Property Graph Exchange Format (PG)

Version 1.0.0 (pre-release draft)

Hirokazu Chiba 0000-0003-4062-8903[1], Jakob Voß 0000-0002-7613-4123[2]

[1]Database Center for Life Science (DBCLS)
[2]Verbundzentrale des GBV (VZG)

This document specifies a common data model of labeled property graphs, a syntax to write property graphs in a compact textual form, and serialization formats of property graphs in JSON and in newline-delimited JSON.

Table of contents

1 Introduction

1.1 Motivation

Property Graphs (also known as **Labeled Property Graphs**) are used as an abstract data structure in graph databases and related applications.

Implementations of property graphs slightly differ in support of data types, restrictions on labels etc. The definition of property graphs used in this specification is aimed to be a superset of property graph models of common graph databases and formats. The model and its serializations have first been proposed by Hirokazu Chiba, Ryota Yamanaka, and Shota Matsumoto (2019, 2022) and revised into this specification together with Jakob Voß.

1.2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 (RFC 2119 and RFC 8174) when, and only when, they appear in all capitals, as shown here.

1.3 Robustness principle

Applications MAY automatically convert documents not fully conforming to the specification of PG-JSON and/or PG-JSONL to a valid form, for instance by:

- creation of implicit nodes for node identifiers referenced in edges
- addition of missing empty fields `labels` and/or `properties`
- removal or mapping of invalid property values such as `null` and JSON objects
- mapping of numeric node identifiers and edge identifiers to strings
- removal of additional fields not defined in this specification

2 Data Model

A **property graph** consists of **nodes** and **edges** between these nodes. Each node has a unique **node identifier**. Each edge is either directed or undirected and can have an optional **edge identifier**. Each of the nodes and edges can have **properties** and **labels**. Properties are mappings from **keys** to non-empty lists of **values**. Node identifiers, labels, and keys are non-empty Unicode strings. A value is a Unicode string, a boolean value, or a number as defined by RFC 8259.

Extended graph features not being part of this data model include graph attributes, hierarchies, hyper-edges and semantics of individual labels and property keys.

3 PG Format

A **PG Format** document allows writing down a property graph in a compact textual form. A PG Format document is a Unicode string that conforms to grammar and rules defined in this specification.

3.1 Basic structure

A PG Format document encodes a property graph as Unicode string. The document MUST be encoded in UTF-8 (RFC 3629). Unicode codepoints can also be given by escape sequences in quoted strings.

The document consists of a sequence of **statements**, each defining a node or an edge, or being empty. Statements are separated from each other by a line break. Optional spaces and/or a comment at the end of a statement are ignored.

3.2 Identifiers

An **identifier** is a string used to uniquely identify a node, an edge, a label, or the name of a property. An identifier can be given as a quoted string or as an unquoted identifier.

An **unquoted identifier** is a non-empty string not including control codes `U+0000` to `U+0020` (tabulator, line breaks, space…), nor any of the characters "`<`" (`U+003C`), "`>`" (`U+003E`), "`"`" (`U+0022`), "`{`" (`U+007B`), "`}`" (`U+007D`), "`|`" (`U+007C`), "`\`" (`U+005C`), "`^`" (`U+005E`), and "`` ` ``" (`U+0060`). An unquoted identifier MUST NOT start with a colon (`U+003A`), comma (`U+002C`), minus (`U+002D`), hash ("`#`"), apostrophe ("`'`"), or quotation mark ("`"`"). Colon, hash, comma, and apostrophe are allowed in an unquoted identifier after its first character.[1]

Example 1: Several unquoted identifiers

```
abc
42

dc:title
http://example.org/?a=-&c=0#x
~',-:
```

3.3 Nodes

A **node** consists of the following elements, given in this order and separated by delimiting whitespace:

- a REQUIRED identifier
- an OPTIONAL list of labels
- an OPTIONAL list of properties

[1] This definition is equivalent to the definition of IRI references in SPARQL and in Turtle excluding empty strings, escape sequences and forbidding some start characters.

Example 2: Some node statements

```
id :label key:value
42 :answer
"node id with spaces"
```

3.3.1 Node merging

A node can be defined with multiple statements having the same node identifier: a node is **merged** with an existing node by appending labels and property values.

Example 3: One node defined by multiple statements

```
a :x k:1 m:true
a :y k:2
```

Example 4: Same node defined by one statement

```
a :x :y k:1,2 m:true
```

3.3.2 Implicit nodes

Nodes can also be defined implicitly as part of an edge: node identifiers referenced in edges imply the existence of nodes with these identifiers.

