RDF Dataset Canonicalization

A Standard RDF Dataset Canonicalization Algorithm

W3C Recommendation 21 May 2024

▼ **More details about this document**

This version:
https://www.w3.org/TR/2024/REC-rdf-canon-20240521/

Latest published version:
https://www.w3.org/TR/rdf-canon/

Latest editor's draft:
https://w3c.github.io/rdf-canon/spec/

History:
https://www.w3.org/standards/history/rdf-canon/
Commit history

Test suite:
https://w3c.github.io/rdf-canon/tests/

Implementation report:
https://w3c.github.io/rdf-canon/reports/

Editors:
Dave Longley (Digital Bazaar)
Gregg Kellogg
Dan Yamamoto

Former editor:
Manu Sporny (Digital Bazaar) (CG Report)

Author:
Dave Longley (Digital Bazaar)

Feedback:
GitHub w3c/rdf-canon (pull requests, new issue, open issues)
public-rch-wg@w3.org with subject line `[rdf-canon]` … *`message topic`* … (archives)

Errata:
Errata exists.

See also **translations**.

Abstract

RDF [RDF11-CONCEPTS] describes a graph-based data model for making claims about the world and provides the foundation for reasoning upon that graph of information. At times, it becomes necessary to compare the differences between sets of graphs, digitally sign them, or generate short identifiers for graphs via hashing algorithms. This document outlines an algorithm for normalizing RDF datasets such that these operations can be performed.

Status of This Document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document describes the RDFC-1.0 algorithm for canonicalizing RDF datasets, which was the input from the W3C Credentials Community Group published as [CCG-RDC-FINAL].

At the time of publication, [RDF11-CONCEPTS] is the most recent recommendation defining RDF datasets and [N-QUADS], however work on an updated specification is ongoing within the W3C RDF-star Working Group. Some dependencies from relevant updated specifications are provided normatively in this specification with the expectation that a future update to this specification will replace those with normative references to updated RDF specifications.

This document was published by the RDF Dataset Canonicalization and Hash Working Group as a Recommendation using the Recommendation track.

W3C recommends the wide deployment of this specification as a standard for the Web.

A W3C Recommendation is a specification that, after extensive consensus-building, is endorsed by W3C and its Members, and has commitments from Working Group members to royalty-free licensing for implementations. Future updates to this Recommendation may incorporate new features.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 03 November 2023 W3C Process Document.

Table of Contents

§ 1. Introduction

This section is non-normative.

When data scientists discuss canonicalization, they do so in the context of achieving a particular set of goals. Since the same information may sometimes be expressed in a variety of different ways, it often becomes necessary to transform each of these different ways into a single, standard representation. With a standard representation, the differences between two different sets of data can be easily determined, a cryptographically-strong hash identifier can be generated for a particular set of data, and a particular set of data may be digitally-signed for later verification.

In particular, this specification is about normalizing RDF datasets, which are collections of graphs. Since a directed graph can express the same information in more than one way, it requires canonicalization to achieve the aforementioned goals and any others that may arise via serendipity.

Most RDF datasets can be canonicalized fairly quickly, in terms of algorithmic time complexity. However, those that contain nodes that do not have globally unique identifiers pose a greater challenge. Normalizing these datasets presents the ***graph isomorphism*** problem, a problem that is believed to be difficult to solve quickly in the worst case. Fortunately, existing real world data is rarely, if ever, modeled in a way that manifests as the worst case and new data can be modeled to avoid it. In fact, software systems that detect a problematic dataset (see 7.1 Dataset Poisoning) can choose to assume it's an attempted denial of service attack, rather than a real input, and abort.

This document outlines an algorithm for generating a canonical serialization of an RDF dataset given an RDF dataset as input. The algorithm is called the **RDF Canonicalization algorithm version 1.0** or RDFC-1.0.

NOTE

RDF 1.1 Concepts and Abstract Syntax [RDF11-CONCEPTS] lacks clarity on the representation of language-tagged strings, where language tags of the form `xx-YY` are treated as being case insensitive. Implementations might represent language tags using all lower case in the form `xx-yy`, retain the original representation `xx-YY`, or use [BCP47] formatting conventions, leading to different canonical forms, and therefore, different hashed values.

- The Canonicalization algorithm is based on the RDF 1.1 definition, in the sense that the language tag `xx-YY` is case insensitive, which might lead to different canonicalizations if the user is not aware of this problem.
- User communities ought to agree to use lower case language tags, while being aware that some implementations might normalize language tags, affecting hash values.
- Future evolution of RDF might regulate this issue, which RDF environments might have to adapt to, and this might lead to an update of RDFC-1.0.

NOTE

See B. URDNA2015 for a comparison with the version of the algorithm published in *RDF Dataset Canonicalization* [CCG-RDC-FINAL].

§ 1.1 Uses of Dataset Canonicalization

There are different use cases where graph or dataset canonicalization are important:

- Determining if one serialization is isomorphic to another.
- Digital signing of graphs (datasets) independent of serialization or format.
- Comparing two graphs (datasets) to find differences.
- Communicating change sets when remotely updating an RDF source.

A canonicalization algorithm is necessary, but not necessarily sufficient, to handle many of these use cases. The use of blank nodes in RDF graphs and datasets has a long history and creates inevitable complexities. Blank nodes are used for different purposes:

- when a well known identifier for a node is not known, or the author of a document chooses not to unambiguously name that node,
- when a node is used to stitch together parts of a graph and the nodes themselves are not interesting (e.g., RDF Collections in [RDF11-MT]),
- when someone is trying to create an intentionally difficult graph topology.

Furthermore, RDF semantics dictate that deserializing an RDF document results in the creation of unique blank nodes, unless it can be determined that on each occasion, the blank node identifies the same resource. This is due to the fact that blank node identifiers are an aspect of a concrete RDF syntax and are not intended to be persistent or portable. Within the abstract RDF model, blank nodes do not have identifiers (although some RDF store implementations may use stable identifiers and may choose to make them portable). See Blank Nodes in [RDF11-CONCEPTS] for more information.

RDF does have a provision for allowing blank nodes to be published in an externally identifiable way through the use of Skolem IRIs, which allow a given RDF store to replace the use of blank nodes in a concrete syntax with IRIs, which then serve to repeatably identify that blank node within that particular RDF store; however, this is not generally useful for talking about the same graph in different RDF stores, or other concrete representations. In any case, a stable blank node identifier defined for one RDF store or serialization is arbitrary, and typically not relatable to the context within which it is used.

This specification defines an algorithm for creating stable blank node identifiers repeatably for different serializations possibly using individualized blank node identifiers of the same RDF graph (dataset) by grounding each blank node through the nodes to which it is connected. As a result, a graph signature can be obtained by hashing a canonical serialization of the resulting canonicalized dataset, allowing for the isomorphism and digital signing use cases. This specification does not define such a graph signature.

As blank node identifiers can be stable even with other changes to a graph (dataset), in some cases it is possible to compute the difference between two graphs (datasets), for example if changes are made only to ground triples, or if new blank nodes are introduced which do not create an automorphic confusion with other existing blank nodes. If any information which would change the generated blank node identifier, a resulting diff might indicate a greater set of changes than actually exists. Additionally, if the starting dataset is an N-Quads document, it may be possible to correlate the original blank node identifiers used within that N-Quads document with those issued in the canonicalized dataset.

NOTE

Although alternative hash algorithms might be used with this specification, applications ought to carefully weigh the advantages and disadvantages of using an alternative hash function. This is the case, in particular, for any representation of the canonical n-quads form or issued identifiers map that does not identify the associated hash algorithm. Any use case that requires reproduction of the same output is expected to unequivocally express or communicate the internal hash algorithm that was used when generating the canonical n-quads form.

§ 1.2 How to Read this Document

This document is a detailed specification for an RDF dataset canonicalization algorithm. The document is primarily intended for the following audiences:

- Software developers that want to implement an RDF dataset canonicalization algorithm.
- Masochists.

To understand the basics in this specification you must be familiar with basic RDF concepts [RDF11-CONCEPTS]. A working knowledge of graph theory and graph isomorphism is also recommended.

§ 1.3 Typographical conventions

This section is non-normative.

The following typographic conventions are used in this specification:

`markup`
: Markup (elements, attributes, properties), machine processable values (string, characters, media types), property names, and file names are in red-orange monospace font.

variable
: A variable in pseudo-code or in an algorithm description is italicized.

definition
: A definition of a term, to be used elsewhere in this or other specifications, is italicized and in bold.

definition reference
: A reference to a definition *in this document* is underlined and is also an active link to the definition itself.

`markup definition reference`
: References to a definition *in this document*, when the reference itself is also a markup, is underlined, in a red-orange monospace font, and is also an active link to the definition itself.

external definition reference
: A reference to a definition *in another document* is underlined and italicized, and is also an active link to the definition itself.

`markup external definition reference`
: A reference to a definition *in another document*, when the reference itself is also a markup, is underlined and italicized in a red-orange monospace font, and is also an active link to the definition itself.

hyperlink
: A hyperlink is underlined and in blue.

[reference]
: A document reference (normative or informative) is enclosed in square brackets and links to the references section.

Explanation
: An expandable area to find a more detailed, non-normative explanation of a particular algorithmic step.

▶ **Explanation**

Logging
: An expandable area to find suggestions for implementations to log information about processing, which may be useful in comparing with other implementations, or with logs provided with each test case.

▶ **Logging**

NOTE

Notes are in light green boxes with a green left border and with a "Note" header in green. Notes are always informative.

EXAMPLE 1

```
Examples are in light khaki boxes, with khaki left border,
and with a numbered "Example" header in khaki.
Examples are always informative. The content of the example is in monospace font
syntax colored.

Examples may have tabbed navigation buttons
to show the results of transforming an example into other representations.

Code examples are generally given in a Turtle or TriG format for brevity,
where each line represents a single triple or quad.
Additionally, have the following implied directives:

BASE <http://example.com/>
PREFIX : <#>

Following the Turtle/TriG syntax rules, blank nodes always appear in the
`_:xyz` format.
```

§ 2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MUST*, *MUST NOT*, and *SHOULD* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

A conforming processor is a system which can generate the canonical n-quads form of an input dataset consistent with the algorithms defined in this specification.

The algorithms in this specification are normative, because to consistently reproduce the same canonical identifiers, implementations *MUST* strictly conform to the steps outlined in these algorithms.

NOTE

Implementers can partially check their level of conformance with this specification by successfully passing the test cases of the RDF Dataset Canonicalization test suite. Note, however, that passing all the tests in the test suite does not imply complete conformance to this specification. It only implies that the implementation conforms to the aspects tested by the test suite.

§ 3. Terminology

§ 3.1 Terms defined by this specification

canonical n-quads form
The canonicalized representation of a quad is defined in A. A Canonical form of N-Quads. A quad in canonical n-quads form represents a graph name, if present, in the same manner as a subject, and each quad is terminated with a single `LF` (line feed, code point `U+000A`).

canonicalization function
A canonicalization function maps RDF datasets into isomorphic datasets [RDF11-CONCEPTS]. Two datasets produce the same canonical result if and only if they are isomorphic. The RDFC-1.0 algorithm implements a canonicalization function. Some datasets may be constructed to prevent this algorithm from terminating in a reasonable amount of time (see 7.1 Dataset Poisoning), in which case the algorithm can be considered to be a ***partial canonicalization function***.

canonicalized dataset
A canonicalized dataset is the combination of the following:

- an RDF dataset — the input dataset,
- the input blank node identifier map — mapping blank nodes in the input dataset to blank node identifiers, and
- the issued identifiers map from the canonical issuer — mapping identifiers in the input dataset to canonical identifiers

A concrete serialization of a canonicalized dataset *MUST* label all blank nodes using the canonical blank node identifiers.

gossip path
A particular enumeration of every incident mention emanating from a blank node. This recursively includes transitively related mentions until any named node or blank node already labeled by a particular identifier issuer is reached. Gossip paths are encoded and operated on in

the RDFC-1.0 algorithm as strings. (See 4.8 Hash N-Degree Quads for more information on the construction of gossip paths.)

hash

The lowercase, hexadecimal representation of a message digest.

hash algorithm

The default hash algorithm used by RDFC-1.0, namely, SHA-256 [FIPS-180-4].

