that* terms$(G) \cap$ terms$(G') \cap \mathbf{BS} = \emptyset$. *It holds that any blank node $b \in N \cap \mathbf{B}$ can only be made non-lean with respect to $G \cup G'$ by a witness in N (see Definition 2.6).* □

In other words, each set of Skolem neighbours $N \in$ skn(G) is "closed" under leanness: a blank node in N can only be made non-lean by a witness in N. Furthermore, if N contains only blank nodes and/or Skolems, then we have all the information about the edges for all terms in N in the local document: if any blank node in N contains an edge in the local document that is unique from all the other blank nodes and/or Skolems in N, then it must be lean for the entire dataset (as per Proposition 7.2).

Proposition 7.6 *For a graph G, let $N \in$ skn(G), $N \subset \mathbf{BS}$ be a set of Skolem neighbours in G comprised solely of blank nodes and Skolems. Let G' be an arbitrary RDF graph such that* terms$(G) \cap$ terms$(G') \cap \mathbf{BS} = \emptyset$. *If a blank node $b \in N \cap \mathbf{B}$ is trivially lean for the graph G as per Proposition 7.2, then it is also lean for $G \cup G'$.* □

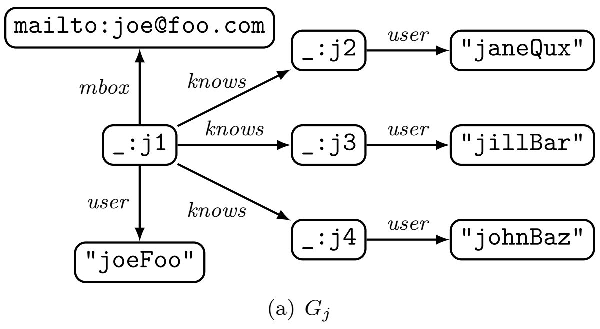
Using this principle, by looking at individual documents with at least one Skolem in the dataset, we can identify further global Skolems while only looking at local data. Furthermore, the process can again be iterative: a newly discovered Skolem may lead to a new set of Skolem neighbours in G that may lead to further Skolems, and so forth.

Example 7.7 *The two graphs in Figure 6 exemplify a common pattern we found in our data, relating to social networks outputting FOAF profiles that use blank nodes to refer to individual users. G_j is a profile for* `joeFoo` *and lists his friends along with some unique information like his email address. G_k is the profile for* `janeBar`*, listing similar friends.*

Only `_:j1` *and* `_:k1` *are trivially non-lean across the merge of the graphs and these two blank nodes are Skolemised per the previous phase. Now we have two sets of Skolem neighbours anchored by these two Skolems. The edge* (`:knows`, `<_:j1>`, $-$) *forms $N_j =$* {`_:j2`, `_:j3`, `_:j4`}. *The edge* (`:knows`, `<_:k1>`, $-$) *forms $N_k =$* {`_:k2`, `_:k3`, `_:k4`}.

Taking N_j for the moment, we know that a blank node in that set can only be made non-lean by a term in that set. Thus, to show that a blank node b is lean in N_j, it is sufficient to find a ground edge unique for the terms of N_j. Since N_j only contains blank nodes, all such edges can be found in G_j. Hence we can ultimately Skolemise all blank nodes in N_j and analogously in N_k: although no such blank node contains any globally-unique ground edge, each such blank node contains an edge unique for the blank nodes connected to the same Skolem.

Implementation Algorithm 2 gives an overview of the method we use to perform the local "closures" of trivially-lean blank nodes. We take as input the data marked with Skolems as output from Algo-



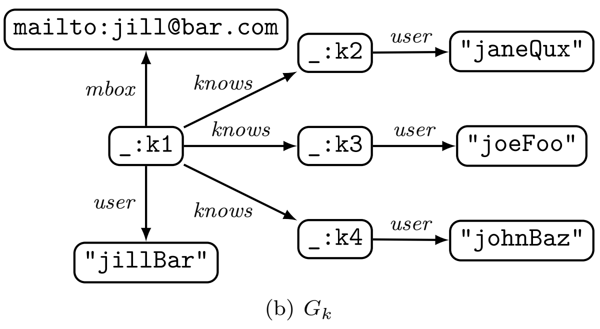


Fig. 6. Two example snippets of connected blank nodes taken from different RDF graphs

rithm 1 (a single-pass). In order to group the data for each document together, on line 2, we first sort the input quadruples lexicographically by graph (we apply GSPO ordering, but any G* ordering would be sufficient). This allows us to sequentially scan the data grouped by source document (as indicated by line 3). For each document with at least one Skolem, we then build a list of all sets of Skolem neighbours in memory (line 10), and discard sets containing a URI or a literal (line 11; these neighbour sets may involve relevant triples outside of the current document). For each remaining set, we load all triples mentioning all neighbour terms into memory (lines 12 & 14) and Skolemise blank nodes with a unique edge therein (lines 13 & 15). Skolems are marked in the document (line 17) and then the next neighbour set is analysed. We recurse until no new Skolems are found (the **repeat** loop starting on line 8), reaching a fixpoint. The next document is then loaded.

Although Algorithm 2 contains many nested loops (stated this way for succinctness), the sum of the cardinalities of all candidate sets that can be created for a graph G is bounded by the size of G since each element of each candidate set must be associated with a triple that connects it to a Skolem, per Definition 7.4. The number of **repeat** loops required to reach a fixpoint is bounded by the total number of blank nodes in G. As we will see however, in practice, one loop is sufficient in the majority of cases.

Timing The total time taken for this phase was 20 h. Applying an external sort of the data for GSPO order took a total of 8.5 h. Running the local leanness checking and outputing the newly Skolemised data took 11.5 h.

```
1:  function PROPAGATELEAN($Q'$)                    ▷ output from Alg. 1
2:      $Q_{\texttt{GSPO}} \leftarrow$ sort $Q$ by GSPO
3:      assume $Q_{\texttt{GSPO}} = \{(u_1, G_1), \ldots, (u_n, G_n)\}$
4:      for $i \leftarrow 1; i \leq n; i{+}{+}$ do
5:          load $G_i$ into memory
6:          define sort orders $G_{\texttt{SPO}}$, $G_{\texttt{OPS}}$ for $G_i$
7:          $S_{new} \leftarrow$ terms$(G_i) \cap \mathbf{S}$, $S_{all} \leftarrow S_{new}$, $G'_i \leftarrow G_i$
8:          repeat
9:              $S_{next} \leftarrow \emptyset$
10:             for all $N \in$ skn$(G_i)$ do                ▷ per Def. 7.4
11:                 if $N \subset \mathbf{BS}$ then
12:                     $G'_\alpha \leftarrow \sigma_{obj \in N}(G'_{\texttt{SPO}})$   ▷ $\sigma$ for selection
13:                     $S_\alpha \leftarrow$ UNIQUEEDGES$(G'_\alpha, 0, 1, 2)$
14:                     $G'_\beta \leftarrow \sigma_{sub \in N}(G'_{\texttt{OPS}})$
15:                     $S_\beta \leftarrow$ UNIQUEEDGES$(G'_\beta, 2, 1, 0)$
16:                     $S_{next} \leftarrow S_{next} \cup S_\alpha \cup S_\beta$
17:                     $G'_i \leftarrow$ mark all Skolems $S_{next}$ in $G'_i$
18:             $S_{new} \leftarrow S_{next} \setminus S_{all}$
19:             $S_{all} \leftarrow S_{all} \cup S_{new}$
20:         until $S_{new} = \emptyset$
21:     return $Q'' = \{(u_i, G'_i), \ldots, (u_n, G'_n)\}$
```

Algorithm 2. Closing trivially lean blank nodes

Results From the total set of 88.677 million unique blank nodes, of which 61.844 million (69.7%) were Skolemised in the previous phase, we could Skolemise an additional 11.009 million blank nodes (12.4%) using the described process. In 98.6% of cases, one iteration was sufficient to reach closure; in the worst case, 1,556 iterations were required.

After this phase, we are left with 15.825 million blank nodes (17.8%) to classify.

7.3 Unique set of ground edges

Thus far, we have looked for lean blank nodes based on having a single unique ground edge, starting with edges that were unique in the global data, moving on to edges that were unique in sets of

Skolem neighbours. Together, these two steps were sufficient to classify 82.2% of all blank nodes in the BTC–2012 dataset as lean. After these phases, we observed that of the remaining 15.825 million blank nodes left to classify, 11.518 million (72.8%) were unconnected; *i.e.*, they only had ground edges (although no such edge was unique). Many such blank nodes were disconnected by the introduction of Skolems in previous phases.

In this third phase, we focus on classifying these unconnected blank nodes by checking to see if any other RDF term in the data has a super-set of the ground edges that it has.

First, we can state that a blank node is lean if all of its ground edges are not all held by another RDF term in the data.

