The PROV-JSONLD Serialization

A JSON-LD Representation for the PROV Data Model

Luc Moreau[0000−0002−3494−120X] and Trung Dong Huynh[0000−0003−4937−2473]⋆

Department of Informatics
King's College London,
Bush House, 30 Aldwych, London, WC2B 4BG.
{luc.moreau,dong.huynh}@kcl.ac.uk

Abstract. Provenance is information about entities, activities, and people involved in producing a piece of data or a thing, which can be used to form assessments about the data or the thing's quality, reliability, or trustworthiness. PROV-DM is the conceptual data model that forms the basis for the W3C provenance (PROV) family of specifications. In this paper, we propose a new serialization for PROV in JSON called PROV-JSONLD. It provides a lightweight representation of PROV expressions in JSON, which is suitable to be processed by Web applications, while maintaining a natural encoding that is familiar with PROV practitioners. In addition, PROV-JSONLD exploits JSON-LD to define a semantic mapping that conforms to the PROV-O specification and, hence, the encoded PROV expressions can be readily processed as Linked Data. Finally, we show that the serialization is also efficiently processable in our evaluation. Overall, PROV-JSONLD is designed to be suitable for interchanging provenance information in Web and Linked Data applications, to offer a natural encoding of provenance for its targeted audience, and to allow for fast processing.

1 Introduction

Since their release in 2013, the PROV Recommendations [4] by the World Wide Web Consortium (W3C) have started being adopted by flagship deployments such as the Global Change Information System,[1] the Gazette[2] in the UK, and other Linked Datasets. PROV, which is used as the data model to describe the provenance of data, is made available in several different representations: PROV-N [13], PROV-XML [6], or in an RDF serialization using the PROV Ontology (PROV-O) [10]. The latter, arguably, is most suitable for Linked Data [5], given

⋆ This work was partially supported by the UK Engineering and Physical Sciences Research Council (EPRSC Grant EP/S027238/1) and the US Department of Navy award (N62909-18-1-2079) issued by the Office of Naval Research Global. The United States Government has a royalty-free license throughout the world in all copyrightable material contained herein.

[1] https://data.globalchange.gov.

[2] https://www.thegazette.co.uk.

that it can readily be consumed by existing Semantic Web tools and comes with the semantic grounding provided by PROV-O. Surprisingly, the PROV-JSON [7] serialization has gained traction, despite simply being a member submission to the W3C, and not having gone through the various stages of a standardization activity. The primary reason for this, we conjecture, is that many Web applications are built to be lightweight, working mainly with simple data formats such as JSON [2].

The very existence of all these serializations is a testament to the approach to standardization taken by the Provenance Working Group, by which a conceptual data model for PROV was defined, the PROV data model (PROV-DM) [12], alongside its mapping to different technologies, to suit users and developers. However, the family of PROV specifications lacks a serialization that is capable of addressing all of the following requirements.

R1 A serialization must support **lightweight** Web applications.
R2 A serialization must look **natural** to its targeted community of users.
R3 A serialization must allow for **semantic** markup and integration with linked data applications.
R4 A serialization must be processable in an **efficient** manner.

Surprisingly, none of the existing PROV serializations supports all these requirements simultaneously. While PROV-JSON is the only serialization to support lightweight Web applications, it does not have any semantic markup, its internal structure does not exhibit the natural structure of the PROV data structures, and its grouping of expressions per categories (e.g. all entities, all activities,. . .) is not conducive to incremental processing. The RDF serialization compatible with PROV-O has been architected to be natural to the Semantic Web community: all influence relations have been given the same directionality with respect to their time ordering, but the decomposition of data structures (essentially n-ary relations) into individual triples, which can occur anywhere in the serialization, is not conducive to efficient parsing. It is reasonable to say that the world has moved on from XML, while the PROV-N notation was aimed at humans rather than efficient processing.

Against that background, JSON-LD [15] allows a semantic structure to be overlaid over a JSON structure, thereby enabling the interpretation of JSON serializations as Linked Data. This was exploited in an early version of this work [8], which applied the JSON-LD approach to a JSON serialization of PROV. The solution, however, did not lead to a natural encoding of the PROV data structure, because a property occurring in different types of JSON objects had to be named differently so that it could be uniquely mapped to the appropriate RDF property; we observe here that what is natural in JSON is not necessarily natural in RDF, and vice-versa. The ability to define scoped contextual mappings was introduced in JSON-LD 1.1 [9] and is a key enabler of this work, allowing for the same natural PROV property names to be used in different contexts while still maintaining their correct mappings to the appropriate RDF properties.

Thus, this paper proposes PROV-JSONLD, a serialization of PROV that is compatible with PROV-DM and that addresses all of our four key requirements

above. It is first and foremost a JSON structure so it supports lightweight Web applications. It is structured in such a way that each PROV expression is encoded as a self-contained JSON object and, therefore, is natural to JavaScript programmers. Exploiting JSON-LD 1.1, we defined contextual semantic mappings, allowing a PROV-JSONLD document to be readily consumed as Linked Data that is conforming to the PROV Ontology. Finally, PROV-JSONLD allows for efficient processing since each JSON object can be readily mapped to a data structure, without requiring unbounded lookaheads, or search within the data structure.

