Reifying RDF: What Works Well With Wikidata?

Daniel Hernández[1], Aidan Hogan[1], and Markus Krötzsch[2]

[1] Department of Computer Science, University of Chile
[2] Technische Universität Dresden, Germany

Abstract. In this paper, we compare various options for reifying RDF triples. We are motivated by the goal of representing Wikidata as RDF, which would allow legacy Semantic Web languages, techniques and tools – for example, SPARQL engines – to be used for Wikidata. However, Wikidata annotates statements with qualifiers and references, which require some notion of reification to model in RDF. We thus investigate four such options: (1) standard reification, (2) n-ary relations, (3) singleton properties, and (4) named graphs. Taking a recent dump of Wikidata, we generate the four RDF datasets pertaining to each model and discuss high-level aspects relating to data sizes, etc. To empirically compare the effect of the different models on query times, we collect a set of benchmark queries with four model-specific versions of each query. We present the results of running these queries against five popular SPARQL implementations: 4store, BlazeGraph, GraphDB, Jena TDB and Virtuoso.

1 Introduction

Wikidata is a collaboratively edited knowledge-base under development by the Wikimedia foundation whose aim is to curate and represent the factual information of Wikipedia (across all languages) in an interoperable, machine-readable format [20]. Until now, such factual information has been embedded within millions of Wikipedia articles spanning hundreds of languages, often with a high degree of overlap. Although initiatives like DBpedia [15] and YAGO [13] have generated influential knowledge-bases by applying custom extraction frameworks over Wikipedia, the often ad hoc way in which structured data is embedded in Wikipedia limits the amount of data that can be cleanly captured. Likewise, when information is gathered from multiple articles or multiple language versions of Wikipedia, the results may not always be coherent: facts that are mirrored in multiple places must be manually curated and updated by human editors, meaning that they may not always correspond at a given point in time.

Therefore, by allowing human editors to collaboratively add, edit and curate a centralised, structured knowledge-base directly, the goal of Wikidata is to keep a single consistent version of factual data relevant to Wikipedia. The resulting knowledge-base is not only useful to Wikipedia – for example, for automatically generating articles that list entities conforming to a certain restriction (e.g., female heads of state) or for generating infobox data consistently across all languages – but also to the Web community in general. Since the launch of Wikidata

in October 2012, more than 80 thousand editors have contributed information on 18 million entities (data items and properties). In comparison, English Wikipedia has around 6 million pages, 4.5 million of which are considered proper articles. As of July 2015, Wikidata has gathered over 65 million statements.

One of the next goals of the Wikidata project is to explore methods by which the public can query the knowledge-base. The Wikidata developers are currently investigating the use of RDF as an exchange format for Wikidata with SPARQL query functionality. Indeed, the factual information of Wikidata corresponds quite closely with the RDF data model, where the main data item (entity) can be viewed as the subject of a triple and the attribute–value pairs associated with that item can be mapped naturally to predicates and objects associated with the subject. However, Wikidata also allows editors to annotate attribute–value pairs with additional information, such as qualifiers and references. Qualifiers provide context for the validity of the statement in question, for example providing a time period during which the statement was true. References point to authoritative sources from which the statement can be verified. About half of the statements in Wikidata (32.5 million) already provide a reference, and it is an important goal of the project to further increase this number.

Hence, to represent Wikidata in RDF while capturing meta-information such as qualifiers and references, we need some way in RDF to describe the RDF triples themselves (which herein we will refer to as "*reification*" in the general sense of the term, as distinct from the specific proposal for reification defined in the 2004 RDF standard [3], which we refer to as "*standard reification*").

In relation to Wikidata, we need a method that is compatible with existing Semantic Web standards and tools, and that does not consider the domain of triple annotation as fixed in any way: in other words, it does not fix the domain of annotations to time, or provenance, or so forth [22]. With respect to general methods for reification within RDF, we identify four main options:

standard reification (SR) whereby an RDF resource is used to denote the triple itself, denoting its subject, predicate and object as attributes and allowing additional meta-information to be added [16,4].

n-ary relations (NR) whereby an intermediate resource is used to denote the relationship, allowing it to be annotated with meta-information [16,8].

singleton properties (SP) whereby a predicate unique to the statement is created, which can be linked to the high-level predicate indicating the relationship, and onto which can be added additional meta-information [18].

Named Graphs (NG) whereby triples (or sets thereof) can be identified in a fourth field using, e.g., an IRI, onto which meta-information is added [10,5].

Any of these four options would allow the qualified statements in the Wikidata knowledge-base to be represented and processed using current Semantic Web norms. In fact, Erxleben et al. [8] previously proposed an RDF representation of the Wikidata knowledge-base using a form of n-ary relations. It is important to note that while the first three formats rely solely on the core RDF model, Named Graphs represents an extension of the traditional triple model,

adding a fourth element; however, the notion of Named Graphs is well-supported in the SPARQL standard [10], and as "RDF Datasets" in RDF 1.1 [5].

Thus arises the core question tackled in this paper: *what are the relative strengths and weaknesses of each of the four formats?* We are particularly interested in exploring this question quantitatively with respect to Wikidata. We thus take a recent dump of Wikidata and create four RDF datasets: one for each of the formats listed above. The focus of this preliminary paper is to gain empirical insights on how these formats affect query execution times for off-the-shelf tools. We thus (attempt to) load these four datasets into five popular SPARQL engines – namely 4store [9], BlazeGraph (formerly BigData) [19], GraphDB (formerly (Big)OWLIM) [2], Jena TDB [21], and Virtuoso [7] – and apply four versions of a query benchmark containing 14 queries to each dataset in each engine.