Proposition 7.8 *Let G be an RDF graph and let $b \in \text{terms}(G) \cap \mathbf{B}$ be a blank node in G. If there does not exist an $x \in \text{terms}(G)$ such that $\text{gedge}(G,b) \subseteq \text{gedge}(G,x)$, then b is lean in G.* $\square$

Furthermore, we can state that if a blank node is unconnected and all of its ground edges are held by another RDF term in the data, then that blank node is non-lean.

Proposition 7.9 *Let G be an RDF graph and let $b \in \text{terms}(G) \cap \mathbf{B}$ be a blank node in G such that $\text{edge}(G,b) = \text{gedge}(G,b)$ (i.e., b is unconnected). If there exists an $x \in \text{terms}(G)$ such that $\text{edge}(G,b) \subseteq \text{edge}(G,x)$, then b is non-lean in G.* $\square$

Together, these two conditions can be used to classify all 11.518 million unconnected blank nodes in the data as either lean or non-lean. Per Proposition 7.8 we can further classify some connected blank nodes as lean.

Example 7.10 *None of the four blank nodes in Figure 7 contains a unique edge by itself. However, the set of ground edges (the combination of year, wins and event) for* `_:b2` *and* `_:b3` *are unique; thus, we can classify these two blank nodes as lean. For the unconnected blank node* `_:b1`*, its set of edges* $\{(\texttt{:wins}, \texttt{:Nadal}, -), (\texttt{:event}, \texttt{:FrenchOpen}, +)\}$ *is a subset of those for* `_:b2`*, and thus we can classify* `_:b1` *as non-lean. Since* `_:b4` *is connected and its ground edges are covered by* `_:b3`*, we do not classify it in this phase.*

Implementation Algorithm 3 presents a high-level overview of the process, which consists of two main steps. Returning to the querying analogy, this step is equivalent to running 15.825 million "star-shaped" queries with one variable. Running this method for a dataset the size of the BTC–2012 and for 15.825 million unclassified blank nodes broaches on significant engineering challenges, where we now give a brief overview of the methods and optimisations employed (some of which are omitted from the algorithm for brevity).

```
 1: function GEDGESET($Q''$)                                ▷ output from Alg. 2
 2:     $B \leftarrow \text{terms}(Q'') \cap \mathbf{B}$, $B_{nl?} \leftarrow \emptyset$     ▷ $B_{nl?}$: non-lean?
 3:     for all $b \in B$ do
 4:         load $\text{edge}(b, Q'')$ into memory
 5:     for all $x \in \text{terms}(Q'')$ do                     ▷ on-disk index join
 6:         $B_{old} \leftarrow B_{nl?}$
 7:         for all $e \in \text{gedge}(Q'', x)$ do
 8:             $B_{new} \leftarrow \{b \in B \setminus B_{old} \mid e \in \text{gedge}(Q'', b)\}$
 9:             for all $b \in B_{new} \setminus \{x\}$ do
10:                 if $\text{gedge}(Q'', b) \subseteq \text{gedge}(Q'', x)$ then
11:                     $B_{nl?} \leftarrow B_{nl?} \cup \{b\}$
12:             $B_{old} \leftarrow B_{new} \cup B_{old}$
13:     $S \leftarrow \text{sk}(B \setminus B_{nl?})$
14:     $B_{nl} \leftarrow \{b \in B_{nl?} \mid \text{edge}(Q'', b) = \text{gedge}(Q'', b)\}$
15:     $Q''' \leftarrow$ mark all Skolems $S$ in $Q''$
16:     return $Q'''$ and $B_{nl}$
```

Algorithm 3. Classify based on ground-edge set

In the first step, lines 2–4, an in-memory index of unconnected blank nodes and their edges is created: the reduced number of blank nodes left to classify after the first two phases makes this feasible. The index is created by scanning over the `SPOG` sorted order (grouped by inlinks) and the `OPSG` sorted order (grouped by outlinks), loading edges for blank nodes into memory. To improve memory efficiency, we dictionary encode edges using sequential integers. The index supports looking up the set of all (unconnected blank nodes) with a particular edge. The index can also return a subset of blank nodes for which the given edge is the most selective (*i.e.*, the most rare edge for each blank node); we can use this feature as an optimisation in the next phase.

In the second step, lines 5–12, the edges for all RDF terms in the data are materialised and checked against the in-memory index. To achieve this, an on-disk index join is applied over the `SPOG` and `OPSG` sorted orders, aligning on the primary RDF term in the subject (`SPOG`)/object (`OPSG`) position. This allows for sequentially scanning through all quadruples where a given RDF term is in the subject or object position, further allowing to generate the set of edges for all RDF terms. The edges for each RDF term are written to disk and are dictionary encoded using the same identifiers as before; to reduce the

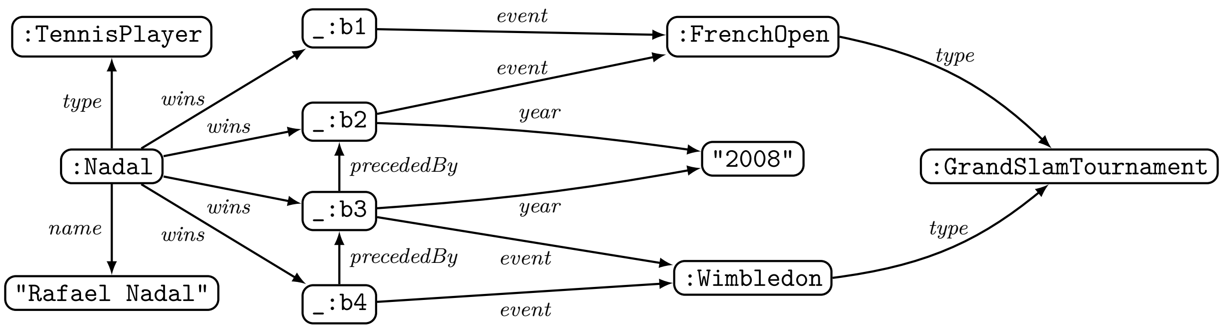


Fig. 7. An RDF graph to illustrate classification according to having a unique set of ground edges

data sizes, only edges associated with at least one blank node from the in-memory index are recorded.

Then for each RDF term, line 8 retrieves all blank nodes sharing an edge with the current RDF term and line 10 checks each such blank node to see if its ground edges are a subset of those for the current RDF term. These checks are performed against the in-memory index. Though not shown in the algorithm, line 8 only retrieves blank nodes for which the given edge e is the most selective. This optimisation considerably reduces the number of blank nodes to be checked by the algorithm; for example, it would avoid repeatedly matching large numbers of blank nodes to large number of terms based on low-selectivity edges like $(\texttt{rdf:type}, \texttt{foaf:Person}, +)$, unless necessary (*i.e.*, it was the most selective edge that they shared, which would be a rare case). We also implement most-recently-used caching methods to skip over redundant RDF terms (line 5) sharing an identical edge-set with one that was recently seen.

Finally, on line 13, any blank node whose ground edges are not found to be a subset of those for another RDF term in the data can be classified as lean, as per Proposition 7.8. On line 14, any unconnected blank node whose edges are a subset of those for another RDF term in the data can be classified as non-lean, as per Proposition 7.9.

Timing The entire process took 13.41 h.

Building the in-memory index for blank nodes and recording the dictionary-encoded edges for all RDF terms took 4.36 h. A total of 6.787 million unique ground edges were found. The Java heap-space cost for the in-memory index was estimated at 20.78 GB.[42]

[42] Using `java.lang.management.ManagementFactory.getMemoryMXBean().getHeapMemoryUsage().getUsed()`.

In the next phase, 61.881 million RDF terms with a total of 134.007 million (relevant) edges were checked against the unconnected blank nodes in the in-memory index. The cache had a 70.9% hit rate suggesting that after filtering irrelevant edges, many of the remaining RDF terms have identical edge sets. Running these checks took 9.04 h.

Results This phase classified 11.520 million blank nodes (13.0% of all original blank nodes; 72.8% of previously unclassified blank nodes). Of these, 10.410 million (90.36%) were classified as lean and 1.110 million were classified as non-lean (9.64%). As aforementioned, all unconnected blank nodes were classified. However, only 1,386 connected blank nodes were classified in this step (as lean).

With respect to the 1.110 million non-lean blank nodes, we found 1.115 million unique corresponding witnesses, creating 9.722 million unique pairs of blank nodes and their witnesses (*i.e.*, pairs of the form (b, x) where $b \neq x$, all edges for b are ground and $\mathrm{gedge}(Q'', b) \subseteq \mathrm{gedge}(Q'', x)$). Only 935 witnesses were URIs; the rest were blank nodes. Of the 9.722 million pairs, 9.467 million (97.37%) were between unclassified blank nodes and 9.459 million (97.28%) involved pairs of isomorphic blank nodes that were witnesses for each other (*i.e.*, (b, b') where $b \neq b'$, all edges for both are ground and both sets of ground edges are equal).