In the remainder of this paper, we provide an illustration of PROV-JSONLD in Section 2; we then define its structure by means of a JSON Schema [1] in Section 3 and its JSON-LD semantic mapping in Section 4. We outline the interoperability testing we put in place to check its compatibility with the PROV data model in Section 5 and evaluate the efficiency of PROV-JSONLD processing in Section 6. Section 7 concludes the paper with an outline for future work.

2 Example

To illustrate the proposed PROV-JSONLD serialization, we consider a subset of the example of PROV-PRIMER [3], depicted in Fig. 1. It can be paraphrased as follows: agent Derek was responsible for composing an article based on a dataset.

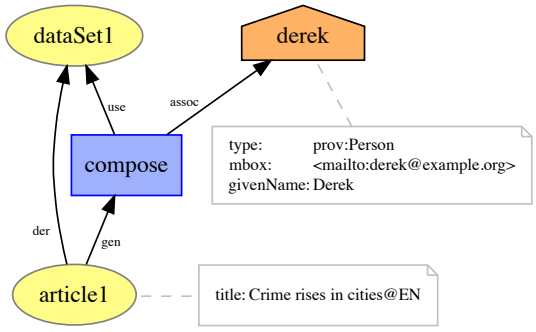

Fig. 1. Provenance expressing that Derek was responsible for composing an article based on a dataset.

The PROV-JSONLD representation of this example can be seen in Listing 1. At the top level, a PROV-JSONLD document is a JSON object with two properties `@context` and `@graph`, as per JSON-LD. A context contains mappings of prefixes to namespaces, and also an explicit reference to https://openprovenance.org/prov-jsonld/context.json, the JSON-LD 1.1 context defining the semantic mapping for PROV-JSONLD, which is described in Section 4. The `@graph`

```
{
  "@context": [ {
    "dcterms": "http://purl.org/dc/terms/",
    "ex"     : "http://example/",
    "foaf"   : "http://xmlns.com/foaf/0.1/"
  }, "https://openprovenance.org/prov-jsonld/context.json" ],
  "@graph" : [ {
    "@type" : "prov:Entity",
    "@id"   : "ex:dataSet1"
  }, {
    "@type" : "prov:Entity",
    "@id"   : "ex:article1",
    "dcterms:title": [
      { "@value" : "Crime rises in cities", "@language" : "EN" }
    ]
  }, {
    "@type"           : "prov:Derivation",
    "generatedEntity" : "ex:article1",
    "usedEntity"      : "ex:dataSet1"
  }, {
    "@type"           : "prov:Agent",
    "@id"             : "ex:derek",
    "foaf:mbox"       : [ { "@value" : "<mailto:derek@example.org>" } ],
    "prov:type"       : [ "prov:Person" ],
    "foaf:givenName"  : [ { "@value" : "Derek" } ]
  }, {
    "@type"    : "prov:Association",
    "activity" : "ex:compose",
    "agent"    : "ex:derek"
  }, {
    "@type"    : "prov:Activity",
    "@id"      : "ex:compose"
  }, {
    "@type"    : "prov:Usage",
    "activity" : "ex:compose",
    "entity"   : "ex:dataSet1"
  }, {
    "@type"    : "prov:Generation",
    "entity"   : "ex:article1",
    "activity" : "ex:compose"
  } ]
}
```

Listing 1: The PROV-JSONLD representation of Fig. 1.

property has an array of PROV expressions as value. Each PROV expression itself is a JSON object with at least a `@type` property (for instance, prov:Entity, prov:Agent or prov:Derivation). Each of these PROV expressions provides a description for a resource, some of which are identified by the `@id` property (for instance, `ex:article1` or `ex:derek`). Some of the resources are anonymous and, therefore, do not have a property `@id`, for instance, the prov:Derivation relation between the dataset and the article (Line 17–19).

PROV expressions can be enriched with a variety of properties. Some of which are "reserved" such as `activity` and `agent` in a prov:Association. Others may be defined in a different namespace such as `foaf:givenName`, for which we expect the prefix `foaf` to be declared in the `@context` property. Finally, further PROV attributes are allowed, for instance, `prov:type` with an array of further types, to better describe the resource.

The property `@type` is mandatory and is associated with a single value, expected to be one of the predefined PROV expression types. From an efficiency viewpoint, this property is critical in determining which internal data structure a PROV expression should map to and, therefore, facilitates efficient processing. On the contrary, `prov:type` is optional and can contain as many types as required; their order is not significant.

3 PROV-JSONLD Schema

In this section, we provide an overview of the JSON schema [1] for PROV-JSONLD; the full schema is available at https://openprovenance.org/prov-jsonld/schema.json.

