# g2gml Documentation

g2glab

Jan 07, 2021

# Contents

CHAPTER 1

# Overview

- There are various **graph databases** such as Neo4j, Oracle PGX, Amazon Neptune, etc.
- Different graph databases use different **property graph models and formats**
- We define the property graph (PG) data model and its serializations (**PG** and **JSON-PG**)
- We also provide **converters** for other related data formats

# CHAPTER 2

# PG formats

- A PG file consists of lines that describe nodes and edges
- Each line describes one node or one edge

`example.pg`

```
# NODES
101 :person name:Alice country:"United States"
102 :person :student name:Bob country:Japan

# EDGES
101 -- 102 :same_school :same_club since:2012
101 -> 102 :likes since:2015
```

## 2.1 Nodes

```
<node_id> :<label1> :<label2> ... <key1>:<value1> <key2>:<value2> ...
```

- All elements are separated by space or tab
- **Node IDs** have to be unique
  - If there are multiple lines with the same Node ID, latter ones are ignored
- Each line can contain arbitrarily many **labels**
- Each line can contain arbitrarily many **properties**
- Each property can have multiple values.
  - The following example has multiple values as `name` property

```
101 :person name:Alice name:Ally country:"United States"
```

## 2.2 Edges

```
<src_node_id> [->|--] <dst_node_id> :<label1> :<label2> ... <key1>:<value1> <key2>:
↪<value2> ...
```

- Basically, edge lines have the same format as node lines
- However, the first three columns contain **source node ID**, **direction**, and **destination node ID**
- An edge can be directed `->` or undirected `--`
- **The combinations of node IDs** do NOT have to be unique. (= multiple edges are allowed)
  - An edge line will be ignored if a non-defined node ID is used

## 2.3 Data type

PG format allows the following data types:

- Integer: Written as a sequence of digits
  - For example, `1`, `009` and `301`
- Double-precision floating-point number (double): Written as a sequence of digits with exact one period
  - For example, `1.0`, `2.321` and `001.002`
- String: Anything else
  - Should be double quoted if it contains a space, tab, or colon (`:`)
  - To escape double quotes, use `\"`
  - For example, `Alice`, `x2`, `"2.00"`, `"United States"` and `"\"Quoted String\""`

Each element can have one of the following data types:

- Node ID: integer or string
- Label: string
- Property key: string
- Property value: integer, double, or string

## 2.4 JSON-PG format

JSON format is useful for being processed by web clients, while the PG (flat file) format above is convenient for users and file systems.

This format basically follows the rules of the general JSON format and our PG format, however:

- **Nodes and edges** are listed under `nodes` and `edges` elements, respectively
- **Edge direction** is defined with the boolean element `undirected`. By default it is `false` (= directed)
- **Labels** are listed under the `labels` element
- **Properties** (= key-value pairs) are listed under the `properties` element

`example.json`

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

## 2.5 Comparing the formats

### 2.5.1 PG

```
# NODES
101 :person name:Alice country:"United States"
102 :person :student name:Bob country:Japan

# EDGES
101 -- 102 :same_school :same_class since:2012
101 -> 102 :likes since:2015
```

### 2.5.2 JSON-PG

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

### 2.5.3 Dot (Graphviz)

```
digraph "graph" {
  "p1" [label="person\lp1\l" name="Bob"]
  "p2" [label="person\lp2\l" name="Alice"]
  "p1" -> "p2" [label="likes\l" since="2013"]
  "p1" -> "p2" [label="friend\l" since="2011" dir=none]
}
```

### 2.5.4 CSV

- Consists of two tables (for nodes and edges, respectively).
- Property keys and their datatypes are defined in the headers or separately.
- No method to define undirected edges.