Furthermore, of the 9.722 million pairs, 8.493 million (87.35%) involved a pair of blank nodes from the same pay-level-domain and, more specifically, 1.718 million (17.67%) involved blank nodes from within the same document. In the former case of blank nodes being on the same domain but in different documents, many such documents were syntactic copies present in different Web locations (giving 200 OK under different URLs) but on the same domain.[43] In the latter more specific case of a non-lean blank node having a blank node witness in the same document, these were due to blank nodes being left "underspecified" in certain exporters. We present one such example:

Example 7.11 *On the Semantic Web Dog Food server (`data.semanticweb.org`), we found many instances of non-lean documents for author profiles. Taking an example of the RDF document available from http://data.semanticweb.org/person/claudio-gutierrez/rdf, in our crawl (and at the time of writing), we found triples of the following form:*

```
_:b1 rdf:_1 swperson:claudio-gutierrez .
_:b2 rdf:_2 swperson:claudio-gutierrez .
_:b3 rdf:_2 swperson:claudio-gutierrez .
...
```

These triples use container-membership predicates of the form `rdf:_`*n. Each blank node refers to an author-list and each blank node only appears in one triple as above: since the document refers to a specific author, only that author is listed in the local container. Intuitively, each blank node represents a positional authorship, where an author can have multiple papers as first author, second author,* etc. *However, authorships cannot be traced back to specific papers and all blank nodes representing authorships at a given position contain precisely the same information, leading to non-leanness within the document. Intuitively, these blank nodes are not redundant since they preserve some information about the cardinality of the authorship position (*e.g.*, how many first-authorships for that person the site knows of); however, this information is lost (due to the lack of a UNA and the existentiality of blank nodes in RDF).*

7.4 Minimal isomorphic graphs

As a result of the previous three phases, we have classified 95.15% of all blank nodes, leaving 4.305 million left to classify. We know that these remaining blank nodes are connected and that their set of ground edges is not a subset of those for any other RDF term in the data.[44]

In the previous phase, although we could only classify unconnected blank nodes as non-lean, we found that the majority of blank nodes classified thusly were isomorphic (97.28%). In this phase, we further identify isomorphism for connected blank nodes: we look for minimal isomorphic graphs that preserve the connectedness of blank nodes and from which we can classify further non-lean blank nodes. Although we will not detect all remaining non-lean blank nodes and the homomorphisms that cause them, the isomorphic case (as defined in Section 2.1) offers computational benefits that allows us to narrow down the search space before going further.

First, in terms of worst-case complexity, the graph isomorphism problem is in NP (not known to be NP-complete or in P), whereas graph homomorphisms are known to be NP-complete. More importantly, graph isomorphism is a well-studied problem with a number of well-known algorithms, such as the Nauty algorithm [50], where efficient implementations are possible for many practical cases (particularly vertex intransitive graphs). In addition, such algorithms tackle simple graphs, whereas RDF graphs contain further "ground" information that can be used to quickly reduce the search space of isomorphic candidates, as we will see later.

First, we give a formal statement of the condition we check in this phase.

Proposition 7.12 *Let G_1 and G_2 be two RDF graphs not sharing any blank nodes. If $G_1 \cong G_2$ (*i.e.*, they are isomorphic per Section 2.1), then all blank nodes in* $\mathrm{terms}(G_1 \cup G_2) \cap \mathbf{B}$ *are non-lean with respect to $G_1 \cup G_2$.* □

In order to check this condition, from the data, we construct a set of minimal RDF graphs that do not share any blank nodes but whose union precisely covers all data containing blank nodes and where each individual graph in the set minimally preserves the connectedness of blank nodes.

Definition 7.13 *We define the* blank-node partition $\mathrm{bnp} : 2^{\mathbf{UBS}\times\mathbf{U}\times\mathbf{UBSL}} \rightarrow 2^{2^{\mathbf{UBS}\times\mathbf{U}\times\mathbf{UBSL}}}$ *of an RDF graph G as the set of RDF graphs $\mathcal{G}$ such that the following three conditions hold:*

[43] As an example, see http://vocab.org/bio/0.1/Event.rdf and http://vocab.org/bio/0.1/Formation.rdf; there were 81 such copies in the data. Though syntactically identical, the resulting RDF graphs are not isomorphic due to the presence of relative URIs without an explicit base URI.

[44] Given that we found some new Skolems in the last phase, we could re-run the second phase again to trigger further Skolems (*e.g.*, `_:b4` in Example 7.10 now connected to `<_:b3>`); however, the last phase only touched upon 1,386 connected blank nodes so we deemed this to not be worth it.

(i) $\bigcup_{G' \in \mathcal{G}} G' = \{(s,p,o) \in G \mid s \in \mathbf{B} \vee o \in \mathbf{B}\}$*;*
(ii) no two graphs in $\mathcal{G}$ *share a blank node;*
(iii) for all $G' \in \mathcal{G}$, $\mathrm{blank}(G')$ *is connected.*
If G *is ground, then* $\mathrm{bnp}(G) := \emptyset$*.*

The blank-node partition of an RDF graph is unique, where each graph in $\mathrm{bnp}(G)$ is associated with a connected component of $\mathrm{blank}(G)$, containing all and only those triples in G mentioning a blank node from that component. Our task now is to search for isomorphic graphs within the blank-node partition of the BTC–2012 data.

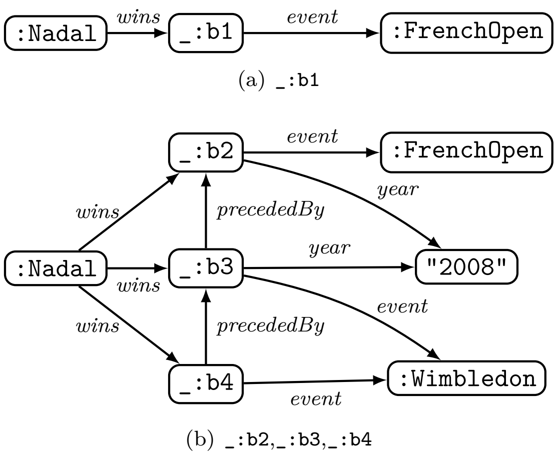


Fig. 8. The two graphs in the blank node partition of the RDF graph in Figure 7.

Example 7.14 *The blank graph taken from the data in Figure 7 contains two connected components with the vertices* {`_:b1`} *and* {`_:b2`, `_:b3`, `_:b4`}*. Figure 8 depicts the corresponding blank node partition with two graphs. The goal in this phase is to construct all such graphs from the BTC–2012 data and look amongst them for isomorphic graphs (see Section 2.1; graphs that are identical up to a one-to-one blank node relabelling). The blank nodes in such isomorphic graphs can then be classified as non-lean. (In fact, since* {`_:b1`} *is unconnected, it would have been classified in the previous phase.)*

Implementation Algorithm 4 outlines the process we use to find isomorphic graphs within the blank-node partition of the data.

First, on lines 3–5, instead of scanning through all of the data again to compute the blank-node partition, we can re-use the edge-sets for blank nodes extracted in the previous phase and load them into memory (as per Algorithm 3). These edge-sets preserve all of the required information to construct the blank-node partition of the data in memory for the remaining unclassified blank nodes.

Second, on lines 6–15, we begin to label the vertices of each graph using similar methods to the Nauty algorithm. In the Nauty algorithm, aside from trivially rejecting isomorphism (based on, *e.g.*, the size of the graph), the first step is to label the considered graphs according to *vertex invariants*, which are preserved in an isomorphism. A prominent example of a vertex invariant would be its degree, where a vertex in one graph must be mapped to a vertex with the same degree in its isomorph, helping to narrow the search space. In the case of RDF graphs, there are many more "vertex invariants" available with which to label vertices (blank nodes), which leads to more discriminating labels and smaller search spaces (*cf.* the signature method of Tzitzikas *et al.* [69]). On lines 6–15, for each blank node that has not already been classified, we compute a hash function over all of its edges, capturing the number of edges, the predicate and directionality of each, and the ground subject/object value of each (if present). Here, the *hash*(.) function denotes a method to compute a hash over a tuple (preserving ordering), whereas the $\oplus$ (XOR) operator combines hashes commutatively and associatively (agnostic to order). [45]

Third, on line 16 we call a function COLOUR, which uses the initial vertex hashes to "colour" the graph (as also performed by, *e.g.*, the Nauty algorithm). The colouring assigns a more detailed hash (what we call a "colour") to each vertex by propagating vertex invariants through edges in the graph to some fixed depth, thus encoding how the vertices are connected. At depth 0, the colour of the graph is the same as the initial hashes produced for each vertex. At depth n, the colour of a vertex combines the colour of that vertex at depth $n-1$ with the colours at depth $n-1$ of its neighbours (including the respective edge direction(s) and predicate(s) to that neighbour). Deeper colourings capture more detailed information about the connections in the graph.