3.1 Preliminary definitions

Some primitive types, namely `DateTime` and `QualifiedName`, occur in PROV serializations. We define their schema[3] as follows.

```
{ "DateTime": {
    "$id": "#/definitions/DateTime",
    "type": "string",
    "format": "date-time" },
  "QualifiedName": {
    "$id": "#/definitions/QualifiedName",
    "type": "string", "default": "",
    "pattern": "^[A-Za-z0-9_]+:(.*)$" },
}
```

In addition, we define typed values (`typed_value`) as JSON objects with properties `@value` and `@type` and string values `lang_string` as objects with properties `@value` and `@language`.

[3] The production rules for qualified names are more complex than the simple regular expression outlined here. A post-processor will need to check that qualified names comply with the definition in [13].

```
{ "typed_value": {
    "type": "object", "required": [ "@value", "@type" ],
    "properties": {
      "@value": { "type": "string" },
      "@type":  { "type": "string" }
    },
    "additionalProperties": false },
  "lang_string": {
    "type": "object", "required": [ "@value" ],
    "properties": {
      "@value":    { "type": "string" },
      "@language": { "type": "string" }
    },
    "additionalProperties": false }
}
```

We also define types for collections of property values, which can be arrays of values (`ArrayOfValues`) or arrays of labels (`ArrayOfLabelValues`).

```
{ "ArrayOfValues": {
    "$id": "#/definitions/ArrayOfValues", "type": "array",
    "items": {
      "anyOf": [
        { "$ref": "#/definitions/QualifiedName" },
        { "$ref": "#/definitions/typed_value" },
        { "$ref": "#/definitions/lang_string" } ]
    }
  },
  "ArrayOfLabelValues": {
    "$id": "#/definitions/ArrayOfLabelValues", "type": "array",
    "items": { "$ref": "#/definitions/lang_string" }
  }
}
```

With these preliminary definitions in place, we can now present the specification of the core data structures of PROV-JSONLD.

3.2 Encoding a PROV expression

Each PROV expression is serialized into a single JSON object in a `@graph` array. For instance, Listing 2 shows the JSON schema for a prov:Entity expression in PROV-JSONLD. All the constituents of an expression become properties of the object as follows:

- The **identifier** (if present) becomes the identifier of the object (`@id` property). For prov:Entity, prov:Activity, and prov:Agent expressions, the `@id` property is *required* while it is *optional* for all other PROV expressions.
- The **type** of the PROV expression, e.g. prov:Activity, prov:Derivation, becomes the *only* value for the object's `@type` property, which is *always required*. Additional types, if any, are added to an array held by the `prov:type` property.

```
{
  "prov:Entity": {
    "type": "object", "required": [ "@type", "@id" ],
    "properties": {
      "@type":           { "pattern": "prov:Entity" },
      "@id":             { "$ref": "#/definitions/QualifiedName" },
      "prov:type":       { "$ref": "#/definitions/ArrayOfValues"},
      "prov:location":   { "$ref": "#/definitions/ArrayOfValues" },
      "prov:label":      { "$ref": "#/definitions/ArrayOfLabelValues"}
    },
    "patternProperties": {
      "^[A-Za-z0-9_]+:(.*)$": { "$ref": "#/definitions/ArrayOfValues" }
    },
    "additionalProperties": false }
}
```

Listing 2: The JSON schema for `prov:Entity`.

- Other formal constituents of the PROV expression are encoded as properties of the object using the same property name as defined in PROV-DM. All those properties are *optional*. For example, the JSON object for a prov:Activity may have the properties `startTime` and/or `endTime`, for which `DateTime` string values are expected; while the object for a prov:Derivation expression may have the properties `activity`, `generation`, `usage`, `generatedEntity`, and `usedEntity`, for which `QualifiedName` string values are expected.
- The object may contain **additional attributes** which are encoded as the object's properties, such as a location (property `prov:location`), a label (property `prov:label`), or any other properties with an *explicit* prefix.

3.3 Encoding a PROV document and a PROV bundle

A PROV document is encoded as a JSON object which must contain a property `@type` with the value `prov:Document`, a JSON-LD context `@context`, and an array of PROV expressions as the value of the property `@graph`. The names of the properties `@context` and `@graph` are specified by JSON-LD [9].

