Towards a Binary Object Notation for RDF

Victor Charpenay[1,2], Sebastian Käbisch[1], and Harald Kosch[2]

[1] Siemens AG — Corporate Technology
victor.charpenay@siemens.com, sebastian.kaebisch@siemens.com
[2] Universität Passau — Fakultät für Informatik und Mathematik
harald.kosch@uni-passau.de

Abstract. The recent JSON-LD standard, that specifies an object notation for RDF, has been adopted by a number of data providers on the Web. In this paper, we present a novel usage of JSON-LD, as a compact format to exchange and query RDF data in constrained environments, in the context of the Web of Things.
A typical exchange between Web of Things agents involves small pieces of semantically described data (RDF data sets of less than hundred triples). In this context, we show how JSON-LD, serialized in binary JSON formats like EXI4JSON and CBOR, outperforms the state-of-the-art. Our experiments were performed on data sets provided by the literature, as well as a production data set exported from Siemens Desigo CC.
We also provide a formalism for JSON-LD and show how it offers a lightweight alternative to SPARQL via JSON-LD framing (with polynomial complexity), which makes it a good candidate as a query mechanism in constrained environments.

Keywords: Web of Things, Internet of Things, SPARQL, RDF, EXI, JSON-LD, HDT, CBOR

1 Introduction

The mismatch between the triple structure of RDF and the object-oriented nature of most programming language on the Web has probably hindered the development of Semantic Web technologies. A standard object notation for RDF, to e.g. manipulate RDF data directly in JavaScript, was a prerequisite for further adoption. Such a standard, JSON for Linked Data (JSON-LD), was recently published by the W3C [26].

JSON-LD documents must link to a *context* that maps terms to Semantic Web entities. Given a proper context, documents can be expanded, compacted or flattened [18]. The standard also includes a bidirectional transformation with RDF triples. A JSON-based query language to match and reshape JSON-LD documents was also requested by the community but it has not been included in the official W3C standard. A new community draft (JSON-LD 1.1) attemps to fill the gap by defining a procedure known as JSON-LD *framing* [25].

JSON-LD has rapidly been adopted by a number of data providers, such as Open Data platforms, as well as by major browser vendors via schema.org [3]. It has also become the default exchange format for Linked Data Notifications, which are e.g. implemented by Mastodon, a successful decentralized microblogging platform[4].

Most reported usages of JSON-LD involve large-scale data exchange[5]. In this paper, we explore the usage of JSON-LD as a compact RDF serialization, in the context of the Internet of Things (IoT). It is envisioned that Web technologies will sustain the massive integration of embedded devices to the Internet, by using RESTful architectures, with JSON as a primary serialization format [8,27]. In what would then become a Web of 'Things', Semantic Web technologies would allow connected devices to be self-descriptive and autonomous [2]. Therefore, JSON-LD appears to be a good candidate as a serialization format for the Web of Things (WoT).

The major issues with using Semantic Web technologies in an embedded environment are (1) the verbosity of RDF and (2) the complexity of processing knowledge. In this paper, we show how JSON-LD can adress both issues. We first provide the theoretical foundations for JSON-LD compaction and framing (Sect. 3). In particular, we show how JSON-LD framing is equivalent to a subset of SPARQL of low complexity, hence a good fit for constrained devices. Then, we experimentally show how JSON-LD compaction using a global context can reduce the size of RDF documents (Sect. 4). Before introducing our work, we present a short overview of the state-of-the-art with respect to using RDF in constrained environments (Sect. 2).

2 Related Work

Until today, a larg part of research towards storing and querying RDF has focused on very large, static datasets stored on powerful machines, sometimes involving parallel computation. In contrast, storage mechanisms for resource-constrained devices remain mostly unexplored. Until recently, no realistic use case could be found where computational devices had limited resources but still IP connectivity. The situation has changed with the coming of the IoT where RDF is predicted to play a significant role.

In our context, constrained devices correspond mostly to low-power microcontrollers (MCUs) with integrated IP communication stack [4]. Typically, such devices are too constrained to support standard Web technologies (HTTP, XML, JSON). A range of technologies were however developed in the last years as a substitute. The Constrained Application Protocol (CoAP), the Efficient XML Interchange format (EXI) and the Constrained Binary Object Representation (CBOR) are the counterpart of HTTP, XML and JSON in the so-called "Embedded Web" [23].

[3] `http://schema.org`
[4] `https://joinmastodon.org/`
[5] `https://github.com/json-ld/json-ld.org/wiki/Users-of-JSON-LD`

The European project SPITFIRE [**?**] paved the way to the use of RDF on the Embedded Web, combining it with architectural principles of the Web of Things. Since then, several methods were proposed to serialize and process RDF data on constrained environments.

2.1 Serializing RDF Data

The first work that addressed constrained devices – and to the best of our knowledge, the only one – is part of SPITFIRE and is called the Wiselib TupleStore [12]. Built on top of Wiselib, a substitute to the C++ standard library designed for embedded systems, the Wiselib TupleStore internally stores RDF nodes in a tree-shaped data structure in order to compact them. In SPITFIRE, the Wiselib TupleStore was ported on wireless motes (iSense, MICAz, ...) and the data was serialized in a binary format called SHDT [11], a streaming version of the Header-Dictionary-Triples format for RDF (HDT).