Fourth, on lines 17–27, we group graphs according to having the same bag of vertex colours and check that the graphs in each group are indeed isomorphic: we know that isomorphic graphs must have the same such "colour bag", and we know that each such

[45] XOR is chosen as a bit-wise operator where the truth tables are balanced in output for 1's and 0's: otherwise the hash would tend towards signed 2^{31} (for OR) or 0 (for AND).

colour bag should be highly discriminating for most practical cases. However, certain graphs (esp. vertex transitive graphs) may share a colour bag and not be isomorphic; though we assume such cases to be rare in practice, we still need to confirm that the graphs in each such group are indeed isomorphic. We thus create and iterate over each such group (line 21) and check that the contained graphs are isomorphic. On lines 23–25, to avoid checking all pairs of graphs, where possible, we skip checks involving graphs that have been confirmed to be isomorphic with one that was already checked; hence, in the common case that all m graphs in a group are isomorphic, only $m-1$ checks are needed (otherwise, if none are isomorphic, the worst case is $\frac{m(m-1)}{2}$ pair-wise symmetric and irreflexive checks).

Finally, line 25 calls the VERIFYISO function with two graphs G and G' and their associated colourings (where the colour bags of both graphs are assumed equal). The function performs an isomorphism check by iterating over all possible bijections from the blank nodes in G to the blank nodes in G' that preserve colouring. Since the colour bags of both graphs are already known to be equal, we know that there is one such bijection; however, we do not know whether or not any such bijection is an isomorphism. Hence, we iterate through all colour-preserving bijections, returning `true` for the first bijection λ such that $\lambda(G) = G'$, or `false` if no such bijection is found. The total set of all possible colour-preserving bijections Λ (defined on line 41) then depends on how effective the colouring scheme is at distinguishing vertices. If we define $B/{\sim} := \{B_1, \ldots, B_n\}$ as the quotient set of blank nodes B in G by the same-colour relation, then the total number of colour-preserving bijections between vertices in G and G' is given as $|\Lambda| = \prod_{1 \leq i \leq n} |B_i|!$. Thus if each vertex in G (and thus in G') has a unique colour, then $|\Lambda| = 1$. Otherwise, in the worst-case, if G (and thus G') contains k blank node vertices with one colour, then $|\Lambda| = k!$. However, we assume that in most practical cases, the colourings of the graphs will be sufficiently discriminating to permit very few permutations of colour-preserving mappings.[46] When isomorphic graphs are found, their blank nodes are classified as non-lean (line 27).

[46] Algorithms like Nauty implement other checks before this phase, such as removing automorphisms from the graphs that could lead to multiple vertexes in a graph sharing the same colour. Our naive algorithm without automorphisms checks was sufficient for the cases considered.

```
 1: static $depth$                                   ▷ how many hops to colour graph
 2: function FINDISO($Q''', B_{nl}$)                  ▷ output from Alg. 3
 3:     for all $b \in (\mathrm{terms}(Q''') \cap \mathbf{B}) \setminus B_{nl}$ do
 4:         load edge$(b, Q'')$ into memory
 5:     $\mathcal{G} \leftarrow \mathrm{bnp}(Q''')$      ▷ using union–find over edge$(b, Q'')$
 6:     init $\mathcal{C}[][]$                          ▷ assoc. array for graph colours
 7:     define $hash(T)$                              ▷ a hash for an ordered tuple $T$
 8:     for all $G' \in \mathcal{G}$ do
 9:         init $H[]$                                ▷ assoc. array of vertex hashes
10:         for all $b \in (\mathrm{terms}(G') \cap \mathbf{B}) \setminus B_{nl}$ do
11:             for all $e \in \mathrm{edge}(b, Q''')$ do     ▷ $\pi$: projection
12:                 if $\pi_2(e) \notin \mathbf{B}$ then $h \leftarrow hash(e)$
13:                 else $h \leftarrow hash(\pi_{1,3}(e))$      ▷ remove b.node
14:                 if $H[b] = $ null then $H[b] \leftarrow h$
15:                 else $H[b] \leftarrow H[b] \oplus h$            ▷ $\oplus$: XOR
16:         $\mathcal{C}[G'] \leftarrow$ COLOUR$(G', H[], depth)$   ▷ vertex colours
17:     define $bag(A[])$                             ▷ the bag of values in $A[]$
18:     define $G_i \sim G_j \longleftrightarrow bag(\mathcal{C}[G_i]) = bag(\mathcal{C}[G_j])$
19:     init $\cong$           ▷ isomorphic equiv. rel. (refl., trans., sym.)
20:     $B'_{nl} \leftarrow B_{nl}$                          ▷ non-lean blank nodes
21:     for all $\mathcal{G}' \in \mathcal{G}/{\sim}$ do           ▷ $\mathcal{G}/{\sim}$ : quotient set by $\sim$
22:         $Done \leftarrow \emptyset$
23:         for $i \leftarrow 1; i < |\mathcal{G}'| \wedge i \notin Done; i{+}{+}$ do
24:             for $j \leftarrow i+1; j \leq |\mathcal{G}'| \wedge j \notin Done; j{+}{+}$ do
25:                 if VERIFYISO$(G'_i, G'_j, \mathcal{C}[G_i], \mathcal{C}[G_j])$ then
26:                     assert $G'_i \cong G'_j$
27:                     $B'_{nl} \leftarrow B'_{nl} \cup \mathrm{terms}(G'_i \cup G'_j) \cap \mathbf{B}$
28:                     $Done \leftarrow Done \cup \{i, j\}$
29:     return $B'_{nl}$
30: function COLOUR$(G', C_0[], d)$                  ▷ $d$: depth, $C_0$ initial
31:     if $d = 0$ then return $C_0[]$                ▷ $C_0$ typically hashes
32:     $C_{d-1}[] \leftarrow$ COLOUR$(G', C_0[], d-1)$
33:     $C_d[] \leftarrow C_{d-1}[]$                       ▷ assoc. array of vertex colours
34:     for all $b \in \mathrm{terms}(G) \cap \mathbf{B}$ do
35:         for all $e \in \mathrm{edge}(b, G) : \pi_2(e) \in \mathbf{B}$ do
36:             $h \leftarrow hash(\pi_1(e), C_{d-1}[\pi_2(e)], \pi_3(e))$
37:             $C_d[b] \leftarrow C_d[b] \oplus h$                        ▷ $\oplus$: XOR
38:     return $C_d[]$
39: function VERIFYISO$(G, G', C[], C'[])$                      ▷ $G \sim G'$
40:     $B \leftarrow \mathrm{terms}(G) \cap \mathbf{B}$, $B' \leftarrow \mathrm{terms}(G') \cap \mathbf{B}$
41:     $\Lambda \leftarrow \{\lambda : B \overset{1:1}{\to} B' \mid C[b] = C'[\lambda(b)]$ for all $b \in \mathbf{B}\}$
42:     for all $\lambda \in \Lambda$ do          ▷ $\lambda$: colour-preserving bijection
43:         if $\lambda(G) = G'$ then
44:             return true                     ▷ $\lambda$ is an isomorphic map
45:     return false
```

Algorithm 4. Finding minimal isomorphic graphs

Timing We first tried the algorithm for $depth = 0$ (directly using the vertex hashes). However, the algorithm failed to terminate in reasonable time: certain graphs contained tens of blank nodes with precisely the same ground edges and predicates, where the number of colour-preserving mappings (generated on line 41) exceeded billions and where not all such cases were isomorphic. Thus we see that without colouring, the "direct" hashes computed for blank nodes are not discriminating enough.

We then set *depth* = 1 and this time, the computation terminated after slightly over 1 h. At this colouring depth, only 11 cases generated more than one mapping, with 8 cases generating two mappings, 1 case generating four mappings ($2!^2$ due to two sets of two blank nodes with the same colour), 1 case generating 663 thousand mapping ($2!^5 \times 3!^2 \times 4!^3$) and the worst case generating 1.493 million mappings ($2!^3 \times 3!^5 \times 4!$). The algorithm required about two minutes to find an isomorphism in each of the latter two cases.

Results The blank node partition of the data (considering only unclassified blank nodes as vertices) resulted in 1.222 million components containing 4.305 million (unclassified) blank nodes (a mean of 3.52 blank nodes per component). Partitioning these 1.222 million graphs into equivalence classes that share a colour-bag resulted in 41,753 sets of graphs, of which, 29,252 contained more than one graph. In all cases, graphs sharing a (depth 1) colour-bag were found to be isomorphic. Figure 9 presents the resulting distribution of isomorphic equivalence classes, where we see that although in the most common case only two graphs are isomorphic, we also found many examples of larger equivalence classes where a variety of peaks are visible. Taking one example, we found 39 equivalence classes containing precisely 1,688 graphs. As per the results of the previous section for unconnected blank nodes, these connected isomorphic cases were again often due to the verbatim replication of RDF documents in different locations. However, we also found isomorphic blank nodes within 19,720 documents, where 19,390 such documents were associated with the `legislation.(data.)gov.uk` domains. For example, at the time of writing, http://legislation.data.gov.uk/mwa/2008/2/data.rdf contained isomorphic connected blank nodes referring to modifications of the same type to the same legislative document on the same grounds; although these isomorphic blank node clusters probably intend to refer to distinct modifications, the data associated with them is insufficient to distinguish them and the existential semantics of blank nodes further (and perhaps problematically) makes each copy redundant.