A PROV bundle is encoded in the same way as a PROV document except that the JSON object for the bundle must contain a property `@type` with the value `prov:Bundle` and additionally an identifier (property `@id`). Listing 3 shows the JSON schema of a prov:Bundle object in PROV-JSONLD. A bundle contains a list of statements (see Listing 4, definition `prov:Statement`), which can be one of the defined PROV expressions as per Section 3.2. Documents, however, can contain statements and/or bundles (see Listing 4, definition `prov:StatementOrBundle`).

```
{ "prov:Bundle": {
    "type": "object",
    "required": [ "@type", "@id", "@graph", "@context" ],
    "properties": {
      "@type":    { "pattern": "prov:Bundle" },
      "@id":      { "$ref": "#/definitions/QualifiedName" },
      "@context": { "$ref": "#/definitions/Context" },
      "@graph":   {
        "type": "array",
        "items": { "$ref": "#/definitions/prov:Statement" } } },
    "additionalProperties": false }
}
```

Listing 3: JSON schema for `prov:Bundle`.

```
{ "prov:Statement": {
    "oneOf": [
      { "$ref": "#/definitions/prov:Entity" },
      { "$ref": "#/definitions/prov:Activity" },
      { "$ref": "..." },
      { "$ref": "#/definitions/prov:Communication" }
    ]
  },
  "prov:StatementOrBundle": {
    "oneOf": [
      { "$ref": "#/definitions/prov:Statement" },
      { "$ref": "#/definitions/prov:Bundle" }
    ] }
}
```

Listing 4: JSON schema for `prov:Statement` and `prov:StatementOrBundle`; 14 other statements are abbreviated in Line 5 to save space.

4 PROV-JSONLD Context

JSON-LD contexts define mappings between terms[4] in a JSON document and IRIs (Internationalized Resource Identifier) [9] and, thus, enable the JSON document to be parsed as Linked Data. Using this mechanism, we define the PROV-JSONLD context in order for terms in a PROV-JSONLD document to be mapped to appropriate PROV properties and classes as defined by PROV-O (when read by a JSON-LD-compliant consumer). Due to the limited space, we present only some typical examples of term mappings in this section. The full context is available at https://openprovenance.org/prov-jsonld/context.json.

[4] A term is a short-hand string that expands to an IRI, a blank node identifier, or a keyword [9].

4.1 Default context elements

The following JSON properties have a *default* meaning *unless* they are redefined in a specific context of a PROV-JSONLD document: `entity`, `activity` and `agent` respectively map to the properties `prov:entity`, `prov:activity`, and `prov:agent`. In addition, the JSON properties `prov:role`, `prov:type`, `prov:label` and `prov:location` have the same meaning in *all* contexts of a PROV-JSONLD document and respectively map to `prov:hadRole`, `rdf:type`, `rdfs:label`, and `prov:atLocation`.

```
{ "entity":        { "@type": "@id", "@id": "prov:entity" },
  "activity":      { "@type": "@id", "@id": "prov:activity" },
  "agent":         { "@type": "@id", "@id": "prov:agent" },
  "prov:role":     { "@type": "@id", "@id": "prov:hadRole" },
  "prov:type":     { "@type": "@id", "@id": "rdf:type" },
  "prov:label":    {                 "@id": "rdfs:label" },
  "prov:location": { "@type": "@id", "@id": "prov:atLocation" }
}
```

4.2 Contexts for PROV elements

JSON objects with types `prov:Entity` and `prov:Agent` can be mapped directly to the corresponding classes defined by PROV-O without any extra specific mapping apart from the default above. JSON objects with type `prov:Activity`, however, can additionally have the JSON properties `startTime` and `endTime`, which map to the RDF data properties `prov:startedAtTime` and `prov:endedAtType`, respectively, and have a range of type `xsd:dateTime`.

```
{ "prov:Activity": {
    "@id": "prov:Activity",
    "@context" : {
      "startTime":{"@type": "xsd:dateTime", "@id": "prov:startedAtTime"},
      "endTime":  {"@type": "xsd:dateTime", "@id": "prov:endedAtTime"} }
  }
}
```

4.3 Contexts for PROV relations

The ontology PROV-O [10] defines the Qualification Pattern, which restates a binary property between two resources by using an intermediate class that represents the influence between the two resources. This new instance, in turn, can be annotated with additional descriptions of the influence that one resource had upon another. Fig. 2, for example, shows the Qualification Pattern defined for class prov:Usage in PROV-O. For each PROV relation expression, we apply its Qualification Pattern and encode the relation as a resource in PROV-JSONLD. The mapping below supports the Qualification Pattern of Fig. 2. The JSON properties `activity` and `time` map to the object property `prov:qualifiedUsage` and the data property `prov:atTime`, respectively.

```
{ "prov:Usage": {
    "@id": "prov:Usage",
    "@context": {
      "activity": { "@type": "@id", "@reverse": "prov:qualifiedUsage" },
      "time":     { "@type": "xsd:dateTime", "@id": "prov:atTime" } } }
}
```

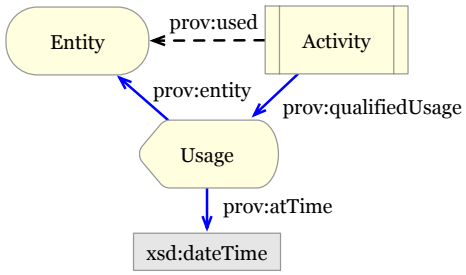

Fig. 2. The Qualification Pattern for class prov:Usage defined by PROV-O [10].