Nodes:

```
p1,person,Bob
p2,person,Alice
```

Edges:

```
p1,p2,friend,2011
p1,p2,likes,2013
```

# CHAPTER 3

# PG Tools

## 3.1 Installation

If **Docker** is installed on your machine, run the following:

```
$ alias pg2dot='docker run --rm -v $PWD:/work g2glab/pg:0.4 pg2dot'
$ pg2dot --version
0.4.0
```

Otherwise, install **Git** and **Node**, then run the following with the latest version number:

```
$ git clone -b v0.4.0 https://github.com/g2glab/pg.git
$ cd pg
$ npm install
$ npm link
$ pg2dot --version
0.4.0
```

## 3.2 Quick start

Create a sample graph:

```
$ vi graph.pg
```

`graph.pg`:

```
p1 :person name:Bob
p2 :person name:Alice
p1 -> p2 :likes since:2013
p1 -- p2 :friend since:2011
```

Run the `pg2dot` command:

```
$ pg2dot graph.pg
"graph.dot" has been created.
```

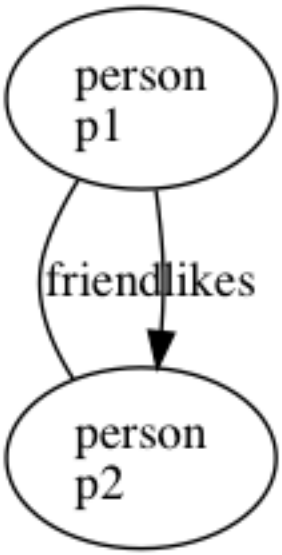
graph.dot:

```
digraph "graph" {
  "p1" [label="person\lp1\l" name="Bob"]
  "p2" [label="person\lp2\l" name="Alice"]
  "p1" -> "p2" [label="likes\l" since="2013"]
  "p1" -> "p2" [label="friend\l" since="2011" dir=none]
}
```

You can now generate a PNG image from the converted file using graphviz.

```
$ dot -T png graph.dot -o graph.png
```

data.png:




## 3.3 Commands

PG to graphviz dot:

```
$ pg2dot graph.pg [-p output_path_prefix]

e.g.
$ pg2dot graph.pg -p output/graph
"output/graph.dot" has been created.
```

PG to CSV format (for loading to graph databases):

```
$ pg2csv graph.pg [-p output_path_prefix] [-d dbms]
  dbms = neo | ora | aws

e.g.
$ pg2csv graph.pg -d neo
```

(continues on next page)

(continued from previous page)

```
"graph.neo.nodes.csv" has been created.
"graph.neo.edges.csv" has been created.
"graph.neo.cypher" has been created.
```

PG to JSON-PG:

```
$ pg2json graph.pg [-p output_path_prefix]
```

JSON-PG to PG:

```
$ json2pg graph.json [-p output_path_prefix]
```

CHAPTER 4

# Loading to databases

## 4.1 Neo4j

### 4.1.1 Convert PG to CSV

Create a sample graph.

```
$ vi graph.pg
```

`graph.pg`

```
p1 :person name:Bob
p2 :person name:Alice
p1 -> p2 :likes since:2013
p1 -- p2 :friend since:2011
```

Create Neo4j style CSV files.

```
$ alias pg2csv='docker run --rm -v $PWD:/work g2glab/pg:0.4 pg2csv'
$ pg2csv graph.pg -d neo4j
```

This command creates 3 files:

- `graph.neo.nodes.csv`
- `graph.neo.edges.csv`
- `graph.neo.cypher` - LOAD CSV Cypher command

### 4.1.2 Load with Cypher command (LOAD CSV)

Run LOAD CSV command using Neo4j Shell.

```
$ $NEO4J_DIR/bin/neo4j-shell -c < graph.neo.cypher
```

### 4.1.3 Bulk load

Remove the existing Neo4j database files.

```
$ rm -r $NEO4J_DIR/data/databases/graph.db
```

Import the graph from the CSV files.

```
$ $NEO4J_DIR/bin/neo4j-import \
  --into $NEO4J_DIR/data/databases/graph.db \
  --nodes graph.neo.nodes.csv \
  --relationships graph.neo.edges.csv
```

Start Neo4j console and access its browser ( http://localhost:7474/browser/ ).

```
$ $NEO4J_DIR/bin/neo4j console
```

### 4.1.4 Bulk load (in Docker environment)

Create Neo4j container.

```
$ docker run -d \
  -p 7474:7474 \
  -p 7687:7687 \
  --name neo4j \
  --env NEO4J_AUTH=none \
  neo4j
```

Copy Neo4j files to the container.

e.g.

```
$ docker cp graph.neo.nodes.csv neo4j:/var/lib/neo4j/import/nodes.csv
$ docker cp graph.neo.edges.csv neo4j:/var/lib/neo4j/import/edges.csv
```

Import data and reload Neo4j server.

```
docker exec neo4j bash -c \
"rm -rf data/databases/graph.db/ && neo4j-admin import \
  --database=graph.db \
  --nodes=import/nodes.csv \
  --relationships=import/edges.csv \
  --delimiter='\t' \
  && chown -R root:root /data" \
  && docker restart neo4j
```

Access to http://localhost:7474 on your web browser to see the imported data.