Implementations *MUST* support a parameter to define the hash algorithm, *MUST* support SHA-256 and SHA-384 [FIPS-180-4], and *SHOULD* support the ability to specify other hash algorithms. Using a different hash algorithm will generally result in different output than using the default.

NOTE

There is no expectation that the default hash algorithm will also be used by any application creating a hash digest of the canonical N-Quads result.

identifier issuer

An identifier issuer is used to issue new blank node identifiers. It maintains a blank node identifier issuer state.

input blank node identifier map

Records any blank node identifiers already assigned to the input dataset. If the input dataset is provided as an N-Quads document, the map relates blank nodes in the abstract input dataset to the blank node identifiers used within the N-Quads document, otherwise, identifiers are assigned arbitrarily for each blank node in the input dataset not previously identified.

NOTE

Implementations or environments might deal with blank node identifiers more directly; for example, some implementations might retain blank node identifiers in the parsed or abstract dataset. Implementations are expected to reuse these to enable usable mappings between input blank node identifiers and output blank node identifiers outside of the algorithm.

input dataset

The abstract RDF dataset that is provided as input to the algorithm.

mention

A node is mentioned in a quad if it is a component of that quad, as a subject, predicate, object, or graph name.

mention set

The set of all quads in a dataset that mention a node n is called the mention set of n, denoted Q_n.

quad

A tuple composed of subject, predicate, object, and graph name. This is a generalization of an RDF triple along with a graph name.

§ 3.2 Terms defined by cited specifications

blank node

A blank node as specified by [RDF11-CONCEPTS]. In short, it is a node in a graph that is neither an IRI, nor a literal.

blank node identifier

A blank node identifier as specified by [RDF11-CONCEPTS]. In short, it is a string that begins with `_:` that is used as an identifier for a blank node. Blank node identifiers are typically implementation-specific local identifiers; this document specifies an algorithm for deterministically specifying them.

Concrete syntaxes, like [Turtle] or [N-Quads], prepend blank node identifiers with the `_:` string to differentiate them from other nodes in the graph. This affects the canonicalization algorithm, which is based on calculating a hash over the representations of quads in this format.

default graph

The default graph as specified by [RDF11-CONCEPTS].

graph name

A graph name as specified by [RDF11-CONCEPTS].

IRI

An IRI (Internationalized Resource Identifier) is a string that conforms to the syntax defined in [RFC3987].

object

An object as specified by [RDF11-CONCEPTS].

predicate

A predicate as specified by [RDF11-CONCEPTS].

RDF dataset

A dataset as specified by [RDF11-CONCEPTS]. For the purposes of this specification, an RDF dataset is considered to be a set of quads

RDF graph

An RDF graph as specified by [RDF11-CONCEPTS].

RDF triple

A triple as specified by [RDF11-CONCEPTS].

string
A string is a sequence of zero or more Unicode characters.

subject
A subject as specified by [RDF11-CONCEPTS].

`true` and `false`
Values that are used to express one of two possible boolean states.

Unicode code point order
This refers to determining the order of two Unicode strings (`A` and `B`), using Unicode Codepoint Collation, as defined in [XPATH-FUNCTIONS], which defines a total ordering of strings comparing code points. Note that for UTF-8 encoded strings, comparing the byte sequences gives the same result as code point order.

§ 4. Canonicalization

Canonicalization is the process of transforming an input dataset to its serialized canonical form. That is, any two input datasets that contain the same information, regardless of their arrangement, will be transformed into the same serialized canonical form. The problem requires directed graphs to be deterministically ordered into sets of nodes and edges. This is easy to do when all of the nodes have globally-unique identifiers, but can be difficult to do when some of the nodes do not. Any nodes without globally-unique identifiers must be issued deterministic identifiers.

NOTE

This specification defines a canonicalized dataset to include stable identifiers for blank nodes, practical uses of which will always generate a canonical serialization of such a dataset.

In time, there may be more than one canonicalization algorithm and, therefore, for identification purposes, this algorithm is named the "RDF Canonicalization algorithm version 1.0" (***RDFC-1.0***).

Figure 1 provides an overview of RDFC-1.0, with steps 1 through 7 corresponding to the various steps described in 4.4.3 Algorithm.

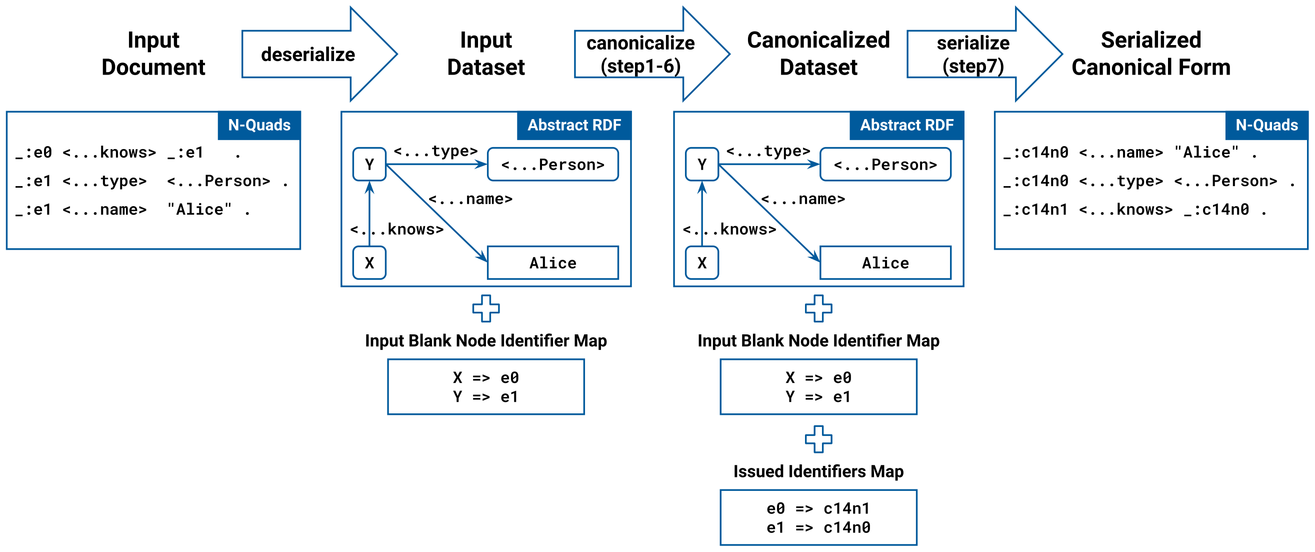

Figure 1 An illustrated overview of the RDFC-1.0 algorithm.
Image available in SVG .

§ 4.1 Overview

This section is non-normative.

To determine a canonical labeling, RDFC-1.0 considers the information connected to each blank node. Nodes with unique first degree information can immediately be issued a canonical identifier via the Issue Identifier algorithm. When a node has non-unique first degree information, it is necessary to determine all information that is transitively connected to it throughout the entire dataset. 4.6 Hash First Degree Quads defines a node’s first degree information via its first degree hash.

Hashes are computed from the information of each blank node. These hashes encode the mentions incident to each blank node. The hash of a string *s*, is the lower-case, hexadecimal representation of the result of passing *s* through a cryptographic hash function. By default, RDFC-1.0 uses the SHA-256 hash algorithm [FIPS-180-4].

NOTE

The "degree" terminology is used within this specification as colloquial way of describing the eccentricity or radius of any two nodes within a dataset. This concept is also related to "degrees of separation", as in, for example, "six degrees of separation". Nodes with unique first degree information can be considered nodes with a radius of one.

§ 4.2 Canonicalization State

When performing the steps required by the canonicalization algorithm, it is helpful to track state in a data structure called the ***canonicalization state***. The information contained in the canonicalization state is described below.

blank node to quads map
: A map that relates a blank node identifier to the quads in which they appear in the input dataset.

hash to blank nodes map
: A map that relates a hash to a list of blank node identifiers.

canonical issuer
: An identifier issuer, initialized with the prefix `c14n` (short for canonicalization), for issuing canonical blank node identifiers.

> NOTE
> Mapping all blank nodes to use this identifier spec means that an RDF dataset composed of two different RDF graphs will issue different identifiers than that for the graphs taken independently. This may happen anyway, due to automorphisms, or overlapping statements, but an identifier based on the resulting hash along with an issue sequence number specific to that hash would stand a better chance of surviving such minor changes, and allow the resulting information to be useful for RDF Diff.

§ 4.3 Blank Node Identifier Issuer State

The canonicalization algorithm issues identifiers to blank nodes. The Issue Identifier algorithm uses an identifier issuer to accomplish this task. The information an identifier issuer needs to keep track of is described below.

identifier prefix
: The identifier prefix is a string that is used at the beginning of an blank node identifier. It should be initialized to a string that is specified by the canonicalization algorithm. When generating a new blank node identifier, the prefix is concatenated with a identifier counter. For example, `c14n` is a proper initial value for the identifier prefix that would produce blank node identifiers like `c14n1`.

identifier counter

A counter that is appended to the identifier prefix to create an blank node identifier. It is initialized to `0`.

issued identifiers map

An ordered map that relates blank node identifiers to issued identifiers, to prevent issuance of more than one new identifier per existing identifier, and to allow blank nodes to be assigned identifiers some time after issuance.

§ 4.4 Canonicalization Algorithm

The canonicalization algorithm converts an input dataset into a canonicalized dataset or raises an error if the input dataset is determined to be overly complex. This algorithm will assign deterministic identifiers to any blank nodes in the input dataset.

§ 4.4.1 Overview

This section is non-normative.

RDFC-1.0 canonically labels an RDF dataset by assigning each blank node a canonical identifier. In RDFC-1.0, an RDF dataset *D* is represented as a set of quads of the form `< s, p, o, g >` where the graph component `g` is empty if and only if the triple `< s, p, o >` is in the default graph. It is expected that, for two RDF datasets, RDFC-1.0 returns the same canonically labeled list of quads if and only if the two datasets are isomorphic (i.e., the same modulo blank node identifiers).

RDFC-1.0 consists of several sub-algorithms. These sub-algorithms are introduced in the following sub-sections. First, we give a high level summary of RDFC-1.0.

1) Initialization. Initialize the state needed for the rest of the algorithm using 4.2 Canonicalization State. Also initialize the canonicalized dataset using the input dataset (which remains immutable) the input blank node identifier map (retaining blank node identifiers from the input if possible, otherwise assigning them arbitrarily); the issued identifiers map from the canonical issuer is added upon completion of the algorithm.

2) Compute first degree hashes. Compute the first degree hash for each blank node in the dataset using 4.6 Hash First Degree Quads.

3) Canonically label unique nodes. Assign canonical identifiers via 4.5 Issue Identifier Algorithm, in Unicode code point order, to each blank node whose first degree hash is unique.

4) Compute N-degree hashes for non-unique nodes. For each repeated first degree hash (proceeding in Unicode code point order), compute the N-degree hash via 4.8 Hash N-Degree Quads of every unlabeled blank node that corresponds to the given repeated hash.

5) Canonically label remaining nodes. In Unicode code point order of the N-degree hashes, issue canonical identifiers to each corresponding blank node using 4.5 Issue Identifier Algorithm. If more than one node produces the same N-degree hash, the order in which these nodes receive a canonical identifier does not matter.

6) Finish. Return the serialized canonical form of the canonicalized dataset. Alternatively, return the canonicalized dataset containing the input blank node identifier map and issued identifiers map.

§ 4.4.2 Examples

This section is non-normative.

EXAMPLE 2: Unique hashes

This example illustrates the Canonicalization Algorithm where all blank nodes have unique first-degree hashes.