The original objective of HDT was to compact large RDF datasets. e.g. to fit in the main memory of a standard PC. But as a binary format, its compression scheme could reasonably be used on small datasets as well. A HDT document is divided into three sections containing respectively metadata (header), resource IRIs (dictionary) and the triples themselves, indexed by subject (triples). Although most RDF stores also implement a similar partitioning, HDT compresses each section separately with dedicated compression methods [7].

In the original proposal for HDT, all triples are merged in an single array while separations between them are stored in a bitmap, easily compressible. HDT achieves high compression ratios compared to classical compression schemes like gzip. An alternative triple indexing method was proposed, performing vertical partitioning with k^2-tree compression (k^2-triples) [1]. k^2-triples achieves better compression ratios while efficiently processing predicate-bound triple pattern matching queries.

HDT and k^2-triples show excellent results in terms of compression and query processing speed. However, a recent study suggests that there exists better alternative for small data sets, i.e. on embedded devices [15]. The study presents data sets of semantically annotated sensor measurements where an EXI serialization of RDF/XML data is more compact than HDT. The two approaches (EXI and HDT) have been combined in a proposal called Efficient RDF Interchange (ERI) [6], which, however, primarily addresses data streams (with time considerations).

A last contribution which is worth mentioning here is the Entity Notation (EN) [22], developed to exchange RDF in a distributed fashion (e.g. among a resource-constrained robot swarm). EN introduces templates to compact RDF. Templates can be derived from domain knowledge, possibly automatically form OWL ontologies. An EN document includes a header referencing a template and a body which only includes the variable parts of the template instance. Further compression is achieved by putting type statements at the beginning of an EN document, so that no explicit type predicate is required.

EN is the closest approach to what we introduce in the present paper in terms of data structure. An EN entity corresponds to a JSON-LD node object,

up to the syntax. However, our approach, which consists of serializing JSON-LD data in a binary format, exclusively relies on in-use standards.

2.2 Processing RDF Data

In most use cases of SPITFIRE, the RDF data exposed by IoT devices was crawled by a Web agent and put in a central RDF repository, which clients could query through a SPARQL endpoint. In practice, IoT networks can be very heterogeneous and complex. A more dynamic approach to querying might be preferable over static crawling.

Without any change on the SPITFIRE infrastructure, one possible approach is what is referred to in the literature as Linked Data (LD) queries [10]. To answer an LD query, Web clients follow links exposed by an RDF source according to a global strategy defined by the client (e.g. based on statistics from the Linked Open Data cloud). This type of processing might be encouraged by the recent standardization by the W3C of Linked Data Platforms (LDPs) [24]. A mapping of LDPs to CoAP has been proposed [19], which in theory makes LD queries a possible candidate for RDF data exchange on contrained environments.

In parallel with LDPs, another architecture has been proposed to expose RDF datasets at low cost: Triple Pattern Fragments (TPFs) [28]. TPFs try to provide an intermediary in terms of computational cost between a full SPARQL endpoint and a LDP that serves static data. In contrast to a SPARQL endpoint, a TPF endpoint restricts queries to triple patterns. As for LD queries, Web clients must define a strategy to order atomic triple patterns and reconstruct the result set from the received fragments. A TPF server must also expose metadata (stastistics) to guide clients.

Both LD queries and TPFs have the drawback of generating many exchanges between the client and the (constrained) server. Given that low-power devices consume most of their energy on data exchange, especially with radio protocols, these approaches would considerably lower their overall lifetime. We addressed this issue in a previous work on a SPARQL engine for MCUs [5]. This engine, which we called the μRDF store, is capable of answering basic graph patterns (BGPs) and exposes a CoAP interface with RDF/EXI serialization. We implemented it for the ESP8266, a microcontroller with an integrated Wi-Fi chip (64kB RAM, 80 MHz).Our experiments show that BGP processing, including EXI coding, is fast in comparison to sending and receiving the data over Wi-Fi, while significantly reducing the amount of data exchanged. In the present work, we show how the μRDF store could support SPARQL beyond BGPs, by implementing JSON-LD framing.

Another work on embededded systems reported similar observations: shifting SPARQL joins (AND operator) to the edges in a wireless sensor network results in 5 to 10 times faster query processing [3]. Although these results apply to exchanges based on non-IP radio protocols with restricted frame size – thus not directly comparable to our results –, it is consistent with our results and encourages more experiments in this direction.

3 Theoretical Background

Although JSON-LD 1.0 officially became a W3C recommandation in 2014, it is still a moving standard. Its version 1.1 is under active development. To the best of our knowledge, no formalim has been provided yet for either JSON-LD 1.0 or JSON-LD 1.1. In the following, we formalize parts of JSON-LD 1.1 (generic syntax, compaction and framing), without aiming at exhaustivity.

3.1 Definitions

Our formalism is based on the flattened representation of a JSON-LD document, i.e. an array of node objects without nesting. The notation we use is borrowed from F-logic, an extension of predicate logic for object-oriented programming [17]. Predicate logic defines terms and formulas. F-logic extends the definition of formulas to include class membership and object fields, where object identifiers, class identifiers and field names and values are constant terms.

Definition 1. *Let I, B, L be pairwise disjoint sets of IRIs, blank nodes and literals, respectively. A JSON-LD (node) object is the formula*

$$\mathsf{id} : \mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{l} \, [\mathsf{p}_1 \rightarrow \mathsf{id}_1, \ldots, \mathsf{p}_\mathsf{m} \rightarrow \mathsf{id}_\mathsf{m}, \mathsf{p}_{\mathsf{m}+1} \rightarrow \mathsf{v}_1, \ldots, \mathsf{p}_\mathsf{n} \rightarrow \mathsf{v}_{\mathsf{n}-\mathsf{m}}] \tag{1}$$

where $\mathsf{id}, \mathsf{id}_\mathsf{i} \in (I \cup B)$, $\mathsf{t}_\mathsf{i}, \mathsf{p}_\mathsf{j} \in I$ *and* $\mathsf{v}_\mathsf{k} \in L$.