## 4.2 Oracle

Create sample data.

```
$ vi graph.pg
```

graph.pg

```
p1 :person name:Bob
p2 :person name:Alice
p1 -> p2 :likes  since:2013
p1 -- p2 :friend since:2011
```

Create Oracle style CSV files and config file.

```
$ alias pg2csv='docker run --rm -v $PWD:/work g2glab/pg:0.4 pg2csv'
$ pg2csv graph.pg -d ora
```

This command creates 3 files:

- graph.ora.nodes.csv
- graph.ora.edges.csv
- graph.ora.json - Loading configuration file

Run stand-alone PGX using JShell and load the data.

```
$ /opt/oracle/graph/bin/opg-jshell
pgx> G = session.readGraphWithProperties("graph.ora.json")
```

## 4.3 Amazon Neptune

Create a sample graph.

```
$ vi graph.pg
```

graph.pg

```
p1 :person name:Bob
p2 :person name:Alice
p1 -> p2 :likes since:2013
p1 -- p2 :friend since:2011
```

Create Neptune style CSV files.

```
$ alias pg2csv='docker run --rm -v $PWD:/work g2glab/pg:0.4 pg2csv'
$ pg2csv graph.pg -d aws
```

- graph.aws.nodes.csv
- graph.aws.edges.csv

Load the graph.

```
$ curl -X POST \
-H 'Content-Type: application/json' \
http://<Neptune-Endpoint>.us-west-2.neptune.amazonaws.com:8182/loader -d'
{ "source" : "s3://<S3-Bucket-Name>/",
  "format" : "csv",
  "iamRoleArn" : "arn:aws:iam::<AWS-account-ID>:role/NeptuneLoadFromS3",
```

(continues on next page)

(continued from previous page)

```
  "region" : "us-west-2",
  "failOnError" : "FALSE"
}'
```

After loading the graph into Neptune you will receive a load-ID that can be used for checking your load status.

```
curl -G \
'http://<Neptune-Endpoint>.us-west-2.neptune.amazonaws.com:8182/loader/<load-ID>'
```

Output:

```
{
"status" : "200 OK",
"payload" : {
    "feedCount" : [
        {
            "LOAD_NOT_STARTED" : 1
        },
        {
            "LOAD_FAILED" : 1
        }
    ],
    "overallStatus" : {
        "fullUri" : "s3://<S3-bucket-name>",
        "runNumber" : 1,
        "retryNumber" : 2,
        "status" : "LOAD_CANCELLED_DUE_TO_ERRORS",
        "totalTimeSpent" : 3,
        "startTime" : 1574269317,
        "totalRecords" : 10656,
        "totalDuplicates" : 0,
        "parsingErrors" : 8,
        "datatypeMismatchErrors" : 0,
        "insertErrors" : 0
    }
}
```

CHAPTER 5

# Property Graph Exchange Format

https://arxiv.org/abs/1907.03936

## 5.1 Abstract

Recently, a variety of database implementations adopting the property graph model have emerged. However, interoperable management of graph data on these implementations is challenging due to the differences in data models and formats. Here, we redefine the property graph model incorporating the differences in the existing models and propose interoperable serialization formats for property graphs. The model is independent of specific implementations and provides a basis of interoperable management of property graph data. The proposed serialization is not only general but also intuitive, thus it is useful for creating and maintaining graph data. To demonstrate the practical use of our model and serialization, we implemented converters from our serialization into existing formats, which can then be loaded into various graph databases. This work provides a basis of an interoperable platform for creating, exchanging, and utilizing property graph data.

## 5.2 Introduction

Increasing amounts of scientific and social data are described and analyzed in the form of graphs. In the context of graph analysis, the **property graph model** is becoming popular; various graph database engines, including Neo4j, Oracle Labs PGX, and Amazon Neptune, adopt this model. These graph database engines support powerful algorithms for traversing or analyzing graphs. In contrast to the standardized RDF, however, property graphs lack a standardized data model.