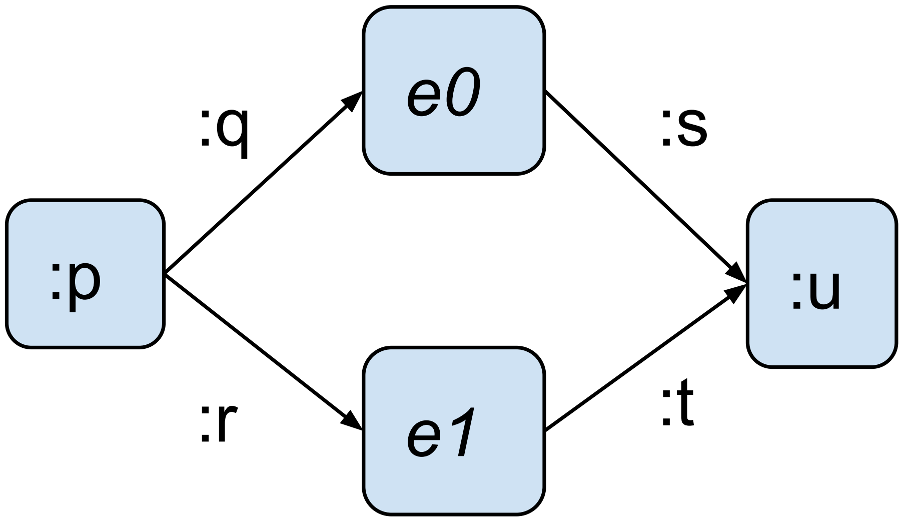

Figure 2 An illustration of a graph resulting in unique hashes.
Image available in SVG .

```
:p :q _:e0 .
:p :r _:e1 .
_:e0 :s :u .
_:e1 :t :u .
```

Step 2 is called twice, with each blank node (`e0` and `e1`) in the input dataset to populate blank node to quads map:

blank node	***Q***

`e0`	`:p :q e0 .`
	`e0 :s :u .`

`e1`	`:p :r e1 .`
	`e1 :t :u .`

Table 1 Blank node to quads map for unique hashes

Step 3 generates the first-degree hash for each blank node, which is explored further in Example 4:

hash	blank node(s)
21d1dd5ba21f3dee9d76c0c00c260fa6f5d5d65315099e553026f4828d0dc77a	e0
6fa0b9bdb376852b5743ff39ca4cbf7ea14d34966b2828478fbf222e7c764473	e1

Table 2 *Hash to blank nodes map for unique hashes*

Step 4 creates canonical identifiers for each blank node which has a unique hash:

blank node	canonical identifier
e0	c14n0
e1	c14n1

Table 3 *Canonical identifiers for blank nodes with unique hashes*

Step 5 has no effect, as there are no remaining blank nodes without canonical identifiers.

Step 6 generates the canonicalized dataset by mapping blank node identifiers in the input dataset with canonical identifiers:

```
:p :q _:c14n0 .
:p :r _:c14n1 .
_:c14n0 :s :u .
_:c14n1 :t :u .
```

EXAMPLE 3: Shared hashes

This example illustrates the Canonicalization Algorithm where hashing the statements mentioning those blank nodes have overlapping results.

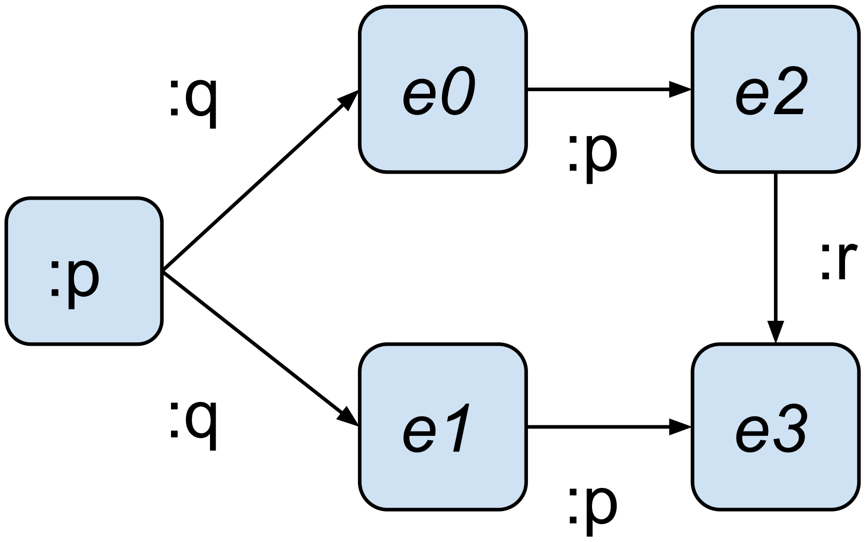

Figure 3 An illustration of a graph resulting in shared hashes.
Image available in SVG .

```
:p :q _:e0 .
:p :q _:e1 .
_:e0 :p _:e2 .
_:e1 :p _:e3 .
_:e2 :r _:e3 .
```

Step 2 is called four times, with each blank node (`e0`, `e1`, `e2`, and `e3`) to populate blank node to quads map:

blank node	***Q***

`e0`	`:p :q e0 .`
	`e0 :p e2 .`

`e1`	`:p :q e1 .`
	`e1 :p e3 .`

`e2`	`e0 :p e2 .`
	`e2 :r e3 .`

blank node	***Q***

`e3`	`e1 :p e3 .`
	`e2 :r e3 .`

Table 4 Blank node to quads map for shared hashes

Step 3 generates the first-degree hash for each blank node, which is explored further in Example 5 (note that the hashes for `e0` and `e1` are shared):

hash	***blank node(s)***
`3b26142829b8887d011d779079a243bd61ab53c3990d550320a17b59ade6ba36`	`e0`, `e1`
`15973d39de079913dac841ac4fa8c4781c0febfba5e83e5c6e250869587f8659`	`e2`
`7e790a99273eed1dc57e43205d37ce232252c85b26ca4a6ff74ff3b5aea7bccd`	`e3`

Table 5 Hash to blank nodes map shared hashes

Step 4 creates canonical identifiers for each blank node which has a unique hash:

blank node	***canonical identifier***
`e2`	`c14n0`
`e3`	`c14n1`

Table 6 Canonical identifiers for blank nodes with shared hashes

Step 5 is run on `e0` and `e1`, separately, which share the same hash, and use separate instances of a *temporary issuer*, (as explored in Example 7) to create the *hash path list* composed of the hash result and temporary identifier mappings from Hash N-Degree Quads algorithm for each of these blank nodes:

	original identifier	***temporary identifier***
temporary issuer mappings	`e0`	`b0`
hash	`fbc300de5afafd97a4b9ee1e72b57754dcdcb7ebb724789ac6a94a5b82a48d30`	

Table 7 Result for `e0`

	original identifier	***temporary identifier***
temporary issuer mappings	`e1`	`b0`
hash	`2c0b377baf86f6c18fed4b0df6741290066e73c932861749b172d1e5560f5045`	

Table 8 Result for `e1`

Step 5.3 creates the canonical identifiers for the temporary identifiers established in the previous step running in order of the *hash* component from each result. This updates the canonical issuer with the following mappings:

blank node	canonical identifier
`e0`	`c14n3`
`e1`	`c14n2`
`e2`	`c14n0`
`e3`	`c14n1`

Table 9 *Blank node to canonical identifiers*

Step 6 updates the canonicalized dataset with the canonical issuer, containing an issued identifiers map mapping blank node identifiers from the input dataset to their canonical identifiers:

```
:p :q _:c14n2 .
:p :q _:c14n3 .
_:c14n0 :r _:c14n1 .
_:c14n2 :p _:c14n1 .
_:c14n3 :p _:c14n0 .
```

§ 4.4.3 Algorithm

The following algorithm will run with a minimal number of iterations in each step for typical input datasets. In some extreme cases, the algorithm can behave poorly, particularly in Step 5. Implementations *MUST* defend against potential denial-of-service attacks by raising suitable exceptions and terminating early. See 7.1 Dataset Poisoning for further information.

NOTE

Implementations can consider placing limits on the number of calls to 4.8 Hash N-Degree Quads based on the number of blank nodes in the hash to blank nodes map. For most typical datasets, more than a couple of iterations on 4.8 Hash N-Degree Quads per blank node would be unusual.

1) Create the canonicalization state. If the input dataset is an N-Quads document, parse that document into a dataset in the canonicalized dataset, retaining any blank node identifiers used within that document in the input blank node identifier map; otherwise arbitrary identifiers are assigned for each blank node.

▶ **Explanation**

2) For every quad *Q* in input dataset:

> **2.1)** For each blank node that is a component of *Q*, add a reference to *Q* from the map entry for the blank node identifier *identifier* in the blank node to quads map, creating a new entry if necessary, using the identifier for the blank node found in the input blank node identifier map.
>
> ▶ **Explanation**

▶ **Logging**

3) For each key *n* in the blank node to quads map:

▶ **Explanation**

> **3.1)** Create a hash, $h_f(n)$, for *n* according to the Hash First Degree Quads algorithm.
>
> **3.2)** Append *n* to the value associated to $h_f(n)$ in hash to blank nodes map, creating a new entry if necessary.

▶ **Logging**

4) For each *hash* to *identifier list* map entry in hash to blank nodes map, code point ordered by *hash*:

▶ **Explanation**

> **4.1)** If *identifier list* has more than one entry, continue to the next mapping.
>
> **4.2)** Use the Issue Identifier algorithm, passing canonical issuer and the single blank node identifier, *identifier* in *identifier list* to issue a canonical replacement identifier for *identifier*.
>
> **4.3)** Remove the map entry for *hash* from the hash to blank nodes map.

▶ **Logging**

5) For each *hash* to *identifier list* map entry in hash to blank nodes map, code point ordered by *hash*:

▶ **Explanation**

▶ **Logging**

> **5.1)** Create *hash path list* where each item will be a result of running the Hash N-Degree Quads algorithm.
>
> ▶ **Explanation**

5.2) For each blank node identifier *n* in *identifier list*:

5.2.1) If a canonical identifier has already been issued for *n*, continue to the next blank node identifier.

5.2.2) Create *temporary issuer*, an identifier issuer initialized with the prefix `b`.

5.2.3) Use the Issue Identifier algorithm, passing *temporary issuer* and *n*, to issue a new temporary blank node identifier b_n to *n*.

5.2.4) Run the Hash N-Degree Quads algorithm, passing the canonicalization state, *n* for *identifier*, and *temporary issuer*, appending the result to the *hash path list*.

▶ **Logging**

5.3) For each *result* in the *hash path list*, code point ordered by the *hash* in *result*:

▶ **Explanation**

5.3.1) For each blank node identifier, *existing identifier*, that was issued a temporary identifier by *identifier issuer* in *result*, issue a canonical identifier, in the same order, using the Issue Identifier algorithm, passing canonical issuer and *existing identifier*.

▶ **Explanation**

▶ **Logging**

6) Add the issued identifiers map from the canonical issuer to the canonicalized dataset.

▶ **Explanation**

▶ **Logging**

7) Return the serialized canonical form of the canonicalized dataset. Upon request, alternatively (or additionally) return the canonicalized dataset itself, which includes the input blank node identifier map, and issued identifiers map from the canonical issuer.

NOTE

Technically speaking, one implementation might return a canonicalized dataset that maps particular blank nodes to different identifiers than another implementation, however, this only occurs when there are isomorphisms in the dataset such that a canonically serialized expression of the dataset would appear the same from either implementation.

▶ **Explanation**

§ 4.5 Issue Identifier Algorithm

This algorithm issues a new blank node identifier for a given existing blank node identifier. It also updates state information that tracks the order in which new blank node identifiers were issued. The order of issuance is important for canonically labeling blank nodes that are isomorphic to others in the dataset.

§ 4.5.1 Overview

The algorithm maintains an issued identifiers map to relate an existing blank node identifier from the input dataset to a new blank node identifier using a given identifier prefix (`c14n`) with new identifiers issued by appending an incrementing number. For example, when called for a blank node identifier such as `e3`, it might result in a issued identifier of `c14n1`.

§ 4.5.2 Algorithm

The algorithm takes an identifier issuer *I* and an *existing identifier* as inputs. The output is a new *issued identifier*. The steps of the algorithm are:

1) If there is a map entry for *existing identifier* in issued identifiers map of *I*, return it.

2) Generate *issued identifier* by concatenating identifier prefix with the string value of identifier counter.

3) Add an entry mapping *existing identifier* to *issued identifier* to the issued identifiers map of *I*.

4) Increment identifier counter.

5) Return *issued identifier*.