In the JSON-LD terminology, id is an object identifier, t a type, p a property and v a value. A JSON-LD graph is a set of JSON-LD objects. It can include several formulas with the same object identifier.

Example 1. The following graph represents a room equiped with a temperature sensor (IRI namespaces were omitted for the sake of conciseness):

r46 : Thing, Room [hasDataPoint → temp21],
r46 [hasName → `'Room 46 (conference room)'`],
device137 : Thing, Sensor [measures → temp21],
temp21 : Property, Temperature [hasUnit → degreeCelsius]

We now define a straightforward transformation from JSON-LD to RDF, denoted σ_R. This transformation is based on the RDF deserialization algorithm of the JSON-LD processor. In the original JSON-LD syntax, properties are allowed to be blank nodes, in which case they should be ignored during RDF deserialization. We exclude blank node properties in the present work.

Definition 2. *Let F be the set of formulas as per Definition 1. Let $f \in F$ be a formula. We define σ_R, as follows:*

$$\sigma_R(f) = \{(\mathsf{id}, \mathit{type}, \mathsf{t}_1), \ldots, (\mathsf{id}, \mathsf{p}_1, \mathsf{id}_1), \ldots, (\mathsf{id}, \mathsf{p}_\mathsf{n}, \mathsf{v}_{\mathsf{n}-\mathsf{m}})\} \tag{2}$$

$\sigma_R(f)$ is an RDF graph (*type* being the RDF type predicate IRI). It is trivial to define the inverse transformation, from RDF to JSON-LD, along the lines of the serialization algorithm of the JSON-LD processor.

Example 2. By applying σ_R on all formulas of Example 1, we obtain the following merged graph:

$\{(\mathsf{r46}, \text{type}, \mathsf{Thing}), \quad (\mathsf{r46}, \text{type}, \mathsf{Room}),$
$(\mathsf{r46}, \mathsf{hasDataPoint}, \mathsf{temp21}), \quad (\mathsf{r46}, \mathsf{hasName}, \texttt{'Room 46 (conference room)'}),$
$(\mathsf{device137}, \text{type}, \mathsf{Thing}), \quad (\mathsf{device137}, \text{type}, \mathsf{Sensor}),$
$(\mathsf{device137}, \mathsf{measures}, \mathsf{temp21}), \quad (\mathsf{temp21}, \text{type}, \mathsf{Property}),$
$(\mathsf{temp21}, \text{type}, \mathsf{Temperature}), \quad (\mathsf{temp21}, \mathsf{hasUnit}, \mathsf{degreeCelsius})\}$

Compaction Compaction consists in mapping IRIs appearing in a JSON-LD object to arbitrary (short) UTF-8 character strings. The IRI map required to turn a compacted JSON-LD object into its original form is called a JSON-LD *context*. Compaction is one of the procedures defined by the JSON-LD processor [18].

Definition 3. *Let U be the set of UTF-8 character strings. A JSON-LD context is a map $c : U \mapsto I$. By abuse of notation, we denote $c(f)$ the result of applying c to all IRIs in the formula f.*

Definition 4. *Let c be a JSON-LD context. The formula f' is a compacted JSON-LD (node) object against c if $c(f')$ is a formula as per Definition 1 (i.e. if $c(f') \in F$).*

It is possible to define a global context c_g that would apply to any formula f, such that JSON-LD interchange and procedures like framing or RDF serialization can be done on the compacted form directly. The notion of global context is key in optimizing the exchange and processing of JSON-LD data.

Example 3. As an example, one can define a simple global context that removes all characters from an IRI but the first letter(s) of its local name (assuming it introduces no name collision). A compacted form of Example 1 would then look like the following:

$\mathsf{r46} : \mathsf{T}, \mathsf{R}\,[\mathsf{hdp} \rightarrow \mathsf{temp21}],$
$\mathsf{r46}\,[\mathsf{hn} \rightarrow \texttt{'Room 46 (conference room)'}],$
$\mathsf{device137} : \mathsf{T}, \mathsf{S}\,[\mathsf{m} \rightarrow \mathsf{temp21}],$
$\mathsf{temp21} : \mathsf{P}, \mathsf{T}\,[\mathsf{hu} \rightarrow \mathsf{dc}]$

In practice, the ontology for a given domain of application includes a finite set of concepts and properties. One can therefore build a context c_g by assigning the shortest possible UTF-8 string to all IRIs of this ontology (or set of ontologies). In Sect. 4, we present experimental results on the performances of this compaction technique.

Framing JSON-LD framing is not part of the official W3C recommendation. The formalism we present in the following is based on the latest community draft for JSON-LD 1.1, as of December 2017 [25]. JSON-LD framing includes two aspects: frame matching and re-shaping, analogous to SPARQL's SELECT and CONSTRUCT clauses. In the following, we only consider frame matching.

Let V be the set of variables (equivalent to the set of SPARQL variables). Intuitively, a frame is a set of JSON-LD objects with variables.