2 The Wikidata Data-model

Figure 1a provides an example statement taken from Wikidata describing the entity Abraham Lincoln. We show internal identifiers in grey, where those beginning with Q refer to entities, and those referring to P refer to properties. These identifiers map to IRIs, where information about that entity or relationship can be found. All entities and relationships are also associated with labels, where the English versions are shown for readability. Values of properties may also be datatype literals, as exemplified with the dates.

The snippet contains a primary relation, with Abraham Lincoln as subject, position held as predicate, and President of the United States of America as object. Such binary relations are naturally representable in RDF. However, the statement is also associated with some *qualifiers* and their *values*. Qualifiers are property terms such as start time, follows, etc., whose values may scope the validity of the statement and/or provide additional context. Additionally, statements are often associated with one or more *references* that support the claims and with a *rank* that marks the most important statements for a given property. The details are not relevant to our research: we can treat references and ranks as special types of qualifiers. We use the term *statement* to refer to a primary relation and its associated qualifications; e.g., Fig. 1a illustrates a single statement.

Conceptually, one could view Wikidata as a "Property Graph": a directed labelled graph where edges themselves can have attributes and values [11,6]. A related idea would be to consider Wikidata as consisting of *quins* of the form (s, p, o, q, v), where (s, p, o) refers to the *primary relation*, q is a *qualifier property*, and v is a *qualifier value* [12]. Referring to Fig. 1a again, we could encode a quin (`:Q91`, `:P39`, `:Q11696`, `:P580`, `"1861/03/14"^^xsd:date`), which states that Abraham Lincoln had relation position held to President of the United States of America under the qualifier property start time with the qualifier value 4 March 1861. All quins with a common primary relation would constitute a statement. However, quins of this form are not a suitable format for Wikidata since a given primary relation may be associated with different groupings of qualifiers. For example, Grover Cleveland was President of the United States for two non-consecutive

terms (i.e., with different start and end times, different predecessors and successors). In Wikidata, this is represented as two separate statements whose primary relations are both identical, but where the qualifiers (start time, end time, follows, followed by) differ. For this reason, reification schemes based conceptually on quins – such as RDF* [12,11] – may not be directly suitable for Wikidata.

A tempting fix might be to add an additional column and represent Wikidata using sextuples of the form (s, p, o, q, v, i) where i is an added statement identifier. Thus in the case of Grover Cleveland, his two non-consecutive terms would be represented as two statements with two distinct statement identifiers. While in principle sextuples would be sufficient, in practice (i) the relation itself may contain NULLs, since some statements do not currently have qualifiers or may not even support qualifiers (as is the case with labels, for example), (ii) qualifier values may themselves be complex and require some description: for example, dates may be associated with time precisions or calendars.[3]

For this reason, we propose to view Wikidata conceptually in terms of two tables: one containing quads of the form (s, p, o, i) where (s, p, o) is a primary relation and i is an identifier for that statement; the other a triple table storing (i) primary relations that can never be qualified (e.g., labels) and thus do not need to be identified, (ii) triples of the form (i, q, v) that specify the qualifiers associated to a statement, and (iii) triples of the form (v, x, y) that further describe the properties of qualifier values. Table 1 provides an example that encodes some of the data seen in Fig. 1a – as well as some further type information not shown – into two such tables: the quads table on the left encodes qualifiable primary relations with an identifier, and the triples table on the right encodes (i) qualifications using the statement identifiers, (ii) non-qualifiable primary relations, such as those that specify labels, and (iii) type information for complex values, such as to provide a precision, calendar, etc.

Compared to sextuples, the quad/triple schema only costs one additional tuple per statement, will lead to dense instances (even if some qualifiable primary relations are not currently qualified), and will not repeat the primary relation for each qualifier; conversely, the quad/triple schema may require more joins for certain query patterns (e.g., find primary relations with a follows qualifier).

Likewise, the quad/triple schema is quite close to an RDF-compatible encoding. As per Fig. 1b, the triples from Table 1 are already an RDF graph; we can thus focus on encoding quads of the form (s, p, o, i) in RDF.

3 From Higher Arity Data to RDF (and back)

The question now is: how can we represent the statements of Wikidata as triples in RDF? Furthermore: how many triples would we *need* per statement? And how might we know for certain that we don't lose something in the translation?

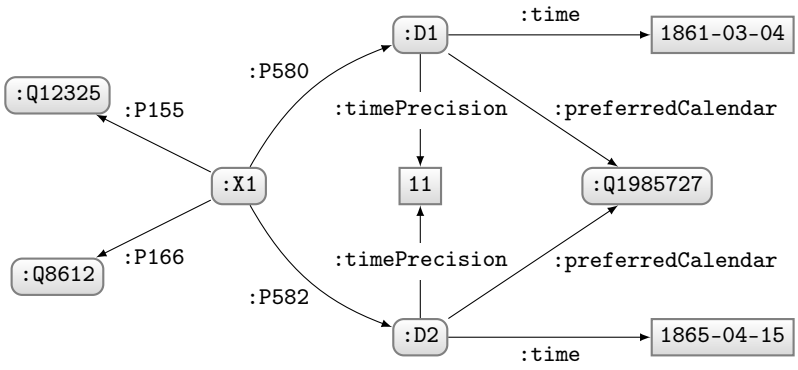
The transformation from Wikidata to RDF can be seen as an instance of schema translation, where these questions then directly relate to the area of

[3] See `https://www.wikidata.org/wiki/Special:ListDatatypes`; retr. 2015/07/11.