§ 4.6 Hash First Degree Quads

This algorithm calculates a hash for a given blank node across the quads in a dataset in which that blank node is a component. If the hash uniquely identifies that blank node, no further examination is necessary. Otherwise, a hash will be created for the blank node using the algorithm in 4.8 Hash N-Degree Quads invoked via 4.4 Canonicalization Algorithm.

§ 4.6.1 Overview

This section is non-normative.

To determine whether the first degree information of a node n is unique, a hash is assigned to its mention set, Q_n. The first degree hash of a blank node n, denoted $h_f(n)$, is the hash that results from 4.6 Hash First Degree Quads when passing n. Nodes with unique first degree hashes have unique first degree information.

For consistency, blank node identifiers used in Q_n are replaced with placeholders in a canonical n-quads serialization of that quad. Every blank node component is replaced with either `a` or `z`, depending on if that component is n or not.

The resulting serialized quads are then code point ordered, concatenated, and hashed. This hash is the first degree hash of n, $h_f(n)$.

§ 4.6.2 Examples

This section is non-normative.

EXAMPLE 4: Unique hashes

This example illustrates hashing quads containing blank nodes where hashing the statements mentioning those blank nodes generates unique results.

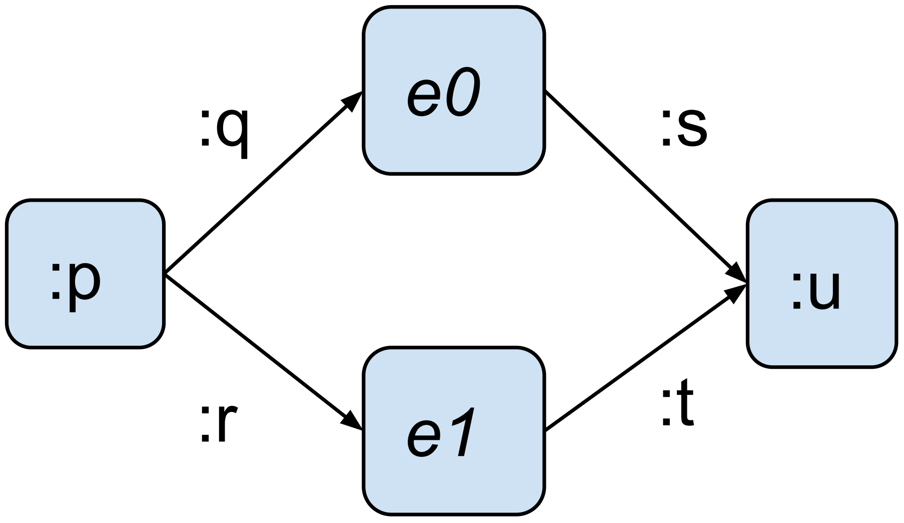

Figure 4 An illustration of a graph resulting in unique hashes.
Image available in SVG .

```
:p :q _:e0 .
:p :r _:e1 .
_:e0 :s :u .
_:e1 :t :u .
```

The algorithm will be called twice, with each blank node (`e0` and `e1`). Running the algorithm with the reference node `e0` results in the following quads, after replacing blank nodes:

```
:p :q _:a .
_:a :s :u .
```

These are then serialized to canonical n-quads form: `'<http://example.com/#p> <http://example.com/#q> _:a .\n_:a <http://example.com/#s> <http://example.com/#u> .\n'`, concatenated and hashed using the hash algorithm (SHA-256) resulting in `21d1dd5ba21f3dee9d76c0c00c260fa6f5d5d65315099e553026f4828d0dc77a`.

The algorithm is run a second time with the reference node *`e1`* resulting in the following quads:

```
:p :r _:a .
_:a :t :u .
```

These are then serialized to canonical n-quads form: `'<http://example.com/#p> <http://example.com/#r> _:a .\n_:a <http://example.com/#t> <http://example.com/#u> .\n'`, concatenated and hashed as before resulting in `6fa0b9bdb376852b5743ff39ca4cbf7ea14d34966b2828478fbf222e7c764473`.

Thus the generated hashes each reference just a single blank node, allowing the canonicalization algorithm to use only the Hash First Degree Quads algorithm.

EXAMPLE 5: Shared hashes

This example illustrates hashing quads containing blank nodes where hashing the statements mentioning those blank nodes have overlapping results.

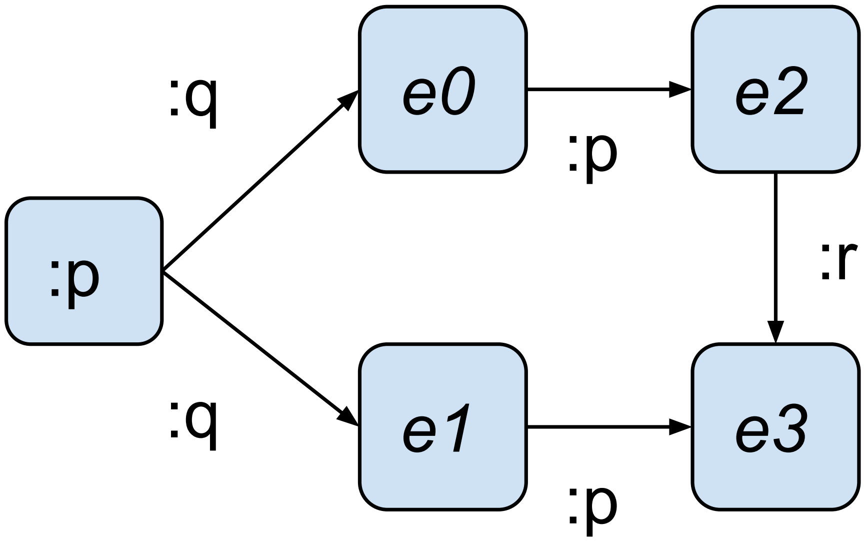

Figure 5 An illustration of a graph resulting in shared hashes.
Image available in SVG .

```
:p :q _:e0 .
:p :q _:e1 .
_:e0 :p _:e2 .
_:e1 :p _:e3 .
_:e2 :r _:e3 .
```

The algorithm will be called four times, with each blank node (`e0`, `e1`, `e2`, and `e3`). Running the algorithm with the reference node `e0` results in the following quads, after replacing blank nodes:

```
:p :q _:a .
_:a :p _:z .
```

Which hashes to: `3b26142829b8887d011d779079a243bd61ab53c3990d550320a17b59ade6ba36`.

Note that using reference node `e1` results in the same quads, and thus results in the same hash.

Using the reference node `e2` results in the following quads:

```
_:z :p _:a .
_:a :r _:z .
```

Which hashes to: `15973d39de079913dac841ac4fa8c4781c0febfba5e83e5c6e250869587f8659`.

Lastly, using the reference node `e3` results in the following quads:

```
_:z :p _:a .
_:z :r _:a .
```

Which hashes to: `7e790a99273eed1dc57e43205d37ce232252c85b26ca4a6ff74ff3b5aea7bccd`.

The hashes for `e2` and `e3` are unique, but `e0`, and `e1` share a common hash, which will require the use of the Hash N-Degree Quads Algorithm, as it is necessary to consider quads further removed from the direct mentions to determine a unique hash.

§ 4.6.3 Algorithm

This algorithm takes the canonicalization state and a ***reference blank node identifier*** as inputs.

1) Initialize ***nquads*** to an empty list. It will be used to store quads in canonical n-quads form.

2) Get the list of quads *quads* from the map entry for reference blank node identifier in the blank node to quads map.

3) For each quad *quad* in *quads*:

  3.1) Serialize the quad in canonical n-quads form with the following special rule:

    3.1.1) If any component in *quad* is an blank node, then serialize it using a special identifier as follows:

      3.1.1.1) If the blank node's existing blank node identifier matches the reference blank node identifier then use the blank node identifier `a`, otherwise, use the blank node identifier `z`.

4) Sort nquads in Unicode code point order.

5) Return the hash that results from passing the sorted and concatenated nquads through the hash algorithm.

▶ **Logging**

§ 4.7 Hash Related Blank Node

This algorithm generates a hash for some blank node component of a quad, considering its position within that quad. This is used as part of the Hash N-Degree Quads algorithm to characterize the blank nodes related to some particular blank node within their mention sets.

§ 4.7.1 Overview

This section is non-normative.

An *identifier* for a blank node, *related*, which is a component of some quad, is used along with information describing its position within the quad to create a representation of that blank node within the quad, which is then hashed. The identifier is first found from the following:

1) An issued identifier from canonical issuer in the canonicalization state,

2) An issued identifier from *issuer*, or

3) the result of the Hash N-Degree Quads algorithm for *related*.

§ 4.7.2 Examples

This section is non-normative.

EXAMPLE 6: With assigned canonical identifier

This example illustrates generating the related hash for a blank node component of a quad, where the component is in the object position and has already had an assigned canonical identifier from the canonical issuer.

Given the predicate `:p`, *related* blank node *`e2`* (already assigned the canonical blank node identifier *`c14n0`*), an *input* is created as `o<http://example.com/#p>_:c14n0`, which hashes to `29cf7e22790bc2ed395b81b3933e5329fc7b25390486085cac31ce7252ca60fa`.

If such an identifier had not yet been established, it would instead use the results of the Hash First Degree Quads algorithm, without the leading `_:`.

§ 4.7.3 Algorithm

This algorithm creates a hash to identify how one blank node is related to another. It takes the canonicalization state, a *related* blank node identifier, a *quad*, an identifier issuer, *issuer*, and a string *position* as inputs.

1) Initialize a string *input* to the value of *position*.

2) If *position* is not `g`, append `<`, the value of the predicate in *quad*, and `>` to *input*.

3) If there is a canonical identifier for *related*, or an identifier issued by *issuer*, append the string `_:`, followed by that identifier (using the canonical identifier if present, otherwise the one issued by *issuer*) to *input*.

▶ **Explanation**

4) Otherwise, append the result of the Hash First Degree Quads algorithm, passing *related* to *input*.

▶ **Explanation**

5) Return the hash that results from passing *input* through the hash algorithm.

▶ **Explanation**

▶ **Logging**

§ 4.8 Hash N-Degree Quads

This algorithm calculates a hash for a given blank node across the quads in a dataset in which that blank node is a component for which the hash does not uniquely identify that blank node. This is done by expanding the search from quads directly referencing that blank node (the mention set), to those quads which contain nodes which are also components of quads in the mention set, called the gossip path. This process proceeds in every greater degrees of indirection until a unique hash is obtained.

§ 4.8.1 Overview

This section is non-normative.

Usually, when trying to determine if two nodes in a graph are equivalent, you simply compare their identifiers. However, what if the nodes don't have identifiers? Then you must determine if the two nodes have equivalent connections to equivalent nodes all throughout the whole graph. This is called the graph isomorphism problem. This algorithm approaches this problem by considering how one might draw a graph on paper. You can test to see if two nodes are equivalent by drawing the graph twice. The first time you draw the graph the first node is drawn in the center of the page. If you can draw the graph a second time such that it looks just like the first, except the second node is in the center of the page, then the nodes are equivalent. This algorithm essentially defines a deterministic way to draw a graph where, if you begin with a particular node, the graph will always be drawn the same way. If two graphs are drawn the same way with two different nodes, then the nodes are equivalent. A hash is used to indicate a particular way that the graph has been drawn and can be used to compare nodes.

When two blank nodes have the same first degree hash, extra steps must be taken to detect global, or *N*-degree, distinctions. All information that is in any way connected to the blank node *n* through other blank nodes, even transitively, must be considered.

To consider all transitive information, the algorithm traverses and encodes all possible paths of incident mentions emanating from *n*, called gossip paths, that reach every unlabeled blank node connected to *n*. Each unlabeled blank node is assigned a temporary identifier in the order in which it is reached in the gossip path being explored. The mentions that are traversed to reach connected blank nodes are encoded in these paths via related hashes. This provides a deterministic way to order all paths coming from *n* that reach all blank nodes connected to n without relying on input blank node identifiers.

This algorithm works in concert with the main canonicalization algorithm to produce a unique, deterministic identifier for a particular blank node. This hash incorporates all of the information that is connected to the blank node as well as how it is connected. It does this by creating deterministic paths that emanate out from the blank node through any other adjacent blank nodes.