Definition 5. *A JSON-LD frame object f^* is one of the following formulas:*

1. none, *which matches no JSON-LD node object;*
2. $\mathsf{wildcard}$, *which matches any JSON-LD node object;*
3. $\mathsf{id}^* : \mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{l}\,[\mathsf{p}_1 \to \mathsf{id}_1^*, \ldots, \mathsf{p}_\mathsf{m} \to \mathsf{id}_\mathsf{m}^*, \mathsf{p}_{\mathsf{m}+1} \to \mathsf{v}_1, \ldots, \mathsf{p}_\mathsf{n} \to \mathsf{v}_{\mathsf{n}-\mathsf{m}}]$

where $\mathsf{id}^*, \mathsf{id}_\mathsf{i}^* \in V \cup I \cup \{\mathsf{none}, \mathsf{wildcard}\}$. *We denote the set of such formulas F^*.*

A JSON-LD frame is a set of flagged JSON-LD frame objects, that is, of pairs $(f^*, b) \in F^* \times \{\text{true}, \text{false}\}$. The boolean flag b indicates that either all properties or at least one property of a frame object should match a node object. If b is *false*, any property can match. This flag is referred to as the 'require all' flag in JSON-LD 1.1.

Example 4. The following frame should match any room that contains a sensor or an actuator (or both):
$(\mathsf{room}^* : \mathsf{Room}\,[\mathsf{hasDataPoint} \to \mathsf{dp}^*], \mathit{true})$,
$(\mathsf{device}^* : \mathsf{Sensor}, \mathsf{Actuator}\,[\mathsf{measures} \to \mathsf{dp}^*, \mathsf{actsOn} \to \mathsf{dp}^*], \mathit{false})$,
$(\mathsf{dp}^*[\mathsf{hasUnit} \to \mathsf{unit}^*], \mathit{true})$

The current JSON-LD 1.1 specification lacks clarity on how to match blank nodes. In the algorithm specification, as well in its reference implementation[6], a frame object identified by a blank node will only match node objects with the same blank node identifier. Yet, blank nodes are supposed to have a local scope, that is, they cannot be shared between RDF graphs. We therefore introduced variables, per analogy with SPARQL. In practice, when evaluating a frame against a graph, query variables can be defined as blank nodes that are disjoint with the set of blank nodes in the graph.

3.2 Semantics and Complexity

Given the transformation from Definition 2, it is straightforward to define JSON-LD entailment according to the well-defined RDF semantics [13,9]. We will focus here on the semantics of JSON-LD framing only. Although we borrowed aspects of F-logic in our formalism, F-structures and object semantics are not relevant in the present work.

Before defining the semantics of JSON-LD framing, we define the transformation σ_F from a JSON-LD frame ϕ to SPARQL. We use the algebraic syntax of Pérez et al. to express SPARQL graph patterns [21].

Definition 6. *Let ϕ be a frame, i.e. a set of pairs $(f_h^*, b_h), 1 \leq h \leq |\phi|$. The transformation function σ_F is defined as follows:*

$$\sigma_F(\phi) = P_1 \ \mathit{AND} \ \ldots \ \mathit{AND} \ P_{|\phi|} \tag{3}$$

[6] `https://github.com/ruby-rdf/json-ld/`

where each P_h is the graph pattern

$$P_h = (((\mathsf{id}^*, \mathit{type}, \mathsf{t_1})\ \mathit{UNION}\ \ldots\ \mathit{UNION}\ (\mathsf{id}^*, \mathit{type}, \mathsf{t_l}))\ \mathit{AND}\ P) \tag{4}$$

such that

$$P = \begin{cases} P_1\ \mathit{AND}\ \ldots\ \mathit{AND}\ P_n, \mathit{if}\ b_h\ \mathit{is\ true} \\ P_1\ \mathit{UNION}\ \ldots\ \mathit{UNION}\ P_n, \mathit{otherwise} \end{cases} \tag{5}$$

where each $P_i, i \leq m$, a graph pattern of the form

$$P_i = \begin{cases} (\mathsf{id}^*, \mathsf{p_i}, \mathsf{id}_\mathsf{i}^*), \mathit{if}\ \mathsf{id}_\mathsf{i}^* \in (V \cup I) \\ (\mathsf{id}^*, \mathsf{p_i}, \mathit{?wildcard}_i), \mathit{if}\ \mathsf{id}_\mathsf{i}^*\ \mathit{is}\ \mathsf{wildcard} \\ (\mathsf{id}^*, \mathsf{p_i}, \mathit{?none}_i)\ \mathit{FILTER}\ (\neg \mathit{bound}(\mathit{?none}_i)), \mathit{if}\ \mathsf{id}_\mathsf{i}^*\ \mathit{is}\ \mathsf{none} \end{cases} \tag{6}$$

and each $P_i, i > m$ is the triple pattern $(\mathsf{id}^*, \mathsf{p_i}, \mathsf{v_{i-m}})$.

Example 5. The frame of Example 4 maps to the following SPARQL query:
(((?room, type, Room) AND (?room, hasDataPoint, ?dp)) AND
(((?device, type, Sensor) UNION (?device, type, Actuator)) AND
((?device, measures, ?dp) UNION (?device, actsOn, ?dp))) AND
(dp?, hasUnit, ?unit))

The range of σ_F is comprised in the subset of SPARQL that includes three operators (UNION, AND, FILTER) and two filtering predicates ($\neg$, *bound*). However, the two are not equivalent. For instance, there is no frame ϕ such that $\sigma_F(\phi) = ((?s_1, ?p_1, ?o_1)$ UNION $(?s_2, ?p_2, ?o_2))$.