Example 5: Simple graph with two nodes and one edge

```
a -> b
```

Example 6: Same graph with explicit node statements

```
a
b
a -> b
```

3.4 Edges

An **edge** consists of the following elements, given in this order and separated by delimiting whitespace:

- an OPTIONAL **edge identifier**
- a REQUIRED source node identifier
- a REQUIRED **direction**
- a REQUIRED target node identifier
- an OPTIONAL list of labels
- an OPTIONAL list of properties

Example 7: Some edge statements

```
a -> b
a -- b key:value
1: a -> b :label key:value
```

Example 8: No edge statements

```
a--b    # a node with node identifier "a--b"
a->b    # syntax error
```

3.4.1 Edge identifiers

An edge identifier is an identifier as the first element of an edge statement, directly followed by a colon (`U+003A`).

Example 9: Graph with two equivalent edges, differentiated by edge identifiers

```
1: a -> b :follows since:2024
"x": a -> b :follows since:2024
```

Colons are not forbidden in edge identifiers:

Example 10: Edge identifiers with colon

```
x::  a -> b  # edge identifier "x:"
":": a -> b  # edge identifier ":"
```

Edge identifiers MUST NOT be repeated.

Example 11: The second statement is invalid because of repeated edge identifier

```
1: a -> b :follows
1: a -> b since:2024
```

No space is allowed between the edge identifier and its colon:

Example 12: Invalid statement

```
1 : a -> b
```

3.4.2 Edge directions

The direction element of an edge is either the character sequence `->` for a directed edge or the character sequence `--` for an undirected edge.

3.4.3 Loops

Edges can connect a node to itself.

Example 13: Directed and undirected loop

```
a -> a
a -- a
```

3.4.4 Multi-edges

The Property Graph Data Model allows for multiple edges between the same node.

Example 14: Graph with two indistinguishable edges

```
a -> b :follows since:2024
a -> b :follows since:2024
```

Edge identifiers can be used to identify and reference individual multi-edges.

3.5 Labels

A label is an identifier following a colon (`U+003A`). Spaces between colon and label identifier are OPTIONAL but NOT RECOMMENDED.

Labels of a node or an edge are unique: repeated labels are ignored. Applications SHOULD preserve the order of labels of a node or an edge.

Example 15: Repeated labels on same node or edge are ignored

```
a :label1 :label2 :label1   # label1 is repeated
a :label1 :label2           # equivalent statement
a : label1 : label2         # equivalent statement
```

Colons are not forbidden in labels:

Example 16: Labels with colons

```
a :b:c                      # label "b:c"
a :http://example.org/      # label "http://example.org/"
```

3.6 Properties

A **property** consists of the following elements, given in this order:

- a REQUIRED **property key**, being an identifier
- a colon (`U+003A`)
- a non-empty list of **property values**, separated by comma (`U+002C`)

Each property value MAY be preceded and followed by delimiting whitespace. If the property key is an unquoted identifier and no delimiting whitespace is given before the first value, then the property key MUST NOT contain a colon.

Example 17: Invalid property

```
node key            # delimiting whitespace not allowed before colon
  : value
```

Example 18: Property with optional spaces and/or whitespace

```
node key: value      # spaces before value
node key:value       # short form
node "key":value     # key can be quoted string
node key:            # delimiting whitespace between colon and value
  value

node key: 1,2        # short form of a list
node key: 1          # delimiting whitespace...
  ,                  # ...after value 1 and before value 2
  2

node a:b:c           # property key "a" with value "b:c"
node a:b: c          # property key "a:b" with value "c"
```

3.6.1 Property values

A **property value** is one of

- a **number value**, given as defined in section 6 of RFC 8259. As mentioned there, implementations MAY set limits on the range and precision of numbers and double precision (IEEE754) is the most likely common limit.
- a **boolean value**, given as one of the literal character sequences `true` and `false`
- a **string value**, given as one of
  - a quoted string
  - an unquoted identifier not including a comma

The data type of a property value in PG Format is either string, or number, or boolean.[2] Applications MAY internally map these types to other type systems. Values of the same property are allowed to have different data types.

Example 19: Property values

```
node n: 1,-1,2e+3          # numbers
     b: true, false        # boolean values
     s: hello,"true",""    # strings
```

3.6.2 Property merging

Value lists of properties of the same property key are concatenated. Value lists are no sets: the same value can be included multiple times.

Example 20: Three nodes with same properties

```
a x:1,2,3       # property values given as list
b x:1 x:2,3     # property values given as two lists
c x:1           # property values given...
c x:2 x:3       # ...in two node statements
```

[2]This is identical to scalar JSON values (string, number, boolean) and every serialized JSON scalar is a valid property value in PG Format.