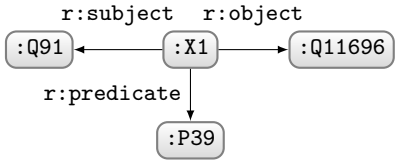
Abraham Lincoln [Q91]

position held [P39]	President of the United States of America [Q11696]
start time [P580]	"4 March 1861"
end time [P582]	"15 April 1865"
follows [P155]	James Buchanan [Q12325]
followed by [P156]	Andrew Johnson [Q8612]

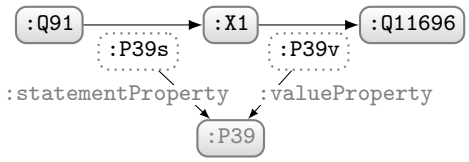
(a) Raw Wikidata format



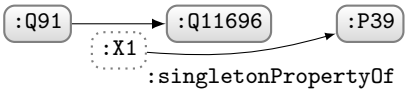
(b) Qualifier information common to all formats



(c) Standard reification



(d) n-ary relations



(e) Singleton properties

(f) Named Graphs

Fig. 1: Reification examples

Table 1: Using quads and triples to encode Wikidata

Quads

s	p	o	i
:Q91	:P39	:Q11696	:X1
...	...	...	...

Triples

s	p	o
:X1	:P580	:D1
:X1	:P582	:D2
:X1	:P155	:Q12325
:X1	:P156	:Q8612
...	...	...
:D1	:time	"1861-03-04"^^xsd:date
:D1	:timePrecision	"11"
:D1	:preferredCalendar	:Q1985727
...	...	...
:Q91	rdfs:label	"Abraham Lincoln"@en
...	...	...

relative information capacity in the database community [14,17], which studies how one can translate from one database schema to another, and what sorts of guarantees can be made based on such a translation. Miller et al. [17] relate some of the theoretical notions of information capacity to practical guarantees for common schema translation tasks. In this view, the task of translating from Wikidata to RDF is a unidirectional scenario: we want to query a Wikidata instance through an RDF view without loss of information, and to recover the Wikidata instance from the RDF view, but we do not require, e.g., updates on the RDF view to be reflected in Wikidata. We therefore need a transformation whereby the RDF view dominates the Wikidata schema, meaning that the transformation must map a unique instance of Wikidata to a unique RDF view.

We can formalise this by considering an instance of Wikidata as a database with the schema shown in Table 1. We require that any instance of Wikidata can be mapped to a RDF graph, and that any conjunctive query (CQ; select-project-join query in SQL) over the Wikidata instance can be translated to a conjunctive query over the RDF graph that returns the same answers. We call such a translation *query dominating*. We do not further restrict how translations are to be specified so long as they are well-defined and computable.

A query dominating translation of a relation of arity 4 (R_4) to a unique instance of a relation of arity 3 (R_3) must lead to a higher number of tuples in the target instance. In general, we can always achieve this by encoding a quad into four triples of the form (s, p_y, o_z), where s is a unique ID for the quad, p_y denotes a position in the quad (where $1 \leq y \leq 4$), and o_z is the term appearing in that position of the quad in R_4.

Example 1. *The quad* `:Q91 :P39 :Q11696 :X1` *can be mapped to four triples:*

```
:S1 :P1 :Q91
:S1 :P2 :P39
:S1 :P3 :Q11696
:S1 :P4 :X1
```

Any conjunctive query over any set of quads can be translated into a conjunctive query over the corresponding triple instance that returns the same answer: for each tuple in the former query, add four tuples to the latter query with a fresh common subject variable, with `:P1` *...* `:P4` *as predicate, and with the corresponding terms from the sextuple (be they constants or variables) as objects. The fresh subject variables can be made existential/projected away.* □

With this scheme, we require $4k$ triples to encode k quads. This encoding can be generalised to encode database tables of arbitrary arity, which is essentially the approach taken by the *Direct Mapping* of relational databases to RDF [1].

Quad encodings that use fewer than four triples usually require additional assumptions. The above encoding requires the availability of an unlimited amount of identifiers, which is always given in RDF, where there is an infinite supply of IRIs. However, the technique does not assume that auxiliary identifiers such as `:S1` or `:P4` do not occur elsewhere: even if these IRIs are used in the given set of quads, their use in the object position of triples would not cause any confusion. If we make the additional assumption that some "reserved" identifiers are not used in the input quad data, we can find encodings with fewer triples per quad.

Example 2. *If we assume that IRIs* `:P1` *and* `:P2` *are not used in the input instance, the quad of Example 1 can be encoded in the following three triples:*

```
:S1 :P1     :Q91
:S1 :P2     :P39
:S1 :Q11696 :X1
```

The translation is not faithful when the initial assumption is violated. For example, the encoding of the quad `:Q91 :P39 :P1 :Q92` *would contain triples* `:S2 :P1 :Q91` *and* `:S2 :P1 :Q92`*, which would be ambiguous.* □

The assumption of some reserved IRIs is still viable in practice, and indeed three of the encoding approaches we look at in the following section assume some reserved vocabulary to be available. Using suitable naming strategies, one can prevent ambiguities. Other domain-specific assumptions are also sometimes used to define suitable encodings. For example, when constructing quads from Wikidata, the statement identifiers that are the fourth component of each quad functionally determine the first three components, and this can be used to simplify the encoding. We will highlight these assumptions as appropriate.

4 Existing Reification Approaches

In this section, we discuss how legacy reification-style approaches can be leveraged to model Wikidata in RDF, where we have seen that all that remains is to model quads in RDF. We also discuss the natural option of using named graphs, which support quads directly. The various approaches are illustrated in Fig. 1, where 1b shows the qualifier information common to all approaches, i.e., the triple data, while 1c–1f show alternative encodings of quads.