It is also interesting to note that, despite the existence of the frame object none which can exclude node objects, it is not necessary to include the operator NOT in the definition of σ_F.

The transformation σ_F allows us to base the semantics of JSON-LD framing on the semantics of SPARQL, which relies on the definition of a mapping μ and an evaluation function $[\![\cdot]\!]_G$ [21]. Here, we override the definition of μ as follows: a mapping $\mu : V \mapsto I \cup B$ as a a partial function representing a match for a frame ϕ against a graph γ (a set of node objects). We also define the evaluation of ϕ against γ as a function, denoted *eval*.

Definition 7. *Let ϕ be a frame, let γ be a graph.*

$$\mathit{eval}(\phi, \gamma) = [\![\sigma_F(\phi)]\!]_{\sigma_R(\gamma)} \tag{7}$$

Example 6. Applying the room frame (Example 4) on the room graph (Example 1) should return the mapping μ, such that $\mu(\mathsf{room}^*) = \mathsf{r46}$, $\mu(\mathsf{device}^*) = \mathsf{device137}$, $\mu(\mathsf{dp}^*) = \mathsf{temp21}$ and $\mu(\mathsf{unit}^*) = \mathsf{degreeCelsius}$.

Proposition 1. *The semantics we define here is equivalent to the procedural semantics of the W3C JSON-LD 1.1 framing algorithm.*

Some elements exposed here differ from the JSON-LD 1.1 draft. For instance, we define the 'require all' flag on a per-object basis while the W3C algorithm defines it once for the whole frame. It is, however, possible to rewrite any frame as per Definition 5 as a (more verbose) frame ϕ such that all flags $b_h, 1 \leq h \leq |\phi|$, are set to the same value.

Similarly, for the sake of simplicity, we did not consider value pattern matching and identifier matching in our formalism. These aspects, however, are mostly syntactic sugar and do not influence the general semantics of framing.

To conclude with theoretical considerations, we give an insight into the complexity of JSON-LD framing evaluation. In particular, the next theorem shows that a SPARQL normal form can be computed in polynomial time for any JSON-LD frame. A normal form is a disjunction of UNION-free graph patterns.

Theorem 1. *Let ϕ be a frame. Let η be the number of frame objects in ϕ, such that the 'require all' flag is false, let l, n be the maximum number of types and properties (respectively) in a node object of ϕ. A SPARQL normal form can be computed for $\sigma_F(\phi)$ in $O(\eta \cdot l \cdot n)$.*

Proof. As a preliminary, we recall that the AND and UNION operators are associative and commutative. Moreover, the following equivalence holds [21]: $(P_1 \text{ AND } (P_2 \text{ UNION } P_3)) \equiv ((P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ AND } P_3))$.

More generally, one can observe that if P_1, P_2 are both normal forms with respectively i and j UNION-free patterns, then $(P_1 \text{ AND } P_2)$ has an equivalent normal form with $i \cdot j$ patterns. This applies in particular to any P_h as defined in Equation 4 ($l \cdot n$ patterns).

$\sigma_F(\phi)$ can be rewritten by commuting AND clauses until there exists an index η, such that $\forall 1 \leq h \leq \eta, b_h$ is *false* and $\forall \eta < h \leq |\phi|, b_h$ is *true*. Let $\overline{P_\eta}$ be the pattern $(P_{\eta+1} \text{ AND } \ldots \text{ AND } P_{|\phi|})$. $\overline{P_\eta}$ is in normal form. Using the same observation as above, we can obtain a normal form NF_η for $(P_\eta \text{ AND } \overline{P_\eta})$. The same applies to $(P_{\eta-1} \text{ AND } \overline{P_{\eta-1}}) \equiv (P_{\eta-1} \text{ AND } \text{NF}_\eta)$, recursively. We can therefore obtain a normal form for $\sigma_F(\phi)$ in η iterations.

The UNION-free patterns we obtain from ϕ include only the operators AND and FILTER. I.e., these graph patterns are equivalent to conjunctive queries (BGPs). For a fixed query, evaluating conjunctive queries can be done in polynomial time [9]. In contrast, evaluating a query with arbitrary UNION patterns is NP-complete [21]. JSON-LD framing is therefore equivalent to an intermediate subset of SPARQL that includes the UNION operator but guarantees polynomial evaluation.

3.3 Summary

Our primary motivation in the context of this paper is the exchange and processing of RDF data on embedded devices (MCUs). The main limitation of MCUs is the low amount of RAM available (8 to 64 kB). In this section, we highlighted two important aspects of JSON-LD that can be leveraged to reduce the memory footprint of RDF processing: (1) it is possible to process JSON-LD data

in a compacted form, given a global context c_g and (2) JSON-LD framing is a lightweight alternative to SPARQL that can be reduced to conjunctive query answering in polynomial time.

4 Experiments & Discussion

We implemented JSON-LD framing on the ESP8266. This implementation extends our previous work on the μRDF store, which already demonstrated the feasibility of answering conjunctive queries (AND, FILTER) on an MCU. JSON-LD frames and graphs are processed in their compacted form and no transformation to RDF or SPARQL is required: our implementation of the matching algorithm uses a native comparator between frame objects and node objects.

In the following, we concentrate on the first aspect mentioned above: JSON-LD compaction. As underlined in Sect. 2, the state-of-the-art in binary formats for RDF includes two main approaches: HDT and RDF/EXI. JSON-LD compaction allows for a third alternative: encoding JSON-LD in its compacted form (using a global context) in a binary JSON format. The binary formats we selected for these experiments are EXI for JSON (EXI4JSON) and CBOR. EXI4JSON is an extension of EXI to represent JSON documents in binary XML [20].