Note that the context above applies only to objects with the `@type` property having the value `prov:Usage` thanks to the support for *type-scoped contexts* by JSON-LD 1.1 [9]. For such objects, the above context for prov:Usage is in effect *in addition to* the default context in Section 4.1 and overrides the default mapping for the term `activity`, replacing it with a reverse mapping to `prov:qualifiedUsage`. The default mapping for the term `entity` (mapping it to `prov:entity`), however, still applies. Along with the additional definition for the term `time`, they complete the JSON-LD context for prov:Usage (Fig. 2).

We follow the same convention for all the remaining PROV relations. Table 1 provides the mappings to the qualification properties for the reverse terms defined in the type-scoped contexts. Note that while PROV-O does not define a Qualification Pattern for Specialization, Alternate, and Membership relations, for uniformity and usability reasons, we adopt similar mappings as other PROV relations, via a Qualification Pattern. However, the mappings are to new classes and properties in the PROV extension namespace (denoted by the prefix `provext`). See the following section for some interoperability considerations.

5 Interoperability Considerations

IC1 There are differences between PROV-DM and PROV-O in terms of the level of requirements set on some expressions. For instance, PROV-DM mandates the presence of an entity in a generation, whereas PROV-O defines an activity as optional. Compliance requirements are not the same in PROV-O as

Table 1. PROV-JSONLD reverse terms defined in type-scoped contexts and their corresponding qualification properties.

PROV-O classes	PROV-JSONLD terms	Qualification properties
prov:Generation	`entity`	`prov:qualifiedGeneration`
prov:Derivation	`generatedEntity`	`prov:qualifiedDerivation`
prov:Invalidation	`entity`	`prov:qualifiedInvalidation`
prov:Attribution	`entity`	`prov:qualifiedAttribution`
prov:Usage	`activity`	`prov:qualifiedUsage`
prov:Start	`activity`	`prov:qualifiedStart`
prov:End	`activity`	`prov:qualifiedEnd`
prov:Association	`activity`	`prov:qualifiedAssociation`
prov:Communication	`informed`	`prov:qualifiedCommunication`
prov:Delegation	`delegate`	`prov:qualifiedDelegation`
prov:Influence	`influencee`	`prov:qualifiedInfluence`
provext:Specialization	`specificEntity`	`provext:qualifiedSpecialization`
provext:Alternate	`alternate1`	`provext:qualifiedAlternate`
provext:Membership	`collection`	`provext:qualifiedMembership`

one could define a qualified generation with an activity but without an entity. Experience shows that there may be good reasons why a generation may not refer to an entity; for instance, because the recorded provenance is not "complete" yet, and further provenance expressions still need to be asserted, received, or merged; in the meantime, we still want to be able to process such provenance, despite being "incomplete". Thus, in PROV-JSONLD, the presence of an entity and an activity in a generation expression is *recommended*, while other properties are optional, and only its `@type` property is *required*.

IC2 In PROV-DM, all relations are n-ary except for specialization, alternate and membership, which are binary, meaning that no identifier or extra properties are allowed for these. In PROV-O, this design decision translates to the lack of qualified relations for specialization, alternate and membership. In PROV-JSONLD, in order to keep the regular structure of JSON objects and the natural encoding of relations, but also to ensure the simplicity and efficiency of parsers, these three relations are encoded using the same pattern as for other relations. Therefore, their mapping to RDF via the JSON-LD context relies on a PROV extension namespace (denoted by the prefix `provext`) in which classes for Specialization, Alternate, and Membership are defined. The PROV-JSONLD serialization also allows for identifier and properties to be encoded for these relations.

IC3 The notion of a PROV document is not present in PROV-DM or PROV-O, but is introduced in PROV-N as a housekeeping construct, and is defined in PROV-XML as the root of a PROV-XML document. A document in PROV-JSONLD is also a JSON object, allowing for a JSON-LD `@context` property to be specified.

IC4 PROV-JSONLD does not introduce constructs for some PROV subtypes (prov:Person, prov:Organization, prov:SoftwareAgent, and prov:Collection) and subrelations (prov:Quotation, prov:PrimarySource, and prov:Revision). Instead, the example of Section 2 illustrates how they can be accommodated within the existing structures. We copy below an agent expression of type `prov:Person` (Line 5) and a derivation of type `prov:Revision` (Line 11). These subtypes and subrelations are specified inside the `prov:type` property. PROV-XML offers a similar way of encoding such subtypes and subrelations, alongside specialized structures. We opted for this single approach to ensure the simplicity and efficiency of parsers.

```
1  { "@graph" : [
2      { "@type" : "prov:Agent",
3        "@id" : "ex:derek",
4        "foaf:mbox" : [ { "@value" : "<mailto:derek@example.org>" } ],
5        "prov:type" : [ "prov:Person" ],
6        "foaf:givenName" : [ { "@value" : "Derek" } ]
7      },
8      { "@type" : "prov:Derivation",
9        "generatedEntity" : "ex:dataSet2",
10       "usedEntity" : "ex:dataSet1",
11       "prov:type" : [ "prov:Revision" ]
12     } ]
13 }
```