Ultimately, the algorithm selects the shortest gossip path (based on its encoding as a string), distributing canonical identifiers to the unlabeled blank nodes in the order in which they appear in this path. The hash of this encoded shortest path, called the N-degree hash of *n*, distinguishes *n* from other blank nodes in the dataset.

For clarity, we consider a gossip path encoded via the string *s* to be shortest provided that:

> **1)** The length of *s* is less than or equal to the length of any other gossip path string *s′*.
>
> **2)** If *s* and *s′* have the same length (as strings), then *s* is code point ordered less than or equal to *s′*.

For example, *abc* is shorter than *bbc*, whereas *abcd* is longer than *bcd*.

The following provides a high level outline for how the N-degree hash of *n* is computed along the shortest gossip path. Note that the full algorithm considers all gossip paths, ultimately returning the hash of the shortest encoded path.

> **1) Compute related hashes**. Compute the related hash H_n set for *n*, i.e., all first degree mentions between *n* and another blank node. Note that this includes both unlabeled blank nodes and those already issued a canonical identifier (labeled blank nodes).
>
> **2) Explore mentions**. Given the related hash *x* in H_n, record *x* in the data to hash D_n. Determine whether each blank node reachable via the mention with related hash *x* has already received an identifier.
>
> > **2.1) Record the identifiers of labeled nodes**. If a blank node already has an identifier, record its identifier in D_n once for every mention with related hash *x*. Skip to the next related hash in H_n and repeat step 2.
> >
> > **2.2) Distribute and record temporary identifiers to unlabeled nodes**. For each unlabeled blank node, assign it a temporary identifier according to the order in which it is reached in the gossip path, recording its given identifier in D_n (including repetitions). Add each unlabeled node to the recursion list $R_n(x)$ in this same order (omitting repetitions).
> >
> > **2.3) Recurse on newly labeled nodes**. For each n_i in $R_n(x)$

2.3.1) Record its identifier in D_n

2.3.2) Append $< r(i) >$ to D_n where $r(i)$ is the data to hash that results from returning to step 1, replacing n with n_i.

3) Compute the *N*-degree hash of n. Hash D_n to return the N-degree hash of n, namely $h_N(n)$. Return the updated issuer I_n that has now distributed temporary identifiers to all unlabeled blank nodes connected to n.

As described above in step 2.3, *HN* recurses on each unlabeled blank node when it is first reached along the gossip path being explored. This recursion can be visualized as moving along the path from n to the blank node n_i that is receiving a temporary identifier. If, when recursing on n_i, another unlabeled blank node n_j is discovered, the algorithm again recurses. Such a recursion traces out the gossip path from n to n_j via n_i.

The recursive hash $r(i)$ is the hash returned from the completed recursion on the node n_i when computing $h_N(n)$. Just as $h_N(n)$ is the hash of D_n, we denote the data to hash in the recursion on n_i as D_i. So, $r(i) = h(D_i)$. For each related hash $x \in H_n$, $R_n(x)$ is called the *recursion list* on which the algorithm recurses.

§ 4.8.2 Examples

This section is non-normative.

EXAMPLE 7: Shared hashes

This example revisits Example 3 to illustrate the operation of the Hash N-Degree Quads Algorithm on the blank nodes (`e0` and `e1`) which resulted in a shared result from the Hash First Degree Quads Algorithm. The other blank nodes (`e2` and `e3`) had unique hashes when processed by the Hash N-Degree Quads Algorithm, so that canonical identifiers have already been issued, as recorded in the identifier issuer instance.

blank node	***canonical identifier***
`e2`	`c14n0`
`e3`	`c14n1`

Table 10 Established canonical identifiers from canonical issuer

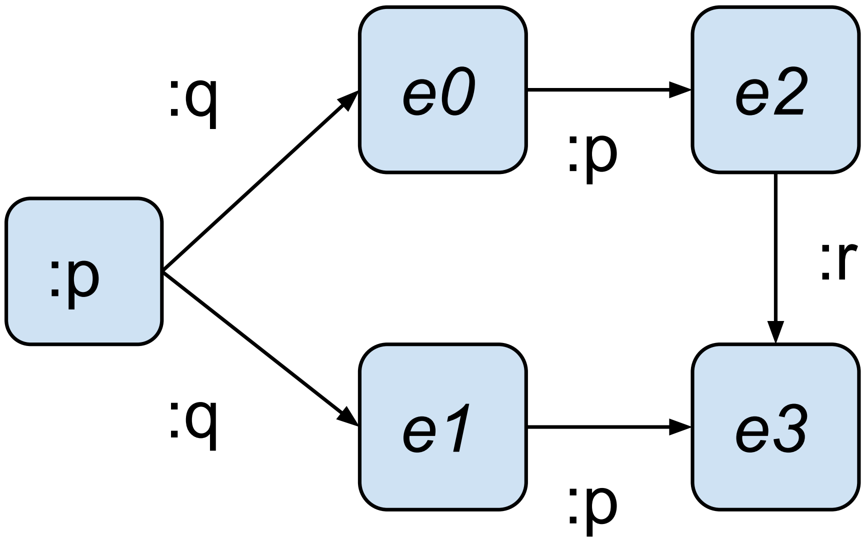

Figure 6 An illustration of a graph resulting in shared hashes.
Image available in SVG .

```
:p :q _:e0 .
:p :q _:e1 .
_:e0 :p _:e2 .
_:e1 :p _:e3 .
_:e2 :r _:e3 .
```

The algorithm will be called twice, with each blank node which did not result in a unique hash from the Hash First Degree Quads Algorithm (`e0` and `e1`). The map entry for `e0` in the blank node to quads map results in the following quads:

```
:p :q _:e0 .
_:e0 :p _:e2 .
```

When called, the *temporary issuer* has the following mappings:

blank node	***temporary identifier***
`e0`	`b0`

Table 11 Initial mappings in temporary issuer for `e0`

Step 3 iterates over each of these quads to find blank node *components* to populate H_n with the result of running the Hash Related Blank Node algorithm using the *position* of that *component* within the quad.

related	***quad***	***position***	***hash***
`e2`	`e0 :p e2 .`	`o`	`29cf7e2279...7252ca60fa`

Table 12 Hash for `e2` related to `e0`

which results in the following H_n:

related hash	***blank node list***
`29cf7e22790bc2ed395b81b3933e5329fc7b25390486085cac31ce7252ca60fa`	`[ e2 ]`

Table 13 Blank node list for hash

Step 5 iterates over each *related hash* which is a key in H_n, which can have one or more related blank nodes, which determines the shortest gossip path between the two nodes (`e0` and `e2`). In this case, the hash maps to just `e2`, for which a canonical identifier has already been chosen (`c14n0`), so there is a single permutation resulting in one candidate path. The resulting *chosen path* is `_:c14n0`.

The string *data to hash* is composed of the single *related hash* and the *chosen path*: `29cf7e22790bc2ed395b81b3933e5329fc7b25390486085cac31ce7252ca60fa_:c14n0`, which hashes to `fbc300de5afafd97a4b9ee1e72b57754dcdcb7ebb724789ac6a94a5b82a48d30`, which, along with the *temporary issuer* used to traverse these paths, is the result of the algorithm:

temporary issuer mappings	***original identifier***	***temporary identifier***
	`e0`	`b0`
hash	`fbc300de5afafd97a4b9ee1e72b57754dcdcb7ebb724789ac6a94a5b82a48d30`	

Table 14 Result for `e0`

The algorithm is called again for `e1`.

blank node	***temporary identifier***
`e1`	`b0`

Table 15 Initial mappings in temporary issuer for `e1`

The map entry for `e1` in the blank node to quads map results in the following quads:

```
:p :q _:e1 .
_:e1 :p _:e3 .
```

Step 3 again calculates blank node *components* using Hash Related Blank Node algorithm.

related	***quad***	***position***	***hash***
`e3`	`e1 :p e3 .`	`o`	`b7956ea1d6...ff6b098216`

Table 16 Hash for `e3` *related to* `e1`

which results in the following H_n:

related hash	***blank node list***
`b7956ea1d654d5824496eb439a1f2b79478bd7d02d4a115f4c97cbff6b098216`	`[ e3 ]`

Table 17 Blank node list for hash

Step 5 runs in essentially the same manner, mapping just `e3` having the canonical identifier `c14n1`, so there is again a single permutation resulting in one candidate path. The resulting *chosen path* is `_:c14n1`.

The string *data to hash* is composed of the single *related hash* and the *chosen path*: `af54b9512b1ef069205e8e41bc5a96e86a108b0389caa5029f2c3fd0bc465246_:c14n1`, which hashes to `767a5e66f509221f45003a16c12a89d4d9675cfa51ffa80459b63606bdfc2ada`.

The string *data to hash* is composed of the single *related hash* and the *chosen path*: `b7956ea1d654d5824496eb439a1f2b79478bd7d02d4a115f4c97cbff6b098216_:c14n1`, which hashes to `fbc300de5afafd97a4b9ee1e72b57754dcdcb7ebb724789ac6a94a5b82a48d30`, which, along with the *temporary issuer* used to traverse these paths, is the result of the algorithm:

temporary issuer mappings	***original identifier***	***temporary identifier***
	`e1`	*`b0`*
hash	`2c0b377baf86f6c18fed4b0df6741290066e73c932861749b172d1e5560f5045`	

Table 18 Result for `e1`

§ 4.8.3 Algorithm

The inputs to this algorithm are the canonicalization state, the *identifier* for the blank node to recursively hash quads for, and path identifier *issuer* which is an identifier issuer that issues temporary blank node identifiers. The output from this algorithm will be a hash and the identifier issuer used to help generate it.

▶ **Logging**

1) Create a new map H_n for relating hashes to related blank nodes.

2) Get a reference, *quads*, to the list of quads from the map entry for *identifier* in the blank node to quads map.

▶ **Explanation**

▶ **Logging**

3) For each *quad* in *quads*:

▶ **Explanation**

3.1) For each *component* in *quad*, where *component* is the subject, object, or graph name, and it is a blank node that is not identified by *identifier*:

3.1.1) Set *hash* to the result of the Hash Related Blank Node algorithm, passing the blank node identifier for *component* as *related*, *quad*, *issuer*, and *position* as either `s`, `o`, or `g` based on whether *component* is a subject, object, graph name, respectively.

3.1.2) Add a mapping of *hash* to the blank node identifier for *component* to H_n, adding an entry as necessary.

▶ **Logging**

4) Create an empty string, *data to hash*.

5) For each *related hash* to *blank node list* mapping in H_n, code point ordered by *related hash*:

▶ **Explanation**

▶ **Logging**

5.1) Append the *related hash* to the *data to hash*.

5.2) Create a string *chosen path*.

5.3) Create an unset *chosen issuer* variable.

5.4) For each permutation *p* of *blank node list*:

▶ **Logging**

5.4.1) Create a copy of *issuer*, *issuer copy*.

5.4.2) Create a string *path*.

5.4.3) Create a *recursion list*, to store blank node identifiers that must be recursively processed by this algorithm.

5.4.4) For each *related* in *p*:

5.4.4.1) If a canonical identifier has been issued for *related* by canonical issuer, append the string `_:`, followed by the canonical identifier for *related*, to *path*.

▶ **Explanation**

5.4.4.2) Otherwise:

5.4.4.2.1) If *issuer copy* has not issued an identifier for *related*, append *related* to *recursion list*.

▶ **Explanation**

5.4.4.2.2) Use the Issue Identifier algorithm, passing *issuer copy* and the *related*, and append the string `_:`, followed by the result, to *path*.

5.4.4.3) If *chosen path* is not empty and the length of *path* is greater than or equal to the length of *chosen path* and *path* is greater than *chosen path* when considering code point order, then skip to the next permutation *p*.

▶ **Explanation**

▶ **Explanation**

▶ **Logging**

5.4.5) For each *related* in *recursion list*:

▶ **Explanation**

▶ **Logging**

5.4.5.1) Set *result* to the result of recursively executing the Hash N-Degree Quads algorithm, passing the canonicalization state, *related* for *identifier*, and *issuer copy* for *path identifier issuer*.

▶ **Logging**

5.4.5.2) Use the Issue Identifier algorithm, passing *issuer copy* and *related*; append the string `_:`, followed by the result, to *path*.