3.7 Quoted Strings

A quoted string starts with an apostrophe (“`'`”) or quotation mark (“`"`”) and ends with the same character. In between, all Unicode characters are allowed, except for the characters that MUST be escaped:

- apostrophe, when the string is quoted with apostrophe
- quotation mark, when the string is quoted with a quotation mark
- reverse solidus (`\` `U+005C`)
- control characters `U+0000` through `U+001F` except line feed (`U+000A`), carriage return (`U+000D`), and tabular (`U+0009`)

All characters can be escaped as defined by JSON specification (RFC 8259, section 7) with the addition of the two-character escape sequence `\'` to escape an apostrophe. Quoted strings in PG Format further differ from JSON by string quoting with apostrophe in addition to quotation mark and by allowing unquoted line feed, carriage return, and tabular.

Example 21: The same string given in multiple quoted forms

```
"hello,\nworld"
'hello,\u000Aworld'
"hello,
world"
```

Example 22: Invalid string escape sequences

```
"h\ello\u21"
```

3.8 Whitespace

A **line break** is either a line feed (`U+000A`) or a carriage return (`U+000D`) optionally followed by a line feed.

Spaces are a non-empty sequence of space (`U+0020`) and/or tabular (`U+0009`).

A **comment** begins with a hash (`#` = `U+0023`) and it ends before the next line break or at the end of the document.

Delimiting whitespace separates elements of a statement. Delimiting whitespace consists of an optional sequence of spaces, comment, and/or line breaks and it ends with spaces. The inclusion of line breaks in delimiting whitespace is also called *line folding*.

Example 23: Line folding

```
a :x  # node id and label
  # this and the following line are empty

  :y  # another label of the same node at continuation line
```

3.9 Grammar

The formal grammar of PG Format is specified in EBNF Notation used in the specification of XML, with the addition of negative lookahead operator (`!A B` matches any expression `B` that does not start with expression `A`) and the terminal symbol `END` denoting the end of a document.

```
/* 3.1 Basic Structure */
PG              ::= ( Statement? Empty LineBreak )* Statement? Empty
Statement       ::= Edge | Node

/* 3.2 Identifiers */
Identifier      ::= QuotedId | UnquotedStart UnquotedChar*
UnquotedChar    ::= [^#x00-#x20<>"{}|^`\]
UnquotedStart   ::= !['":#,-] UnquotedChar
Node            ::= Identifier Labels Properties

/* 3.3 Nodes & 3.4 Edges */
Edge            ::= ( EdgeIdentifier )? EdgeNodes Labels Properties
EdgeIdentifier  ::= QuotedKey DWS | UnquotedKey DWS
EdgeNodes       ::= Identifier DWS Direction DWS Identifier
Direction       ::= "--" | "->"

/* 3.5 Labels */
Labels          ::= ( DWS ":" Label )*
Label           ::= ":" Spaces? Identifier

/* 3.6 Properties */
Properties      ::= ( DWS Property )*
Property        ::= Key ValueList
Key             ::= QuotedKey | UnquotedKey DWS | UnquotedKeyCol
QuotedKey       ::= QuotedId ":"
UnquotedKey     ::= UnquotedStart ( ( !":" UnquotedChar )* ":" )+
UnquotedKeyCol  ::= UnquotedStart ( !":" UnquotedChar )* ":"
ValueList       ::= DWS? Value ( DWS? "," DWS? Value )*

/* 3.6.1 Property Values */
Value           ::= Number | Boolean | QuotedString | UnquotedValue
Number          ::= "-"? ("0" | [1-9] [0-9]*) ( "." [0-9]+ )? ([eE] [+-]? [0-9]+)?
Boolean         ::= "true" | "false"
UnquotedValue   ::= UnquotedStart (!"," UnquotedChar)*
QuotedString    ::= "'" SingleQuoted* "'" | '"' DoubleQuoted* '"'

/* 3.7 Quoted Strings */
QuotedId        ::= "'" SingleQuoted+ "'" | '"' DoubleQuoted+ '"'
SingleQuoted    ::= Unescaped | '"' | Escaped
DoubleQuoted    ::= Unescaped | "'" | Escaped
Unescaped       ::= [^#x00-#x08#x0B#x0C#x0E-#x1F"'\]+
Escaped         ::= "\" ( '"' | "'" | "\" | "/" | [bfnrt] | "u" Codepoint )
Codepoint       ::= [0-9a-fA-Z] [0-9a-fA-Z] [0-9a-fA-Z] [0-9a-fA-Z]

/* 3.8 Whitespace */
Spaces          ::= [#x20#x09]+
LineBreak       ::= [#x0A] | [#x0D] [#x0A]?
Comment         ::= "#" [^#x0D#x0A]*
Empty           ::= Spaces? Comment?
DWS             ::= (Empty LineBreak)* Spaces
```

4 PG-JSON

A **PG-JSON** document serializes a property graph in JSON. A PG-JSON document is a JSON document (RFC 8259) with a JSON object with exactely two fields:

- `nodes` an array of nodes
- `edges` an array of edges

Each node is a JSON object with exactely three fields:

- `id` the node identifier, being a non-empty string. Node identifiers MUST be unique per PG-JSON document.
- `labels` an array of labels, each being a non-empty string. Labels MUST be unique per node. The array SHOULD be sorted by unicode codepoints.
- `properties` a JSON object mapping non-empty strings as property keys to non-empty arrays of scalar JSON values (string, number, boolean) as property values.