Standard Reification The first approach we look at is standard RDF reification [16,4], where a resource is used to denote the statement, and where additional information about the statement can be added. The scheme is depicted in Fig. 1c. To represent a quad of the form (s, p, o, i), we add the following triples: $(i, \texttt{r:subject}, s)$, $(i, \texttt{r:predicate}, p)$, $(i, \texttt{r:object}, o)$, where `r:` is the RDF vocabulary namespace. We omit the redundant declaration as type `r:Statement`, which can be inferred from the domain of `r:subject`. Moreover, we simplify the encoding by using the Wikidata statement identifier as subject, rather than using a blank node. We can therefore represent n quadruples with $3n$ triples.

***n*-ary Relation** The second approach is to use an n-ary relation style of modelling, where a resource is used to identify a relationship. Such a scheme is depicted in Fig. 1d, which follows the proposal by Erxleben et al. [8]. Instead of stating that a subject has a given value, the model states that the subject is involved in a relationship, and that that relationship has a value and some qualifiers. The `:subjectProperty` and `:valueProperty` edges are important to be able to query for the original name of the property holding between a given subject and object.[4] For identifying the relationship, we can simply use the statement identifier. To represent a quadruple of the form (s, p, o, i), we must add the triples (s, p_s, i), (i, p_v, o), $(p_v, \texttt{:valueProperty}, p)$, $(p_s, \texttt{:statementProperty}, p)$, where p_v and p_s are fresh properties created from p. To represent n quadruples of the form (s, p, o, i), with m unique values for p, we thus need $2(n+m)$ triples. Note that for the translation to be query dominating, we must assume that predicates such as `:P39s` and `:P39v` in Fig. 1c, as well as the reserved terms `:statementProperty` and `:valueProperty`, do not appear in the input.

Singleton Properties Originally proposed by Nguyen et al. [18], the core idea behind singleton properties is to create a property that is only used for a single statement, which can then be used to annotate more information about the statement. The idea is captured in Fig. 1e. To represent a quadruple of the form (s, p, o, i), we must add the triples (s, i, o), $(i, \texttt{:singletonPropertyOf}, p)$. Thus to represent n quadruples, we need $2n$ triples, making this the most concise scheme so far. To be query dominating, we must assume that the term `:singletonPropertyOf` cannot appear as a statement identifier.

Named Graphs Unlike the previous three schemes, Named Graphs extends the RDF triple model and considers sets of pairs of the form (G, n) where G is an RDF graph and n is an IRI (or a blank node in some cases, or can even be omitted for a *default graph*). We can flatten this representation by taking the union over $G \times \{n\}$ for each such pair, resulting in quadruples. Thus we can encode a quadruple (s, p, o, i) directly using N-Quads, as illustrated in Fig. 1f.

[4] Referring to Fig. 1c, another option to save some triples might be to use the original property `:P39` (position held) instead of `:P39s` or `:P39v`, but this could be conceptually ugly since, e.g., if we replaced `:P39s`, the resulting triple would be effectively stating that (Abraham Lincoln,position held,[*a statement identifier*]).

Table 2: Number of triples needed to model quads ($n = 57,088,184$, $p = 1,311$)

SCHEMA:	SR ($3n$)	NR ($2(n+p)$)	SP ($2n$)	NG (n)
TUPLES:	171,264,552	114,178,990	114,176,368	57,088,184

Other possibilities? Having introduced the most well-known options for reification, one may ask if these are all the reasonable alternatives for representing quadruples of the form (s, p, o, i) – where i functionally determines (s, p, o) – in a manner compatible with the RDF standards. As demonstrated by singleton properties, we can encode such quads into two triples, where i appears somewhere in both triples, and where s, p and o each appear in one of the four remaining positions, and where a reserved term is used to fill the last position. This gives us 108 possible schemes[5] that use two triples to represent a quad in a similar manner to the singleton properties proposal. Just to take one example, we could model such a quad in two triples as $(i, \texttt{r:subject}, s), (i, p, o)$—an abbreviated form of standard reification. As before, we should assume that the properties p and `r:subject` are distinct from qualifier properties. Likewise, if we are not so concerned with conciseness and allow a third triple, the possibilities increase further. To reiterate, our current goal is to evaluate existing, well-known proposals, but we wish to mention that many other such possibilities do exist in theory.

5 SPARQL Querying Experiments

We have looked at four well-known approaches to annotate triples, in terms of how they are formed, what assumptions they make, and how many triples they require. In this section, we aim to see how these proposals work in practice, particularly in the context of querying Wikidata. Experiments were run on an Intel E5-2407 Quad-Core 2.2GHz machine with a standard SATA hard-drive and 32 GB of RAM. More details about the configuration of these experiments are available from `http://users.dcc.uchile.cl/~dhernand/wrdf/`.

To start with, we took the RDF export of Wikidata from Erxleben et al. [8] (2015-02-23), which was natively in an n-ary relation style format, and built the equivalent data for all four datasets. The number of triples common to all formats was 237.6 million. With respect to representing the quads, Table 2 provides a breakdown of the number of output tuples for each model.

[5] There are 3! possibilities for where i appears in the two triples: *ss*, *pp*, *oo*, *sp*, *so*, *po*. For the latter three configurations, all four remaining slots are distinguished so we have 4! ways to slot in the last four terms. For the former three configurations, both triples are thus far identical, so we only have half the slots, making 4×3 permutations. Putting it all together, $3 \times 4! + 3 \times 4 \times 3 = 108$.