In summary, this phase classified an additional 4.268 million blank nodes as non-lean (99.14% of previously unclassified blank nodes, 4.81% of original blank nodes), leaving only 36,956 blank nodes unclassified (00.86% of previously unclassified blank nodes, 00.04% of original blank nodes).

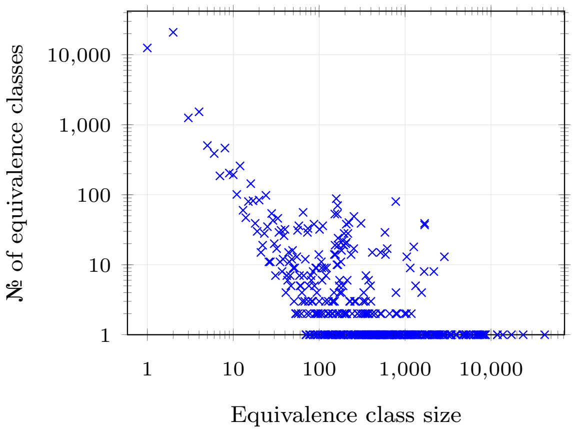


Fig. 9. Distribution of the number of graphs that are isomorphic

Final homomorphisms The previous four phases have classified 99.96% of blank nodes, of which 93.94% were classified as lean and 6.06% were classified as non-lean.[47] We are left with 36,956 blank nodes. We know that each of these blank nodes is connected, that their ground edge set is a subset of that for another term in the dataset, and that none of the connected components are isomorphic.

Since we have reduced the set of unclassified blank nodes by three orders of magnitude, and since we have run out of "tricks" to reduce the search space, we now test the remaining blank nodes for the standard homomorphism condition.

Implementation We use a method similar to that already described in Section 4.2 (see also Example 4.3), where SPARQL basic graph pattern matching is used to evaluate (non-)leanness. This is only now feasible since we have reduced the problem space considerably after the previous phases: considering the remaining unclassified blank node partition of the data, in engineering terms, the task is now equivalent to running 13,547 SPARQL queries

[47] If a blank node is classified as non-lean in either of the previous two phases, then all of the blank nodes it is connected to (if any) must also be classified as non-lean. Hence, none of the unclassified blank nodes are connected to a blank node that has been classified as non-lean.

over the full BTC–2012 data with an average of 2.73 variables and 10.42 triple patterns per query.

To run these checks, we implement a similar mechanism to standard basic graph pattern matching in SPARQL but implement custom optimisations for reducing the data sizes such that they can be indexed in memory. Algorithm 5 provides an overview of the process.

On lines 2–6, we begin by loading the edges of the remaining unclassified blank nodes into memory: we selectively load these from the list of dictionary encoded edges used in the previous phases. We also load the predicates associated with blank nodes, which are used later to filter triples from the dataset that are relevant for basic graph pattern matching: any triple with a predicate not in this set is irrelevant to the process (recall that since a blank node cannot appear in the predicate position, all of our "triple patterns" have bound predicates).

Next, on lines 8–12, we create an in-memory map from the remaining blank nodes to all terms in the data whose ground edges are a super-set of those for the blank node: this process is analogous to that sketched in Algorithm 3, lines 6–12, but where we also consider connected blank nodes. A blank node with no ground edges will be matched by all terms: in practice (not shown in the algorithm for brevity), we use a special symbol to denote this case rather than load all such terms for each such blank node. This candidate set narrows down the set of terms that blank nodes can be bound to in the basic graph pattern matching phase later and can again be used to filter irrelevant triples from the main dataset.

On line 13, we again load the blank node partition, considering only unclassified blank nodes. This blank node partition is built directly from the in-memory index of blank node edges loaded earlier. Each graph in the partition represents a "query" that we want to generate solutions for to test for the existence of the non-trivial homomorphisms that indicate non-leanness.

Thereafter, on lines 14–17, we scan the full data and, using the predicate set and candidate set produced earlier, we filter irrelevant triples that cannot be mapped from the blank edges loaded in memory. As aforementioned, blank nodes with no ground edges could potentially match a large number of triples: in practice (again not shown in the algorithm for brevity), we implement further checks for such cases by looking at the predicates associated with such blank nodes, the candidates for the blank nodes they are connected to, *etc.*, to reduce the number of relevant triples insofar as possible. All triples that are considered relevant are loaded into a dictionary-encoded in-memory index for graph matching later. [48]

In the last phase, lines 19–26, for each graph in the blank node partition, we apply basic graph pattern matching—considering the blank nodes in that graph as variables—against the in-memory index. If more than one solution is found (the identity must always be found), then the graph contains non-lean blank nodes. The solutions are checked to identify any blank nodes that are not mapped to more than one term (a term other than themselves); these blank nodes are classified as non-lean. If there is only one solution for the graph, all blank nodes are classified as non-lean.

The BGPMATCH function itself is similar to standard basic graph pattern matching, considering blank nodes as variables in the "query" graph G. However, instead of matching ground edges in the graph, the candidate set can be used instead, allowing the algorithm to focus on matching the blank edges in the graph. The candidate set is also used to generate selectivity estimates for triples in the query graph, enabling join-ordering optimisations on line 31. Solutions for blank edges are generated in order of selectivity using standard nested-loop equi-joins; on line 35, these solutions are checked to ensure that they correspond with the candidate sets for each blank node based on its ground edges. Though not shown in the algorithm, if neither the subject nor the object of the triple have been previously mapped (*i.e.*, both are unbound blank nodes), where possible, instead of querying all triples with the given predicate, we pre-bind the most selective blank node with all terms from the candidate set.

Timing The entire process took 7.41 h. The majority of time was spent scanning the data to build a new candidate set, scanning the data a second time to identify relevant triples, and dictionary encoding relevant data for loading the index. A total of 44.546 million dictionary-encoded triples were loaded into memory, taking approximately 30GB of heap-space. Once all of the data were filtered, prepared and

[48] The in-memory index consists of nested maps in two orders: $p \to \{s \to O\}$ and $p \to \{o \to S\}$. We know that predicates must be bound since blank nodes cannot appear in this triple position, hence two orders are sufficient.

```
function HOMOMORPHISM($Q''', B'_{nl}$)    ▷ from Algs. 3 & 4
    $B \leftarrow (\text{terms}(Q''') \cap \mathbf{B}) \setminus B'_{nl}$    ▷ $B$ : unclassified
    $P \leftarrow \emptyset$    ▷ in-mem
    for all $b \in B$ do
        load edge($b, Q'''$) into memory
        $P \leftarrow P \cup \pi_1(\text{edge}(b, Q'''))$    ▷ collect predicates
    init $\mathcal{M}[]$    ▷ in-mem assoc. array of candidate bindings
    for all $x \in \text{terms}(Q''')$ do
        $B_x \leftarrow \{b \in B \mid \text{gedge}(b, Q'') \subseteq \text{gedge}(x, Q'')\}$
        for all $b_x \in B_x$ do
            $\mathcal{M}[b_x] \leftarrow \mathcal{M}[b_x] \cup \{x\}$
        $SO \leftarrow \bigcup_{b \in B} \mathcal{M}[b]$    ▷ in-mem
    $\mathcal{G} \leftarrow \{G \in \text{bnp}(Q''') \mid \text{terms}(G) \cap B \neq \emptyset\}$    ▷ in-mem
    $G' \leftarrow \emptyset$    ▷ in-mem
    for all $(s, p, o) \in Q'''$ do    ▷ ignores g term in quads
        if $s \in SO \wedge p \in P \wedge o \in SO$ then    ▷ filter triples
            $G' \leftarrow G' \cup \{(s, p, o)\}$    ▷ collect relevant triples
    $B''_{nl} \leftarrow \emptyset$, $S \leftarrow \emptyset$
    for all $G \in \mathcal{G}$ do    ▷ run each bnp($Q'$) as query
        $M \leftarrow$ BGPMATCH($G, G', \mathcal{M}$)
        if $|M| \neq 1$ then    ▷ if not just the identity
            for $\mu \in M$ do
                for $(b, x) \in \mu : b \neq x$ do
                    $B''_{nl} \leftarrow B''_{nl} \cup (\text{terms}(G) \cap \{b\})$
        else
            $S \leftarrow B''_{nl} \cup (\text{terms}(G) \cap \mathbf{B})$
    $Q'''' \leftarrow$ mark all $S$ in $Q'''$
    return $Q''''$ and $B''_{nl}$
function BGPMATCH($G, G', \mathcal{M}$)    ▷ processed in memory
    $G_b \leftarrow G \cap (\mathbf{B} \times \mathbf{U} \times \mathbf{B})$    ▷ $\mathcal{M}$ binds for ground edges
    $G_b[1, \ldots, n] \leftarrow$ order $G_b$ by desc. selectivity
    $M_0 \leftarrow \{\mu_\emptyset\}$    ▷ $\mu$ : map per §2.1; $\mu_\emptyset$: blank map
    for $1 \leq i \leq n$ do
        $M'_i \leftarrow M_{i-1} \bowtie \{\mu \mid \mu(G_b[i]) \subseteq G'\}$    ▷ $\bowtie$: join
        $M_i \leftarrow \{\mu \in M'_i \mid \text{for all } (b, x) \in \mu : x \in \mathcal{M}(b)\}$
    return $M_n$    ▷ solutions for BGP matching
```

Algorithm 5. Classifying final homomorphisms

loaded, it took less than a second to perform the in-memory basic graph pattern matching required for the 13,547 graphs in the blank node partition.