5.4.5.3) Append `<`, the hash in *result*, and `>` to *path*.

5.4.5.4) Set *issuer copy* to the identifier issuer in result.

5.4.5.5) If *chosen path* is not empty and the length of *path* is greater than or equal to the length of *chosen path* and *path* is greater than *chosen path* when considering code point order, then skip to the next *p*.

▶ **Explanation**

5.4.6) If *chosen path* is empty or *path* is less than *chosen path* when considering code point order, set *chosen path* to *path* and *chosen issuer* to *issuer copy*.

5.5) Append *chosen path* to *data to hash*.

▶ **Logging**

5.6) Replace *issuer*, by reference, with*chosen issuer*.

6) Return *issuer* and the hash that results from passing *data to hash* through the hash algorithm.

▶ **Logging**

§ 5. Serialization

This section describes the process of creating a serialized [N-Quads] representation of a canonicalized dataset.

The ***serialized canonical form*** of a canonicalized dataset is an N-Quads document [N-QUADS] created by representing each quad from the canonicalized dataset in canonical n-quads form, sorting them into code point order, and concatenating them. (Note that each canonical N-Quads statement ends with a new line, so no additional separators are needed in the concatenation.) The resulting document has a media type of `application/n-quads`, as described in C. N-Quads Internet Media Type, File Extension and Macintosh File Type of [N-QUADS].

When serializing quads in canonical n-quads form, components which are blank nodes *MUST* be serialized using the canonical label associated with each blank node from the issued identifiers map component of the canonicalized dataset.

EXAMPLE 8: Canonical N-Quads representation of Unique Hashes

This example illustrates the result of serializing the canonicalized dataset described in Example 2.

```
<http://example.com/p> <http://example.com/q> _:c14n0 .
<http://example.com/p> <http://example.com/r> _:c14n1 .
_:c14n0 <http://example.com/s> <http://example.com/u> .
_:c14n1 <http://example.com/t> <http://example.com/u> .
```

§ 6. Privacy Considerations

This section is non-normative.

The nature of the canonicalization algorithm inherently correlates its output, i.e., the canonical labels and the sorted order of quads, with the input dataset. This could pose issues, particularly when dealing with datasets containing personal information. For example, even if certain information is removed from the canonicalized dataset for some privacy-respecting reason, there remains the possibility that a third party could infer the omitted data by analyzing the canonicalized dataset. If it is necessary to decouple the canonicalization algorithm's input and output, some suitable post-processing methods for the output of the canonicalization should be performed. This specification has been designed to help make additional processing easier, but other specifications that build on top of this one are responsible for providing any specific details. See Selective Disclosure in *Verifiable Credential Data Integrity 1.0* [VC-DATA-INTEGRITY] for more details about such post-processing methods.

§ 7. Security Considerations

This section is non-normative.

§ 7.1 Dataset Poisoning

This section is non-normative.

The canonicalization algorithm examines every difference in the information connected to blank nodes in order to ensure that each will properly receive its own canonical identifier. This process can be exploited by attackers to construct datasets which are known to take large amounts of computing time to canonicalize, but that do not express useful information or express it using unnecessary complexity. Implementers of the algorithm are expected to add mitigations that will, by default, abort canonicalizing problematic inputs.

Suggested mitigations include, but are not limited to:

- providing a configurable timeout with a default value applicable to an implementation's common use
- providing a configurable limit on the number of iterations of steps performed in the algorithm, particularly recursive steps and permutations of long lists

Additionally, software that uses implementations of the algorithm can employ best-practice schema validation to reject data that does not meet application requirements, thereby preventing useless poison datasets from being processed. However, such mitigations are application specific and not directly applicable to implementers of the canonicalization algorithm itself.

§ 7.2 Insecure Hash Algorithms

This section is non-normative.

It is possible that the default hash algorithm used by RDFC-1.0 might become insecure at some point in the future. To mitigate this, this algorithm and implementations of it can be parameterized to use a different hash function, without the need to make any changes to the canonicalization algorithm itself. However, using a different hash algorithm will generally lead to different results; applications making use of this specification should carefully weigh the advantages and disadvantages of using an alternative hash function.

NOTE

The possible implications of the default hash algorithm becoming insecure are mitigated by that fact that no internal hash values are revealed, and the canonicalization algorithm is designed to cope with first-degree hash collisions.

§ 8. Use Cases

This section is non-normative.

The use cases that have driven the development of the RDF Dataset Canonicalization algorithm are documented in a separate document. It includes further background and explanations for the design decisions taken [RCH-EXPLAINER].

§ 9. Examples

This section is non-normative.

§ 9.1 Duplicate Paths

This example illustrates a more complicated example where the same paths through blank nodes are duplicated in a graph, but use different blank node identifiers.

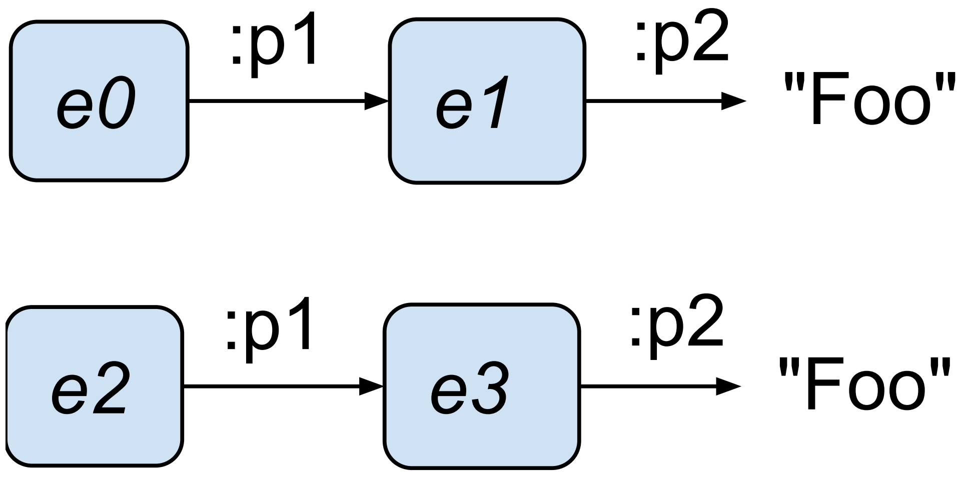

Figure 7 An illustration of a graph with duplicated paths.
Image available in SVG .

```
_:e0 :p1 _:e1 .
_:e1 :p2 "Foo" .
_:e2 :p1 _:e3 .
_:e3 :p2 "Foo" .
```

The following is a summary of the more detailed execution log found here.

EXAMPLE 9: Duplicate Paths

Step 2 of the Canonicalization Algorithm is called four times, with each blank node (`e0`, `e1`, `e2`, and `e3`) to populate blank node to quads map:

blank node	***Q***
`e0`	`_:e0 :p1 _:e1 .`

`e1`	`_:e0 :p1 _:e1 .`
	`_:e1 :p2 "Foo" .`

`e2`	`_:e2 :p1 _:e3 .`

`e3`	`_:e2 :p1 _:e3 .`
	`_:e3 :p2 "Foo" .`

Table 19 Blank node to quads map for duplicate paths

Step 3 generates the first-degree hash for each blank node.

For `e0`, the Hash First Degree Quads algorithm uses the nquad `_:a :p1 _:z .` to calculate the hash `24da9a4406b4e66dffa10ad3d4d6dddc388fbf193bb124e865158ef419893957`.

For `e1`, the Hash First Degree Quads algorithm uses the nquads `_:z :p1 _:a .` and `_:a :p2 "Foo" .` to calculate the hash `a994e40b576809985bc0f389308cd9d552fd7c89d028c163848a6b2d33a8583a`.

For `e2`, the Hash First Degree Quads algorithm uses the nquad `_:a :p1 _:z .` to calculate the hash `24da9a4406b4e66dffa10ad3d4d6dddc388fbf193bb124e865158ef419893957`.

For `e3`, the Hash First Degree Quads algorithm uses the nquads `_:z :p1 _:a .` and `_:a :p2 "Foo" .` to calculate the hash `a994e40b576809985bc0f389308cd9d552fd7c89d028c163848a6b2d33a8583a`.

hash	***blank node(s)***
`24da9a4406b4e66dffa10ad3d4d6dddc388fbf193bb124e865158ef419893957`	`e0` and `e2`
`a994e40b576809985bc0f389308cd9d552fd7c89d028c163848a6b2d33a8583a`	`e1` and `e3`

Table 20 Hash to blank nodes map shared hashes

Step 4 would create canonical identifiers for each blank node which has a unique hash, but no blank node has a unique hash.

Step 5 is run on `e0`, `e1`, `e2`, and `e3`, separately, which share hash values. each use separate instances of a *temporary issuer*, to create the *hash path list* composed of the hash result and temporary identifier mappings from Hash N-Degree Quads algorithm for each of these blank nodes/

The Hash N-Degree Quads algorithm is first called for hash `24da9a4406b4e66dffa10ad3d4d6dddc388fbf193bb124e865158ef419893957` and blank node `e0` related to the nquad `_:e0 :p1 _:e1 .`.

blank node	*temporary identifier*
`e0`	`b0`

Step 3 calls the Hash Related Blank Node algorithm for each position in this quad:

related	*quads*	*position*	*hash*

`e3`	`z :p1 a .`
	`a :p2 "Foo" .`
		`o`	`a994e40b57...2d33a8583a`

Table 21 Hash for `e2` related to `e0`

which results in the following H_n:

related hash	*blank node list*
`a994e40b576809985bc0f389308cd9d552fd7c89d028c163848a6b2d33a8583a`	`[ e3 ]`

Table 22 Blank node list for hash

Step 5 iterates over each *related hash* which is a key in H_n, which can have one or more related blank nodes.

At Step 5.4.5, the *recursion list* is `[ e3 ]`, so the Hash N-Degree Quads algorithm is called recursively resulting in the following:

temporary issuer mappings	*original identifier*	*temporary identifier*
	`e2`	`b0`
	`e3`	`b1`
hash	`c484f98e6cbf9e21f287433c8b1caa7f1486fd61d84ab220a494bf8184751b8c`	

Table 23 Result for `e3`

Back in step 5.4.5.4, the *path* and *issuer copy* are: `_:b1_:b1<c484f98e6cbf9e21f287433c8b1caa7f1486fd61d84ab220a494bf8184751b8c>` and `{e2: b0, e3: b1}`.

This results in the *chosen path* `_:b1_:b1<c484f98e6cbf9e21f287433c8b1caa7f1486fd61d84ab220a494bf8184751b8c>` and *data to hash* `3d96946f27fc34a78e8d067135a1cb1b77083aebc4b2c6cbdc536f067242686c_:b1_:b1<c484f98e6cbf9e21f287433c8b1caa7f1486fd61d84ab220a494bf8184751b8c>`

temporary issuer mappings	***original identifier***	***temporary identifier***
	`e2`	`b0`
	`e3`	`b1`
hash	`39d609fcd8236b74c70744f492cd2baaf0a55765b380ff9e0811ce23e2f409d7`	

Table 24 *Result for* `e2`

The Hash N-Degree Quads algorithm is next called for the same hash `24da9a4406b4e66dffa10ad3d4d6dddc388fbf193bb124e865158ef419893957` but this time with blank node `e2` related to the nquad `_:e2 :p1 _:e3 .`.

blank node	***temporary identifier***
`e2`	`b0`

Step 3 calls the Hash Related Blank Node algorithm for each position in this quad:

related	***quads***	***position***	***hash***

`e3`	`z :p1 a .`
	`a :p2 "Foo" .`
		`o`	`3d96946f27...184751b8c`

Table 25 *Hash for* `e2` *related to* `e0`

which results in the following H_n:

related hash	***blank node list***
`3d96946f27fc34a78e8d067135a1cb1b77083aebc4b2c6cbdc536f067242686c`	`[ e3 ]`

Table 26 *Blank node list for hash*

Step 5 iterates over each *related hash* which is a key in H_n, which can have one or more related blank nodes.

	original identifier	***temporary identifier***
temporary issuer mappings	`e0`	`b0`
hash	`fbc300de5afafd97a4b9ee1e72b57754dcdcb7ebb724789ac6a94a5b82a48d30`	

Table 27 Result for `e0`

Step 5.3 back in the Canonicalization Algorithm creates canonical identifiers for the temporary identifiers just issued:

blank node	***canonical identifier***
`e0`	`c14n0`
`e1`	`c14n1`
`e2`	`c14n2`
`e3`	`c14n3`

Next, back in step 5.1, now using hash `a994e40b576809985bc0f389308cd9d552fd7c89d028c163848a6b2d33a8583a`, canonical identifiers have already been created for the two blank nodes `e1` and `e3`, so no further processing is necessary.