Here, we considered the general requirements for representing property graphs and designed two serialization formats as flat text and JSON. These formats can be converted into specific formats for each of the databases mentioned above. The serialization formats independent of certain database implementations will increase the interoperability of graph databases and will make it easier for users to import accumulated graph data.

## 5.3 Model definition

Here, we define the property graph model independent of specific graph database implementations. For the purpose of interoperability, we incorporate differences in property graph models, taking into consideration multiple labels or property values for nodes and edges, as well as mixed graphs with both of directed and undirected edges. The property graph model we redefine here requires the following characteristics:

- A property graph contains nodes and edges.
- Each of the nodes and edges can have zero or more labels.
- Each of the nodes and edges can have properties (key-value pairs).
- Each property can have multiple values.
- Each edge can be directed or undirected.

More formally, we define the property graph model as follows.

**Definition 1. Property Graph Model**

(to be updated..)

## 5.4 Serialization

According to our definition of the property graph model, we propose serialization in flat text and JSON. The flat text format (PG) is better for human readability and line-oriented processing, while the JSON format (JSON-PG) is best used for server-client communication.

The flat text PG format has the following characteristics, and an example is given in **Figure 1**.

- Each line describes a node or an edge.
- All elements in each line are separated by space or tab.
- In each of the node lines, the first column contains the node ID.
- In each of the edge lines, the first three columns contain the source node ID, the direction, and the destination node ID.
- Each line can contain an arbitrary number of labels.
- Each line can contain an arbitrary number of properties (key-value pairs).

**Figure 1. Example of PG**

```
# NODES
101 :person name:Alice age:15 country:"United States"
102 :person :student name:Bob country:Japan country:Germany

# EDGES
101 -- 102 :same_school :same_club since:2012
102 -> 101 :likes since:2015
```

More formally, we describe the PG format in the EBNF notation as follows.

**Definition 2. PG Format**

EBNF notation of the PG format.

```
PG          ::= (Node | Edge)+
Node        ::= NODE_ID Labels Properties NEWLINE
Edge        ::= NODE_ID Direction NODE_ID Labels Properties NEWLINE
Labels      ::= Label*
Properties  ::= Property*
Label       ::= ':' STRING
Property    ::= STRING ':' Value
Value       ::= STRING | NUMBER
Direction   ::= '--' | '->'
```

Next, we describe the JSON-PG format which follows the JSON syntax in addition to the above definition of the property graph model. The JSON-PG format has the following characteristics, and an example of the format is shown in **Figure 2**. It is to be noted that, whereas the set of labels or property values are represented as arrays in JSON, those elements are supposed to have no specific order according to the the property graph model.

- Nodes and edges are listed under `nodes` and `edges` elements, respectively.
- Edge direction is defined with the boolean element `undirected`. By default it is false (directed).
- Labels are listed under the `labels` element.
- Properties (key-value pairs) are listed under the `properties` element.

**Figure 1. Example of PG-JSON**

```
{
  "nodes":[
    {
     "id":101,
     "labels":["person"],
     "properties":{"name":["Alice"], "age":[15], "country":["United States"]}
    },
    {
     "id":102,
     "labels":["person", "student"],
     "properties":{"name":["Bob"], "country":["Japan", "Germany"]}
    }
  ],
  "edges":[
    {
     "from":101,
     "to":102,
     "undirected":true,
     "labels":["same_school", "same_class"],
     "properties":{"since":[2012]}
    },
    {
     "from":102,
     "to":101,
     "labels":["likes"],
     "properties":{"since":[2015]}
    }
  ]
}
```

Furthermore, we have implemented command-line tools to convert between PG and JSON-PG, as well as to transform them into formats for well-known graph databases such as Neo4j, Oracle Labs PGX, and Amazon Neptune. The practical use cases of our tools demonstrate that the proposed data model and formats have the capability to describe property graph data begin used in existing graph databases (see https://github.com/g2glab/pg).

## 5.5 Conclusion

In this work, we redefined the property graph model independent of specific graph database implementations and also proposed serialization formats based on the data model. Further, we implemented practical tools to convert our formats into existing ones. Our model and serialization will increase the interoperability of existing graph databases and make it easier for users to create, exchange, and utilize property graph data.

## 5.6 For more information

- Project Home - https://github.com/g2glab
- For Citation - https://arxiv.org/abs/1907.03936