Results Of the 13,547 graphs checked, 13,513 (99.75%) returned only one (identity) solution, indicating that the blank nodes in those graphs were lean. Multiple solutions were found for 34 graphs (00.25%), indicating the presence of non-lean blank nodes; of the 189 blank nodes in these graphs, 167 were confirmed as non-lean. Only one URI was found as a witness; all of the remaining mappings involved blank nodes on the same pay-level-domain, where 15 non-lean blank nodes were mapped to blank nodes in the same document (there were 5 such documents). In summary, of the 36,956 blank nodes left to classify after the previous phases, 36,789 (99.55%) were classified as lean and 167 (00.45%) were classified as non-lean. All blank nodes in the BTC–2012 data have now been classified.

7.5 Summary

In the BTC–2012 dataset, containing a sample of 8.373 million RDF documents crawled from the Web, we found a total of 88.678 million unique blank nodes. Of these, 83.299 million (93.93%) are lean and 5.378 million (6.07%) are non-lean. Of the non-lean blank nodes, the vast majority are isomorphic cases, which occur for two main reasons: (i) either documents are copied verbatim in multiple locations on the Web, typically due to quirks in how the data are hosted; or (ii) blank nodes within local documents are left "underspecified" and thus referentially ambiguous, where we would conjecture that the intent is often to refer to different real-world things with each blank node. Aside from isomorphic cases, non-lean blank nodes are relatively rare in the dataset, with very few "proper" homomorphisms found relative to the size of the data considered.

With respect to our process of classifying all of the blank nodes, we presented five phases of increasing complexity to incrementally reduce and refine the problem space. Figure 10 presents this iterative reduction, showing the distribution of the number of unclassified blank nodes in each graph of the blank node partition after every phase (log/log). We argue that every phase is necessary: each algorithm grows in complexity and in memory requirements and no algorithm could have been run without the reduction in the problem size provided by the algorithms that preceded it.

Most importantly, building upon our treewidth analysis, we have demonstrated that although in a worst-case analysis simple entailment is NP-complete and leanness-checking is coNP-complete, and although the isomorphism and homomorphism algorithms we have presented are indeed exponential, problematic exponential cases do not occur in practice. In real-world RDF graphs, blank nodes are associated with selective, ground information that can be used to restrict the search space for the graph matching problems that their existential semantics gives rise to. In this section, we have demonstrated that processing lean/non-lean blank nodes is feasible over the merge of 8.373 million real-world RDF

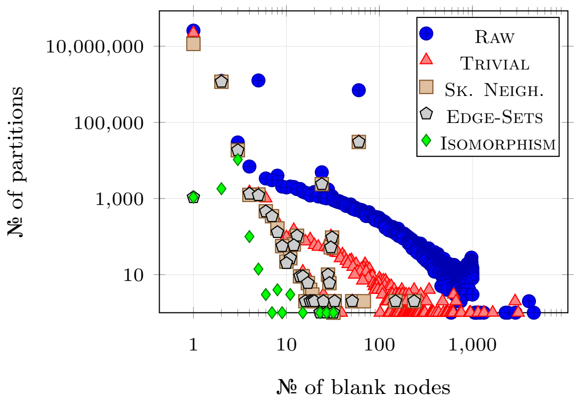


Fig. 10. Distribution of number of unclassified blank nodes in each component of the blank node partition after each phase (log/log)

graphs containing 88.678 million blank nodes. [49]

8 Alternatives for blank nodes

A number of alternatives have been proposed for how blank nodes can be treated in the blank node standards, particularly in the context of the RDF 1.1 Working Group. We now discuss some of these alternatives.

8.1 Deprecate/disallow blank nodes

The first alternative is to deprecate or disallow the use of blank nodes in RDF. However, blank nodes are a useful convenience for publishers. They enable succinct shortcuts in the various RDF syntaxes, making, for example, RDF collection shortcuts easier to read and to write in RDF/XML, Turtle and JSON-LD. They also make the assignment of URIs to resources optional, which may be useful in cases where a particular resource has a transient nature (*e.g.*, a resource representing the last access of a document or a current sensor reading), or more generally where a particular resource should not be externally referenceable (*e.g.*, closed RDF lists or n-ary predicates used for OWL axioms).

Removing blank nodes altogether would also require revision to a number of W3C standards. All of the standard RDF syntaxes would need to be revised, including the removal of shortcut syntaxes or their revision to instead use generative URIs. RDFS entailment would have to be changed to not rely on the use of surrogate blank nodes for completeness. The OWL–RDF mapping would need to be revised to not require the use of blank nodes. SPARQL would need to be revised to not allow blank nodes in the query syntax and to support generative blank nodes in `CONSTRUCT` clauses by other means, and so forth. Such revisions would be non-trivial on both a technical and social level.

It is also unclear how legacy RDF data should be handled if blank nodes were to be removed. As we have seen, blank nodes are prevalent in Web data. New versions of RDF tools would not be able to support legacy RDF data, or they would have to employ some deterministic method of creating URIs for blank nodes. This raises other questions as to how URIs should be minted for blank nodes in a consistent manner (discussed in the next section).

A much simpler alternative along these lines is to continue discouraging the "unnecessary" use of blank nodes, as per the non-normative Linked Data guidelines laid out by Heath and Bizer [37].

8.2 Ground semantics

Instead of defining blank nodes as existential variables, another option is to assign them a ground semantics, such that they are interpreted in a similar fashion to URIs. All blank nodes would thus be considered lean and constant and simple entailment would be reduced to set containment of RDF graphs. However, applying a ground semantics to blank nodes would raise some non-trivial issues.

First, if blank nodes are considered as constants local to a given scope (*e.g.*, a given RDF document or a given version of an RDF document), then an RDF graph containing blank nodes from one scope can never simple-entail an RDF graph containing blank nodes from another scope. Thus if two applications operating in different scopes parse the same document at the same time [50], these RDF graphs would not (simple-)entail each other. The notion of two graphs being semantically equivalent up to blank node labelling would no longer be a logical corollary in the case of a ground semantics.

[49] Though admittedly it was not at all straightforward in engineering terms.

[50] Or in the lingo of the RDF 1.1 Working Group, "if the two applications retrieve two RDF Graphs from the one Graph Container with one Graph Serialisation". See http://www.w3.org/2011/rdf-wg/wiki/Graph_Terminology.

Another alternative would be to consider a hybrid semantics for blank nodes where they are considered as ground within a local scope but existential across scopes. Thus for example, in the two triples "`:John :telephone _:b1 , _:b2`" from our survey, neither triple would be considered redundant under this semantics. Instead, a new operation called a "lean-merge" could be defined for combining RDF graphs in multiple scopes. The lean-merge operation would merge two RDF graphs but, roughly speaking, would remove non-lean blank nodes whose homomorphisms require data from multiple graphs. Thus, the lean-merge of two RDF graphs originating from, say, the same document would not duplicate all triples with blank nodes, but would rather preserve the cardinality of the original RDF graph. Although this would preserve all blank nodes in the original RDF graph and not consider any term as redundant, such a semantics would be even more complicated and far less intuitive than the existing semantics.

Otherwise, if blank nodes are considered as global constants similar to a URI, then schemes are required to generate identifiers for unlabelled blank nodes. Such schemes have been discussed under the heading of "Skolemisation" in the community and by the RDF 1.1 Working Group [20, 36]. The first question would be what type of identifiers these are; should blank nodes be mapped to URIs or a disjoint set of RDF terms? If URIs are used, how should the mapping from unlabelled blank nodes to global URIs be performed? It would seem unfeasible to retro-fit identity onto (unlabelled) blank nodes in an RDF graph: as the graph changes and blank nodes are reordered and edges are added/removed, identifying which blank node originated from which would be a seemingly arbitrary process. Hence novel URIs would probably need to be generated each time a document is parsed to avoid clashes; such a proposal for "Skolem IRIs" has been made in the working drafts of the upcoming RDF 1.1 standard [20, 36]. Skolem IRIs could solve issues relating to, *e.g.*, round-tripping in SPARQL discussed in Section 5.4, but would conservatively yield a large number of redundant "single-use" URIs.