IC5 The interoperability of the PROV-JSONLD serialization can be tested in different ways: In a roundtrip testing, consisting of the serialization of an internal representation in some programming language to PROV-JSONLD, followed by deserialization from PROV-JSONLD back to the same programming language, the source and target representations are expected to be equal. Likewise, in a roundtrip testing, consisting of the serialization of an internal representation in some programming language to PROV-JSONLD, followed by a conversion of PROV-JSONLD to another RDF representation such as Turtle [14], followed by a reading of the Turtle representation back to the same programming language, the source and target representations are also expected to be equal. Both interoperability tests have been implemented in the Java-based ProvToolbox, with:

- The first roundtrip testing is implemented in https://github.com/lucmoreau/ProvToolbox/blob/master/modules-core/prov-jsonld/src/test/java/org/openprovenance/prov/core/RoundTripFromJavaJSONLD11Test.java
- The second roundtrip testing is implemented in https://github.com/lucmoreau/ProvToolbox/blob/master/modules-legacy/roundtrip/src/test/java/org/openprovenance/prov/core/roundtrip/RoundTripFromJavaJSONLD11LegacyTest.java

6 Implementation and Evaluation

We have introduced PROV-JSONLD as a PROV serialization that is lightweight, natural, semantic and efficient. So far, this paper has focused on the first three characteristics. The purpose of this section is to discuss its performance.

The PROV-JSONLD serialization is implemented in ProvToolbox,[5] a JVM-based library for processing PROV standardized representations. The library can build representations of the PROV data model in Java and can convert such data structures to PROV-JSONLD, PROV-N, and PROV-JSON, and vice-versa.

From an implementation viewpoint, each PROV expression has an associated class implementing it, which we refer to as JAVA-PROV. For instance, the JAVA-PROV class for a PROV Generation has fields for an activity and an entity, and further optional fields.

```
public class WasGeneratedBy implements
 org.openprovenance.prov.model.WasGeneratedBy, HasAttributes {

 QualifiedName activity;
 QualifiedName entity;
 Optional<QualifiedName> id=Optional.empty();
 Optional<XMLGregorianCalendar> time=Optional.empty();
 List<org.openprovenance.prov.model.LangString> labels=new LinkedList<>();
 List<org.openprovenance.prov.model.Location> location=new LinkedList<>();
 List<org.openprovenance.prov.model.Other> other = new LinkedList<>();
 List<org.openprovenance.prov.model.Type> type = new LinkedList<>();
 List<org.openprovenance.prov.model.Role> role = new LinkedList<>();

 // constructors, accessors and mutators ...
}
```

Serialization/Deserialization to/from PROV-JSONLD is performed by the library Jackson[6], which can be configured by means of mix-ins: in the interface below, the sister interface `JLD_WasGeneratedBy` allows for configuration annotation for Jackson to be mixed-in with the definition of the JAVA-PROV definition of `WasGeneratedBy`, while keeping the original JAVA-PROV code intact. For instance, for each field, the mix-in has the opportunity to control specialized methods for serialization or deserialization. It also specifies an order in which the JSON properties are expected to be serialized.

```
@JsonPropertyOrder({ "@id", "entity", "activity", "atTime" })
@JsonInclude(JsonInclude.Include.NON_NULL)
public interface JLD_WasGeneratedBy extends JLD_Generic, HasRole {

    @JsonDeserialize(using = CustomQualifiedNameDeserializer.class)
    public QualifiedName getEntity();
```

[5] https://lucmoreau.github.io/ProvToolbox/.
[6] https://github.com/FasterXML/jackson

```
    @JsonDeserialize(using = CustomQualifiedNameDeserializer.class)
    public QualifiedName getActivity();

    XMLGregorianCalendar getTime();
}
```

A PROV document consists of a sequence of statements or bundles. Given that a JAVA-PROV statement is declared by an interface, the Jackson deserializer is able to determine the necessary constructor to invoke by relying on the `@type` property, automatically inserted at serialization time, by the declaration below. It makes explicit, e.g., how prov:Usage class should be parsed when encountering `@type` value `"prov:Usage"`.

```
@JsonPropertyOrder({ "@context", "@graph"})
public interface JLD_Document {
  @JsonProperty("@context")
  Namespace getNamespace();

  @JsonTypeInfo(use=JsonTypeInfo.Id.NAME,
                include=JsonTypeInfo.As.PROPERTY,
                property="@type")
  @JsonSubTypes({
    @JsonSubTypes.Type(value = WasGeneratedBy.class,
                                              name = "prov:Generation"),
    @JsonSubTypes.Type(value = Used.class,    name = "prov:Usage"),
    @JsonSubTypes.Type(value = Activity.class, name = "prov:Activity"),
    @JsonSubTypes.Type(value = Agent.class,   name = "prov:Agent"),
    @JsonSubTypes.Type(value = Entity.class,  name = "prov:Entity"),
    // all other statements and bundles ...
  })
  @JsonProperty("@graph")
  List<StatementOrBundle> getStatementOrBundle();
}
```