Step 6 ends with the issued identifiers map containing the following mappings:

blank node	***canonical identifier***
`e0`	`c14n0`
`e1`	`c14n1`
`e2`	`c14n2`
`e3`	`c14n3`

§ 9.2 Double Circle

This example illustrates another complicated example of nodes that are doubly connected in opposite directions.

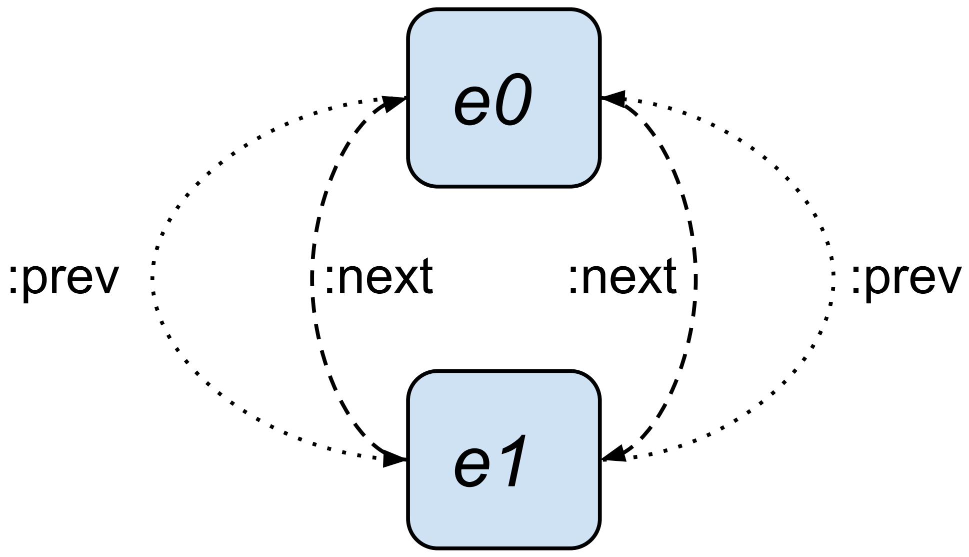

Figure 8 An illustration of a graph back and forth links to nodes.
Image available in SVG .

```
_:e0 :next _:e1 .
_:e0 :prev _:e1 .
_:e1 :next _:e0 .
_:e1 :prev _:e0 .
```

The example is not explored in detail, but the execution log found here shows examples of more complicated pathways through the algorithm

§ 9.3 Dataset with Blank Node Named Graph

This example illustrates an example of a dataset, where one graph is named using a blank node, which is also the object of a triple in the default graph.

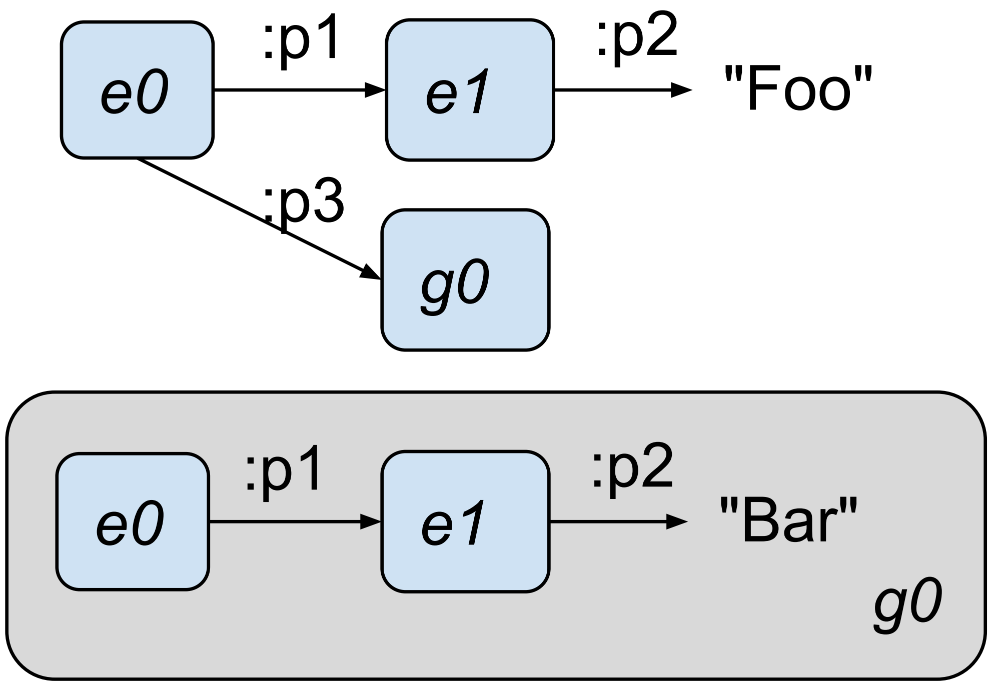

Figure 9 An illustration of a dataset containing a graph named with a blank node. Image available in SVG.

```
_:e0 :p1 _:e1 .
_:e1 :p2 "Foo" .
_:e1 :p3 _:g0 .
_:e0 :p1 _:e1 _:g0 .
_:e1 :p2 "Bar" _:g0 .
```

The following is a summary of the more detailed execution log found here.

EXAMPLE 10: Dataset Blank Node Graph

Step 2 of the Canonicalization Algorithm is called three times, with each blank node (`e0`, `e1`, and `g0`) to populate blank node to quads map:

blank node	Q

`e0`	`_:e0 :p1 _:e1 .`
	`_:e0 :p1 _:e1 _:g0 .`

`e1`	`_:e0 :p1 _:e1 .`
	`_:e1 :p2 "Foo" .`
	`_:e1 :p3 _:g0 .`
	`_:e0 :p1 _:e1 _:g0 .`
	`_:e1 :p2 "Bar" _:g0 .`

`g0`	`_:e1 :p3 _:g0 .`
	`_:e0 :p1 _:e1 _:g0 .`
	`_:e1 :p2 "Bar" _:g0 .`

Table 28 Blank node to quads map for dataset with blank node named graph

Step 3 generates the first-degree hash for each blank node.

For `e0`, the Hash First Degree Quads algorithm uses the nquads `_:a :p1 _:z .` and `_:a :p1 _:z _:z .`to calculate the hash `39fa92e7a9ea0b32e3e08aedebc1cbbcd9ba9945a18f2594677598fcfb517345`.

For `e1`, the Hash First Degree Quads algorithm uses the nquads `_:z :p1 _:a .`, `_:a :p2 "Foo" .`, `_:a :p3 _:z .`, `_:z :p1 _:a _:z .`, and `_:a :p2 "Bar" _:z .` to calculate the hash `5274652aa97d22f9f2998d1837ffaf9ad820fb813574adc4a403f105021e48bf`.

For `g0`, the Hash First Degree Quads algorithm uses the nquads `_:z :p3 _:a .`, `_:z :p1 _:z _:a .`, and `_:z :p2 "Bar" _:a .` to calculate the hash `efb0096125247801199082c6f213cda5377d8ddac20a83eaf6c7d88335c12bf4`.

hash	blank node(s)
`39fa92e7a9ea0b32e3e08aedebc1cbbcd9ba9945a18f2594677598fcfb517345`	`e0`
`5274652aa97d22f9f2998d1837ffaf9ad820fb813574adc4a403f105021e48bf`	`e1`
`efb0096125247801199082c6f213cda5377d8ddac20a83eaf6c7d88335c12bf4`	`g0`

Table 29 Hash to blank nodes map shared hashes

Step 4 creates canonical identifiers for each blank node which has a unique hash:

blank node	***canonical identifier***
`e0`	`c14n0`
`e1`	`c14n1`
`g0`	`c14n2`

Table 30 Canonical identifiers for dataset with blank node named graph

Step 5 has no effect, as there are no remaining blank nodes without canonical identifiers.

Step 6 ends with the canonical issuers containing the following mappings:

blank node	***canonical identifier***
`e0`	`c14n0`
`e1`	`c14n1`
`g0`	`c14n2`

§ A. A Canonical form of N-Quads

This section defines a canonical form of N-Quads which has a completely specified layout. The grammar for the language remains unchanged.

Canonical N-Quads updates and extends Canonical N-Triples in [N-TRIPLES] to include `graphLabel`.

While the N-Quads syntax [N-QUADS] allows choices for the representation and layout of RDF data, the canonical form of N-Quads provides a unique syntactic representation of any quad. Each code point can be represented by only one of `UCHAR`, `ECHAR`, or unencoded character, where the relevant production allows for a choice in representation. Each quad is represented entirely on a single line with specified white space.

Canonical N-Quads has the following additional constraints on layout:

- White space *MUST NOT* be used except after `subject`, `predicate`, `object`, and `graphLabel`, each of which *MUST* be a single space (code point `U+0020`).

- Literals with the datatype `http://www.w3.org/2001/XMLSchema#string` *MUST NOT* use the datatype IRI part of the literal, and are represented using only STRING_LITERAL_QUOTE.
- `HEX` *MUST* use only digits (`[0-9]`) and uppercase letters (`[A-F]`).
- Within STRING_LITERAL_QUOTE:
  - Characters `BS` (backspace, code point `U+0008`), `HT` (horizontal tab, code point `U+0009`), `LF` (line feed, code point `U+000A`), `FF` (form feed, code point `U+000C`), `CR` (carriage return, code point `U+000D`), `"` (quotation mark, code point `U+0022`), and `\` (backslash, code point `U+005C`) *MUST* be encoded using `ECHAR`.
  - Characters in the range from `U+0000` to `U+0007`, `VT` (vertical tab, code point `U+000B`), characters in the range from `U+000E` to `U+001F`, `DEL` (delete, code point `U+007F`), and characters not matching the Char production from [XML11] *MUST* be represented by `UCHAR` using a lowercase `\u` with 4 `HEX`es.
  - All characters not required to be represented by `ECHAR` or `UCHAR` *MUST* be represented by their native [UNICODE] representation.
- The token `EOL` *MUST* be a single `LF` (line feed, code point `U+000A`).
- The final `EOL` *MUST* be provided.

§ B. URDNA2015

This section is non-normative.

RDF Dataset Canonicalization [CCG-RDC-FINAL] describes "Universal RDF Dataset Normalization Algorithm 2015" (***URDNA2015***), essentially the same algorithm as RDFC-1.0, and generally implementations implementing URDNA2015 should be compatible with this specification. The minor change is in the canonical n-quads form where some control characters were previously represented without escaping. The version of the algorithm defined in A. A Canonical form of N-Quads clarifies the representation of simple literals and the characters within STRING_LITERAL_QUOTE that are encoded using `ECHAR`.

§ C. URGNA2012

This section is non-normative.

A previous version of this algorithm has light deployment. For purposes of identification, the algorithm is called the "Universal RDF Graph Canonicalization Algorithm 2012" (***URGNA2012***), and differs from the stated algorithm in the following ways:

- In 4.6 Hash First Degree Quads, if any blank node was used in the graph name position in the quad, then the value was serialized using the special blank node identifier, `g`, instead of `z`.
- In 4.7 Hash Related Blank Node, value of the predicate was not delimited by `<` and `>`; there were no delimiters.
- In 4.8 Hash N-Degree Quads, the *position* parameter passed to the Hash Related Blank Node algorithm was instead modeled as a *direction* parameter, where it could have the value `p`, for property, when the related blank node was a subject and the value `r`, for reverse or reference, when the related blank node was an object. Since URGNA2012 only canonicalized graphs, not datasets, there was no use of the graph name position.
- In 4.8 Hash N-Degree Quads, building the H_n was done as follows:

    1) For each *quad* in *quads*:

    1.1) If the *quad*'s subject is a blank node that does not match *identifier*, set *hash* to the result of the Hash Related Blank Node algorithm, passing the blank node identifier for subject as *related*, *quad*, *issuer*, and `p` as *position*.