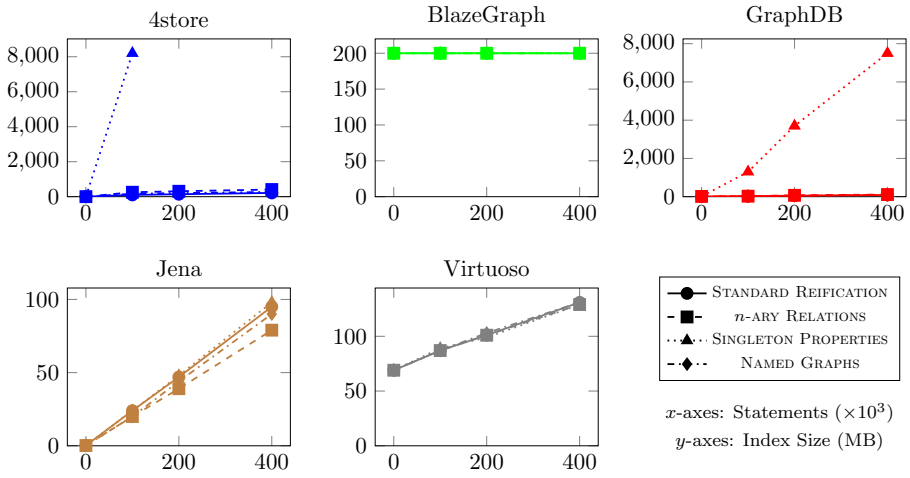


Fig. 2: Growth in index sizes for first 400,000 statements

Loading data: We selected five RDF engines for experiments: 4store, BlazeGraph, GraphDB, Jena and Virtuoso. The first step was to load the four datasets for the four models into each engine. We immediately started encountering problems with some of the engines. To quantify these issues, we created three collections of 100,000, 200,000, and 400,000 raw statements and converted them into the four models.[6] We then tried to load these twelve files into each engine. The resulting growth in on-disk index sizes is illustrated in Figure 2 (measured from the database directory), where we see that: (i) even though different models lead to different triple counts, index sizes were often nearly identical: we believe that since the entropy of the data is quite similar, compression manages to factor out the redundant repetitions in the models; (ii) some of the indexes start with some space allocated, where in fact for BlazeGraph, the initial allocation of disk space (200MB) was not affected by the data loads; (iii) 4store and GraphDB both ran into problems when loading singleton properties, where it seems the indexing schemes used assume a low number of unique predicates.[7] With respect to point (iii), given that even small samples lead to blow-ups in index sizes, we decided not to proceed with indexing the singleton properties dataset in 4store or GraphDB. While later loading the full named graphs dataset, 4store slowed to loading 24 triples/second; we thus also had to kill that index load.[8]

[6] We do not report the times for full index builds since, due to time constraints, we often ran these in parallel uncontrolled settings.

[7] In the case of 4store, for example, in the database directory, two new files were created for each predicate.

[8] See `https://groups.google.com/forum/#!topic/4store-support/uv8yHrb-ng4`; retr. 2015/07/11.

Benchmark queries: From two online lists of test-case SPARQL queries, we selected a total of 14 benchmark queries.[9] These are listed in Table 3; since we need to create four versions of each query for each reification model, we use an abstract quad syntax where necessary, which will be expanded in a model-specific way such that the queries will return the same answers over each model. An example of a quad in the abstract syntax and its four expansions is provided in Table 4. In a similar manner, the 14 queries of Table 3 are used to generate four equivalent query benchmarks for testing, making a total of 56 queries.

In terms of the queries themselves, they exhibit a variety of query features and number/type of joins; some involve qualifier information while some do not. Some of the queries are related; for example, Q1 and Q2 both ask for information about US presidents, and Q4 and Q5 both ask about female mayors. Importantly, Q4 and Q5 both require use of a SPARQL property path (`:P31/:P279*`), which we cannot capture appropriately in the abstract syntax. In fact, these property paths cannot be expressed in either the singleton properties model or the standard reification model; they can only be expressed in the n-ary relation model (`:P31s/:P31v/(:P279s/:P279v)*`), and the named graph model (`:P31/:P279*`) *assuming* the default graph can be set as the union of all graphs (since one cannot do property paths across graphs in SPARQL, only within graphs).

Query results: For each engine and each model, we ran the queries sequentially (Q1–14) five times on a cold index. Since the engines had varying degrees of caching behaviour after the first run – which is not the focus of this paper – we present times for the first "cold" run of each query.[10] Since we aim to run $14 \times 4 \times 5 \times 5 = 1,400$ query executions, to keep the experiment duration manageable, all engines were configured for a timeout of 60 seconds. Since different engines interpret timeouts differently (e.g., connection timeouts, overall timeouts, etc.), we considered any query taking longer than 60 seconds to run as a timeout. We also briefly inspected results to see that they corresponded with equivalent runs, ruling queries that returned truncated results as failed executions.[11]

The query times for all five engines and four models are reported in Figure 3, where the y-axis is in log scale from 100 ms to 60,000 ms (the timeout) in all cases for ease of comparison. Query times are not shown in cases where the query could not be run (for example, the index could not be built as discussed previously, or property-paths could not be specified for that model, or in the case of 4store, certain query features were not supported), or where the query failed (with a timeout, a partial result, or a server exception). We see that in terms of engines, Virtuoso provides the most reliable/performant results across all models. Jena failed to return answers for singleton properties, timing-out on all queries (we expect some aspect of the query processing does not perform well

[9] `https://www.mediawiki.org/wiki/Wikibase/Indexing/SPARQL_Query_Examples` and `http://wikidata.metaphacts.com/resource/Samples`

[10] Data for other runs are available from the web-page linked earlier.