Each edge is a JSON object with one optional and four mandatory fields:

- `id` (optional) the edge identifier, being a non-empty string, or the value `null` equivalent to no edge identifier. Edge identifiers MUST be unique per PG-JSON document.
- `undirected` (optional) a boolean value whether the edge is undirected
- `from` an identifier of the source node from `nodes` array
- `to` an identifier of the target node from `nodes` array
- `labels` and `properties` as defined above at nodes

Example 24: Example graph in PG-JSON

```
{
  "nodes": [{
    "id": "101", "labels": [ "person" ],
    "properties": {
      "name": [ "Alice", "Carol" ],
      "country": [ "United States" ]
    }
   },{
    "id": "102", "labels": [ "person", "student" ],
    "properties": { "name": [ "Bob" ], "country": [ "Japan" ] }
  }],
  "edges": [{
    "from": "101", "to": "102", "undirected": true,
    "labels": [ "same_class", "same_school" ],
    "properties": { "since": [ 2012 ] }
   },{
    "from": "101", "to": "102",
    "labels": [ "likes" ],
    "properties": { "engaged": [ false ], "since": [ 2015 ] }
  }]
}
```

5 PG-JSONL

A **PG-JSONL** document or stream serializes a property graph in JSON Lines format, also known as newline-delimited JSON. A PG-JSONL document is a sequence of JSON objects, separated by line separator (`U+000A`) and optional whitespace (`U+0020`, `U+0009`, and `U+000D`) around JSON objects, and an optional line separator at the end. Each object is

- either a node with field `type` having the string value `"node"` and the same mandatory node fields from PG-JSON format,
- or an edge with field `type` having the string value `"edge"` and the same mandatory edge fields from PG-JSON format.

Node objects SHOULD be given before their node identifiers are referenced in an edge object, but applications MAY also create implicit node objects for these cases. Applications MAY allow multiple node objects with identical node identifier in PG-JSONL but they MUST make clear whether nodes with repeated identifiers are ignored, merged into existing nodes, or replace existing nodes.

6 References

6.1 Normative References

- Bradner, S.: *Key words for use in RFCs to Indicate Requirement Levels.* BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/info/rfc2119.
- Bray, T.: *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259, December 2017. https://tools.ietf.org/html/rfc8259
- Bray, T. et al: *Section 6 (Simple Extended Backus-Naur Form (EBNF) notation).* In: *W3C Extensible Markup Language (XML) 1.0 (Fifth Edition).* November 2008. https://www.w3.org/TR/REC-xml/#sec-notation
- Leiba, B.: *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words.* BCP 14, RFC 8174, May 2017, http://www.rfc-editor.org/info/rfc8174.
- The Unicode Consortium: *The Unicode Standard.* http://www.unicode.org/versions/latest/
- Yergeau, F.: *UTF-8, a transformation format of ISO 10646.* RFC 3629, November 2003. https://tools.ietf.org/html/rfc3629

6.2 Informative references

- Property Graph Exchange Format Homepage https://pg-format.github.io/ including PG Test Suite, PG Syntax Highlighting, PG Format discussion forum, and links to implementations.
- JSON Schema schema language
- IEEE Standard for Floating-Point Arithmetic
- Chiba, H., Yamanaka, R., Matsumoto, S.: Property Graph Exchange Format. 2019

Appendices

The following information is non-normative.

JSON Schemas

The PG-JSON format can be validated with a non-normative JSON Schema file `pg-json.json` in the specification repository. Rules not covered by the JSON schema include:

- nodes referenced in edges must be defined (no implicit nodes)
- node identifiers must be unique per graph
- edge identifiers must be unique per graph

The PG-JSONL format can be validated with a non-normative JSON Schema file `pg-jsonl.json` in the specification repository. Validation is limited in the same way as validation of PG-JSON with its JSON Schema.

Changes

This document is managed in a revision control system at https://github.com/pg-format/specification, including an issue tracker.

- **Version 1.0.0** (*not published yet*)

  Introduced comments, line folding, edge identifiers. Aligned property values with JSON syntax. Added more formal rules for quoted strings and unquoted identifiers. Added PG-JSONL. Changed node identifiers to be strings.

- **Version 0.3**

  Less formal specification first published in 2019. See latest draft from 2020.

Acknowledgements

Many thanks to Ryota Yamanaka (Meer Consulting Group) and Shota Matsumoto (Lifematics Inc.) for their contribution to the first versions of PG Format.