Our investigation of the performance of PROV-JSONLD focuses on deserialization, which consists of reading a PROV-JSONLD serialization and constructing the corresponding JAVA-PROV representation, using the kind of classes described above. To compare the performance, we use an alternative set of mix-in configurations also to allow (de)serialization for PROV-JSON. We also compare with the PROV-N (de)serialization: the serialization is written by handcrafted code, whereas the deserialization is implemented using the ANTLR grammar. We are not interested in measuring the cost of inputs and outputs; therefore, our benchmarks generate serializations in a memory buffer, and vice-versa, deserialization operates on an input memory buffer.

Two further points of comparison were used for the evaluation. First, a native JSON parser, without any customization code, generates native Java objects (consisting of Java Maps for JSON objects and Java arrays for JSON sequences) from a PROV-JSONLD serialization. Such internal representations based on

Java native objects are not as conducive to processing as the JAVA-PROV classes above, since types are not made explicit, and therefore cannot exploit the Java object-oriented style with its inheritance and static typing. Second, a copy procedure performs a deep copy of a JAVA-PROV data structure, which involves the creation of the JAVA-PROV data structure and the necessary checks and initializations, to make it ready for subsequent processing.

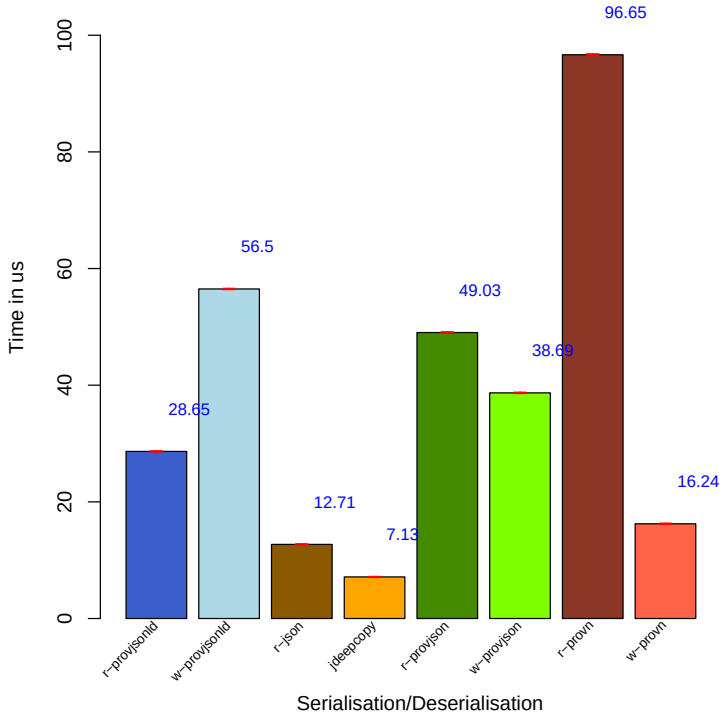

Fig. 3. (De)Serialization benchmark: `r-provjsonld`: reading time from PROV-JSONLD to JAVA-PROV; `w-provjsonld`: writing time from JAVA-PROV to PROV-JSONLD; `r-json`: reading time from PROV-JSONLD to native Java objects; `jdeepcopy`: time for deep copy of JAVA-PROV; `r-provjson`: reading time from PROV-JSON to JAVA-PROV; `w-provjson`: writing time from JAVA-PROV to PROV-JSON; `r-provn`: reading time from PROV-N to JAVA-PROV; `w-provn`: writing time from JAVA-PROV to PROV-N.

The results of the benchmarking operations are displayed in Fig. 3. We can see that the efficiency of `r-provjsonld` ($28.65\mu s$) which is barely 45% over the `r-json` + `jdeepcopy` (i.e. $19.84\mu s$). This result is very good as there are additional operations included in `r-provjsonld` but not in `r-json` or `jdeepcopy`, such as indexing and preparing prefix and namespace mappings.

The PROV-JSONLD reading time (`r-provjsonld`) significantly outperforms PROV-N reading time (`r-provn`), which is penalized by currently relying on an intermediary abstract syntax tree before constructing JAVA-PROV.

We were surprised how the handcrafted PROV-N writer `w-provn` outperforms any of JSON serializations `w-provjsonld` and `w-provjson`. We have not been able to ascertain the origin of this difference. We conjecture that the handcrafted technique of `w-provn` could be applied to PROV-JSONLD and give a similar performance.

Overall, the plot in Fig. 3 demonstrates a serialization/deserialization technique for PROV-JSONLD that is efficient.