    1.2) Otherwise, if *quad*'s object is a blank node that does not match *identifier*, set *hash* to the result of the Hash Related Blank Node algorithm, passing the blank node identifier for object as *related*, *quad*, *issuer*, and `r` as *position*.

    1.3) Otherwise, continue to the next quad.

    1.4) Add a mapping of *hash* to the blank node identifier for the component that matched (subject or object) to H_n, adding an entry as necessary.

§ D. Index

§ D.1 Terms defined by this specification

§ D.2 Terms defined by reference

[BCP47] defines the following:

- formatting conventions

[INFRA] defines the following:

- boolean type
- entry (for `map`)
- key (for `map`)
- list
- map
- string

[N-QUADS] defines the following:

- C. N-Quads Internet Media Type, File Extension and Macintosh File Type
- ECHAR
- EOL
- graphLabel
- HEX
- literal
- STRING_LITERAL_QUOTE
- UCHAR

[N-TRIPLES] defines the following:

- Canonical N-Triples

[RDF11-CONCEPTS] defines the following:

- blank node identifiers
- blank nodes
- Blank Nodes
- default graph
- graph name
- IRIs
- isomorphic datasets
- language tags
- language-tagged strings
- lexical form
- literal
- literal term equality
- literal value
- object type
- predicate
- RDF datasets
- RDF graph
- RDF source
- RDF triple
- Section 3.3
- simple literals
- Skolem IRIs
- subject

[RDF11-MT] defines the following:

- RDF Collections

[VC-DATA-INTEGRITY] defines the following:

- Selective Disclosure

[XML11] defines the following:

- Char

[XPATH-FUNCTIONS] defines the following:

- Unicode code point order

§ E. Changes since the First Public Working Draft of 24 November 2022

This section is non-normative.

- The algorithm, and the examples, have been changed to systematically use the *`xyz`* format for blank node identifiers, instead of `_:xyz`. See Issue 46 for the discussion.
- 4.6 Hash First Degree Quads was simplified to remove the `simple` flag, which was unused in existing implementations. The original design of the algorithm was to use the assigned canonical blank node identifier, if available, instead of `_:a` or `_:z`, similar to how it is used in the related hash algorithm, but this text never made it into the spec before implementations moved forward.

Therefore, the hashes never change, making the loop based on the `simple` flag that calls this algorithm unnecessary. See Issue 23 for the discussion.

- Add definition for canonical n-quads form. Eventually, this should be a citation from [N-Quads], when it is updated. Canonical n-quads form is used in 4.6 Hash First Degree Quads.
- Removed issue marker for Issue 15 in 4.4 Canonicalization Algorithm, adding a note that literal components of quads are not normalized, and two literals with different syntactic representations remain distinct resources.
- Changed the way Blank Node identifiers are described (see Issue 46), generally omitting the leading `_:` which is a serialization artifact. This is still required in the algorithms, but the distinction between what is an identifier, and the serialization form is clarified.
- Changed the name of the algorithm from URDNA2015 to RDFC-1.0.
- Changed the term **normalized dataset** to canonicalized dataset, which is composed of the input dataset, input blank node identifier map, and issued identifiers map.

§ F. Changes since the Candidate Recommentation Snapshot of 31 October 2023

This section is non-normative.

- Clarified that detecting a poison dataset will result in an exception and early termination of the Canonicalization Algorithm.

§ G. Acknowledgements

This section is non-normative.

The editors would like to thank Jeremy Carroll for his work on the graph canonicalization problem, Andy Seaborne and Gavin Carothers for providing valuable feedback and testing input for the algorithm defined in this specification, Sir Tim Berners-Lee for his thoughts on graph canonicalization over the years, Jesús Arias Fisteus for his work on a similar algorithm, and Aiden Hogan, whose publication [Hogan-Canonical-RDF] provided an important contemporary analysis of the canonicalization problem and served as an independent justification of the development of RDFC-1.0.

The editors would also like to thank the chairs of the Working Group, Phil Archer and Markus Sabadello, and specific members of the Working Group whose active contributions were critical in completing this work: Pierre-Antoine Champin, Ivan Herman, David Lehn, Kazue Sako, Manu Sporny, and Ted Thibodeau Jr.

Members of the RDF Dataset Canonicalization and Hash Working Group Group included Ahamed Azeem, Ahmad Alobaid, Andy Seaborne, Benjamin Goering, Benjamin Young, Brent Zundel, Damien Graux, Dan Brickley, Dan Yamamoto, Daniel Pape, Dave Longley, David Lehn, Duy-Tung Luu, Gregg Kellogg, Ivan Herman, Jean-Yves Rossi, Jennifer Meier, Jesse Wright, Kazue Sako, Leonard Rosenthol, Mahmoud Alkhraishi, Manu Sporny, Markus Sabadello, Michael Prorock, Phil Archer, Pierre-Antoine Champin, Sebastian Crane, Ted Thibodeau Jr, Timothée Haudebourg, and Tobias Kuhn.

This specification is based on work done in the W3C Credentials Community Group published as [CCG-RDC-FINAL]. Contributors to the Community Group Final Report include: Blake Regalia, Dave Longley, David Lehn, David Lozano Jarque, Gregg Kellogg, Manu Sporny, Markus Sabadello, Matt Collier, and Sebastian Schmittner.

Portions of the work on this specification have been funded by the European Union's StandICT.eu 2023 program under sub-grantee contract numbers No. 08/12 and 09/25. The content of this specification does not necessarily reflect the position or the policy of the European Union and no official endorsement should be inferred.

Portions of the work on this specification have also been funded by the U.S. Department of Homeland Security's Silicon Valley Innovation Program under contracts 70RSAT21T00000020 and 70RSAT23T00000006. The content of this specification does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

The Working Group acknowledges that the success of this specification is dependent on a long history of work performed over multiple decades in both academia and industry. We thank the individuals who iterated on the science which led to the completion of this specification. A partial list of these papers is found below, to the best of the Working Group's recollection. Omission from this list, whether intentional or unintentional, is not meant to imply that such an unlisted paper was not similarly important to the development of this work.

- Berners-Lee, T., Connolly, D.: *Delta: an ontology for the distribution of differences between RDF graphs* W3C Unofficial (2001). https://www.w3.org/DesignIssues/Diff [DesignIssues-Diff].
- Carroll, J.J.: *Signing RDF Graphs*. In: Fensel, D., Sycara, K., and Mylopoulos, J. (eds.) The Semantic Web - ISWC 2003. pp. 369–384. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39718-2_24 [HPL-2003-142].

- Sayers, C., Karp, A.: Computing the digest of an RDF graph, Tech. Rep. HPL-2003-235 (R. 1), Hewlett Packard Laboratories (2004). https://www.hpl.hp.com/techreports/2003/HPL-2003-235R1.html.
- Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. Journal of Web Semantics. 3, 247–267 (2005). https://doi.org/10.1016/j.websem.2005.09.001.
- Tummarello, G., Morbidoni, C., Bachmann-Gmür, R., Erling, O.: RDFSync: Efficient Remote Synchronization of RDF Models. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., and Cudré-Mauroux, P. (eds.) The Semantic Web. pp. 537–551. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_39.
- Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: a document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies. 3, 37–52 (2008). https://doi.org/10.1504/IJMSO.2008.021204.
- Fisteus, J.A., Fernández García, N., Sánchez Fernández, L., Delgado Kloos, C.: Hashing and canonicalizing Notation 3 graphs. Journal of Computer and System Sciences. 76, 663–685 (2010). https://doi.org/10.1016/j.jcss.2010.01.003.
- Kasten, A., Scherp, A., Schauß, P.: *A Framework for Iterative Signing of Graph Data on the Web*. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., and Tordai, A. (eds.) The Semantic Web: Trends and Challenges. pp. 146–160. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07443-6_11 [eswc2014Kasten].
- Hogan, A.: *Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes*. ACM Trans. Web. 11, 22:1-22:62 (2017). https://doi.org/10.1145/3068333 [Hogan-Canonical-RDF].
- Arnold, R., Longley, D.: RDF Dataset Normalization. Report submitted to the W3C Credentials Community Group mailing list (2020). https://lists.w3.org/Archives/Public/public-credentials/2021Mar/att-0220/RDFDatasetCanonicalization-2020-10-09.pdf.

§ H. References

§ H.1 Normative references

[FIPS-180-4]
FIPS PUB 180-4: Secure Hash Standard (SHS). U.S. Department of Commerce/National Institute of Standards and Technology. August 2015. National Standard. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[INFRA]
Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[N-Quads]
RDF 1.1 N-Quads. Gavin Carothers. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/n-quads/

[N-TRIPLES]
RDF 1.1 N-Triples. Gavin Carothers; Andy Seaborne. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/n-triples/

[RDF11-CONCEPTS]
RDF 1.1 Concepts and Abstract Syntax. Richard Cyganiak; David Wood; Markus Lanthaler. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-concepts/

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc2119

[RFC3987]
Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF. January 2005. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc3987

[RFC8174]
Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[Turtle]
RDF 1.1 Turtle. Eric Prud'hommeaux; Gavin Carothers. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/turtle/

[UNICODE]
The Unicode Standard. Unicode Consortium. URL: https://www.unicode.org/versions/latest/

[XML11]
Extensible Markup Language (XML) 1.1 (Second Edition). Tim Bray; Jean Paoli; Michael Sperberg-McQueen; Eve Maler; François Yergeau; John Cowan et al. W3C. 16 August 2006. W3C Recommendation. URL: https://www.w3.org/TR/xml11/

[XPATH-FUNCTIONS]
XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). Ashok Malhotra; Jim Melton; Norman Walsh; Michael Kay. W3C. 14 December 2010. W3C Recommendation. URL: https://www.w3.org/TR/xpath-functions/

§ H.2 Informative references

[BCP47]
Tags for Identifying Languages. A. Phillips, Ed.; M. Davis, Ed.. IETF. September 2009. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc5646

[CCG-RDC-FINAL]
RDF Dataset Canonicalization. Dave Longley. W3C. October 9, 2022. CG-FINAL. URL: https://www.w3.org/community/reports/credentials/CG-FINAL-rdf-dataset-canonicalization-20221009/

[DesignIssues-Diff]
Delta: an ontology for the distribution of differences between RDF graphs. Tim Berners-Leee. W3C. September 25, 2015. unofficial. URL: https://www.w3.org/DesignIssues/Diff

[eswc2014Kasten]
A Framework for Iterative Signing of Graph Data on the Web. Andreas Kasten; Ansgar Scherp; Peter Schauß . ISWC 2014. 2014. unofficial. URL: https://doi.org/10.1007/978-3-319-07443-6_11

[Hogan-Canonical-RDF]
Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes. Aiden Hogan. ACM. November 2017. ACM Trans. Web 11, 4, Article 22. URL: https://aidanhogan.com/docs/rdf-canonicalisation.pdf

[HPL-2003-142]
Signing RDF Graphs. Jeremy J. Carroll. HP Laboratories Bristol. July 23, 2003. unofficial. URL: https://web.archive.org/web/20230129125726/https://www.hpl.hp.com/techreports/2003/HPL-2003-142.pdf

[RCH-EXPLAINER]
: *RDF Dataset Canonicalization and Hash Working Group — Explainer and Use Cases*. Phil Archer. W3C. 19 October 2023. W3C Working Group Note. URL: https://www.w3.org/TR/rch-explainer/

[RDF11-MT]
: *RDF 1.1 Semantics*. Patrick Hayes; Peter Patel-Schneider. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-mt/

[VC-DATA-INTEGRITY]
: *Verifiable Credential Data Integrity 1.0*. Manu Sporny; Dave Longley; Greg Bernstein; Dmitri Zagidulin; Sebastian Crane. W3C. 28 April 2024. W3C Candidate Recommendation. URL: https://www.w3.org/TR/vc-data-integrity/

[YAML]
: *YAML Ain't Markup Language (YAML™) Version 1.2*. Oren Ben-Kiki; Clark Evans; Ingy döt Net. 1 October 2009. URL: http://yaml.org/spec/1.2/spec.html

↥