Building a global context, as mentioned in Sect. 3.1, is somewhat arbitrary. For instance, one could either choose to include all ontological concepts exposed on the Semantic Web in a single context or tailor an application-specific context covering a limited set of ontologies. In our experiments, we generated two contexts for every test set. The first one acts as a 'minimal' context, for comparison purposes: for a test set with n distinct IRIs, the minimal context maps the n first UTF-8 characters to them. The second context acts as a more realistic one: for a test set with n distinct IRIs defined in m distinct vocabularies, the second context maps the $|vocab_1| + \ldots + |vocab_m|$ first UTF-8 characters to all IRIs found in thes vocabularies. Compaction using this context will not perform as good as with the minimal context but it better estimates the performances one should expect in a production environment.

We compared the compaction performances of the three approaches on four data sets: a sample from the Billion Triples Challenge (BTCSAMPLE), a single sensor measurement (NODE), the output of a proxy service for sensor data (SSP) and an export of a building model from the Siemens Desigo CC platform (DESIGO). The first three data sets were provided by Hasemann et al. in the context of SPITFIRE. We added the last one to provide a comparison on real-world data, exported from a production environment. We provide statistics on these data sets on Table 1. For each of them, the set of ontologies we used to build a realistic JSON-LD context is as follows:

BTCSAMPLE – RDF, FOAF, DC, SIOC, SKOS, SWIVT;
NODE – RDF, RDFS, OWL, DC, SPITFIRE, SSNX, DUL, QUDT;
SSP – RDF, RDFS, OWL, DC, SPITFIRE, SSNX, DUL, SWEET;
DESIGO – RDF, CC, CCBA, SAREF, TD

Table 1: Test Set Statistics

	BTCSAMPLE	NODE	SSP	DESIGO
Nb. of triples	174	73	4859	84908
Nb of distinct subjects/nodes	174	26	1345	21527
Nb. of pieces	-	-	313	1843
Avg. nb. of triples (piecewise)	-	-	15	70
Min. nb. of triples (piecewise)	-	-	13	5
Max. nb. of triples (piecewise)	-	-	19	662

4.1 SPITFIRE Data Sets

The BTCSAMPLE data set includes 174 random triples from a large social network. All subject, predicate and object IRIs are disjoint. These IRIs come from a variety of vocabularies, such as FOAF[7], GeoNames[8] or SKOS[9]. The main purpose of BTCSAMPLE as a test set is to evaluate how the different approaches perform on IRI compaction, regardless of the data structure.

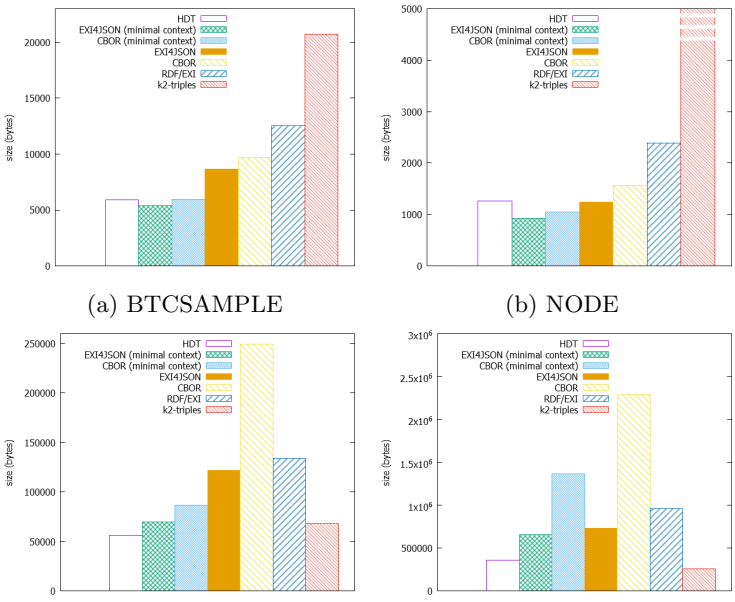
In contrast to BTCSAMPLE, NODE and SSP mostly use the SSNX vocabulary[10] to express the semantics of sensor measurements. The former is a small data set produced by one single sensor (73 triples) while the latter includes the measurements of hundreds of sensing devices (4859 triples). SSP shows many redundances in the data. Both NODE and SSP are realistic WoT data sets.

4.2 Desigo CC Data Set

Our last data test, which we denote DESIGO, was generated from a simulated Siemens building managed by the Desigo CC platform, at a real scale. The data is exported from Desigo CC as a collection of WoT Thing Descriptions (TDs) [14]; the examples of Sect. 3 are taken from this data set. A TD is a document providing pointers to Web resources allowing a WoT client to interact with the server exposing them. These Web resources can e.g. encapsulate measurement values (room temperature reading), commands (start/stop ventilation) or events (fire alarm). The data set, which includes around ten thousand data points, is the sum of 1843 interlinked TDs.

In the Web of Things architecture, TD documents are not meant to be centralized in a unique data set. Instead, every IoT device should carry its own TD as a self-descriptive agent, so that machine-to-machine interaction is possible [16]. We therefore consider DESIGO both as a whole and as an aggregation of separate pieces of information, scattered across a network. Similarly, SSP is an aggregation of sensor measurements, which could also be exchanged in a machine-to-machine fashion. This WoT principle drove our experiments, as later shown in Sect. 4.3.