[11] The results for queries such as Q5, Q7 and Q14 may (validly) return different answers for different executions due to use of `LIMIT` without an explicit order.

Table 3: Evaluation queries in abstract syntax

#**Q1**: US presidents and their wives

```
SELECT ?up ?w ?l ?wl WHERE { <:Q30 :P6 ?up _:i1> . <?up :P26 ?w ?i2> . OPTIONAL {
   ?up rs:label ?l . ?w rs:label ?wl . FILTER(lang(?l) = "en" && lang(?wl) = "en") } }
```

#**Q2**: US presidents and causes of death

```
SELECT ?h ?c ?hl ?cl WHERE { <?h :P39 :Q11696 ?i1> .  <?h :P509 ?c ?i2> . OPTIONAL {
 ?h rs:label ?hl . ?c rs:label ?cl . FILTER(lang(?hl) = "en" && lang(?cl) = "en") } }
```

#**Q3**: People born before 1880 with no death date

```
SELECT ?h ?date WHERE { <?h :P31 :Q5 ?i1> . <?h :P569 ?dateS ?i2> . ?dateS :time ?date .
 FILTER NOT EXISTS { ?h :P570s [ :P570v ?d ] . }
 FILTER (datatype(?date) = xsd:date && ?date < "1880-01-01Z"^^xsd:date) } LIMIT 100
```

#**Q4**: Cities with female mayors ordered by population

```
SELECT DISTINCT ?city ?citylabel ?mayorlabel (MAX(?pop) AS ?max_pop) WHERE {
 ?city :P31/:P279* :Q515 . <?city :P6 ?mayor ?i1> . FILTER NOT EXISTS { ?i1 :P582q ?x }
 <?mayor :P21 :Q6581072 _:i2> .  <?city :P1082 ?pop _:i3> .  ?pop :numericValue ?pop .
 OPTIONAL { ?city rs:label ?citylabel . FILTER ( LANG(?citylabel) = "en" ) }
 OPTIONAL { ?mayor rs:label ?mayorlabel . FILTER ( LANG(?mayorlabel) = "en" ) } }
GROUP BY ?city ?citylabel ?mayorlabel ORDER BY DESC(?max_pop) LIMIT 10
```

#**Q5**: Countries ordered by number of female city mayors

```
SELECT ?country ?label (COUNT(*) as ?count) WHERE { ?city :P31/:P279* :Q515 .
 <?city :P6 ?mayor ?i1> . FILTER NOT EXISTS { ?i1 :P582q ?x }
 <?mayor :P21 :Q6581072 ?i2> .  <?city :P17 ?country ?i3> .  ?pop :numericValue ?pop .
 OPTIONAL { ?country rs:label ?label .  FILTER ( LANG(?label) = "en" ) } }
GROUP BY ?country ?label ORDER BY DESC(?count) LIMIT 100
```

#**Q6**: US states ordered by number of neighbouring states

```
SELECT ?state ?label ?borders WHERE { { SELECT ?state (COUNT(?neigh) as ?borders)
   WHERE { <?state :P31 :Q35657 ?i1> . <?neigh :P47 ?state _:i2> .
    <?neigh :P31 :Q35657 ?i3 > . } GROUP BY ?state }
 OPTIONAL { ?state rs:label ?label . FILTER(lang(?label) = "en") } } ORDER BY DESC(?borders)
```

#**Q7**: People whose birthday is “today”

```
SELECT DISTINCT ?entity ?year  WHERE { <?entityS :P569 ?value ?i1> . ?value :time ?date .
 ?entityS rs:label ?entity . FILTER(lang(?entity)="en")
 FILTER(date(?date)=month(now()) && date(?date)=day(now())) } LIMIT 10
```

#**Q8**: All property–value pairs for Douglas Adams

```
SELECT ?property ?value  WHERE {  <:Q42 ?property ?value ?i> }
```

#**Q9**: Populations of Berlin, ordered by least recent

```
SELECT ?pop ?time WHERE { <:Q64 wd:P1082 ?popS ?i> . ?popS :numericValue ?pop .
 ?i wd:P585q [ :time ?time ] . } ORDER BY (?time)
```

#**Q10**: Countries without an end-date

```
SELECT ?country ?countryName WHERE { <?country wd:P31 wd:Q3624078 ?i> .
 FILTER NOT EXISTS { ?g :P582q ?endDate } ?country rdfs:label ?countryName .
 FILTER(lang(?countryName)="en") }
```

#**Q11**: US Presidents and their terms, ordered by start-date

```
SELECT ?president ?start ?end WHERE { <:Q30 :P6 ?president ?i > .
 ?g :P580q [ :time ?start ] ; :P582q [ :time ?end ] . }  ORDER BY (?start)
```

#**Q12**: All qualifier properties used with "head of government" property

```
SELECT DISTINCT ?q_p WHERE { <?s :P6 ?o ?i > . ?g ?q_p ?q_v . }
```

#**Q13**: Current countries ordered by most neighbours

```
SELECT ?countryName (COUNT (DISTINCT ?neighbor) AS ?neighbors) WHERE {
 <?country :P31 :Q3624078 ?i1> . FILTER NOT EXISTS { ?i1 :P582 ?endDate }
 ?country rs:label ?countryName FILTER(lang(?countryName)="en") OPTIONAL {
  <?country :P47 ?neighbor ?i2 > <?neighbor :P31 :Q3624078 ?i3> .
  FILTER NOT EXISTS { ?i3 :P582q ?endDate2 } } }
GROUP BY(?countryName) ORDER BY DESC(?neighbors)
```