7 Conclusion

In this paper, we have defined the PROV-JSONLD serialization, a JSON and Linked Data representation for the PROV data model. It provides a lightweight representation of PROV expressions in JSON, which is suitable to be processed by Web applications, while maintaining a natural encoding that is familiar with PROV practitioners. Using JSON-LD 1.1, we define a semantic mapping for PROV-JSONLD such that the encoded PROV expressions can be readily processed as Linked Data that conforms to the PROV-O specification. Finally, we show that the serialization is processable in an efficient manner with our implementation in the open-source ProvToolbox library. With the combined advantages of both JSON and Linked Data representations, we envisage that PROV-JSONLD will gradually replace PROV-JSON and other PROV-compliant RDF serializations. To that end, the PROV-JSONLD serialization reported in this paper is being documented in a formal technical specification to be submitted to World Wide Web Consortium [11].

As a JSON serialization, PROV-JSONLD can be readily exploited by existing JSON stores to provide storage for PROV documents. ProvToolbox, for instance, is exploiting this opportunity by using MongoDB,[7] a document-oriented database for JSON documents, to persist PROV documents. While JSON stores offer generic query capabilities for JSON documents, in the future, it would be useful to define a query language that exploits the structure and semantics of PROV-JSONLD. Likewise, learning from MongoDB and its BSON binary encoding of JSON, an efficient, compact binary encoding of PROV-JSONLD could be specified.

As illustrated by the different compliance requirements discussed in **IC1** (Section 5), the various PROV serializations do not interoperate fully. At the W3C, the Provenance Working group attempted to maintain an informal mapping between PROV-DM and its encoding in RDF (as per PROV-O). This was a manual task without any tool support and, therefore, error-prone and hard to maintain. The JSON-LD context used by PROV-JSONLD appears to be a promising mechanism to systematically encode these mappings and to help iron out outstanding interoperability issues.

[7] https://www.mongodb.com.

References

1. Andrews, H., Wright, A., Hutton, B.: JSON schema: A media type for describing JSON documents. Internet draft, Internet Engineering Task Force (IETF) (2019), https://tools.ietf.org/html/draft-handrews-json-schema-02
2. Bray, T.: The JavaScript object notation (JSON) data interchange format. Request for Comments 8259, Internet Engineering Task Force (IETF) (2017), https://tools.ietf.org/html/rfc8259
3. Gil, Y., Miles, S.: PROV model primer. W3C Working Group Note, World Wide Web Consortium (Apr 2013), http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/
4. Groth, P., Moreau, L.: PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note, World Wide Web Consortium (Apr 2013), http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/
5. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space, Synthesis Lectures on the Semantic Web: Theory and Technology, vol. 1. Morgan & Claypool (2011). https://doi.org/10.2200/S00334ED1V01Y201102WBE001
6. Hua, H., Tilmes, C., Zednik, S.: PROV-XML: The PROV XML Schema. W3C Working Group Note, World Wide Web Consortium (Apr 2013), http://www.w3.org/TR/2013/NOTE-prov-xml-20130430/
7. Huynh, T.D., Jewell, M.O., Keshavarz, A.S., Michaelides, D.T., Yang, H., Moreau, L.: The PROV-JSON serialization. W3C Member Submission, World Wide Web Consortium (April 2013), https://www.w3.org/Submission/2013/SUBM-prov-json-20130424/
8. Huynh, T.D., Michaelides, D.T., Moreau, L.: PROV-JSONLD—: A JSON and Linked Data Representation for Provenance. In: Mattoso, M., Glavic, B. (eds.) Provenance and Annotation of Data and Processes. IPAW 2016, Lecture Notes in Computer Science, vol. 9672, pp. 173–177. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40593-3_15, http://link.springer.com/10.1007/978-3-319-40593-3_15
9. Kellogg, G., Champin, P.A., Longley, D.: JSON-LD 1.1: A JSON-based serialization for linked data. W3C Recommendation, World Wide Web Consortium (2020), https://www.w3.org/TR/2020/REC-json-ld11-20200716/
10. Lebo, T., Sahoo, S., McGuinness, D.: PROV-O: The PROV Ontology. W3C Recommendation, World Wide Web Consortium (2013), http://www.w3.org/TR/2013/REC-prov-o-20130430/
11. Moreau, L., Huynh, T.D.: The PROV-JSONLD serialization: A JSON-LD representation for the PROV data model. Editor's draft, King's College London (2020), https://openprovenance.org/prov-jsonld/
12. Moreau, L., Missier, P.: PROV-DM: The PROV Data Model. W3C Recommendation, World Wide Web Consortium (2013), http://www.w3.org/TR/2013/REC-prov-dm-20130430/
13. Moreau, L., Missier, P.: PROV-N: The Provenance Notation. W3C Recommendation, World Wide Web Consortium (2013)
14. Prud'hommeaux, E., Carothers, G.: RDF 1.1 Turtle. W3C Recommendation, World Wide Web Consortium (2014), http://www.w3.org/TR/2014/REC-turtle-20140225/
15. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: A JSON-based serialization for linked data. W3C Recommendation, World Wide Web Consortium (2014), https://www.w3.org/TR/2014/REC-json-ld-20140116/