[7] `http://xmlns.com/foaf/0.1/`
[8] `http://www.geonames.org/ontology`
[9] `http://www.w3.org/2004/02/skos/core`
[10] `http://purl.oclc.org/NET/ssnx/ssn`



(a) BTCSAMPLE (b) NODE

(c) SSP (d) DESIGO

Fig. 1: Compaction Results (whole data sets)

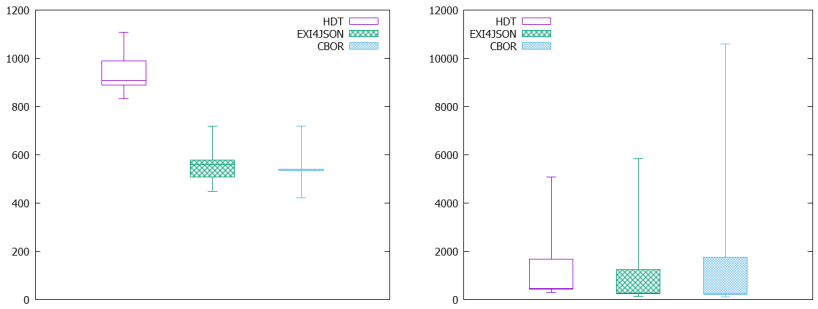
Desigo CC includes a core object-oriented data model, which has been formalized in OWL (CC)[11]. The CCBA ontology[12] is a specialization of CC for building automation, with alignments with the SAREF ontology[13].

4.3 Results

We compared the following compaction methods: HDT with bitmap encoding, HDT with k^2-triples encoding, JSON-LD with minimal context (both with EXI4JSON and CBOR encoding), JSON-LD with ontology-based context (again with EXI4JSON and CBOR encoding) and RDF/EXI. The results are shown on Figs. 1a, 1b, 1c & 1d.

What our results first show is that k^2-triples, although highly efficient on large datasets, performs poorly on small ones: on NODE, the size for k^2-triples is 16kB while all others remain under 3kB. This observation consolidates our previous results on the μRDF store. Moreover, on all data sets, RDF/XML is outperformed by other approaches. In particular, it is outperformed by HDT, contrary to what earlier results suggested [15]. The difference in performances most likely rests on the fact that EXI performs good on data sets with many

[11] currently in the process of being publicly released
[12] idem
[13] `https://w3id.org/saref`



(a) SSP (b) DESIGO

Fig. 2: Compaction Result Distributions (piecewise data sets)

non-string literals (e.g. numeric sensor values), which happens not to be the case in these data sets. It is interesting to note, however, that EXI performs better than CBOR on compressing structural patterns (as in SSP).

HDT was originally designed to compress very large datasets but it also performs good on medium size data sets, like SSP and DESIGO. It is more than twice as performant as EXI4JSON and by far better than CBOR (which is consistently outperformed by EXI4JSON). However, on the small NODE dataset, EXI4JSON and HDT have comparable results, regardless of the context used. The results on BTCSAMPLE illustrate the impact of choosing a context on the overall compaction performances: the minimal context allows for 37% compaction compared to the ontology-based context (5409 kB / 8639 kB). Indeed, the higher the variety of IRIs used in an application, the more arduous it is to design a context that fits.

JSON-LD compaction shows best results on datasets of less than hundred triples, the typical amount of data carried by constrained WoT agents. Figures 2a & 2b shows how EXI4JSON and CBOR are more efficient than HDT on SSP and DESIGO, when pieces of data are serialized separately (using the same ontology-based context). For median results on DESIGO, EXI4JSON and CBOR achieve a compaction ratio of 58% and 50% compared to HDT (respectively). Compaction ratios on SSP are similar: 62% and 59%. Interestingly, CBOR performs better on median results but it shows a higher variance than EXI4JSON. One can also note that the DESIGO data set appears skewed towards small TDs. The reason is that many TDs in the data set are logical entities, without any Web resource attached (and thus relatively small TDs). It is the case for e.g. buildings, floors, rooms, etc.

The overhead of HDT on small data sets is due to the dictionary it embeds in all individual documents (mostly redundant). One could note that the principle of defining a global context for a set of documents could also apply to HDT dictionaries. However, splitting a dictionary into global and local parts would lower the compaction ratio and require decompression on the MCU before processing (e.g. a SPARQL or JSON-LD framing query).

5 Conclusion

Throughout this paper, we considered the use of JSON-LD as a serialization format for embedded devices (MCUs). Our formalization of JSON-LD compaction and framing highlights that these two procedures directly address the verbosity and complexity of RDF. JSON-LD objects can be processed in their compact form, given a global context shared across agents in a network; JSON-LD framing extends SPARQL basic graph patterns without increasing the theoretical complexity of query processing.

This work follows on from of our development of the μRDF store, an RDF store for MCUs. We regard JSON-LD framing as an important building block for autonomous systems, a a driver of machine-to-machine communication.

Experimentally, we found that JSON-LD compaction, coupled with EXI4-JSON or CBOR, can lead to compaction ratios around 50-60% compared to the state-of-the-art (HDT) for small WoT data sets (around hundred triples). JSON-LD has already been adopted by many providers in the Web of data but the results we presented in this paper suggests that it might as well become a reference in the Web of Things.