#**Q14**: People who have Wikidata accounts

```
SELECT ?person ?name ?uname WHERE { <?person :P553 wd:Q52 ?i > .
 ?i :P554q ?uname . ?person rs:label ?name . FILTER(LANG(?name) = "en") . } LIMIT 100
```

Table 4: Expansion of abstract syntax quad for getting US presidents

ABSTRACT SYNTAX:	`<:Q30 :P6 ?up ?i> .`
STD. REIFICATION:	`?i r:subject :Q30 ; r:predicate :P6 ; r:object ?up .`
n-ARY RELATIONS:	`:Q30 :P6s ?i . ?i :P6v ?up .`
SING. PROPERTIES:	`:Q30 ?i ?up . ?i1 r:singletonPropertyOf :P6 .`
NAMED GRAPHS:	`GRAPH ?i { :Q30 :P6 ?up . }`

assuming many unique predicates). We see that both BlazeGraph and GraphDB managed to process most of the queries for the indexes we could build, but with longer runtimes than Virtuoso. In general, 4store struggled with the benchmark and returned few valid responses in the allotted time.

Returning to our focus in terms of comparing the four reification models, Table 5 provides a summary of how the models ranked for each engine and overall. For a given engine and query, we look at which model performed best, counting the number of firsts, seconds, thirds, fourths, failures (FA) and cases where the query could not be run (NR). For example, referring back to Figure 3, we see that for Virtuoso, Q1, the fastest models in order were standard reification, named graphs, singleton properties, n-ary relations. Thus, in Table 5, under Virtuoso, we add a one to standard reification in the 1st column, a one to named graphs in the 2st column, and so forth. Along these lines, for example, the score of 4 for singleton-properties (SP) in the 1st column of Virtuoso means that this model successfully returned results faster than all other models in 4 out of the 14 cases. The total column then adds the positions for all engines. From this, we see that looking across all five engines, named graphs is probably the best supported (fastest in 17/70 cases), with standard reification and and n-ary relations not far behind (fastest in 16/70 cases). All engines aside from Virtuoso seem to struggle with singleton properties; presumably these engines make some (arguably naive) assumptions that the number of unique predicates in the indexed data is low.

Although the first three RDF-level formats would typically require more joins to be executed than named graphs, the joins in question are through the statement identifier, which is highly selective; assuming "equivalent" query plans, the increased joins are unlikely to overtly affect performance, particularly when forming part of a more complex query. However, additional joins do complicate query planning, where aspects of different data models may affect established techniques for query optimisation differently. In general, we speculate that in cases where a particular model was much slower for a particular query and engine, that the engine in question selected a (comparatively) worse query plan.

6 Conclusions

In this paper, we have looked at four well-known reification models for RDF: standard reification, n-ary relations, singleton properties and named graphs. We were particularly interested in the goal of modelling Wikidata as RDF, such that

Table 5: Ranking of reification models for query response times

№	4store				BlazeGraph				GraphDB				Jena				Virtuoso				**Total**			
	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG
1st	2	3	0	0	6	2	0	3	2	2	0	7	2	7	0	3	4	2	4	4	16	16	4	17
2rd	2	1	0	0	1	3	2	4	5	2	0	4	6	0	0	5	5	2	1	6	19	8	3	19
3rd	–	–	–	–	3	3	1	2	4	7	0	0	2	2	0	3	3	3	2	3	12	15	3	8
4th	–	–	–	–	0	1	6	2	–	–	–	–	0	0	0	0	2	4	4	1	2	5	10	3
FA	5	5	0	0	4	3	3	3	3	1	0	3	4	3	12	3	0	1	1	0	16	13	16	9
NR	5	5	14	14	0	2	2	0	0	2	14	0	0	2	2	0	0	2	2	0	5	13	34	14

it can be indexed and queried by existing SPARQL technologies. We sketched a conceptual overview of a Wikidata schema based on quads/triples, thus reducing the goal of modelling Wikidata to that of modelling quads in RDF (quads where the fourth element functionally specifies the triple), and introduced the four reification models in this context. We found that singleton-properties offered the most concise representation on a triple level, but that n-ary predicates was the only model with built-in support for SPARQL property paths. With respect to experiments over five SPARQL engines – 4store, BlazeGraph, GraphDB, Jena and Virtuoso – we found that the former four engines struggled with the high number of unique predicates generated by singleton properties, and that 4store likewise struggled with a high number of named graphs. Otherwise, in terms of query performance, we found no clear winner between standard reification, n-ary predicates and named graphs. We hope that these results may be insightful for Wikidata developers – and other practitioners – who wish to select a practical scheme for querying reified RDF data.

Acknowledgements This work was partially funded by the DFG in projects DIAMOND (Emmy Noether grant KR 4381/1-1) and HAEC (CRC 912), by the Millennium Nucleus Center for Semantic Web Research under Grant No. NC120004 and by Fondecyt Grant No. 11140900.