8.3 Well-behaved RDF

Another option would be to restrict the use of blank nodes in RDF graphs to avoid problematic cases for simple entailment and leanness checks. In the previous version of this paper [47], we mentioned the possibility of disallowing blank node cycles in RDF graphs, potentially by disallowing blank node labels in the pertinent syntaxes.[51] This was based on the observation that the vast majority of RDF documents published on the Web at that time featured acyclical blank nodes and that such cases were easier to implement simple entailment and leanness checks for. (Since the original publication of that paper, and in the results currently presented in Section 6.2, we now find that the prominent `data.gov.uk` site is producing large numbers of cyclical blank nodes.)

Booth [14] further developed this idea into a proposal for a profile of RDF called "Well-Behaved RDF". The core motivation for this profile is to allow implementers to develop tractable lightweight methods (such as leaning, or "canonicalisation" in the words of Booth) that support the semantics of blank nodes for acyclical cases, which fall within the Well-Behaved RDF profile. These tools could then claim full compliance for Well-Behaved RDF graphs. Any RDF documents falling outside the profile (such as data from the `data.gov.uk` site) would not be well-supported by such tools.

The proposal is a practical one, saving developers implementation costs that are required to support the rather niche case of cyclical blank nodes. By defining Well-Behaved RDF as a profile, no changes would be required to the existing standards. On the other hand, the definition of Well-Behaved RDF graphs requires further refinement: Booth defines the profile as any RDF graph that can be serialised in Turtle without using explicit blank node labels. However, this is only a subset of RDF graphs with acyclical blank nodes (*e.g.*, trees expanding in a object-to-subject direction would be excluded although their implementation would be no more difficult). Furthermore, the core rationale of Well-Behaved RDF—being able to support polynomial-time entailment—applies equally to a much broader range of RDF graphs with bounded treewidth, as we have discussed in Section 6.2. Hence, the practical benefits to consumers of assuming a Well-Behaved

[51] In fact, the original 1999 W3C Recommendation for RDF did not allow such labels, where blank nodes could only form directed trees. This was later seen as a missing feature and `rdf:nodeID` was added to RDF/XML for the 2004 standard. See http://www.w3.org/2000/03/rdf-tracking/#rdfms-syntax-incomplete.

RDF profile would need more thorough analysis alongside different variations of formal definitions.

8.4 No change

Given the varied use of blank nodes in numerous standards and a large volume of published data, any change to the core semantics of blank nodes would incur a huge cost at this stage. Even if the core semantics could be conveniently changed, it is not clear what a better alternative would be. Although, for example, the process of leaning an RDF graph can change the results given by SPARQL queries, or can remove information about identical tuples in the direct mapping of RDB2RDF, leaning is an optional process and can be (and often is) ignored by practitioners. The existential semantics also succinctly captures some of the intuitive meaning of the RDF data model, including the equivalence of RDF graphs modulo arbitrary blank node labels. Furthermore, the standard semantics of blank nodes is not incompatible with the intent of publishers, but rather generalises their intents.

The problems with the existential semantics of blank nodes are thus mainly an academic issue, affecting worst-case analyses of entailment for RDF graphs. Similarly, a developer of a Semantic Web application may have to implement methods to support entailment checks, equivalence checks and leanness operations, which involves a heavy implementation cost. But applications can optionally assume a Herbrand semantics to avoid having to analyse blank node homomorphisms. Even if they choose to consider simple entailment or to perform leaning, as we have shown in this paper, not only is simple entailment tractable for all real-world graphs (which have bounded treewidth), but in practice, blank nodes are associated with rich ground information that allows for anchoring graph matching problems, reducing the search space considerably for such tasks.

The RDF 1.1 Working Group has similarly decided not to change the core semantics of blank nodes. Instead, more practical alternatives are to discourage the overuse/abuse of blank nodes and to provide Skolemisation schemes for mapping blank nodes to globally "novel" URIs. Though not explored by the RDF 1.1 Working Group, rubber-stamping profiles of RDF that enable tractable entailment could be another pragmatic compromise.

9 Conclusions

In this paper, we have provided a detailed discourse on the issue of blank nodes in RDF.

We first provided a formal background for blank nodes, relating their existential semantics to similar concepts from first-order logic and nulls in database theory. We then discussed some practical issues relating to simple entailment, where we discussed tighter bounds for the complexity of simple entailment and discussed the relation of simple entailment and leanness checking to the problem of basic graph pattern matching in SPARQL.

Next we looked through all of the standards built on top of RDF, highlighting how they support blank nodes, which of their features rely on blank nodes, and what sorts of issues blank nodes have caused. We discussed how blank nodes enable convenient shortcuts in the various RDF syntaxes, how the RDFS entailment rules use surrogate blank nodes to ensure completeness, how the OWL standard requires the use of blank nodes for n-ary predicates and lists in the RDF–OWL mapping, how SPARQL uses blank nodes in the `CONSTRUCT` as a Skolem function but may return different answers for RDF graphs that are equivalent under the RDF semantics, how the interpretation of blank nodes as non-distinguished variables causes issues for SPARQL entailment, how simple entailment can be supported under RIF BLD, and how the direct mapping in the RDB2RDF standard can produce non-lean RDF output when identical tuples without primary keys are present in the source relational table.

We then began surveying the use of blank nodes in published data. Analysing the BTC–2012 dataset, we found that 25.7% of unique terms were blank nodes, that 44.9% of documents contained at least one blank node, and that 66.2% of domains use some blank node(s). We thus concluded that blank nodes are prevalent in real-world data despite, *e.g.*, the fact that they are discouraged from use in various Linked Data guidelines [37]. The largest producers of blank nodes were the `data.gov.uk`, `freebase.com` and `livejournal.com` sites, where the former two are in the LOD cloud. Looking at the complexity of real-world blank node structures, we found that the `data.gov.uk` site produces a large number of cyclical blank graphs, but that aside from this domain, the vast majority of blank nodes form trees. The highest treewidth we found for a real-world blank graph was 6, translating into an empirical upper bound of $O(n^2 + mn^{14})$ for the simple entailment check $G \models H$ where $|G| = m$ and $|H| = n$. Otherwise, almost all blank graphs were acyclical. We then conducted a survey to see why publishers use blank nodes; based on 88 responses, we found that the current semantics of blank nodes generalised the intent of publishers when using them in 97.7% of cases.

We also looked at the prevalence of lean vs. non-lean blank nodes in the BTC–2012 dataset. Using a five-stage process designed to iteratively reduce the search space of unclassified blank nodes, we classified 93.9% of the blank nodes as lean and 6.1% as non-lean. Almost all of the non-lean cases were due to isomorphisms, where documents are replicated in multiple Web locations or where documents contain underspecified blank nodes. By computing this result, we also demonstrated that, despite being intractable, simple entailment and leaning procedures are feasible at large scale over real-world data: most blank nodes are associated with highly-selective ground information that can be used to significantly narrow the search space for homomorphisms. Furthermore, difficult cases such as graphs with high treewidth or vertex-transitivity, *etc.*, are not (often) encountered in practice.

Finally, we discussed a number of possible alternatives that have been proposed for blank nodes, by us, the community, and the RDF 1.1 Working Group. We generally concluded that any change to the core semantics of blank nodes would now incur a great cost, particularly given their use in numerous standards, their support implemented by various tools, and their presence in millions of RDF documents published on the Web. The most practical solution going forward would seem to be to leave the standard semantics of blank nodes as they are, offering methods for practitioners to optionally map them to Skolem IRIs where necessary. And indeed, this would seem to be the strategy that the RDF 1.1 Working Group have adopted [36, 20].

Acknowledgements *The work presented in this article has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Líon-2), by an IRCSET postgraduate grant, by Marie Curie action IRSES under Grant No. 24761 (Net2), by FONDECYT Grant No. 1131049 and by the Millennium Nucleus Center for Semantic Web Research under Grant No. NC120004.*

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] S. Abiteboul, P. C. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):158–187, 1991.

[3] M. Arenas, A. Bertails, E. Prud'hommeaux, and J. Sequeda. Direct Mapping of Relational Data to RDF. W3C Recommendation, Sept. 2012. http://www.w3.org/TR/rdb-direct-mapping/.

[4] M. Arenas, M. Consens, and A. Mallea. Revisiting Blank Nodes in RDF to Avoid the Semantic Mismatch with SPARQL. RDF Next Steps Workshop, June 2010.

[5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Application.* Cambridge University Press, 2002.

[6] J.-F. Baget. RDF entailment as a graph homomorphism. In *International Semantic Web Conference*, pages 82–96, 2005.