References

1. Álvarez-García, S., Brisaboa, N.R., Fernández, J.D., Martínez-Prieto, M.A.: Compressed k2-Triples for Full-In-Memory RDF Engines. CoRR abs/1105.4004 (2011)
2. Atzori, L., Iera, A., Morabito, G.: The Internet of Things: A survey. Computer Networks 54(15), 2787–2805 (Oct 2010)
3. Boldt, D., Hasemann, H., Karnstedt, M., Kroeller, A., Weth, C.v.d.: SPARQL for Networks of Embedded Systems. pp. 93–100. IEEE (Dec 2015)
4. Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained-Node Networks (May 2014), `https://tools.ietf.org/html/rfc7228`
5. Charpenay, V., Käbisch, S., Kosch, H.: μRDF Store: Towards Extending the Semantic Web to Embedded Devices. In: The Semantic Web: ESWC 2017 Satellite Events, vol. 10577, pp. 76–80. Springer International Publishing, Cham (2017), dOI: 10.1007/978-3-319-70407-4_15
6. Fernández, J.D., Llaves, A., Corcho, O.: Efficient RDF Interchange (ERI) Format for RDF Data Streams. In: The Semantic Web ISWC 2014, vol. 8797, pp. 244–259. Springer International Publishing, Cham (2014), dOI: 10.1007/978-3-319-11915-1_16
7. Fernández, J.D., Martínez-Prieto, M.A., Gutierrez, C.: Compact Representation of Large RDF Data Sets for Publishing and Exchange. In: The Semantic Web ISWC 2010, vol. 6496, pp. 193–208. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Guinard, D.: A Web of Things Application Architecture – Integrating the Real-World into the Web. Ph.D. thesis, ETH Zurich, Zurich, Switzerland (Aug 2011)
9. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of semantic web databases. p. 95. ACM Press (2004)
10. Hartig, O.: An Overview on Execution Strategies for Linked Data Queries. Datenbank-Spektrum 13(2), 89–99 (Jul 2013)
11. Hasemann, H., Kroller, A., Pagel, M.: RDF provisioning for the Internet of Things. pp. 143–150 (2012)

12. Hasemann, H., Kröller, A., Pagel, M.: The Wiselib TupleStore: A Modular RDF Database for the Internet of Things abs/1402.7228 (Mar 2014)
13. Hayes, P.J., Patel-Schneider, P.: RDF 1.1 Semantics, http://www.w3.org/TR/rdf11-mt/
14. Käbisch, S., Kamiya, T.: Web of Things (WoT) Thing Description. W3c First Public Working Draft, W3C (Sep 2017), https://www.w3.org/TR/wot-thing-description/
15. Käbisch, S., Peintner, D., Anicic, D.: Standardized and Efficient RDF Encoding for Constrained Embedded Networks. In: The Semantic Web. Latest Advances and New Domains, vol. 9088, pp. 437–452. Springer International Publishing, Cham (2015), dOI: 10.1007/978-3-319-18818-8_27
16. Kazuo, K., Kovatsch, M., Davuluru, U.: Web of Things (WoT) Architecture. W3c First Public Working Draft, W3C (Sep 2017), https://www.w3.org/TR/wot-architecture/
17. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM 42(4), 741–843 (Jul 1995)
18. Longley, D., Kellogg, G., Lanthaler, M., Sporny, M.: JSON-LD 1.0 Processing Algorithms and API. W3c Recommendation, W3C (Jan 2014), https://www.w3.org/TR/json-ld-api/
19. Loseto, G., Ieva, S., Gramegna, F., Ruta, M., Scioscia, F., Di Sciascio, E.: Linked Data (in low-resource) Platforms: a mapping for Constrained Application Protocol. In: Proceedings of 15th International Semantic Web Conference (ISWC 2016). Kobe (Oct 2016)
20. Peintner, D., Brutzman, D.: EXI for JSON (EXI4json). W3c Working Draft, W3C (Aug 2016), https://www.w3.org/TR/exi-for-json/
21. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: The Semantic Web - ISWC 2006, vol. 4273, pp. 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
22. Riekki, J., Su, X., Haverinen, J.: Connecting Resource-Constrained Robots to Knowledge-Based Systems. In: Proceeding of International Conference on Modelling, Identification and Control. ACTA Press (2008)
23. Shelby, Z.: Embedded web services. IEEE Wireless Communications 17(6), 52–57 (Dec 2010)
24. Speicher, S., Arwe, J., Malhotra, A.: Linked Data Platform 1.0 (Feb 2015), https://www.w3.org/TR/ldp/
25. Sporny, M., Kellogg, G., Longley, D., Lanthaler, M.: JSON-LD Framing 1.1. W3c Draft Community Group Report (Oct 2017), https://json-ld.org/spec/latest/json-ld-framing/
26. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: JSON-LD 1.0 - A JSON-based Serialization for Linked Data. W3c Recommendation, W3C (Jan 2014), http://www.w3.org/TR/json-ld/
27. Trifa, V.M.: Building Blocks for a Participatory Web of Things: Devices, Infrastructures, and Programming Frameworks. Ph.D. thesis, ETH Zurich, Zurich, Switzerland (Aug 2011)
28. Verborgh, R., Hartig, O., Meester, B.D., Haesendonck, G., Vocht, L.D., Sande, M.V., Cyganiak, R., Colpaert, P., Mannens, E., Walle, R.V.d.: Querying Datasets on the Web with High Availability. In: The Semantic Web ISWC 2014, pp. 180–196. Springer International Publishing (Jan 2014)