References

1. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J. (eds.): Direct Mapping of Relational Data to RDF. W3C Recommendation (27 September 2012), `http://www.w3.org/TR/rdb-direct-mapping/`
2. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: a family of scalable semantic repositories. Sem. Web J. 2(1), 33–42 (2011)
3. Brickley, D., Guha, R. (eds.): RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (10 February 2004), `http://www.w3.org/TR/rdf-schema/`
4. Brickley, D., Guha, R. (eds.): RDF Schema 1.1. W3C Recommendation (25 February 2014), `http://www.w3.org/TR/rdf-schema/`
5. Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J.J., McBride, B. (eds.): RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (25 February 2014), `http://www.w3.org/TR/rdf11-concepts/`

6. Das, S., Srinivasan, J., Perry, M., Chong, E.I., Banerjee, J.: A tale of two graphs: Property Graphs as RDF in Oracle. In: EDBT. pp. 762–773 (2014), http://dx.doi.org/10.5441/002/edbt.2014.82
7. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. 35(1), 3–8 (2012)
8. Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., Vrandečić, D.: Introducing Wikidata to the linked data web. In: ISWC. pp. 50–65 (2014)
9. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. In: Workshop on Scalable Semantic Web Systems. CEUR-WS, vol. 517, pp. 94–109 (2009)
10. Harris, S., Seaborne, A., Prud'hommeaux, E. (eds.): SPARQL 1.1 Query Language. W3C Recommendation (21 March 2013), http://www.w3.org/TR/sparql11-query/
11. Hartig, O.: Reconciliation of RDF* and Property Graphs. CoRR abs/1409.3288 (2014), http://arxiv.org/abs/1409.3288
12. Hartig, O., Thompson, B.: Foundations of an alternative approach to reification in RDF. CoRR abs/1406.3399 (2014), http://arxiv.org/abs/1406.3399
13. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. Artif. Intell. 194, 28–61 (2013)
14. Hull, R.: Relative information capacity of simple relational database schemata. In: PODS. pp. 97–109 (1984)
15. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. Sem. Web J. 6(2), 167–195 (2015)
16. Manola, F., Miller, E. (eds.): Resource Description Framework (RDF): Primer. W3C Recommendation (10 February 2004), http://www.w3.org/TR/rdf-primer/
17. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: Schema equivalence in heterogeneous systems: bridging theory and practice. Inf. Syst. 19(1), 3–31 (1994)
18. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like RDF reification? Making statements about statements using singleton property. In: WWW. pp. 759–770. ACM (2014)
19. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF graph database. In: Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014)
20. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Comm. ACM 57, 78–85 (2014)
21. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D., Ding, L.: Supporting scalable, persistent Semantic Web applications. IEEE Data Eng. Bull. 26(4), 33–39 (2003)
22. Zimmermann, A., Lopes, N., Polleres, A., Straccia, U.: A general framework for representing, reasoning and querying with annotated Semantic Web data. J. Web Sem. 11, 72–95 (2012)

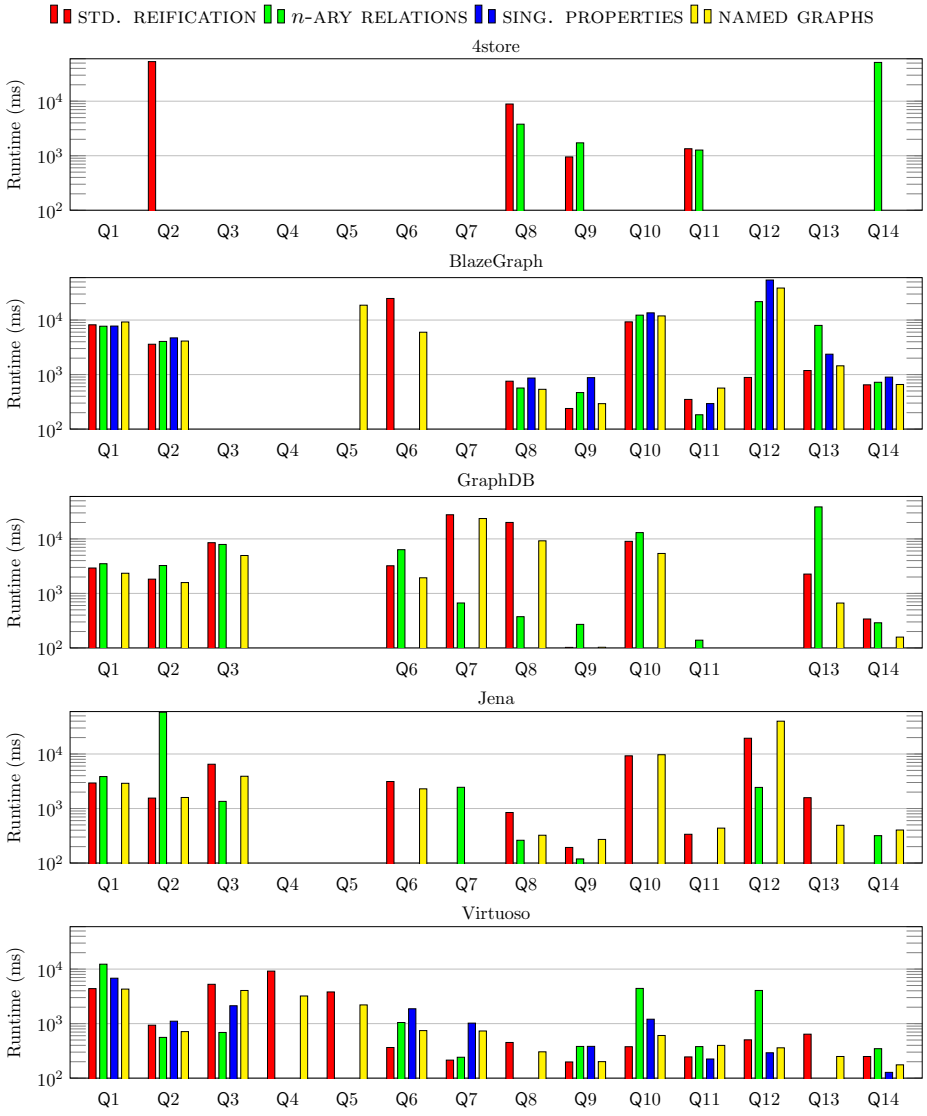


Fig. 3: Query results for all five engines and four models (log scale)