[7] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. RDF 1.1 Turtle: Terse RDF Triple Language. W3C Recommendation, Feb. 2014. http://www.w3.org/TR/turtle/.

[8] D. Beckett, G. Carothers, and A. Seaborne. RDF 1.1 N-Triples: A line-based syntax for an RDF graph. W3C Recommendation, Feb. 2014. http://www.w3.org/TR/n-triples/.

[9] D. Beckett and B. McBride. RDF/XML Syntax Specification (Revised). W3C Recommendation, Feb. 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

[10] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping between RDF and XML with XSPARQL. *J. Data Semantics*, 1(3):147–185, 2012.

[11] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data – The Story So Far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[12] H. Boley, G. Hallmark, M. Kifer, A. Paschke, A. Polleres, and D. Reynolds. RIF Core Dialect. W3C Recommendation, June 2010. http://www.w3.org/TR/rif-core/.

[13] H. Boley and M. Kifer. RIF Basic Logic Dialect. W3C Recommendation, June 2010. http://www.w3.org/TR/rif-bld/.

[14] D. Booth. Well Behaved RDF: A Straw-Man Proposal for Taming Blank Nodes, Dec. 2012. http://dbooth.org/2013/well-behaved-rdf/Booth-well-behaved-rdf.pdf.

[15] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/rdf-schema/.

[16] S. R. Buss. On Herbrand's Theorem. In D. Leivant, editor, *LCC*, volume 960 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 1994.

[17] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. Datalog±: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS*, pages 228–242, 2010.

[18] J. J. Carroll. Signing RDF Graphs. In *ISWC*, pages 369–384, 2003.

[19] A. K. Chandra and P. M. Merlin. Optimal implemen-

tation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[20] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, Feb. 2014. http://www.w3.org/TR/rdf11-concepts/.

[21] S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF Mapping Language. W3C Recommendation, Sept. 2012. http://www.w3.org/TR/r2rml/.

[22] J. de Bruijn and S. Heymans. Logical foundations of (e)rdf(s): Complexity and reasoning. In *ISWC/ASWC*, pages 86–99, 2007.

[23] J. de Bruijn and C. Welty. RIF RDF and OWL Compatibility. W3C Recommendation, June 2010. http://www.w3.org/TR/rif-rdf-owl/.

[24] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *TODS*, 30(1):174–210, 2005.

[25] B. Glimm and M. Krötzsch. Sparql beyond subgraph matching. In *International Semantic Web Conference (1)*, pages 241–256, 2010.

[26] B. Glimm and C. Ogbuji. SPARQL 1.1 Entailment Regimes. W3C Recommendation, Mar. 2013. http://www.w3.org/TR/sparql11-entailment/.

[27] V. Gogate and R. Dechter. A Complete Anytime Algorithm for Treewidth. In *UAI*, pages 201–208, 2004.

[28] G. Grahne. *The Problem of Incomplete Information in Relational Databases*, volume 554 of *Lecture Notes in Computer Science*. Springer, 1991.

[29] B. C. Grau, B. Motik, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language: Profiles. W3C Recommendation, Oct. 2009. http://www.w3.org/TR/owl2-profiles/.

[30] B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *WWW*, 2004.

[31] C. Gutierrez, C. Hurtado, and A. O. Mendelzon. Foundations of Semantic Web Databases. In *23rd ACM SIGACT-SIGMOD-SIGART*, June 2004.

[32] V. Haarslev, K. Hidde, R. Möller, and M. Wessel. The RacerPro knowledge representation and reasoning system. *Semantic Web*, 3(3):267–277, 2012.

[33] S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, Mar. 2013. http://www.w3.org/TR/sparql11-query/.

[34] S. Hawke and A. Polleres. RIF In RDF (Second Edition). W3C Working Group Note, Feb. 2013. http://www.w3.org/TR/rif-in-rdf/.

[35] P. Hayes. RDF Semantics. W3C Recommendation, Feb. 2004.

[36] P. J. Hayes and P. F. Patel-Schneider. RDF 1.1 Semantics. W3C Recommendation, Feb. 2014. http://www.w3.org/TR/rdf11-mt/.

[37] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*, volume 1. Morgan & Claypool, 2011.

[38] P. Hell and J. Nešetřil. The core of a graph. *Discrete Mathematics*, 109(1-3):127–126, 1992.

[39] I. Herman, B. Adida, M. Sporny, and M. Birbeck. RDFa 1.1 Primer – Second Edition. W3C Working Group Note, Aug. 2013. http://www.w3.org/TR/xhtml-rdfa-primer/.

[40] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[41] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing Linked Data Dynamics. In *ESWC*, pages 213–227. Springer, 2013.

[42] T. Käfer, J. Umbrich, A. Hogan, and A. Polleres. Towards a Dynamic Linked Data Observatory. In *LDOW*, 2012.

[43] P. J. Kelly. A congruence theorem for trees. *Pacific Journal of Mathematics*, 7(1), 1957.

[44] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4):741–843, 1995.

[45] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[46] I. Kollia, B. Glimm, and I. Horrocks. SPARQL query answering over OWL ontologies. In *ESWC*, volume 6643 of *LNCS*, pages 382–396. Springer, 2011.

[47] A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On blank nodes. In *International Semantic Web Conference*, pages 421–437, 2011.

[48] F. Manola, E. Miller, and B. McBride. RDF Primer. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[49] M. Marano, P. Obermeier, and A. Polleres. Processing RIF and OWL2RL within DLVHEX. In *Web Reasoning and Rule Systems - Fourth International Conference (RR2010)*, pages 244–250, 2010.

[50] B. McKay. Practical Graph Isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1980.

[51] B. Motik, P. F. Patel-Schneider, and B. C. Grau. OWL 2 Web Ontology Language Direct Semantics. W3C Recommendation, Oct. 2009. http://www.w3.org/TR/owl2-direct-semantics/.

[52] B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C Recommendation, Oct. 2009. http://www.w3.org/TR/owl2-syntax/.

[53] B. Motik, R. Shearer, and I. Horrocks. Hypertableau Reasoning for Description Logics. *J. Artif. Intell. Res. (JAIR)*, 36:165–228, 2009.

[54] S. Muñoz, J. Pérez, and C. Gutiérrez. Minimal Deductive Systems for RDF. In *ESWC*, pages 53–67, 2007.

[55] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and Efficient Minimal RDFS. *J. Web Sem.*, 7(3):220–234, 2009.

[56] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and efficient minimal RDFS. *J. Web Sem.*, 7(3):220–234, 2009.

[57] P. F. Patel-Schneider and B. Motik. OWL 2 Web Ontology Language Mapping to RDF Graphs. W3C Recommendation, Oct. 2009. http://www.w3.org/TR/owl2-direct-semantics/.

[58] R. Pichler, A. Polleres, F. Wei, and S. Woltran. dRDF: Entailment for Domain-Restricted RDF. In *ESWC*, pages 200–214, 2008.

[59] A. Polleres, F. Scharffe, and R. Schindlauer. SPARQL++ for mapping between RDF vocabularies. In *OTM 2007, Part I : Proceedings of the 6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, pages 878–

896, 2007.

[60] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, Jan. 2008. http://www.w3.org/TR/rdf-sparql-query/.

[61] M. Schneider. OWL 2 Web Ontology Language RDF-Based Semantics. W3C Recommendation, Oct. 2009. http://www.w3.org/TR/owl2-rdf-based-semantics/.

[62] J. Sequeda and D. P. Miranker. Ultrawrap: SPARQL execution on relational data. *J. Web Sem.*, 22:19–39, 2013.

[63] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[64] M. K. Smith, C. Welty, and D. L. McGuinness. OWL Web Ontology Language Guide. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/owl-guide/.

[65] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström. JSON-LD 1.0: A JSON-based Serialization for Linked Data. W3C Recommendation, Jan. 2014. http://www.w3.org/TR/json-ld/.

[66] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[67] H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *J. of Web Sem.*, 3:79–115, 2005.

[68] D. Tsarkov, I. Horrocks, and P. F. Patel-Schneider. Optimizing Terminological Reasoning for Expressive Description Logics. *J. Autom. Reasoning*, 39(3):277–316, 2007.

[69] Y. Tzitzikas, C. Lantzaki, and D. Zeginis. Blank Node Matching and RDF/S Comparison Functions. In *International Semantic Web Conference (1)*, pages 591–607, 2012.

[70] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable Distributed Reasoning Using MapReduce. In *International Semantic Web Conference*, pages 634–649, 2009.

[71] T. van Dijk, J.-P. van den Heuvel, and W. Slob. Computing treewidth with LibTW, 2006. http://www.treewidth.com/docs/LibTW.pdf.

[72] J. Weaver and J. A. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *International Semantic Web Conference (ISWC2009)*, pages 682–697, 2009.

[73] G. T. Williams. SPARQL 1.1 Service Description. W3C Recommendation, Mar. 2013. http://www.w3.org/TR/sparql11-service-description/.